edv aspekte

4|89
Herausgegeben
von der Redaktion
rechentechnik
datenverarbeitung
DDR 5.00 M

logisches Programmieren

Softwarewerkzeuge für den P8000 compact

Inhalt

Dr. Christian Horn, Mirko Dziadzka, Matthias Horn:
Sprachbeschreibung HU-PROLOG 2

Tom Bihr:
Programmieren mit PROLOG – einige Beispiele 27

Dr. Bodo Hohberg, Olga Wikarski:
Portierung von PASCAL-Software mit PCC 32

Dr. Wilfried Grafik, Heinz Werner:
Modula 2 für den P 8000 39

Falk Nisius, Kai-Uwe Scherer:
Modula-2-Interpreter für den P 8000 44

Jens-Peter Redlich:
Ein moderner Editor für UNIX-kompatible Betriebssysteme 47

Reinhard Hartung:
SPIDER – Ein System zur Datenübertragung 53

Dr. Jan-Peter Bell:
Erfahrungen bei der Vernetzung von P 8000 unter WEGA mittels UUCP 59

Jahresinhaltsverzeichnis 1988/89 64

Д-р Кристиан Хорн, Мирко Дзядзка, Маттиас Хорн:
Описание языка программирования HU-PROLOG 2

Том Бир:
Программирование на языке PROLOG – некоторые примеры 27

Д-р Бодо Хоберг, Ольга Викарски:
Перевод программного обеспечения на языке PASCAL с помощью PCC 32

Д-р Вильфрид График, Хайнц Вернер:
«Модула 2» для P 8000 39

Фальк Низиус, Кай-Уве Шерер:
Интерпретатор языка «Модула-2» для P 8000 44

Йенс-Петер Редлих:
Современный редактор для операционных систем, совместимых с UNIX 47

Райнхард Хартунг:
SPIDER – система передачи данных 53

Д-р Ян-Петен Белль:
Опыт применения пакета программ UUCP для создания вычислительных сетей ЭВМ типа P 8000 с ОС WEGA 59

Перечень статей, помещенных 6 журнале в 1988/89 г. 64

In this Issue

Dr. Christian Horn, Mirko Dziadzka, Matthias Horn:
HU-PROLOG speech description 2

Tom Bihr:
Programming with PROLOG-several examples 27

Dr. Bodo Hohberg, Olga Wikarski:
The portability of PASCAL software with PCC 32

Dr. Wilfried Grafik, Heinz Werner:
Modula 2 designed for use with the P 8000 39

Falk Nisius, Kai-Uwe Scherer:
Modula 2 interpreter for use with the P 8000 44

Jens-Peter Redlich:
An advanced editor for UNIX-compatible operating systems 47

Reinhard Hartung:
SPIDER – A data transmission system 53

Dr. Jan-Peter Bell:
Experience gained from networking P 8000 under WEGA using UUCP 59

Index for 1988/89 64

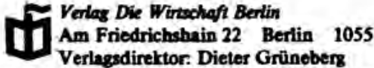
An unsere Autoren

Zur Rationalisierung der redaktionellen Bearbeitung Ihrer Manuskripte bitten wir Sie (sofern möglich), der Redaktion Ihre Beiträge zukünftig auf Diskette zu senden.
Dazu sind folgende Bedingungen Voraussetzung:
1. Die Autorendisketten müssen am BAP 3001 lesbar sein
2. Erfassung des Textes mittels Textprogramm (Robotron)
3. Verwendung des SCP-Formates
624 K – 80 Spuren, doppelseitig, 16 × 256, 3 Systemspuren
Folgende SCP-Formate sind ebenfalls möglich, erfordern jedoch von der Redaktion eine Umformatierung der Disketten:
– 800 K – 80 Spuren, doppelseitig, 5 × 1 024, Spuren 0–159
– 780 K – 80 Spuren, doppelseitig, 5 × 1 024, Spuren 3–159
– 148 K – 40 Spuren, einseitig, 16 × 256
Im DCP findet folgendes Format Verwendung:
– 360 K – 40 Spuren, doppelseitig, 9 × 512, ohne Systemspuren
4. Alle Tabellen und Abbildungen zum Beitrag müssen als Sondermanuskript (Anhang) aufbereitet sein.
5. Der Beitrag sollte neben der Diskette auch als Manuskript an uns geschickt werden
6. Für die an uns gesendete(n) Diskette(n) benötigen wir folgende Informationen:
– Disketteninhalt (Dateiname(n), Formatangabe)
7. Die Texte der Beiträge sind endlos zu erfassen.
Bei Hervorhebungen im Text sind folgende Schriftarten zugelassen:
– kursiv, halbfett.
Für umfangreiche Beiträge sollten mehrere Dateien erstellt werden (etwa 10 Manuskriptseiten = eine Datei).
Nach Übernahme Ihrer Textdaten auf unseren Computer erhalten Sie Ihren Datenträger umgehend zurück.

Ihre Redaktion rd

edv
aspekte

8. Jahrgang 4/1989


Verlag Die Wirtschaft Berlin
Am Friedrichshain 22 Berlin 1055
Verlagsdirektor: Dieter Grüneberg

edv-aspekte
Zeitschrift für spezielle Themen
der Informationsverarbeitung,
herausgegeben von der Redaktion
rechentechnik/datenverarbeitung,
1055 Berlin, Am Friedrichshain 22
Chefredakteur: Franz Loll 4 38 73 41
Redakteur: Claudia Schulz 4 38 73 16
Sekretariat: 4 38 72 33
Fernschreiber: 114 566
Titelgestaltung: Marlies Hawemann

Redaktionsschluß: 30. 9. 1989

Lizenz des Presseamtes beim Vorsitzenden
des Ministerrates der DDR Nr. 1529

edv-aspekte
Erscheinungsweise vierteljährlich zum Bezugspreis DDR 5,00 M je Heft
EDV-Artikel-Nr. 1331
Auslandspreise sind dem Zeitschriftenkatalog des Außenhandelsbetriebes
Buchexport zu entnehmen.

Satz: Verlag Die Wirtschaft, Berlin
Druck: (140) „Neues Deutschland", Berlin

Anzeigenannahme und -verwaltung:
Berliner Verlag, Karl-Liebknecht-Str. 29,
DDR Berlin, 1026
Telefon: 2 70 33 02

Im Ausland:
INTERWERBUNG GmbH – Gesellschaft
für Werbung und Auslandsmessen der DDR,
Hermann-Duncker-Str. 89 Berlin 1157

*Bestellungen aus der DDR sind an den
Postzeitungsvertrieb zu richten.*
Inkasso-Zeitraum: vierteljährlich

Im Ausland:
In den sozialistischen Ländern nur der zuständige Postzeitungsvertrieb. In allen anderen
Staaten der örtliche Buch- und Zeitschriftenhandel. Bestellungen des Buch- und Zeitschriftenhandels sind zu richten an

BUCHEXPORT
Volkseigener Außenhandelsbetrieb der DDR,
DDR – Leninstr. 16, Leipzig 7010,
Postfach 160
oder an Verlag Die Wirtschaft,
DDR – Am Friedrichshain 22, Berlin 1055

Mitglieder des Redaktionsbeirates
Dr. Claus Goedecke · Dr. Rolf Gräßler
Prof. Dr. sc. Gerhard Keßler · Dr. Rolf Kilian
Hans Kunau · Walter Münch · Axel Rathsack
Prof. Dr. sc. Gerd Rossa ·
Prof. Dr. sc. Claus Sattler ·
Prof. Dr. sc. Wolfgang Schoppan (Vorsitzender)
Dr. Werner Schulze · Horst Stoll
Prof. Dr. Franz Stuchlik · Dr. Dieter Urban

Logisches Programmieren – Softwarewerkzeuge für den P 8000

Als „Expertensystem" bzw. „wissensbasiertes System" werden gegenwärtig weltweit Programmsysteme und Untersuchungen bezeichnet, die es dem Anwender gestatten, in einem jeweils abgegrenzten Gegenstandsbereich Situationen zu analysieren und Entscheidungen vorzubereiten. Wissensbasierte Systeme sollten ihre Schlüsse „erklären" können. Oft werden sie so organisiert, daß sie neues Wissen, d. h. neue Fakten und Beziehungen zwischen Fakten, adaptieren können.

Im Zusammenhang mit wissensbasierten Systemen kommen deshalb der Wissensdarstellung und den Algorithmen, mit deren Hilfe aus der Wissensbasis Schlußfolgerungen abgeleitet werden können – also einem Inferenzmechanismus – besondere Bedeutung zu.

Für das Problemlösen mit Hilfe wissensbasierter Systeme sind neben hardwaremäßigen Voraussetzungen softwareseitig Mittel zum Entwurf und zum Aufbau wissensverarbeitender Programme bereitzustellen. Dafür hat sich die Bezeichnung „Werkzeuge" eingebürgert, als wären diese Hilfsmittel physikalisch benutzbar wie Hammer und Säge. Das ist sicherlich eine treffende Begriffserweiterung.

Das Nachdenken über Lösungen für die Faktendarstellung und für Inferenzmechanismen erfolgte in zwei Richtungen, nämlich

1. Inwieweit ist es mit den „klassischen" prozeduralen Sprachen möglich, das Problemlösen als ein Ableiten aus einer Datensammlung zu codieren?
2. Welche Eigenschaften sollten Programmiersprachen haben, die unter dem Gesichtspunkt einer solchen Informationsverarbeitungstechnologie entworfen werden?

Für die Praxis der Entwicklungs- und Forschungsarbeit heißt das konkret, einerseits die bekannten prozeduralen Sprachen auf neue Hardwarearchitekturen zu übernehmen, ihre Leistungsfähigkeit zu untersuchen und eventuell geeignete Spracherweiterungen vorzunehmen. Andererseits geht es darum, deskriptive Sprachen zu entwickeln und zum Einsatz zu bringen.

Es zeichnet sich ab, daß für Expertensysteme und Probleme der künstlichen Intelligenz Sprachen wie PROLOG und LISP favorisiert sind, daß aber sowohl für spezielle Anwendungen als auch für die Zusammenarbeit mit deskriptiv formulierten Systemen moderne prozedurale Sprachen wie PASCAL, MODULA-2, C u. a. eine nicht zu übersehende Bedeutung erhalten. Die in dem vorliegenden Heft vorgestellten Arbeiten bringen aktuelle Konzeptionen und Realisierungen von Softwarewerkzeugen, die im Rahmen von Untersuchungen zu moderner Softwaretechnologie im oben beschriebenen Sinne entstanden sind. Das Schwergewicht haben wir auf ein PROLOG-System gelegt, das in den letzten Jahren entwickelt und auf unterschiedlichen Rechnertypen erprobt wurde, und das ein leistungsfähiges Werkzeug für zeitgemäße Informationsverarbeitung ist.

Nichtsdestoweniger werden auch die Arbeiten zu anderen, aktuellen Programmiersprachen und verbreiteten Dienstprogrammen das Interesse kreativ arbeitender Informatiker finden.

Prof. Dr. sc. Hans Schmiemangk, Dr. Gerhard Paulin

Sprachbeschreibung HU – PROLOG

Dr. Christian Horn, Mirko Dziadzka, Matthias Horn
Humboldt-Universität zu Berlin

	Seite
1. Einleitung	2
1.1. Ein Beispiel	3
1.2. Zeichenvorrat	5
1.3. Terme	5
1.4. Programme	7
2. Ein- und Ausgabe	7
2.1. Fileoperationen	7
2.2. Eingabeoperationen	8
2.3. Ausgabeoperationen	9
3. Termbehandlung	10
3.1. Termklassifikation	10
3.2. Termanalyse und -synthese	10
3.3. Analyse und Synthese von Atomen	11
3.4. Termvergleich	12
4. Manipulation der Datenbasis	12
4.1. Einfügen von Klauseln	13
4.2. Entfernen von Klauseln	13
4.3. Abfragen von Klauseln	14
4.4. Operatordeklarationen	14
5. Arithmetik und funktionale Programmierung	15
5.1. Einleitung	15
5.2. Standardfunktionen und -operationen	16
5.3. is/2	16
5.4. Globale Variablen und Wertzuweisung	16
5.5. Auswertung von Funktionen und Variablen	17
5.6. :=/2	17
5.7. Arithmetische Vergleiche (,= ,=:=, =, ,= =)	17
6. Programmierumgebung	18
6.1. Ablaufsteuerung	18
6.2. Globale Steuerung	18
6.3. Das Modul-Konzept von HU-PROLOG	20
6.4. Die Schnittstelle zur Systemumgebung	21
6.5. Flags	22
6.6. Testunterstützung	22
6.7. Entwicklungsumgebung von HU-PROLOG	23
6.8. stats/0	23
6.9. Systemstart	23
7. Beispiele	24
7.1. Fibonacci-Zahlen	24
7.2. Taylorreihenentwicklung	24
7.3. Polynome und Horner-Schema	24
7.4. Schwachbesetzte Matrizen	24
7.5. Repetitionskommando	25
7.6. Dienstprogramm copy	25
7.7. Dienstprogramm more	25

1. Einleitung

PROLOG ist eine der faszinierenden Neuschöpfungen der Programmierkunst, eine Programmiersprache, die in schier unerschöpflicher Weise immer wieder verblüffend einfache und klare Lösungsvarianten eröffnet. Obwohl zur selben Zeit wie PASCAL entstanden, erscheint sie für den Praktiker heute jedoch immer noch unnahbar. Eine Ursache dafür liegt sicher darin, daß sie gleich mehrere qualitativ neue Elemente in die Programmierung einbringt: symbolische Manipulation, Backtracking, Rekursion und relationale Programmierung.
Erst Anfang der achtziger Jahre begann international eine verstärkte Hinwendung zu PROLOG, ausgelöst durch den Anwendungsdruck nach „intelligenten" Problemlösungen, die sich mit klassischen Sprachen nur schwer realisieren ließen. Die aufsehenerregenden Ankündigungen der japanischen fünften Rechnergeneration taten ein Übriges: Nach diesen Plänen soll PROLOG bzw. eine geeignete Erweiterung als Kernsprache der neuen Rechnergeneration fungieren. PROLOG bekommt damit eine Rolle zugewiesen, die heute vielleicht im Hinblick auf ihre Universalität und Maschinennähe nur der Sprache C in UNIX-Systemen zukommt.
An der Sektion Mathematik der Humboldt-Universität beschäftigt sich seit mehreren Jahren eine kleine Gruppe von Wissenschaftlern mit Fragen der Anwendung von PROLOG. Dabei zeigte sich bald ein Problem: Die einfachen, auf sehr hohem Abstraktionsniveau stehenden sprachlichen Mittel von PROLOG reduzieren zwar deutlich den Programmieraufwand, verlagern aber die letztendlich immer notwendigen Implementationsentscheidungen auf die Ebene der Sprachimplementierung. Jede Implementierung von PROLOG wird daher einen gewissen Programmierstil bedingen. Die abstrakte, von rein logischen Gesichtspunkten ausgehende Propagierung eines PROLOG-Stils führt genau wie jedes Außerachtlassen der Eigenschaften eines PROLOG-Systems zu ineffizienten, unnötige Ressourcen beanspruchenden Programmen. Die Folge davon sind ernste Realisierungs- und Portierungsprobleme für Anwenderlösungen auf PROLOG-Quelltextniveau insbesondere dann, wenn die Problemlösung die gegebenen technischen Möglichkeiten voll ausreizen muß.

Vor anderthalb Jahren begannen daher die Arbeiten an einer portablen PROLOG-Implementierung. Ziel war die Bereitstellung eines bis auf konfigurierbare Leistungsparameter identischen PROLOG-Systems für ein breites Spektrum von 16- und 32-bit-Rechnern, das den von Clocksin und Mellish beschriebenen Sprachumfang vollständig überdeckt und darüber hinaus offen für Spracherweiterungen ist. Am Anfang stand dabei die Untersuchung verschiedener experimenteller PROLOG-Systeme. Die Implementierung erfolgte schließlich in C und konnte sich auf Erfahrungen in der Gestaltung portabler Systemlösungen stützen. Dadurch ist das HU-PROLOG-System heute nicht nur unter UNIX sondern auch unter MS-DOS und VMS sowohl auf IBM-PC-kompatiblen (von 8086/8088 bis 80386) Systemen, auf P 8000 als auch auf VAX verfügbar. Bezüglich der Spracherweiterungen verfolgten wir ursprünglich die Strategie, dem Systemanwender Werkzeuge zur Verfügung zu stellen, mit denen er das PROLOG-System um die für seine Anwendung relevanten Funktionen (wie z. B. 2D/3D-Grafik-Funktionen, Simulationsverfahren oder Datenbankschnittstellen) erweitern kann. Diese Vorgehensweise halten wir nach wie vor für relevant, doch erfordert sie beim Anwender ein tiefes Eindringen in die internen Strukturen des PROLOG-Systems, was letztlich nur für wenige in Frage kommt. Wir haben uns daher entschlossen, das HU-PROLOG-System standardmäßig um die Funktionen zu erweitern, die nach unseren Erfahrungen für die praktische PROLOG-Programmierung besonders zweckmäßig sind, von international gebräuchlichen PROLOG-Systemen zur Verfügung gestellt werden und/oder wesentlichen Entwicklungslinien bezüglich der Spracherweiterungen entsprechen. Dazu gehören neben einigen ergänzenden built-in-Prädikaten für die Ein- und Ausgabe und einem elementaren Modulkonzept in erster Linie das Konzept der globalen Zustandsvariablen sowie der verallgemeinerten funktionalen Auswertung von Termen. Grundsätzlich sind wir dabei so vorgegangen, daß die zusätzlichen Prädikate entweder im Sinne der logischen Geschlossenheit des Systems zwangsweise notwendig waren, oder daß sich diese Prädikate in Standard-PROLOG explizit, wenn auch weniger effizient, definieren lassen.

Der vorliegende Beitrag gibt eine zusammenfassende Darstellung von HU-PROLOG (ab Version 1.53, April 1989) im Sinne einer Sprachbeschreibung. Es ist keine Einführung in die PROLOG-Programmierung. Der interessierte Leser sei dafür auf die Vielzahl inzwischen auch in deutscher Sprache vorliegender Lehrtexte verwiesen. Der Beitrag soll dem zukünftigen, praktizierenden PROLOG-Programmierer eine Hilfe sein. Wir haben uns daher ganz bewußt auf die Probleme konzentriert, die etwa 90 Prozent aller PROLOG-Programmtexte ausmachen, aber kaum in Lehrbüchern behandelt werden: das sind Fragen der klassischen, prozeduralen und funktionalen Programmorganisation. Die Glanzseiten von PROLOG spielen kaum eine Rolle, darüber ist an dieser Stelle schon genug geschrieben worden. Wer sich überhaupt daran macht, PROLOG zu benutzen, muß schon von den Vorteilen der Sprache überzeugt sein.

1.1. Ein Beispiel

Vom Betriebssystem aus ruft man PROLOG im einfachsten Fall ohne jeden Parameter auf. Das System meldet sich darauf mit seiner Identifikationsmeldung und wartet nach Ausgabe des PROLOG-Prompt-Zeichens "?-" auf Nutzeraktionen:

```
% prolog (ET)
HU-Prolog Interpreter Release ...
Ready
?-
```

Wir wollen im folgenden die Zeichenfolge (ET) benutzen, um das Drücken der Enter-Taste zu symbolisieren. Das PROLOG-Prompt-Zeichen "?-" deutet an, daß die nachfolgenden Eingaben des Nutzers direkt interpretierend abgearbeitet werden. Das Prompt-Zeichen braucht und darf nicht noch einmal eingegeben werden. Nutzereingaben werden durch einen Punkt und anschließendes Drücken der Enter-Taste abgeschlossen. Bei regulärer Beendigung der vom Nutzer eingegebenen Kommandos antwortet der PROLOG-Interpreter mit "yes" bzw. "no", um das Finden bzw. Nichtfinden einer Lösung anzuzeigen.

```
?- write(hallo). (ET)
hallo
yes
?-
```

Mit der erneuten Ausgabe des PROLOG-Prompts zeigt der Interpreter an, daß er auf den nächsten Befehl wartet. Das PROLOG-System bleibt ständig in dieser Interpreterschleife. Das Verlassen des PROLOG-Interpreters ist nur durch einen speziellen Befehl (z. B. "end", "halt" oder "exit(...)" möglich.

```
?- end. (ET)
[Leaving Prolog]
%
```

Ein PROLOG-Kommando, das vom Top-Level-Interpreter verarbeitet wird, kann aus mehreren, durch Komma getrennten Aufrufen bestehen, die nacheinander abgearbeitet werden. Das Kommando kann sich über mehrere Zeilen erstrekken, dabei darf aber nur die letzte Zeile mit Punkt und Enter abgeschlossen werden. Wenn das System aus irgendwelchen unersichtlichen Gründen mit der Abarbeitung eines Kommandos nicht beginnt, so ist eine der häufigsten Ursachen der vergessene Punkt am Ende der Eingabe. Das System nimmt in solcher Situation stets an, daß die Eingabe in der nächsten Zeile weitergeht. Das Einfügen von Leerzeichen oder Zeilenwechseln zur übersichtlicheren Gestaltung eines PROLOG-Textes hat keinen Einfluß auf dessen Abarbeitung.

Enthält ein PROLOG-Kommando Variablen, so wird nach erfolgreicher Abarbeitung des Kommandos die Lösung des gestellten Problems in Form der Variablenbelegung ausgegeben. Das System wartet dann auf eine Nutzereingabe. Einfaches

Drücken der Enter-Taste beendet den Abarbeitungsprozeß, das PROLOG-System hat eine Lösung gefunden und antwortet mit "yes".

```
?- X is 2*(3 + 4). ⟨ET⟩
X = 14      ⟨ET⟩
yes
?-
```

Eine wesentliche Eigenschaft von PROLOG ist jedoch die Fähigkeit zur Generierung aller Lösungen eines Problems durch schrittweises Backtracking. Nach der Ausgabe einer Lösung hat man daher alternativ die Möglichkeit, durch Eingabe eines Semikolons und anschließendes Drücken der Enter-Taste die Suche nach einer weiteren Lösung zu erzwingen. Gelingt dies, so wird die entsprechende Variablenbelegung ausgegeben. Findet PROLOG jedoch keine weiteren Lösungen, so antwortet es mit "no".

```
?- X is 2*3 + 4. ⟨ET⟩
X = 10   ;⟨ET⟩
no
?-
```

In PROLOG wurde das Prozedurkonzept entsprechend verallgemeinert: eine Prozedur ist erfolgreich (und liefert dabei eine oder mehrere Lösungen) oder schlägt fehl. Beim erfolgreichen Verlassen einer Prozedur wird zunächst nur eine Lösung in Form von Variablenbindungen an die Umgebung weitergegeben. Das Fehlschlagen einer der nachfolgend aufgerufenen Prozeduren initiiert die Suche nach weiteren Lösungen. Dazu wird eine Rückverfolgung der Berechnungsspur (das sogenannte Backtracking) angestoßen. Der gesamte Berechnungsprozeß wird auf den letzten Verzweigungspunkt zurückgesetzt, bei dem (noch) alternative Lösungen generiert werden können. Die Berechnung wird an diesem Punkt mit einer neuen Teillösung wieder aufgenommen und zu Ende geführt.

Wir wollen nun versuchen, eine Wertetabelle für das logische Und zu erstellen. Ein Teilproblem ist dabei das Erzeugen der Wahrheitswertebelegungen für aussagenlogische Variablen. Hier benötigt man eine einstellige Prozedur boole/1, die bei ihrem Aufruf nacheinander zwei Lösungen liefert, 0 und 1. Eine solche Prozedur wird in PROLOG am einfachsten durch zwei Fakten beschrieben:

```
boole(0).
boole(1).
```

Diese beiden Fakten bilden zusammen das PROLOG-Wissen über die Prozedur boole/1. In klassischer Terminologie würde man sagen, deren Quelltext. Die Eingabe von PROLOG-Quelltext erfolgt durch das Konsultieren eines Quellfiles, das zuvor mit einem Editor erstellt wurde, oder durch interaktive Quelltexteingabe, wo gewissermaßen vom Terminal als Quellfile gelesen wird. Auf einem Quellfile werden Klauseln grundsätzlich mit Punkt und Leerzeichen oder Punkt und Zeilenende abgeschlossen. Bei interaktiver Eingabe wiederum durch Punkt und Enter-Taste. Die interaktive Eingabe wird durch "end." oder Fileende abgeschlossen. Wir wollen diesen zweiten Weg gehen:

```
?- [user]. ⟨ET⟩
user⟩ boole(0). ⟨ET⟩
user⟩ boole(1). ⟨ET⟩
user⟩ end. ⟨ET⟩
yes
?-
```

Die Wahrheitswertebelegungen für X, Y bzw. X & Y erhalten wir nun nacheinander als Lösungen vor:

```
?- boole(X),boole(Y),Z is X & Y. ⟨ET⟩
X= 0
Y= 0
Z= 0   ;⟨ET⟩
X= 0
Y= 1
Z= 0   ;⟨ET⟩
X= 1
Y= 0
Z= 0   ;⟨ET⟩
X= 1
Y= 1
Z= 1   ;⟨ET⟩
no
?-
```

Zur Erzeugung einer Wertetabelle schreibt man nun zweckmäßig jeweils einen Term der Form "X & Y = Z" pro Zeile aus und löst durch ein anschließendes "fail" automatisch das Backtracking aus. Die Rückverfolgung führt auf neue Lösungen für X und Y und nachfolgend zu den entsprechenden Ausgaben, bis keine weiteren Lösungen für X und Y mehr gefunden werden können. Daher schlägt der Gesamtaufruf am Ende fehl.

```
?- boole(X), boole(Y), ⟨ET⟩
Z is X & Y, write(X & Y = Z), nl, fail. ⟨ET⟩
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
no
?-
```

Für dieses PROLOG-Kommando kann man der Abkürzung halber eine parameterlose Prozedur zeile/0 einführen. Diese besteht aus genau einer Regel, die die Abarbeitungsfolge innerhalb der Prozedur festlegt. Die Eingabe erfolgt wiederum im interaktiven Modus.

```
?- [user]. ⟨ET⟩
user⟩ zeile:- ⟨ET⟩
        boole(X), boole(Y), ⟨ET⟩
        Z is X & Y, ⟨ET⟩
        write(X & Y = Z), nl. ⟨ET⟩
user⟩ end. ⟨ET⟩
?-
```

Der Aufruf von "zeile" bewirkt die Ausgabe genau einer Zeile der Wertetabelle. Durch ein anschließendes "fail" läßt sich das Zurücksetzen der Abarbeitung erzwingen, die innerhalb der Prozedur "zeile" einen Aufsetzpunkt findet, indem neue Belegungen für X und Y bestimmt werden können. Die Variablen X, Y und Z gehen zwar nicht als Lösung nach außen, ihre Belegung beschreibt aber den inneren Zustand der Prozedur "zeile", der für die Fortführung der Berechnung nach dem Backtracking noch benötigt wird. Aus diesem Grunde bleiben im Unterschied zu klassischen Programmiersprachen in PROLOG die Variablen einer Prozedur bei deren erfolgreichen Verlassen im allgemeinen erhalten. Zur Erzeugung der Wertetabelle genügt nun ein PROLOG-Kommando der Form:

```
?- zeile, fail. ⟨ET⟩
0&0=0
0&1=0
1&0=0
1&1=1
no
?-
```

Will man die vollständig Erzeugung der Tabelle in einer Prozedur zusammenfassen, die am Ende erfolgreich ist, so benötigt man dazu zwei Klauseln:

```
?– [user]. ⟨ET⟩
user⟩ tabelle:–zeile,fail. ⟨ET⟩
user⟩ tabelle. ⟨ET⟩
user⟩ end. ⟨ET⟩
yes
?–
```

Die erste Klausel (von der syntaktischen Gestalt her eine Regel) ruft die Prozedur zeile/0 sooft es geht auf. Wenn keine weiteren Zeilen mehr erzeugt werden können, schlägt die erste Klausel fehl und die zweite Klausel wird aktiviert. Hier könnte irgendeine Form der Abschlußbehandlung erfolgen. Im vorliegenden Fall reicht aber ein Fakt aus, der eine fiktive Lösung liefert und einzig die Funktion hat, der Prozedur tabelle/0 einen positiven Ausgang zu geben. Damit kann die Prozedur tabelle/0 nun beliebig in eine Aufruffolge eingeordnet werden, ohne die Abarbeitungsfolge durch (u. U. ungewolltes) Backtracking zu stören. Zur Erzeugung der Tabelle reicht nun ein Aufruf aus:

```
?– tabelle. ⟨ET⟩
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
yes
?–
```

1.2. Zeichenvorrat

PROLOG nutzt den vollen ASCII-Zeichensatz, wobei nur die druckbaren Zeichen mit einer Bedeutung unterlegt sind.

```
Zeichen = Buchstabe | Ziffer |
          Sonderzeichen | Einzelzeichen |
          Listenseparator | Klammer |
          Atombegrenzer | Stringbegrenzer |
          Kommentarzeichen | Leerzeichen .

Buchstabe =          Großbuchstabe | Kleinbuchstabe .
Großbuchstabe =      "–" | "A" | "B" | ... | "Z" .
Kleinbuchstabe =     "$" | "a" | "b" | ... | "z" .

Ziffer =             "0" | "1" | "2" | ... | "9" .

Sonderzeichen =      "+" | "–" | "*" | "/" | "\" | "^" |
                     "<" | ">" | "=" | "," | "–" | ":" |
                     "." | "?" | "@" | "#" | "!" | "&" .

Einzelzeichen =      "," | ";" .
Listenseparator =    "|" .
Klammer =            "(" | ")" | "[" | "]" | "{" | "}" .

Atombegrenzer =      " ' " .
Stringbegrenzer =    " " " .
Kommentarzeichen =   "%" .
Leerzeichen =        " " .

darstellbares_Zeichen = Zeichen | spezielles_Zeichen .

spezielles Zeichen = "\\" | "\'" | "\"" | "\n" | "\a" |
                     "\r" | "\b" | "\t" | "\f" | "\v" .
```

Lexikalische Einheiten von HU-PROLOG sind Atome, Zahlen, Variablen, Strings, Einzelzeichen, Klammern und der Listenseparator. Zwischen zwei lexikalischen Einheiten können beliebig viele, wenn es die Eindeutigkeit der Interpretation erfordert (so zwischen zwei Atomen, Variablen, Zahlen oder Strings), jedoch mindestens ein Layoutzeichen eingefügt werden. Layout-Zeichen sind Leerzeichen, Tabulatoren, Zeilenendezeichen und Kommentare. PROLOG unterstützt dabei zwei Formen von Kommentaren: Zeilenendkommentare, die von einem Kommentarzeichen (%) bis zum Ende der jeweiligen Zeile reichen, und geklammerte Kommentare, die von "/*" bis zum nächsten "*/" reichen.

1.3. Terme

```
Term = einfacher_Term | strukturierter_Term .

einfacher Term = Atom | Zahl | Variable .

strukturierter_Term = Funktorterm |
                      Operatorterm |
                      Curlyterm |
                      Listenterm .
```

PROLOG benutzt ein einheitliches Format zur Speicherung von Daten und Programmen, die sogenannten Terme. Die elementaren Terme sind Zahlen, Atome und Variablen. Strukturierte Terme besitzen unabhängig von ihrem äußerem Erscheinungsbild eine homogene interne Struktur. Sie bestehen aus einem (Haupt-)Funktor und einer Folge von Argumenten, wobei die Anzahl der Argumente genau der Stelligkeit des Funktors entspricht. Die Argumente ihrerseits sind wiederum Terme. Diese interne Struktur wird durch die syntaktische Gestalt der Funktorterme widergespiegelt.

1.3.1. Atome

Atom = einfaches_Atom | gequotetes_Atom .

einfaches_Atom = Bezeichner |
Spezialatom |
Einzelzeichen .

Bezeichner = Kleinbuchstabe { Buchstabe | Ziffer } .
Spezialatom = Sonderzeichen { Sonderzeichen } .

gequotetes_Atom = "'" { darstellbares_Zeichen } "'" .

Atome sind die elementaren symbolischen Dateneinheiten, die zwar mit speziellen built-in-Prädikaten erzeugt und analysiert werden können, aber für den normalen PROLOG-Berechnungsprozeß unteilbare Einheiten darstellen. Innerhalb gequoteter Atome und innerhalb von Strings können spezielle Zeichen mit den üblichen, von C her bekannten Backslash-Konventionen dargestellt werden.

1.3.2. Zahlen

Integerzahl = Ziffernfolge .
Ziffernfolge = Ziffer { Ziffer } .

Realzahl = Ziffernfolge "." Ziffernfolge [Exponent] |
Ziffernfolge Exponent .
Exponent =("E" | "e") ["+" | "–"] Ziffernfolge .

Zahlen sind die elementaren numerischen Daten von PROLOG. HU-PROLOG unterscheidet syntaktisch zwischen Integerzahlen und Realzahlen. Integerzahlen sind ganze Zahlen zwischen –2147483648 und 2147483647 (entsprechend dem Zahlentyp long des zugrundeliegenden C-Dialekts). Realzahlen liegen zwischen –0.13e309 und 0.13e309 (entsprechend dem Zahlentyp double des zugrundeliegenden C-Dialekts). Integerzahlen zwischen –32768 und 32767 (entsprechend dem Typ int des zugrundeliegenden C-Dialekts) werden direkt abgespeichert. Arithmetische Operationen über diesem Zahlenbereich sind damit auch die schnellsten und überaus speichereffektiv. Große Integer- sowie Realzahlen erfordern intern eine Sonderbehandlung. Arithmetische Operationen über diesen Zahlenbereichen sind deshalb langsamer und speicheraufwendiger.

1.3.3. Variablen

Variable = (Großbuchstabe) { Buchstabe | Ziffer } .

PROLOG ist eine typfreie Sprache. Alle Daten haben die Struktur von Termen, die jedoch unterschiedlich groß ausfallen können. Die Variablen nehmen in PROLOG daher nur Verweise auf Terme auf. Variablenbindungen entstehen bei der Unifikation. Das ist der elementare Prozeß der Angleichung zweier Term(muster) durch Belegung der in diesen Termen auftretenden freien Variablen. Die Unifikation wird in PROLOG als bidirektional wirkender Parameterübergabemechanismus genutzt. Die Bindung einer Variablen an einen Term erfolgt jeweils nur einmal. Eine neue Variablenbindung kann nur vorgenommen werden, wenn die alte zuvor im Backtracking wieder aufgehoben wurde.

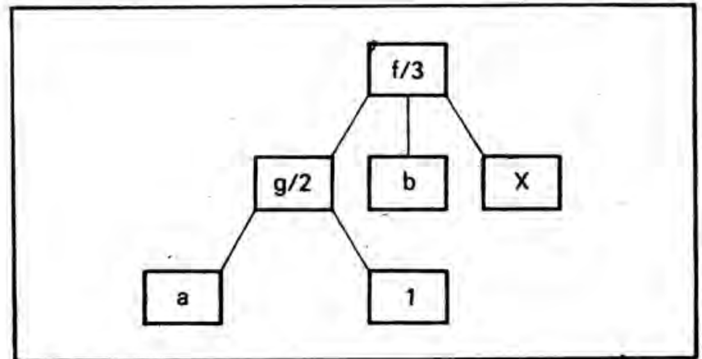


Abb. 1 Interne Struktur des Funktorterms f(g(a,1),b,X)

1.3.4. Funktorterme

Funktorterm = Funktor "(" Subterm { "," Subterm } ")" .
Funktor = Atom .
Subterm = Term .

Für Funktorterme ist entscheidend, daß zwischen dem Funktor und der öffnenden Klammer kein Layoutzeichen steht. Anderenfalls wäre eine eindeutige Unterscheidung von Termen in Präfixnotation nicht gegeben. Die Kommata zwischen den Klammern eines Funktorterms trennen die Argumente, es sind keine Operatoren. Um die syntaktische Eindeutigkeit zu wahren, dürfen die Operatoren in den Argumenten nur einen Vorrang kleiner als das Komma besitzen. Operatorterme höherer Priorität müssen explizit geklammert werden (Abb. 1).

1.3.5. Operatorterme

Operatorterm = Operator Term |
Term Operator Term |
Term Operator |
"(" Term ")" .

Operatorterme erlauben die vereinfachte Notation ein- und zweistelliger Funktoren in Präfix-, Infix- bzw. Postfix-Schreibweise. Die Syntaxanalyse und die Systemausgaberoutine werden zur Laufzeit durch die Tabelle der aktuell gültigen Operatorvereinbarung gesteuert, die sowohl die Priorität als auch die Assoziativität der Operatoren festlegt (vgl. 4.4). Die aus Gründen der syntaktischen Eindeutigkeit notwendige explizite Klammerung von Termen wird nicht mit abgespeichert, sondern vielmehr bei der Ausgabe wieder automatisch generiert (Abb. 2).

1.3.6. Curlyterme

Curlyterm = "{" "}" |
"{" Term "}" .

Curlyterme erlauben eine spezielle syntaktische Notation wie z. B. für Mengen: {}, {a}, {a,b}, {a,b,c} usw. Der leere Curlyterm wird durch ein Atom {}/0 repräsentiert. Nichtleere Curlyterme werden durch den Funktor {}/1 und einen Argumentterm, der dem in geschweifte Klammern eingeschlossenen Term entspricht, dargestellt. Die Kommata innerhalb eines

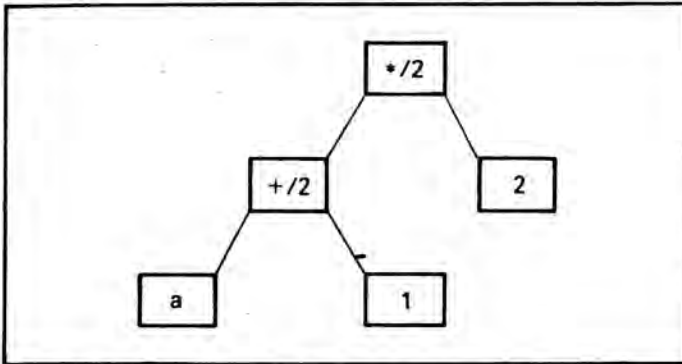


Abb. 2 Interne Struktur des Operatorterms (a + 1)•2

Curlyterms werden als zweistellige Operatoren interpretiert (Abb. 3).

1.3.7.
Listenterme

```
Listenterm = leere_Liste |
             Standardliste |offene_Liste |String .
leere_Liste = "[" "]" .
Standardliste = "[" Termfolge "]".
offene_Liste = "[" Termfolge "|" Listenrest "]".
Listenrest = Term .
Termfolge = Subterm { "," Subterm } .
String = " " " { darstellbares_Zeichen } " " " .
```

Die leere Liste wird durch das Atom []/0 repräsentiert. Eine Standardliste entsteht aus einem Term und der leeren Liste bzw. einer Standardliste durch Anwendung des Funktors ./2 . Strings sind eine spezielle Form von Standardlisten. Sie beschreiben eine Liste, deren Elemente ganze Zahlen sind, die den ASCII-Codes der Zeichen zwischen den Anführungsstrichen entsprechen (Abb. 4).

Eine offene Liste ist eine Liste, die aus einem Term und einer freien Variable bzw. einer offenen Liste durch Anwendung des Funktors ./2 entsteht (Abb. 5).

1.4.
Programme

PROLOG-Programme bestehen aus einer Menge von Klauseln, die das Wissen über einen gewissen Gegenstandsbereich repräsentieren. Klauseln sind dabei entweder Fakten, die das Grundwissen bilden, oder Regeln mit deren Hilfe ein gestelltes Problem schrittweise auf bekanntes Grundwissen reduziert werden kann. Fakten und Regeln sind dabei syntaktisch gesehen Terme. Sie können also durch Programme manipuliert werden. Fakten haben die Form

```
name(arg1. ..., argN).
```

und Regeln die Form

```
name(arg1, ..., argN) :-
  name1(arg, ...),
  name2(arg, ...),
  ...
  namel(arg, ...).
```

Die Eingabe in das PROLOG-System besteht aus einer Frage der Form: Gibt es x1,x2,... , die gewissen Eigenschaften genügen. Die Berechnung erfolgt als logischer Problemreduktionsprozess. Ausgehend von der gestellten Frage wird versucht, die Eigenschaft durch schrittweise Reduktion auf bekannte Fakten zu beweisen. Dabei werden nach und nach Variablenbindungen konstruiert. Das Ergebnis der Berechnung ist die Systemantwort "yes" bzw. "no": es gibt Werte für x1,x2,... bzw. es gibt keine Werte. Wenn es Werte für die Variablen gibt, so werden diese als Teil der Systemantwort ausgegeben.

2.
Ein- und Ausgabe

Das Eingabe/Ausgabe Konzept von HU-PROLOG stellt eine Erweiterung des von Clocksin und Mellish beschriebenen Quasistandards dar. HU-PROLOG kennt zu jedem Zeitpunkt genau einen aktuellen Eingabestrom und einen aktuellen Ausgabestrom. Diese können entweder mit Files, oder mit der Standard-Eingabe/Ausgabe des Interpreters verbunden sein. Ein-Ausgabeoperationen beziehen sich immer auf den jeweils aktuellen Ein- bzw. Ausgabestrom. Die Zuordnung des Eingabe- bzw. Ausgabestroms zu den physischen Files wird durch Fileoperationen hergestellt bzw. modifiziert.

2.1.
Fileoperationen

Die Fileoperationen dienen dem Eröffnen und Abschließen physischer Files sowie dem temporären Zuweisen dieser Files zu den aktuellen Eingabe- bzw. Ausgabe-Strömen. Filenamen sind PROLOG-Atome, die ggf. weiteren betriebssystemspezifischen Einschränkungen genügen. An Stelle von Filenamen können auch die vordefinierten Atome stdin/0, stdout/0, stderr/0 und user/0 benutzt werden. Sie bezeichnen die Standardeingabe, Standardausgabe bzw. Standardfehlerausgabe des Interpreters. Das Atom user/0 ist dabei äquivalent zu stdin/0 (bei Operationen für Eingabefiles) bzw. stdout/0 (bei Operationen für Ausgabefiles). Mit dem Start des Interpreters sind der aktuelle Ein- bzw. Ausgabestrom, der Standard-Ein- bzw. Ausgabe des Interpreters zugeordnet.
Im Fehlerfall schlagen die hier beschriebenen Prädikate fehl oder sie führen nach einer Fehlermitteilung zu einem Abbruch der laufenden Abarbeitung. Die Art und Weise der Fehlerreaktion wird durch das Prädikat fileerrors/1 festgelegt.

2.1.1.
open/1

Das Prädikat open/1 eröffnet ein File zum Lesen und zum Schreiben. Wenn dieses nicht möglich ist, so entsteht ein Fehler.

2.1.2.
close/1

Das Prädikat close/1 schließt ein File physisch ab. Wenn das File dem aktuellen Eingabestrom oder Ausgabestrom zugeordnet ist, wird impliziert ein seen/0 bzw. ein told/0 abgear-

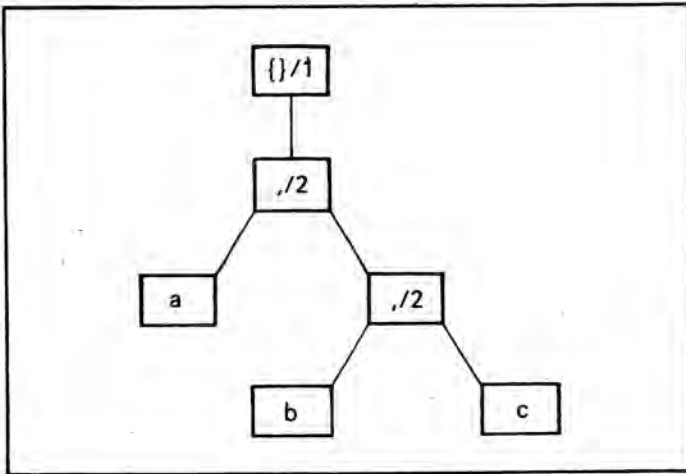


Abb. 3 Interne Struktur des Curlyterms {a,b,c}

beitet. Die Files stdin, stdout, stderr und user sind standardmäßig eröffnet, open/1 und close/1 haben auf diese Files keine Wirkung. Ein Fehler entsteht, wenn close/1 auf ein nicht eröffnetes File angewendet wird.

2.1.3.
see/1, seen/0 und seeing/1

Das Prädikat see/1 dient dem temporären Zuweisen eines Files zum aktuellen Eingabestrom. Ist das File noch nicht eröffnet, so wird es implizit zum Lesen eröffnet. Ein Fehler tritt auf, wenn das Eröffnen zum Lesen nicht möglich, das File nur zum Schreiben eröffnet, oder bereits dem aktuellen Ausgabestrom zugeordnet ist. Der Aufruf von see(user) ist äquivalent zu see(stdin).
Das Prädikat seen/0 schließt das File, welches dem aktuellen Eingabestrom zugeordnet ist, physisch ab und stellt den Eingabestrom auf die Standardeingabe des Interpreters zurück.
Das Prädikat seeing/1 unifiziert sein Argument mit dem Name des dem aktuellen Eingabestrom zugeordneten Files, das heißt dem Atom, das beim letzten Aufruf von see/1 als Parameter angegeben wurde. Wenn see/1 mit dem Argument user/0 aufgerufen wurde, unifiziert seeing/1 sein Argument mit stdin/0.

2.1.4.
tell/1, told/0 und telling/1

Das Prädikat tell/1 dient dem temporären Zuweisen eines Files zum aktuellen Ausgabestrom. Wenn das File noch nicht eröffnet ist, wird es als leeres File zum Schreiben erzeugt. Ein Fehler tritt auf, wenn sich das File nicht eröffnen läßt, es nur zum Lesen eröffnet, oder bereits dem aktuellen Eingabestrom zugeordnet ist. Der Aufruf von tell(user) ist äquivalent zu tell(stdout).
Das Prädikat told/0 schließt das File, welches dem aktuellen Ausgabestrom zugeordnet ist, physisch ab und stellt den Ausgabestrom auf die Standardausgabe des Interpreters zurück.
Das Prädikat telling(X) unifiziert X mit dem Name des dem aktuellen Ausgabestrom zugeordneten Files, das heißt dem Atom, das beim letzten tell/1 angegeben wurde. Wenn tell/1 mit dem Atom user/0 aufgerufen wurde, unifiziert telling/1 sein Argument mit stdout/0.

2.1.5.
seek/2

Das Prädikat seek/2 dient dem Verschieben des Schreib/Lese-Zeigers im File. Zu jedem eröffneten physischen File gehört ein Zeiger, der die aktuelle Position des Files beschreibt. Dieser Zeiger dient, solange das File dem Eingabestrom zugeordnet ist, als Lesezeiger. Er beschreibt dann die Position, von der aus die nächste Leseoperation beginnt, und wird von dieser implizit weitergesetzt. Wird das File dem Ausgabestrom zugeordnet, so dient die gleiche Position als Schreibzeiger. Er beschreibt die Position, auf die das nächste Zeichen geschrieben werden soll und wird bei der Ausgabe weitergestellt. Die Fileposition bleibt dabei an das File gebunden und wird von Veränderungen in der Zuordnung eines Files zum aktuellen Ein- bzw. Ausgabestrom nicht betroffen.
Das erste Argument von seek/2 muß der Name eines eröffneten Files sein. Das zweite Argument kann entweder eine Variable, eine ganze Zahl oder das Atom end/0 sein. Wenn es eine Variable ist, so wird diese mit der aktuellen Position des Schreib/Lese-Zeigers (als Byte-Offset zum Fileanfang) unifiziert. Ist es eine nichtnegative Zahl, so wird der Schreib/Lese-Zeiger auf diese Position relativ zum Fileanfang gesetzt. Ist das zweite Argument das Atom end/0 wird auf das Fileende positioniert.
Wenn das angegebenen File nicht eröffnet ist, oder ein seek nicht möglich ist (z. B. wenn es ein Terminal oder ein anderes Gerät ist) wird ein Fehler ausgelöst.

