PROLOG 64

Brainware

Ihr Experte in Expertensystemen
Consulting · Schulung · Software

PROLOG 64

Vers. 1.1

Prolog Interpreter für den C 64

Dr. Berthold Daum

BRAINWARE

Gesellschaft für Artificial Intelligence

Systementwicklung und -beratung mbH

Kirchgasse 24

6200 Wiesbaden

Tel. 06121-372011

In einem nostalgischen Retro-Anfall wurde dieses Handbuch in der Zeit zwischen dem 15.11.2011 und dem 26.11.2011 gescannt, korrekturgelesen und geTEXt. Da mir bisher noch keine Kopie der Software und des Handbuchs im Internet untergekommen ist, und es sehr schade wäre, wenn dieses Stück 64 er-Geschichte im Datennirwana verschwunden bliebe, habe ich das Handbuch als Prototyp benutzt, um LYX 2.0 auszuprobieren und dabei mit der freien TrueType-Schriftart **Pet Me 64** (The Ultimate Commodore Font) zu experimentieren. Ich hoffe, dass der eine oder andere etwas Spaß mit dem Programm hat.

*Wer schläft am Tag und **knackt** in der Nacht, es ist der Hamster, der sowas macht.*
— Hamster

Handbuch-Version: 1.0

Inhaltsverzeichnis

1 Bedienungsanleitung

A) Inbetriebnahme

1. Schalten Sie den C64 ein.
   - Wollen Sie zusätzlich das EPSON-Software-Interface benutzen, so laden Sie es jetzt. Geben Sie dabei Code 3 ein.
2. Legen Sie die PROLOG 64 - Floppy ein und laden Sie Prolog mit:

```
LOAD "PROLOG",8
```

   oder

```
LOAD "*",8
```

   - Wenn Sie einen der folgenden Parameter verändern wollen, so können Sie das jetzt tun (siehe Kapitel 16 Anhang auf Seite 45). Folgende Parameter sind im Lader abgelegt:

   - Rahmenfarbe
   - Hintergrundfarbe
   - Zeichenfarbe
   - Tracefarbe
   - Fehlerfarbe
   - Startupbefehl
   - RS 232
   - 5 verschiedene Klangfarben für die eingebaute Funktion SOUND.

3. Starten Sie PROLOG mit

```
RUN
```

4. Sobald sich PROLOG 64 mit der Copyright-Zeile meldet, ist es betriebsbereit.

Im folgenden finden Sie eine kurze Beschreibung von PROLOG 64:

Die Kapitel 'Einführung in PROLOG' und 'Programmiermethoden' sollen in die Technik der Programmierung einführen.

Das Kapitel 'Syntax' gibt Ihnen einen Überblick über die Schreibweise von PROLOG-Ausdrücken.

Die Kapitel 'Eingebaute Funktionen' und 'PROLOG 64 Erweiterungen' erläutern alle eingebauten Funktionen von PROLOG 64.

Wie Sie selbst Programme erstellen und speichern können, ist im Kapitel 'PROLOG Bibliotheken' beschrieben.

Das Einlesen der Bibliotheken in das PROLOG-System ist im Kapitel 'Ein- und Ausgaben' beschrieben.

Dort finden Sie auch die notwendigen Informationen über die Dateiverarbeitung. Darüberhinaus gibt es weitere Kapitel, die die Grafik- und Klangmöglichkeiten des C64 und das Testsystem behandeln.

Da in den einzelnen Kapiteln einige Male auf Programme in den mitgelieferten Bibliotheken bezug genommen wird, ist eine Auflistung aller Bibliotheken beigefügt.

Diese Dokumentation bietet nur eine kurze Beschreibung der Sprache PROLOG. Eine umfangreichere Beschreibung finden Sie in dem Buch

> Programming in PROLOG
> W.F. Clocksin, C.S. Mellish
> Springer Verlag, Heidelberg, New York 1981

PROLOG 64 weicht in einigen Punkten von diesem Buch ab. Die Unterschiede sind im Anhang aufgelistet.

2 Einführung in PROLOG

PROLOG bedeutet PROgramming in LOGic. Die Entwicklung von PROLOG reicht bis ins Jahr 1970 zurück (Alain Colmerauer). Seit dieser Zeit sind verschiedene Dialekte von PROLOG entstanden. 1981 erschien das Buch 'Programming in PROLOG' von Clocksin und Mellish. In diesem Buch wurde ein sogenanntes Kern-PROLOG definiert — der 'Edinburgh'-Standard. Heutige PROLOG Implementierungen müssen sich an diesem Standard messen.

Die Einsatzmöglichkeiten von PROLOG liegen weniger auf dem Gebiet der numerischen Datenverarbeitung, sondern dort, wo Symbole verarbeitet werden:

- Verarbeitung natürlicher Sprache, Übersetzer, Gesprächsführung
- Expertensysteme
- Formelbearbeitung z.B. Bildung des Differentialquotienten, Vereinfachung von Formeln, Verarbeitung biochemischer Formeln
- mathematische Logik und automatisches Beweisen
- relationale Datenbanken
- Wissensbasen
- Rapid Prototyping (Spezifikation eines Programmes erfolgt in PROLOG und kann gleich getestet werden, dann Programmierung in konventioneller Sprache)
- intelligente Spiele
- Bildverarbeitung (Szenenanalyse)

PROLOG arbeitet in vollkommen anderer Weise als herkömmliche Programmiersprachen. Herkömmliche Programmiersprachen arbeiten prozedural, d.h. der Programmierer bestimmt die Reihenfolge der Computeroperationen mittels der Programmbefehle.

Ganz anders in PROLOG. Hier werden nicht Programmabläufe festgelegt, sondern es werden Sachverhalte beschrieben. Dies geschieht mit einfachen Wenn-dann-Regeln und mit Fakten. Ein PROLOG-Programm gleicht deshalb mehr einer ungeordneten Ansammlung von Wissen als einem Programm.

2.1 Die Darstellung von Wissen

Die hervorgehobenen Passagen sind die PROLOG-Formulierung für den jeweiligen Sachverhalt.

2.1.1 Fakten

sokrates war ein grieche.

```
grieche(sokrates).
```

clocksin ist ein autor.

```
autor(clocksin).
```

prolog ist eine Programmiersprache.

```
programmiersprache(prolog).
```

cos(x) ist eine Ableitung von sin(x).

```
ableitung(sin(x),cos(x)).
```

boris spielt tennis.

```
spielen(boris,tennis).
```

adam und eva sind eltern von kain.

```
eltern(adam,eva,kain).
```

adam und eva sind eitern von abel.

```
eltern(adam,eva,abel).
```

Die Form, in der hier Fakten in PROLOG dargestellt werden, nennt sich Prädikatendarstellung.

Prädikate sind: `grieche`, `autor`, `programmiersprache`, `ableitung`, `spielen`, `eltern`. Die Elemente in den Klammern werden *Argumente* genannt.

Die zweite Darstellungsform in PROLOG ist die Operatorendarstellung:

Prädikatendarstellung: `is(5,+(2,3)).`
Operatorendarstellung: `5 is 2 + 3.`

Dabei kann jedes Prädikat vom Programmierer zum Operator erklärt und eine bestimmte Operatorenpriorität verliehen werden.

Die Aussagen in Prädikaten- und Operatorendarstellung können aus folgenden Elementen bestehen:

Integers: Positive ganze Zahlen.

Atome: einfachste Elemente, die sich nicht weiter zerlegen lassen, z.B. `sokrates`, `+` , `:-` , `'PROLOG'` . Wenn Atome mit einem Buchstaben anfangen, muß es ein *Kleinbuchstabe* sein.

Variable: Variable werden erst im Laufe der Verarbeitung mit Inhalten belegt. Variablennamen müssen mit einem *Gro buchstaben* oder einem speziellen Zeichen, dem Jokerzeichen, anfangen.

Mit Variablen können jedoch auch verallgemeinerte Fakten abgelegt werden, z. B.

```
0 is X * 0.
```

was heißt, daß jede Zahl mit Null multipliziert Null ergibt.

2.1.2 Regeln

Jemand, der etwas stiehlt, ist ein Dieb.

```
dieb(Jemand) :- stehlen(Jemand,Etwas).
```

(`Jemand` und `Etwas` sind Variable.)

A und B sind Geschwister, wenn A und B gleiche Eltern haben.

```
geschwister(A,B) :- eltern(V,M,A),eltern(V,M,B).
```

$x'y + xy'$ist eine Ableitung von xy, wenn x' Ableitung von x ist und y' Ableitung von y ist.

```
ableitung(X*Y,X1*Y+Y1*X) :-
    ableitung(X,X1),ableitung(Y,Y1).
```

Wir haben hier zwei neue Operatoren kennengelernt:

`:-` kennzeichnet eine Regel.

`a :- b.` bedeutet, daß `a` wahr ist, wenn `b` wahr ist.

Das Komma kennzeichnet ein logisches UND.

`a :- b,c.` bedeutet, daß `a` wahr ist, wenn `b` und `c` wahr sind.

Alle diese Regeln und Fakten, aus denen letzten Endes PROLOG-Programme bestehen, werden in einer internen Datenbank abgelegt. Der Benutzer kann nun Fragen an PROLOG stellen, die PROLOG auf Grund der gespeicherten Regeln und Fakten beantwortet.

Ist Sokrates ein Grieche?

```
?-grieche(sokrates).
```

PROLOG antwortet hier mit `yes.` Wäre die Frage

```
?-roemer(sokrates).
```

so wäre die Antwort `no.`

Enthält die Frage eine oder mehrere Variable, so gibt es mehrere Möglichkeiten, die Frage zu beantworten.

Wer hat Adam und Eva als Eltern?

```
?-eltern(adam,eva,Wer).
```

wird zunächst beantwortet mit

```
eltern(adam,eva,kain).
```

Der Benutzer hat nun die Möglichkeit, die Antwort zu akzeptieren durch Drücken der RETURN -Taste, oder eine weitere Antwort durch Eingabe des Semikolons (logisches ODER) zu verlangen. In unserem Fall wäre die zweite Antwort

```
eltern(adam,eva,abel).
```

und die dritte Antwort

```
no.
```

Stellen wir jetzt die Frage:

Sind Kain und Abel Geschwister?

```
?-geschwister(kain,abel).
```

In der Datenbank ist die Regel

```
geschwister(A,B) :-
    eltern(V,M,A),eltern(V,M,B).
```

enthalten. Zur Beantwortung der Frage wird der Variablen `A` der Wert `kain` und `B` der Wert `abel` zugewiesen. PROLOG versucht zunächst, den Regelteil

```
eltern(V,M,kain)
```

zu beweisen und findet in der Datenbank die Regel

```
eltern(adam,eva,kain).
```

Damit sind auch die Variablen `V` und `M` belegt. Dann wird versucht, den zweiten Regelteil zu beweisen:

```
eltern(adam,eva,abel)
```

Dies ist als Faktum in der Datenbank. Damit ist die Regel bewiesen und PROLOG antwortet mit *yes*.

Stellen wir uns vor, daß wir statt unserer vier biblischen Personen größere Stammbäume gespeichert und Fragen dazu haben. Hier ist es durchaus möglich, daß infolge von Namensgleichheiten PROLOG in einen falschen Zweig des Stammbaums gerät, der nicht zur Lösung führt. Das bezeichnet man als Sackgasse. PROLOG erkennt eine solche Situation und geht Schritt für Schritt zurück (Backtracking), um jeweils zu überprüfen, ob es noch eine Alternative für den eingeschlagenen falschen Weg gibt.

Diese Fähigkeiten von PROLOG bedeuten für den Programmierer, daß er sich nicht darum kümmern muß, wie die Lösung erreicht wird. PROLOG sucht die Lösung aufgrund der Regeln und Fakten, ähnlich wie der Ausgang in einem Labyrinth gesucht wird.

Der Leser mag sich überlegen, wie PROLOG die Antwort auf die Frage

```
?-ableitung(sin(x)*sin(x),A).
```

findet. Die Antwort lautet:

```
ableitung(sin(x)*sin(x),
    cos(x)*sin(x)+sin(x)*cos(x)).
```

Regeln können auch rekursiv angewendet werden:

```
vorfahr(X,Y) :- eltern(Y,_,X).
vorfahr(X,Y) :- eltern(_,Y,X).
vorfahr(X,Y) :- eltern(Z,_,X),vorfahr(Z,Y).
vorfahr(X,Y) :- eltern(_,Z,X),vorfahr(Z,Y).
```

Das Zeichen '_' ist der Joker. Dies ist eine Variable, die überall paßt.

Die Frage

```
?-vorfahr(X,Y).
```

findet alle Vorfahren von `X`, Eltern, Großeltern, usw. Die Regel `vorfahr` wird nicht nur auf `X` selbst angewendet, sondern auch auf die Eltern von `X`, dann die Großeltern, usw.

2.1.3 Der Match

Eine sehr leistungsfähige Operation ist der 'Match'. Damit können Strukturen verglichen werden. Der 'Match' ist rekursiv definiert, d.h. der Vergleich von Strukturen wird aufgelöst in Vergleiche zwischen Einzelelementen. Befinden sich unter diesen Elementen Variablen, so wird der Inhalt der Variable für den 'Match' verwendet. Falls die Variable noch keinen Wert hat, bekommt sie den Wert ihres Gegenparts zugewiesen. Beispiel:

`eltern(X,Y,abel) = eltern(adam,eva,Z)`

resultiert in: `X = adam`, `Y = eva`, `Z = abel`.

Treffen zwei Variable aufeinander, so werden sie 'geshared', d.h. sie werden für die künfige Verarbeitung als eine Variable betrachtet. Variablenzuweisungen sind jeweils nur innerhalb einer Regel oder eines Fakts gültig, d.h. gleichnamige Variablen in verschiedenen Regeln können verschiedene Werte haben.

PROLOG wendet intern den 'Match' bei der Auswahl der Regeln an, aber es ist auch möglich, den Match mittels Gleichheitszeichen anzusprechen.

2.1.4 Listen

Eine Liste ist eine Aufkettung verschiedener Elemente von beliebiger Länge. Die Schreibweise ist `[a,b,c,d]`. Das letzte (nicht dargestellte) Element einer solchen Liste ist die leere Liste `[]`.

Ein wichtiger Operator in Zusammenhang mit Listen ist der Listenseparator `|` . Er trennt den Anfangs- vom Endteil einer Liste. So sagt `L = [a,b|L1]` aus, daß Liste `L`

mit den Elementen `a` und `b` anfängt und der Rest aus Liste `L1` besteht. So kann man Listen in Bestandteile zerlegen oder auch durch Anfügen Listen verlängern. Beispiel:

```
append([],L,L).
append([Elem|L1],L2,[Elem|L3]) :-
    append(L1,L2,L3).
```

`append` sagt aus, daß `L3` eine Verkettung von `L1` und `L2` ist. Die Definition ist rekursiv: Ist `L1` leer, ist `L3` gleich `L2`. Ist `L1` nicht leer, so wird von `L1` und `L3` das erste Element abgespalten (muß bei beiden gleich sein) und `append` auf die so verkürzten Listen angewandt. Man kann mit dieser Definition in verschiedene Richtungen arbeiten:

`append(L1,L2,X)`

legt die Verkettung von `L1` und `L2` in Variable `X` ab.

`append(X,Y,L)`

liefert alle möglichen Zerlegungen der Liste `L`.