2.1.6.
Fehlerbehandlung

Die Prädikate fileerror/0, fileerror/1 und nofileerror/0 bestimmen die Reaktion auf Fehler bei den Fileoperationen. Wenn filerrors oder fileerrors(on) abgearbeitet wurde, wird bei Auftreten eines Fehlers die Abarbeitung abgebrochen und eine Fehlermeldung generiert. Der Interpreter kehrt wie bei abort/0 auf den Toplevel zurück. Wenn nofileerrors oder fileerrors(off) abgearbeitet wurde, erfolgt keine spezielle Fehlerbehandlung, das jeweilige Prädikat schlägt einfach fehl.

2.2.
Eingabeoperationen

Die im folgenden beschriebenen Eingabeprädikate beziehen sich alle auf den aktuellen Eingabestrom. Jedem dieser Prädikate kann man den Präfix "tty" voranstellen. Sie beziehen sich dann auf die Standardeingabe des Interpreters, unabhängig von der aktuellen Zuordnung des Eingabestromes.

2.2.1.
read/1

Das Prädikat read/1 liest einen Term vom aktuellen Eingabestrom und unifiziert diesen anschließend mit seinem Argument. Der Term muß mit einem Punkt abgeschlossen sein, dem ein Leerzeichen, Tabulator oder ein Zeilenendezeichen folgt. Wenn der Term syntaktisch nicht korrekt ist, wird eine

Fehlermeldung generiert, die die fehlerhafte Position genau markiert und die laufende Abarbeitung abgebrochen. Am Fileende aufgerufen liefert read/1 das Atom end/0 zurück. Wird PROLOG als Stapelverarbeitungssystem genutzt, so hat das Fileende also die gleiche Funktion wie die Eingabe von "end." im interaktiven Modus. An Stelle von "end." kann man im interaktiven Modus auch versuchen, das Fileendekennzeichen des jeweiligen Betriebssystems direkt einzugeben.

2.2.2.
get0/1 und get/1

Das Prädikat get0/1 liest das nächste Zeichen vom aktuellen Eingabestrom und unifiziert dessen ASCII-Wert mit dem angegebenen Argument. Am Fileende wird der Wert −1 zurückgegeben. Das Prädikat get/1 arbeitet wie get0/1, überliest aber alle nichtdruckbaren Zeichen.

Semantik:

```
get(Code):-
    repeat,
        get0(Char),
    ( Char > 32, Char < 127 ;Char = -1 ),
    !,
    Code = Char.
```

2.2.3.
eof/0 und eoln/0

Die Prädikate eof/0 und eoln/0 testen, ob das Fileende bzw. das Zeilenende erreicht wurde. Diese Prädikate haben keine Seiteneffekte. Das Prädikat eof/0 wird genau dann wahr, wenn das Fileende erreicht ist. Das ist der Fall, wenn das letzte Zeichen des Files bereits eingelesen wurde. Wenn das File ein Terminal ist, wird eof/0 erst nach Lesen des Fileendekennzeichens erfüllt. Das Prädikat eoln/0 wird genau dann wahr, wenn das nächste Zeichen ein Zeilenendezeichen ist, bzw. wenn das Fileende erreicht ist.

2.2.4.
unget/0

Das Prädikat unget/0 sorgt dafür, daß das zuletzt eingelesene Zeichen noch einmal eingelesen werden kann, unabhängig vom eventuellen Umschalten des Eingabestroms. Wiederholtes unget/0 hat keinen Effekt. unget/0 wird immer wahr. unget/0 erlaubt die relativ effiziente Implementierung von Algorithmen, die mit einem Zeichen Vorausschau arbeiten.

2.2.5.
skip/1 und ask/1

Das Prädikat skip/1 dient zum Überspringen von Zeichen auf dem aktuellen Eingabestrom. Das Argument von skip/1 muß ein auswertbarer arithmetischer Ausdruck sein, der eine Integerzahl ergibt. skip/1 liest solange Zeichen vom aktuellen Eingabestrom, bis ein Zeichen mit dem ASCII-Wert dieser Integerzahl gelesen oder das Fileende erreicht wurde. Dieses Prädikat wird meistens zum Überlesen bis zum Zeilenende in

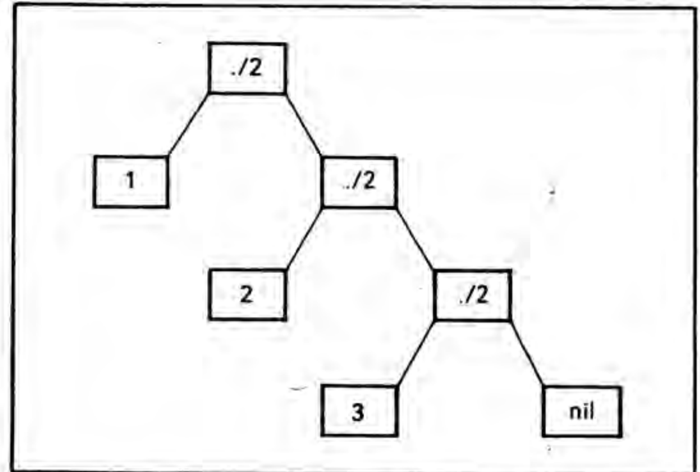


Abb. 4 Interne Darstellung der Standardliste [1,2,3]

der Form skip(10) eingesetzt. Damit lassen sich die bei der interaktiven Arbeit beim Abschließen einer Eingabe zwangsläufig entstehende Zeichen am einfachsten übergehen.
Mit dem Prädikat ask/1 läßt sich das erste Zeichen einer Nutzerantwort bzw. das erste Zeichen einer Zeile abtesten. Das Argument von ask/1 muß ein auswertbarer arithmetischer Ausdruck sein, der eine Integerzahl ergibt. ask/1 liest eine Zeile vom aktuellen Eingabestrom. Das Prädikat ist erfoglreich, wenn das erste Zeichen dieser Zeile einen ASCII-Wert hat, der dieser Integerzahl entspricht.

Semantik:

```
skip(Arith_Term):-
    Value is Arith_Term,
    ( integer(Value) ; abort),
    repeat,
        ( get0(Value) ; eof),!.

ask(Arith_Expression):-
    Value is Arith_Expression,
    ( integer(Value) ; abort),
    get0(First_Char),
    unget,skip(10), % Restzeile überlesen
    !,
    Value = First_Char.
```

2.3.
Ausgabeoperationen

Die im folgenden beschriebenen Ausgabeprädikate beziehen sich alle auf den aktuellen Ausgabestrom. Jedem dieser Prädikate kann man den Präfix "tty" voranstellen. Sie beziehen sich dann auf die Standardausgabe des Interpreters, unabhängig von der aktuellen Zuordnung des Ausgabestromes.

2.3.1.
write/1, writeq/1 und display/1

Diese Prädikate schreiben ihr Argument als PROLOG-Term auf den aktuellen Ausgabestrom. write/1 ist die Standardausgabeform und benutzt eine für den Nutzer gut lesbare Schreibweise, die die aktuellen Operatordeklarationen nutzt. Atome werden nicht gequotet. Ungebundene Variablen werden als Unterstrich, gefolgt von einer Zahl dargestellt. writeq/1 arbeitet wie write/1, jedoch werden Atome, wenn nötig, gequotet. Mit writeq/1 ausgegebene Terme können daher später immer wieder eingelesen werden. display/1 dagegen

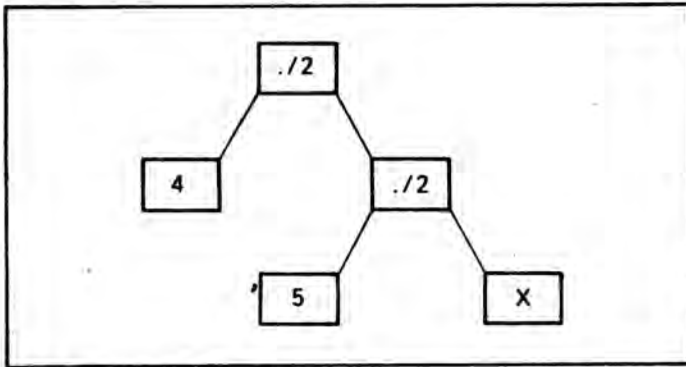


Abb. 5 Interne Darstellung der offenen Liste [4,5|X]

nutzt die Operatorschreibweise von PROLOG nicht. Alle Terme werden in ihrer Standardnotation als Funktorterm ausgegeben. Dies ist ein Hilfsmittel, um die Interpretation von PROLOG-Termen, die in Operatorschreibweise vorliegen, zu testen.

Beispiel:

```
? – X = 'atom 1' + atom2, writeq(X),nl, display(X),nl.
'atom 1' + atom2
+('atom 1',atom2)
X = atom 1 + atom2
yes
?–
```

2.3.2.
put/1

Das Argument von put/1 muß ein auswertbarer arithmetischer Ausdruck, der eine Integerzahl ergibt, oder eine Liste von Integerzahlen sein. Das Zeichen mit dem ASCII-Wert dieser Zahl wird auf den aktuellen Ausgabestrom ausgegeben. Wenn diese Integerzahl nicht im Bereich von 0 bis 255 liegt, ist das Resultat undefiniert. Wenn das Argument von put/1 eine Liste von Integerzahlen ist, so werden die zugehörigen Zeichen nacheinander ausgegeben.

Beispiel:

```
?– put(64 + 1),nl,put("prolog"),nl.
A
prolog
yes
?–
```

2.3.3.
nl/0 und tab/1

Das Prädikat nl/0 schreibt das Zeichen bzw. die Zeichenfolge für das Zeilenende auf den aktuellen Ausgabestrom.
Das Prädikat tab/1 schreibt eine Anzahl von Leerzeichen auf den Ausgabestrom. Das Argument von tab/1 muß ein auswertbarer arithmetischer Ausdruck sein, der eine nichtnegative Integerzahl ergibt. Eine entsprechende Anzahl Leerzeichen wird auf den aktuellen Ausgabestrom geschrieben.

2.3.4.
cls/0 und gotoxy/2

Das Prädikat cls/0 sendet das Zeichen bzw. die Zeichenfolge zum Löschen des Bildschirms auf das aktuelle Ausgabefile.
Das Prädikat gotoxy(X_Val,Y_Val) sendet die Zeichenfolge zur Cursorpositionierung auf den aktuellen Ausgabestrom. X_Val und Y_Val müssen dabei auswertbare arithmetische Ausdrücke sein, die nichtnegative Intergerzahlen ergeben. Die zulässigen Wertintervalle ergeben sich aus den zulässigen Bildschirmgrößen. Der Cursor wird auf die Spalte X_Val und die Zeile Y_Val positioniert. Wenn die Werte von X_Val und Y_Val nicht im vorgegebenen Bereich des Bildschirms liegen, ist das Ergebnis undefiniert. Die linke obere Ecke hat die Koordinaten 1,1.

3. Termbehandlung

3.1. Termklassifikation

Zur Termklassifikation stellt HU-PROLOG die Prädikate var/1, nonvar/1, atom/1, integer/1, number/1, atomic/1, invar/1 und ground/1 zur Termklassifikation zur Verfügung. Diese Prädikate haben ein deterministisches Verhalten. Beim Backtracking erzeugen sie keine neuen Lösungen.
Tab. 1 zeigt, welche Bedingungen die Argumente der Prädikate erfüllen müssen, damit ein Aufruf des jeweiligen Prädikats erfolgreich ist:
Diese Bedingungen sind rein syntaktischer Natur, es spielt also z. B. keine Rolle, ob eine Real-Zahl einen ganzzahligen Wert repräsentiert. Die Übersichten in Tab. 2 zeigen für einige Beispielterme, wie sich die Prädikate verhalten. Dabei bedeutet eine 0, daß der Aufruf fehlschlägt und eine 1, daß er wahr wird.
Die Prädikate var/1, atom/1, real/1 und integer/1 sind elementar und widerspiegeln die interne Struktur der PROLOG-Terme. Dagegen lassen sich die Prädikate nonvar/1, number/1, atomic/1, invar/1 und ground/1 in PROLOG definieren.

3.2. Termanalyse und -synthese

Die Prädikate =../2, functor/3 und arg/3 dienen der Analyse und Synthese von Termen. Sie verhalten sich deterministisch und erzeugen daher beim Backtracking keine weiteren Lösungen.

3.2.1.
=../2

Das sogenannte univ-Prädikat =../2 (in Operatorschreibweise) dient der Umwandlung zwischen beliebig strukturierten Termen und Listen. Die zu einem Term äquivalente Liste ist entsprechend der Termform definiert:
– zu einer Zahl ist die einelementige Liste, bestehend aus ebendieser Zahl, äquivalent
– zu einem Atom ist die einelementige Liste, bestehend aus diesem Atom, äquivalent
– zu einem strukturierten Term der Form f(t1,...,tn) ist die (n + 1)-elementige Liste, bestehend aus dem Hauptfunktor f des Terms, gefolgt von den einzelnen Argumenten, äquivalent.
Die Funktion des Prädikats =../2 wird durch das erste Argu-

Prädikat	ist erfolgreich, falls:
atom(X)	X ein Atom ist
integer(X)	X eine Integerzahl ist
real(X)	X eine Realzahl ist
number(X)	X eine Zahl (Integer oder Real) ist
atomic(X)	X ein Atom oder eine Zahl ist
var(X)	X eine freie Variable ist
nonvar(X)	X keine freie Variable ist
ground(X)	X keine freie Variable enthält
invar(X)	X eine freie Variable enthält

Tab. 1

ment bestimmt. Handelt es sich dabei um eine freie Variable, so synthetisiert das Prädikat =../2 aus dem zweiten Argument, das in diesem Fall eine vollständige nichtleere (Standard)Liste sein muß, einen Term zu dem diese Liste äquivalent ist und unifiziert das erste Argument mit diesem Term. Gibt es keinen solchen Term oder ist das zweite Argument eine nichtleere Liste, so wird ein Fehler ausgelöst. Ist das erste Argument keine freie Variable, so wird die zu diesem Term äquivalente Liste konstruiert und mit dem zweiten Parameter unifiziert. Das univ-Prädikat ist ein Standardhilfsmittel, um Prologprozeduren zu realisieren, die beliebig strukturierte Terme verarbeiten können, indem diese in eine einheitliche Listenform transformiert werden.

3.2.2. functor/3

Die Funktion des Prädikates functor/3 wird durch das erste Argument bestimmt. Handelt es sich dabei um eine freie Variable, so synthetisiert das Prädikat functor/3 aus den anderen beiden Argumenten, die in diesem Fall vollständig instantiiert sein müssen, einen neuen Term und bindet die Variable des ersten Argumentes an diesen Term. Ist das erste Argument keine freie Variable, so wird die Struktur dieses Terms analysiert. Die weiteren Argumente werden dann mit den Parametern der Termstruktur, d. h. dem Funktor und der Stelligkeit unifiziert. Der Aufruf ist erfolgreich, wenn beide Unifizierungen erfolgreich sind, andernfalls schlägt er fehl. Die Termsynthese erzeugt den allgemeinsten Term, dessen anschließende Analyse genau die vorgegebenen Parameter liefern würde.

Semantik:

Sei length/2 ein Prädikat, daß zur einer gegebenen Liste L ihre Länge L bestimmt oder zu einer gegebenen Länge N eine Liste L aus genau N frischen Variablen erzeugt. Dieses Prädikat läßt sich in PROLOG wie folgt beschreiben:

```
length(0,[]).
length(N,[_|T]) :- length(M,T), N is M + 1.
```

Mit Hilfe von length/2 läßt sich die Semantik von functor/3 auf =../2 zurückführen:

```
functor(T,T,0) :-
        atomic(T),
        !.
functor(T,F,N) :-
        nonvar(T),
        T =..[F|L],
        length(N,L),
        !.
functor(T,F,N) :-
        var(T),atom(F),
        integer(N),0 =< N,
        length(N,L),
        T =.. [F|L],
        !.
```

3.2.3. arg/3

arg(N,T,A) unifiziert das N-te Argument des strukturierten Terms T mit A. Falls N keine Integerzahl oder T kein strukturierter Term ist oder falls N eine Integerzahl kleiner 1 oder größer als die Stellenzahl des strukturierten Terms T ist oder falls die Unifikation des N-ten Arguments von T mit A nicht möglich ist, schlägt arg/3 fehl.

Semantik:

Sei n_th/3 ein Prädikat, das bei einem Aufruf n_th(N,L,A) das Argument A mit dem N-ten Argument der Liste L unifiziert. Ein solches Prädikat läßt sich in Prolog z. B. folgendermaßen definieren:

```
n_th(1,[A|_],A).
n_th(N,[_|T],A) :- N > 1, M is N - 1, n_th(M,T,A).
```

Mit Hilfe von n_th/3 läßt sich die Semantik von arg/3 auf =../2 reduzieren:

```
    arg(N,T,A) :- T =.. [F|L],n_th(N,L,A).
```

3.3. Analyse und Synthese von Atomen

Atome sind die elementaren symbolischen Daten in PROLOG. Für verschiedene Anwendungen kann es trotzdem notwendig werden, Atome näher zu analysieren oder neue Atome zu erzeugen.

3.3.1. current_atom/1

Ein Aufruf von current_atom(Name / Arity) unifiziert Name und Arity mit dem Namen und der Stellenzahl eines dem Interpreter bekannten Atoms. Beim Backtracking liefert das Prädikat alle dem System aktuell bekannten Atome.

3.3.2. name/2

name(A,L) konvertiert Atome und Zahlen in Strings, d. h. Listen von Integerzahlen, die die ASCII-Codes der Zeichen des Atoms repräsentieren und umgekehrt ASCII-Listen in Atome. Falls A ein Atom oder eine Zahl ist, wird eine Liste der ASCII-Werte der Zeichen des Atoms oder der Zahl erzeugt und mit L unifiziert. Ist diese Unifikation möglich, wird name/2 erfolgreich. Andernfalls schlägt es fehl. Falls A eine Variable ist, muß L an eine ASCII-Liste gebunden sein. Aus dieser Liste wird ein Atom mit den Zeichen entsprechend der ASCII-Codes konstruiert. Anschließend wird A mit dem konstruierten Atom unifiziert.

Beispiel:

```
?-name(prolog,X).
X = "prolog"        % ASCII-Liste
yes
?-name(X,[112,114,111,108,111,103]).
X = prolog
yes
?-name([],X).
X = "[]"
yes
?-
```

Objekt	atom	integer	real	number	atomic	var	nonvar	invar	ground
X	0	0	0	0	0	1	0	1	0
–	0	0	0	0	0	1	0	1	0
98	0	1	0	1	1	0	1	0	1
9.8e1	0	0	1	1	1	0	1	0	1
'98'	1	0	0	0	1	0	1	0	1
"98"	0	0	0	0	0	0	1	0	1
[]	1	0	0	0	1	0	1	0	1
[1,e]	0	0	0	0	0	0	1	0	1
hallo	1	0	0	0	1	0	1	0	1
f(X,a)	0	0	0	0	0	0	1	1	0
sin(pi/2)	0	0	0	0	0	0	1	0	1

Tab. 2

3.4. Termvergleich

PROLOG stellt verschiedene Operatoren zum Vergleich von Termen zur Verfügung. Terme können auf Unifizierbarkeit, Identität und ihre lexikographische Ordnung untersucht werden.

3.4.1. Unifizierbarkeit (=/2 und \=/2)

Die Prädikate =/2 und \=/2 vergleichen Terme durch Test auf Unifizierbarkeit. Sie verhalten sich deterministisch und erzeugen daher beim Backtracking keine weiteren Lösungen. Das Prädikat =/2 beschreibt die explizite Unifizierbarkeit. Falls diese Unifikation möglich ist, wird sie ausgeführt und =/2 wird wahr. Andernfalls schlägt es fehl. Das Prädikat \=/2 ist die Umkehrung des Prädikates =/2. Es wird wahr, falls die Unifikation der Argumente nicht möglich ist und schlägt fehl, wenn sie möglich ist. Dabei wird die Unifikation in keinem Fall ausgeführt.

3.4.2. Identität (==/2 und \==/2)

Die Prädikate ==/2 und \==/2 vergleichen ihre Argumente auf Identität. Sie verhalten sich deterministisch und erzeugen beim Backtracking keine weiteren Lösungen.
Die Identitätsrelation zweier Terme ist wie folgt definiert:
– Zahlen sind mit Zahlen identisch, die vom gleichen Typ (Integer bzw. Real) sind und denselben Wert repräsentieren. Mit anderen Termen sind Zahlen nicht identisch.
– Atome sind nur mit sich selbst identisch.
– Variablen sind nur mit Variablen identisch, mit denen sie vorher unifiziert wurden. Mit anderen Termen sind Variablen nicht identisch.
– Strukturierte Terme sind mit strukturierten Termen identisch, die denselben Hauptfunktor und dieselbe Stelligkeit besitzen und deren Argumente paarweise identisch sind. Mit anderen Termen sind strukturierte Terme nicht identisch.
Das Prädikat ==/2 testet seine Argumente auf Identität. Es wird genau dann erfolgreich, wenn die Argumente identisch sind. Das Prädikat \==/2 ist die Umkehrung von ==/2 und wird genau dann erfolgreich, wenn die Argumente nicht identisch sind.

3.4.3. Lexikographische Ordnung (@<,@=<,@=,@>=,@>,@\=)

Die Prädikate @<,@=<,@>,@>=,@= und @\= vergleichen zwei Terme in Bezug auf ihre lexikographische Ordnung. Sie verhalten sich deterministisch und erzeugen beim Backtrakking keine weiteren Lösungen.
Die lexikographische Ordnung zwischen Termen ist wie folgt definiert (Tab. 3):
– Variablen sind kleiner als alle anderen Terme und untereinander gleich.
– Zahlen sind kleiner als alle nichtvariablen Terme und sind untereinander nach ihrem numerischen Wert sortiert. Dabei spielt der Zahlentyp (Integer oder Real) keine Rolle.
– Atome sind größer als Zahlen und Variablen und kleiner als alle anderen Terme. Untereinander sind Atome nach ihrer lexikographischen Ordnung entsprechend dem ASC II-Code sortiert.
– Strukturierte Terme sind in der Ordnung die größten Terme. Untereinander sind sie nach den Hauptfunktoren und (falls diese gleich sind) nach den Argumenten gemäß ihrer lexikographischen Ordnung sortiert.

4. Manipulation der Datenbasis

Dieser Abschnitt beschreibt die Prädikate zur Modifikation und Abfrage der Datenbasis. HU-PROLOG basiert auf einer einheitlichen internen Datenbasis. Die Eintragungen in die Datenbasis (Klauseln) haben die Form spezieller PROLOG-Terme. Die Datenbasis zerfällt logisch in Prozeduren. Das sind Teilmengen von Klauseln mit gleichem Hauptfunktor. Innerhalb einer Prozedur sind die Klauseln streng sequentiell geordnet. Die interne Ordnung entspricht zunächst genau der Reihenfolge des Ladens und damit der Reihenfolge im Quelltext. Klauseln können aber während der Abarbeitung des Programmes dynamisch erzeugt und gestrichen werden. Die direkte Bezugnahme auf Klauseln der Datenbasis wird durch symbolische Datenbasisreferenzen der Form $ db_ref(n) ermöglicht. Diese Datenbasisreferenzen entstehen als Seiteneffekte beim Eintragen oder Suchen einer Klausel und können beim wiederholten Zugriff auf die Klausel gerade im Falle großer Datenbasen die Effizienz der Abarbeitung deutlich erhöhen.

4.1. Einfügen von Klauseln

4.1.1.
asserta/1 und asserta/2

Die Prädikate asserta/1 und asserta/2 dienen zum Einfügen neuer Klauseln am Anfang der jeweiligen Prozedur. Das erste Argument ist der einzufügende Term, in der Form:

```
Head
bzw.
(Head :- Body)
```

Head darf dabei weder eine freie Variable sein, noch darf der Hauptfunktor von Head der Name eines Systemprädikats sein. Die Prädikate asserta/1 und asserta/2 tragen die neue Klausel vor allen anderen Klauseln der entsprechenden Prozedur (mit dem selben Hauptfunktor von Head) in die Datenbasis ein. Das zweite Argument von asserta/2 muß eine freie Variable sein. Diese wird an die Datenbasisreferenz der neu eingefügten Klausel gebunden. Diese Datenbasisreferenz kann später in assert/3, clause/3 und retract/2 zum gezielten Zugriff auf diese Klausel genutzt werden.

4.1.2.
assertz/1 und assertz/2

Diese Prädikate dienen zum Einfügen neuer Klauseln an das Ende der jeweiligen Prozedur. Sie entsprechen ansonsten den Prädikaten asserta/1 und asserta/2. Der Unterschied wird dadurch angedeutet, daß a der erste und z der letzte Buchstabe des Alphabets sind.

4.1.3.
assert/1 und assert/2

Diese Prädikate sind synonym zu assertz/1 bzw. assertz/2. Sie werden am häufigsten verwendet und entsprechen dem Einfügen am Ende. Dadurch entspricht die interne Reihenfolge der Klauseln genau der Reihenfolge des Erzeugens.

4.1.4.
assert/3

Das Prädikat assert/3 stellt eine Verallgemeinerung der oben beschriebenen Prädikate der assert-Familie dar. Das erste Argument entspricht der einzufügenden Klausel in Termform. Das zweite Argument muß eine freie Variable sein, die an den Datenbasisreferenzterm der eingefügten Klausel gebunden wird. Das dritte Argument steuert die Position, an der die Klausel in die Datenbasis eingefügt werden soll.
Wenn dieses Argument eine Zahl N ist, wird die neue Klausel nach der N-ten, zu diesem Funktor schon existierenden Klausel in die Datenbasis eingefügt. Wenn N gleich 0 ist, so wird sie vor der ersten Klausel eingefügt. Ein Fehler tritt auf, wenn N negativ oder größer als die Anzahl der Klauseln der jeweiligen Prozedur in der Datenbasis ist. Wenn das dritte Argument von assert/3 das Atom end/0 ist, so wird die neue Klausel nach allen anderen Klauseln dieses Prädikats in die Datenbasis eingefügt. Das dritte Argument kann auch eine Datenbasisreferenz zu einer Klausel dieses Prädikats sein. Dann wird die neue Klausel nach der durch diese Datenbasisreferenz bezeichneten Klausel in die Datenbasis eingetragen. In allen anderen Fällen tritt ein Fehler auf. Die Prädikate der assert-Familie lassen sich auf assert/3 zurückführen.

Semantik:

```
asserta(Clause)  :-assert(Clause,_,0).
assertz(Clause)  :-assert(Clause,_,end).
assert(Clause)   :-assert(Clause,_,end).

asserta(Clause,DB_REF)   :-assert(Clause,DB_REF,0).
assertz(Clause,DB_REF)   :-assert(Clause,DB_REF,end).
assert(Clause,DB_REF)    :-assert(Clause,DB_REF,end).
```

4.2. Entfernen von Klauseln

4.2.1.
retract/1 und retract/2

Das Prädikat retract/1 löscht die erste Klausel der Datenbasis, deren Kopf bzw. Kopf und Körper mit dem Argument von retract/1 unifizierbar ist. Beim Backtracking werden die weiteren mit dem Argument unifizierbaren Klauseln gelöscht. Nach dem Löschen einer Klausel ist das Argument an einen Term gebunden, der genau der gestrichenen Klausel entspricht. Das Argument von retract/1 muß mindestens soweit instantiiert sein, daß der Name und die Arität des Prädikats feststehen. retract/1 schlägt fehl, wenn keine Klausel existiert, die unifiziert werden kann. Ein Fehler tritt auf, wenn versucht wird, ein Systemprädikat zu löschen.
retract/2 arbeitet analog zu retract/1, erwartet aber als zweites Argument eine freie Variable oder eine Datenbasisreferenz. Es löscht die erste Klausel aus der Datenbasis, die mit dem ersten Argument und deren Datenbasisreferenz mit dem zweiten Argument von retract/2 unifiziert werden kann. Beim Backtracking werden – wenn möglich – neue Lösungen generiert und die entsprechenden Klauseln gelöscht. Wenn als zweites Argument eine Variable angegeben ist, unterliegt das erste Argument den gleichen Einschränkungen wie bei retract/1, d. h. der Hauptfunktor der Klausel muß eindeutig bestimmt sein.

Tab. 3

X op Y	ist erfolgreich, wenn in der lexikographischen Ordnung
X @< Y	X kleiner als Y ist
X @=<Y	Y kleiner oder gleich Y ist
X @> Y	X größer als Y ist
X @>=Y	Y größer oder gleich Y ist
X @= Y	X gleich Y ist
X @\= Y	X ungleich Y ist

1200	xfy	:–	globale
1200	fx	:– , ?–	
1100	xfy	;	Steuerung
1050	xfy	– >	
1000	xfy	,	
900	fy	not , \+	
700	xfx	:= , is	Arithmetik
700	xfx	\== , \= , @= , @>= @> , @== , @=< , @< >= , > , \== , =< =:= , = , < , =\=	Vergleichs-operationen
700	xfx	=..	univ
650	xfy	\ , \\ , & , &&	arithmetische Operationen
650	fy		
600	xfy	>> , <<	
500	yfx	+ , – ,	
400	yfx	* , / , // , mod	
350	xfy	**	
300	xfy	.	
300	fy	– , /	

Tab. 4

4.2.2.
retractall/1

Das Prädikat retractall/1 löscht alle Klauseln aus der Datenbasis, deren Kopf sich mit dem Argument von retractall/1 unifizieren läßt. Diese Unifizierung wird jedoch nicht ausgeführt; das heißt, der Parameter ist nach dem Aufruf von retractall/1 unverändert. Das Prädikat retractall/1 wird immer wahr.

Semantik:

```
retractall(X):-
        X \= (_:-_),
        (retract(X);retract((X :-_)),
    fail.
retractall(_).
```

4.2.3.
abolish/1 und abolish/2

Das Prädikat abolish/2 erwartet als erstes Argument den Namen eines Prädikats und als zweites eine nichtnegative Zahl – die Stelligkeit des Prädikats. Der Aufruf abolish(Name,Arity) löscht alle Klauseln, die zu diesem Namen und dieser Stellenzahl in der Datenbasis eingetragen sind. Das Prädikat wird immer wahr.
abolish/1 erwartet als Argument den Namen eines Prädikats. Es löscht alle Klauseln, die zu diesem Namen in der Datenbasis existieren.

Semantik:

```
abolish(Name,Arity):-
    functor(F,Name,Arity),
    retractall(F).
abolish(X):-
    current_atom(X / N),
    abolish(X,N),
    fail.
abolish(_).
```

4.3.
Abfragen von Klauseln

4.3.1.
clause/2 und clause/3

Diese Prädikate dienten zum Abfragen der Datenbasis. Beim Aufruf von clause(Head,Body) wird eine Klausel gesucht, deren Kopf mit Head und deren Körper mit Body unifizierbar ist. Diese Unifizierung wird ausgeführt. Für Fakten wird als Klauselkörper true/0 angesetzt. Beim Backtracking werden – wenn möglich – weitere Lösungen erzeugt. Aus dem ersten Argument muß der Hauptfunktor des Prädikats hervorgehen.
Das Prädikat clause/3 arbeitet analog zu clause/2, unifiziert aber gleichzeitig das dritte Argument mit der Datenbasisreferenz der Klausel. Es muß entweder aus dem ersten Argument der Name und die Stellenzahl des Prädikats hervorgehen, oder das dritte Argument muß eine Datenbasisreferenz und keine freie Variable sein.

4.3.2.
sys/1

Das Prädikat sys/1 testet, ob sein Argument der Name eines "Systemprädikats" ist. Systemprädikate sind die Prädikate, die als built-in-Prädikate im Interpreterkern vorliegen oder beim Systemstart mit Hilfe der -s-Option bzw. aus dem .prologrc File geladen wurden. Der Zugriff auf diese Prädikate ist beschränkt, man hat keinen expliziten Zugriff auf die Klauseln dieser Prädikate. Das Argument von sys/1 muß die Form Name/Arity haben.

4.3.3.
listing/1 und listing/0

Das Prädikat listing/0 gibt alle vom Nutzer definierten Klauseln auf den Standardausgabestrom aus. Das Prädikat listing/1 spezifiziert die auszugebenden Klauseln näher. listing(all) ist äquivalent zu listing/0. listing(Atom) gibt alle Klauseln aus, die Atom unabhängig von der Stellenzahl als Hauptfunktor des Klauselkopfes haben. listing(Atom / Arity) gibt alle Klauseln zum Prädikat Atom/Arity aus.

4.3.4.
dict/1 und sdict/1

Ein Aufruf von sdict(X) unifiziert X mit der Liste aller Funktoren von Systemprädikaten in der Form Atom/Arität. dict(X) liefert die Liste aller nutzerdefinierten Prädikate sowie der inplizit benutzten Funktoren.

4.4.
Operatordeklarationen

4.4.1.
op/3

Der Aufruf von op(V,A,N) macht dem Interpreter einen neuen Operator bekannt. N muß ein Atom, das für den Namen des Operators stehen soll, oder eine Liste solcher Namen sein. V muß ein Integerwert sein, der den Vorrang des neuen

Name	Resultattyp	Bedeutung
maxint	int	größte darstellbare ganze Zahl (2147483647)
minint	int	kleinste darstellbare ganze Zahl (–2147483648)
e	real	Eulersche Zahl (2.7182818284)
pi	real	Kreiszahl Pi (3.1415926535)

Tab. 5

Operators bestimmt. Der Wert von V muß zwischen 0 und 999 liegen. Je größer der Wert von V ist, desto stärker trennt der Operator. Eine Liste aller zum aktuellen Zeitpunkt definierten Operatoren kann man mit current_op/3 erhalten. A gibt die Assoziativität des neuen Operators an. Für A sind folgende Atome erlaubt:

xfx , xfy , yfx	für infix Operatoren
xf , yf	für postfix Operatoren
fx , fy	für prefix Operatoren.

Dabei symbolisiert f den Operator sowie x und y seine Argumente. Anstelle von x darf nur ein Ausdruck stehen, dessen Vorrang kleiner als V ist. Bei y darf der Vorrang auch gleich V sein. Die Unterscheidung von x und y definiert damit die Klammerung der Operatoren.

Beispiel:

Die Assoziativität von –/2 ist standardmäßig yfx. Damit wird ein Ausdruck der Form a–b–c wie gewohnt als (a–b)–c interpretiert. Beim Operator ./2 wird hingegen die Assoziativität xfy benutzt. Deswegen wird a.b.c.nil ist als a.(b.(c.nil)) interpretiert.
Eine einfache Möglichkeit, die Interpretation der Operatoren zu veranschaulichen, besteht im folgenden kleinen Programm:

```
?- repeat,
      write('enter expression: '),
      read(Term),
      display(Term), % Ausgabe ohne Operatorschreibweise
   nl, Term == end.
enter expression: a + b * c.
+(a, *(b, c))
enter expression: a + b – c * d.
–(+(a, b), *(c, d))
enter expression: a / b / c.
/(/(a, b), c)
enter expression: a.b.c.nil.
.(a, .(b, .(c, nil)))
enter expression: test(X) :- X > 5 , X < 10.
:-(test(_1), ',' (>(_1, 5), <(_1, 10)))
enter expression: end.
end
T = end
yes
?-
```

Es ist nicht möglich, einen Operatornamen gleichzeitig als Infix und Postfix zu vergeben, da dieser Fall beim Einlesen nicht eindeutig zu behandeln ist. Der Aufruf op (Ø, A, N) löscht eine eventuell vorhandene Operatordeklaration. Ein Fehler tritt auf, wenn eines der Argumente von op/3 einen unzulässigen Wert hat.

4.4.2.
current_op/3

Ein Aufruf von current_op(V,A,N) unifiziert V,A und N mit dem Vorrang, der Assoziativität und dem Namen eines dem System bekannten Operators. Beim Backtracking werden neue Lösungen generiert. Das folgende Programm listet alle dem Interpreter bekannten Operatoren auf.

```
list_op :-
      current_op(V,A,N),
      write((?-op(V,A,N))),nl,
      fail.
```

4.4.3.
Standardoperatoren

In HU-PROLOG sind die Operatoren in Tab. 4 standardmäßig deklariert:

5. Arithmetik und funktionale Programmierung

5.1. Einleitung

Die herkömmliche Integer- und Real-Arithmetik ist in HU-PROLOG in drei Richtungen verallgemeinert worden:

- Die von C bekannten und bewährten Operationen wurden, soweit sie mit den PROLOG-Grundkonzepten verträglich waren, einschließlich Syntaxregeln, Prioritäten und Assoziativitäten übernommen. Die Standardfunktionen von HU-PROLOG entsprechen in ihrer Funktion den gleichnamigen Funktionen der C-Standardbibliothek.
- Operationen und Funktionen zur Manipulation von Termstrukturen wurden aufgenommen, so daß beliebige Terme sowohl als Argument, wie auch als Resultat von Operationen auftreten können.
- Die funktionale Auswertung von Prädikaten sowie die Wertzuweisung und das Dereferenzieren globaler Variablen werden unterstützt.

Die ersten beiden Punkte sind mit den Grundkonzepten von PROLOG ohne weiteres verträglich. Daher wurde der is/2-Operator als primitiver Grundbaustein entsprechend erweitert.
Die funktionalen und prozeduralen Erweiterungen erweisen sich in der praktischen PROLOG-Programmierung als überaus leistungsfähig, erlauben sie doch das Umgehen einiger konzeptioneller Schwachstellen von PROLOG. Doch gerade diese Konstruktionen werden von einigen Schulen der "reinen" PROLOG-Programmierung verpönt und sind in anderen PROLOG-Systemen auch nicht realisiert. Wir haben daher ei-

Name	Argument-typ	Resultattyp	Bedeutung
–X	int/ real	int/ real	arithmetische Negation (Minus)
/(X) ~(X)	int	int	logische Negation (!X) bitweise Negation (~X)
real(X) entier(X)	int real	real int	Typkonvertierung
exp(X) ln(X) log10(X) sqrt(X) sin(X) cos(X) tan(X) asin(X) acos(X) atan(X) floor(X) ceil(X)	real	real	Exponentialfunktion (Basis e) Logarithmusfunktion (Basis e) Logarithmusfunktion (Basis 10) Quadratwurzel Sinusfunktion Cosinusfunktion Tangensfunktion Arcussinusfunktion Arcuscosinusfunktion Arcustangensfunktion nächstkleinere ganze Zahl nächstgrößere ganze Zahl
\`(X) eval(X)	term	term	unterdrückt Argumentauswertung doppelte Auswertung des Argumentes

Tab. 6

nen verallgemeinerten Wertzuweisungsoperator :=/2 eingeführt, mit dem die Benutzung der erweiterten Komponente sauber gekapselt werden kann. Die Portabilität zu anderen PROLOG-Systemen bleibt gewahrt, indem sich eine, wenn auch nicht so effektive Standard-PROLOG-Defnition von :=/2 angeben läßt.

5.2. Standardfunktionen und -operationen

Die Standardfunktionen und -operationen von HU-PROLOG werden ausschließlich auf der rechten Seite der Prädikate is/2 und :=/2, sowie auf beiden Seiten der Vergleichsprädikate </2, >/2, =</2, >=/2, =:=/2 und =\=/2 ausgewertet. Die folgenden Tabellen geben eine Übersicht über die verfügbaren Funktionen und Operationen und deren mögliche Argument- und Resultattypen.

5.3. is/2

Ein Aufruf T is E wertet den Ausdruck E aus und unifiziert das Ergebnis mit dem Term 'T'. Je nachdem, ob diese Unifikation möglich ist oder nicht, ist is/2 erfolgreich bzw. schlägt fehl. Die Auswertung eines Ausdrucks geschieht analog der Auswertung in klassischen Programmiersprachen.

• Die Funktion '/1 (Quote) wird direkt zu ihrem Argument ausgewertet.

• Bei Funktionen und Operationen werden zuerst alle Argumente in der Reihenfolge ihres Auftretens, von links nach rechts ausgewertet. Die Funktion wird anschließend auf diese Ergebnisse angewendet. Der Resultattyp der Funktion ist im allgemeinen von den Argumenttypen abhängig. Eine Typanpassung von integer nach real erfolgt automatisch. Falls die Argumenttypen nicht mit den geforderten übereinstimmen und sich nicht anpassen lassen, wird die Funktion zu einem Term ausgewertet, der aus der Funktion selbst und den ausgewerteten Argumenten besteht.

• Zahlen werden zu ihren Werten ausgewertet.

• Freie Variablen werden zu Variablen ausgewertet.

Beispiele:

Das Prädikat genint/1 erzeugt beim Backtracking fortlaufend wachsende Integerzahlen.

```
genint(0).
genint(I) :– genint(Y), I is Y+1.
```

fak(F,N) berechnet die Fakultät F von N. run/0 erzeugt eine Tabelle der Fakultätsfunktion

```
fak(1,0) :– !.              % 0! ist 1
fak(F,N) :–
        N1 is N - 1,N1 >= 0,
        fak(F1,N1),         %N! ist (N–1)! * N
        F is F1 * N,!.

run:– genint(X),
        fak(F,X),
        write(X),write('!   = '), write(F),nl,
        X>=10.
?-run.
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
yes
?–
```

5.4. Globale Variablen und Wertzuweisungen

Der Wert w einer globalen Variablen v läßt sich in PROLOG am einfachsten durch einen Fakt der Form v(w) beschreiben. Der Wert einer Variablen gilt als undefiniert, wenn in der Datenbasis keine Klausel dieser Form existiert. Beim Auswerten eines Terms, der eine Variable v enthält, wird diese zu ihrem Wert dereferenziert, indem eine Datenbasisabfrage v(X) ausgeführt wird. Das Ergebnis dieser Abfrage bindet den Wert von v an eine Hilfsvariable X, die in die weitere Auswertung des Terms eingeht. Die Zuweisung eines Wertes zu einer globalen Variablen erfolgt in zwei Schritten. Zunächst werden mit einem speziellen retract alle Informationen über den Wert der Variablen aus der Datenbasis gestrichen. Anschließend wird mit asserta(v(w)) der neue Wert eingetragen. Der Effekt der Wertzuweisung v:=v+1 läßt sich in PROLOG durch folgenden Aufruf beschreiben:

```
?– v(X), Y is X+1, retractall(v(_)),assert(v(Y)).
```

Name	Argumenttypen	Resultattyp	Bedeutung
B << N B >> N	int, int	int	bitweise Links- bzw. Rechts-verschiebung von B um N Stellen
A & B A \ B A && B A \\ B	int, int	int	bitweise Konjunktion Disjunktion logische Konjunktion Disjunktion
B ** E	real, int	real	Potenzfunktion (B hoch E)
A + B A – B A * B	real bzw. int	real bzw. int	Addition Subtraktion Multiplikation
A / B	real, real	real	reelle Division
A // B A mod B	int, int	int	ganzzahlige Division Modulo (A % B)

Tab. 7

Dieses Prinzip läßt sich von einfachen Variablen auf strukturierte Variablen übertragen und führt zu einer Verallgemeinerung von Feldern und Datensatztypen, indem als Indexbereich nicht nur Zahlenintervalle und Bezeichner, sondern allgemein Termstrukturen zugelassen werden.
Eine n-stellige strukturierte Variable wird durch ein (n+1)-stelliges Prädikat beschrieben, dessen erstes Argument stets den Wert der Variablen an der durch die restlichen n Argumente gegebenen Stelle enthält.

```
anzahl(schraube(m4)) := anzahl(schraube(m4)) – 1
```

entspricht der PROLOG-Aufruffolge

```
?- anzahl(X,schraube(m4)),
   Y is X-1,
   retractall(anzahl(_,schraube(m4))),
   asssserta(anzahl(Y,schraube(m4))).
```

5.5. Auswertung von Funktionen und Variablen

Die Auswertung n-stelliger Funktionen und strukturierter Variablen ordnet sich kanonisch in dieses Konzept ein, indem sie als (n+1)-stellige Prädikate definiert werden. Die Parameterübergabe erfolgt dabei durch "call-by-name". Die Art und Weise der weiteren Parameterauswertung muß durch das Prädikat selber bestimmt werden.
Die Auswertung von Termen erfolgt von außen nach innen:
- Zahlen und freie Prolog-Variablen werden zu sich selbst ausgewertet.
- Bei der Auswertung eines n-stelligen Terms f(x1,...,xN) wird zunächst ein Aufruf call(f(Y,x1,...,xN)) für das entsprechende (N+1)-stellige Prädikat abgearbeitet. Nach erfolgreicher Beendigung dieses Aufrufes geht der Wert von Y in die weitere Berechnung des Ausdrucks ein. Falls es keine Klausel für f/(N+1) gibt, wird geprüft, ob es sich bei f/N um eine Standardfunktion oder -operation handelt. Dann wird sie wie bei is/2 ausgewertet, d. h. Auswertung der Argumente und anschließend Berechnung des Funktionswertes. Falls die Auswertung fehlschlägt oder f/N keine Standardfunktion bzw. -operation ist, geht der Aufrufterm f(x1,...,xN) unverändert in die weitere Berechnung ein.

5.6. :=/2

Allgemein bewirkt ein Aufruf T := E, daß der Term E top-down ausgewertet und das Ergebnis T zugewiesen wird: falls es sich bei T um eine Variable handelt, wird das Ergebnis mit dieser Variablen unifiziert; wenn T eine Zahl ist, so wird das Ergebnis mit T verglichen; und wenn es sich bei T um einen anderen Term handelt, wird eine Wertzuweisung an die globale PROLOG-Variable T ausgeführt. Der Aufruf f(x1,...,xN) := E hat allgemein zur Folge, daß nach der Auswertung von E alle Fakten aus der Datenbasis entfernt werden, die N+1-stellig sind, den Funktornamen f haben, deren erstes Argument beliebig und deren nachfolgenden N Argumente mit den Argumenten von f(x1,...,xN) strukturell äquivalent sind. Abschließend wird asserta(f(R,x1,...,xN)) ausgeführt, wobei R das Ergebnis der Auswertung von E sei. Damit realisiert diese Form der Wertzuweisung die Modifikation eines Elementes (x1,...,xN) des globalen Feldes f.
Wir sehen die explizite Nutzung des Assignoperators :=/2 als eine Möglichkeit an, in PROLOG-Programmen zu einer sauberen Trennung zwischen Berechnungszustand und Programmzustand zu gelangen. Der Berechnungszustand wird durch explizite Anwendung des Assign-Operators geändert, während die Datenbasisoperationen der direkten Manipulation des Programmes dienen. Dadurch lassen sich die sonst weit über ein PROLOG-Programm verteilten Datenbasisoperationen besser überschauen.