2.1.5 Primitive

Wer die konventionellen Computersprachen kennt, wird sich bereits gefragt haben, wo denn die Befehle der Sprache PROLOG sind. Tatsächlich gibt es Befehle, um den Computer zu bestimmten Aktionen zu veranlassen. Diese Befehle sind die sogenannten Primitive. Primitive gibt es für verschiedene Zwecke:

- Ein- und Ausgabe einzelner Zeichen und ganzer Ausdrücke.
- Umschalten der Ein- und Ausgabe auf Dateien oder auf angeschlossene Geräte.
- Einlesen von Regeln und Fakten aus einer Bibliothek in die interne Datenbank.
- Anzeigen, Löschen, Erstellen und Hinzufügen von Regeln und Fakten. Damit ist es möglich, daß ein PROLOG-Programm seinerseits den Inhalt der internen Datenbank verändert.
- Vergleiche und Arithmetik.
- Testen von PROLOG Programmen. (Spypunkte und Trace).
- Dazu kommen noch Primitive, die die Fähigkeiten des jeweiligen Computersystems ausnutzen. So gibt es in PROLOG 64 Primitive für Turtlegraphik und die Klangmöglichkeiten des C64.

Die Primitive sind als eingebaute Fakten verfügbar, d.h. wenn man ein Zeichen ausdrucken will, beweist man die Druckbarkeit des Zeichens. Die Frage

```
?-put(65).
```

druckt das Zeichen 'a'. Enthält die Datenbank die Regel

```
hello :- put(62),put(69),put(76),put(76),put(79).
```

so bewirkt die Frage

```
?-hello.
```

die Ausgabe von '`hello`' .

Hier sieht man schon, wie einfach Programme verknüpft werden können. Sie werden einfach innerhalb einer Regel untereinandergeschrieben. Die Übergabe von Zwischenergebnissen erfolgt über Variablen.

So besteht z.B. ein Gesprächsführungsprogramm aus drei Unterprogrammen:

`readin` liest einen Satz ein und wandelt ihn in Listenformat.

`process` analysiert die Eingabe und produziert die Ausgabe in Listenformat.

`printout` bringt die Ausgabe in das normale Satzformat.

Das ganze Programm sieht dann so aus:

```
dialog :- repeat,readin(X),process(X,Y),
    printout(Y),fail.
```

Die '`repeat`, ..., `fail`'-Kombination bewirkt, daß das Programm nicht zu Ende kommt, sondern nach Ausgabe eines Satzes jeweils eine neue Eingabe verlangt.

Damit ist unsere kurze Einführung zu Ende. Zum vertiefenden Studium von PROLOG empfiehlt sich das Ausprobieren.

Weiterführende Literatur:

Programming in PROLOG
W.F. Clocksin, C.S. Mellish
Springer Verlag, Heidelberg, New York 1985

3 Programmiermethoden

3.1 Die axiomatische Methode

Ein Beispiel für diese Methode ist das Programm zum formalen Differenzieren in der Bibliothek '`math`'.
Die Schlußregeln des Differenzierens sind direkt als PROLOG-Regeln abgelegt.

3.2 Die indirekte axiomatische Methode.

Hier werden die Schlußregeln als PROLOG-Fakten abgelegt. Ein PROLOG-Programm bewirkt die Anwendung der Fakten als Schlußregeln.
Ein Beispiel für diese Technik ist das Programm zur Formelvereinfachung in der Bibliothek '`math`'.
Die meisten Expertensysteme sind so aufgebaut. Die Methode läßt sich so erweitern, daß die Fakten auf der Floppy stehen und diese jeweils durchsucht wird. Wegen der Laufzeit ist dies jedoch nur in Sonderfällen interessant.

3.3 Die Notizblock-Methode

Suchverfahren nach der axiomatischen Methode können in Schleifen laufen. Beispielsweise gibt es Labyrinthe, in denen man ständig im Kreis laufen kann, ohne ans Ziel zu kommen. In diesem Fall ist es nützlich, sich zu merken, wo man schon gewesen ist. Als Merker benutzt man eine Liste, in der man einträgt, wo man bereits war.
Beispiele für diese Technik sind die Programme '`maze`' und '`go`' in der Bibiliothek '`search`'. (Den berühmten Ariadnefaden führt PROLOG selbst zum Zweck des Backtracking.)

3.4 Die Bewertungsmethode

Oft ist man nicht an irgendeiner Lösung interessiert, sondern an der besten. Oder die Lösung wird besonders schnell erreicht, wenn man bestimmte Wege bevorzugt. In diesem Fall führt man eine Bewertungsfunktion ein und entscheidet anhand dieser Bewertungsfunktion, welche von mehreren möglichen Alternativen ausgewählt wird.
Ein Beispiel für diese Methode ist das Programm '`path`' in der Bibliothek '`search`'.

3.5 Die rekursive Methode

Diese Methode eignet sich besonders gut zum Verarbeiten von Listen und Zahlenproblemen. Man verarbeitet jeweils das erste Listenelement und wendet dieselbe Funktion auf den Rest der Liste an. Die leere Liste gilt als Abbruchkriterium.

Beispiele: Programm '`play`' in Bibliothek '`music`'.

Auch kleinere Schleifen lassen sich auf diese Weise programmieren. Beispiel: '`circle`' in Bibliothek '`graph`'

Werden jedoch die Schleifen zu groß, droht ein Stacküberlauf. In diesem Falle empfiehlt sich die

3.6 retract-assert-Methode

```
retractall(counter($)),asserta(counter(1)),
    repeat,retract(counter(X)),process(X),
    X1 is X+1,asserta(counter(X1)),X1 = endwert.
```

Hier werden zunächst eventuelle Leichen entfernt, dann der Startwert gesetzt. Die eigentliche Verarbeitung sei in `process(X)` enthalten, das solange ausgeführt wird, bis `X1` den Endwert erreicht. Auf diese Weise wird der aktuelle Wert des Zählers in der PROLOG-Datenbank gehalten.

3.7 Die Generator-Filter Methode

Bei dieser Methode werden erst alle Kombinationsmöglichkeiten von Lösungselementen erzeugt. Bei jeder neuen Kombination wird überprüft, ob die Lösung erreicht ist.

Beispiel für diese Technik ist das Programm '`sort`' in der Bibliothek '`mapping`' .

Dort werden alle Permutationen einer gegebenen Zahlenfolge erzeugt und bei jeder neuen Permutation überprüft, ob die Sortierreihenfolge erfüllt ist.

Man sollte diese Methode nur anwenden, wenn einem nichts besseres einfällt.

3.8 Die Pipeline-Methode

Dies ist eine Methode zur Verknüpfung von Unterprogrammen. Die Zwischenergebnisse werden jeweils mittels Variablen übergeben.

Beispiel: '`translate`' in Bibliothek '`logic`'.

Eine rekursive Anwendung der Pipeline-Methode ist das Programm '`sentence`' in der Bibliothek '`grammar`'. Allerdings müssen Sie sich '`sentence`' nach dem `consult(grammar)` ansehen, da es erst während des consult in PROLOG-Regeln übersetzt wird.

4 Syntax

4.1 Zugelassene Zeichen

Alle Zeichen die auf der Tastatur des C64 im Groß-/Kleinschreibungsmodus erreichbar sind. Die Zeichen sind in folgende Gruppen unterteilt:

- Großbuchstaben
- Kleinbuchstaben
- Ziffern
- Sonderzeichen und Grafikzeichen

4.2 Integers

Integers bestehen aus Ziffern und dürfen im Bereich von 0 – 8191 liegen.

4.3 Atome

Atome können folgende Form haben:

- Zeichenfolge aus Buchstaben und Ziffern, das erste Zeichen muß ein Kleinbuchstabe sein.
- Zeichenfolge aus Sonder- und Grafikzeichen
  `:- == /=`
- Jede beliebige Zeichenfolge,die von Apostrophen eingeschlossen ist.
  `'PROLOG' '4e' 'mueller-thurgau'`

4.4 Variablen

Variablen werden als Zeichenfolgen von Buchstaben und Ziffern dargestellt. Das erste Zeichen muß ein Großbuchstabe oder das Jokerzeichen '_' (Commodore/Klammeraffe) sein. (Das Jokerzeichen gilt als Großbuchstabe.)

```
X   Wer_oder_was   _19   _
```

Das Zeichen _ allein kennzeichnet den Joker, eine Variable, die zu allem paßt und nie belegt wird.

4.5 Strukturen

Strukturen bestehen aus einem Prädikat gefolgt von einem oder mehreren Argumenten in Klammern. Das Prädikat muß ein Atom sein.

```
grieche(sokrates)
eltern(X,Y,kain)
ableitung(sin(x),Z)
+(X,25)
```

4.6 Operatorenschreibweise

```
X + 25
```

4.7 Listen

Listen sind eine besondere Form von Strukturen:

```
[a,b,c,d]
[d(30,50,3),d(33,50,3),d(37,100,3)]
[A | B]
```

Das Zeichen `|` (Shift/-) trennt Listenkopf von Listenrest.

4.8 Strings

`"abc"`

Strings werden intern als Listen dargestellt. `"abc"` ist äquivalent zu `[65,66,67]`.

4.9 Weiteres

Das Grafikzeichen ▨ (Shift + £) ist das EOF-Zeichen.

Kommentare sind zwischen `/*` und `*/` anzugeben und können mehrzeilig sein.

5 Eingebaute Funktionen

5.1 Allgemeine Primitive

`consult X` Consult dient der Eingabe von Regeln und Fakten von der Tastatur her oder von einer Bibliothek (siehe Kapitel 7 Ein- und Ausgaben auf Seite 25).

In PROLOG 64 besitzt '`consult`' einen Benutzerausgang: Wird vom Benutzer die Funktion '`tco`' definiert, so werden alle Eingaben in consult vor dem Eintragen in den Datenspeicher zuerst mit '`tco`' bearbeitet. Auf diese Weise können z.B. Grammatikregeln eingegeben werden — siehe Bibliothek `tcogrammar`.

`reconsult X` Mit '`reconsult`' werden Prädikate, die bereits im Datenspeicher enthalten sind, durch neue ersetzt. Bei Eingabe der ersten Regel oder des ersten Fakts eines bestimmten Prädikats werden zunächst alle vorhandenen Regeln und Fakten dieses Prädikats gelöscht. Ansonsten gilt das bei '`consult`' Gesagte.

`true` Dieses Prädikat ist immer wahr und ziemlich überflüssig.

`fail` Dieses Prädikat ist immer falsch.

Es findet sich hauptsächlich in der 'cut-fail' Kombination '`!,fail`' und in Verbindung mit '`repeat`' zur Programmierung von Schleifen.

`var(X)` `var(X)` ist wahr, wenn `X` eine freie Variable ist.

`nonvar(X)` ist wahr, wenn `X` keine freie Variable ist.

`atom(X)` ist wahr, wenn `X` ein Atom ist.

`integer(X)` ist wahr, wenn `X` eine Integerzahl ist.

`atomic(X)` ist wahr, wenn `X` ein Atom oder eine Integerzahl ist.

`listing a` listet alle Fakten und Regeln des Prädikats `a` auf.

clause(X,Z) findet Regeln vom Typ `X :- Y.`
`X` muß das jeweilige Prädikat ausreichend spezifizieren.
Ein Backtrack auf `clause(X,Y)` findet die nächste passende Regel.

asserta(X), assertz(X). fügen die Regel oder den Fakt X dem Datenspeicher zu, `asserta` an den Beginn, `assertz` an das Ende der Prädikatdefinition.

retract(X) entfernt Regel oder Fakt `X` aus dem Datenspeicher. Dabei muß `X` ausreichend definiert sein.

5.2 Listen

X . Y '.' ist der Listenoperator. `X` ist der Kopf der Liste und `Y` der Schlußteil.

[] steht für eine leere Liste.

|X,Y,Z| ist eine bequeme Schreibweise für `.(X,.(Y,.(Z,[])))`

[X,Y|Z] ist eine Liste mit Anfangsteil `X,Y` und Schlußteil `Z`. Diese Schreibweise wird vor allem benutzt, um Listen in Einzelteile zu zerlegen, bzw. Listen aus Elementen oder anderen Listen zusammenzusetzen.

functor(T,F,N) `T` ist ein Funktor mit Kopf `F` und `N` Argumenten. `functor` kann benutzt werden, um Funktoren aufzubauen (`T` frei), oder Funktoren zu zerlegen (`F` und `N` frei).

arg(N,T,A) `A` ist das `N`-te Argument des Funktors `T`. Damit kann man sowohl ein Argument aus einem Funktor herauslösen (`A` frei) oder die noch unbestimmten Argumente eines mit '`functor`' aufgebauten Funktors konkretisieren (`A` gebunden).

X =.. L `X` ist ein Funktor, `L` eine Liste, deren erstes Element der Kopf des Funktors ist, die weiteren Elemente die Argumente. Ist `X` frei, wird aus `L` ein Funktor zusammengesetzt, ist `L` frei, wird Funktor `X` in eine Liste zerlegt.

name(A,L) `L` ist die Zerlegung des Atoms `A` in Zeichen. Da Zeichen in PROLOG als Integerzahlen dargestellt werden, ist `L` eine Liste von Integerzahlen. Ist `L` frei, so wird `A` zerlegt; ist `A` frei, so wird aus einer Liste von Zahlen ein Atom zusammengesetzt.

`!` Cut Der Cut friert bisherige Lösungswege ein. Beispiel:

```
a :- b,c,d,!,e
```

Der Cut verhindert einen Backtrack zwischen `a` und dem Cut — stellt sich `e` als logisch falsch heraus, so ist ein Backtrack von `d,c,b` nicht mehr möglich, sondern `a` wird ebenfalls als logisch falsch betrachtet.

`repeat` `repeat` eröffnet eine Programmschleife.

Beispiel:

```
repeat,get(13)
```

ist gleichwertig zu `skip(13)`

`X,Y` UND-Verbindung von Prädikaten. In PROLOG 64 sind UND-Verbindungen intern als Listen dargestellt, und zwar ohne leere Liste am Ende.

Es wird weniger Speicherplatz benötigt, und das Konstruieren von Ausdrücken innerhalb eines Programms ist einfacher. Die Notation `(X,Y)` ist nicht erlaubt, statt dessen muß `X,Y` geschrieben werden.

Beispiel:

statt `call((X,Y))` wird `call(|X,Y|)` geschrieben.

`X;Y` ODER-Verbindung von Prädikaten.

`call(X)` Aufruf des Ausdrucks `X`. `X` kann eine Variable sein, die erst während der Programausführung belegt wird.

`not(X)` ist wahr, wenn `call(X)` falsch ist,

`X = Y` MATCH von `X` zu `Y`.

Sind `X` und `Y` Atome oder Zahlen, so ist `X = Y` wahr, wenn `X` und `Y` gleich sind. Ist `X` oder `Y` eine freie Variable, so ist `X = Y` immer wahr und `X` bzw. `Y` bekommen den Wert des Partners zugewiesen.

Sind sowohl `X` und `Y` freie Variablen, so ist `X = Y` wahr und `X` und `Y` werden zusammengebunden (shared).

Diese Regeln sind entsprechend auf Funktoren und Listen erweitert, indem die einzelnen Elemente der Funktoren und Listen paarweise mit MATCH verarbeitet werden.

Beispiel:

`f(a,X) = f(Y,b)` ist wahr und belegt `X` mit `b`, `Y` mit

`a`, sofern `X` und `Y` frei sind.

`f(X) = f(Y)` ist wahr, `X` wird mit `Y` gebunden, sofern `X` und `Y` frei sind.

`X /= Y` ist wahr, wenn `X = Y` falsch ist.