Beispiel:

Die funktionale Anwendung des oben definierten Prädikats fak/2 läßt sich mit einem Hilfsprädikat erzielen:

```
!(Res,Arg):- X:=Arg,
             integer(X), % Prüfen der Parameter
             X>=0,
             fak(Res,X).
?- op(100,xf,!).
yes
?- a := 3!.
yes
?- X := a!.
X = 720
yes
?- X := (3!)!.
X = 720
yes
?-
```

5.7. Arithmetische Vergleiche (<,=<,=:=,>=,>,=\=)

HU-PROLOG stellt sechs Prädikate zum Vergleich arithmetischer Ausdrücke zur Verfügung. Alle Vergleichsprädikate sind zweistellig und werden als Infixoperator notiert. Nach

op	E1 op E2 ist erfolgreich, wenn
=:=	E1 und E2 gleich sind
=\=	E1 und E2 ungleich sind
>=	E1 größer oder gleich E2 ist
=<	E1 kleiner oder gleich E2 ist
>	E1 größer als E2 ist
<	E1 kleiner als E2 ist

Tab. 8

Aufruf eines Vergleichsprädikates werden beide Terme wie bei :=/2 ausgewertet und numerisch verglichen. Wenn die Ergebnisse der Auswertung keine numerischen Werte sind, führt das zum Abbruch der Abarbeitung:

6. Programmierumgebung

6.1. Ablaufsteuerung

6.1.1.
,/2

Das Prädikat ,/2 beschreibt die sequentielle Abarbeitungsfolge in PROLOG und entspricht damit dem logischen Und, wenn man das deduktive Abarbeitungsmodell unterlegt. Der Operator ,/2 tritt normalerweise nur bei der Konstruktion des Regelkörpers auf. Das separierende Komma zwischen den Argumenten eines Funktorterms ist aber kein Operator. Um die syntaktische Eindeutigkeit zu gewährleisten, ist festgelegt, daß die Argumente eines Funktorterms kleinere Priorität haben müssen als der Kommaoperator. Das heißt, Operatorterme mit größerer Priorität müssen explizit geklammert werden.

6.1.2.
;/2

Das Prädikat ;/2 beschreibt logische Alternativen in der Abarbeitungsfolge. Zunächst werden die links vom Semikolon stehenden Aufrufe abgearbeitet. Sind diese erfolgreich, so ist damit bereits eine Lösung gegeben. Beim Backtracking werden dann zunächst alle Lösungen des linken Zweiges generiert. Erst beim endgültigen Fehlschlagen der linken Seite werden die Aufrufe rechts vom Semikolon abgearbeitet.

6.1.3.
true/0 und fail/0

true/0 ist das PROLOG-Äquivalent der Leeranweisung. Ein Aufruf von true/0 ist stets erfolgreich und hat keinerlei Nebeneffekt. fail/0 ist hingegen ein Prädikat, das stets fehlschlägt, und damit die elementare Aktion zur Auslösung des Backtrackings ist. fail/0 wird häufig in der Kombination "!,fail" benutzt, um das zwangsweise Fehlschlagen einer Prozedur zu bewirken. fail/0 verhält sich wie ein Prädikat, für das es keine Klauseln gibt, nur erzeugt das System keine Warnung und führt keine Sonderbehandlung mit unknown/1 durch.

6.1.4.
repeat/0

repeat/0 ist ein Prädikat, das stets erfolgreich ist und beim Backtracking immer neue Lösungen generiert. Mit repeat,...,test– werden typische iterative Steuerstrukturen in PROLOG realisiert. repeat,...,fail beschreibt einen unendlichen Zyklus, der nur durch Ausnahmebedingungen (wie Unterbrechung oder Programmabbruch) oder durch die Kombination "!,fail" verlassen werden kann.

6.1.5.
!/0

Der Cut-Operator !/0 ist das wichtigste Hilfsmittel zur Steuerung des Backtrackverhaltens von PROLOG-Prädikaten. Durch den Aufruf von "!" innerhalb eines Prädikats werden mögliche alternative Lösungen dieses Prädikats, die aus dem bisherigen Berechnungsprozeß resultieren, „abgeschnitten". Dadurch kann zum Beispiel im Laufzeitkeller der Speicherplatz für die Zustandsvariablen von Aufrufen eingespart werden, die nun über das Backtracking nicht mehr erreichbar sind. Der Cut-Aufruf läßt sich damit zur Effektivierung von PROLOG-Programmen einsetzen, ohne daß sich deren Semantik ändert, wenn die abgeschnittene Lösungsmenge leer ist: eine Eigenschaft, die sich nicht automatisch herleiten läßt. Ein Cut-Aufruf kann aber auch die Semantik eines Prädikats wesentlich prägen. Als letzter Aufruf eines Prädikats angewandt, erzwingt Cut das deterministische Verhalten des Prädikats. In Verbindung mit fail/0 erlaubt der Cut-Aufruf das zwangsweise Fehlschlagen eines Prädikats. Die Prädikate ,/2, ;/2 und –>/2 sind transparent bezüglich Cut-Anwendung, d. h. ein Cut in einem Zweig von ,/2, ;/2 bzw. –>/2 wirkt auf den umgebenden Aufruf.

Beispiel:

Umfangreiche deterministische PROLOG-Berechnungen, die in Instantiierungen der Aufrufparameter resultieren, können sehr viel Speicherplatz erfordern. Wenn die instantiierten Aufrufparameter nach erfolgreicher Beendigung des Prädikats jedoch relativ kleine Terme sind, kann man mit Hilfe des folgenden !/1-Prädikats die Freigabe des Speicherplatzes erzwingen:

```
!(X):–exec(X, Res),Res,asserta(solution(X)),fail.
!(X):–retract(solution(X)),!.
exec(X,true):–X,!.
exec(X,(!,fail)).
```

6.1.6.
–>/2

Das –>/2 Prädikat beschreibt eine eingeschränkte Form der logischen Folgerung. Wenn die Bedingung links vom "–>" erfüllt ist, dann wird die Aktion rechts vom "–>" abgearbeitet, wenn die Bedingung nicht erfüllt ist, schlägt das Prädikat fehl. Im Zusammenhang mit ;/2 ergibt sich eine if-then-else-Konstruktion:

```
(Bedingung –> Then_Zweig ; Else_Zweig)
```

Die Bedingung wird genau einmal ausgewertet, ein Backtrakking vom Then-Zweig in die Bedingung oder in den Else-Zweig ist nicht möglich, in diesem Fall schlägt die Alternative insgesamt fehl.

6.1.7.
call/1

Aufrufe innerhalb eines PROLOG-Prädikats können dyna-

misch während der Abarbeitung erzeugt werden. Dafür gibt es zwei Wege: die Verwendung einer PROLOG-Variablen in der Aufruffolge, die zur Laufzeit mit einem ausführbaren Aufruf instantiiert sein muß, und der Metaaufruf mittels call/1. Der direkte Aufruf einer PROLOG-Variablen ist dem call-by-name in klassischen Sprachen vergleichbar. Der Aufrufterm wird zur Ausführungszeit in dem konkreten Kontext interpretiert und kann die Ablaufsteuerung innerhalb des rufenden Prädikats grundsätzlich beeinflussen, beispielsweise wenn die Variable an "!" oder "(!,fail)" gebunden wird. Der Metaaufruf hat bei normalen Anwendungen die gleiche Semantik, jedoch kann er die Ablaufsteuerung des rufenden Prädikats nicht stören: Ein Metaaufruf kann nur erfolgreich sein oder fehlschlagen. Die Wirkung des Cut wird durch den Metaaufruf „gekapselt".

6.1.8.
not/1 und \+/1

Das Prädikat not/1 beschreibt die Negation in Prolog. +/1 ist synonym zu not/1. Die Negation hat in Prolog allerdings eine etwas unübliche Bedeutung, die sich kurz als Negation-als-Fehler charakterisieren läßt. Die grundlegende Annahme besteht darin, daß dem Prologsystem stets alles über „seine" Welt bekannt ist. Unbekannte Aussagen oder Aussagen, deren Gültigkeit sich nicht herleiten läßt, werden prinzipiell als falsch angesehen und schlagen in der Abarbeitung fehl. Bei einem Aufruf von "not X" wird zunächst versucht, die Gültigkeit des Arguments X zu beweisen. Ist dies erfolgreich, so schlägt "not X" zwangsweise fehl. Wenn der Aufruf von X jedoch fehlschlägt, so wird "not X" erfolgreich.

Semantik:

```
not X:-X, !, fail.
not _.
\+ X:- not X.
```

Beispiel:

Einen Test, ob ein Aufruf erfolgreich ist, ohne die aus diesem Aufruf resultierenden Unifizierungen tatsächlich herbeizuführen, kann man mit Hilfe der doppelten Negation realisieren:

```
?(X):-not not X.
```

6.1.9.
:-/2

Der Operator :-/2 trennt den Kopf vom Körper einer Regel und wird in der Regel nur in Quellfiles sowie als Hauptfunktor in Argumenten von Datenbasisprädikaten angewendet. Mit seiner Hilfe lassen sich PROLOG-Regeln aufbauen und analysieren. Syntaktisch trennt :-/2 stärker als alle anderen Operatoren, daher müssen Teilterme, die diesen Operator enthalten, immer zusätzlich geklammert werden.

6.1.10.
?-/1 und :-/1

Mit ?-/1 werden Aufruffolgen eingeleitet, die vom PROLOG-System z. B. beim Konsultieren eines Quellfiles direkt abzuarbeiten sind. Im interaktiven Modus auf dem Standardtoplevel hat ?-/1 keinen Effekt, da alle Eingaben als Argument von ?-/1 interpretiert werden. Um dies anzudeuten, erscheint die Zeichenfolge "?-" als PROLOG-Prompt. :-/1 kann benutzt werden, um die Ausgabe unerwünschter Variablenbelegungen und Systemantworten (yes/no) sowie Nutzerabfragen auf weitere Lösungen zu unterdrücken. Bei der Stapelverarbeitung ist ?-/1 äquivalent zu :-/1.

6.2.
Globale Steuerung

6.2.1.
toplevel/0

HU-PROLOG besitzt einen eingebauten Toplevel, der die Kommunikation mit dem Nutzer regelt. Dieser ist in der Einleitung zu dieser Dokumentation beschrieben worden. Das Prädikat toplevel/0 erlaubt dem Nutzer, sein eigenes Toplevel zu schreiben. Wenn in der Datenbasis Klauseln mit dem Hauptfunktor toplevel/0 vorhanden sind, werden diese immer dann aufgerufen, wenn der normale Toplevel von PROLOG aktiviert werden würde.
Es folgt eine mögliche Definition eines eigenen Toplevels, der ungefähr dasselbe wie der Standardtoplevel leistet.

```
toplevel :-
    write('?-'), % Prompt ausgeben
    warn(Oldwarn),
    warn(off), % Warnungen ausschalten
    read(Term), % Nutzereingabe einlesen
    warn(Oldwarn),
    (ground(Term),Ground = true;Ground = fail),
    % merken, ob ein ground-Term vorliegt
    ( call(Term) % Aufruf von Term
    ; nl,write('no'),nl,restart % wenn keine Lösung
    ),
    % Lösung ist gefunden
    (Ground ;nl,write(Term),not ask(59)),
    nl,write(yes),nl,!.
```

6.2.2.
abort/0 und restart/0

Das Prädikat abort/0 erzeugt eine Fehlermeldung und beendet die laufende Abarbeitung. Der Interpreter kehrt auf den Toplevel zurück. Das Prädikat restart/0 beendet ebenso die laufende Abarbeitung, erzeugt aber keine Fehlermeldung. Dies ist zum Beispiel bei eigenen Fehlermeldungen sinnvoll.

6.2.3.
exit/1, halt/0 und end/0

Das Prädikat exit/1 führt zum Verlassen des PROLOG-Interpreters. Dabei wird an die aufrufende Shell der exit-Code zurückgegeben, der im Argument von exit/1 angegeben wird. Dieses Argument sollte ein arithmetischer Ausdruck sein, der eine Zahl zwischen 0 und 255 liefert.
Das Prädikat halt/0 ist äquivalent zu exit(0).

Nach Abarbeitung des Prädikats end/0 wird nur noch der aktuelle Aufruf ausgeführt, danach wird der PROLOG-Interpreter mit halt verlassen. end/0 bewirkt also ein verzögertes Verlassen des PROLOG-Interpreters, wenn das Toplevel wieder die Steuerung erhält.

6.2.4.
interrupt/0

Ein während der laufenden Abarbeitung auftretender asynchroner Interrupt (z. B. vom Nutzer durch Drücken der dafür vorgesehenen Taste der Tastatur erzeugt) führt normalerweise zum Abbruch der Abarbeitung und zur Rückkehr auf den Toplevel. Wenn dagegen in der Datenbasis Klauseln mit dem Namen interrupt/0 vorhanden sind, werden diese abgearbeitet. Wenn interrupt/0 erfolgreich abgearbeitet wurde, wird danach die Abarbeitung normal fortgesetzt. Anderenfalls wird ein Backtracking ausgelöst.

Beispiel:

```
interrupt: :-
      ttynl, ttywrite('interrupt:'),ttynl,
      repeat,
      ttywrite('[a]bort/[t]race/[c]ommand :'),
      ttyget(Char),
      ([Char] == "a" , abort
      ;[Char] == "t" , trace(on)
      ;[Char] == "c",
         ttyread(Command),
         call(Command),
         fail
      ),!.
```

6.2.5.
error/2

Die meisten Fehlermeldungen, die zum Abbruch des laufenden Programms führen, können abgefangen werden. Ausgenommen sind Fehlermeldungen wegen Speicherplatzproblemen des Interpreters.
Wenn in der Datenbasis Klauseln mit dem Namen error/2 definiert sind, wird das Prädikat error/2 bei einem Fehler aufgerufen. Als erstes Argument erhält es den Aufruf, bei dem der Fehler auftrat, als zweites die Fehlernummer. Wenn error/2 erfolgreich abgearbeitet wurde, wird danach die Abarbeitung normal fortgesetzt. Anderenfalls wird ein Backtrakking ausgelöst.

Beispiel:

```
error(Call,Errornumber):-
      tell(stderr),
      write('Error'), write(Errornumber),
      write(' in '), write(Call),
      restart.
?- name(X,Y). % ein fehlerhafter Aufruf
Error 2 in name(_1,_2)
yes.
?-
```

6.2.6.
unknown/1

Wenn in PROLOG ein Prädikat abgearbeitet werden soll, das weder ein eingebautes noch ein vom Nutzer definiertes Prädikat ist, schlägt dieser Aufruf normalerweise fehl. Wenn allerdings vom Nutzer das Prädikat unknown/1 definiert ist, wird stattdessen dieses aufgerufen. Als Argument erhält unknown/1 den ursprünglichen Aufruf. Damit kann sich der Nutzer die Reaktion auf nicht vorhandene Prädikate selbst definieren, zum Beispiel ein Konzept des dynamischen Nachladens von Prädikaten.

Beispiel:

```
unknown(Call) :-
      functor(Call,Name,Arity),
      library(LIB),
      ensure(LIB,Name,Arity),
      % es wird versucht, Klauseln nachzuladen
      % siehe Modulkonzept
      !,
      Call.
unknown(Call) :-
      functor(Call,Name,Arity),
      ttywrite('Warning: no clause for relation '),
      ttywrite(Name / Arity),
      ttynl,
      !,fail.
      Library('/user/lib/prolog').
      Library('/y/z/prolib').
```

6.2.7.
ancestors/1

Ein Aufruf von ancestors/1 unifiziert sein Argument mit der Liste der noch aktiven Aufrufe, die vom Top-Level bis zu diesem Aufruf von ancestors/1 geführt haben. Damit ist ancestors/1 ein ausgezeichnetes Hilfsmittel für das dynamische Debugging: man erhält ein retrospektives Trace in Form eines in PROLOG auswertbaren Terms. Daneben ist ancestors/1 aber auch ein Hilfsmittel für den Aufbau der Erklärungskomponente eines Expertensystems, wenn man den Abarbeitungsmechanismus von PROLOG direkt als Inferenzmaschine nutzen will.

6.3.
Das Modul-Konzept von HU-PROLOG

In HU-PROLOG ist ein einfaches, aber wirkungsvolles Modulkonzept implementiert. Die Grundidee besteht darin, die Sichtbarkeit von Atomen explizit zu steuern. Dadurch lassen sich zum Beispiel Prädikate oder bestimmte Termstrukturen lokal zu Quelltextfiles anlegen. Dies ist ein wertvolles Hilfsmittel beim Erstellen größerer Programmsysteme. Seine Hauptanwendung findet das Modulkonzept beim Aufbau von Programmbibliotheken.

6.3.1.
private/1

Das Prädikat private/1 erwartet als Argument entweder ein Atom, oder eine Liste von Atomen. Diese Atome werden als lokal deklariert. Es wird also, wie in klassischen Programmiersprachen ein neues Deklarationsniveau erzeugt. Diese Deklarationsniveaus können verschachtelt werden. Wenn jetzt ein Atom mit einem dieser Namen erzeugt wird (dies geschieht immer bei read/1 und bei name/2), ist es von den vor dem Aufruf von private/1 erzeugten Atomen gleichen Namens verschieden.

6.3.2.
hide/1

Dieses Prädikat erwartet als Argument ein Atom oder eine Liste von Atomen. hide/1 kennzeichnet das Ende des Sichtbarkeitsbereiches dieser Atome. Die bisher sichtbaren angegebenen Atome erhalten einen neuen Namen, und die vorher mit private/1 verdeckten Atome werden wieder sichtbar.
Wenn hide/1 auf ein Atom auf dem obersten Deklarationsniveau angewendet wird, wird auch dieses versteckt. Es ist also möglich, Systemfunktionen umzudeklarieren, indem man das vordefinierte Prädikat mit hide/1 versteckt, und anschließend neu definiert.

Beispiel:

```
% Dieses Modul stellt nach außen das Prädikat
% random/1 zur Verfügung
% Die Zustandsvariable seed ist lokal zu diesem
% Modul.

?-private(seed).   % neues Deklarationsniveau
?-seed := 17.      % Die lokale Zustandsvariable
                   % wird initialisiert
random(X) :-
        seed := (seed * 117 + 29) mod 53 ,
        X := seed.

?- hidden(seed). % Ende des Deklarationsniveaus
```

6.3.3.
ensure/3

Dieses Prädikat ermöglicht das sinnvolle Aufbauen von Bibliotheken, indem es das mehrfache Laden von Files verhindert. Das Prädikat ensure/3 erwartet als Argumente zwei Atome und eine Zahl. Das erste Argument sollte der Name eines Verzeichnisses sein, das zweite der Name eines Atoms und das dritte die Stellenzahl des Atoms. Wenn ensure/3 auf das zweite und dritte Argument bereits einmal erfolgreich angewendet wurde, so ist ensure/3 ohne weiteres erfolgreich. Sonst wird aus den drei Argumenten ein Filename gebildet, der folgende Form hat:
Directory/Prädikatsname.Arity
Wenn dieses File nicht existiert, schlägt ensure/3 fehl. Sonst wird das File mit reconsult/1 eingelesen, und ensure/3 wird wahr.
ensure/3 liest also ein bestimmtes File höchstens einmal ein.

6.4.
Die Schnittstelle zur Systemumgebung

Dieser Abschnitt beschreibt Prädikate, die den Zugriff auf bestimmte Daten der Systemumgebung ermöglichen.

6.4.1.
date/3, time/3 und weekday/1

Diese Prädikate erlauben den (lesenden) Zugriff auf die Systemzeit. Das Prädikat date/3 bindet seine Argumente an das Jahr, den Monat und den aktuellen Tag. Das Prädikat time/3 bindet seine Argumente an die aktuelle Stunde, Minute und Sekunde. Das Prädikat weekday/1 bindet sein Argument an die Nummer des aktuellen Wochentages.

6.4.2.
timer/1

timer/1 bietet eine Möglichkeit zur Messung der dem Interpreter vom Betriebssystem zugeteilten Rechenzeit. Der Timer zählt diese Zeit in 1/100 Sekunden. Wird timer/1 mit einer Zahl aufgerufen, so wird der Zähler auf diese Zahl gesetzt. Wird timer/1 mit einer Variablen aufgerufen, so wird diese an den aktuellen Wert von Timer gebunden.

Beispiel:

```
?- timer(0),testprogramm,timer(Time).
Time = 126
yes
?-
```

Das Prädikat testprogramm/0 hat zur Abarbeitung 1.26 Sekunden gebraucht.

6.4.3.
getenv/2

Dieses Prädikat erlaubt den Zugriff zu Umgebungsvariablen. Das zweite Argument muß der Name der Variable sein, das erste Argument wird mit ihrem Wert unifiziert.

Beispiel:

```
?- getenv(Homedirectory, 'HOME').
Homedirectory = /y/prolog/techn
yes
?- getenv(Editor, 'EDITOR').
Editor = vi
yes
?-
```

6.4.4.
system/1

Das Prädikat system/1 führt das als Argument angegebene Programm aus. Das Argument muß eine Liste von Atomen sein. Das erste Element ist der Name des Programms, die weiteren sind seine Argumente.

Beispiel:

```
% Aufruf des durch die Environment Variable 'EDITOR'
```

```
% spezifizierten Editors (z. B. vi) und anschließendes
% Laden des Files.
% Beim Aufruf ohne Argument wird das zuletzt editierte
% File noch einmal editiert.
edit(Filename) :-
      file:='Filename,
      gentenv(Editor,'EDITOR'),
      system([Editor, Filename ]),
      reconsult( Filename ).
edit :- Filename=file, edit(Filename).
```

6.5. Flags

Die Arbeitsweise von HU-PROLOG wird über die Flags echo, warn, log und ocheck gesteuert. Jedes dieser Flags kann die Zustände on(gesetzt) oder off(nicht gesetzt) annehmen. Mit Hilfe der gleichnamigen Prädikate echo/1, warn/1, log/1, ocheck/1 können die Zustände der Flags geändert und abgefragt werden. Werden die Prädikate mit dem Argument on oder off aufgerufen, so wird das entsprechende Flag auf diesen Wert gesetzt. Ist das Argument eine freie Variable, so wird diese mit dem aktuellen Zustand des entsprechenden Flags unifiziert. Jedes andere Argument führt zu einem Fehler. Jedes Flag hat bei Interpreterstart eine Standardeinstellung (siehe unten).
Mit dem echo-Flag wird die Art und Weise des Einlesens von Termen gesteuert. Ist das echo-Flag nicht gesetzt, so werden alle Terme nur gelesen und interpretiert. Falls das Flag gesetzt ist, werden alle gelesenen Terme vor der Interpretation nochmals auf die Standardausgabe ausgegeben. Dieser Mechanismus ist nicht auf das Termlesen im Toplevel beschränkt, sondern erstreckt sich auf alle Eingabeoperationen. Die Standardeinstellung ist off.
Mit dem log-Flag wird die Erzeugung des Ein-/Ausgabeprotokolls gesteuert. Es gibt in HU-PROLOG die Möglichkeit, alle Ein- und Ausgaben des Interpreters in einer Datei zu protokollieren. Dazu muß beim Aufruf des Interpreters der Filename der Protokolldatei spezifiziert werden (siehe unten). Wenn das log-Flag gesetzt ist, wird in der spezifizierten Datei protokolliert, wenn es nicht gesetzt ist, nicht. Die Standardeinstellung ist off. Wird das log-Flag gesetzt und vorher keine Protokolldatei spezifiziert, wird als Standard /dev/null angenommen. log/0 ist eine Abkürzung für log(on) und nolog/0 eine Abkürzung für log(off).
Mit dem Warnflag wird die Ausgabe von Warnungen aus- bzw. eingestellt. Die Standardeinstellung ist on.
Mit dem ocheck-Flag wird der Unifikationsmechanismus gesteuert. Wenn das Flag gesetzt ist, sind Variablen nur noch mit Termen unifzierbar, in denen sie nicht vorkommen. Diese Arbeitsweise entspricht der korrekten Definition von 'Unifizierbarkeit' und verhindert das Entstehen zyklischer Terme. Ist das ocheck-Flag nicht gesetzt, so sind Variablen mit beliebigen Termen unifizierbar. Die Unifikation arbeitet bei gesetztem ocheck-Flag langsamer als bei nicht gesetztem. Die Standardeinstellung ist off.

6.6. Testunterstützung

Zur Unterstützung der Programmentwicklung mit dem System stellt der Interpreter zwei Testhilfen zur Verfügung. Mit Hilfe der Prädikatfamilie trace/spy ist es möglich, den Ablauf der Interpretation vollständig bzw. selektiv zu verfolgen.

6.6.1. trace/0, trace/1 und notrace/0

Das Verfolgen des Ablaufes mittels trace wird über ein Flag gesteuert. Das trace-Flag wird analog oben beschriebener Flags genutzt. Zusätzlich setzt das Prädikat trace/0 das Flag auf on, während das Prädikat notrace/0 es zurücksetzt. Die Standardeinstellung ist off.
Wenn der Interpreter mit gesetztem trace-Flag betrieben wird, so erzeugt er beim Aufruf und beim Verlassen jedes Prädikates eine Ausschrift auf der Standardausgabe. Es gibt für jede Form des Betretens bzw. Verlassens eines Prädikates eine Ausschrift. In den folgenden Darstellungen bedeutet term jeweils das Prädikat und n die Verschachtelungstiefe, in der sich der Interpreter befindet. Wird ein Prädikat aufgerufen, so erscheint die Ausschrift:

```
(n) CALL: term        [fsan?\n]
```

Wird ein Prädikat während des Backtrackings erneut aufgerufen, so erscheint die Ausschrift:

```
(n) REDO: term        [fsan?\n]
```

In beiden Fällen erwartet der Interpreter danach eine Eingabe. Diese kann aus den Zeichen sanft? bestehen. Wird ein Zeichen außer sanft?, gefolgt von einem ⟨ET⟩ oder nur ein ⟨ET⟩ eingegeben, so wird das Prädikat term aufgerufen.
Die Funktion der Zeichen sanft?:

[f]ail

Der Interpreter löst an dieser Stelle Backtracking aus.

[s]kip

Das Prädikat term wird aufgerufen, jedoch dabei die Ablaufverfolgung ausgeschaltet. Als nächste Testausschrift erscheint die Information über das Verlassen des Prädikates term.

[a]bort

Die Interpretation wird abgebrochen und der Interpreter kehrt in das Toplevel zurück.

[t]race
[n]otrace

Das trace-Flag wird on bzw. off gesetzt und das Prädikat term aufgerufen.

?

Es wird eine Hilfesinformation über die Funktion der Testoptionen ausgegeben.
Beim erfolgreichen Verlassen eines Prädikates erscheint die Ausschrift:

(n) EXIT: term

Beim Fehlschlagen eines Prädikates erscheint die Ausschrift:

(n) FAIL: term

6.6.2.
spy/1 und nospy/1

Mit dem spy-Mechanismus ist es möglich, die Menge der Ausschriften, die bei gesetztem trace-Flag erzeugt werden, erheblich zu reduzieren. Mittels spy/1 werden einzelne Prädikate ausgezeichnet, für die trace-Ausschriften erzeugt werden sollen. spy/1 erwartet als Argument ein Atom oder Atom/Stellenzahl. Es zeichnet das Prädikat mit dem entsprechenden Namen und der entsprechenden Stellenzahl aus. Falls die Stellenzahl nicht angegeben wird, so werden alle Prädikate mit dem entsprechenden Namen ausgezeichnet. spy(all) zeichnet alle Prädikate aus, die dem Interpreter zur Zeit der Abarbeitung von spy bekannt sind. nospy/1 nimmt die Auszeichnung von Prädikaten wieder zurück. Es erwartet als Argumente dieselben wie spy/1 und wirkt auf dieselben Prädikate. Die trace- und spy-Möglichkeiten stehen separat nebeneinander und beeinflussen sich gegenseitig nicht. Jeder Testausschrift für ein Prädikat, das ausgezeichnet ist, wird jedoch ein * vorangestellt.

6.7.
Entwicklungsumgebung von HU-PROLOG

6.7.1.
consult/1 und reconsult/1

Durch einen Aufruf von consult(Filename) wird das File mit dem Namen Filename eröffnet und eingelesen. Klauseln in diesem File werden mit assertz/1 in die Datenbasis eingefügt. Anfragen mit ?-/1 werden abgearbeitet. reconsult(Filename) arbeitet analog zu consult/1, löscht aber Prädikate, die in dem File vorkommen, vorher aus der Datenbasis. Dabei gibt es folgende Vereinfachungen: consult(- Filename) ist äquivalent zu reconsult(Filename). Des weiteren gibt es die sogenannten Filelisten. Wenn eine Liste als Prädikate aufgerufen wird, ist dies äquivalent zu einem consult/1 aller ihrer Elemente.

6.7.2.
save/1

Das Prädikat save/1 speichert den aktuellen Stand der Datenbasis in dem als Argument von save/1 angegebenen File ab. Dieses File hat ein Format, das es erlaubt, es später als ausführbares Programm aufzurufen. Der so gestartete PROLOG-Interpreter befindet sich in dem Zustand, der bei der Abarbeitung von save/1 vorlag – bis auf folgende Änderungen:

– Es sind nur die Files stdin, stdout und stderr eröffnet.
– Der Interpreter befindet sich auf dem Toplevel.

Mit diesem Prädikat wird dem Nutzer die Möglichkeit gegeben, fertige PROLOG-Anwendungen kompakt abzuspeichern. Dies ermöglicht den schnellen Start einer solchen Anwendung, ohne explizit PROLOG aufrufen und die Datenbasis einladen zu müssen.

6.8.
stats/0

Das Prädikat stats/0 ermittelt die Speicherauslastung des PROLOG-Interpreters. In einer kurzen Nachricht auf der Standardausgabe werden die Auslastungen der einzelnen Speicherbereiche ausgegeben.
Der Interpreter verwaltet fünf Speicherbereiche. Terme in werden im node-Speicher abgelegt. Atome, Variablen und Integerzahlen belegen genau ein node-Element. Strukturierte Terme belegen die node-Elemente, die für die Argumente nötig sind, zuzüglich eines Elements für den Hauptfunktor. Klauseln werden als Terme abgespeichert, denen noch ein Element zugeordnet ist, welches Verwaltungsinformationen enthält. Atome benötigen außer einem node-Element noch ein atom-Element und char-Elemente für die einzelnen Zeichen. Im trail-Speicher werden alle Variablen vermerkt, die während des Backtrackings wieder freigegeben werden müssen. Im environment-Speicher werden Verweise auf alle aktiven Klauseln vermerkt.
Die node-, atom- und char-Speicher untergliedern sich jeweils in zwei Bereiche. Im heap-Bereich werden alle Elemente abgespeichert, die für Terme der Datenbasis benötigt werden. Im stack-Bereich sind alle dynamisch erzeugten Terme abgespeichert, die beim Backtracking wieder freigegeben werden.
Die Ausgabe von stats/0 schlüsselt die Auslastung jedes einzelnen Speicherbereiches auf und gibt zusätzlich deren maximale Größe und die prozentuale Auslastung aus.

6.9.
Systemstart

Mit Hilfe der Kommandozeilenparameter (Optionen) -e, -w, -o, -l, -L können die Standardeinstellungen der Flags beim Systemstart geändert werden. Die Optionen -e, -w und -o haben folgende Auswirkungen auf die Standardeinstellungen der Flags:

-e : echo-Flag = on
-w : warn-Flag = off
-o : ocheck-Flag = on

Die Optionen -l und -L dienen der Angabe des Dateinamens für das Protokollfile. Der Dateiname muß dabei als folgendes Kommandozeilenargument angegeben werden. Die Option -l läßt dabei die Standardeinstellungen der Flags unverändert, während die Option -L das log-Flag setzt. Die Option -s, gefolgt von einem Dateinamen spezifiziert eine PROLOG-Quelltextdatei, die vor dem Interpreterstart eingelesen und interpretiert wird. Alle Prädikate, die in dieser Datei definiert sind, erhalten den Status von Systemprädikaten. Existiert beim Interpreterstart im aktuellen Directory oder im Homedirectory eine Datei/.prologrc und wird die -s Option nicht benutzt, so wird diese Datei als Systemstartdatei interpretiert.

7. Beispiele

7.1. Fibonacci-Zahlen

Das Prädikat fibo/2 dient der Berechnung der n-ten Zahl aus der Folge 0,1,1,2,3,5,8,... der Fibonaccizahlen, die definiert ist durch:

```
fibo(n)  = 0                                , falls n = 0
         = 1                                , falls n = 1
         = fib(n – 2) + fib(n – 1), sonst
```

```
% fibo(F,N) unifiziert F mit der N-ten Fibonaccizahl

fibo(0,0).          % 0. Fibonaccizahl ist 0
fibo(1,1).          % 1. Fibonaccizahl ist 1
fibo(F,N) :–
        N1 is N – 1, N2 is N – 2, % N – 1 und N – 2 berechnen
        fibo(F1,N1), fibo(F2,N2),
        F is F1 + F2.

?-fibo(X,3).
X = 2
yes
?-fibo(X,10).
X = 55
yes
?-fibo(X,12).
X = 144
yes
```

7.2. Taylorreihenentwicklung

Im folgenden Beispiel soll die Taylorreihe für exp(x) bis zum n-ten Glied berechnet werden. Die Taylorreihe dient der beliebig genauen Approximation des Funktionswertes einer Funktion. Die Genauigkeit nimmt mit wachsendem n (Nummer des Abbruchgliedes) zu.

```
% f(x,n) = 1 + x/1! + x^2/2! + x^3/3! + ... + x^n/n!

exp(Y,X,Epsilon) :–
     accu := 1, index := 1, pf := X,      % Initialisierung
     repeat,
          accu := accu + pf,              % Reihe entwickeln
          index := index + 1,             % Zähler erhöhen
          pf := pf • X/index,             % pot/fak erhöhen
     pf =< Epsilon,                       % Test, ob Genauigkeit erreicht
     Y := accu.                           % Ergebnisübergabe im 1. Argument
```

7.3. Polynome und Horner-Schema

Mit Hilfe globaler Variablenfelder kann eine Polynombehandlung in PROLOG realisiert werden. Ein Polynom der Form

f(x) = x^3 + 17 • x^2 + 13 • x^1 + 1

kann folgendermaßen abgespeichert werden:

```
?– poly(grad) := 3,poly(3) := 1, poly(2) := 17,
     poly(1) := 13, poly(0) := 1.
yes
?– listing(poly).
poly(1,0).
poly(13,1).
poly(17,2).
poly(1,3).
poly(3,grad).
yes
?–
```

Das Prädikat poly_value(V,X) berechnet den Wert des Polynoms, das in der einstelligen Variablen poly gespeichert ist, an der Stelle X. Dabei wird das 'Horner-Schema' genutzt:

```
poly_value(V,X) :–
          accu := 0, index := poly(grad),
          repeat,
               I := index,index := index – 1,
               accu := (accu + poly(I)) • X,
          index < 0,
          V := accu.
```

7.4. Schwachbesetzte Matrizen

Das Konzept der strukturierten Variablen erlaubt beispielsweise die Realisierung großer, sparsam besetzter Matrizen:

```
?– unit(_,_) := 0,                 % alle Elemente sind 0
     unit(X,X) := 1,                  % alle Diagonalelemente sind 1
unit(12,44) := 13,                 % Element 12,44 hat den Wert 13
unit(12345,6493) := 55.            % Element 12345,6493 hat den Wert 55
yes
?– listing(unit).
unit(55,12345,6493).
unit(13,12,44).
unit(1,X,X).
unit(0,_,_).
yes
?–
```

Das folgende Beispiel zeigt den Zugriff auf die oben erzeugte Matrix.

```
?– trace(on).
?– X := unit(3,7).
 (4)     CALL:     _1 := unit(3,7) [f]ail/[s]kip/[a]bort?
 (8)     CALL:     call(unit(_1, 3, 7))[f]ail/[s]kip/[a]bort?
(10)     CALL:     unit(_1, 3, 7) [f]ail/[s]kip/[a]bort?
(10)     EXIT:     unit(0, 3, 7)
 (8)     EXIT:     call(unit(0, 3, 7))
 (4)     EXIT:     0 := unit(3, 7)
x = 0

yes
?– X := unit(12,44).
```

```
 (4)   CALL:  _1:= unit(12, 44) [f]ail/[s]kip/[a]bort?
 (8)   CALL:  call(unit(_1, 12, 44)) [f]ail/[s]kip/[a]bort?
(10)   CALL:  unit(_1, 12, 44) [f]ail/[s]kip/[a]bort?
(10)          unit(13, 12, 44)
 (8)          call(unit(13, 12, 44))
 (4)          13:= unit(12, 44)

X = 13
yes
?-
```

7.5. Repetitionskommando

Das 'multiexecution'-Prädikat X • C führt den Aufruf C so oft aus, wie es durch den arithmetischen Ausdruck X gefordert wird. Die iterative Zyklusorganisation hat gegenüber der rekursiven den Vorteil, daß bei großen Durchlaufzahlen keine Speicherplatzprobleme entstehen.

```
X • C:=
        Y:= X,integer(Y),Y >= 0,  % falls Y nicht positiv & ganz
                                  % → fail
        n:= 1,          % Initialisieren
        repeat,
           n:= n + 1,   % Zähler erhöhen
           call(C),     % C aufrufen
        n > Y.          % fertig, wenn n > Y
?- 20•write('-'), nl.
yes
```

7.6. Dienstprogramm copy

```
% copy(Inputfile, Outputfile)        -- Kopieren eines Files
copy(Inputfile, Outputfile):=
     seeing(OldInFile),               % Filezuordnung der
     telling(OldOutFile),             % E/A-Ströme retten
     see(Inputfile),                  % von Inputfile lesen
     tell(Outputfile),                % auf Outputfile schreiben
     repeat,
         get0(Char),                  % Lese ein zeichen
         (
             Char == -1               % Fileende
         ;
             put(Char)                % Zeichen wieder ausgeben
             fail                     % und weiter bei repeat
         ),
     seen,   % Eingabefile schlie-
             ßen
     told,   % Ausgabefile
             schließen
     see(OldInFile),                  % Rücksetzen der
     tell(OldOutFile),                % E/A-Ströme
     !.                               % Prädikat deterministisch
% cat(File) -- Ausgabe eines Files auf Bildschirm
cat(File) :- copy(File,stdout).
```

7.7. Dienstprogramm more

Das Programm more gibt ein File seitenweise auf den Bildschirm aus. Nach jeweils einer Seite wartet das System auf das Drücken der Enter-Taste.

```
more(File) :-
     seeing(OldInFile),
     see(File),
     linecounter:= 1,          % globaler Zeilenzähler
     repeat,
          get0(Char),
          (    Char >= 0,
               put(Char),
               Char == 10,
               linecounter:= linecounter + 1,
               linecounter > 22,
               linecounter:= 1,
               info_user,
               fail
          ;
               Char == -1          % eof
          ),
     seen,
     see(OldInFile).
info_user:=
     ttygotoxy(0,23),
     ttyput("Please press ET"),
     ttyskip(10),
     ttygotoxy(0,22),
     tab(15),
     ttygotoxy(0,22),
     fail.
```

Interessenten wenden sich bitte an

Dr. Christian Horn, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Telefon 20 93 22 15.

Computerliteratur aus dem Verlag Die Wirtschaft Berlin

Die Arbeitsplatzcomputer A 7100 und A 7150

Von Dr.-Ing. Eckehart Stamer und Gerhard Ziese
Herausgeber: Kombinat Robotron
192 Seiten, Broschur, 12,00 M
Bestellangaben: ISBN 3-349-00306-0
676 146 2 / A 7100/A 7150

Betriebssystem SCP für Personalcomputer

Von Doz. Dr.-Ing., Dr. sc. nat. Kuno Schmidt
Herausgeber: Kombinat Robotron
224 Seiten, Broschur, 14,00 M
Bestellangaben: ISBN 3-349-00255-2
676 086 5 / Betriebssystem SCP/PC

Der Personalcomputer 1715

Von einem Autorenkollektiv
Leitung: Prof. Dr.-Ing. Rolf Zeth
Herausgeber: Kombinat Robotron
2. Auflage
224 Seiten, Broschur, 14,00 M
Bestellangaben: ISBN 3-349-00231-5
676 061 1 / PC 1715

Datenbanken mit Personalcomputern

Von Dipl.-Ing. Ursula Hempel und
Dipl.-Ing. Hans Loley
Herausgeber: Kombinat Robotron
2. Auflage
160 Seiten, Broschur, 10,00 M
Bestellangaben: ISBN 3-349-00431-8
676 250 2 / Hempel, Datenbanken

Der Personalcomputer EC 1834

Von einem Autorenkollektiv
Leitung Prof. Dr.-Ing. Rolf Zeth
Herausgeber: Kombinat Robotron
350 Seiten, Broschur, 22,00 M
Bestellangaben: ISBN 3-349-00574-8
676 392 7 / EC 1834

Verlag Die Wirtschaft
Am Friedrichshain 22, Berlin, 1055

Programmieren mit PROLOG – einige Beispiele

Tom Bihr
Humboldt-Universität zu Berlin, Sektion Mathematik

Der vorliegende Artikel ist geschrieben für all diejenigen, die sich noch nicht mit der Programmiersprache PROLOG befaßt, jedoch Erfahrungen mit prozeduralen Programmiersprachen wie PASCAL, Modula oder C haben. An kleinen, kommentierten Beispielen aus verschiedenen Bereichen kann man sich in die Philosophie dieser Sprache hineindenken und erste Schritte unternehmen.

Zwei kleine numerische Beispiele

① Die Summe der ersten n natürlichen Zahlen läßt sich auf zwei Wegen berechnen. Man kann die Summenformel benutzen und berechnet n•(n + 1)/2, oder man summiert schrittweise auf. Der zweite Fall ist durch das Prädikat 's' dargestellt. Sofort zu sehen ist, daß die Abbruchbedingung s(0,0) nur dann wirksam werden kann, wenn für n ein erlaubter Wert (d. h. eine nichtnegative ganze Zahl) eingesetzt wurde. Um das abzusichern, ist das Prädikat 'sum' angegeben, das den Eingabewert für N auf Gültigkeit testet und dann erst die Berechnung der Summe vornimmt.

```
s(0,0).
s(N,S) :- N1 is N - 1,
          s(N1,S1), S is S1 + N.
sum(N,S) :- integer(N),
            N >= 0, s(N,S).
```

Beispiel:

```
?- sum(9,X).
X = 45.
```

② Den größten gemeinsamen Teiler (ggT) zweier natürlicher Zahlen a und b (mit a > 0 und b > 0) kann man nach dem euklidischen Algorithmus wie folgt bestimmen: Wenn b Teiler von a ist, dann ist b auch der größte gemeinsame Teiler von a und b. Sonst gilt die Beziehung ggt(a,b) = ggt(b,a mod b). Diese Vorschrift läßt sich sofort in ein PROLOG-Prädikat 'ggt' umsetzen (Man lese ggt(A,B,E) als „der größte gemeinsame Teiler von A und B ist E"). Analog zu den Prädikaten 's' und 'sum' ist zu 'ggt' noch das Prädikat 'ggT' angegeben, das die Eingabewerte vor der Berechnung auf ihre Zuverlässigkeit hin prüft. Noch eine Bemerkung zum Operator '=:=': Er ist in seiner Wirkung identisch zum '='-Operator für den Vergleich numerischer Ausdrücke in PASCAL. Es werden zunächst die Werte der numerischen Ausdrücke links und rechts vom Operator berechnet und anschließend verglichen.

```
ggt(A,B,B)   :- A mod B =:= 0.
ggt(A,B,G)   :- C is A mod B,
                ggt(B,C,G).
ggT(A,B,G)   :- integer(A),
                integer(B), A > 0, B > 0,
                ggt(A,B,G).
```

Beispiele:

```
?- ggT(456,258,T).
T = 6
?- ggT(19,7,X).
X = 1.
```

Listenmanipulation

Mit der Liste steht in PROLOG eine Struktur zur Verfügung, mit der Objekte variabler Anzahl zusammengefaßt werden können. Für Listenausdrücke gibt es verschiedene Notationsformen. In den nachfolgenden Beispielen erfolgt allerdings eine Beschränkung auf eine Form. Die in einer Liste enthaltenen Objekte (zulässig ist jeder PROLOG-Term) werden Elemente der Liste genannt. Beispiele für Listen sind [a, b], [feld(mitte,gruen), hallo, 3 + 6] oder [] (die leere Liste). Für Listen ist ein Operator '|' – der Listenseparator – definiert. Er ermöglicht die Trennung des Kopfes einer Liste vom Listenrest.

Beispielsweise führen die PROLOG-Fragen
?- [a,b,c] = [X|Y].
zur Bindung von X an a und von Y an [b,c] sowie
?- [A,B,C|R] = [3,4,7,hallo,3 + 4]
zur Bindung von A an 3, B an 4, C an 7 und R an [hallo, 3 + 4].

Nun zu einigen Operationen mit Listen:

■ Das Prädikat 'member' prüft, ob in einer Liste L ein Element X vorhanden ist oder nicht. Die Klauseln sind dabei leicht zu interpretieren: Wenn X das erste Element der Liste ist, so ist es enthalten. Sonst kann X höchstens noch in der Restliste enthalten sein.

```
member(X,[X|_]).
member(X,[_|R]) :- member(X,R).
```

Beispiele:

```
?- member(a,[a,b,c,d]).
yes
?- member(3 + 4,[4,7,9,3 + 6]).
no.
```

■ Das Prädikat 'countel' bestimmt die Anzahl der Elemente einer Liste. Ist sie leer, so ergibt sich die Anzahl zu Null. Enthält die Liste Elemente, so wird zunächst die Anzahl der Elemente der Restliste bestimmt und dann 1 addiert.

```
countel([],0).
countel([_|R],N) :- countel(R,N1),
                    N is N1 + 1.
```

Beispiele:

```
?- countel([7,hallo,9],3).
yes
?- countel([rot, gruen, gelb, blau],X).
X = 4.
```

■ Um aus einer Liste L ein Element X zu streichen, bedient man sich des Prädikats 'select'. Die zwei Klauseln zeigen, welche Fälle beachtet werden müssen. Ist X das erste Element der Liste, so bildet die Restliste das Ergebnis der Operation. Ansonsten separiert man Kopf und Rest der Liste, streicht X aus der Restliste, erhält R1 als Ergebnis und fügt an diese Ergebnisliste den Kopf wieder an.

```
select(X,[X|R],R).
select(X,[K|R],[K|R1]) :- select(X,R,R1).
```

Beispiel:

```
?- select(a,[1,z,2,y,3,a,4,b],X).
X = [1,z,2,y,3,4,b].
```

■ Falls man innerhalb von Listen eine bestimmte Ordnung der Elemente vornimmt, kann es vorkommen, daß man ein Element X vor einem Element Y in eine Liste einfügen möchte. Das Verfahren für 'insert' gleicht dem von 'select'. Die zusätzliche dritte Klausel ist nicht unbedingt erforderlich. Für den Fall, daß Y nicht Element der Liste ist, sorgt sie dafür, daß X an die Liste angehängt wird.

```
insert(X,Y,[Y|R],[X,Y|R]).
```

```
insert(X,Y,[K|R],[K|R1]) :-
    insert(X,Y,R,R1).
insert(X,_,[],[X]).
```

Beispiele:

```
?- insert(80386,80486,
[8086,80286,80486],L).
L = [8086,80286,80386,80486]
?- insert(c,d,[a,b],L).
L = [a,b,c].
```

■ Bei einigen Problemen (insbesondere bei Sortiervorgängen) sind mehrfach auftretende Listenelemente nicht sinnvoll bzw. erwünscht. Das Prädikat 'nodouble' streicht aus einer Liste alle mehrfach vorkommenden Elemente bis auf ein einziges. Das Verfahren funktioniert wie folgt:
Auf die leere Liste angewendet, erhält man die leere Liste. Sonst separiere man Kopf und Rest einer Liste. Falls der Kopf in der Restliste enthalten ist (member(K,R)), wende man das Verfahren auf die Restliste an. Das Ergebnis dieser Operation ist zu übernehmen. Sonst – also wenn der Kopf nicht in der Restliste enthalten ist – wende man das Verfahren auf die Restliste an und stelle dem dabei erhaltenen Ergebnis den Kopf voran.

```
nodouble([],[]).
nodouble([K|R],R1) :-
    member(K,R), nodouble(R,R1).
nodouble([K|R],[K|R1]) :-
    nodouble(R,R1).
```

Beispiel:

```
?- nodouble([rot, rot, gelb, rot,
gelb, gruen],X).
X = [rot, gelb, gruen].
```

■ Ein Prädikat zum Verketten zweier Listen soll die Vorstellung von *reinen* Listenoperationen abschließen. Der Weg zum Verständnis des Prädikats 'append' öffnet sich über eine geeignete Interpretation, z. B.: *Man verkettet zwei Listen, indem man der zweiten die erste Liste voranstellt.* Damit werden die beiden Klauseln lesbar. Die erste behandelt den Fall, daß eine leere Liste einer zweiten vorangestellt wird. Das Ergebnis ist mit der zweiten Liste identisch. Die zweite Klausel gibt die Vorschrift für den Fall an, daß die erste Liste Elemente enthält. Dann ist der Kopf abzutrennen, die Restliste der zweiten Liste und dam dabei erhaltenen Ergebnis noch der Kopf voranzustellen.

```
append([],L,L).
append([K|R],L,[K|Y]) :-
    append(R,L,Y).
```

Beispiel:

```
?- append([f, c],[m],L).
L = [f, c, m].
```

Sortieren mit Listen

Eine Anwendung der Listenmanipulation kann das Sortieren einer Sequenz von Objekten sein. Im folgenden werden drei Beispiele für das Sortierverfahren Bubblesort und ein Beispiel für Quicksort angeben. Zunächst aber einige Bemerkungen zur Ordnung von Termen in PROLOG. Bezüglich der Ordnung werden vier Arten von Termen unterschieden: Variablen, Zahlen, Atome und Strukturen. Die Reihenfolge gibt dabei schon die Ordnung zwischen den verschiedenen Termarten an: Variablen sind immer kleiner als Zahlen, diese immer kleiner als Atome und diese immer kleiner als Strukturen. Von zwei Variablen ist diejenige kleiner, die dem System eher bekannt war. Bei Zahlen gilt die naürliche und bei Atomen die alphabetische Ordnung. Bei Strukturen entscheidet die Stelligkeit (damit ist feld(3,7) größer als a(3)). Ist die Stellenzahl gleich, entscheidet die alphabetische Ordnung des Funktors (damit ist append([a,b],[c],L) kleiner als select(a,[b,c,a],L). Sollten Funktoren und Stellenzahl übereinstimmen, so entscheidet das erste unterschiedliche Argumentepaar über die Ordnung (damit ist member(a,[a,b,c]) kleiner als member (b,[a,b,c])).
Zum Termvergleich existieren die Operatoren '@<' (kleiner), '@=<' (kleiner oder gleich), '@>=' (größer oder gleich), '@>' (größer), '==' (gleich) und '\==' (ungleich).