`X == Y` Identität, wie `X = Y`, nur daß `X == Y` (`X,Y` frei) falsch ist. Wahr ist `X == Y`, wenn `X` bereits mit `Y` gebunden ist.

`X /== Y` ist wahr, wenn `X == Y` falsch ist.

`X > Y`

`X >= Y`

`X < Y`

`X =< Y` sind wahr, wenn die entsprechenden Bedingungen wahr sind.

`get0(X)` dient zur Eingabe eines Zeichens.

`get(X)` dient zur Eingabe eines abdruckbaren Zeichens. PROLOG wartet, bis ein abdruckbares Zeichen eingegeben wurde.

`skip(X)` PROLOG wartet, bis ein Zeichen mit Zeichencode `X` eingegeben wurde. z.B. `skip(13)` wartet auf die RETURN -Taste.

`read(X)` liest einen PROLOG-Ausdruck. Dabei gelten die im Kapitel Ein- und Ausgabe genannten Regeln.

`put(X)` gibt das Zeichen mit Zeichencode `X` aus.

`nl` beendet eine Zeile und führt einen Zeilenvorschub aus.

`tab(X)` gibt `X` Leerzeichen aus.

`write(X)` gibt einen PROLOG-Ausdruck aus. Dabei werden Operatoren in Operatorenschreibweise ausgegeben.

`display(X)` gibt einen PROLOG-Ausdruck aus. Dabei werden Operatoren in Funktor-Schreibweise ausgegeben.

see(X) schaltet das Eingabemedium auf die in X definierte Datei um (siehe Kapitel 7 Ein- und Ausgaben auf Seite 25).

seeing(X) legt den Namen des aktuellen Eingabemediums in X ab.

seen schließt das aktuelle Eingabemedium und schaltet auf 'user' zurück.

tell(X) schaltet das Ausgabemedium auf die in X definierte Datei um (siehe Kapitel 7 Ein- und Ausgaben auf Seite 25).

telling(X) legt den Namen des aktuellen Ausgabemediums in X ab.

told schließt das aktuelle Ausgabemedium und schaltet auf 'user' zurück.

X is Y wertet den arithmetischen Ausdruck Y aus und weist ihn X zu.

Der zulässige Zahlenbereich in PROLOG 64 ist 1 – 8191. Wird dieser Zahlenbereich während einer Rechnung unter- oder überschritten, wird sooft 8192 addiert oder subtrahiert, bis der gültige Zahlenbereich wieder erreicht wird.

Außer mit 'is' können in PROLOG 64 arithmetische Ausdrücke in Operanden folgender Funktionen auftreten:

=<, >, >=, <, put, skip, tab, move, draw, plot, screen, pen, random, seed, sound, poke.

Arithmetische Ausdrücke können aus folgenden Operatoren aufgebaut werden:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
mod	Modulo- (Rest-) Funktion

5.3 Testfunktionen

Die folgenden Funktionen dienen dem Test von PROLOG-Programmen (siehe auch Kapitel 11 Das Testsystem auf Seite 33).

trace schaltet den PROLOG-Trace (Ablaufverfolger) ein.

notrace schaltet den PROLOG-Trace aus.

spy X setzt einen Spion auf Prädikat X.

nospy X entfernt den Spion auf Prädikat X.

debugging zeigt alle Spione an.

nodebug entfernt alle Spione.

5.4 Definition von Operatoren:

op(P,S,O) definiert Priorität (P) und Spezifikation (S) eines Operators (O).
Zulässige Prioritäten: 1 – 63
Zulässige Spezifikationen:

fx	Prefix-Operator
xf	Su x-Operator
xfy	rechtsassoziativer Infix-Operator
yfx	linksassoziativer Infix-Operator

Folgende Definitionen sind bereits eingebaut:

Operator	Priorität	Spezifikation
:-	63	xfx
?-	63	fx
;	62	xfy
]	62	xf
|	62	xfy
,	61	xfy
spy	58	fx
nospy	58	fx
consult	56	fx
reconsult	56	fx
listing	57	fx
not	30	fx
.	25	xfy
is	20	yfx
=..	20	yfx
=	20	yfx
/=	20	yfx
==	20	yfx
/==	20	yfx
<	20	yfx

Operator	Priorität	Spezifikation
`=<`	20	yfx
`>`	20	yfx
`>=`	20	yfx
`=:=`	20	yfx
`=/=`	20	yfx
`-`	16	yfx
`+`	16	yfx
`*`	11	yfx
`/`	11	yfx
`mod`	6	yfx

6 PROLOG 64 Erweiterungen

PROLOG 64 kennt die üblichen eingebauten Funktionen und besitzt darüberhinaus noch einige weitere eingebaute Funktionen. Außerdem sind noch nützliche PROLOG-Programme fest in PROLOG 64 einprogrammiert.

Zusätzlich eingebaute Funktionen (primitives) :

`dir` gibt eine Übersicht über den Inhalt der Datenbank.

`=:=` vergleicht zwei arithmetische Ausdrücke auf Gleichheit.

`=/=` vergleicht zwei arithmetische Ausdrücke auf Ungleichheit.

`random(n,X)` erzeugt eine Zufallszahl zwischen 1 und n .

`seed(s)` setzt einen Anfangswert für den Zufallszahlengenerator.

`poke(a,b)` bringt den Wert `b` (0 – 255) in den Speicher des C64 und zwar nach Adresse `a+$D000`. `a` muss zwischen 0 und 4095 liegen.

`poke(a,X)` liest den Bytewert von Adresse `a+$D000`.
Sie können `poke` z.B. dazu benutzen, Paddles abzufragen, den VIDEO-Chip zu beeinflussen, usw.
oder: `poke(12*256+4,X),seed(X)` initialisiert den Zufallsgenerator mit einem zufälligen Wert (timer).

`sound(t,d,k)` siehe Kapitel 10 Die Klangmöglichkeiten auf Seite 31.

`pause(d)` siehe dort

`screen(b,c)` siehe Kapitel 9 Grafikbetrieb auf Seite 29.

`pen(c)` siehe dort

`pen(X)` siehe dort

`move(a,l)` siehe dort

`draw(x,y)` siehe dort

`plot(x,y)` siehe dort

`plot(X,Y)` siehe dort

`clear` löscht den PROLOG-Datenspeieher (reset).

`user` Editor einschalten.

`eof` End of File: Im consult-Mode wird mit mit `eof` der `consult` beendet, im normalen Mode wird mit `eof` PROLOG beendet und nach BASIC zurückgeschaltet. `X=eof` kann zur Abfrage auf Dateiende benutzt werden.

`islist(l)` stellt fest, ob l eine Liste ist.

error nach einer PROLOG-Fehlermeldung wird das laufende Programm abgebrochen. Findet PROLOG in seiner Datenbank Regeln vom Typ: `error :- .....` , so werden diese ausgeführt.

In vielen eingebauten Funktionen dürfen in PROLOG 64 auch arithmetische Ausdrücke als Parameter angegeben werden, die dann vor der Ausführung der Funktion ausgerechnet werden.

Im einzelnen sind das die Funktionen:

`is`, `=<`, `<`, `>=`, `>`, `put`, `skip`, `tab`, `random`, `seed`, `poke`, `move`, `draw`, `plot`, `screen`, `pen`, `sound`, `pause`.

Innerhalb arithmetischer Ausdrücke können auch Listen als Operanden angegeben werden. In diesem Fall wird das erste Listenelement als Operand verwendet. Da `"a"` in PROLOG eine Liste ist, ist deshalb auch `put("a")` möglich. Auf diese Weise kann auch die Cursorsteuerung mittels `put` durchgeführt werden (ähnlich BASIC). Oder es ist die Konstruktion `get(X),X is "y"` möglich.

Folgende Prolog-Programme sind fest einprogrammiert: (Mit '`listing`' kann man sie anzeigen.)

retractall(a) entfernt alle Regeln, deren Kopf mit mit `a` 'matcht', bzw. alle Fakten, die mit `a` 'matchen'.

append(L1,L2,L3) `L3` ist die Verkettung der Listen `L1` und `L2`.

last(x,L) `x` ist das letzte Element der Liste `L`.

nextto(x,z,L) stellt fest, ob die Elemente `x` und `z` in Liste `L` benachbart sind.

rev(L1,L2) `L1` ist die Umkehrung der Liste `L2`.

efface(x,L1,L2) Liste `L2` ergibt sich aus Liste `L1` durch Entfernen des ersten Vorkommens des Elementes `x`.

delete(x,L1,L2) Liste `L2` ergibt sich aus Liste `L1` durch Entfernen aller Vorkommen von Element `x`.

subst(x,L1,y,L2) Liste `L2` entsteht aus Liste `L1`, indem alle Vorkommen des Elementes `x` durch Element `y` ersetzt werden.

sublist(L1,L2) stellt fest, ob `L1` eine Unterliste von `L2` ist.

member(x,L) stellt fest, ob `x` in Liste `L` vorkommt.

In PROLOG 64 können alle eingebauten Funktionen vom Benutzer erweitert werden, indem Regeln bzw. Fakten dieses Prädikats der Datenbank zugefügt werden. Die Benutzerregeln werden immer als erste ausgeführt, die eingebaute Funktion tritt in Kraft, wenn keine Benutzerregel greift.

Beispiel:

```
[] < [B|_].
[A|A1] < [B|B1] :- A < B,!.
[A|A1] < [A|B1] :- A1 < B1.
```

erweitert die Definition der Relation 'kleiner' von Integer-Zahlen auf Strings bzw. Listen von Zahlen.

7 Ein- und Ausgaben

Normalerweise erfolgen die Eingaben von Tastatur und die Ausgaben auf Bildschirm. Beim Eingeben von der Tastatur beachten Sie bitte, daß das Betriebssystem des C64 eine maximale Eingabelänge von 80 Zeichen pro logischer Zeile erlaubt. Drücken Sie deshalb spätestens nach 80 Zeichen die Returntaste. Solange Sie die Eingabe nicht mit Punkt abgeschlossen haben, fordert PROLOG weitere Eingabezeilen. Somit können PROLOG-Ausdrücke länger als 80 Zeichen werden, maximal 1000 Zeichen (ein Bildschirm). Nachdem Sie einen Prolog-Ausdruck mit Punkt abgeschlossen haben, wird er ausgeführt. Nach der Ausführung antwortet PROLOG mit '`yes`', '`no`' oder, falls die Eingabe Variablen enthält, zeigt es den Ausdruck mit belegten Variablen an. Im letzten Falle können Sie mit '`;`' ein Backtrack veranlassen oder mit RETURN akzeptieren. Wartet Prolog auf eine Anweisung Ihrerseits, so zeigt es das durch die Ausgabe von '`?-`' (Prompting).

Wollen Sie dem PROLOG-System neue Regeln und Fakten zufügen, so können Sie das mit `asserta` oder `assertz` tun. Eine andere Möglichkeit ist die Verwendung von '`consult`'. `consult(user)` bewirkt, daß Sie direkt von der Tastatur aus Regeln und Fakten eingeben können. Dabei erscheint der Prompt '`?-`' nicht.

Haben Sie alle Regeln und Fakten eingegeben, so beendigen Sie '`consult`' durch Eingabe von '`eof.`' oder des EOF-Zeichens ▨ (Shift + £).

Wollen Sie Regeln und Fakten aus einer Bibliothek einlesen, so geben Sie ein:

```
consult(name).
```

Dabei ist '`name`' der Name der Bibliothek auf der Floppy. Wenn PROLOG alle Regeln und Fakten dieser Bibliothek eingelesen hat, so meldet es sich wieder mit '`yes`' und '`?-`'. Eine Bibliothek kann Ihrerseits wieder die Anweisung enthalten:

```
?-reconsult(name2).
```

Dies bewirkt, daß während des Konsultierens der ersten Bibliothek auch die Bibliothek '`name2`' konsultiert wird. (`reconsult` deshalb, weil die gleiche Bibliothek vielleicht schon einmal eingelesen wurde.)

Beachten Sie, daß diese `consult`-Schachtelung nur maximal zweimal möglich ist, da das Betriebssystem der Floppy maximal drei Dateien gleichzeitig geöffnet halten kann.

Ist in einer Bibliothek, die Sie konsultieren, ein Fehler enthalten, so bricht der '`consult`' mit einer Fehlermeldung ab.

Dabei wird der PROLOG-Eingabepuffer auf Bildschirm ausgegeben. So wird der fehlerhafte Ausdruck bis zur Fehlerstelle angezeigt, und eine leichte Fehlerlokalisierung ist möglich.

Bei der Eingabe von Listen sollten Sie darauf achten, daß die Listen nicht mehr als 63 Elemente enthalten. Längere Listen müssen Sie per Programm aufbauen.

Mit der eingebauten Funktion '`see`' können Sie in PROLOG-Programmen die Eingabe umschalten.

`see(user)` Tastatur

`see(name)` P-Datei Gerät 8 (Floppy)

`see(dn(g))` Datei `dn` auf Gerät `g`. Ist das Gerät eine Floppy, so muss `dn` eine sequentielle ASCII-Datei sein (`S`). Die Bedeutung verschiedener Gerätenummern entnehmen Sie bitte Ihrem C64-Handbuch.
Auch Gerätenummer 3 (Bildschirm) ist als Eingabegerät möglich. Dabei wird der Bildschirminhalt als Eingabequelle benutzt. Eine sinnvolle Verarbeitung ist dabei allerdings nur mit `get` bzw. `get0` möglich.

`see(user)` bzw. `seen` schalten die Eingabe auf die Tastatur. Dabei wird der Tastaturpuffer gelöscht.

Mit der eingebauten Funktion '`tell`' können Sie in PROLOG-Programmen die Ausgabe umschalten.

`tell(user)` Bildschirm

`tell(name)` P-Datei Gerät 8

`tell(dn(g))` Datei `dn` auf Gerät `g`.

Besondere Bedeutung hat Gerät 4 (Drucker). Z.B. bewirkt

```
tell(printer(4)),trace.
```

einen Trace auf Drucker.

Bei der Eröffnung eines Ausgabegerätes mit einer Gerätenummer größergleich 8 wird dem Gerät ein Dateieröffnungsbefehl geschickt (wie bei Floppy üblich). So kann auf ASCII-Dateien auf verschiedenen Floppys geschrieben werden. Mit

```
tell(dn(8,append))
```

kann an eine vorhandene ASCII-Datei angefügt werden.

Beachten Sie, daß maximal drei Floppy Dateien, bzw. eine Banddatei gleichzeitig geöffnet sein dürfen.

Sorgen Sie dafür, daß alle geöffneten Dateien auch wieder geschlossen werden. (`seen`, `told`)

Ein Beispiel für Dateiverarbeitung mit ASCII-Dateien ist die Bibliothek '`minishell`'.

8 PROLOG Bibliotheken

PROLOG-Bibliotheken dienen dazu, bestimmte häufig benutzte PROLOG-Anweisungsfolgen (Programme) zu speichern. Sie finden Beispiele solcher Bibliotheken auf Ihrer PROLOG-Diskette.

Im Anhang sind diese Bibliotheken aufgelistet. Wie können Sie Bibliotheken selbst erstellen? PROLOG-Bibliotheken werden mit dem C64-Editor erstellt. Dabei wird der Editor jedoch vom PROLOG-System modifiziert, damit die Umsetzung in BASIC-Tokens nicht durchgeführt wird.

Aus diesem Grund dürfen Sie den Editor nur von PROLOG aus benutzen.