■ Nun zum Sortierverfahren Bubblesort. Der dem Verfahren zugrundeliegende Algorithmus ist recht einfach. Man untersucht innerhalb der Sequenz von Objekten nacheinander jeweils zwei benachbarte. Stehen sie nicht in der Ordnungsrelation (d. h. bei angestrebter aufsteigender Ordnung ist das erste Objekt größer als das zweite), so werden sie vertauscht. Dieses Verfahren muß bei einer Sequenz von n Objekten höchstens $n-1$ mal ausgeführt werden, um die geordnete Sequenz zu erhalten. Im vorliegenden Beispielprogramm übernimmt das Prädikat 'steige' die Aufgabe des einmaligen Durchsuchens der Liste und des eventuellen Tauschens von benachbarten Elementen. Das dreistellige Prädikat 'bubblesort' kontrolliert die Anzahl der steige-Operationen. Es wird seinerseits gerufen vom zweistelligen 'bubblesort', das zunächst die Anzahl N der Listenelemente bestimmt und dann das Sortieren auslöst.

```
bubblesort([],[]).
bubblesort(L,Lsort) :-
    countel(L,N),bubblesort(N,L,Lsort).
bubblesort(1,L,L).
bubblesort(N,L,Lsort) :-
    steige(L,L1), N1 is N - 1,
    bubblesort(N1,L1,Lsort).
steige([E],[E]).
steige([A,B|R],[B|Y]) :- A @> B,
    steige([A|R],Y).
steige([A,B|R],[A|Y]) :-steige([B|R],Y).
```

Beispiel:

```
?- bubblesort([1,5,3,6,0,2],L).
L = [0, 1, 2, 3, 5, 6].
```

Bei genauer Betrachtung entdeckt man, daß die angegebene Version noch recht uneffektiv ist. Nach der ersten 'steige'-Operation hat nämlich das größte Element bereits seinen Platz eingenommen und muß nicht mehr untersucht werden. Die zweite 'steige'-Operation bringt das zweitgrößte Element an seinen Platz usw. Für 'bubblesort' interessant sind also beim ersten Durchlauf alle n Listenelemente, beim zweiten Durchlauf die ersten $n-1$ und beim letzten Durchlauf nur die ersten beiden. Die Version 'bubblesort1' berücksichtigt das durch Veränderung des 'steige'-Prädikats (hier 'steige1'). Es erhält zusätzlich über die (von N beginnend absteigende) Nummer des Durchlaufs die Information über die Anzahl der noch zu betrachtenden Elemente.

```
bubblesort1([],[]).
bubblesort1(L,Lsort) :-
    countel(L,N),
    bubblesort1(N,L,Lsort).
bubblesort1(1,L,L).
bubblesort1(N,L,E) :-
    steige1(N,L,L1), N1 is N – 1,
    bubblesort1(N1,L1,E).
steige1(1,L,L).
steige1(N,[A,B|R],[B|E]) :-
    A @> B, N1 is N – 1,
    steige1(N1,[A|R],E).
steige1(N,[A,B|R],[A|E]) :-
    N1 is N – 1, steige1(N1,[B|R],E).
```

Die beiden ersten vorgestellten Versionen für das Sortieren nach Bubblesort haben ihren Ursprung in den *klassischen,* d. h. prozeduralen Programmiersprachen wie PASCAL. PROLOG ermöglicht eine weitere – vielleicht etwas ungewöhnliche – Version. Dazu nochmals ein Blick auf das Prädikat 'append'. Wie fast alle Prädikate kann auch 'append' auf verschiedene Weise genutzt werden. Es ermöglicht nicht nur das Verketten von Listen, sondern auch das Aufspalten in Teillisten oder das Bilden einer *Listendifferenz.* Einige Beispiele sollen das erläutern:

```
?- append([a,b,c],[d,e,f],L).
L = [a,b,c,d,e,f]
?- append(X,Y,[a,b]).
X = []
Y = [a,b];
X = [a]
Y = [b];
X = [a,b]
Y = [];
no
?- append(X,[3,4|R],[1,2,3,4,5]).
X = [1,2]
R = [5].
```

Die Möglichkeit, Teillisten zu generieren, wird für die folgende Bubblesortversion ausgenutzt. Man zerlegt die Anfangsliste L in eine (geordnete) Teilliste L1 und eine zweite Liste, deren ersten beiden Elemente nicht in der richtigen Reihenfolge stehen. Durch Vertauschen von A und B und Verketten der beiden Teillisten erhält man eine neue, weiterzubearbeitende Liste. Falls keine zwei benachbarten Elemente mehr ungeordnet sind, ist die Liste sortiert.

```
bubblesort2([],[]).
bubblesort2([X],[X]).
bubblesort2(L,Lsort) :-
    append(L1,[A,B|R],L),
    A @> B,
    append(L1,[B,A|R],L2),
    bubblesort2(L2,Lsort).
bubblesort2(L,L).
```

■ Wesentlich effektiver als nach dem Bubblesortverfahren können Listen durch Quicksort sortiert werden. Dieses Sortierverfahren beruht darauf, daß eine zu sortierende Liste in zwei Teillisten zerlegt, diese sortiert und dann zur sortierten Liste zusammengesetzt werden. Die Zerlegung in die Teillisten erfolgt nach einem Schlüsselelement K so, daß die erste Teilliste alle Elemente die kleiner als K und die zweite Teilliste alle übrigen Elemente der Liste enthält. Wegen des unkomplizierten Zugriffs wählt man als Schlüsselelement das jeweils erste Element der zu sortierenden Liste.

```
quicksort([],[]).
quicksort([K|R],Lsort) :-
split(K,R,L1,L2),
quicksort(L1,L1sort),
quicksort(L2,L2sort),
append(L1sort,[K|L2sort],Lsort).
split(_,[],[],[]).
split(K,[E|R],[E|L1],L2) :-
E @< K, split(K,R,L1,L2).
split(K,[E|R],L1,[E|L2]) :-
split(K,R,L1,L2).
```

Beispiel:

```
?- quicksort([berta, heinrich,
alfons, suse],L).
L = [alfons, berta, heinrich, suse]
```

Binäre Bäume

Ist eine größere Menge von Objekten zu verwalten, sucht man in der Regel nach einer Datenstruktur, die einen schnellen Zugriff auf einzelne Objekte gestattet. Eine solche Datenstruktur ist der binäre Baum.

Das Prädikat 'baum_aufbau' überführt eine Liste von PROLOG-Termen (das sind in diesem Falle die Daten) in einen binären Baum durch schrittweises *Einhängen* in den jeweils aktuellen Baum. Die vier Klauseln für das Prädikat 'einhängen' sind dabei wie folgt zu interpretieren:

1. Das Objekt K in den leeren Baum eingehängt, ergibt den Baum t, bestehend aus einem Knoten mit dem Objekt K und zwei leeren Teilbäumen.
2. Enthält ein Knoten schon das eigentlich einzuhängende Objekt K, ergibt sich der alte zum neuen Baum.
3. Ist K kleiner als das Knotenelement K, so ergibt sich der neue Baum durch Einhängen von K in den linken Teilbaum.
4. Sonst ergibt sich der neue Baum durch Einhängen von K in den rechten Teilbaum.

```
baum_aufbau([],leer).
baum_aufbau([K|R],T) :-
    baum_aufbau(R,T1),    einhaengen
    (K,T1,T).
einhaengen(K,leer,t(K,leer,leer)).
einhaengen(K,t(K,L,R),t(K,L,R)).
einhaengen(K,t(E,L,R),t(E,L1,R)) :-
    K @< E, einhaengen(K,L,L1).
einhaengen(K,t(E,L,R),t(E,L,R1)) :-
    einhaengen(K,R,R1).
```

Beispiel:

```
?- baum_aufbau([das, ist,
ein, baum]),T).
T = t(baum,leer,t(ein,t(das,
leer,leer),t(ist,leer,leer))).
```

Gelegentlich kann es erforderlich sein, die in einem Baum enthaltenen Elemente wieder geordnet auszulesen. Diese Aufgabe übernimmt das Prädikat 'lies'. Es generiert eine geordnete Liste der in einem Baum enthaltenen Elemente.

```
lies(leer,[]).
lies(t(E,TL,TR),L)) :-
    lies(TL,L1), lies(TR,L2),
    append(L1,[E|L2],L).
```

Beispiel:

```
?- baum_aufbau([das, ist,
ein, baum],T), lies(T,L).
T = t(baum,leer,t(ein,t(das,
leer,leer),(ist,leer,leer)))
L[baum, das, ein, ist].
```

Zwei weitere Beispiele

Das Färben einer Landkarte

Jede politische Landkarte läßt sich mit vier Farben so färben, daß keine zwei benachbarten Länder die gleiche Farbe besitzen. Diese Feststellung aus den Anfangszeiten der Kartographie hatte große Bedeutung für die Entwicklung der Graphentheorie. Bis heute konnte man nicht nachweisen, daß die Aussage für jede normale Karte gilt (*normale Karte* ist ein Begriff aus der Topologie, trifft aber recht genau die umgangssprachlichen Vorstellungen).
Nachfolgend wird ein Programm angegeben, das Färbevorschläge für die einzelnen Stadtbezirke einer Berlin-Karte generiert. Zunächst sind alle für den Programmlauf wichtigen Informationen wie die Nachbarschaftsbeziehungen der einzelnen Stadtbezirke untereinander, die Liste der Beteiligten (die man natürlich auch aus den einzelnen Klauseln des Prädikats 'n' gewinnen könnte) und die zur Verfügung stehenden Farben anzugeben.

```
n(treptow,koepenik).
n(treptow,lichtenberg).
n(treptow,friedrichshain).
n(koepenik,lichtenberg).
n(koepenik,marzahn).
n(koepenik,hellersdorf).
n(friedrichshain,mitte).
n(friedrichshain,lichtenberg).
n(friedrichshain,prenzlauer_berg).
n(lichtenberg,prenzlauer_berg).
n(lichtenberg,marzahn).
n(lichtenberg,weissensee).
n(marzahn,weissensee).
n(marzahn,hellersdorf).
n(mitte,prenzlauer_berg).
n(prenzlauer_berg,weissensee).
n(prenzlauer_berg,pankow).
n(weissensee,pankow).

beteiligte([koepenik,treptow,
friedrichshain,prenzlauer_berg,
mitte,lichtenberg,marzahn,
hellerdorf,weissensee,pankow]).

farbe(rot).
farbe(blau).
farbe(gruen).
farbe(gelb).
```

Da das Prädikat 'n' die Symmetrie der Nachbarschaftsrelation (wenn A Nachbar von B ist, dann ist auch B Nachbar von A) nicht zum Ausdruck bringt, wird noch ein Prädikat 'nachbar' benötigt.

```
nachbar(X,Y) :- n(X,Y).
nachbar(X,Y) :- n(Y,X).
```

Die Koordinierung des Programmablaufs übernimmt das Prädikat 'faerbe_karte'. Es stellt zunächst die Liste L der Beteiligten Regionen zur Verfügung, läßt dann durch das Prädikat 'faerbe' jeder in L enthaltenen Region unter Berücksichtigung der (noch leeren) Liste der schon gefärbten Regionen eine Farbe zuordnen und veranlaßt anschließend die Ausgabe des Ergebnisses. Das Prädikat 'nochmal' dient lediglich dem Auslösen von Backtracking, falls alternative Lösungen gewünscht werden.

```
faerbe_karte :- beteiligte(L),
                faerbe(L,[],E),
                ausgabe(E),
                nochmal.
```

Die Prädikate 'faerbe' und 'prüfe' bilden den Kern des Programms. Dabei wird durch 'faerbe' jeweils für den Kopf K der Liste der noch zu färbenden Region eine Farbe F ermittelt und geprüft, ob K – gefärbt mit der Farbe F – nicht mit den schon gefärbten Regionen (enthalten in der Liste B) in Konflikt steht. Falls keine Konflikte auftreten, wird die Liste der bisher gefärbten Regionen um r(K,F) erweitert und versucht, den Rest der zu färbenden Regionen zu färben.

```
faerbe([],L,L).
faerbe([K|R],B,E) :- farbe(F),
pruefe(K,F,B), faerbe(R,[r,(K,F)|B],E).

pruefe(_,_,[]).
pruefe(K,F,[r(K1,F1)|R]) :-
      F \= F1, pruefe(K,F,R).
pruefe(K,F,[r(K1,_)|R]) :-
      not (nachbar(K,K1)),
pruefe(K,F,R).

ausgabe([]).
ausgabe([r(K,F)|R]) :-
      write(' Die Region '), write(K),
      write(' hat die Farbe '), write(F),
      nl, ausgabe(R).

nochmal :- write(' Alternative Loesungen? j/. '), get(X), X \= 106.
```

Beispiel:

```
?- faerbe_karte.
Die Region pankow hat die Farbe gruen
Die Region weissensee hat die Farbe rot
Die Region hellersdorf hat die Farbe gruen
Die Region marzahn hat die Farbe blau
Die Region lichtenberg hat die Farbe gruen
Die Region mitte hat die Farbe gruen
Die Region prenzlauer_berg hat die Farbe blau
Die Region friedrichshain hat die Farbe rot
Die Region treptow hat die Farbe blau
Die Region koepenik hat die Farbe rot
Alternative Loesungen? j/. n
yes.
```

Die Wortkorrektur

Die vier wohl häufigsten Fehler beim Schreiben von Text über eine Tastatur (ob nun mit Schreibmaschine oder Computer) sind das Weglassen oder Einfügen eines Buchstaben, das Vertauschen zweier benachbarter Buchstaben oder das Schreiben eines fehlerhaften Buchstaben innerhalb eines Wortes. Das folgende Programm testet eine eingegebene Zeichenfolge auf der Basis eines Wörterbuches auf diese Fehler und gibt – falls möglich – die korrekte Schreibweise an. Das Programm nutzt eine besondere Form der Listennotation für solche Integer-Listen, die nur darstellbare ASCII-Zeichencodes enthalten: die *Zeichenkettennotation*. Danach ist zur Liste [79, 84, 84, 79] die Notation "OTTO" gleichwertig.
Das Programm wird gesteuert über das Prädikat 'check'. Dieses liest solange Zeichenketten ein und testet sie auf korrekte Schreibweise, bis der Nutzer "ende" eingibt.

```
check:-repeat,nl,write('Wort: '),
      read_in(Wort), check(Wort),
      Wort="ende".
read_in(Wort) :- sammel(Wort), !.
sammel([X|R]) :- get0(X),
      X \= 10, sammel(R).
sammel([]).
```

Die eigentliche Überprüfung der Zeichenkette übernimmt das einstellige 'check'. Drei Fälle sind dabei zu unterscheiden:

- Es wurde "ende" eingegeben. Dann darf kein Test auf Korrektheit durchgeführt werden und 'check' muß sofort gelingen.
- Die Zeichenkette ist im Wörterbuch vorhanden. Dann erfolgt eine entsprechende Ausschrift und 'check' endet erfolgreich. Cut am Klauselende verhindert das Backtracking.
- Das Wort ist unbekannt. Es erfolgt dann ebenfalls eine entsprechende Ausschrift, in die Datenbasis wird eine (zu dieser Zeit leere) Liste der bisherigen Alternativen für das Wort eingeschrieben und die Suche nach Alternativen eingeleitet. Anschließend entfernt man die alternativen Schreibweisen wieder aus der Datenbasis und gibt sie aus. Cut verhindert wiederum ein Backtracking.

```
check("ende").
check(W):-l(W),write('Wort korrekt'),
        nl, !.
check(W):-write('Wort unbekannt, '),
    assert(alt([])),
    cs(W),
    retract(alt(Vorschlaege)),
    ausg(Vorschlaege),!.
```

Das Prädikat 'cs' vergleicht über 'cmp' die eingegebene unkorrekte Zeichenkette mit jedem Wort im Wörterbuch entsprechend den oben beschriebenen Fehlern. Werden dabei Alternativen gefunden, überprüft das Prädikat 'neu', ob diese Alternativen nicht schon bekannt sind. Werden keine Alternativen mehr gefunden, gelingt 'cs'.

```
cs(W) :- l(L), cmp(W,L),
        neu(L), fail.
cs(_).

neu(L) :- alt(V), member(L,V),!.
neu(L) :- retract(alt(V)),
         assert(alt([L|V])),!.
```

Die einzelnen Fehlertests übernimmt das Prädikat 'cmp'. Entsprechend den angegebenen Fehlern wird (in dieser Reihenfolge) getestet, ob die eingegebene Zeichenkette einen Buchstaben zuviel, einen Buchstaben zuwenig, zwei vertauschte Buchstaben oder einen falschen Buchstaben enthält.

```
cmp(WC,LC):-dch(WC,LC).
cmp(WC,LC):-dch(LC,WC).
cmp(WC,LC):-swp(WC,LC).
cmp(WC,LC):-chd(WC,LC).

dch([_|R],R).
dch([K|R],[K|Q]):-dch(R,Q).

swp([K1|[K2|R]],[K2|[K1|R]]):-K1\=K2.
swp([K|R],[K|Q]):-swp(R,Q).

chd([KX|R],[KY|R]):-KX\=KY.
chd([K|RX],[K|RY]):-chd(RX,RY).

ausg([]) :- write('keine Alternative
gefunden'), nl, !.
ausg(L) :- write('Alternativen: '),
         nl, a(L).

a([]) :- nl, !.
a([X|R]) :- name(A,X),
         write (A), tab(4), a(R).
```

Zum Abschluß sei hier noch ein Wörterbuch angegeben, es enthält Vornamen. Das Programm zur Wortkorrektur wurde während eines Programmierpraktikums mit Schülern einer 11. Klasse erarbeitet. Das Wort "Ollliver" wurde aufgenommen, um zu zeigen, daß auch verschiedene Alternativen möglich sind (siehe Beispiel).

```
l("Ingmar").
l("Thomas").
l("Oliver").
l("Ollliver").
l("Tarek").
l("Christian").
l("Andreas").
l("Daniel").
l("Matthias").
l("Gunnar").
l("Ramona").
l("Katharina").
l("Cornelia").
l("Susan").
l("Ulrike").
l("Armin").
```

Beispiel:

```
?- check.
Wort: hallo
Wort unbekannt, keine Alternative
gefunden
Wort: Ramna
Wort unbekannt, Alternativen:
Ramona
Wort: Kahtarina
Wort unbekannt, Alternativen:
Katharina
Wort: Olliver
Wort unbekannt, Alternativen:
Oliver, Ollliver
Wort: ende
yes.
```

Literatur:

/1/ Clocksin, W. F.; Mellish, C. S.: Programming in Prolog. Springer-Verlag, Berlin Heidelberg New York, 1982
/2/ Sterling, L.; Shapiro, E.: The Art of Prolog – Advanced Programming Techniques. The MIT Press, Cambridge (Massachusetts) London (England), 1986
/3/ Geske, U.: Programmieren mit Prolog. Akademie-Verlag, Berlin, 1988
/4/ Kleine Enzyklopädie Mathematik. Bibliographisches Institut Leipzig, 1977

Portierung von PASCAL-Software mit PCC

Dr. Bodo Hohberg, Olga Wikarski
Humboldt-Universität zu Berlin, Sektion Mathematik

Mit der schnellen Verbreitung von 16- und 32-Bit-Rechentechnik und UNIX-ähnlichen Betriebssystemen wächst der Bedarf, die auf CP/M- oder MS-DOS-Rechnern vorhandenen PASCAL-Programme in neue Hard- und Softwareumgebungen zu übertragen. Hauptsächlich zu diesem Zwecke wird seit Frühjahr 1988 in vielen Betrieben und Hochschulen der an der Humboldt-Universität zu Berlin, Sektion Mathematik, Bereich Informationsverarbeitung, entwickelte Portierungscompiler PCC eingesetzt.
PCC ist ein portabler PASCAL↔C-Compiler, der PASCAL-Quelltexte in C-Quelltexte (Kernighan/Ritchie-Standard) transformiert. Für die Abarbeitung der generierten C-Programme steht ein weitgehend portables Laufzeitsystem zur Verfügung.
In diesem Artikel möchten wir über Nutzung dieses Softwarewerkzeuges in der Praxis und über die dabei gesammelten Erfahrungen berichten. Der theoretische Ansatz wurde in /1/ vorgestellt.

Konzeptionelle Zielstellungen

Die Konzeption sowie die Implementation des PCC wurden unter Berücksichtigung folgender Anforderungen durchgeführt:
- Portabilität von Compiler und Laufzeitsystem durch ihre Implementation in C
- Portabilität der PASCAL-Programme durch Sicherung der Portabilität des generierten C-Codes
- hohe Effektivität der Übertragung nach C mit dem Ziel, die Gesamtübersetzungszeit von PASCAL über C in die Maschinensprache gering zu halten
- besonders gute Anpassung an UNIX-kompatible Betriebssysteme
- Einsetzbarkeit als Portierungscompiler und als universell verfügbarer PASCAL-Compiler für UNIX-kompatible Betriebssysteme
- Lauffähigkeit des Compilers in 64-K-Hauptspeichern auch für große PASCAL-Programme. Diese Forderung ergab sich aus der ursprünglichen Entwicklungsumgebung. Effektive Speicherplatznutzung ist der heutige Vorteil.

Praktische Erfahrungen

Die erste Version des PCC war im August 1987 auf P 8000 unter WEGA einsatzfähig. Danach wurde der PCC in kurzer Zeit und mit relativ geringem Aufwand auf einer ganzen Reihe von in der DDR eingesetzten Rechner/Betriebssystem-Kombinationen installiert:
P 8000: WEGA
K 1630, K 1840: MUTOS
IBM-kompatible PC: VENIX, XENIX, MS-DOS
EC 1834: DCP
ESER 1 und 2: PSU, VMX.
Das Zeitverhältnis zwischen der Transformation PASCAL↔C und der anschließenden Übersetzung von C in die Maschinensprache liegt zwischen 1:5 und 1:10. Beispielsweise lieferten umfangreiche Vergleichsmessungen auf P 8000 unter WEGA folgende Ergebnisse:

PCC:	
PASCAL↔C	5 s
C↔Maschinensprache	40 s
WEGA-PASCAL-Compiler:	
PASCAL↔Maschinencode	1,50 min.

Die Zeiten beziehen sich auf 100 Quelltextzeilen mit etwa 3 000 Zeichen. Sie wurden im Einzelnutzerbetrieb ermittelt.
Der vergleichsmäßig geringe Zeitaufwand der PASCAL-C-Übersetzung bringt den Vorteil, daß Korrekturen im PASCAL-Quelltext sehr schnell vorgenommen werden können.
Eine weitere starke Seite zeigt der PCC in bezug auf die Effektivität des erzeugten Maschinenprogramms. Er nutzt viele Möglichkeiten zur direkten Abbildung von PASCAL-Konstruktionen auf UNIX-Systemrufe. Als Beispiel sei ein Kopierprogramm erwähnt, das, übersetzt mit dem PCC, 40 mal schneller arbeitet als dasselbe, übersetzt mit dem in WEGA vorhandenen PASCAL-Compiler.
Beim Einsatz als selbständiger Standard-PASCAL-Compiler für umfangreiche Studentenpraktika hat der PCC seine Effektivität, Stabilität und einfache Dialog-Nutzung bewiesen. Mehrere zweisemestrige PASCAL-Praktika im Rechentechnischen Labor der Sektion Mathematik an der Humboldt-Universität mit 180 Fernstudenten, 25 Direktstudenten und 20 Spezialschülern sind Beispiele dafür.
Die Praxis hat gezeigt, daß der überwiegende Teil aller bei uns angemeldeten Portierungswünsche von TurboPASCAL-Nutzern kam. Um die produktive Nachnutzung des PCC in breiterem Maße zu gestatten, wurden portable Elemente der Sprachversion TurboPASCAL 3.0 aufgenommen. Dadurch wurden natürlich nicht alle Probleme aus dem Weg geräumt, und die Nachnutzer hatten noch mit einigen Schwierigkeiten zu kämpfen:
- Nicht implementierte TurboPASCAL-Sprachelemente mußten durch geeignete Standard-PASCAL-Konstruktionen ersetzt werden.
- Wer betriebssystemabhängige Standardroutinen benutzt hat, mußte dafür eigene externe C-Funktionen schreiben.
- Das Verwenden von hardwareabhängigen Programmkonstrukten (z. B. absolute Längen von Datensätzen) beeinträchtigte stark die Portabilität.

In der Anfangsphase enthielt der PCC selbst noch Fehler. Alle an den Entwickler gemeldeten Fehler wurden schnell beseitigt.

Benutzung des PCC

Vom Bereich Informationsverarbeitung der Sektion Mathematik der Humboldt-Universität werden für alle Interessenten am PCC Konsultationen mit Vorführungen angeboten. Die Gäste können eigene Programme mitbringen und versuchen, sie an Ort und Stelle nach C zu übertragen bzw. zum Laufen zu bringen.
Für die Nutzung des PCC wurden in jedem unterstützten Betriebssystem einige Kommandoprozeduren bereitgestellt. Sie werden im folgenden an Beispielen für WEGA auf P 8000 erläutert, wobei "pascal" immer die gerade aktuell behandelte Kommandoprozedur bezeichnet und "$" das Promtzeichen des Betriebssystems ist. Die Nutzereingaben sind fett gedruckt.

Einführendes Beispiel

Gegeben sei ein fehlerfreies Standard-

PASCAL-Programm, das im File "intsort.pas" vorliegt. Das Terminalprotokoll (Abb. 1) zeigt den Weg vom PASCAL-Quelltext bis zur Abarbeitung.
Die generierten Kommentare mit den ursprünglichen Zeilennummern des PASCAL-Quelltextes sind als Hilfe bei der Arbeit des Nutzers mit dem erzeugten C-Quelltext gedacht.
Die Transformation von Bezeichnern erfolgt nach folgenden Regeln:
- alle Kleinbuchstaben des PASCAL-Bezeichners werden in Großbuchstaben umgewandelt
- PASCAL-Bezeichner werden als voneinander verschieden betrachtet, falls sie sich mindestens in einem Zeichen unterscheiden
- Bezeichnerkonflikte werden durch Voranstellen eines Präfixes beseitigt. Treten Bezeichnerkonflikte für denselben Bezeichner während der Modifikation wiederholt auf, so erfolgt die Generierung des Präfixes in der Reihenfolge: a, b,...,z, aa,...,zz,....

Die hier verwendete Kommandoprozedur hat folgendes Aussehen:
p1 <$1.pas | p2 | p3 | p4 >$1.c
cc $1.c pclib.a −o $1.
Der Parameter $1 dieser C-shell-Prozedur repräsentiert den PASCAL-Quelltextnamen ohne Extension. Das Ergebnis der parallelen Arbeit aller vier Pässe des Compilers (p1, p2, p3, p4), die über Pipes gekoppelt sind, ist der generierte C-Quelltext im File "intsort.c". Das Datenfluß-Diagramm (Abb. 2) veranschaulicht den Ablauf.
Anschließend wird "intsort.c" vom C-Compiler übersetzt und unter Berücksichtigung der Querbezüge zum PCC-Laufzeitsystem "pclib.a" ein ausführbares Programm "intsort" erzeugt.

Compiler-Optionen

Die Arbeit des Compilers kann durch eine Reihe von Optionen beeinflußt werden, die entsprechend ihrer Bedeutung nur auf bestimmte Pässe des Compilers wirken.
Option -C (Paß p1) schaltet das Übernehmen der Kommentare aus dem PAS-

```
$ cat intsort.pas
program pointertest;
type tree=@node;
     node=record info: integer; left, right: tree end;
var t: tree; i: integer;
procedure insert(var p: tree; x: integer);
begin
 if p=nil then
  begin new(p); p@.info:=x; p@.left:=nil; p@.right:=nil end
  else if x<p@.info then insert(p@.left,x)
                    else insert(p@.right,x);
end;
procedure outtree(p:tree);
begin
 if p<>nil then
  begin outtree(p@.left); writeln(p@.info); outtree(p@.right)
  end;
end;
begin t:=nil; writeln;
 writeln('Geben Sie einzeln ganze Zahlen ein!');
 writeln('(0 - schliesst die Eingabe ab)');
 repeat write(':'); readln(i); if i>0 then insert(t,i);
 until i=0;
 writeln('Die sortierte Reihenfolge:');
 outtree(t);
end.

$ pascal intsort
PASCAL-->C (Rel. 2.3)   (C)  Humboldt-Uni. Berlin 1989
Compiling C

$ intsort
Geben Sie einzeln einige ganze Zahlen ein!
( 0  -  schliesst die Eingabe ab )
6
13
2345
89
1
0
          Die sortierte Reihenfolge:
          1
          6
         13
         89
       2345

$ cat intsort.c
#include "pas.h"

int standard=1; /**1**/

typedef ref(NODE) TREE;

typedef struct rec_0
{ int  INFO; TREE  LEFT, RIGHT; } NODE;

TREE  T;
int  I;

extern int argc();

ainsertl (P, cX) /**6**/
     TREE *P;
     int cX;
{ initfunc(7);
   if ((*P)==nil)
     { (*P)=new(NODE); /**8**/
       deref((*P),NODE).INFO= cX;
       deref((*P),NODE).LEFT= nil;
       deref((*P),NODE).RIGHT= nil;
       } /**9**/
   else
     if (cX<deref((*P),NODE).INFO)
       ainsertl(&(deref((*P),NODE).LEFT),cX); /**10**/
     else
       ainsertl(&(deref((*P),NODE).RIGHT),cX);
 } /**11**/

OUTTREE (aP) /**13**/
     TREE aP;
{ initfunc(14);
   if (aP!=nil)
     { OUTTREE(deref(aP,NODE).LEFT);
       { wi(deref(aP,NODE).INFO,10); wln(); }  /**15**/
       OUTTREE(deref(aP,NODE).RIGHT); /**16**/
       }
 } /**17**/

main(largc,largv) int largc; char *largv[]; /**18**/
 { inimain(18,&largc,largv);
   T= nil;
   { wln(); }
   { ws("Geben Sie einzeln ganze Zahlen ein!"); wln(); }  /**19**/
   { ws("(0 - schliesst die Eingabe ab)"); wln(); }  /**20**/
   do /**21**/
   { { wc(':'); }
     { reading(input); I=ri(); rln();  }
      if (I>0)
       ainsertl(&(T),I);
     } /**22**/
   while (!(I==0));
   { ws("Die sortierte Reihenfolge:"); wln(); }  /**23**/
   OUTTREE(T); /**24**/
 } /**25**/
```

Abb. 1 Einführendes Beispiel

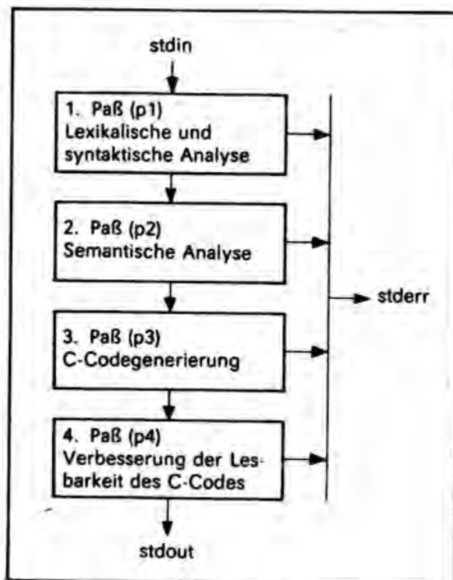


Abb. 2 Arbeitsweise des PCC

CAL-Programm in den generierten C-Quelltext aus.

Option -L (Paß p4) schaltet das Eintragen der Zeilennummern als Kommentare in den generierten C-Quelltext aus.

Option -Ennn (Paß p1 und Paß p2) legt die maximal angezeigte Anzahl von Fehlern auf "nnn" fest. Der Standardwert ist 30.

Option -D (Paß p2) stellt die Position des ersten Quelltextfehlers bereit. Sie bewirkt die Ausgabe des Quelltextnamens oder des Include-Filenamens und der Zeilennummer, in der der erste Fehler enthalten ist, in das File "/tmp/pccnnn", wobei "nnn" der Rückkehrcode des zweiten Passes ist. Diese Information kann im Aufruf eines Editors nach dem zweiten Paß verwendet werden.

Option -D dient der Organisation einer Übersetzung mit Korrekturen im Dialog für große Programme, wobei include-Dateien berücksichtigt werden.

Option -I (Paß p3) schaltet das Einfügen von Anweisungen für die Erzeugung eines Laufzeitprotokolls in den generierten C-Quelltext aus. In diesem Fall entsteht ein kürzerer Objektcode. Falls ein Laufzeitprotokoll vorgesehen ist, wird bei jedem Aufruf einer Funktion die Zeilennummer der ersten Zeile dieser Funktion im PASCAL-Quelltext ausgegeben. So kann der Weg durch das Programm verfolgt werden.

Option -H (Paß p4) bewirkt das Zusammenstellen der externen Definitionen für alle im Programm vereinbarten Bezeichner in einem Headerfile. Diese Option kann nicht für TurboPASCAL-Programme mit beliebiger Reihenfolge der Vereinbarungen verwendet werden.

Option -S (Paß p4) mit den drei Parametern "rx", "ry" und "headerfile" schaltet die Generierung des C-Quelltextes von Routine "rx" bis einschließlich Routine "ry" und das Einfügen von *#include <headefile>* als erste Zeile des Quelltextes ein.

Diese Option kann im Zusammenwirken mit der Option -H (ausgehend vom Zwischencode nach dem dritten Paß) zur Zerlegung eines Programms in ein C-Headerfile und mehrere C-Quellen, die unabhängig kompilierbar sind, verwendet werden.

Option -F (Paß p1) mit dem Parameter "filename" fügt ein File mit dem angegebenen Filenamen in den PASCAL-Quelltext hinter dem Vereinbarungsteil ein.

Diese Option kann zum Anschließen zusätzlicher C-Nutzerroutinen verwendet werden. Die Routinenköpfe, denen das Schlüsselwort "extern" folgen muß, können in einem File zusammengefaßt und durch die Option -F in das Hauptprogramm eingelesen werden. Damit ist gesichert, daß die Routinennamen bei der Übersetzung bekannt sind. Der Objektcode ihrer C-Implementationen muß im Link-Kommando angegeben werden.

Option -P (Paß p1) mit dem Parameter "filename" muß für die Bearbeitung von TurboPASCAL-Programmen verwendet werden. Durch diese Option wird gesichert, daß

- die später beschriebenen syntaktischen und semantischen Besonderheiten korrekt verarbeitet werden,
- die externen PASCAL-Definitionen für die aufgenommenen Standardroutinen von TurboPascal 3.0 (File "filename") während der Übersetzung zur Verfügung stehen.

Einige Beispiele zur Benutzung von Compiler-Optionen sind in den nächsten Abschnitten enthalten.

Arbeit im Dialog

Beim Portieren von TurboPASCAL-Programmen wird man selten ohne Änderungen im PASCAL-Quelltext auskommen können. Dafür und auch für PASCAL-Anfänger, die den PCC als PASCAL-Compiler benutzen möchten, sind zwei andere Kommandoprozeduren, die eine Übersetzung im Dialog gestatten, besser geeignet. Diese Dialog-Kommandoprozeduren unterscheiden sich dadurch, daß die eine nur für Programme mit weniger als 255 Zeilen und ohne include-Dateien benutzt werden kann, wobei die andere diesen Einschränkungen nicht unterliegt. Abb. 3 demonstriert ihre Arbeitsweise an einem kleinen Programm.

Nach der Eingabe von y wurde in der Kommandoprozedur der Editor "vi" aufgerufen und sofort auf die erste fehlerhafte Zeile positioniert: der Cursor steht vor der writeln-Anweisung in der Zeile 7. Der Nutzer bewegt ihn bis zur fehlerhaften Stelle, fügt die fehlende runde Klammer ein (mit I wird der Eingabemodus eingestellt, mit **<esc>** verlassen) und beendet mit **:x** den Editiervorgang. Das korrigierte Programm wird automatisch neu übersetzt. Es entstehen der C-Quelltext "kurve.c" und das ausführbare Programm "kurve".

Sprachumfang

Standard-PASCAL

Die ursprüngliche Fassung des PCC bezog sich auf Standard-PASCAL /4/. In diesem Rahmen wurde der PCC mit der PASCAL Validation Suite /5/ getestet, wobei einige Einschränkungen gegenüber Standard-PASCAL (zum großen Teil durch Eigenschaften der Sprache C bedingt) festgestellt worden sind:

- Sprünge aus einer Routine heraus sind verboten
- einer Funktion kann nur im Köper dieser Funktion (nicht in anderen enthaltenen Funktionen) der Funktionswert zugewiesen werden; das Fehlen einer Wertzuweisung an einen Funktionsbezeichner wird nicht bemängelt

• leere Datenfelder in Recordtypen sind verboten, weil sonst Fehler im generierten C-Text entstehen
• eine formale Funktion darf beim Aufruf nicht durch den Bezeichner einer rekursiven Funktion ersetzt werden, Verstöße gegen diese Einschränkung werden vom Compiler nicht bemerkt
• das PACKED-Attribut wird ignoriert
• der Basistyp eines Files muß ein Typidentifier sein.
• als Zeichenketten werden alle (gepackten und ungepackten) Felder von Zeichen betrachtet, die mit dem Index 0 oder 1 beginnen. Alle Zeichenketten sind untereinander typverträglich. Bei Ergibtanweisungen und aktuellen Parametern erfolgt keine Längenkontrolle für Zeichenketten
• es kann nicht überprüft werden, ob
– die Grenzen eines Teilbereichstyps zulässig sind
– die Werte, die einem Teilbereich angehören sollen, tatsächlich in diesem auch enthalten sind (Indizes eines Feldelementes, CASE-Selektor, rechte Seite einer Ergibtanweisung)
– eine Variante eines RECORD-Feldes zu diesem Zeitpunkt benutzt werden darf (keine Kontrolle des Wertes eines "tag fields"),
– eine Gleitkommazahl im zulässigen Wertebereich einer bestimmten C-Implementation liegt
– Pointervariablen mit legitimen Werten belegt sind.

Erweiterungen für Standard-PASCAL

Es gibt viele PASCAL-Dialekte, die echte Erweiterungen von Standard-PASCAL sind. Das heißt, daß sie Syntax und Semantik von diesem Standard einhalten und außerdem die Sprache durch Konstruktionen für die Arbeit mit Dateien, Zeichenkettenverarbeitung usw. bereichern. Der PCC bietet auch eine Reihe solcher Spracherweiterungen an, die wir kurz vorstellen möchten.
Es ist möglich, aus einem PASCAL-Programm heraus auf Programmparameter mit Hilfe der vordefinierten Routinen **argc** und **argv** zuzugreifen.
function argc : integer; extern;
procedure argv (w:integer;
str:stringt);extern;
Zwei zusätzliche Typen wurden vordefiniert:
• *undefined* als "undefinierter" Typ, um externe Routinen zu benutzen, deren Parameter keinem Typ fest zugeordnet werden können
• *stringt* als ein zu allen Zeichenkettentypen kompatibler Typ zur Parameterspezifikation von Zeichenkettenparametern.
Kommentare der Form **(*$c <C-Quelltext> *)**dienen zur Übernahme von C-Quelltexten in unveränderter Form in den generierten C-Code.
Zur Zerlegung großer PASCAL-Programme können include-Kommentare der Form **(*$i <dateiname> *)** benutzt werden.
Beispiel: Ein Programm ist in die Teile teil1.pas und teil2.pas zerlegt. Dem PCC ist ein Quelltext mit folgendem Inhalt zu übergeben:
(*$i teil1.pas *)
(*$i teil2.pas *).
Die Direktive *extern* besagt, daß der zu einem Routinenkopf gehörige Routinenblock sich außerhalb des Programmblocks befindet. Mit dem Konzept der externen Routinen hat der Nutzer die Möglichkeit, Funktionen aus C-Standard- oder Nutzerbibliotheken im PASCAL-Quelltext anzukündigen und damit die Übersetzung von Aufrufen dieser Funktionen zu ermöglichen.
Bei der Implementation der Dateioperationen werden *geblockte Dateien* und *Textdateien* unterschieden. Die Implementation der geblockten Dateien erfolgte mit Hilfe der Systemrufe open, creat, read, write und close, während die Implementation der Textdateien auf die Standardfunktionen fopen, getc, fscanf usw. zurückgeführt wurde. Das eröffnet die Möglichkeit des Direktzugriffs auf geblockte Dateien und der Benutzung von in C-Systemen vorhandenen Funktionen für die Arbeit mit Textdateien.

```
$ pascal kurve
PASCAL-->C (Rel. 2.3)   (C)  Humboldt-Uni. Berlin 1989
error:109: file stdin line 8 pos 50 error in expression
error:105: file stdin line 8 pos 50 missing ')'
First error in line 8
edit(y/n)?
y
program kurve;
var i: integer; x: real;
begin
 for i:=0 to 100 do
  begin x:=exp(-i/20)*sin(i/5);
   writeln(x:10:7,' : ','*':round(30+30*x) I)<esc>;
  end
end.
:x
no errors
Compiling C

$ cat kurve.c
#include "pas.h"
int standard=1; /**1**/

int  I;
double  X;

extern int argc();

main(largc,largv) int largc; char *largv[]; /**3**/
{ inimain(3,&largc,largv);
  { int for_h;I=0;for_h=100;
    if(I<=for_h) while(1){
        { X= exp( -(I/(double)20))*sin(I/(double)5);
          { wf(X,10,7); /**6**/
            ws(" : ");
            wcl('*',round(30+30*X));
            wln();
          }
          } /**8**/
      if(I++ >= for_h) break;
    }}
 }
```

Abb. 3 Arbeit im Dialog

```
procedure assign (var f:filetyp; st:stringt); extern;
procedure seek (var filevar:undefined; n:integer); extern;
procedure close (var filevar:undefined); extern;
procedure erase (var filevar:undefined); extern;
procedure rename (var filvar:undefined; str:stringt); extern;
function  filepos (var filevar:undefined):integer; extern;
function  filesize (var filevar:undefined):integer; extern;
```

```
procedure delete  (var str:stringt; pos,num:integer); extern;
procedure insert  (obj:stringt; var target:stringt;
                   pos,len:integer); extern;
procedure sprintf (var str:stringt; muster:stringt; x:undefined);
                   extern;
procedure sscanf  (str:stringt; muster:stringt; var x:undefined);
                   extern;
function copy     (st:stringt; pos, num:integer):stringt; extern;
function concat   (st1, st2:stringt); extern;
function length   (st:stringt):integer; extern;
function pos      (obj, target:stringt):integer; extern;
```

Abb. 4 Zusätzliche E/A-Standardroutinen

Abb. 5 Zusätzliche Routinen für die Arbeit mit Zeichenketten

Die zusätzlichen Standardroutinen für die Ein- und Ausgabe sind entsprechend ihrer externen Vereinbarung (Abb. 4) zu benutzen.

Die *Zeichenketten* sind anders als in TurboPASCAL implementiert. Zeichenkettenkonstanten erhalten bei der Transformation nach C ein zusätzliches letztes Zeichen mit dem internen Wert Null als Längenbegrenzung. Bei jeder Ausgabeoperation, Wertzuweisung zwischen Strings und bei den Stringoperationen wird dieses Zeichen benutzt.

Auf die Länge von Zeichenketten kann (im Unterschied zu TurboPASCAL) nur über die Funktion "length" zugegriffen werden. Der Index 0 adressiert das erste Zeichen einer Stringvariablen.

Die Benutzung der Zeichenketten-Funktionen und -Prozeduren erfolgt entsprechend den externen Vereinbarungen (Abb. 5). Die Standardroutinen *exit, random, randomize,* und *sizeof* sind in ihrer Funktionsweise den gleichnamigen TurboPASCAL-Routinen äquivalent.

Zur Aufnahme der externen PASCAL-Definitionen (File "pas.pas") der zusätzlich zum Standard-PASCAL bereitgestellten Standardroutinen in den C-Code dient die Option -F:

```
p1<$1.pas -F pas.pas|p2|p3|p4>$1.c
cc $1.c pclib.a -o $1.
```

Die Laufzeitbibliothek "pclib.a" enthält die Implementation dieser Routinen

Teilimplementation von TurboPASCAL 3.0

Zur Bearbeitung von TurboPASCAL-Programmen sollte die Kommandoprozedur

```
p1<$1.pas -P pct.-pas|p2|p3|p4>$1.c
cc $1.c pclib.a -o $1.
```

benutzt werden. Mit Hilfe der Compiler-Option -P wird die Fähigkeit des PCC, die aufgenommenen TurboPASCAL-Sprachelemente zu akzeptieren, eingeschaltet. Die externen PASCAL-Definitionen für die aufgenommenen Standardroutinen von TurboPASCAL 3.0 befinden sich im File "pct.pas", der Objektcode der entsprechenden C-Implementationen wird beim Linken der Laufzeitbibliothek "pclib.a" entnommen. Diese Compiler-Option beeinflußt sowohl die Syntaxkontrollen als auch die Transformation nach C. Dabei stehen entgegen der TurboPASCAL-Konvention die Sprachelemente von Standard-PASCAL (und auch die o. g.) weiterhin zur Verfügung.

Es folgt eine kurze Zusammenfassung der zusätzlich zu Standard-PASCAL akzeptierten Erweiterungen, die eine Teilimplementation von TurboPASCAL 3.0 darstellen:

- Standardkonstanten, hexadezimale Konstanten
- Unterstreichungszeichen in den Bezeichnern, Steuer-Zeichen in Zeichenketten
- Kommentarklammern als { } bzw. (* *) und verschachtelte Kommentare
- Freie Anordnung der Sektionen innerhalb des Deklarationsteils
- Typen BYTE und STRING (ohne Abbildung auf TurboPASCAL-ähnliche interne Darstellungen)
- Logische Operationen mit ganzen Zahlen, Operatoren SHL, SHR, XOR und Typumwandlung durch "retype"
- ELSE-Klausel in CASE-Anweisungen
- Prozeduren und Funktionen:

READ und WRITE (auch für Nicht-Text-Files)
EXIT, RANDOM, RANDOMIZE, SIZEOF,
GETMEM, FREEMEM,
DELETE, INSERT, COPY, CONCAT, LENGTH, POS, VAL, STR,
ASSIGN, SEEK, CLOSE, ERASE, RENAME, FILEPOS, FILESIZE, SEEKEOLN, SEEKEOF,
HALT, MOVE, FILLCHAR, DELAY, FRAC, SWAP, HI, LO, UPCASE, PARAMCOUNT, PARAMSTR, IORESULT,
GOTOXY, CLREOL, CLRSCR.

TurboPASCAL-Nutzer werden darauf aufmerksam gemacht, daß folgende Sprachelemente *nicht* implementiert sind:

- Overlay-Routinen
- Absolute Speicheradressierung durch Eingabe einer Konstanten: Für ein konkretes Betriebssystem kann der Nutzer äquivalente Sprachkonstruktionen mit Hilfe externer Routinen schaffen, da in C direkte Adressen angegeben werden können
- INLINE-Anweisungen: Diese Anweisungen besitzen eine vom Betriebssystem abhängige Semantik. Auch die Form (z. B. Assemblerbefehle) ist nicht portabel
- Typisierte Konstanten für Recordtypen
- Routinen MARK, RELEASE und MAXAVAI für die Arbeit mit Pointern
- Routine FLUSH
- Bildschirmsteuerroutinen außer GOTOXY, CLRSCR, CLREOL.

Portabilität von PASCAL-Programmen

Bei der Nutzung des PCC haben wir die Erfahrung gemacht, daß die Portabilität

von vorhandener PASCAL-Software sehr unterschiedlich ist. In einigen Fällen mußten wir ganz von dem Versuch abraten, ein Softwareprodukt mit dem PCC zu portieren. Die wichtigsten Faktoren, die den Portierungsaufwand mit Hilfe des PCC bestimmen, sollen im folgenden genannt werden:

- Lokalisierung nicht portabler Teile (Direktzugriffe zum Hauptspeicher, Bildschirmsteuerung, Verwendung von Betriebssystemrufen usw.)
- Neuprogrammierung von Teilen, die der PCC nicht übersetzen kann
- Übersetzung mit Hilfe des PCC
- Änderungen am C-Quelltext zur Berücksichtigung spezieller Eigenschaften des Zielbetriebssystems
- Testarbeiten (Regressionstest)
- Änderungen an der Dokumentation
- Transformation der zu verarbeitenden Dateien.

Der Portierungsaufwand von Standard-PASCAL-Programmen beträgt im allgemeinen wenige Stunden. Er reduziert sich auf die Übersetzung der Quellen und einen Regressionstest, der die einwandfreie Portierung bestätigen soll.

Besonders kritisch ist die Portierung von Software, die viele schlecht lokalisierbare, nicht portable Teile enthält. Die Wahrscheinlichkeit ist groß, daß nicht portable Teile erst beim Regressionstest bemerkt werden. Eine aufwendige Fehlersuche und Überarbeitung der PASCAL-Quellen oder der generierten C-Texte sind in diesem Fall zu erwarten.

Existiert vergleichbare nachnutzbare Software, so könnte es ökonomischer sein, vorhandene Datenverarbeitungsprojekte auf diese Software umzustellen. Die letzte Möglichkeit ist die Neuprogrammierung der Software (oder einzelner Teile) unter Ausnutzung der vorhandenen Erfahrungen.

Anpassung von TurboPASCAL-Programmen

Enthalten TurboPASCAL-Programme Sprachelemente, die nicht durch den PCC nach C transformiert werden, so bietet es sich als erstes an, die entsprechenden Programmteile mit Hilfe transformierbarer Sprachelemente neu aufzuschreiben. Hierbei ist folgende Vorgehensweise zu empfehlen:

- Start des PCC und damit der Fehlerkontrolle
- Analyse der vom PCC ausgegebenen Fehlermitteilungen
- Auf die Frage des Compilers 'edit (y,n)?' mit 'y' antworten
- sofort korrigierbare Fehler beseitigen
- Weiterarbeit des PCC abwarten und mit den nächsten angezeigten Fehlern ebenso verfahren.

Diese Vorgehensweise hat den Vorteil, daß man den PCC nutzt, um die notwendigen Änderungen zu erkennen. Als Voraussetzung dafür besitzt der PCC eine gute Fehlerstabilisierung. Man ist also nicht gezwungen, sofort alle Fehler zu beseitigen. Außerdem gehört zur Anwenderbeschreibung eine ausführliche Liste der Fehlermitteilungen, in der die verschiedenen Fehlerursachen beschrieben sind.

An einigen Beispielen sollen die Anpassungsarbeiten für TurboPASCAL 3.0 demonstriert werden.

① *Standardfunktionen Ptr und Addr*

Bei einer Transformation dieser Standardfunktionen muß man sich die Arbeit mit Adressen genauer ansehen. Mit der TurboPASCAL-Anweisung

```
P:= Ptr(Addr(X));
```

soll dem Pointer P die Adresse der Variablen X zugewiesen werden. Beim Aufruf einer Prozedur wird für einen Variablen-Parameter die Adresse übergeben. Das läßt sich hier ausnutzen. Bei einer Neudefinition der Standardfunktion Addr als externe Funktion

```
function Addr(var X: undefined):
   undefined; extern;
```

wird für einen beliebigen Parameter die Adresse des Parameters übergeben. Nun wäre nur noch zu sichern, daß diese Adresse das Ergebnis der Funktion Addr ist. Das würde aber im erzeugten C-Programm zu Typkonflikten führen. Besser ist es daher, durch eine Makrodefinition den Aufruf selbst durch den Parameter zu ersetzen:

```
#define addr(x) x
```

(Namen externer Routinen werden im erzeugten C-Quelltext klein geschrieben.)

Diese Makrodefinition läßt sich in einem C-Quelltext-Kommentar angeben. Insgesamt sind also im Quelltext folgende Änderungen notwendig:

```
(*$c
#define addr(x) x
*)
function Addr(var X: undefined):
      undefined; extern;
:
p:=Addr(X);
```

Falls Pointer-Adreß-Zuweisungen wiederholt zu transformieren sind, lassen sich die beiden Definitionen in das File für die externen Definitionen pct.pas aufnehmen und so standardmäßig zur Verfügung stellen (ab Rel. 2.4. im PCC enthalten).

② *Standardprozedur DelLine*

Einige der fehlenden Standardroutinen lassen sich in PASCAL implementieren. Andere können leicht in C realisiert werden.

Die TurboPASCAL-Prozedur DelLine kann für das Betriebssystem WEGA auf P 8000 wie folgt in PASCAL formuliert werden:

```
procedure DelLine;
begin write(^['R') end;
```

③ *In C implementierte Routinen*

Zu unterscheiden ist zwischen der Verwendung von Systemrufen, bereits in Bibliotheken enthaltenen C-Routinen und selbst implementierten C-Routinen.

- Soll z. B. der Systemruf "getpid" für die Generierung prozeßabhängiger Namen benutzt werden, so benötigt man im PASCAL-Programm eine extern-Definition, die zur Kontrolle der richtigen Benutzung dieser Funktion erforderlich ist:

```
function getpid:integer; extern;
```

Die Benutzung der Funktion erfolgt wie üblich, also z. B.

```
pid:=getpid; (* Bereitstellen der Prozeßnummer *).
```

- Wird eine Routine, die in einer Bibliothek (z. B. libXXX.a) steht, benutzt, so ist beim Linken des generierten und dann übersetzten C-Programms darauf zu achten, daß diese Bibliothek mit angegeben wird. Für ein generiertes Pro-

gramm prog.c hat der C-Compiler-Aufruf die Form:
cc prog.c pclib.a -lm libXXX.a -o prog.

• Als Beispiel für in C zu implementierende Routinen sollen zwei Routinen zur Zeitmessung geschrieben werden:
starttime: Festhalten der Startzeit (in Sekunden)
timediff: Ausgabe der Zeitdifferenz seit der Startzeit (in Sekunden).
Zur Implementation dieser Routinen soll der Systemruf
long time(X) long X;
benutzt werden. Dieser Ruf liefert einen ganzzahligen Wert vom Typ long, der nicht in einer INTEGER – Variablen gespeichert werden kann. Daher ist eine direkte Benutzung dieses Systemrufs als Funktion in PASCAL nicht möglich.
Die Implementation der gewünschten Routinen liefert den folgenden C-Text:

```
long time();/*externe Definition*/
long startt;/*fuer die Startzeit*/
starttime() { startt = time(0); }
int timediff()
{long tl; int t;
tl=time(0);
t=tl–startt; return(t);
}
```

Steht dieser C-Quelltext in der Datei ttest.c und ergibt die Übersetzung den Objektmodul ttest.o, so ist ttest.o als zusätzlicher Parameter beim Linken eines die obigen Funktionen benutzenden Programms anzugeben. Es sind also die gleichen Aktionen erforderlich wie bei der Nutzung externer Routinen in C.
Die in das benutzende PASCAL-Programm einzufügenden extern-Definitionen haben die Form:
procedure starttime; extern;
function timediff:integer; extern;.

Änderungen am generierten C-Quelltext

In dem einführenden Artikel zum PCC /1/ wurde ein Überblick über die Transformation von PASCAL-Sprachelementen nach C gegeben. Die Beispiele in diesem Artikel zeigen, daß für die meisten Sprachelemente ein gut lesbarer C-Quelltext entsteht. Eine Ausnahme machen erstens die Globalisierung lokaler Variabler bei der Auflösung von verschachtelten Routinendefinitionen und zweitens die Transformation von case-Strukturen in Records. Damit hängt der Zugriff zu Record-Komponenten und zu globalisierten Parametern zusammen. Der PCC kann z. B. Rekursivitäten nicht erkennen und muß daher in jedem Fall eine Transformation vornehmen, die eine Rekursivität zuläßt und daher zum Teil komplizierter als nötig ist.
Änderungen am generierten C-Code sind schwierig und fehleranfällig, besonders wenn PASCAL-Programme die oben angegebenen Sprachelemente enthalten. Änderungen am generierten C-Quelltext sind daher auch C-Programmierern nicht in allen Fällen zu empfehlen. Eine Ausweichmöglichkeit besteht darin, in den PASCAL-Programmteilen, die weiter bearbeitet werden sollen, z. B. Prozedurverschachtelungen zu beseitigen und so für einen besser lesbaren C-Code zu sorgen.

Portabilität des generierten C-Codes

Um die Portabilität des generierten C-Codes zu sichern, wurde darauf geachtet, daß der generierte C-Code nur solche Sprachelemente enthält, die auch bei der Implementation des Compilers benutzt wurden. Damit sind die erzeugten C-Programme überall dort lauffähig, wo der PCC läuft. Die gleiche Zielstellung galt für das Laufzeitsystem des PCC, durch das die Standardfunktionen und -prozeduren bereitgestellt werden. Hierdurch wurde z. B. erreicht, daß man ein PASCAL-Programm im Betriebssystem WEGA des P 8000 nach C transformieren und dann zusammen mit dem im Quelltext vorliegenden Laufzeitsystem in der PSU (ESER) übersetzen und ausführen kann.
Bei derartigen Portierungen sind folgende Besonderheiten zu beachten:
• Die Kommendoprozedur zu Erzeugung des Laufzeitsystems ist abhängig vom konkreten Betriebssystem.
• Unterschiede zwischen dem Betriebssystem MS-DOS und UNIX-artigen Betriebssystemen erfordern, daß vor der Generierung des Laufzeitsystems für MS-DOS ein *#define MSDOS 1* in das Headerfile des Laufzeitsystems einzufügen ist.
Bei der Überführung des Laufzeitsystems in ein neues Betriebssystem wird ein Regressionstest empfohlen, um eventuelle Abweichungen von der geforderten Arbeitsweise erkennen zu können.
Die für den Nutzer des PCC so wichtige Portabilität des Laufzeitsystems machte es unmöglich, die betriebssystemabhängigen Standardroutinen (z. B. Bildschirmansteuerung) in das Laufzeitsystem aufzunehmen. Derartige Standardroutinen müssen für jedes Betriebssystem, in das sie übernommen werden sollen, neu implementiert werden. Das ist für einen Nutzer, der nur wenige dieser Standardroutinen für ein konkretes Betriebssystem benötigt, eine realisierbare Aufgabe.

Portierung vorhandener Dateien

Auf die Möglichkeit, daß neben den Programmen unter Umständen auch Dateien portiert werden müssen, soll besonders hingewiesen werden. Oft stößt man auf derartige Probleme erst, wenn man bei der Erprobung der portierten Programme ist und dann den Fehler in den Programmen sucht. In 16-Bit-Rechnern erfolgt das Speichern von ganzen Zahlen nur auf geraden Adressen, während in 8-Bit-Rechnern beliebige Adressen benutzt werden. Dadurch können sich bei der Ausführung der gleichen PASCAL-Programme unterschiedliche Record-Größen und folglich auch unterschiedliche Datensatzgrößen ergeben.
Sei

```
type
alfa = array[1..15] of char;
Ware = record
         Name : alfa;
         EVP: integer;
         verfuegbar : boolean
       end;
Warendatei = file of Ware;
```

so ist die Länge des Record Ware auf 8-Bit-Rechnern 18 Byte aber auf 16-Bit-Rechnern 19 Byte, da dort EVP auf einer durch zwei teilbaren Adresse beginnen muß.
Eine Warendatei muß also nach der Portierung eines entsprechenden PASCAL-Programms transformiert werden, indem man die Datensätze um das eine zusätzliche Zeichen erweitert.

Interessenten werden sich bitte an
Dr. Bodo Hohberg bzw. Olga Wikarski, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Telefon: 20 93 29 82.

Literatur

/1/ Bothe, K.; Hohberg, B.; Horn, Ch.; Wikarski, O.: PASCAL↔C – Portabilität durch Quelltexttransformation. edv-aspekte 4/1987, S. 44–47
/2/ Kernighan, B. W.; Ritchie, D. M.: The C Programming Language. Prentice-Hall, 1978.
/3/ TurboPASCAL Version 3.0, reference manual; BORLAND, 1985.
/4/ Däßler, K.; Sommer, M.: PASCAL. Einführung in die Sprache. Norm-Entwurf DIN 66256. Springer, 1983.
/5/ Wichmann, B. A.: Sale A. H. J.: A PASCAL processor validation suite. NPL Report CSU 7, 1980.
/6/ Saeltzer, G.: Portabilitäts-Technologie. rd 22 (1985) 1, S. 28–32.

Modula-2 für den P 8000

Dr. Wilfried Grafik, Heinz Werner
Humboldt-Universität zu Berlin, Sektion Mathematik

Modula-2 nimmt seit langem einen festen Platz unter den Hochsprachen ein und konnte diesen auf verschiedenen Rechnerklassen weiter ausbauen. Die Sprache selbst sowie ihre Vorzüge und Anwendungsmöglichkeiten sind in Veröffentlichungen beschrieben /1/. Modula-2 kann ihren Platz unter den Programmiersprachen nur behaupten, wenn Compiler und Programmierumgebungen mit der Entwicklung der Softwaretechnik Schritt halten /2/. Implementationen für die 16-Bit-Technik unter dem Betriebssystem DCP sind verfügbar und in breitem Einsatz. Problematischer wird es mit der kommerziellen Verfügbarkeit von Modula-2 in UNIX-Umgebungen. Deshalb wurde an der Humboldt-Universität ein Modula-2-Compiler für den P 8000 unter dem Betriebssystem WEGA entwickelt, der sich an die übliche UNIX-Philosophie anlehnt. Dieser Beitrag beschreibt die Besonderheiten der Modula-2-Implementation für das Betriebssystem WEGA.

Quellsprache

Der Modula-2-Compiler für den P 8000 (im folgenden als M 8 bezeichnet), ist quelltextkompatibel zum Modula-2-Compiler (Version 1.0) vom Kombinat Robotron für das Betriebssystem DCP. Realisiert sind die Sprachänderungen an der Sprache Modula-2 aus dem Jahre 1984 /3/. Die markantesten Änderungen betreffen die Gestaltung des Modulinterface: Alle in einem Definitionsmodul definierten Bezeichner werden exportiert. Die Exportliste entfällt im Definitionsmodul. Alle anderen Änderungen von Wirth an Modula-2 betreffen Details, die beim Umstieg auf diese Sprachversion keine Probleme darstellen. Die dadurch im Quelltext vorzunehmenden Änderungen sind gering. Sie werden vom Compiler durch Fehlermeldungen ausgewiesen. Dadurch erübrigt sich hier eine Diskussion dieser Änderungen.

Übersetzungsstrategie

Der M-8-Compiler erzeugt aus Implementationsmodulen (.mod-Files) Assemblerquelltext für den Assembler **cas** (.s-Files). Er übernimmt damit die im Betriebssystem WEGA durch die C-Compiler **cc** und **scc** verfolgte Strategie. Die Assemblersprache des **cas** ist somit eine Schnittstelle für die Verbindung mit Programmen, die in anderen Sprachen formuliert sind. Vom M-8-Compiler erzeugte Assemblerprogramme und daraus entstehende Objektmodule sind linkfähig mit anderen Objektmodulen der C-Umgebung, insbesondere mit den Routinen der C-Bibliotheken. Somit lassen sich in Modula-2 zu langsam verlaufende Routinen durch effektive Lösungen in C beschleunigen, ohne die Prinzipien der modularen Programmierung zu verletzen. Ein weiterer Vorteil dieser Strategie ist die mögliche Sichtkontrolle des generierten Programms. Dessen Lesbarkeit kann durch die wahlweise Übernahme der Modula-2-Quelltextzeilen in den Assemblertext noch erhöht werden.

Die Nachteile dieser Vorgehensweise liegen in den relativ langen generierten Assemblerquelltexten (.s-Files) und der erhöhten Anzahl von Pässen (vier Compilerpässe, zwei Assemblerpässe, Linken). Da die Assemblerquelltexte des M-8-Compilers sich nicht von denen des C-Compilers unterscheiden, ist es naheliegend, nach der Generierung dieser Assemblerquelltexte die vom C-Compiler benutzten Programme zu verwenden. Somit schließt sich an die eigentliche Modula-2-Übersetzung das Assemblieren und Linken an. Es wird kein spezieller Modula-2-Linker, sondern einer der Programmverbinder **sld** oder **gld** des WEGA-Betriebssystems verwendet. Das ist nur ein Grund dafür, daß Übersetzungszeiten, wie sie bei Turbo-Compilern unter CP/M und MS-DOS erreicht werden, für den M-8-Compiler nicht zu erwarten sind. Bei der Implementation des M-8-Compilers wurden diese Nachteile bewußt in Kauf genommen, da diese Strategie ihrer Vorteile wegen von vielen C-Implementationen auf 8-, 16- und 32-Bit-Technik unter verschiedenen Betriebssystemen angewendet wird.

Bezüglich der Übersetzung von Definitionsmodulen wurde die ursprünglich durch Wirth und Jacobi entworfene Strategie der Generierung von Symbolfiles (.sym-File) beibehalten.

Modulinterface

Eine der wesentlichen Stärken von Modula-2 besteht in der separaten Übersetzung. Darunter verstehen wir die Übersetzung von Modulen einschließlich der syntaktischen und semantischen Überprüfung und der Versionskontrollen der Importe während der Übersetzungszeit. Der Test der syntaktischen und semantischen Korrektheit der Importe wird einheitlich von allen bekannten Compilern realisiert. Die Einhaltung der zeitlichen Reihenfolge der Definitionen der Modulschnittstellen (Definitionsmodul – Importe) ist in den einzelnen Implementationen unterschiedlich. Möglich ist der Zugriff auf die Zeit des Betriebssystems, die Verwaltung einer Modula-2-internen Zeit oder die Identifikation der Module durch Kontrollsummen. Im M-8-Compiler wird der letzte Weg beschritten.

Betrachten wir einen Modul m, der aus Modulen m_i importiert. Die m_i können ihrerseits aus Modulen m_{ik} importieren usw., so daß die Importhierarchie einen Graphen bildet, der im Idealfall ein Baum ist. Dieser Graph ist während der Übersetzungszeit bezüglich der zeitlichen Reihenfolge der Interfacedefinitionen kontrollierbar. Während der Linkzeit kann der Modul m_i mit anderen Modulen verbunden werden, die ihrerseits zeitlich überprüfte Importstrukturen (Graph, Baum) besitzen. Die zeitliche Übereinstimmung aller Importe in allen zugehörigen Teilgraphen ist somit während der Übersetzungszeit nicht kontrollierbar. Modula-2 verlangt aber, daß nur dann ein ausführbares Programm entsteht, wenn alle zeitlichen Abhängigkeiten korrekt sind. Daraus folgt die Notwendigkeit einer Überprüfung der Zeitabhängigkeiten während der Link- oder Ausführungszeit. Beide Wege werden in verschiedenen Implementationen beschritten. So kontrolliert der DCP-Modula-2-Compiler /4/ die Zeitabhängigkeiten während der Ausführungszeit. Das belastet insgesamt die Laufzeit, erfordert aber keinen speziellen Linker. Modula-2-Implementatio-

nen, die sich an den m2m-Compiler von Jacobi anlehnen, schaffen eine spezielle Modula-2-Umgebung /5/ mit eigenem Modula-Linker. Diese Modula-2-Linker sind im allgemeinen schneller als universelle Linker. Die Laufzeit der verbundenen Programme wird nicht belastet. Probleme entstehen in solchen Modula-2-Umgebungen, wenn Verbindungen zu Teilen außerhalb dieser Umgebung hergestellt werden sollen.
Der M-8-Compiler generiert in den zu linkenden Objektcode Versionskontrollen. Stimmen benutzte Versionen eines Moduls nicht überein, so führt das zu unaufgelösten Referenzen während des Linkens. Um den Bezug auf einheitliche Versionen herstellen zu können, wurde ein Programm *m2make* geschaffen, das zu einem angegebenen Hauptmodul ein Make-File erstellt. Darunter verstehen wir ein File, welches vom Programm *make* zur Erzeugung eines abarbeitungsfähigen Programms benutzt wird /6/.
m2make kann wahlweise verwendet werden, was der Technologie moderner Modula-2-Implementationen entspricht /2/.

Typdarstellung

Die Standardtypen werden wie folgt dargestellt:

INTEGER	2 Byte
CARDINAL	2 Byte
BITSET	2 Byte
BOOLEAN	2 Byte
CHAR	2 Byte (1 Byte in ARRAY OF CHAR)
REAL	4 Byte in Version 0, 8 Byte in Version 1
ADDRESS	4 Byte
WORD	2 Byte
PROCEDURE	4 Byte.

Bei allen Typen erfolgt keine Drehung der Bytes oder Wörter bei der Speicherung, was bei der Benutzung von überlagernden Variantenteilen in Records von Bedeutung ist. Hier könnten Veränderungen beim Wechsel zwischen den Modula-2-Implementationen erforderlich sein (K 16xx, CM 4, BC 5120). Jede Variable beginnt, bedingt durch die Architektur des Prozessors U8000, auf Wortgrenzen.
Im Typ BITSET werden die Bits von rechts nach links im Wort numeriert. Der Typ SET ist, wie BITSET, auf 16 Elemente beschränkt. Größere Mengen müssen durch Felder von Mengen (ARRAY [0..high] OF (settype)) realisiert werden.
Der Typ BOOLEAN belegt zwei Byte, benutzt in allen Operationen aber nur das niederwertigste Bit b_0. TRUE ist realisiert durch $b_0 = 1$, FALSE durch $b_0 = 0$. Alle anderen Bits des Wortes sind undefiniert.
Eine Variable vom Typ CHAR belegt zwei Byte, wobei das Zeichen im niederwertigen Byte steht. Das höherwertige Byte ist undefiniert. Ein ARRAY [0..high] OF CHAR belegt high + 1 Byte. Danach erfolgt eine Ausrichtung auf Wortgrenzen (Adresse durch zwei teilbar), so daß u. U. intern ein Byte zusätzlich freigehalten wird. Zeichenkettenkonstanten von n Zeichen belegen n + 1 Byte, wobei das letzte Byte die den String begrenzende Null (0C) ist.
REAL-Größen werden als Gleitkommazahlen von vier Byte bzw. acht Byte (Version 1) dargestellt. Ihr interner Aufbau ist identisch mit den im C-Compiler scc benutzten Gleitkommazahlen (float bzw. double).
Der Typ ADDRESS und alle Pointertypen belegen vier Byte. Sie entsprechen der Architektur der segmentierten Version des Prozessors (U8001) und sind auf die Benutzung des vollen Adreßraums des U8001 ausgelegt. Die Verwendung kurzer Adressen ist für Pointertypen, also im Datenteil, nicht möglich. Im Codeteil (Textteil) des übersetzten Programms werden Sprünge innerhalb eines Segments als relative Sprünge (ohne Angabe der Segmentnummer) übersetzt. Der Zugriff zu lokalen Größen erfolgt unter Verwendung segmentierter Adressen mit kurzer Verschiebung (short offset). Programme, die auf einer Gleichsetzung der Typen CARDINAL und ADDRESS basieren, müssen daher überarbeitet werden! Die CARDINAL-Arithmetik ist nicht mehr auf segmentierte Adressen anwendbar. Für die Verschiebung innerhalb einer segmentierten Adresse kann die CARDINAL-Arithmetik weiterhin benutzt werden. Es liegt in der Verantwortung des Nutzers, einen möglichen Adreßüberlauf zu behandeln. Der wahlweise einschaltbare Überlauftest für Größen der Typen CARDINAL und INTEGER kann dafür benutzt werden. Das Überschreiten von Segmentgrenzen ist durch die Benutzung der CARDINAL-Arithmetik auf die Verschiebung innerhalb einer segmentierten Adresse nicht möglich. Routinen zur Adreßarithmetik (IncAddr, DecAddr) und für den Zugriff zur Verschiebung (offset) einer Adresse (TYPE AddrType) liefert der Modul 'P 8System'. Der Adreßaufbau ist in Abb. 1 angegeben. Die Belegung des zweiten Bytes ist undefiniert.

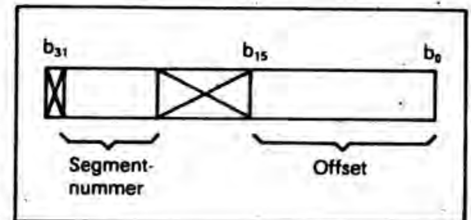


Abb. 1 Segmentierte Adresse mit langem Offset beim U8000

Interne Namenskonventionen

Da der M-8-Compiler Assemblerquelltext generiert und das Ex- und Importkonzept einen wesentlichen Bestandteil der Sprache Modula-2 darstellt, ist es naheliegend, die eigentliche Verbindung der ex- und importierten Objekte durch den Linker herstellen zu lassen. Das führt zur Verwendung globaler (exportierter) Bezeichner im Assemblerquelltext. Möchte man auch von anderen Sprachen importieren bzw. in diese exportieren, so müssen gleiche Konventionen für die Namensgebung auf der Assemblerebene gelten – Bezeichner aus C-Programmen müssen in Modula-2 erreichbar (notierbar) sein und umgekehrt.
Für welche Modulaobjekte wird die Verbindung während des Linkens hergestellt? Konstanten- und Typbezeichner werden während der Übersetzungszeit dem importierenden Programm vollständig verfügbar. Sie gehen damit

nicht in das Interface auf der Assemblerebene ein. Es verbleiben Variable und Prozeduren. Das Herstellen der Verbindung zwischen der Definition der Objekte und den verschiedenen Stellen ihre Benutzung bedeutet für den Linker eine Überarbeitung von Adressen. Für exportierte Variable und Prozeduren werden im Assemblerquelltext Namen der Art
(modulname)(name)
generiert. Exportiert z. B. ein Modul m eine Prozedur p, so entsteht die Assemblerzeile
_m_p::
Ein C-Programm kann die externe Routine m_p aufrufen, wobei die Compiler cc und scc beide den Namen _m_p auf Assemblerebene generieren.

Laufzeitroutinen

Das Laufzeitsystem des M-8-Compilers besteht nur aus wenigen Bytes. Einerseits werden viele Laufzeitroutinen, wie in Modula-2 üblich, durch Standardmodule realisiert. Andererseits werden Routinen der C-Bibliotheken benutzt. So werden die Grundoperationen für reelle Zahlen auf die im **cas** enthaltenen Gleitkommabefehle abgebildet. Arbeitet der P 8000 ohne Arithmetikprozessor, so führt das zu einer Programmausnahme (trap). Die dadurch aufgerufene Interruptserviceroutine ist die Softwarerealisierung der entsprechenden Gleitkommaoperation. Besitzt der P 8000 einen Gleitkommaprozessor oder werden verbesserte Routinen der Realarithmetik für den C-Compiler bereitgestellt, so werden diese ohne eine Änderung am M-8-Compiler oder an den Modula-2-Programmen angesprochen.
Die Standardfunktionen von Modula-2 werden teils als inline-Code generiert (INC, DEC, ABS, CAP, CHR, ORD) oder es werden Laufzeitroutinen der C-Bibliothek angeschlossen (sin, cos usw.).

Bibliotheksmodule

Unter einem Bibliotheksmodul verstehen wir einen in Modula-2 geschriebenen Definitionsmodul einschließlich seiner Implementation in Modula-2 oder einer anderen Sprache (C, Assembler), der zu einem bestimmten Modula-2-System gehört. Obwohl das Bestreben, diese Module zu standardisieren, so alt ist wie die Sprache Modula-2 selbst, stellen reale Implementationen (Jakobi: Lilith; Blach: CP/M; Robotron: DCP; Logitech und JPI: MS-DOS u. a.) im Detail unterschiedliche Bibliotheksmodule zur Verfügung. Wir folgen aus Gründen der Quelltextkompatibilität der Robotron-Implementation für DCP. Ersetzt wurde der betriebssystem- und hardwareabhängige Modul 'DCPSystem' durch den Modul 'P 8System'.
Im einzelnen stehen die Module
P 8System
(implementiert in Modula-2 und Assembler),
UNIXcall
(implementiert in Assembler),
FileSystem
(implementiert in C),
Terminal
(implementiert in C),
Storage
(implementiert in C),
InOut
(implementiert in Modula-2) und
MathLib
(Anschluß von Routinen der C-Bibliothek)
zur Verfügung. Die Notwendigkeit der aus anderen Implementationen verwendeten Module 'TTIO', 'Files' und 'Streams' ist nicht mehr gegeben.

Schnittstellen

Die Schnittstellen eines Moduls in Modula-2 zu anderen Sprachen sind in drei Ebenen realisiert: durch

- den Export oder Import über die Namenskonventionen,
- die Benutzung von CODE-Prozeduren und
- die direkte Bezugnahme auf den generierten Assemblerquelltext.

Wir beziehen uns im folgenden auf die Verbindung zur Sprache C. Die Namenskonventionen wurden bereits beschrieben. Damit ist es möglich, auf exportierte Variable und Prozeduren zuzugreifen. Zu prüfen ist ferner die Darstellung der Datentypen in beiden Sprachen. Es ergeben sich die folgenden Gleichheiten:

Modula-2	**C**
INTEGER	int
CARDINAL	unsign int
CHAR	char
REAL	float (V. 0), double (V. 1)

Eine Wertübergabe über globale Größen ist somit gesichert. Files eignen sich ebenfalls zur Datenübergabe von großen Datenmengen. Werden dazu Files mit dem Elementtyp CHAR verwendet, so entstehen keine Probleme mit der internen Typdarstellung in den Compilern. Hinzu kommt allerdings auf beiden Seiten eine Konvertierung:
⟨ausgangstyp⟩ ↔ ⟨char⟩ ↔ ⟨zieltyp⟩.
Um den Datenaustausch zwischen fremdsprachigen Routinen über Prozedurparameter zu gewährleisten, benutzt der M-8-Compiler die Parameter- und Registerkonventionen des WEGA-Systems (siehe Registerkonventionen).
Der M-8-Compiler läßt CODE-Prozeduren zu, die Assemblerquelltext enthalten. Die zwischen CODE und END stehende Zeichenfolge wird ohne Kontrolle in den Assemblerquelltext übertragen. Er muß also aus gültigen Assemblerinstruktionen bestehen. Für CODE-Prozeduren generiert der M-8-Compiler keinen sogenannten Coderahmen, d. h., die erste Codezeile folgt unmittelbar dem Namen der Prozedur im Assemblerquelltext (Marke). Desgleichen wird kein Code zum Verlassen einer Codeprozedur vom M-8-Compiler generiert. Der Return-Befehl (ret) ist also Bestandteil des Codestücks. Während jede in Modula-2 geschriebene Prozedur einen Lokalitätsbereich auf der Assemblerebene bildet, eröffnen Codeprozeduren keinen eigenen Lokalitätsbereich. Das hat Konsequenzen für die möglichen Assemblerinstruktionen im Körper der Codeprozedur. Insbesondere können (an von der Programmsteuerung nicht erreichbaren Stellen) Konstanten- und Variablendefinition eingefügt werden. Wechsel zwischen Text- und Datensegment (.psec) sind möglich.

Codeprozeduren können Parameter besitzen. Die Parameterübergabe entspricht den Konventionen des WEGA-Systems.
Schließlich kann der generierte Assemblerquelltext eingesehen und gegebenenfalls ediert werden.

Registerkonventionen

Die Register r0 bis r7 des U8000 werden als sogenannte Scratch-Register betrachtet, d. h. ihr Inhalt kann überschrieben werden. Außerdem werden die simulierten Gleitkommaregister f0 bis f3 als Scratch-Register benutzt. r2 bis r7 und f0 bis f3 dienen u. a. der Parameterübergabe. Reichen diese Register zur Parameterübergabe nicht aus, so werden weitere Parameter (im folgenden als Add-Parameter bezeichnet) auf dem Keller übergeben: Für die gerufene Prozedur stehen die Add-Parameter dann unterhalb ihrer Rückkehradresse im Keller.
Den Algorithmus der Zuordnung Parameter–Register definieren wir in zwei Stufen (Tab. 1). Zuerst wird einem Parameter des Typs T und der Art A ein Register bestimmter Länge und Art zugeordnet. r bezeichnet ein 2-Byte-Register, rr ein 4-Byte-Register des U8000.

Was ist das jeweilige nächste freie Register? Die Vergabe beginnt mit den Registern r7 bzw. f3 und erfolgt in Richtung fallender Registernummern. Bleibt durch die Zuordnungsvorschrift ein Register ungenutzt, so wird es auch nachträglich nicht belegt. Verlangt ein Parameter ein Register und kann nach der obigen Zuordnungsvorschrift kein Register gefunden werden, so ist dieser Parameter ein Add-Parameter. Er wird dem Keller übergeben und belegt dort so viele Bytes wie das geforderte Register. Wurde der erste Add-Parameter einer Prozedur ermittelt, so sind auch alle weiteren Parameter dieser Prozedur Add-Parameter. Eventuell noch freie Register für Parameter anderer Typen bleiben unbenutzt.
Zwei Beispiele sollen diesen Algorithmus verdeutlichen:
① Gegeben ist der Prozedurkopf
PROCEDURE p1 (i:INTEGER;

Parameter Typ	Art	gefordertes Register
2-Byte-Typ	Werteparameter	nächstes freies r-Register
ADDRESS	Werteparameter	nächstes freies rr-Register
PROCEDURE	Werteparameter	das nächste freie rr-Register
REAL	Werteparameter	nächstes freies f-Register
ARRAY, RECORD	Werteparameter	nächstes freies rr-Register für Anfangsadresse
open ARRAY	Werteparameter	nächstes r-Register und das nachfolgende rr-Register
bel.	VAR-Parameter	nächstes freies rr-Register

Abb. 3 Zuordnung Parameter–Register

a:ADDRESS; b:BOOLEAN; r:REAL);
Die Registerzuordnung ergibt sich als
i – r7
a – rr4
b – r3
r – f3.
Dem Parameter a wird rr4 zugeordnet, da rr5 kein Doppelregister ist, rr6 (bestehend aus r6 und r7) aber nicht frei ist. r6 bleibt unbenutzt.
② Die Prozedur
PROCEDURE WriteString
(s: ARRAY OF CHAR)
im Modul InOut soll durch eine C-Routine realisiert werden. S benötigt als offenes Feld ein r-Register und ein rr-Register:
s – r7 (Länge)
rr4 (Adresse der Zeichenkette).
Paßfähig dazu ist eine C-Routine mit dem folgenden Kopf
InOut_WriteString(length, s)
int length; char *s;.

Compilerstruktur

Der M-8-Compiler lehnt sich in der Version 0 an den bekannten 4-Paß-Compiler für Modula-2 an. Alle vier Pässe sind in Modula-2 geschrieben und zu einem ausführbaren File verbunden. Dadurch entfällt der sonst erforderliche Overlaymanager und die für das Laden der fünf Überlagerungen erforderliche Zeit (Initialisierung und vier Pässe). Offensichtlich entstehen dadurch auch Uneffektivitäten, z. B. durch gleiche Routinen in sonst verschiedenen Überlagerungen und bei der Benutzung der Interpaßfiles. Die Benutzung der Interpaßfiles wird dahingehend effektiviert, daß die Pufferbereiche vergrößert werden. Das Rücksetzen eines Interpaßfiles beim Übergang vom Paß n zum Paß n + 1 reduziert sich somit zu einem Verwaltungsakt, wenn das Interpaßfile vollständig im Pufferbereich stehen kann. Die Interpaßfiles haben im wesentlichen die gleichen Längen wie bei den bekannten Modula-2-Compilern. Eine geringfügige Verlängerung tritt dadurch ein, daß im M-8-Compiler 4-Byte-Pointer verwendet werden.
Hinsichtlich des mehrfachen Vorhandenseins von Programmteilen wurden in der Version 0 keine Veränderungen vorgenommen. Angestrebt werden in nachfolgenden Versionen die Reduktion dieser Redundanz und eine teilweise Parallelisierung durch Pipes im UNIX.

Coroutinen

Coroutinen werden ab der Version 0 unterstützt. Sie sind nach den bekannten klassischen Verfahren implementiert. Auf die spezielle UNIX-Umgebung wird dabei nicht Bezug genommen.

Modula-2-Umgebung

Die hier beschriebene Modula-2-Implementation besitzt keine abgeschlossene Modula-2-Umgebung. Sie nutzt vielmehr die Werkzeuge des Wirtssystems UNIX und Ideen moderner Modula-2-Implementationen. Eine Gesamtübersicht gibt Abb. 2. Der M-8-Compiler kooperiert mit einem Full-screen-Editor **fse**. Dieser Editor ist WordStar-ähnlich. Mit ihm können, als Erweiterung gegenüber WordStar, Texte in zwei Fenstern bearbeitet werden. Gegenüber den im WEGA-System verfügbaren Editoren **ed**, **vi** usw. besitzt er erhebliche Vorteile,

allerdings auch den Nachteil, daß nur Quelltexte bis zu 64 KByte editiert werden können. Er kooperiert aber insofern mit dem M-8-Compiler, daß letzterer beim Erkennen von Fehlern im aktuellen Verzeichnis ein File mit der Namenserweiterung **.fse** erzeugt. Der **fse** wird danach vom M-8-Compiler aufgerufen, liest das .fse-File und positioniert im Modula-Quelltext auf den ersten Fehler. Ein Fortschreiten von Fehler zu Fehler ist durch die Tastenkombination QN möglich /7/.

Eine weitere Komponente dieser offenen Umgebung des M-8-Compilers ist der Fensteroptimierer **m2opt**. Er arbeitet über dem generierten Assemblerquelltext und ersetzt bzw. streicht Folgen uneffektiver Assemblerinstruktionen, z. B. push-pop-Paare. Da jeweils nur ein kleiner Ausschnitt des zu optimierenden Objekts betrachtet wird (Folgen von zwei bis vier Befehlen), können wir diesen Ausschnitt als ein Fenster betrachten, das fortschreitend über den Text geführt wird (peep optimization). Im Sinne des UNIX ist der **m2opt** ein Filter. Das wiederum gestattet den Aufbau einer Pipe zwischen dem M-8-Compiler und dem Fensteroptimierer, d. h. beide arbeiten als parallele Prozesse im UNIX.

Eine dritte wesentliche Komponente der Modula-2-Umgebung ist das Programm **m2make** zum Generieren eines Make-Files (siehe oben).

Zur Unterstützung des Laufzeittests werden vorerst Trace-Routinen angeboten. Sie unterstützen die Spurverfolgung von Prozeduraufrufen und Rückkehrpunkten auf Quelltextebene. Die interne Realisierung besteht aus zwei Teilen. Erstens werden in den Assemblerquelltext Aufrufe der Trace-Routinen generiert. Durch die Compileroption -trace (-notrace) wird die Generierung von Tracepunkten ein- und ausgeschaltet. Im Quelltext wird das gleiche durch die Pseudokommentare $Q+ und $Q− erreicht. Während der Abarbeitung des so übersetzten Programms werden für eine zu verfolgende Aufrufstelle der rufende Modul, die rufende Prozedur und die Nummer der rufenden Zeile ausgegeben. Eine zu verfolgende gerufene Prozedur liefert den Namen des Moduls in dem sie deklariert ist und ihren Namen. Ein zu verfolgender Rückkehrpunkt einer Prozedur liefert den Modulnamen, den Prozedurnamen und die Nummer der Zeile der Rückkehrstelle. Die zugehörigen Routinen für die Ausgabe befinden sich im Modula-2-Laufzeitsystem. Die implementierte Tracefunktion gestattet so eine auszugsweise Spurverfolgung auf Prozedurniveau. Trace ist auch auf Coroutinen anwendbar, wodurch Laufzeiteigenschaften der Coroutinen und ihr Zusammenspiel verfolgt werden können.

Implementiert sind drei Varianten der Ausgabe der Traceinformationen. In der ersten Variante erfolgt die Ausgabe auf dem Bildschirm. In der zweiten Variante erfolgt die Ausgabe in ein sequentielles File. Die dritte Variante ist für große Programme gedacht, bei denen die letzten Traceinformationen vor dem illegalen Programmende interessieren: die Traceinformationen werden zyklisch in ein File geschrieben. Dadurch wird erreicht, daß der durch die Traceinformationen benötigte Speicherplatz im Filesystem eine bestimmte Schranke nicht übersteigt (generierungsabhängig).

Von Interesse ist weiterhin die Arbeit im hierarchischen Fileverzeichnis des UNIX. Im einfachsten Fall sucht der M-8-Compiler ein zu übersetzendes File und eventuell importierte Objekte (.sym-Files) im aktuellen Verzeichnis. Ist ein voller Pfadname für das Modula-2-Quellprogramm angegeben, so wird es in diesem Verzeichnis gesucht, die .sym-Files im aktuellen Verzeichnis. Existiert im aktuellen Verzeichnis jedoch ein File mit dem Namen 'm8.red', so bestimmt dieses die Suchpfade für die einzelnen Filetypen. Ein mögliches File 'm8.red' ist:

```
.sym  = /z/sym;/z/sysym
.def  = /z/def;/z/sysdef
.s    = /z/mydir
```

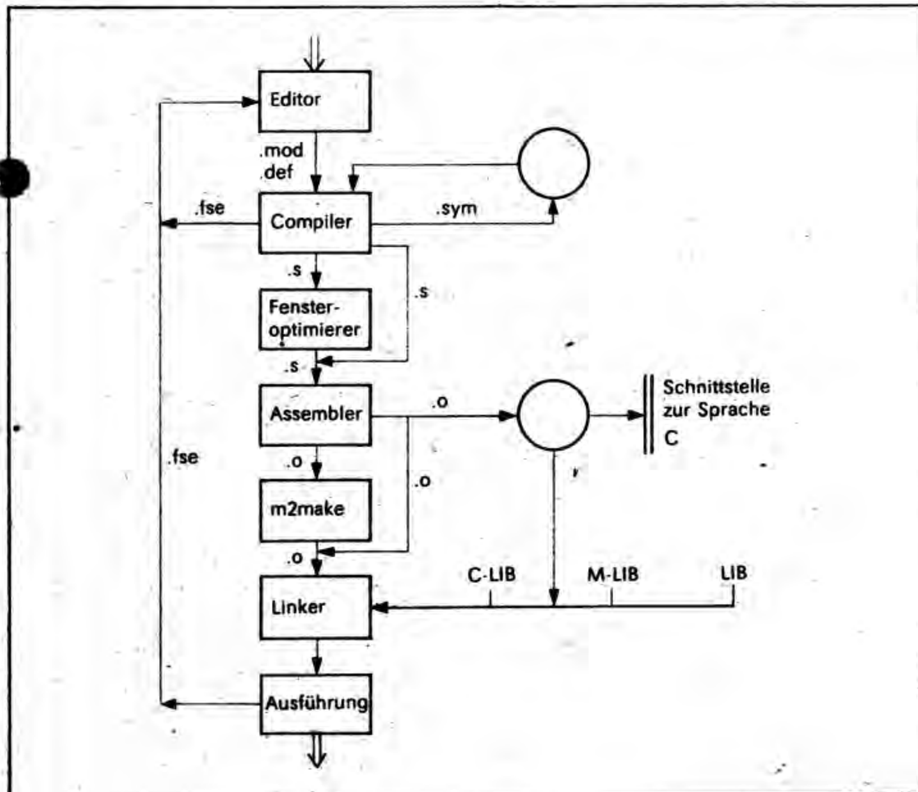


Abb. 2 Das M-8-System

mit folgender Bedeutung: Alle .sym-Files werden zuerst im Verzeichnis /z/sym und danach im Verzeichnis /z/syssym gesucht. Ein .def-File wird zuerst im Verzeichnis /z/def und danach im Verzeichnis /z/sysdef gesucht usw. Auf diese Weise kann der Nutzer seine eigenen Verzeichnisse anlegen und die in ihnen zu suchenden und zu speichernden Filetypen bestimmen.

Interessenten wenden sich bitte an *Dr. sc. Wilfried Grafik bzw. Heinz Werner, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Telefon: 20 93 27 21.*

Literatur

/1/ Schiemangk, H.: Modula-2. Verlag Technik, Reihe Technische Informatik, Berlin 1989

/2/ JPI Modula-2 Owners Handbook. Jemsens & Partners International, London 1987

/3/ Wirth, N.: Revision and Amendments to Modula-2. Publikation der ETH, Zürich 1984

/4/ Modula-2 für DCP. Robotron, Dresden 1987

/5/ Blach, R.: Zur Implementation von Modula-2. Dissertation (A). Humboldt-Universität, Berlin 1989

/6/ Polze, Ch.: UNIX-Werkzeuge. Verlag Technik, Reihe Technische Informatik, Berlin 1989

/7/ Redlich, J.-P.: Ein moderner Editor für UNIX-kompatible Betriebssysteme. edv-aspekte 4/1989

Modula-2-Interpreter für den P 8000

Falk Nisius, Kai-Uwe Scherer
Humboldt-Universität zu Berlin, Sektion Mathematik

Modula-2 ist die Nachfolgerin der Programmiersprache PASCAL. Sie wurde 1970 von Niklaus Wirth an der ETH Zürich entwickelt. Modula-2 hat gegenüber PASCAL Vorteile, vor allem das durchgängige Modulkonzept, das es ermöglicht, die Software für ein Projekt bausteinartig aufzubauen. Die einzelnen Bausteine können abgeschlossen, jeder für sich programmiert werden. Es muß vorher nur festgelegt werden, wie die Objekte (Typen, Konstanten, Variablen und Prozeduren) heißen sollen bzw. wie die Prozedurköpfe aussehen, damit sie später beliebig verwendet werden können. Dies erfolgt im Beschreibungsteil (definition module) jedes Bausteins. Es werden alle zu exportierenden Objekte dieses Bausteins angegeben und meist mittels Kommentaren deren Anwendung erläutert. Im Realisierungsteil (implementation module) steht dann die konkrete Implementierung der Objekte sowie dazu notwendige Importe aus anderen Bausteinen. Die beiden Teile eines Bausteins (Moduls) stehen in getrennten Files.