Sie rufen den Editor durch Eingabe von '`user.`' auf. Nun meldet sich der Editor mit dem gewohnten Bildschirm. Erstellen Sie Ihre PROLOG Bibliotheken nun auf die von BASIC her gewohnte Weise.

Falls Sie ein neues Programm erstellen wollen, geben Sie das Kommando NEW ein.

Nach der Zeilennummer folgt ein PROLOG-Ausdruck. Dabei kann ein Ausdruck länger als eine Zeile sein. Das Ende eines Ausdrucks wird jeweils mit einem Punkt dargestellt.

Sie können das Ende einer Bibliothek mit dem EOF-Zeichen ▨ (Shift + £) oder '`eof`' kennzeichnen. PROLOG 64 erkennt aber auch ohne Markierung das Dateiende.

Sichern Sie Ihre Bibliotheken wie ein BASIC-Programm mit `SAVE` und `VERIFY`. Achten Sie darauf, daß Bibliotheksnamen nicht als Prädikate verwendet werden. Wollen Sie eine vorhandene Bibliothek ändern, so lesen Sie sie mit `LOAD`.

Die im BASIC-RAM stehenden Programme bzw. Bibliotheken bleiben bis zum nächsten Editieren erhalten. Allerdings wird der dafür benötigte Speicherplatz vom PROLOG-Stack abgezogen. Wollen Sie das verhindern, so geben Sie nach `SAVE` das Kommando `NEW` ein. Beim nächsten Editieren müssen Sie dann das Programm mit `LOAD` erneut in den Editierbereich laden.

Wollen Sie nach der Bibliotheksbearbeitung in das PROLOG-System zurück, so drücken Sie STOP/RESTORE.

Sie können auch Programme, die sie mit '`consult user`' während des PROLOG Betriebs erstellt haben, auf Floppy schreiben. Dazu geben Sie ein:

```
tell(dn(8)),listing(p1),listing(p2),...,told.
```

mit den jeweiligen Datei- und Prädikatsnamen. Die Prädikate werden auf eine sequentielle ASCII-Datei geschrieben. Sie können diese Datei mit dem mitgelieferten BASIC-Programm '`s to p`' in eine Programmdatei umsetzen. Damit haben Sie die

Möglichkeit, sie mit dem Editor weiter zu bearbeiten, z.B. Kommentare einzufügen. **Wichtig:** In einigen Fällen ist es nötig, für korrekte Schreibweise zu sorgen, da das Prädikat '`listing`' Atome, die nur mit Apostrophen eingegeben werden können, nicht mit Apostrophen ausgibt.

9 Grafikbetrieb

PROLOG 64 enthält einige eingebaute Funktionen zur Nutzung der Grafikmöglichkeiten des C64. Der Grafikbildschirm ist in 320 (0 – 319) horizontale Punkte und 200 (0 – 199) vertikale Punkte aufgeteilt. Die Grafik ist eine Turtlegrafik, der Anfangspunkt der (unsichtbaren) Turtle ist der untere linke Eckpunkt des Bildschirmfensters $(0, 0)$. Die Anfangsrichtung ist horizontal nach rechts (0).

`screen(b,c)` setzt die Hintergrundfarbe auf `b` und die Zeichen- bzw. die Turtlefarbe auf `c` und löscht den Grafikspeicher.

`pen(t)` setzt die Turtlefarbe `t`. `t=0` bedeutet, daß die Turtle nicht schreibt.

`pen(T)` legt die Turtlefarbe in Variable T ab.

`plot(x,y)` setzt einen einzelnen Punkt, ohne die Turtle zu beeinflussen.

`plot(X,Y)` legt die Position der Turtle in den Variablen ab.

`draw(x,y)` bewegt die Turtle zum Punkt (x, y), ohne die Richtung der Turtle zu beeinflussen.

`move(a,l)` dreht die Turtle erst um `a` Altgrad nach rechts und bewegt dann die Turtle um Länge `l` vorwärts.

PROLOG 64 schaltet bei jedem Grafikbefehl (außer `screen`) den Grafikbildschirm und bei jeder Textausgabe den Textbildschirm wieder ein.

Das heißt, wenn Sie das gezeichnete Bild betrachten wollen, müssen Sie PROLOG zum Anhalten bringen. Das geschieht mit der eingebauten Funktion '`pause(n)`' für $n/100$ Sekunden, oder mit '`skip(c)`' bis zum Drücken der Taste mit Tastencode `c`. Beispiel:

```
plot(160,100),skip(13)
```

zeichnet einen Punkt in die Bildschirmmitte und wartet auf die RETURN -Taste.

PROLOG 64 nimmt den für den Grafik-Betrieb erforderlichen Bildspeicher vom PROLOG-Stack weg, und zwar 12,5 kB. Wollen Sie keinen Grafikbetrieb mehr betreiben, so sollten Sie mit '`screen`' den Grafikspeicher freigeben.

Auf der PROLOG-Diskette finden Sie die Bibliothek '`graphic`'.

Diese enthält einige Beispiele für nützliche Grafik-Programme:

`triangle(h)` zeichnet ein gleichschenkliges Dreieck mit Höhe `h`.

`square(s)` zeichnet ein Quadrat mit Seitenlänge `s`.

`circle(r)` zeichnet einen Kreis mit Radius `r`.

`histogram(b,l)` zeichnet ein Balkendiagramm.
Die Breite der Balken ist `b`, die Höhe der einzelnen Balken ist in Liste `l` enthalten.

Die PROLOG 64 Grafik arbeitet intern mit Ganzzahlarithmetik. Dadurch kann es zu Rundungsfehlern in der Größenordnung von einem Bildpunkt kommen.

Bei komplizierten Zeichnungen können sich in ungünstigen Fällen die Ungenauigkeiten summieren.

Mollen Sie ganz genau arbeiten, so sollten Sie die Turtle ab und zu auf feste Bezugspunkte normieren. Dafür eignet sich '`draw`' sehr gut. Beispiel:

```
screen(1,6),repeat,draw(160,100),square(100),fail.
```

läßt ein Quadrat exakt um den Bildmittelpunkt rotieren.

9.1 Sprites

Sprites können Sie im Normalbetrieb verwenden (nicht im Grafikbetrieb). Die Form der Sprites und die Zuordnung der Speicherblöcke zu Spritenummern (Addressen 2040 – 2047) müssen im Lader definiert werden.

Benutzen können Sie die Speicherblöcke 11,13,14,15.

Mollen Sie weitere Speicherblöcke verwenden, müssen Sie vor Aufruf des Laders die BASIC-Anfangsadresse hoch setzen.

Die Kontrolle der Sprites (Ein- und Ausschalten, Farbe, Größe, Bewegung, Priorität, Kollision) geschieht von PROLOG aus mit '`poke`'.

Z.B. `poke(0,160)` setzt die X-Koordinate von Sprite 0 auf Bildschirmmitte.

10 Die Klangmöglichkeiten

Mit der eingebauten Funktion

`sound(t,d,k)` können sie Klänge erzeugen.

Dabei ist `t` die Tonnummer, wie in Ihrem C64-Handbuch angegeben. Die Verwendung von Nummern, statt Frequenzen oder Namen hat den Vorteil, daß Sie mit Addition oder Subtraktion einfach Intervalle erzeugen können.

`d` ist die Dauer des Tones in $d/100$ Sekunden,

`k` $(0-4)$ ist die Nummer der Klangfarbe. Welche der 5 Klangfarben wie klingt, ist im PROLOG-Lader festgelegt (siehe Kapitel 16 Anhang auf Seite 45).

In der Bibliothek '`music`' finden Sie Beispiele für die Verwendung von '`sound`', z.B. die Funktion '`play`' für das Spielen nach Tonnummern, bzw. '`play2`' für das Spielen nach Tonnamen mit Oktav- und Tempiangabe.

Zusätzlich sind Pausen mit der eingebauten Funktion

`pause(n)` für $n/100$ Sekunden möglich.