Für jeden Modul wird ein Zeitstempel mitgeführt, um immer die aktuelle Version zu sichern. Dies hat weitgehende Folgen, sichert aber die Konsistenz der Software. Wenn man in einem zugrundeliegenden Baustein den Beschreibungsteil ändert, müssen alle darauf aufbauenden (daraus importierenden) Bausteine neu übersetzt werden. Das Austauschen von Realisierungsteilen ist problemlos möglich. Für einige grundlegende Bausteine (E/A, Files,...), die maschinenabhängig sind, werden vom Vertreiber die Moduln mitgeliefert. Modula-2-Systeme existieren inzwischen auf fast allen Rechnertypen in der DDR.

Das Übersetzen und Abarbeiten von Modula-2-Programmen geht im allgemeinen in drei Schritten vor sich. Jedes Anwenderprogramm bzw. jeder Beschreibungs- oder Realisierungsteil eines Bausteins wird getrennt übersetzt. Dabei entsteht Objektcode oder ein Symbolfile mit den Informationen aus dem Beschreibungsteil. Anschließend werden mit dem Linker alle zur Arbeit erforderlichen Bausteine (deren Objektcodes) verbunden. Das Abarbeiten des Programms geschieht dadurch, daß es beim Aufruf durch den Loader geladen und gestartet wird. Dies geschieht in der Regie des Modula-2-Systems, d. h., dieses System muß vor dem Übersetzen und Abarbeiten gestartet werden. Es hat seinen eigenen Kommandointerpreter, seinen residenten Teil und sein Laufzeitsystem. Zum Modula-2-System gehört oft noch ein Debugger. Alle wichtigen Teile des Modula-2-Systems (residenter Teil, Grundbausteine, Compiler, Linker, Debugger) liegen als Modula-2-Quelltext vor. Neben allen für höhere Programmiersprachen typischen Elementen enthält Modula-2 auch Programmierelemente tieferer Ebene. Dadurch sind absolute Adressierung, allgemeine Typen, Prozesse, Interrupt-Behandlung und anderes möglich. Somit ist Modula-2 sowohl zum Schreiben von Betriebssystemen als auch von komplexen Anwendungslösungen und Echtzeitverarbeitung geeignet. /1/

Für die Ausbildung stehen vielerorts Rechner des Typs P 8000 zur Verfügung. Dafür waren kein Modula-2-Compiler und -Linker verfügbar. Ein Maschinencode generierendes Modula-2-System befindet sich erst in der Entwicklung. Um ein Praktikum durchführen zu können, mußte eine Lösung gefunden werden, die hinsichtlich Betriebssicherheit und Geschwindigkeit den praktischen Anforderungen genügt.

Realisierung des Modula-2-Systems

Genutzte Quellen

Aus vorangegangenen Implementierungen waren die Quellen des an der ETH Zürich entwickelten, abstrakten M-Code generierenden Modula-2-Compilers vorhanden. Weiterhin stand ein lauffähiges, M-Code generierendes Modula-2-System auf dem Bürocomputer A 5120 zur Verfügung. So lag es nahe, einen M-Code-Interpreter auf dem P 8000 zu entwickeln, so daß das existierende Modula-2-System mit wenigen Anpassungen der E/A-Schnittstellen ohne weitere Änderung übernommen werden konnte. Gestützt wurde diese Entscheidung durch das Vorliegen eines

Rohentwurfs des M-Code-Interpreters in C und der auf dem Bürocomputer unmöglichen Neuübersetzung des Modula-2-Compilers.

Übertragung des Modula-2-Systems

Die ausführliche Beschreibung des M-Codes findet sich in /2/ und /3/. Der Modula-2-Quelltext eines M-Code-Interpreters (ebenfalls veröffentlicht in /2/) war leicht nach C zu übertragen. Für den Test wurde auf dem Bürocomputer eine Anzahl sehr kleiner Modula-2-Programme übersetzt und verbunden. Das Hauptaugenmerk lag dabei auf der Überschaubarkeit der Programme und dem Auftreten ausgewählter M-Codes, um die Korrektheit des Interpreters zu überprüfen. Dabei war es notwendig, die Abarbeitung von M-Code-Instruktionen einzeln zu verfolgen und ihre Wirkungen im Interpreter zu beobachten. Für diesen Zweck wurde ein Debugger entwickelt, der neben den üblichen Funktionen eine Vielzahl von Maskierungen zum Setzen von Unterbrechungspunkten bot. So war es möglich, weit über 90 Prozent der verwendeten M-Code-Instruktionen zu testen. Um den zeitlichen Aufwand gering zu halten, wurden die nicht getesteten M-Code-Instruktionen mit einer Warnung versehen, die bei ihrer Abarbeitung an den Nutzer ausgegeben wird. Bei Programmentwicklungen sowie bei der Nutzung komplexerer Modula-2-Programme ist bisher keiner der maskierten M-Codes generiert und abgearbeitet worden.

Im ursprünglichen Modula-2-System befand sich ein Modul zum Laden von gelinkten M-Code-Files. Dieser Modul wird benötigt, um den residenten Monitor, unter dessen Kontrolle das Modula-2-System läuft, sowie alle anderen Programme und Überlagerungsstrukturen zu laden. Für ein schnelles Laden wurde dieser Modul in C ausgeführt und in das Modula-2-System eingebunden. Hier machte sich die Architektur der wortorientierten M-Code-Maschine negativ bemerkbar. Die Lage der höher- und niederwertigen Bytes in Worten ist auf dem P 8000 anders als in der M-Code-Maschine. Die Ein- und Ausgabe des Linkers und des Compilers ist gemischt wort- und byteorientiert. Die Folge davon ist, daß es nicht ausreicht, in allen Worten der von Linker und Compiler erzeugten Files die Bytes zu vertauschen. Das Auffinden aller Stellen, an denen die Byteanordnung zu belassen war, benötigte etwa die Hälfte der aufgewendeten Arbeitszeit.

Neben dem Lademodul waren die Ein- und Ausgabe für das Terminal und das Filesystem zu erstellen. Für eine hohe Portabilität wäre es vernünftig, die benötigten Routinen in Modula-2 zu formulieren und sie von einem einzigen Modul, welcher die Betriebssystemschnittstellen realisiert, abhängig zu machen. Damit wird die Ein- und Ausgabe natürlich recht langsam. So wurde auch hier der Weg beschritten, alle benötigten Prozeduren in C zu formulieren und sie in das System einzubinden. Die Einbindung von Prozeduren in C in das Modula-2-System erfolgt über spezielle M-Codes. Ein für diese Zwecke vorgesehener M-Code wird durch eine Nummer ergänzt, die der Nummer der gewünschten C-Prozedur entspricht. Die Daten werden aus dem Modula-Stack durch den Interpreter an die C-Prozeduren übergeben. Da die Adressen, mit denen der M-Code-Interpreter arbeitet, verschieden von denen in normalen C-Programmen sind, wurde ein System von Makros geschaffen, welches es gestattet, M-Code-Adressen in C-Adressen umzuwandeln.

Der residente Monitor, unter dessen Steuerung das Modula-2-System arbeitet, wurde als Modula-2-Programm übernommen. Er wird beim Start des Modula-2-Systems geladen und lädt alle weiteren Modula-2-Programme als Überlagerungsstruktur.

In Modula-2-Programmen auftretende Laufzeitfehler führen immer zum Programmabbruch. Der Interpreter behandelt alle Laufzeitfehler durch Angabe des Fehlers, der Prozedurrufkette bis zur Fehlerposition und dem Neustart des residenten Monitors. Alle auf dem Bürocomputer zur Verfügung stehenden Standardmoduln wurden mit eventuell geringfügigen Änderungen in die Implementation übernommen.

Das so entstandene Modula-2-System erfüllte die Erwartungen an Geschwindigkeit und Betriebssicherheit bei weitem. Auf die Implementation eines komfortablen Prozeßsystems wurde verzichtet, da das Arbeiten mit Prozessen erst in späteren Ausbildungsabschnitten vorgesehen ist. Die Übersetzung kleinerer Files ist durch die schnelle Ein- und Ausgabe im Vergleich zum Bürocomputer um eine Größenordnung schneller. Die Abarbeitung des M-Codes auf dem A 5120 durch einen in Assembler geschriebenen M-Code-Interpreter benötigt etwa die vier- bis sechsfache Zeit. Trotzdem wurde der Versuch unternommen, durch Nachbereitung des durch Übersetzung entstehenden Assemblertextes die Geschwindigkeit des Interpreters zu erhöhen. Dabei lohnte es sich nur, den Befehlshole und -dekodierzyklus zu optimieren, da er am häufigsten abgearbeitet wird. Eine Geschwindigkeitserhöhung von 20 Prozent konnte durch das Streichen überflüssiger Befehle erreicht werden. Das Kopieren der Instruktionen zum Befehlsholen und -dekodieren an das Ende jeder Anweisung brachte mit der Einsparung eines Sprungbefehls in jedem Zyklus noch einmal 10 Prozent mehr Geschwindigkeit. Auf weitere Optimierungen wurde verzichtet.

Erfahrungen

Unser für den Interpreter entwickelte Einzelschrittdebugger erwies sich nicht nur in der Testphase als außerordentlich nützlich. Er ermöglicht neben dem Überwachen von Adressen, dem Anzeigen von Speicherbereichen in mehreren Formaten auch das Setzen von Unterbrechungspunkten auf Adressen und beim Auftreten bestimmter M-Code-Folgen.

Der auf dem P 8000 vorhandene Debugger *adb* genügt nur begrenzt den Ansprüchen des Programmierers beim Testen von Programmen. Oft mußte der mühsame Weg über Zwischenübersetzung in Assembler und dem Vereinbaren zusätzlicher Marken gewählt werden, um Fehlerstellen aufzufinden. Bei den recht langen Übersetzungs- und Verbindungszeiten des C-Systems trägt

das nicht zu einer zügigen Programmentwicklung bei. Hinzu kommen Fehler in der Codegenerierung des C-Compilers. Der Optimierer des Systems ist nicht in der Lage, größere Prozeduren, wie sie ein Interpreterkern darstellt, zu optimieren. Hier hilft ebenfalls nur Handarbeit auf Assemblerniveau. Alle Versuche für Schleifen- und Registeroptimierung, auch bei anderen Arbeiten, führten zur Zerstörung der Programmstruktur. Die Optimierung von Schleifen in Interpreterkernen bringt nicht zu unterschätzende Effekte. Das Übertragen größerer Systeme, die auf abstrakten Maschinen arbeiten, ist schnell und bequem möglich. Nachteilig erweist es sich aber, wenn in diesen Systemen interne Darstellungen von der Lage der höher- und niederwertigen Bytes im Maschinenwort abhängen. Letzteres kann unter Umständen eine einfache Übertragung völlig verhindern.

Nutzerbeschreibung

Installation

Für die Installation auf P 8000 genügt das Einspielen des Interpreters und aller benötigten Files von einer Diskette im Tar-Format. Das gesamte System belegt etwa 400 Blöcke im Filesystem. Der Interpreter kann an einer beliebigen Stelle des Filesystems plaziert werden und ist in seiner Ein- und Ausgabe voll an das Betriebssystem WEGA angepaßt. Das Modula-2-System erwartet in einem Verzeichnis /usr/modula2/sys den gelinkten residenten Monitor (unter dem Namen "modula.m2i"). Alle anderen zum System gehörenden Files können in diesem oder im aktuellen Verzeichnis plaziert werden. In den Verzeichnissen /usr/modula2/def, /usr/modula2/mod und /usr/modula2/doc befinden sich die Beschreibung der Standardmoduln sowie die Listen der Compilerfehler, Laufzeitfehler und der erweiterten M-Codes. Ein Manual-Eintrag und ein Newsfile gehören ebenso zur Ausstattung.

Arbeit mit dem Modula-2-System auf P 8000

Der Aufruf des Systems erfolgt mit dem Kommando m2. Es unterstützt traditionell ein Nutzergerät "dk" und ein Systemgerät "sy". Das Gerät "sy" entspricht dem Verzeichnis /usr/modula2/sys, das Gerät "dk" dem aktuellen Verzeichnis. Auf dem Systemgerät erwartet unser Modula-2-System den residenten Monitor. Hier sind auch der Compiler "mcomp.lod" mit seinen Überlagerungsstrukturen, der Linker "mlink.lod" sowie die Objekt- und Symbolfiles der Standardmoduln zu finden. Zu den Standardmoduln zählen: *Clock, Conversions, Exceptions, FileLookup, FileNames, FilePool, Files, Loader, NewStreams, Options, Resident-Monitor, Storage, SystemTypes, TTIO.*
Die Files "wega.lod" zum Aufruf einer C-Shell, "edit.lod" zum Starten eines Editors sowie "end.lod" zum Verlassen des Modula-2-Systems sind hier ebenfalls zu finden.
Das System meldet sich mit einigen Ausschriften, u. a. dem aktuellen Verzeichnis unter DK: und dem Systemverzeichnis unter SY: . Nach dem Erscheinen des Prompt "*" können M-Code-Files abgearbeitet werden. M-Code-Files müssen die Endung ".lod" haben und werden zuerst auf dem Nutzergerät und dann auf dem Systemgerät gesucht. Mit Hilfe des Compilers und des Linkers kann man Modula-2-Programme entwickeln, übersetzen und testen. Die erzeugten Files werden im aktuellen Verzeichnis angelegt. Die Benutzung von Standardmoduln wird durch das Vorliegen ihrer Quelltexte unterstützt. Auf Wunsch ist es möglich, den menügesteuerten Laufzeitdebugger zu benutzen. Für diesen liegt aber keine ausführliche Beschreibung vor. Wenn man ein Modula-2-Programm während der Abarbeitung unterbrechen will, kann man dies jederzeit durch Ctrl\ bzw. DEL erreichen. Man erhält den Punkt der Unterbrechung lokalisiert und befindet sich wieder im residenten Monitor.
Eine Besonderheit besteht im Anlegen von Überlagerungsstrukturen. Die in jedem M-Code-File vorhandene Tabelle der zusammengelinkten Moduln kann vom Linker wegen der Lage der höher- und niederwertigen Bytes nicht ausgewertet werden. Um trotzdem Überlagerungsstrukturen herstellen zu können, muß man das M-Code-File mit einem Filter "aiswab" behandeln, welches die Bytefolge in der Modultabelle für den Linker umdreht.
Zum Ergänzen weiterer C-Prozeduren sind wenige Schritte erforderlich. Nachdem eine Nummer für diese Prozedur ausgewählt wurde, wird die Prozedur in einen Sprungverteiler eingetragen. Die Parameteranpassung erfolgt durch das erwähnte Makrosystem. Die Prozedur ist dann nur noch zum Modula-2-System hinzuzulinken und kann über Inline-Code aufgerufen werden. Am Interpreterkern sind dabei keine Änderungen notwendig.

Das Modula-2-System kann jederzeit kostenlos bezogen werden.
Interessenten wenden sich bitte an
Falk Nisus bzw. Kai-Uwe Scherer, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Telefon: 20 93 29 82.

Literatur

/1/ Schiemangk, H.: Modula-2. Verlag Technik, Berlin 1989
/2/ Wirth, N.: The personal computer LILITH. Berichte des Instituts für Informatik der ETH Zürich, Nr. 40, April 1981
/3/ Jacobi, Ch.: Code Generation and the Lilith Architectur. Diss. ETH Zürich Nr. 7195, 1983
/4/ Blach, R.: Zur Implementation von Modula-2. Diss.(A) Humboldt-Universität zu Berlin, 1989
/5/ Geissmann, L.: A user guide to the Modula-2 System. Institut für Informatik der ETH Zürich, Sept. 1981

Ein moderner Editor für UNIX-kompatible Betriebssysteme

Jens-Peter Redlich
Humboldt-Universität zu Berlin, Sektion Mathematik

Auf fast allen Anwendungsgebieten der Rechentechnik dürften Editoren wohl zu den am häufigsten benutzen Programmen gehören. Dieser Beitrag soll den vor etwa einem Jahr vom Autor entwickelten Editor fse (full-screen-editor) vorstellen. Er ist vollständig in C implementiert und mit relativ geringem Aufwand auf UNIX-kompatible Betriebssysteme portierbar.
Bei seiner Entwicklung wurden folgende Schwerpunkte gesetzt:
• Ausnutzung des gesamten Bildschirmes für die Textdarstellung (d. h. 24 × 80 bzw. 25 × 80 Zeichen)
• Änderungen am Text müssen sofort auf dem Bildschirm sichtbar werden
• die Grundfunktionen müssen leicht erlernbar sein
• parallele Verarbeitung von zwei Dateien in zwei Fenstern
• keine Forderungen an die Struktur der Datei. Das bedeutet, daß Texte mit beliebig langen Zeilen sowie beliebige Nicht-ASCII-Dateien bearbeitet werden können.
Weiter war eine relative Hardwareunabhängigkeit bezüglich des verwendeten Bildschirms und der Tastatur gefordert. Obwohl vorrangig für die Bearbeitung von Quelltexten und Nicht-ASCII-Dateien vorgesehen, läßt sich fse auch sinnvoll für die Textverarbeitung einsetzen. Zusätzlich wird dem Anwender durch eine Vielzahl praktischer Standardaktionen die Arbeit mit dem Editor erleichtert. Der fse ist derzeit für die Betriebssysteme WEGA, VENIX, XENIX vorhanden und umfassend erprobt.

Übersicht

fse stellt Funktionen zum Editieren beliebiger Dateien bereit. Elementare Textverarbeitung wird ebenso unterstützt wie die Verarbeitung von Nicht-ASCII-Dateien oder die Bearbeitung von Quelltexten. Es ist jederzeit möglich, ein zweites Editorfenster für eine weitere Datei zu eröffnen und beide Dateien unabhängig voneinander zu bearbeiten. Blockkommandos und die Möglichkeit der Definition von Metakeys helfen die Arbeit mit dem Editor zu effektivieren. Für die Korrektur von Syntaxfehlern in Quelltexten stehen für die Sprachen C und Modula-2 Kopplungen von Compiler und Editor zur Verfügung, so daß dem Anwender beide Programme als eine Einheit erscheinen.

Zusammenstellung wichtiger Kommandos in der Helpfunktion (^QH):

^E up
^S left
^X down
^D right
^R start of page
^C end of page
^QR start of file
^QC end of file
^QS start of line
^QD end of line
^A word left
^F word right
^QP goto line
^QU exchange location
^QB goto blkbegin
^QK goto blkend
^QE def metakey
^QT help metakey
^UI set margin
^QF find string
^QA replace
^QQ repeat ^QF
^QJ goto last error
^QN goto next error
^B format paragraph
^UL delete file
^UM change directory
^UT change filemode
^QZ save linenumber
^I tabulator
^QI autotab on/off
^QL retake
^QV view current directory
^V insert/overwrite | ascii/hex
^W change window
^Z dump/text
^N new page
^QB backup
^UA new filename
^US save
^UQ quit, not save
^T del word
^O del start of line
^QY del end of line
^UB, ^UK set blockmark
^UH del blockmark
^UC copy block
^UZ compress
^UD save, quit
^UX compr, save
^UU compr, save, quit
^UP forget old, load new
^UN save, load
^UE return to system
^Y del line
(DEL) del left
^G del right
^P ins control
^QM bytemask
^QX displaymode
^UF copy from window
^UY del block
^UV move block
^UR read block
^UW write block
^UJ mark word
^UG mark line.

Fenster

Durch ^W kann zur Bearbeitung einer zweiten Datei übergegangen werden. Beide Fenster teilen sich den Bildschirm und arbeiten vollständig unabhängig voneinander. Anschließend kann mit ^W zwischen den Fenstern gewechselt werden.
^UD, ^UQ usw. schließen das Fenster, in dem sich der Cursor gerade befindet (es ist auch möglich, das zuerst vorhandene vor dem zuletzt eröffneten Fenster zu schließen; dann übernimmt letzteres die Rolle des ersten Fensters).
Bei Aufruf des Editors mit der Option m wird für jedes Fenster ein 64 KByte großer Puffer angelegt. Standardgemäß müssen sich beide Dateien einen 64 KByte umfassenden Puffer teilen. Die Dateilänge ist durch die Größe des Puffers begrenzt. Das heißt, daß fse nur Dateien mit einer maximalen Länge von 64 KByte bearbeitet.

Bearbeitung von Quelltextdateien

Bearbeitungen von Quelltextdateien stellen die häufigste Anwendung des fse dar. Deshalb wird hierfür eine Vielzahl in ihrer Leistungsfähigkeit stark differenzierter Kommandos bereitgestellt. Die einfachsten unter ihnen dienen der Kursorpositionierung, dem Einfügen und dem Löschen von Text.

Die Positionierung des Cursors erfolgt direkt über die Cursortasten oder die dazu äquivalenten Controltasten (siehe Helpmenü). Hinzu kommen Kommandos zur Positionierung des Cursors auf das nächste Wort links (^A) oder das nächste Wort rechts (^F), den Beginn einer Zeile (^QS), das Ende einer Zeile (^QD), den Beginn eines markierten Blockes (^QB), das Ende eines Blockes (^QK), den Beginn einer Seite (^R), das Ende einer Seite (^C), den Beginn der Datei (^QR) sowie das Ende der Datei (^QC). Weiter existieren Kommandos zur Positionierung des Cursors auf eine Zeile mit einer dezimal anzugebenden Zeilennummer (^QP).
^QZ schreibt die aktuelle Zeilennummer (nach Fenstern getrennt) in einen Puffer.
^QU legt die Nummer der aktuellen Zeile im Puffer ab und bewegt anschließend den Cursor in die Zeile mit der Zeilennummer, die zuvor im Puffer stand. War der Puffer leer, so wird ^QP ausgeführt. Die Befehle ^QZ und ^QU sind besonders wertvoll, wenn bei der Bearbeitung des Textes oft zwischen verschiedenen Textstellen gewechselt werden muß (z. B. zwischen Deklarations- und Anweisungsteil in einem Quelltext). Des weiteren sind sie sinnvoll, wenn durch ^QF oder ^QA Zeichenketten gesucht werden, denn anschließend befindet sich der Cursor an einer anderen Textstelle und die alte Ausgangsposition ist oft nicht sofort wiederauffindbar. Hier hilft ^QZ vor und ^QU nach dem Suchkommando. (Wem die Kommandofolge zu lang erscheint, kann sich dafür Metakeys definieren.)
^QF ist ein weiteres nützliches Positionierkommando. Es bietet die Möglichkeit, bis zu 30 Zeichen lange Zeichenketten im Text zu suchen. Die Ausführung kann durch Optionen beeinflußt werden. Als solche sind b für *rückwärts suchen* (standardmäßig vorwärts) und u für *upcase* möglich (letzteres besagt, daß beim Suchen einer Zeichenfolge zwischen Großbuchstaben und Kleinbuchstaben nicht unterschieden wird).
^QA (Suchen und Ersetzen) wertet zusätzlich noch folgende Optionen aus: Eine Zahl, die angibt, wie oft der Befehl automatisch zu wiederholen ist. Wird ein Stern angegeben, so wird der Befehl so oft ausgeführt, bis das Dateiende erreicht wird. Die Option n bewirkt die Unterdrückung von Anfragen vor jedem Austauschvorgang. Ansonsten ist auf diese Fragen mit y (ja, Zeichenkette soll ausgetauscht werden), n (nein, Zeichenkette soll nicht getauscht werden), c (cancel, für Abbruch des Kommandos) oder einem Stern (für Ersetzen ohne Anfragen bei jedem weiteren Auftreten der Zeichenkette im Text) zu antworten.
^QQ wiederholt das letzte Suchkommando (bzw. Suchen und Ersetzen).
Wichtig ist in diesem Zusammenhang, daß beim Suchen eine Bytemaske (siehe ^QM) über die Zeichen gelegt wird, so daß gegebenenfalls auch die Zeichen gefunden werden, die einen internen Code größer als 127 haben (diese treten z. B. bei Texten auf, die mit WordStar bearbeitet wurden; dort dient das höchstwertige Bit der Markierung von Textstellen).

Das Einfügen von Text geschieht in recht einfacher Weise. Das Einfügen einzelner Zeichen wird durch Betätigen der jeweiligen Taste bewirkt. Standardgemäß wird das Zeichen so in den Text eingefügt, daß der Text ab Curserposition um eine Spalte nach rechts rückt und das neue Zeichen dort erscheint, wo zuvor der Cursor stand.
^V schaltet zwischen overwrite-Modus (das heißt, daß von diesem Kommando an der eingegebene Text den bereits in der Zeile vorhandenen überschreibt) und insert-Modus (der neue Text wird in den alten eingefügt) um.
^P, einer beliebigen Taste (z. B. einer Controltaste) vorangestellt, verhindert deren Interpretation als Editorkommando. Statt dessen wird das ihr zugeordnete Byte unvermittelt in den laufenden Text eingefügt. Insbesondere werden mit ^P^I *harte Tabulatoren* in den Text geschrieben. Sie eignen sich hervorragend zum Aufbau von Tabellen. Beim Einrücken nach ⟨ET⟩ oder ^I werden sie an den entsprechenden Stellen aus der vorhergehenden Zeile übernommen, so daß sie nur in der ersten Zeile einer Tabelle eingegeben werden müssen.
Werden nicht direkt angebbare Zeichen (z. B. solche mit einem Code größer 127) benötigt, so kann mit ^Z in den Hexadezimal-Modus gewechselt und dort das gewünschte Zeichen eingegeben werden. Mit ^Z gelangt man wieder in den Ausgangszustand zurück.
^I stellt einen Pseudotabulator dar (meist ist hierfür eine spezielle Taste auf der Tastatur vorhanden). Durch dieses Kommando werden soviele Leerzeichen vor dem Cursor eingefügt, daß er unter dem nächsten Wort der darüberliegenden Zeile zu stehen kommt (geeignet zum Aufbau von Tabellen). Befinden sich in der darüberliegenden Zeile *harte Tabulatoren*, so werden diese anstelle der Leerzeichen eingefügt. Es erfolgt standardmäßig ein Aufruf dieses Kommandos nach jedem ⟨ET⟩, so daß fortlaufend geschriebener Text (z. B. Quelltext) automatisch eingerückt wird.
^QI unterdrückt oder aktiviert (im Wechsel) diesen Dienst. Es ist unbedingt zu beachten, daß durch das Kommando ^I im Normalfall keine echten *(harten)* Tabulatoren eingefügt werden, sondern Leerzeichen.
^QL sei hier noch als ein weiteres Einfügekommendo genannt. Es kann immer direkt nach einem Löschkommando ausgeführt werden und holt den gelöschten Text wieder zurück.

Zuletzt seien noch einige Blockkommandos erwähnt, die ebenfalls Text einfügen können.
^UB definiert den Beginn und ^UK das Ende eines Textabschnittes als Block. Um diesen Arbeitsgang für häufig auftretende Anwendungsfälle zu vereinfachen, sind noch die Kommandos
^UG zum Markieren der Zeile und
^UJ zum Markieren des Wortes, auf dem der Cursor gerade steht, vorhanden.
^UH löscht Blockmarken im aktuellen Fenster. Blockmarken können auch neu gesetzt werden, ohne die alten explizit zu löschen (sie werden dann entsprechend verschoben).
^UC kopiert einen markierten Block an die aktuelle Cursorposition.

^UV ebenfalls, jedoch wird anschließend der markierte Block gelöscht.
^UR fügt ganze Dateien in den Text ein (ab aktueller Cursorposition). Dies ist besonders sinnvoll für Textverarbeitung in Verbindung mit Metakey, z. B. für Briefköpfe, Vordrucke, Formulare.
^UF transportiert einen Block aus dem jeweils anderen in das aktuelle Fenster. Dieses ist die einzige Möglichkeit des direkten Datenaustausches zwischen den Fenstern, wenn einmal von der Möglichkeit auf externe Dateien zu schreiben und wieder von ihnen zu lesen abgesehen wird.

Die dritte Gruppe der simplen Editierkommandos bilden die Löschbefehle. Durch sie werden einzelne Zeichen, Zeichenfolgen oder größere Textabschnitte aus der Datei entfernt (wird direkt anschließend ^QL ausgeführt, so kann der gelöschte Text wieder zurückerhalten werden).
Durch ⟨DEL⟩ wird das Zeichen vor dem Cursor gelöscht; es wirkt auch über Zeilenenden hinweg (die verbleibende Zeile wird an die vorangehende angefügt).
^G löscht das Zeichen am Cursor, wirkt aber nicht über das Zeilenenden hinaus.
^T löscht das Wort, auf dem der Cursor gerade steht. Befindet sich der Cursor auf einem Sonderzeichen, so wird dieses gelöscht. Befindet er sich auf einem Leerzeichen, so werden alle Leerzeichen bis an das nächste Wort heran gelöscht.
^O löscht den Text vom Zeilenanfang bis ausschließlich Cursorposition.
^QY löscht den Text von (einschließlich) aktueller Cursorposition bis Zeilenende.
^UY löscht einen markierten Block.
^UW schreibt einen markierten Textabschnitt in ein externes File. Durch Schreiben eines Blockes auf das File /dev/lp kann ohne großen Aufwand ein Teil einer Datei ausgedruckt werden.

Textverarbeitung

Auch für elementare Textverarbeitung läßt sich der fse sinnvoll einsetzen.
^UI gestattet das Festlegen eines rechten Randes. Die mögliche Spaltenanzahl ist nicht begrenzt, muß aber größer als zwei sein. Ein Wert kleiner als drei schaltet den Rand aus. Bei gültiger Randeinstellung erfolgt ein automatischer Wortumbruch beim Erreichen des Zeilenendes. Dabei werden Leerzeichen so in den Text eingefügt, daß ein gerader rechter Rand entsteht. Einrückungen bleiben erhalten.
^B wird verwendet, um größere Textabschnitte neu nach dem rechten Rand auszurichten. Durch dieses Kommando wird der gesamte Absatz, in dem sich der Cursor befindet, neu formatiert. Zur eindeutigen Kennzeichnung eines neuen Absatzes sollte entweder das erste Wort eingerückt werden oder die Zeile mit einem Controlzeichen beginnen, welches von dem Drucker (oder dem Druckprogramm) ignoriert wird. Oft ist es nötig, bestimmte Controlzeichen in den Text einzufügen, die dann auf dem Bildschirm für Verwirrung sorgen können.
^QX gestattet das Festlegen eines Textdarstellungmodus, in dem Controlzeichen nicht angezeigt werden (mit demselben Kommando kann diese Einstellung auch wieder rückgängig gemacht werden).
Sind bestimmte Textabschnitte häufiger zu schreiben, so hat sich die Benutzung von Metakeys bewährt. Entweder wird der Text direkt eingefügt, oder aber für längere Abschnitte durch einen Metakey das Einfügen einer Datei veranlaßt, die den gewünschten Text beinhaltet.
^N fügt ein ^L (Seitenvorschub) in den Text ein, so daß beim späteren Ausdrucken eine neue Seite begonnen wird. Ein Seitenvorschub wird auf dem Bildschirm von einer durchgehenden Linie gekennzeichnet. Je nach Einstellung (Option L kein Editoraufruf) erfolgt die Zeilenzählung relativ zum Dateibeginn oder relativ zum letzten Seitenvorschub.

Arbeit im Hexadezimaldump-Modus

Wie schon erwähnt, stellt der Editor fse keine speziellen Anforderungen an die Struktur der zu bearbeitenden Datei. Handelt es sich um Nicht-ASCII-Dateien, so wird die Darstellung des Dateiinhaltes als Text wenig Information bieten.
^Z gestattet es darum, den fse in einen Modus umzuschalten, in dem der Dateiinhalt als Hexadezimaldump dargestellt wird. Durch ^V kann der Cursor zwischen der Hexadezimal- und der ASCII-Darstellung bewegt werden. Ansonsten funktioniert alles wie im Textmodus. Speziell sind alle Kommandos zur Cursorpositionierung wirksam; mit folgenden Ausnahmen:
– ^QP positioniert nicht auf eine Zeile, sondern auf eine Adresse
– ^QF sucht nicht Zeichenketten, sondern Bytefolgen (ebenfalls hexadezimal anzugeben).
Des weiteren kann jederzeit Text überschrieben oder an die Datei angefügt werden. Soll eine Zeichenfolge innerhalb der Datei eingefügt werden, so ist ein Wechsel in den Textmodus gestattet. Dort können diese Aufgaben erledigt und anschließend wieder durch ^Z in den Ausgangszustand zurückgekehrt werden. Es ist möglich, den Darstellungsmodus der Datei zu ändern.

```
File:   /bin                                                      Line:     1

0000 | 00 03 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 |  >..............<
0010 | 00 02 2E 2E 00 00 00 00 00 00 00 00 00 00 00 00 |  >..............<
0020 | 00 40 61 64 62 00 00 00 00 00 00 00 00 00 00 00 |  >.@adb.........<
0030 | 00 00 67 65 6C 6F 65 73 63 68 74 00 00 00 00 00 |  >..geloescht1...<
0040 | 00 42 61 73 00 00 00 00 00 00 00 00 00 00 00 00 |  >.Bas..........<
0050 | 00 43 62 63 00 00 00 00 00 00 00 00 00 00 00 00 |  >.Cbc..........<
0060 | 00 00 62 63 6D 70 00 00 00 00 00 00 00 00 00 00 |  >..bcmp........<
0070 | 00 45 62 64 00 00 00 00 00 00 00 00 00 00 00 00 |  >.Ebd..........<
0080 | 00 46 63 61 73 00 00 00 00 00 00 00 00 00 00 00 |  >.Fcas.........<
0090 | 00 47 63 61 74 00 00 00 00 00 00 00 00 00 00 00 |  >.Gcat.........<

^QH help  ^Z text  ^V ascii/hex  ^QP position  ^W window  ^UD save,quit
```

Abb. 1 Beispiel für ein Verzeichnis

In Abb. 1 ist ein Verzeichnis dargestellt. Jede Zeile stellt einen Eintrag dar. Die ersten beiden Byte einer Zeile bilden die Inode-Nummer der Datei. Ist sie 0, so ist der Eintrag ungültig. Die folgenden 14 Byte stellen den Dateinamen dar. Innerhalb des Editors kann die Datei beliebig bearbeitet werden, jedoch gestattet der fse kein Rückschreiben von Verzeichnissen. (Eine Ausnahme gilt für den Superuser. Es ist dabei zu beachten, daß beim Rückschreiben keine Backupdateien angelegt werden.)

Abspeichern der Datei und Beenden des Editierens

Der fse legt beim Laden eine Kopie der Datei im Hauptspeicher an und nimmt alle Veränderungen vorerst nur an dieser Kopie vor. Das heißt, daß sich während der Arbeit mit dem fse das Aussehen der Datei auf dem externen Datenträger nicht verändert.
Mit ^US kann ein Abspeichern der Kopie bewirkt werden. Gewöhnlich wird dabei eine Backupdatei angelegt (alte Datei erhält am Namensende eine Tilde). Letzteres kann jedoch durch die Option −b beim Editoraufruf oder durch das Kommando ^QO zur Laufzeit abgestellt werden.
^UQ verwirft die im Hauptspeicher befindliche Kopie und beläßt die Datei in ihrem ursprünglichen Zustand. Wurden Modifikationen am Text vorgenommen, so wird zur Vergewisserung angefragt: *workfile has been modified, save before quit?* Als Antwort sind hierauf n für nein (d. h. nicht abspeichern, Editierfenster schließen), y für ja (d. h. abspeichern und anschließend Fenster schließen) und c für Abbruch des Kommandos (d. h. es geschieht nichts und fse arbeitet normal weiter) möglich. Sind alle Fenster geschlossen worden, so beendet fse seine Arbeit.
^UD speichert die im Hauptspeicher befindliche Datei ab und schließt das Fenster. Wurde durch ^W ein zweites Fenster eröffnet, so bleibt dieses erhalten. Treten während des Abspeicherns Fehler auf (z. B. Gerätefehler oder fehlende Zugriffsrechte) so wird der Editor nicht verlassen, sondern es erscheint eine Fehlerausschrift; anschließend wird die Arbeit fortgesetzt.
^UA bietet die Möglichkeit, der im Hauptspeicher befindlichen Dateikopie einen anderen Namen zu geben, unter dem sie beim nächsten Savekommando abgespeichert wird. Dabei wird nicht überprüft, ob der Name gültig ist oder ob Zugriffsrechte auf die im Pfadnamen enthaltenen Verzeichnisse bestehen.
Nach Beenden der Eingabe einer Zeile durch ⟨ET⟩ werden gewöhnlich am unmittelbaren Zeilenende stehende Leerzeichen gelöscht. Werden sie jedoch nachträglich in eine Zeile eingefügt, so bleiben sie erhalten und machen die Dateien länger als nötig.
^UZ entfernt überflüssige Leerzeichen am Zeilenende aus der Datei. Dieses Kommando tritt noch in folgenden Modifikationen auf:
– ^UX: Leerzeichen am Zeilenende streichen und Datei abspeichern;
– ^UU: wie ^UX, jedoch danach das Fenster schließen.
Bisweilen geschieht es, daß ein anderer Dateiname als beabsichtigt in der Kommandozeile oder beim Neuladen einer Datei angegeben wird. In diesem Fall ist es nicht nötig, den fse zu beenden und erneut aufzurufen.
^UP ermöglicht jederzeit das Verwerfen der Kopie und das Laden einer neuen Datei. Dabei werden alle Parameter für das jeweilige Fenster zurückgesetzt (z. B. Randeinstellung usw.). Wurden schon Veränderungen am Text vorgenommen, so vergewissert sich der Editor mit der Frage *save bevor quit?* analog zu ^UQ über die Ernsthaftigkeit dieses Kommandos. Soll die bearbeitete Datei abgespeichert und anschließend eine neue Datei in das Fenster geladen werden, so kann dafür das Kommando ^UN verwendet werden. Auch hier erfolgt ein Rücksetzen aller für das Fenster gültigen Parameter auf ihren Standardwert.
^UE ist ein weiteres Kommando zum Beenden des Editors, ohne die Datei abzuspeichern. Der fse kehrt dabei mit einem Rückkehrcode von 1 zum Betriebssystem zurück. Ein make, das den fse aufgerufen hat wird dadurch abgebrochen. So bietet dieses Kommando auch die Möglichkeit, fsc abzubrechen und damit aus dem Zyklus von Übersetzen und Editieren auszutreten. (siehe Zusatzkomponente fsc).

Metakey

Ein ⟨ESC⟩, gefolgt von einem Kleinbuchstaben, ist ein Metakey. Für jeden Kleinbuchstaben des Alphabets befindet sich in einem fse-internen Puffer eine Folge von Zeichen, die nach Betätigung des entsprechenden Metakey in den Eingabestrom eingefügt wird (auch Editierkommandos sind zulässig). Ein Überblick über die Belegung ist durch das Kommando ^QT abrufbar.
^QE gestattet es dem Nutzer, eigene Sequenzen als Metakey zu definieren. Diese sind aber lediglich im gerade aktiven fse gültig. Sollen die Definitionen für spätere fse-Aufrufe verfügbar bleiben, so ist der Editor mit der Option -k aufzurufen. Die aktuellen Definitionen werden dann beim Verlassen des fse in dem File .fse abgelegt (dieses File darf jederzeit gelöscht werden).
Bei der Definition sind folgende Besonderheiten zu beachten:
– Zuerst wird ein Kleinbuchstabe zur Kennzeichnung des Metakey eingegeben. Ihm folgt die gewünschte Tastenfolge.
– ⟨DEL⟩-Taste und Cursortasten liefern nur ihren internen Code, bewirken aber keine Korrektur der Eingabe. Bei Fehlern muß darum die gesamte Definition wiederholt werden. Aus Portabilitätsgründen sollten Aufrufe von Editorfunktionen als Controlsequenz und nicht als Funktionstaste eingegeben werden, denn der Code von Funktionstasten kann von Rechner zu Rechner verschieden sein.
– Beendet wird die Definition mit ^D. Soll ein ^D oder ein ^P in der Sequenz enthalten sein, so ist ihnen ein ^P voranzustellen.

Weitere Editierkommandos und Optionen

Durch eine Vielzahl von Optionen können Parameter des fse gezielt voreingestellt werden. Sie werden beim Aufruf des fse als erstes Argument in der Kommandozeile (als mit einem Minuszei-

chen beginnende Zeichenfolge), angegeben. Bei Mehrfachcodierungen der gleichen Option wird ihr Wert bei jedem Auftreten negiert. Die Reihenfolge der Optionen ist ohne Bedeutung.
Beispiel:
fse -bm100 testfile
Folgende Buchstaben sind zulässig und werden wie folgt ausgewertet:
a: setzt eine Randmarkierung auf die Spaltenposition 66 (siehe Textverarbeitung)
b: unterdrückt das Anlegen von Backupdateien
c: unterdrückt die Ausgabe der Eröffnungsausschrift
e: sucht beim Programmstart im aktuellen Verzeichnis nach einem File 'errmsg' und benutzt die Einträge für die Zuordnung von Fehlerausschriften zu den betreffenden Textstellen (siehe ^QJ und ^QN)
k: bewirkt das Laden und nach Beendigung des fse das Sichern der Metakeytabelle (benutzt File .fse im aktuellen Verzeichnis)
m: legt einen größeren Puffer an (für jedes Fenster 64 KByte)
n: fügt bei ⟨ET⟩ CR LF in den Text ein (MS-DOS Zeilenende). Auf UNIX-Systemen wird gewöhnlich nur LF zur Zeilenendekennzeichnung verwendet.
r: unterdrückt explizit das Laden der zuletzt bearbeiteten Datei (wenn kein Dateiname in der Kommandozeile angegeben war).
y: tauscht logisch die ⟨Y⟩- und die ⟨Z⟩-Taste auf der Tastatur aus.

Die Angabe einer Zahl in der Optionenzeichenkette bewirkt die sofortige Positionierung des Cursors in die Zeile mit der entsprechenden Zeilennummer.
Um ständig benutzte Optionenfolgen nicht immer in der Kommandozeile angeben zu müssen, besteht die Möglichkeit, eine Umgebungsvariable FSE zu definieren (mit setenv in der C-Shell), die als Wert eine Zeichenkette hat. Soweit vorhanden, wird sie bei jedem Start von fse vor die in der Kommandozeile angegebene Optionenzeichenfolge gesetzt. Es ist möglich, durch Angabe der entsprechenden Option in der Kommandozeile eine durch die Umgebungsvariable FSE voreingestellte Option zu neutralisieren.
Einige Parameter des fse können auch zur Laufzeit noch verändert werden. Hier seien die wichtigsten aufgezählt:
^QX legt die Darstellungsweise von Zeichen fest, die laut ASCII nicht darstellbar sind (z. B. Steuerzeichen). Dabei stehen vier Möglichkeiten zur Auswahl: (1) keine Darstellung, (2) Darstellung als Leerzeichen, (3) Darstellung als Punkt, (4) Darstellung als ^Buchstabe für Controlzeichen und als Punkt sonst.
^QM wählt eine Maske aus, mit der die Bytes einer Datei vor der Ausgabe auf dem Bildschirm oder beim Suchen einer Zeichenkette überdeckt werden. Als Masken werden (1) 01111111 und (2) 11111111 angeboten. Standardmäßig ist die erste Maske aktiv. Sie bewirkt (im Gegensatz zur Maske 2, die keine Wirkung auf die Bytes der Datei hat) das Ausblenden des höchstwertigen Bits eines Zeichens. Dieses Bit wird von einigen Textverarbeitungsprogrammen zur Kennzeichnung markanter Textstellen benutzt. Damit ist aber genaugenommen das Zeichen laut ASCII nicht mehr darstellbar und würde sich auch beim Suchen von Zeichenketten (^QF bzw. ^QA) vom ursprünglichen Zeichen unterscheiden, und die Zeichenkette würde nicht gefunden werden. Die Maske (1) schafft hier Abhilfe. Es sei noch einmal erwähnt, daß durch die Maske die Bytes im Text nicht verändert werden. Vielmehr wird diese Maske nur bei der Interpretation des Dateiinhaltes durch den fse angewendet.
^QO legt fest, ob beim Überschreiben bereits vorhandener Dateien durch den fse Backupdateien angelegt werden sollen (das sind Dateien mit einem Dateinamen bestehend aus dem ursprünglichen Namen, gefolgt von einer Tilde; so wird z. B. aus dem File test.c eine Backupdatei test.c~, die den alten Text beinhaltet und test.c ist die Datei mit dem neuen Text). Es wird dabei der gerade eingestellte Modus angezeigt.
^UL bewirkt das Löschen einer Datei.
^UM wechselt das aktuelle Arbeitsverzeichnis des fse. Dieser Befehl hat keinen Einfluß auf das Arbeitsverzeichnis der Shell, von der aus fse aufgerufen wurde. Dort ist nach Beendigung des fse die ursprüngliche Einstellung gültig.
^UT bietet die Möglichkeit, Zugriffsrechte für Dateien zu ändern. Dieses Kommando ist dann sinnvoll, wenn z. B. durch das Nichtvorhandensein von Schreibrechten das Abspeichern der Datei durch das Betriebssystem unterbunden wird. Dieses Kommando wird jedoch nur dann abgearbeitet, wenn der Ausführende auch Eigentümer der Datei ist. Anderenfalls sollte mit ^UA dem File ein anderer Name gegeben werden, so daß es in ein Verzeichnis geschrieben wird, für das die Schreiberlaubnis vorhanden ist.
^QH listet eine Zusammenstellung der wichtigsten Editorkommandos mit kurzer Bedeutungsangabe auf. Sind zwei Fenster aktiv, so wird der Text auf zwei kleineren Seiten dargestellt, die umgeblättert werden. Durch ⟨DEL⟩ kann das Kommando abgebrochen werden. Ansonsten ist durch das Drücken einer beliebigen Taste die Helpfunktion fortzusetzen bzw. zu beenden.
^QT stellt eine Liste der aktuellen Metakey-Definitionen zusammen.
Beim Beenden des fse wird das im aktuellen Arbeitsverzeichnis befindliche File .fse aktualisiert oder angelegt. Dieses File darf jederzeit gelöscht werden. Es beinhaltet Informationen über den Namen und die Position des Cursors der zuletzt bearbeiteten Datei. Wird fse ohne Spezifikation eines Filenamens in der Kommandozeile aufgerufen, so wird versucht, die zuletzt bearbeitete Datei wieder zu laden und den Cursor in die Zeile zu setzen, in der er vor dem letzten Verlassen des fse stand. Die Zeilennummer kann durch explizite Angabe in der Kommandozeile überschrieben werden. Wurde fse mit der Option -k aufgerufen, so wird zusätzlich aus .fse die Tabelle der Metakeydefinitionen geladen und nach Beendigung des fse wieder in diesem File abgelegt.

Die Zusatzkomponente fsc

Jedem Programmierer ist wohl der oft erhebliche Zeitaufwand für die Korrektur von Syntaxfehlern in Quelltexten bekannt. Und wer hat noch nicht mit den

zeitraubenden und ständig wiederkehrenden, oft zeilenfüllenden Kommandofolgen für Aufruf von Compiler und Editor Bekanntschaft gemacht. Dies sollte endlich der Vergangenheit angehören.
fse stellt deshalb einen besonderen Dienst zur Fehlerkorrektur in Quelltexten bereit, der von beliebigen Programmen genutzt werden kann.
Derzeit existiert Software zur Kopplung von C-Sprachübersetzern sowie eine Kopplung mit einem an der Humboldt-Universität zu Berlin entwickelten Modula-2-System. Die Kopplung mit dem C-Compiler soll hier als Beispiel vorgestellt werden.
fsc ist ein selbständiges Programm, das die oben erwähnte Schnittstelle des fse nutzt und die Zusammenarbeit und den Datenaustausch zwischen Compiler und Editor organisiert.
Nach dem Start wird eine erste Übersetzung des Quelltextes vorgenommen. Verläuft die Übersetzung fehlerfrei, dann endet fsc. Treten Syntaxfehler auf, so wird sofort der Editor fse gestartet. Dort enthält die unterste Bildschirmzeile eine kurze Fehlermeldung und der Cursor befindet sich in der betreffenden Zeile. Zusätzlich zum normalen Befehlssatz des fse kann nun mit ^QJ zur vorhergegangenen und mit^QN zur nächsten fehlerhaften Zeile gegangen werden.
Die zugehörige Fehlermeldung befindet sich immer dann in der untersten Bildschirmzeile, wenn der Cursor in einer der fehlerhaften Zeilen positioniert ist. Eventuelle Verschiebungen der Textstellen durch Einfügen oder Löschen von Text werden durch den fse berücksichtigt. Nach Beendigung des Editiervorganges wird erneut der Compiler gestartet. Dieser Arbeitsablauf wird wiederholt, bis die Übersetzung fehlerfrei verläuft. Ist dies nicht erwünscht, so ist fse mit dem Kommando ^UE zu verlassen oder fsc mit ⟨DEL⟩ abzubrechen.
Können Fehler nicht behoben werden, so endet fsc mit dem Rückkehrcode 1, so daß ein make ordnungsgemäß abbricht. In Verbindung mit dem Hilfsprogramm make läßt sich der gesamte Arbeitsablauf weitestgehend automatisieren und kann hier sehr empfohlen werden.

Beispiel für ein makefile:

```
CC = fsc
beispiel:      beispiel1.o beispiel2.o
beispiel3.o
beispiel1.o beispiel1.c beispiel.h
```

Übrigens speichert fse nach dem Kommando ^UD die Datei nur dann ab, wenn der Text wirklich modifiziert wurde, so daß durch make für eine Datei, die zwar geladen, aber nicht verändert wurde, keine erneute Übersetzung gestartet wird.
Interessenten (Installation für die Betriebssysteme WEGA, VENIX, XENIX) wenden sich bitte an

Jens-Peter Redlich, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Telefon: 20 93 29 82.

aspekte-vorschau 1/90

Beratungs-/Expertensysteme

In der heutigen Informatik-Forschung vollzieht sich die Entwicklung von der Informations- zur Wissensverarbeitung.
Expertensysteme bieten aufgrund ihrer Fähigkeiten, gespeichertes Wissen mittels heuristischer Regeln zu Entscheidungsvorschlägen verarbeiten zu können, eine überaus effektive Unterstützung bei der Lösung vielfältigster Problemstellungen in Industrie, Forschung, Ökonomie.
Viele derzeitige Lösungen, die bereits den Namen Expertensysteme tragen, erfüllen diesen hohen Anspruch nur begrenzt. Trotz mancher Erfolgsmeldung existiert für Euphorie kein Grund. Handlungsbedarf ist jedoch anzumelden, Ansätze und einige wenige reife Lösungen sind bereits vorhanden.
In edv-aspekte 1/90 wollen wir mit einer Auswahl von Beiträgen aus den verschiedensten Bereichen unseren Lesern einen Überblick über Stand und Entwicklungsrichtungen von Wissensverarbeitung in der DDR geben und damit anregen zu Diskussionen und zum Austausch zwischen Entwicklern und Anwendern.

Nachtrag zu edv-aspekte 1/89:
Zum Autorenkollektiv des Beitrags *Standards für lokale Netze – Qualitäts- und Produktivitätsanspruch* gehört auch Frau Dr. Sylvia Busch.

Die Redaktion

SPIDER – Ein System zur Datenübertragung

Reinhard Hartung
Humboldt-Universität zu Berlin, Sektion Mathematik

Mit der quantitativen Zunahme von Computern kleiner und mittlerer Leistungsfähigkeit und der Verfügbarkeit weniger, aber leistungsfähiger Geräte erwächst bei vielen Anwendern der Wunsch, diese Computer lokal zu vernetzen, um einen direkten Informationsaustausch der Rechner untereinander zu ermöglichen. Als lokales Netz (LAN) wird ein Informationsübertragungssystem bezeichnet, das in territorial begrenzten Bereichen dem Informationsaustausch zwischen zahlreichen unabhängigen Kommunikationspartnern dient. Einige Hauptanwendungsgebiete sind Industrieautomatisierung, Büroautomatisierung, rechenzentrumstypische Anwendungen, Forschung und Entwicklung, Ausbildung.

Mit dem Einsatz von LAN ergeben sich unter Nutzung spezieller Kommunikationsdienste neue Möglichkeiten wie z. B.:

- Prozeßsynchronisation und Datenaustausch bei Parallelbetrieb
- Nutzung von verteilten Ressourcen
- schneller und problemloser Nachrichten- und Filetransfer zwischen verschiedenen Rechnern unterschiedlicher Ausführung
- entfernter Terminalzugriff
- Nutzung zentraler Datenbankbestände und Serverdienste (z. B. Datei-, Drucker- oder Plotterserver)
- Einrichtung virtueller Geräte (z. B. Laufwerke).

Für den LAN-Anschluß eines Rechners ist ein spezieller Controller (NIU) erforderlich. Leider sind zur Zeit Controller eines leistungsfähigen lokalen Netzes für die Vielfalt der einzubeziehenden Rechentechnik oft nicht verfügbar bzw. ist die Gesamtaufwendung für das Netz ökonomisch unakzeptabel. Aus den genannten Gründen kann es zweckmäßig sein, die sehr verschiedenen, als Personal-, Büro- oder Arbeitsplatzcomputer bezeichneten Geräte geringer Leistungsfähigkeit mit einem Klein-Netz (SAN) zusammenzufassen. Über ein Gateway kann dann die Verbindung zu leistungsfähigen LAN hergestellt werden.

Mit dem Kommunikationssystem SPIDER wurde nach einer Möglichkeit gesucht, verschiedenartigste Rechner kostengünstig zu vernetzen. Dabei wurde wegen der zu berücksichtigenden unterschiedlichen Rechentechnik auf die Verwendung von hardwareabhängigen Netzcontrollern verzichtet und eine Standardschnittstelle benutzt, die für alle Rechner zur Verfügung steht. Im System SPIDER wird die von allen Herstellern angebotene Schnittstelle V.24 (CCITT)/RS-232C (EIA) verwendet.

Da diese Schnittstelle für eine Punkt-zu-Punkt-Verbindung konzipiert wurde und die Leitungstreiber über kein Tristate-Verhalten verfügen, muß eine sternförmige Netzstruktur verwendet werden. Durch den Einsatz der V.24-Schnittstelle sind dem System bezüglich Übertragungsgeschwindigkeit und Entfernung der Teilnehmer Grenzen gesetzt, die jedoch für viele Anwendungsfälle als akzeptabler Kompromiß angesehen werden können. Bei der maximalen Übertragungsgeschwindigkeit des Systems von 38,4 KBit/s konnten Entfernungen von über 100 m erreicht werden.

Der Realisierungskonzeption lagen dabei folgende Prämissen zugrunde:

- Aufbau eines lokalen Datenverbundsystems in einem Rechentechnischen Labor der Humboldt-Universität zu Berlin (HUB).
- Einbeziehung aller zur Zeit verfügbaren und der in den nächsten Jahren zu erwartenden Rechentechnik des Labors. Das bedeutet die Kopplung von Rechnern der 8-Bit-BC/PC-Klasse mit CP/M-kompatiblen Betriebssystemen, der 16-Bit-XT/AT-Klasse unter MS-DOS sowie der P-8000-Rechner unter WEGA.
- Realisierung eines Systems, das den wahlfreien Zugriff zu allen angeschlossenen Teilnehmern ermöglicht.
- Schaffung einer Struktur, die einen gleichzeitigen Datenaustausch zwischen den Kommunikationspartnern zuläßt.
- Benutzung von vorhandenen Rechnerschnittstellen, so daß möglichst ein Umbau der Rechner bzw. deren Nachrüstung mit speziellen Controllern vermieden wird.
- Einbeziehung von Lösungsmöglichkeiten, die den wahlfreien Zugriff zu Druckern erlauben.
- Erweiterbarkeit des Systems zur Einbeziehung in übergeordnete leistungsfähige Kommunikationssysteme mit 32-Bit-Rechnern.
- Beachtung der spezifischen Bedingungen, wie sie bei der gemeinsamen Nutzung des Systems von Forschung und Entwicklung einerseits und der Ausbildung andererseits auftreten.
- Berücksichtigung der derzeitigen Liefersituation von Baugruppen und Bauelementen, die einen kurzfristigen Aufbau und problemlosen Nachbau der Lösung ermöglichen.
- Realisierung eines *offenen Systems* mit einheitlicher definierter Schnittstelle, so daß prinzipiell die Adaption bereits vorhandener Kommunikationssoftware ermöglicht wird.

Kurzcharakteristik

SPIDER ist ein zentraler Controller für ein sternförmiges lokales Datenverbundsystem auf der Basis der Standardschnittstelle V.24. SPIDER kann als V.24-Vermittlungsrechner bezeichnet werden, der sich aus einer ZRE K 2521 (Robotron) und speziellen Vermittlungsbaugruppen (HUB) zusammensetzt.

An einer der Vermittlungsbaugruppen können bis zu acht V.24-Interface angeschlossen werden. Vier derartige Baugruppen sind kaskadierbar, so daß ein lokales Datennetz mit maximal 32 Teilnehmern realisiert werden kann. Neben den beiden Datenleitungen RxD und TxD der V.24-Schnittstelle werden die Steuerleitungen RTS und CTS benötigt. Der Datenstrom des Senders muß sich durch die CTS-Leitung sperren lassen. Die RTS-Leitung wird für den Auf- und Abbau (An- und Abmeldung) der Verbindung benötigt. Die übrigen Steuerleitungen der V.24-Schnittstelle werden nicht benutzt.

SPIDER ermöglicht die Übertragung von Datentelegrammen im Asynchronmodus mit den Raten 9 600, 19 200, 28 800 bzw. 38 400 Bit/s.

Durch die bewußte Festlegung auf eine einfache Standardschnittstelle, die praktisch für alle Rechner angeboten wird, ist es möglich, unterschiedlichste Geräte verschiedener Hersteller ohne nennens-

Signal		Bezeichnung	Anschluß •1)	Anschluß •2)	Richtung	Verw.
V102	GND	Betriebserde	7	A1	⇄	X
V103	TxD	Sendedaten	2	A3	→	X
V104	RxD	Empfangsdaten	3	B4	←	X
V105	RTS	Sendeaufforderung	4	A5	→	X
V106	CTS	Sendebereitschaft	5	B6	←	X
V107	DSR	Betriebsbereitschaft	6	A7	←	(X)
V108	DTR	DTE betriebsbereit	20	B8	→	(X)
V109	DCD	Empfangssignalpegel	8	A9	←	(X)

•1) – Belegung 25-poliger Sub-D-Steckverbinder
•2) – Belegung 26-poliger Steckverbinder bei ASS K 8025 u. a.

Tab. 1 V.24-Anschlußbelegung einer Datenendeinrichtung (DTE)

werten Aufwand in das System zu integrieren.
Außer den genannten Vermittlungsbaugruppen ist der Einsatz einer zusätzlichen Leiterplatte zum Anschluß von acht Druckern mit V.24-Interface vorgesehen. Dabei wird die DTR-Leitung des Druckers zur Steuerung des Datensenders benutzt. Damit ist der wahlfreie Zugriff von 32 Rechnern auf acht Drukker möglich.
In das Gesamtsystem wurden eine spezielle Servicebaugruppe und Testsoftware für eine effektive Inbetriebnahme und Fehlersuche integriert.

Funktionsbeschreibung

Als Steuerrechnerbaugruppe wird die Zentrale Recheneinheit ZRE K 2521 vom Kombinat Robotron eingesetzt.
Da mit dieser ZRE die für den Controller erforderlichen Steuerungsaufgaben gelöst werden können, wurde auf die Entwicklung einer speziellen ZRE verzichtet. Entscheidend für den Einsatz dieser ZRE war neben ihrer prinzipiellen Eignung auch deren hohe Verfügbarkeit.
Die notwendige Serien-Parallel-Wandlung der Daten erfolgt programmtechnisch, so daß auf den Einsatz eines speziellen Schaltkreises für serielle Ein-/Ausgabe verzichtet werden kann.
Die Vermittlungsbaugruppen realisieren die eigentliche Kopplungsfunktion zwischen den Rechnerschnittstellen.
Die ankommenden Datensignale (TxD) werden durch Leitungsempfänger (75154) aufbereitet und vom V.24-Pegel auf TTL-Pegel umgesetzt. Über ein Eingangsmultiplexerfeld gelangen die Daten auf eine Leitung eines 8-Bit-parallelen internen Leitungsbusses und werden dann einem Ausgangsmultiplexerfeld zugeführt. Den Ausgangsmultiplexern sind V.24-Leitungstreiber (75150) nachgeschaltet, die den TTL-Pegel in den erforderlichen V.24-Pegel umsetzen. Die Adressierung der Multiplexerfelder erfolgt über Register, die vom Steuerungsprogramm beschrieben werden.
Weitere Leitungsempfänger bereiten die RTS-Steuersignale für die interne Verarbeitung auf. Der Zustand dieser Steuerleitungen, die dem Auf- und Abbau der Verbindungen (An- und Abmeldung) dienen, werden von der ZRE über Eingabe-Tore abgefragt. Über als Ausgabe-Tore geschaltete Register kann die ZRE die CTS-Signale zur Steuerung der Datensender freigeben. Eine spezielle logische Verknüpfung dieser Signale verhindert die Senderfreigabe (CTS aktiv), wenn die Anforderungsleitung (RTS) inaktiv geworden ist.
Die Druckervermittlungsbaugruppe ermöglicht den direkten Anschluß von Druckern mit V.24-Interface an das Datennetz.
Bei dieser Baugruppe handelt es sich um eine Bestückungsvariante der bereits beschriebenen Vermittlungsbaugruppe. Die Druckervermittlungsbaugruppe verfügt über keinen eigenen Dateneingangsteil, da der Datenstrom nur vom Rechner zum Drucker erfolgt. Die Daten gelangen vom internen Leitungsbus auf diese Baugruppe und werden über einen Bustreiber den Ausgangsmultiplexern zugeführt. Die Datenausgangsstufen sind identisch mit der oben beschriebenen Vermittlungsbaugruppe.
Leitungsempfänger bereiten die Steuersignale DTR1 bis DTR8 der Drucker für die interne Verarbeitung auf. Der Zustand dieser acht Leitungen dient hier jedoch nicht dem Auf- und Abbau einer Verbindung, sondern der Steuerung des Datenstromes zum Drucker. Dazu können diese Leitungen von der ZRE über ein Register abgefragt werden. Ist eine Datenleitung zu einem Drucker durchgeschaltet, so ermittelt das Controllerprogramm den zugehörigen aktiven Partner und steuert dessen CTS-Leitung entsprechend dem Zustand der DTR-Leitung des Druckers.
Als Hilfsmittel für Inbetriebnahme und Fehlersuche ist eine Servicebaugruppe in das System integriert worden.
Für den Anschluß eines Teilnehmers werden die beiden Datenleitungen TxD und RxD sowie die Steuerleitungen RTS und CTS und die Betriebserde benötigt. Als Übertragungsmedium können zum Einsatz kommen:
– vier Paare verdrillter Zweidrahtleitung
– einfach geschirmtes Informationskabel (mindestens fünfadrig)
– vieradriges Informationskabel, jede Ader einzeln geschirmt.
Beim Anschluß von Geräten an den Druckerverteiler werden nur eine Datenleitung und die Steuerleitung DTR benutzt, dementsprechend verringert sich die Anzahl der benötigten Leitungen (Tab. 1).
Bei der Implementation der Software auf verschiedenen Rechnern wurde festgestellt, daß bei einigen Schnittstellen nicht alle Steuersignale zur Verfügung stehen. Des weiteren stellte sich heraus, daß selbst bei gleichen Rechnertypen die Druckertreiber verschiedener Betriebssystemversionen unterschiedliche Steuersignale benutzen. Aus den genannten Gründen ist es erforderlich, die Eingangssteuersignale CTS, DSR und DCD eines Teilnehmers zu verbinden. Damit wird ein einheitliches Anschlußschema bei Verwendung unterschiedlicher Signale erzielt.

Steuerprogrammsystem

Die ZRE K 2521 des Controllers dient zur Steuerung der Vermittlungsbaugruppen. Dabei muß das Steuerprogramm folgende Hauptaufgaben erfüllen:
• Initialisierung des Rechners einschließlich der Erkennung
– der Anzahl der eingesetzten Vermittlungsbaugruppen
– des Einsatzes einer Druckervermittlungsbaugruppe
– der gewählten Übertragungsgeschwindigkeit
– der eingestellten Betriebsart;
• ständige Überwachung der RTS-Steu-

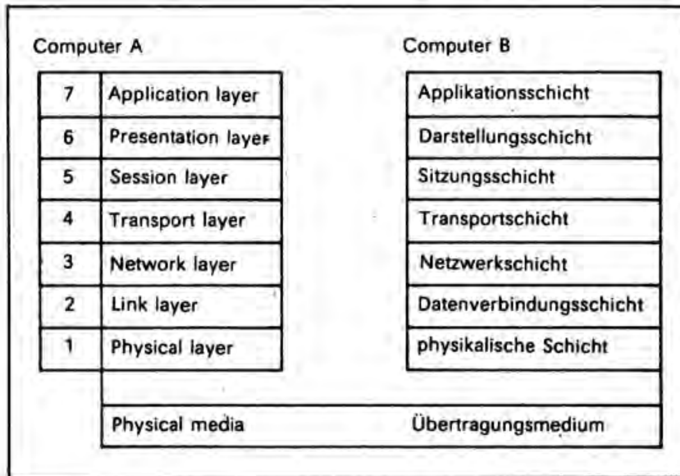
erleitungen zur Erkennung von Forderungen zum Auf- oder Abbau von Verbindungen;
• Aufgaben bei gefordertem Verbindungsaufbau:
– Ermittlung der Stationsnummer des Anrufers
– Ermittlung einer freien internen Datenleitung
– Datenleitung des Senders auf freie interne Leitung schalten
– Kurzzeitige Freigabe des Datensenders mittels CTS zur Übertragung der Stationsnummer des Zielrechners
– Dekodierung der Stationsnummer
– Überprüfen der empfangenen Stationsnummer auf Gültigkeit
– Kontrolle, ob der Zielrechner noch frei ist
– Durchschalten der internen Datenleitung auf Empfängerleitung
– Freigabe des Datensenders mittels CTS;
Der Datensender wird nicht freigegeben, wenn
– keine der acht internen Datenleitungen frei ist
– keine gültige Stationsnummer dekodiert wurde
– der Zielrechner schon durch eine andere Verbindung besetzt ist;
• Aufgaben bei Verbindungsabbau:
– Sperrung des Datensenders über CTS
– Freigabe der benutzten internen Datenleitung
– Verbindungsfreigabe des Zielrechners.

Systemzugriff

Die Anmeldung einer Teilnehmerstation zum Verbindungsaufbau erfolgt durch die Bereitstellung der Stationsnummer des Zielrechners und die Aktivierung der RTS-Leitung. Die Charakterlänge der Stationsnummer muß acht Bit betragen. Paritätsbits sind optional und werden gegebenenfalls ignoriert. Nach Erkennen einer gültigen Stationsnummer wird die Sendedatenleitung (TxD) des Absenders auf die Empfängerdatenleitung (RxD) der Zieladresse durchgeschaltet und der Sender über die CTS-Steuerleitung freigegeben.
Ist die Zielstation bereits durch eine andere Verbindung besetzt, erfolgt keine Freigabe des Senders der anmeldenden Station. Durch den Controller wird jetzt, solange die anmeldende Station die Verbindungsanforderungen über ihr RTS-Signal aufrecht erhält, zyklisch die Zieladresse überprüft und gegebenenfalls die Verbindung nach dem Freiwerden der Zielstation geschaltet. Die Überwachung der Zielstation wird erst durch ein inaktiv gewordenes RTS-Signal beendet.
Eine aufgebaute Verbindung wird ebenfalls durch ein inaktiv gewordenes RTS-Signal des Absenders wieder abgebaut.
Da nach dem Verbindungsaufbau die Datenleitung des Senders direkt physisch mit der Leitung des Empfängers verbunden ist, kann jetzt sowohl der Übertragungsmode als auch die Datenrate geändert werden.
Bei Übertragung der Zielnummer 0 wird die Verbindung auf den eigenen Empfänger (unabhängig von der eigenen Stationsnummer) geschaltet. Damit kann die Verbindung zwischen der eigenen Station und dem Controller getestet werden.
Für den zentralen Controller SPIDER ist die Stationsnummer 254 (FEH) reserviert. Über diese Adresse können Informationen von einer beliebigen Station direkt an SPIDER übertragen werden.
Die Zielnummer 255 (FFH) ist eine globale Adresse. Das bedeutet, daß die Verbindung zu allen Stationen zu schalten ist. Die Verbindungen zur Druckervermittlungsbaugruppe bleiben dabei jedoch unverändert.
Bezüglich des Verbindungsaufbaus sind zwei verschiedene Varianten vorgesehen. Im Mode 0 (Normalbetrieb) wird nur eine Datenleitung vom Sender der anmeldenden Station zum Empfänger der Zielstation geschaltet, während im Mode 1 die Datenrückleitung vom Zielrechner zum anmeldenden Rechner automatisch durchgeschaltet wird, ohne daß die angerufene Station die Stationsnummer an SPIDER überträgt. Die Auswahl der gewünschten Verbindungsart erfolgt über die Adressierung des Zielrechners.
Die an der Druckerbaugruppe angeschlossenen Stationen können selbst nicht aktiv werden. Wurde von einer aktiven Station eine Verbindung zu einer passiven Einheit aufgebaut, so dienen die Leitungen DTR1 bis DTR8 zur Steuerung des Datenstromes von der aktiven zur passiven Einheit, d. h., die DTR-Leitung der passiven Einheit steuert über die CTS-Leitung den Sender der aktiven Station. Eine Datenrückleitung existiert nicht.

Struktur des Kommunikationssystems

International hat es sich in den letzten Jahren durchgesetzt, Rechnerkommunikationssysteme in logischen Schichten zu strukturieren. Generell kann der Entwickler eines solchen Systems dessen Struktur nach eigenem Ermessen gestalten. Damit ist es aber nur schwer möglich, verschiedene Systeme untereinander in Beziehung zu bringen. Die ISO begann im Jahr 1977 mit der Entwicklung einer Architektur für flächendekkende Rechnernetze, die sich allgemein durchgesetzt hat und inzwischen als OSI-Referenzmodell im internationalen Standard ISO 7498 /6/ beschrieben ist. Mit dem Aufbau einer derartigen Kommunikationsarchitektur wird dem Entwickler ein implementationsunabhängiges Modell angeboten, das den konzeptionellen Rahmen für die Gliederung der Kommunikationssoftware festlegt (Abb. 1).
Ein vollständiges Kommunikationssubsystem wird im OSI-Modell in sieben Schichten mit definierten Funktionen unterteilt. Diese, den Schichten zugeordneten Funktionen lassen sich zu den datenübertragungsorientierten (Schicht 1, 2, 3) und datenverarbeitungsorientierten (Schicht 5, 6, 7) Komplexen zusammenfassen. Zwischen beiden Komplexen liegt die Transportschicht, die eine Vermittlungsfunktion zwischen den Aufgaben der oberen Schichten und der gesicherten Datenübertragung übernimmt. In jeder Schicht werden definierte Dienste (Service) angeboten, die bestimmte Kommunikations- und Steuerungsaufgaben erfüllen. Ein Dienst wird der jeweils nächst höheren Schicht an einer gemeinsamen Schnittstelle, dem Dienstzugriffspunkt (SAP) zur Verfügung gestellt. Die Interaktio-



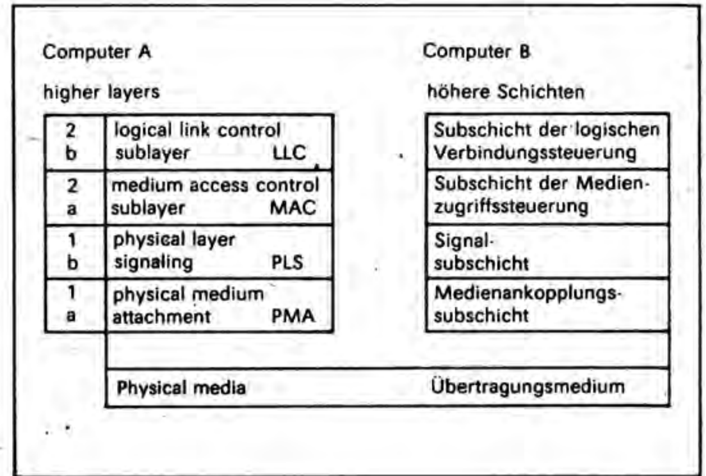


Abb. 1 OSI-Schichtenarchitektur nach ISO 7498
Abb. 2 LAN-Referenzmodell

nen zwischen dem Dienstbenutzer (Service User) und dem Diensterbringer (Service Provider) werden als abstraktes Hilfsmittel in Form von Dienstprimitiven (Service Primitives) beschrieben. Es werden vier Typen von Dienstprimitiven unterschieden:

– *Request:*
Anforderung eines Dienstes
– *Indication:*
Anzeige am SAP des Kommunikationspartners, daß ein Dienst angefordert wurde
– *Response:*
Antwort auf das Indication-Primitiv, das zuvor am selben SAP angezeigt wurde
– *Confirm:*
Bestätigung der Kommunikationsprozedur, die durch das Request-Primitiv angefordert wurde.

Der vollständige Name eines Dienstprimitives setzt sich wie folgt zusammen:

• Initial des Dienstes einer bestimmten Schicht
• Name der Dienstart
• Typ des Dienstprimitives.

Beispiel:
T_DATA_request:
Ein Benutzer der Transportschicht fordert einen Datentransfer an.

Im LAN-Referenzmodell (Abb. 2) werden die unteren beiden Schichten in Subschichten mit speziellen Aufgaben gegliedert.

Für die Realisierung von LAN kann man das Jahr 1980 als den Beginn der Standardisierung mit der Veröffentlichung des Ethernet-Blaubuch von DEC, Intel und Xerox ansehen. Inzwischen existieren eine Reihe weiterer bedeutender Werkstandards aber vor allem auch internationale Standards von ISO, IEEE, IEC, ECMA und CCITT.

In Tab. 2 sind einige wichtige Standards genannt, die sich auf die Schichten 1 und 2 des OSI-Referenzmodells beziehen und sich hinsichtlich des Kanalzugriffsverfahrens unterscheiden.

Unabhängig vom realisierten Kanalzugriffsverfahren wird im Standard ISO/DIS 8802/2 /10/ bzw. IEEE 802.2 /5/ das gemeinsame Protokoll der logischen Verbindungssteuerung (LLC-Subschicht) geregelt. Dieser Standard wurde der Implementation von Gerätetreibern im System SPIDER zugrunde gelegt. Damit steht eine einheitliche und standardgerechte LLC-Nutzerschnittstelle zur Verfügung.

Es ist im weiteren Ausbau des Systems vorgesehen, eine Schnittstelle zwischen den Schichten 4 und 5 des OSI-Referenzmodells nach den Standards ISO 8072 /7/ und ISO 8073 /8/ anzubieten. In dieser Ebene ist der Nutzer von allen Details der Datenübertragungsspezifika befreit.

Datenübertragung

Das verwendete Telegrammformat entstand in Anlehnung an den Aufbau eines MAC-Frames nach dem CSMA/CD-Verfahren entsprechend ISO 8802/3 /11/.

Die von der ISO für die einzelnen Kanalzugriffsverfahren vorgeschlagenen Telegrammrahmen wurden auf ihre Verwendbarkeit für das System SPIDER untersucht. Ein Rahmen nach dem CSMA/CD-Verfahren erweist sich als günstige bzw. technisch realisierbare Variante. Die einzelnen Feldlängen wurden jedoch, wo es möglich war, aus Gründen der Verringerung der Übertragungszeit des Frames gekürzt (Tab. 3).

Das Präambelfeld dient zur Einstellung definierter Verhältnisse in der physikalischen Schicht und zur Synchronisation von Sender und Empfänger. Es besteht aus einer Folge von fünf gleichen Bytes (10101010B – AAH). Daran schließt sich das Byte für die Erkennung der Startbegrenzung (10101011B – ABH) an.

Für die Protokollsteuerinformationen der LLC-Subschicht werden drei Oktetts benötigt. Diese Oktetts enthalten eine Ziel- und Quelladresse (DSAP und SSAP) und ein Steuerfeld (Control). Damit ergibt sich das in Abb. 3 dargestellte Telegrammformat.

Ziel- und Quelladresse (DSAP und SSAP) der LLC-Schicht enthalten keine physischen Adressen des Rechnernetzes, sondern beziehen sich auf die Dienstzugriffspunkte (SAP) dieser Schicht.

Das Steuerfeld entspricht im Aufbau dem des HDLC-Telegrammformates. Es werden Informations- und Überwachungsformate und ein Format für Steuerfunktionen ohne Folgenummern eingesetzt. Abb. 4 zeigt die möglichen Formate des Steuerfeldes.

N (S) und N (R) sind Sende- und Empfangsfolgenummern, die zyklisch den Bereich von 0 bis 7 durchlaufen. Die S-Bits verschlüsseln die Überwachungsfunktionen und die M-Bits sind zur Spezifizierung der Steuerungsaufgaben vorgesehen. Das P/F-Bit (Poll/Final) bewirkt einen Sendeaufruf bzw. eine Endeanzeige.

Das letzte Feld des Telegramms (FCS) dient der Fehlerkontrolle. Die Datensicherung erfolgt durch redundante Codierung zur Erkennung von Übertragungsfehlern. Dazu wird der CRC-Wert des Telegramms, außer der Präambel, dem Startbegrenzer und dem Fehlerkontrollfeld berechnet und im Kontrollfeld übertragen. Zwei der vorgeschrie-

ISO	IEEE	ECMA	
8802/3	802.3	81	– CSMA/CD
8802/4	802.4	90	– Token bus
8802/5	802.5	89	– Token ring
8802/6			– Slotted ring
	802.6		– Metropolitan area network

Field	Bezeichnung	Länge CSMA/CD	Länge SPIDER
Preamble	Präambel	7	5
SFD	Startbegrenzung	1	1
DA	Zieladresse	2/6	1
SA	Quelladresse	2/6	1
LI	Längenangabe	2	2
LLC-PDU	Datenfeld	m	m
Pad	Framefüllung	x	x
FCS	Fehlerkontrolle	4	2

Tab. 2 Auswahl internationaler Standards für die Schichten des LAN-Referenzmodells
Tab. 3 Vergleich der MAC-Frames CSMA/CD und SPIDER

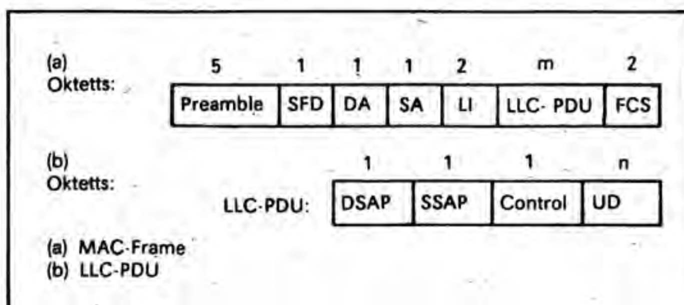
benen Generatorpolynome sind in Tab. 4 enthalten.
Aus Tab. 5 sind einige typische Systemzeiten und aus Tab. 6 ist der prozentuale Zeitverlust durch den Verbindungsauf- und -abbau zu ersehen. Mit einem im Netz installierten Filedienstprogramm, das in TurboPASCAL implementiert wurde, konnten effektiv 100 KByte/min übertragen werden.

Treiber-Schnittstellen

In der LLC-Subschicht werden prinzipiell verbindungslose und verbindungsorientierte Nutzerdienste unterschieden. Insgesamt können vier verschiedene Servicetypen definiert werden /4/:
Type 1 – Verbindungslose Datenübertragung ohne Bestätigung
Type 2 – Verbindungsorientierte Datenübertragung
Type 3 – Verbindungslose Datenübertragung mit Bestätigung
Type 4 – Verbindungslose Anforderung einer Rückantwort
Aus Tab. 7 ist die Einteilung der LLC-Klassen ersichtlich. Für das System SPIDER stehen die Klassen 1 und 2 zur Verfügung /10/.
Dienstprimitivtypen der LLC-Subschicht sind request, indication und confirm. Nachfolgend sind die in SPIDER implementierten Dienstprimitive mit ihren Parametern aufgeführt. Die Parameter local_address und remote_address enthalten die jeweilige physische Stationsadresse und/oder den SAP. Für l_sdu sind ein Pufferzeiger und die Pufferlänge als Parameter implementiert.

Verbindungslos Typ 1:

L_DATA_request _indication	local_address remote_address l_sdu service_class
L_CONNECT_request	local_address remote_address service_class
L_CONNECT_indication _confirm	local_address remote_address status service_class
L_DATA_CONNECT _request _indication	local_address remote_address l_sdu
L_DATA_CONNECT _confirm	local_address remote_address status
L_DISCONNECT _request	local_address remote_address
L_DISCONNECT _indication	local_address remote_address reason
L_DISCONNECT _confirm	local_address remote_address status
L_RESET_request	local_address remote_address
L_RESET_indication	local_address remote_address reason
L_RESET_confirm	local_address remote_address status



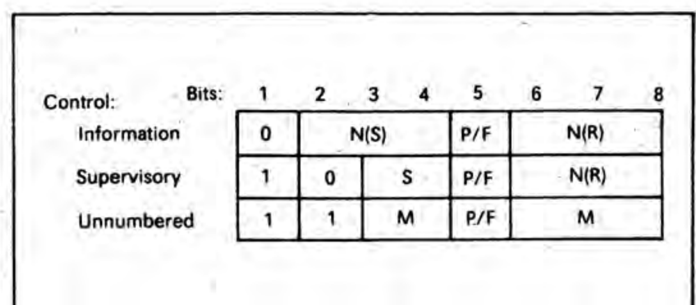


Abb. 3 Telegrammaufbau SPIDER
Abb. 4 Formate des Steuerfeldes

Anwendung Standard	Generatorpolynom
lokale Netze ISO 8802 IEEE 802	$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$
HDLC-Prozedur CCITT, X.25	$G(x) = x^{16} + x^{12} + x^{5} + 1$

Übertragungsrate/KBit/s	9.6	19.2	28.8	38.4
Typ. Gesamtzeit von Verbindungsanforderungen bis zur Verbindungsfreigabe	2.3	1.8	1.6	1.5
Geforderter minimaler Abstand zwischen Abbau und Neuaufbau der Verbindung	3.3	2.8	2.6	2.5
Typ. Gesamtzeit für Auf- und Abbau einer Verbindung	5.6	4.6	4.2	4.0
Gesamtzeit für Übertragung eines LLC-Nutzerdatenblockes mit Telegrammrahmen und Verbindungsauf- und -abbau				
– 128 Datenbyte/Block	154.6	79.1	53.9	41.2
– 256 Datenbyte/Block	287.9	145.7	98.3	75.4
– 512 Datenbyte/Block	544.6	279.1	187.2	141.2

Blockgröße /Byte/	Übertragungsrate /KBit/s: 9.6	19.2	28.8	38.4
128	4.2	6.7	9.1	11.5
256	2.1	3.4	4.6	5.8
512	1.1	1.7	2.3	3.0

Serviceklasse	Type 1	2	3	4
1	X			
2	X	X		
3	X	X	X	
4	X		X	X

Tab. 4 Generatorpolynome zur Fehlererkennung
Tab. 5 Typische Systemzeiten (in ms) bei verschiedenen Übertragungsgeschwindigkeiten
Tab. 6 Zeitverlust (in Prozent) durch den Auf- und Abbau der Verbindungen
Tab. 7 Definition der LLC-Klassen

Initialisierung:

L_init(station_address,rate_code);
station_address: : eigene Stationsadresse
rate_code : Code für Festlegung der Übertragungsgeschwindigkeit Übertragungsrate = rate_code * 9 600 Bit/s

Hardware-Abschaltung:

L_END;
Diese Routine ist vor dem Verlassen des Systems aufzurufen.

Interessenten wenden sich bitte an *Reinhard Hartung, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Tel.: 20 93 22 14.*

Abkürzungen

CCITT Consultative Committee of the International Telegraph and Telephone
CRC Cyclic Redundancy Check
CSMA/CD Carrier Sense Multiple Access with Collision Detection
CTS Clear To Send
DA Destination Address
DCD Data Carrier Detect
DCE Data Communication-terminating Equipment
DSAP Destination Service Access Point
DSR Data Set Ready
DTE Data Terminal Equipment
DTR Data Terminal Ready
ECMA European Computer Manufacturers Association
EIA Electrical Industries Association
FCS Frame Check Sequence
GND Ground
HDLC High-level Data Link Control
IEC International Electrotechnical Commission
IEEE Institute of Electrical and Electronics Engineers
ISO International Organization for Standardization
ISO/DIS ISO/Draft International Standard
ISO/DP ISO/Draft Proposal
LAN Local Area Network
LI Length Indicator
LLC Logical Link Control
MAC Medium Access Control
NIU Network Interface Unit
OSI Open Systems Interconnections
PDU Protocol Data Unit
PLS Physical Layer Signaling
PMA Physical Medium Attachment
RTS Request To Send
RxD Received Data
SA Source Address
SAN Small Area Network
SAP Service Access Point
SDU Service Data Unit
SFD Start Frame Delimiter
SSAP Source Service Access Point
TTL Transistor Transistor Logic
TxD Transsmitted Data
UD User Data

Literatur

/1/ CCITT: Recommendation of the V Series: Data Communication over the Telephone Network. Book, Vol. VIII – Fascicle VIII.1 (1985)
/2/ CCITT: Recommendation X.1 – X.29: Data Communication Networks/Services and Facilities: Interfaces. Book, Vol. VIII – Fascicle VIII.2 (1985)
/3/ DEC, Intel, Xerox: Ethernet: A Local Area Network – Data Link Layer and Physical Link Layer Specification. V. 1.0, Stanford 1980
/4/ Halsall, Fred: Data Communications, Computer Networks and OSI, Addison-Wesley Publishing Company, 1988
/5/ IEEE 802: IEEE Standards for Local Area Networks (1985)
/6/ ISO/DIS 7498: Information Processing Systems – Open Systems Interconnection – Basic Reference Model, ISO/TC97/SC21 (1984)
/7/ ISO/DIS 8072: Information Processing Systems – Open Systems Interconnection – Transport Service Definition, ISO/TC97/SC16 (1984)
/8/ ISO/DIS 8073: Information Processing Systems – Open Systems Interconnection – Transport Protocol Specification, ISO/TC97/SC16 (1984)
/9/ ISO/DIS 8802/1: Data Communication – Local Area Networks – Part 1: General Introduction, ISO/TC97/SC6 (1985)
/10/ ISO/DIS 8802/2: Data Communication – Local Area Networks – Part 2: Logical Link Control, ISO/TC97/SC6 (1985)
/11/ ISO/DIS 8802/3: Data Communication – Local Area Networks – Part 3: CSMA/CD Access Method and Physical Layer Specification, ISO/TC97/SC6 (1985)
/12/ ISO/DIS 8802/4: Data Communication – Local Area Networks – Part 4: Token Passing Bus Access Method and Physical Layer Specification, ISO/TC97/SC6 (1985)
/13/ ISO/DP 8802/5: Data Communication – Local Area Networks – Part 5: Tokenring Access Method and Physical Layer Specification, ISO/TC97/SC6 (1986)
/14/ Lindemann, Bernd R.: Lokale Computernetze. Verlag Die Wirtschaft, Berlin 1988
/15/ Löffler, Helmut: Lokale Netze. Akademie-Verlag, Berlin 1987

Vernetzung mehrerer P 8000 unter WEGA mit UUCP

Dr. Jan-Peter Bell
Humboldt-Universität zu Berlin, Sektion Mathematik

Der folgende Beitrag beschäftigt sich mit der Erprobung und Weiterentwicklung des Programmpaketes UUCP zur Kopplung mehrerer P 8000 unter Steuerung des Betriebssystems WEGA. Es werden Funktionsweise und Kommandostruktur einer verbesserten Version von UUCP mit beschleunigter Einleitung der Datenübertragung, erhöhter Zuverlässigkeit und verbesserter Nutzerschnittstelle vorgestellt.

Arbeitsweise und Komponenten des Programmpaketes UUCP

Das Programmpaket UUCP /1, 2, 3/ gestattet den Transport von Files zwischen direkt miteinander verbundenen Rechnern UNIX (Programm *uucp* – Unix to Unix CoPy) und die Ausführung von Programmen auf benachbarten Rechnern (Programm *uux*– Unix to Unix eXecution), wobei Files ebenfalls von benachbarten Rechnern stammen können. Die im UNIX üblichen Schutzmechanismen sind unter UUCP wirksam. Die Programme *uucp* und *uux* hinterlegen die entgegengenommenen Aufträge in Form von Befehls- und Datenfiles in einem Spool-Verzeichnis, z. B. */usr/spool/uucp*. Anschließend starten sie einen Dämonprozeß für das Programm *uucico* (*uucico* – Unix Unix Copy In Copy Out), der dann den eigentlichen Datenaustausch bzw. die Kommandoübermittlung entsprechend den im Spoolverzeichnis vorhandenen Befehls- und Datenfiles realisiert. Mehrere derartige Dämonprozesse können gleichzeitig auf einem Rechner aktiv sein. Sie synchronisieren sich mittels entsprechender LCK-Files. Sollten auf dem entfernten Rechner Kommandos ausgeführt werden, so wird dort zusätzlich zum Programm *uucico* das Programm *uuxqt* aktiv, das seinerseits mittels Kommandointerpreter *sh* die Abarbeitung der Kommandos steuert. Zur Kommunikation werden Terminalschnittstellen benutzt. Der die Datenübertragung zwischen den verschiedenen Rechnern sichernde Pakettreiber ist in der Regel Bestandteil der Komponente *uucico*.

Das Programmpaket UUCP ist primär für die Kommunikation über verschiedenartige, langsame Übertragungsleitungen konzipiert worden. Insbesondere wird die Möglichkeit von Wahlleitungen unterstützt (nicht bei der Variante des VEB KEAW). Dies hat zur Folge, daß bei jeder neuen Aktivierung des Programms *uucico* auf einem anderen Rechner die Verbindung zu diesem neu aufgebaut werden muß, d. h., es wird eine vollständige Prüfung der Zulässigkeit der Verbindung durchgeführt, eine LOGIN-Prozedur auf dem Terminalkanal zum entfernten Rechner ausgeführt und das Programm *uucico* auf dem anderen Rechner gestartet.

Erfahrungen bei der Benutzung eines Standard-UUCP-Pakets

Das UUCP-Paket des VEB KEAW Berlin wurde entsprechend der in /3/ gegebenen Anleitung im März 1988 unter dem Betriebssystem WEGA 3.0 installiert. Einige wenige Exemplare zeigten die prinzipielle Funktionsfähigkeit des UUCP-Paketes. Es wurde mit den Übertragungsgeschwindigkeiten 9 600 Baud und 19 200 Baud gearbeitet. Beide Geschwindigkeiten arbeiteten problemlos, allerdings brachte die doppelte Geschwindigkeit nur eine um 40 Prozent höhere Datenübertragungsrate (etwa 700 Byte/s). Nach den ersten Experimenten wurde der in Abb. 1 dargestellte Rechnerverbund aufgebaut.

Die Verbindungen zwischen den Rechnern P 8000 und den Bürocomputern BC 5120 wurden über IFSS-Kanäle realisiert und mit dem Programmpaket REMOTE betrieben. Die Verbindung zwischen dem Rechner P 8000 und dem Personalcomputer IBM/PC-XT wurden über eine V.24-Schnittstelle sowohl mit dem Programmpaket REMOTE getestet als auch später mit dem Programmpaket UUCP genutzt. Die Rechner P 8000 wurden untereinander ebenfalls über V.24-Schnittstellen verbunden und mit dem Programmpaket UUCP genutzt. Der so aufgebaute Rechnerverbund wurde nun intensiv getestet. Dabei zeigten sich bezüglich des Programmpakets UUCP folgende Probleme:

- Das Zeitverhalten war nicht akzeptabel. Die Übertragung selbst kürzester Files zwischen zwei unbelasteten Rechnern dauerte über eine Minute.
- Die Übertragung von Files zwischen nicht benachbarten Rechnern war unmöglich (-e Option funktioniert nicht). Nach jeder Übertragung war der Synchronisationszähler in dem File */usr/lib/uucp/SQFILE* falsch gestellt, so daß alle weiteren Datenübertragungen verhindert wurden. Der Einsatz des Programms *uusend* aus dem UUCP-Paket von UNIX-BSD 4.2 ermöglichte zwar den Datentransport zwischen beliebigen Rechnern, wies aber Fehler bei der Übertragung von Nicht-ASCII-Files auf.
- Das von der Komponente *uucico* benutzte Verfahren zum Deaktivieren der Terminalkanäle arbeitet nicht korrekt. Unter bestimmten Umständen, die nicht verifizierbar waren, wurde die Eintragung mit dem File */etc/inittab* für das benutzte Terminal verfälscht, so daß eine weitere Arbeit mit UUCP über diesen Terminalkanal ohne manuelle Korrektur durch einen Superuser unmöglich war.
- Unter bestimmten Umständen wird der Druckerspooler in einen undefinierten Zustand gebracht, aus dem er nicht mehr zu aktivieren ist ohne Neustart des WEGA-Systems (Ursache ist unbekannt).

Insgesamt muß festgestellt werden, daß das vom VEB KEAW ausgelieferte UUCP-Paket, das im wesentlichen dem UUCP-Paket der UNIX-Version 7 entspricht, für den oben skizzierten Einsatzfall nur bedingt anwendbar war. Dies gab Veranlassung, das UUCP-Pa-

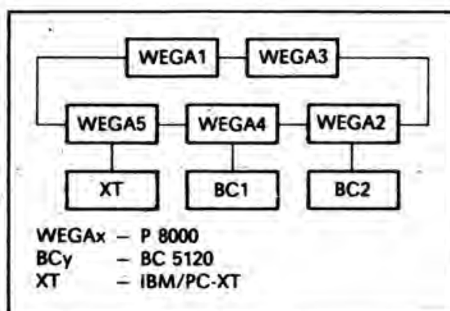


Abb. 1 Rechnerverbund von Rechnern des Typs P 8000, BC 5120 und IBM/PC mittels V.24 und IFSS

ket genauer zu untersuchen. Das REMOTE-Paket für 8-Bit-Rechner unter CP/M arbeitete nach Modifizierung der Portadressen entsprechend der eingesetzten Hardware problemlos und zuverlässig. Die Übertragungsgeschwindigkeit entspricht den Erwartungen bezüglich IFSS-Schnittstellen mit 9 600 Baud. Das REMOTE-Paket für MS-DOS arbeitete ebenfalls zufriedenstellend. Die Kopplung mittels Programmpaket UUCP an einen unter UNIX laufenden IBM-PC/XT war ebenfalls möglich.

Modifikation des Programmpaketes UUCP

Nähere Untersuchungen des UUCP-Paketes an Hand der Quelltexte der UNIX-Systeme Version 7 und Version BSD 4.2 zeigten folgende Ursachen für das schlechte Zeitverhalten auf:

- Auch bei Standleitungen wird bei jedem Neustart des Programms *uucico* eine neue Verbindung aufgebaut (siehe oben). Dies bedingt durch die gewählte Realisierung bei UUCP (csh-Prozedure) einen Zeitaufwand von bis zu einer Minute.
- Jede Datenübertragung mittels Programm *uucp* und jede Kommandoübertragung mittels Programm *uux* bedingt einen Neustart des Programms *uucico* auf dem Partnerrechner mit allen oben beschriebenen Aktivitäten.
- Die Systempfadnamen bei der Datenübertragung zwischen nicht benachbarten Rechnern sind notwendig, da in keinem Rechner Informationen über die Übertragungskanäle und Systemnamen entfernter (nicht benachbarter) Rechner enthalten sind.

Ausgehend von der Voraussetzung, daß bei vielen Installationen ausschließlich festgeschaltete Standleitungen benutzt werden, also ein Neuaufbau der Verbindungen für jede Übertragung nicht zwingend notwendig ist, wurde ein neues Programm für *uucico unetio* entwickelt, das die Aufgaben von *uucico* übernimmt aber ständig die Verbindung über einen Terminkanal zu einem benachbarten Rechner aufrecht erhält. Dadurch konnte die Reaktionszeit wesentlich verkürzt werden. Weiterhin wurden neue Systemfiles */usr/lib/uucp/L route* und */usr/lib/uucp/L mail* geschaffen, die die Systempfadnamen zu entfernten Rechnern und Nutzern enthalten, so daß der Nutzer jeweils bei der File- und Nachrichtenübertragung mit Zielsystemnamen bzw. Zielnutzernamen arbeiten kann. Um die neuen Möglichkeiten effektiv nutzen zu können, waren geringe Veränderungen an den Programmen *uccop, uux, uuxgt, uusend* und *mail* notwendig. Die modifizierten Programme von UUCP sind ebenfalls in der Lage, mittels Programm *uucico* über Wahlleitungen entsprechend dem ursprünglichen Modus zu arbeiten.

Das Programm unetio

Das Programm *unetio* dient zum Datentransport zwischen benachbarten Rechnern. Es muß dazu auf beiden Rechnern jeweils für den Partnerrechner aktiviert sein. Der Aufruf des Programms *unetio* erfolgt durch folgendes Kommando:

unetio [–x⟨n⟩] -s⟨Systemname⟩.

Dabei bedeuten –x⟨n⟩ die gewünschte Debugging-Ebene (⟨n⟩ = 1 Kurzprotokoll und ⟨n⟩ = 8 ausführliches Protokoll) und -s⟨Systemname⟩ spezifiziert den Namen des benachbarten Rechnersystems. Alle weiteren Informationen für den Aufbau der Verbindung werden aus den Files */usr/lib/uucp/L. sys, /usr/lib/uucp/L-devices* und */usr/lib/uucp/USERFILE* geholt. Die LCK-Files */usr/spool/uucp/LCK..⟨Terminalkanal⟩* und */usr/spool/uucp/LCK..⟨Systemname⟩* werden ebenfalls angelegt. Falls sie bereits existieren, wird entsprechend den bei UUCP üblichen Regeln verfahren.

Das Programm *unetio* sollte als Dämon gestartet werden. Dazu steht das Kommando:

uustart ⟨Systemname⟩ {⟨Systemname⟩}

zur Verfügung, das der Reihe nach für die angegebenen Systeme jeweils einen Dämon mit „unetio -s⟨Systemname⟩" startet. Falls LCK-Files existieren, werden diese durch das Programm *uustart* vorher gestrichen.

Im folgenden sei die Arbeitsweise des Programms *unetio* kurz dargestellt. Nach dem Start wird als erstes an Hand des Systemnamens der zugehörige Terminalkanal bestimmt. Die LCK-Files werden überprüft und entsprechend angelegt bzw. *unetio* verlassen. Die LCK-Files enthalten neben der üblichen Prozeßnummer auch noch ein Flagwort, das anzeigt, ob mit Standleitung (*unetio*) oder Wahlleitung (*uucico*) gearbeitet wird. Anschließend wird der zugehörige Terminalkanal eröffnet und initialisiert. Danach geht *unetio* in seinen Grundzyklus über.

Der Grundzyklus beginnt mit der Durchsuchung des Fileverzeichnisses */usr/spool/uucp* nach Aufträgen für den Daten- bzw. Befehlstransport zum angegebenen Partnerrechner. Wird ein entsprechender Auftrag gefunden, geht *unetio* in den Mastermodus über und beginnt mit der Datenübertragung zum Partnerrechner. Wird kein Auftrag gefunden, geht das Programm in den Slavemodus über. Im Slavemodus wartet *unetio* auf das Eintreffen von Daten vom Partnerrechner auf dem zugeordneten Terminalkanal bzw. auf das Eintreffen von Aufträgen vom lokalen Rechner für den Partnerrechner. Dieser Wartezustand wird realisiert, in dem ein Signal SIGINT zugelassen wird und gleichzeitig versucht wird, über den Terminalkanal Daten zu lesen. Werden entsprechende Daten vom benachbarten Rechner angeboten, so wird das Signal SIGINT gesperrt und nach einem Handshake in den normalen Übertragungsmodus mittels Pakettreiber übergegangen (Slave), der dem des Programms *uucico* vollständig entspricht. Wird vom lokalen Rechner eine Daten- oder Befehlsübertragung gefordert (Signal SIGINT für *unetio*), so wird der Wartezustand von *unetio* mittels eines Signals SIGINT unterbrochen und *unetio* sucht in dem Fileverzeichnis */usr/spool/uucp* nach entsprechenden Aufträgen. Werden Aufträge gefunden, so geht das Programm in den Mastermodus über. Andernfalls bleibt es im Slavemodus und wartet weiter.

Im Mastermodus wird nach einem erfolgreichen Handshake mit der eigentlichen Datenübertragung mittels Pakettreiber begonnen. Versuchen beide Systeme gleichzeitig im Mastermodus die

Übertragung zu beginnen, so wird dies erkannt und ein System geht in den Slave-Modus über. Liegen auf beiden Rechnern keine Aufträge für den benachbarten Rechner vor, so geht jeder wieder in den Slavemodus über und wartet. Das Programm *unetio* muß auf beiden benachbarten Systemen vor Beginn der ersten Übertragung jeweils für den benachbarten Rechner gestartet werden (siehe oben).

unetio ist so konzipiert, daß es beim Auftreten von Fehlern wieder zum Anfang des Grundzyklusses zurückkehrt. Vorher wird jeweils der Terminalkanal neu initialisiert. Dadurch ist gewährleistet, daß selbst bei Hardwarefehlern eine Fortsetzung der Arbeit nach Behebung des Fehlers möglich ist. Tritt im Mastermodus ein Fehler mehr als dreimal unmittelbar hintereinander auf, so geht *unetio* trotz vorliegender Aufträge in den Slavemodus über und wartet auf Daten des Partnerrechners bzw. auf neue Aufträge. Diese Situation tritt in der Regel nur bei permanenten Hardwarefehlern auf.

Mit Hilfe des Kommandos
uupoll ⟨Systemname⟩
kann der Nutzer einen für das System ⟨Systemname⟩ gestarteten Dämon des Programms *unetio* nach einem technischen Fehler veranlassen, den Wartezustand (Slave) zu verlassen und das Verzeichnis */usr/spool/uucp* nach Aufträgen zu durchsuchen, gegebenenfalls in den Mastermodus überzugehen und mit der Übertragung zu beginnen.

Die Programme uusend und uuget

Die Programme *uusend* und *uuget* ermöglichen das Senden bzw. Empfangen von Files zu bzw. von beliebigen Rechnern, die über UUCP erreichbar sind. Der Nutzer muß dabei lediglich als Ziel- bzw. Quellsystempfand den Namen des Ziel- bzw. Quellsystems angeben. *uusend* bzw. *uuget* bestimmen die entsprechenden Systempfade selbständig an Hand der Eintragungen im Route-File */usr/lib/uucp/L route*, wenn der angegebene Systemname nicht in dem File */usr/lib/uucp/L. sys* als benachbartes System aufgeführt ist.

Das Programm *uusend* kann folgendermaßen aufgerufen werden:
uusend ⟨Quellfilename⟩
⟨Systempfad⟩ ⟨Zielfilename⟩
Dabei bedeuten
⟨Quellfilename⟩:
Filename (Pfadname absolut oder relativ) des zu sendenden Files auf dem lokalen Rechner. Es darf nur genau ein File spezifiziert werden.
⟨Systempfad⟩:
⟨Systemname⟩"!"{⟨Systemname⟩"!"}.
Es darf sowohl der Name des Zielsystems als auch der komplette Systempfad zum Zielsystem angegeben werden.
⟨Zielfilename⟩:
⟨absoluter Pfadname⟩ |
"~" ⟨Nutzername⟩"/"⟨relativer Pfadname⟩.
⟨Nutzername⟩ ist der Name des Nutzers auf dem Zielrechner. Der relative Pfadname bezieht sich dann auf das Home-Directory des spezifizierten Nutzers.
Verweist der Basisname des Files auf ein Fileverzeichnis, so wird der Basisname des Quellfiles als Basisname des Zielfiles benutzt. Achtung! Das Verzeichnis, in dem das File abgelegt werden soll, muß die Zugriffsrechte „lesen und schreiben für alle" haben.
Das Programm *uusend* benutzt intern weitere Optionen und Parameter, um die Datenübertragung zu organisieren:

-m ⟨ooo⟩ – ⟨ooo⟩ Zugriffsrechte für das zu übertragende File.
-f⟨name⟩ – Basisname des zu übertragenden Files.
-x – Debug-Ebene.
-r – -r Option für uux.

Die Möglichkeit der Angabe eines Zielsystems oder eines Zielsystempfades wurde zugelassen, um die Aufrufkompatibilität zum UUCP des Systems UNIX BSD 4.x zu sichern.

Das Programm *uuget* kann folgendermaßen aufgerufen werden:
uuget ⟨Systempfad⟩⟨Quellfilename⟩
⟨Zielfilename⟩
Dabei bedeuten
⟨Quellfilename⟩: ⟨absoluter Pfadname⟩|
"~"⟨Nutzername⟩"/"⟨relativer Pfadname⟩.
⟨Nutzername⟩ ist der Name des Nutzers auf dem Quellrechner. Der relative Pfadname bezieht sich auf das entsprechende Home-Directory. Es darf genau ein File spezifiziert werden.
⟨Systempfad⟩:
⟨Systemname⟩"!"{⟨Systemname⟩"!"}.
Es darf sowohl der Name des Quellsystems als auch der komplette Systempfad zum Quellsystem angegeben werden.
⟨Zielfilename⟩:
⟨absoluter Pfadname⟩|
"~"⟨Nutzername⟩"/"⟨relativer Pfadname⟩.
⟨Nutzername⟩ ist der Name des Nutzers auf dem Zielrechner. Der relative Pfadname bezieht sich auf das entsprechende Home-Directory. Ist der Zielfilename ein Fileverzeichnis, so wird der Basisname des Quellfiles als Basisname des Zielfiles benutzt. Das Verzeichnis, in dem das File abgelegt werden soll, muß die Zugriffsrechte „lesen und schreiben für alle" haben.
Das Programm *uuget* benutzt zur internen Steuerung folgende weitere Optionen und Parameter:

-s⟨Systempfad⟩ – Systempfad, auf dem das Kommando uuget zum Quellrechner transportiert wird
-d⟨System⟩ – Name des Systems, zu dem das File transportiert werden soll.

Die Shell- bzw. C-Shell-Sonderzeichen müssen beim Aufruf der Programme *uusend* und *uuget* gegebenenfalls mit "\" maskiert werden.
Die Programme *uusend* und *uuget* müssen auf allen Systemen vorhanden sein, die in den jeweiligen Systempfaden enthalten sind. Das Programm *uuget* benutzt auf dem Quellsystem das Programm *uusend*, um das zu übertragende File zum Zielsystem zu übertragen. Da bei der Benutzung von *uuget* das Kommando zweimal über den angegebenen Systempfad transportiert werden muß (hin nur das Kommando, zurück Kommando und Daten), benötigt das Kommando *uuget* mehr Zeit als das Kommando *uusend* für den Transport des gleichen Files zwischen zwei beliebigen Rechnern.

Das Route-File */usr/lib/uucp/L route* enthält für jedes in den UUCP-Verbund integrierte System eine Zeile mit folgender Struktur:
⟨Systemname⟩ {⟨Systempfad⟩}\!
Dabei bedeutet ⟨Systempfad⟩::=⟨Systemname⟩{"\!"⟨Systemname⟩}.
Das heißt, für jedes beteiligte System muß ein entsprechender Eintrag vorhanden sein, auch für das lokale System. Für das später beschriebene Testsystem sollten noch folgende Eintragungen enthalten sein:
*⟨Zielsystemname⟩ ⟨Systempfad⟩.
Dabei bedeutet ⟨Systempfad⟩::=⟨Systemname⟩"\!"{Systemname"\!"}.
Der letzte Systemname im Systempfad muß dem Zielsystemnamen entsprechen. Ein Beispiel für ein Route-File ist in Abb. 2 dargestellt. Die nähere Erläuterung der mit "*" beginnenden Systemnamen erfolgt weiter unten bei der Darstellung der Komponente *uutest*.
Es ist denkbar, die Komponenten *uusend* und *uuget* zu einer gemeinsamen Komponente zusammenzufassen bzw. in die Komponente *uucp* zu integrieren. Ob das notwendig oder sinnvoll ist, muß erst der Einsatz zeigen. Problematisch wäre dann auf jeden Fall die Kompatibilität zu anderen UUCP-Versionen.

Das Programm mail

Das ursprüngliche Programm *mail* im UNIX-System Version 7 unterstützt bereits das Übertragen von Mail-Nachrichten an Nutzer anderer Systeme, die über UUCP erreichbar sind. Der Nutzer muß dabei neben dem eigentlichen Nutzernamen noch den Systempfad angeben, der zu dem Rechner des Nutzers führt. Das Kommando *mail* sieht in diesem Fall wie folgt aus:
mail {[⟨Systempfad⟩]⟨Nutzername⟩}
mit ⟨Systempfad⟩::=⟨Systemname⟩"!" {⟨Systemname⟩"!"} und ⟨Nutzername⟩ = realer Name des Nutzers auf dem spezifizierten System. Dieses Verfahren ist recht umständlich und verlangt außerdem von jedem Nutzer die genaue Kenntnis der Topologie des UUCP-Netzes. Es wird in der Regel deshalb wenig oder nicht genutzt. Da in den meisten Einsatzfällen jeder Nutzer feste Rechner und Terminals benutzt, ist es zur Verbesserung der Kommunikation zwischen den Nutzern möglich, ein praktikables Werkzeug zum Übertragen von Mail-Nachrichten zwischen Nutzern verschiedener Rechner zu schaffen. Durch die sowieso vorhandenen Informationen über die Topologie des Netzes in dem Route-File */usr/lib/uucp/L route* ist eine Voraussetzung für ein automatisches Finden des Systempfades gegeben. Es fehlt lediglich eine Verbindung zwischen den Nutzernamen und dem zugehörigen Systemnamen. Dieses Problem wurde durch die Schaffung eines weiteren Adress-Files */usr/lib/uucp/L mail* gelöst, das neben einem logischen (im UUCP-Netz eindeutigen) Nutzernamen den Systemnamen und den realen Nutzernamen auf diesem System beinhaltet.
Das verbesserte Programm *mail* erlaubt nun das Versenden von Mail-Nachrichten innerhalb des gesamten UUCP-Netzes. Dabei wertet das Programm *mail* die Files */usr/lib/uucp/L mail* und */usr/lib/uucp/L route* erst dann aus, wenn der angegebene Nutzername nicht auf dem lokalen Rechner gefunden wird (*/etc/passwd*).
Das verbesserte Programm *mail* wird wie folgt aufgerufen:
mail {[⟨Systempfad⟩]⟨Nutzername⟩}
Dabei bedeuten ⟨Systempfad⟩::=⟨Systemname⟩"!"{⟨Systemname⟩"!"} und ⟨Nutzername⟩ einen realen Nutzernamen, wenn dieser auf dem lokalen Rechner existiert oder einen logischen Nutzernamen, wenn ein entsprechender Name auf dem lokalen Rechner nicht gefunden wurde.
Das verbesserte Programm *mail* erfüllt alle geforderten Bedingungen und ist mit dem ursprünglichen voll verträglich, d. h. kann es ersetzen.

```
WEGA1     \!
WEGA2     WEGA3\!WEGA2\!
WEGA3     WEGA3\!
WEGA4     WEGA5\!WEGA4\!
WEGA5     WEGA5\!
VENIX1    WEGA5\!VENIX1\!
*WEGA2    WEGA3\!WEGA2\!
*WEGA2    WEGA5\!WEGA4\!WEGA2\!
```

Abb. 2 Beispiel für ein Route-File /usr/lib/uucp/L route

Das Adress-File */urs/lib/uucp/L mail* enthält für jeden logischen Nutzer einen Eintrag mit folgender Struktur:
⟨logischer Nutzername⟩ ⟨Systemname⟩ ⟨realer Nutzername⟩.
Jeder Eintrag wird mit einem Zeilenende abgeschlossen. Die logischen Nutzernamen sollten innerhalb eines UUCP-Netzes eindeutig sein. Dies ist aber nicht notwendig; es ist zuläs[illegible] daß ein realer Nutzer mehrere logische Nutzernamen erhält.
Ein Beispiel für ein Adress-File */usr/lib/uucp/L mail* ist in Abb. 3 dargestellt.

Das Programm uutest

Nach der Installation des oben beschriebenen UUCP-Netzes ist es für den Superuser äußerst schwierig zu überprüfen, ob alle Verbindungen funktionsfähig sind. Er müßte sich dazu in jedem Rechner nacheinander anmelden und einen Filetransport zu den benachbarten Rechnern starten oder eine Kommandoausführung mittels *uux* anstoßen und die Ergebnisse von Hand verifizieren. Dies ist ein sehr zeitaufwendiges Verfahren.
Es wurde deshalb die Komponente *uutest* entwickelt, die es erlaubt, von einem Rechner aus gezielt einzelne V[illegible]bindungen bzw. alle Verbindungen zu überprüfen. *uutest* bedient sich dabei *uux*. Entsprechend der in den Parametern angegebenen Verbindungen werden *uutest*-Kommandos mit speziellen Parametern an die entsprechenden Rechner gesendet, die ihrerseits wieder in Abhängigkeit von den übermittelten Parametern *uutest*-Kommandos an weitere Rechner und den Absenderrechner senden. Auf dem Absenderrechner werden die eintreffenden Kommandos auf dem Masterterminal protokolliert, so daß leicht sichtbar wird, welche Verbindung gestört ist (es trifft keine entsprechende Antwort ein). Dies garantiert, daß jede Verbindung von beiden Seiten aus geprüft wird. Dies ist sehr wichtig, da häufig eine Verbindung von einer Seite aus nutzbar ist, aber von der anderen Seite aus nicht. Das Programm *uutest* wird wie folgt aufgerufen:

jan	WEGA5	bell
polze	WEGA2	polze
hans	WEGA2	schiem
gandre	WEGA5	gandre
wolfgang	WEGA5	gandre
pascal	WEGA4	olga
prolog	WEGA1	md
dieter	WEGA2	burkhard

Abb. 3 Beispiel für ein Adress-File /usr/lib/uucp/L mail

uutest [−x(n)] [(Systemname)].
Dabei bedeutet −x(n) die gewünschte Debugging-Ebene (n = 1: *uutest* protokolliert auf den Masterterminal eines jeden beteiligten Systems die noch zu prüfenden Verbindungen und n = 2: zusätzlich werden die ausgeführten *uux*-Kommandos protokolliert). (Systemname) spezifiziert das System, zu dem hin die Verbindungen überprüft werden sollen. Der entsprechende Systempfad wird mit Hilfe des Files */usr/lib/uucp/L route* ermittelt. Wird kein Systemname spezifiziert, so werden aus dem File */usr/lib/uucp/L route* alle Verbindungen herausgesucht, die mit „#" gekennzeichnet sind.

Einige Beispiele

Die hier dargestellten Beispiele beziehen sich auf die Rechnerkonfiguration in Abb. 1 und auf die in Abb. 2 und 3 angegebenen Fileinhalte für den Rechner mit dem Namen WEGA1. Die Kommandos sollen jeweils am Rechner WEGA1 eingegeben werden.

■ Aktivieren des Übertragungsprogramms *unetio*:
uustart WEGA3 WEGA5
oder
unetio -sWEGA3 &
unetio -sWEGA5 &.
Beide Kommandofolgen bewirken, daß für die Kommunikation mit den Systemen WEGA3 und WEGA5 jeweils ein Prozeß mit dem Übertragungsprogramm *unetio* gestartet wird und die Kommunikationslinien eröffnet werden.

■ Transport des Files */z/xx* vom Rechner WEGA4 zum Rechner WEGA1:
uuget WEGA4\!/z/xx /z/e/yy.
Dieses Kommando bewirkt, daß das File */z/xx* vom Rechner WEGA4 zum Rechner WEGA1 transportiert wird und dort unter dem Namen */z/e/yy* abgelegt wird, wenn das File *yy* kein Filezverzeichnis ist, bzw. unter dem Namen */z/e/yy/xx*, wenn *yy* ein Fileverzeichnis ist.

■ Transport des Files */z/aa* vom Rechner WEGA1 zum Rechner WEGA2:
uusend /z/aa WEGA2\!/z/d/bb.
Dieses Kommando bewirkt, daß das File */z/aa* vom Rechner WEGA1 zum Rechner WEGA2 transportiert wird und dort unter dem Namen */z/d/bb* abgelegt wird, wenn das File *bb* kein Fileverzeichnis ist und unter dem Namen */z/d/bb/aa*, wenn *bb* ein Fileverzeichnis ist.

■ Senden von Nachrichten an Nutzer:
mail jan polze pascal prolog
Das ist ein Test.
^D.
Dies bewirkt, daß der Text „Das ist ein Test." an folgende Nutzer übertragen wird: bell auf WEGA5, polze auf WEGA2, olga auf WEGA4 und md auf WEGA1

■ Test der Funktionsfähigkeit der Verbindungen:
uutest
Es werden die Verbindungen zu den Rechnern WEGA2, WEGA3, WEGA4 und WEGA5 getestet (siehe #-Option im Route-File).
uutest VENIX1
Es wird die Verbindung zum Rechner VENIX1 getestet.

Erfahrungen beim Einsatz des modifizierten UUCP-Paketes

Das modifizierte UUCP-System wurde auf der oben beschriebenen Hardware installiert und getestet. Dabei konnten folgende Ergebnisse im praktischen Einsatz festgestellt werden:

• Die Reaktionszeit gegenüber dem ursprünglichen UUCP ist deutlich kürzer geworden. Dauerte die Überprüfung aller Verbindungen mittels *uutest* zwischen 18 und 24 Minuten, so benötigt die verbesserte Version lediglich 2,5 bis 3 Minuten. Gleichfalls wurde die Wartezeit zwischen dem Starten einer Fileübertragung und dem Bereitstellen des Files im Verzeichnis deutlich kürzer. Files mit einer Länge von ein bis zwei KByte stehen bereits nach 5 bis 8 Sekunden auf dem Zielrechner zur Verfügung.
• Die Zuverlässigkeit von UUCP ist deutlich gestiegen. Während zuvor der Test mittels *uutest* für das Gesamtsystem höchstens in zwei von drei Fällen erfolgreich war, sind bei dem modifizierten System neun von zehn Tests erfolgreich. Fehler können in der Regel durch Benutzung von *uupoll* behoben werden und sind in 90 Prozent aller Fälle auf die Hardware zurückzuführen.
• Die eigentliche Fileübertragungsgeschwindigkeit von etwa 500 Byte bei 9 600 Baud und 700 Byte bei 19 200 Baud konnte nicht erhöht werden.
• Störungen des Druckerspoolers konnten nicht mehr festgestellt werden.
• Die verbesserte Komponente *mail* wird häufiger zur Nutzerverständigung genutzt.

Es kann insgesamt festgestellt werden, daß das modifizierte UUCP-System den Anforderungen an ein lokales Netz für den Einsatz im praktischen Betrieb im begrenzten Umfang gerecht wird. Fileübertragungen zwischen beliebigen Rechnern sind möglich. Für kleinere Files ist die Reaktionszeit akzeptabel. Für umfangreiche Filetransporte ist das Medium V.24 mit 19 200 Baud im Einzelzeichentransport zu langsam. Hier kann nur mit einem im Blockmodus arbeitenden Treiber eine deutliche Verbesserung erreicht werden.

Interessenten wenden sich bitte an
Dr. Jan-Peter Bell, Humboldt-Universität zu Berlin, Sektion Mathematik, PF 1297, Berlin, 1086, Tel.: 20 93 29 82.

Literatur

/1/ D. A. Nowitz, M. E. Lesk: A Dial-Up Network of UNIX-TM Systems.
Bell Laboratories, Murray Hill, New Jersey in UNIX PROGRAMMER'S MANUAL, 7. Edition, Vol. 2B
/2/ D. A. Nowitz: Uucp Implementation Description. Bell Laboratories, Murray Hill, New Jersey, Unix Verteilerband Version 7
/3/ UUCP Implementation; WEGA-Dienstprogramme (B). VEB KEAW Berlin, 1987

Jahresinhaltsverzeichnis 1988/89

1/88 Büroautomatisierung

Prof. Dr. Kurt Sack: Anwenderspezifische Lösungen mit PC-Standardsoftware 2
Barbara Kaminski: Bildschirmarbeitsplätze-Vorbereitung eines DDR-Standards 5
Jochen Picht: Aufgaben der Endnutzer bei Softwareentwicklung und -nachnutzung 7
Prof. Dr. Klaus Fuchs-Kittowski, Dr. Christian Hartmann: Büroautomatisierung – Ziele, Aufgaben, Wirkungen 9
Dr. Margit, Falck, Prof. Dr. Klaus Fuchs-Kittowski, Dr. Christian Hartmann, Dr. Günther Klatt: Probleme, Methoden, Erfahrungen bei der Einbeziehung der Nutzer 14
Günter Wirth, Günter Gräfe: Anleitung zu rationeller und einheitlicher Einsatzvorbereitung 18
Matthias Geißer, Jochen Picht, Lutz Teichmann: Gestaltung einer Software-Produktionsumgebung für 16-Bit-PC 20
Helmut Noack: Kooperative Einsatzvorbereitung – Erfahrungen und Probleme 24
Volker Heinrich: RDBM/SCP – Menüprogramm für die Arbeit mit REDABAS 26
Jürgen Leuschner, Manfred Kühling, Jochen Göhl: Kommunikationsbeziehungen zwischen REDABAS, KP und TP30 30
Wolfgang Geinitz, Rainer Trillhase: Software-Informations-Datenbank SODABA 34
Dr. Wolfgang Bär, Dr. Peter Weber: REDABAS-Basissoftware: Universelles Rechercheprogramm 36
Frieder Tautenhahn, Ingrid Saworra: Rationalisierung der Büro- und Sekretariatsarbeit mit RABA 39
Manfred Günther, Hans-Günter Meißler, Christian Künzer: Leitungsinformationssystem „Investitionen“ 45
Prof. Dr. Werner Kunitz, Detlef Wilke, Gert Gräßler: BILAP – ein Bilanziererarbeitsplatz 49
Eckhard Jahn: Enwicklung eines Rechnernetzes 53
Dr. Hans-Götz Teubert, Peter Stryczek: Wissenschaftsplanerarbeitsplatz im System der Leitung und Planung 57
Softwaremarkt 61

2/88 TESYS Softwarewerkzeuge Konstruktionselemente für 16-Bit-Technik

Gunter Chmiel, Rolf Werner: TESYS-3 – Überblick 2
Peter Elbinger: TESYS-3-Einsatzvorbereitung 19
Kristian Hänsel: TESYS-3 – Anwendung für den funktionellen Entwurf 22
Klaus-Peter-Bräuer: TESYS-3 – Anwendung für den technischen Entwurf 33
Christian Schütze: TESYS-3 – Anwendung für das Implementieren und Testen 44
Peter Elbinger: TESYS-3 – Erfahrungen 50
Ingo Sturm: Adaptive Produktionstechnologie mittels TESYS-3 54

3/88 PC-Software aus dem VEB LfA

Rechnergestütztes Konstruieren 2
Grafisches Editiersystem 5
Rechnergestütze Programmierung von NC-Maschinen 8
Integriertes System zur Bürorationalisierung 11
Datenverwaltung 15
Statistische Datenanalyse 17
Rechnerverbund für den Dateitransfer mit Auftragssteuerung 20
Dateitransfer auf Basis des KERMIT-Protokolls 21
Integration von Personalcomputern in Rechnernetzwerke 24
Programmiersystem C DCP 26
Compiler für die REDABAS-Datenbanksprache 28
Archivierung/Komprimierung von Dateien 32
Dienstprogramme für effektive Personalcomputernutzung 33
Festplattensicherungs- und Restaurierungssystem 35
Verarbeitung von SCP-Diskettenformaten unter DCP 36
Datenprüfung und Korrektur auf der Basis von REDABAS 37
Festplatten-Manager 40
Kommando-Editor 42
Definition und Druck teiltypisierter Listen auf der Basis von REDABAS 43
Datei- und Diskettenverwaltungsprogramm 45
Schutz von Nutzerverzeichnissen 46
Bildschirmpufferung 47
Speicherresidentes Assistenzprogramm 48
Editor für Programmquelltexte 49
Bildschirmorientierter Programmdebugger 50
Speicher-Mapping 51
Projekt-/Ressourcenplanung und -verwaltung 52
Textverarbeitungssystem für die rechnergestützte Dokumentationsverarbeitung 55
Elektronisches Layout-Gestaltungssystem 57
Kalkulationsprogramm 59
Auskunftssystem für die Materialwirtschaft/Materialplanerarbeitsplatz 59
Referenzhandbuch 61
Fachbroschüre: Anwendung von Disketten im ESER 64

4/88 DCP-Software aus dem Kombinat Robotron

Wolfgang Krampen, Brigitta Wulf: Vertriebsformen und Vertriebsbedingungen 2
Harald Meichsner: Betriebssystem DCP Version 3.30 4
Ursula Hempel, Rolf-Gunter Riedel: Das Datenbankbetriebssystem REDABAS-3 6
Hanns-Dieter Hartmann: Informationsrecherchesystem AIDOS/M-1 10
Michael Philipp: CAD-Lösungen mit 16-Bit-Technik 17
Rainer Korzen: GKS16 – Grafisches Kernsystem für 16-Bit-Rechner 20
Dieter Köhler: MultiCAD für EC 1834 21
Prof. Dr. Helmut Kupper, Dr. Martin Mai, Hans-Joachim Cüppers: Grafisches Modelliersystem GRAMOS-16 27
Dr. Martin Mai, Michael Hofmann, Siegfried Peter, Rolf Paul: GRAMOS-ALPHA – Alphanumerische Datenhandhabung 31
Dr. Stefan Wiesmann, Dr. Bernd Sänger, Peter Deutscher: GRAMOS-GEOMETRIE 37
Christel Börner, Irina Ecke: ARIADNE DCP – Ein integriertes Softwaresystem 48
Gerhard Dynowski, Osmar Günzel, Günter Schade: MULTICOMP – Ein integriertes Standardsoftwaresystem 55
Klaus Fahr: Leistungsmerkmale von TPASCAL 64

1/89 Softwarestandardisierung

Standards – Mittel zur Durchsetzung der Schlüsseltechnologien 2
Termini und Definition der Informationsverarbeitung 3
Allgemeine Termini und Definitionen 4
Termini und Definitionen für Softwarequalitätssicherung 9
Standardisierung höherer Programmiersprachen 13
Bildschirmarbeitsplatz – Forderungen an die Arbeitsstätten 14
CAD/CAM-Grundlagenstandards 16
Datenschnittstelle für Finite-Elemente-Modelle 18
Die Fachsprache STADAS 19
Austausch produktbeschreibender Daten 20
CENIT 1: DDR-Standard für Datenbasen 20
Standards für lokale Netze; Qualität und Produktivität 21
Qualitätssicherung von Software 23
Qualitätssicherung von Software – Qualitätssicherungssysteme 25
Gebrauchseigenschaften von Software 29
Phasenmodell des Softwarelebenszyklus 36
Bereitstellung von Standards zur Informationsverarbeitung/Software 39
Dokumentation von Software 40
Sachmerkmale für die Informationsverarbeitung 50
Hardwareergonomie – Bildschirmarbeitsplatz 52
Die Datenbanksprache SQL 62
Standards der Informationsverarbeitung – Übersicht 63

2/89 Verteilte Verarbeitung

Prof. Dr. Gerd Rossa: Rechnergestützte Betriebswirtschaft und verteilte Verarbeitung 2
Prof. Dr. Claus Sattler: Gestaltung netzfähiger Software für Arbeitsprozesse im Büro 5
Prof. Dr. Gerd Rossa: Datenbasis-Entwurfs-System DES 10
Uwe Schulze: Dienstorganisation in lokalen Netzen mittels Fernaufruf 14
Prof. Dr. Gerd Rossa: Kopplung von DBS und XPS Semantische Datenbeschreibung 17
Axel Wüstemann: Netzbetriebssysteme für den Verbund von Arbeitsplatzrechnern 18
Uwe Schulze: NETBIOS – standardisierter Netzzugang für PC-Technik 22
Prof. Dr. Gerd Rossa: Modell eines verteilten Datenbank-Betriebs-Systems (VDB) 25
Uwe Schulze, Axel Wüstemann, Matthias Ohlerich: Bürokommunikation mit Micro-NET-80 30
Heidrun Ortlieb: Verteilte Verarbeitung – Datenkonvertierung bei relationalen Datenbanken 33
Frank Försterling, Ingo Kerlikowski, Prof. Dr. Claus Sattler, Dr. Christine Teßmer: Rechnergestützte Arbeit mit Mitteilungen im Büro 35
Dr. Gerald Hartung: Auftragstransfer – Ein Anwendungsdienst im ESER-Rechnerverbund
Dr. Peter Erward: DAFEMA-PC: Kopplungssoftware für die verteilte Verarbeitung 42
Rolf Stickler, Axel Wüstemann: NetWare – Hochtechnologie bei PC-Netzen 46
Dieter Lenz, Volker Nawrotzki, Martin Holzhauser: Zimmerreservierungssystem HODIS-2 unter Nutzung von SCOMLAN 50
Axel Wüstemann: Der Einsatz von lokalen Netzen in Betrieben 56
Prof. Dr. Gerd Rossa, Prof. Dr. Martin Graef, Frank Bolz: Drei Beiträge aus LO + EDV '88 61

3/89 Modulbausteine für rechnergestützte Arbeitsplätze

Grundlagen für arbeitsplatzbezogene Programmlösungen 2
Menügestaltung und Menüauswertung 11
Dateiarbeit 22
Einsatz von Strukturbeschreibungsdateien 35
Datenerfassung 41
Blättern in Datenbankdateien 51
Erzeugen von Befehlsfolgen für Kalkulationsprogramme, Textprogramme und Datenbanksysteme 58

4/89 Logisches Programmieren – Software-Werkzeuge für den P 8000

Dr. Christian Horn, Mirko Dziadzka, Matthias Horn: HU-PROLOG Sprachbeschreibung 2
Tom Bühr: Programmieren mit PROLOG – einige Beispiele 27
Dr. Bodo Hohberg, Olga Wikarski: Portierung von PASCAL-Software mit PCC 32
Dr. Wilfried Grafik, Heinz Werner: Modula 2 für den P 8000 39
Falk Nisius, Kai-Uwe Scherer: Modula-2-Interpreter für P 8000 44
Jens-Peter Redlich: Ein moderner Editor für UNIX-kompatible Betriebssysteme 47
Reinhard Hartung: SPIDER – Ein System zur Datenübertragung 53
Dr. Jan-Peter Bell: Vernetzung mehrerer P 8000 unter WEGA mit UUCP 59

HU-PROLOG – Referenzkarte

Die folgende Tabelle ist eine Übersicht über die Built-In-Prädikate von HU-PROLOG mit Angabe der Gliederungsnummer in diesem Heft. Es werden auch (noch) nicht dokumentierte Prädikate mit aufgeführt. Dies sind Prädikate zum Arbeiten mit dBaseIII-Datenbanken (gekennzeichnet mit dBase3), Prädikate für eine erweiterte Ein- und Ausgabe, die auch das Arbeiten mit Windows ermöglicht (gekennzeichnet mit I/O) sowie Prädikate eines online-Help-Systems (gekennzeichnet durch help).

abolish (Atom) ⟨4.2.2.⟩
abolish (Atom, Arity) ⟨4.2.2⟩
abort ⟨6.2.2⟩
ancestors (X) ⟨6.2.7⟩
arg (N,Term,Arg) ⟨3.2.3⟩
ask (Char) ⟨2.2.5⟩
assert (Term) ⟨4.1.3⟩
assert (Term,Ref) ⟨4.1.3⟩
assert (Term,Ref,Pos) ⟨4.1.4⟩
asserta (Term) ⟨4.1.1⟩
asserta (Term,Ref) ⟨4.1.1⟩
assertz (Term) ⟨4.1.2⟩
assertz (Term,Ref) ⟨4.1.2⟩
assign (Stream,File) I/O
assign (Stream,Stream) I/O
assgin (Stream, window (X,Y,Dx, Dy,Name,List) I/O
atom (Term) ⟨3.1⟩
atomic (Term) ⟨3.1⟩
call (Call) ⟨6.1.7⟩
clause (Head,Body) ⟨4.3.1⟩
clause (Head,Body,Ref) ⟨4.3.1⟩
close (File) ⟨2.1.2⟩
closedbf (File,Struc) dBase3
cls ⟨2.3.4⟩
consult (File) ⟨6.7.1⟩
createdbf (File,Struc) dBase3
current_atom (Name/Arity) ⟨3.3.1⟩
current_op (Prec,Ass,Name) ⟨4.4.2⟩
date (Year,Month,Day) ⟨6.4.1⟩
dict (X) ⟨4.3.4⟩
display (Term) ⟨2.3.1⟩
echo (Flag) ⟨6.5.1⟩
end ⟨6.2.3⟩
ensure (Lib,Name,Arity) ⟨6.3.3⟩
eof ⟨2.2.3⟩
eoln ⟨2.2.3⟩
erasedbf (File,Pos) dBase3
error (Call,Errornum) ⟨6.2.5⟩
exit (Exitcode) ⟨6.2.3⟩
fail ⟨6.1.3⟩
fileerrors ⟨2.1.10⟩
fileerrors (Flag) ⟨2.1.10⟩
functor (Term,Name,Arity) ⟨3.2.2⟩
get (X) ⟨2.2.2⟩
get0 (X) ⟨2.2.2⟩
getenv (Value,Envvar) ⟨6.4.4⟩
gotoxy (XPos,YPos) ⟨2.3.4⟩
ground (Term) ⟨3.1⟩
halt ⟨6.2.3⟩
help help
help (Topic) help
hidden (Atom) ⟨6.3.2⟩
hidden (Atomlist) ⟨6.3.2⟩
integer (Term) ⟨3.1⟩
interrupt ⟨6.2.4⟩
invar (Term) ⟨3.1⟩
is (Term,Expr) ⟨5.3⟩
listing ⟨4.3.3⟩
listing (Atom) ⟨4.3.3⟩
listing (Atom/Arity) ⟨4.3.3⟩
log ⟨6.5.2⟩
log (Flag) ⟨6.5.2⟩
name (Atom,String) ⟨3.3.2⟩
nl ⟨2.3.3⟩

nofileerros ⟨2.1.10⟩
nolog ⟨6.5.2⟩
nonvar (Term) ⟨3.1⟩
nospy (Atom) ⟨6.6.2⟩
notrace ⟨6.6.1⟩
not (Call) ⟨6.1.8⟩
number (X) ⟨3.1⟩
ocheck (Flag) ⟨6.5.4⟩
op (Prec,Ass,Name) ⟨4.4.1⟩
open (File) ⟨2.1.1⟩
opendbf (File,Struc) dBase3
private (Atom) ⟨6.3.1⟩
private (Atomlist) ⟨6.3.1⟩
put (Char) ⟨2.3.2⟩
read (X) ⟨2.2.1⟩
real (Term) ⟨3.1⟩
readdbf (File,Data,Pos) dBase3
reconsult (File) ⟨6.7.1⟩
repeat ⟨6.1.4⟩
restart ⟨6.2.2⟩
retract (Term) ⟨4.2.1⟩
retract (Term,Ref) ⟨4.2.1⟩
retractall (Term) ⟨4.2.2⟩
save (File) ⟨6.7.2⟩
sdict (X) ⟨4.3.4⟩
see (File) ⟨2.1.3⟩
seeing (X) ⟨2.1.3⟩
seek (File,Position) ⟨2.1.5⟩
seekdbf (File,Pos) dBase3
seen ⟨2.1.3⟩
skip (Char) ⟨2.2.5⟩
spy (Atom) ⟨6.6.2⟩
stats ⟨6.8⟩
sys (Name/Arity) ⟨4.3.2⟩
system ([Cmd, ...]) ⟨6.4.4⟩
tab (N) ⟨2.3.3⟩
tell (File) ⟨2.1.4⟩
telling (X) ⟨2.1.4⟩
time (Hour,Min,Sec) ⟨6.4.1⟩
timer (CPU_Time) ⟨6.4.2⟩
told ⟨2.1.4⟩
toplevel ⟨6.2.1⟩
trace ⟨6.6.1⟩
trace (Flag) ⟨6.6.1⟩
true ⟨6.1.3⟩
unget ⟨2.2.4⟩
unknown (Call) ⟨6.2.6⟩
var (Term) ⟨3.1⟩
warn (Flag) ⟨6.5.3⟩
weekday (Weekday) ⟨6.4.1⟩
write (Term) ⟨2.3.1⟩
writedbf (File,Data,Pos) dBase3
writeq (Term) ⟨2.3.1⟩
:-(Term,Term) ⟨6.1 9⟩
:-(Term) ⟨6.1.10⟩
?-(Term) ⟨6.1.10⟩
;(Term,Term) ⟨6.1.2⟩
-⟩(Term,Term) ⟨6.1.6⟩
,(Term,Term) ⟨6.1.1⟩
\+(Term) ⟨6.1.8⟩
:=(Term,Expr) ⟨5.6⟩
\==(Term,Term) ⟨3.4.2⟩
\=(Term,Term) ⟨3.4.1⟩
@=(Term,Term) ⟨3.4.3⟩
@⟩=(Term,Term) ⟨3.4.3⟩

@⟩(Term,Term) ⟨3.4.3⟩
@==(Term,Term) ⟨3.4.3⟩
@=⟨(Term,Term) ⟨3.4.3.⟩
@⟨(Term,Term) ⟨3.4.3⟩
⟩=(Expr,Expr) ⟨5.7⟩
⟩(Expr,Expr) ⟨5.7⟩
\==(Term,Term) ⟨3.4.2⟩
=⟨(Expr,Expr) ⟨5.7⟩
=:=(Expr,Expr) ⟨5.7⟩
=(Term,Term) ⟨3.4.1⟩
⟨(Expr,Expr) ⟨5.7⟩
=\=(Expr,Expr) ⟨5.7⟩
=..(Term,List) ⟨3.2.1⟩

Operatoren:

Name	Assoziativität	Vorrang
is	xfx	800
:-	xfy	1200
:-	fx	1200
?-	fx	1200
;	xfy	1100
-⟩	xfy	1050
,	fy	1000
not	fy	900
\+	fy	900
:=	xfx	700
is	xfx	700
\==	xfx	700
\=	xfx	700
@=	xfx	700
@⟩=	xfx	700
@⟩	xfx	700
@==	xfx	700
@=⟨	xfx	700
@⟨	xfx	700
⟩=	xfx	700
⟩	xfx	700
\==	xfx	700
=⟨	xfx	700
=:=	xfx	700
=	xfx	700
⟨	xfx	700
=\=	xfx	700
=..	fx	700
\	xfy	650
\\	xfy	650
&	xfy	650
&&	xfy	650
\	fy	650
⟫	xfy	600
⟪	xfy	600
-	yfx	500
+	yfx	500
*	yfx	400
/	yfx	400
//	yfx	400
mod	yfx	400
**	xfy	350
.	fy	300
-	fy	300
/	fy	300

1331 4 175 218 233
SPRENGSTOFFW
8600 7002 1821
PSF 220

ISSN DDR 0232-6833 · edv-aspekte · Berlin 8 (1989) 4 · S. 1–64

aspekte **blickpunkt**

Der P 8000 compact ist das Ergebnis der konsequenten Weiterentwicklung des P 8000 unter Einbeziehung modernster Schaltkreise der DDR-Produktion und modernster Technologien in der Fertigung.

Entsprechend dem internationalen Trend bei Entwicklungssystemen ist im P 8000 compact, wie auch bereits im P 8000, das UNIX-kompatible Betriebssystem WEGA als Hauptbetriebssystem implementiert. Damit steht dem Anwender eine äußerst leistungsfähige Systemsoftware zur Verfügung. Durch die Einbindung weiterer Betriebssysteme wird die Anwendungsbreite des P 8000 compact noch wesentlich erhöht.

Mit der Möglichkeit des Anschlusses von bis zu acht Terminals oder PC über V.24 oder IFSS (bei IFSS bis 500 m) bzw. über Modem (zwei Schnittstellen) eignet es sich insbesondere für solche Anwendungen, bei denen die Multiuserfähigkeit von entscheidender Bedeutung ist.

Der P 8000 compact zeichnet sich durch folgende wesentliche Eigenschaften aus:

- 16-Bit-Mikrorechner mit bis zu 4 MByte RAM
- Festplattenkapazität 44 MByte oder 88 MByte
- 8 serielle Schnittstellen V.24 (davon 6 wahlweise IFSS)
- EPROM-Programmer für die Typen 2716 bis 27512
- Echtzeituhr (batteriegestützt)
- vier Betriebssysteme:
  WEGA (kompatibel UNIX)
  W-DOS (kompatibel MS-DOS)
  OS/M (kompatibel CP/M)
  UDOS (kompatibel RIO)
- P-8000-In-Circuit-Emulator für Einchiprechner
- Zusatzsoftware:
  WEGA-DATA Datenbanksystem
  WEGA-CALC Tabellenkalkulation
  WEGA-WORD Textverarbeitungsprogramm
  WEGA-CROSS U880, U881, K1810WM86 – ASM, Linker
  WEGA-REMOTE Anschluß von PC
  WEGA-LIB Mathematikbibliothek
  WEGA-BASIC BASIC-Compiler/Interpreter
  WEGA-PASCAL PASCAL-Compiler
  WEGA-FORTRAN FORTRAN-77-Compiler
  WEGA-COBOL COBOL-Compiler
  WEGA-EMSCP Multi-User-SCP-Emulator