11 Das Testsystem

Für das Testen von PROLOG-Programmen bestehen folgende Möglichkeiten:

11.1 Die eingebauten Funktionen

`trace` schaltet den Trace ein.
Jede ausgeführte PROLOG-Regel wird angezeigt. Dabei werden folgende Möglichkeiten unterschieden:

`call` Aufruf der Regel

`exit` erfolgreiche Ausführung

`fail` nicht erfolgreiche Ausführung

`redo` backtrack auf die Regel

`notrace` schaltet den Trace aus.

`spy(a)` setzt einen Spion auf alle Regeln, deren Kopf mit Atom `a` beginnt.
Spione bewirken einen selektiven Trace nur der Spypunkte.

`nospy(a)` entfernt den Spion `spy(a)`.

`debugging` zeigt alle gesetzten Spione an.

`nodebug` entfernt alle Spione.

Spione zeigen nicht nur die Ausführung der betreffenden Regeln an, sondern bringen PROLOG beim ersten Durchlaufen des Spypunktes in den Stopzustand.

11.2 Der Stopzustand ist die zweite Testmöglichkeit

Nicht nur Spione können PROLOG in den Stopzustand bringen, sondern auch die Betätigung der Taste RUN/STOP .
Im Stopzustand gibt PROLOG ein Fragezeichen aus. Sie können jetzt durch Eingabe von Buchstabencodes bestimmen, wie es weitergeht.

`a` abort, Abbruch des Programms.

`c` continue, weiter wie gehabt.

`l` leap, weiter bis zum nächsten Spion.

`n` notrace, Trace ausschalten.

`s` single step, einen Schritt weiter.

`t` trace, Trace einschalten.

Die RUN/STOP-Taste ist insbesondere dann nützlich, wenn Sie die Befürchtung haben, daß Ihr Programm in einer Endlosschleife steckt.

Sie drücken dann RUN/STOP und geben entweder t für Trace oder s für single step ein, um zu sehen, was Ihr Programm macht.

Dies ist auch im Grafik-Betrieb möglich. PROLOG wechselt in diesem Fall ständig zwischen Grafik- und Textbildschirm.

12 Zurückschalten nach BASIC

Von PROLOG 64 aus können Sie nach BASIC zurückschalten: Entweder geben Sie '`eof`' ein oder das EOF-Zeichen ▨ (Shift + £).

BASIC meldet sich dann mit dem gewohnten Bildschirm. Jetzt können Sie nach Belieben BASIC-Programme ausführen, z.B. auch das EPSON-Software-Interface nachträglich laden.

Wollen Sie zurück nach PROLOG, so drücken Sie RUN/STOP +RESTORE .

Sie werden sehen, daß PROLOG alle Benutzerdaten behalten hat. Der Grafikbildschirm wird allerdings zurückgesetzt.

An dieser Stelle soll auf die Speicheraufteilung von PROLOG 64 eingegangen werden.

`$0000-$07ff` Systembereich

`$0800-$5bff` PROLOG-Stack

`$5c00-$91ff` Stack Erweiterung (nicht bei Grafik)

`$5c00-$5fff` Grafik Farb-RAM

`$6000-$8015` Grafik RAM

`$9200-$95ff` RAM für Garbage Collection

`$9600-$99ff` Eingabepuffer

`$9a00-$9a8f` Klangregister

`$9a90-$9fff` Platz für EPSON-Interface, bzw. für RS232 Puffer.

`$a000-$cfff` PROLOG-Interpreter (unter BASIC)

`$e000-$fff6` PROLOG-Datenspeicher (unter Betriebssystem).

Normalerweise braucht Sie diese Speicheraufteilung nicht zu kümmern. Sie soll jedoch Interessierten Einblick in die Arbeitsweise von PROLOG 64 bieten.

Sie können PROLOG auch aus einem BASIC-Programm heraus aufrufen:

```
SYS 49152
```

Mollen Sie PROLOG von einem eigenen Programm aus laden, so nehmen Sie sich den Lader als Beispiel und achten Sie darauf, daß alle Pokes gemacht werden.

13 Allgemeine Daten

- Integerzahlenbereich 0 – 8191
- Maximale Länge von Namen für Atome und Variablen: 123
- Bis zu 112 verschiedene Variablennamen möglich.
- Bis zu 15 verschiedene Variablen innerhalb einer Regel
- Prädikate mit l bis 15 Argumenten.
- PROLOG-Datenspeicher: 8 kB
- PROLOG-Stack bis 34 kB
- Maximale Eingabelänge einer Struktur: 1000 Zeichen
- Maximale Länge einer eingegebenen Liste: 63 Elemente.
- Gleichzeitig geöffnete Floppydateien: maximal 3.

14 Fehlermeldungen

Sie erkennen Fehlermeldungen an der Farbe 'purpur'.

1. **abort**
   ist keine echte Fehlermeldung, sondern ein gewollter Abbruch mit RUN/STOP -Taste und a .
2. **io-error**
   Fehler beim Zugriff auf externe Geräte.
   Hauptursachen:
   - Gerät ist ausgeschaltet.
   - Falsche oder keine Floppy eingelegt,
   - Dateiname falsch
   - Datei ist unvollständig
3. **out of stack error**
   Ihr Programm ist zu kompliziert (zu viele Rekursionen). Da in PROLOG Schleifen (ausser '`repeat`') durch Rekursionen dargestellt werden, besteht auch die Möglichkeit, daß der Fehler durch eine zu hohe Zahl von Schleifendurchläufen verursacht wird.
4. **out of heap error**
   Zu viele Regeln und Fakten wurden dem PROLOG-Datenspeicher zugefügt. Löschen Sie überflüssige Regeln und Fakten mit '`retract`' oder '`retractall`' oder löschen Sie alles mit '`clear`'.
5. **too many var error**
   Zu viele verschieden Variablennamen wurden verwendet. Verwenden Sie bereits benutzte Variablennamen oder beginnen Sie eine neue Sitzung '`clear`'.
   Die zweite Ursache ist die Verwendung von mehr als 15 verschiedenen Variablen innerhalb der selben Regel.
6. **division error**
   Es wurde eine Division durch 0 versucht.
7. **primitive error**
   Die eingebauten Funktionen wurden falsch verwendet.
   - Die Zahl der angegebenen Parameter ist falsch.
   - Ein arithmetischer Ausdruck ist falsch (falsche Operateuren, Argument ist keine Zahl).

Die folgenden Fehlermeldungen kommen vom Compiler.

8. **atom error**
   Syntaxfehler beim Übersetzen eines Atoms. Häufig ist ein fehlendes Leerzeichen die Ursache.
9. **structure error**
   Syntaxfehler beim Übersetzen einer Struktur. Häufig ist die falsche Verwendung von Kommas oder Klammern die Ursache.
10. **list error**
    Syntaxfehler beim übersetzen einer Liste. Häufig ist die falsche Verwendung von Kommas, Punkten oder Indexklammern die Ursache.

Tritt ein Fehler während eines `read`, `consult` oder `reconsult` auf, so wird der bis zum Auftreten des Fehlers abgearbeitete Ausdruck ausgegeben. So ist eine leichte Fehlerortung möglich.

Falls ein Fehler innerhalb einer Dateiverabeitung auftritt, so schliesst PROLOG 64 die gerade in Verarbeitung befindliche Datei und schaltet auf '`user`' zurück. Sind weitere Dateien offen, müssen Sie dafür Sorge tragen, daß sie geschlossen werden.

Sind in der Datenbank Regeln vom Typ

'`error :- .... `'

enthalten, so werden diese ausgeführt.

15 Expertensysteme

Expertensystem sind sehr in Mode gekommen, besonders seit einige Expertensysteme spektakuläre Erfolge verzeichnen konnten.

Expertensysteme gibt es heute auf verschiedenen Gebieten, in der Medizin, der Geologie, im technischen Bereich, usw.

Es ist damit zu rechnen, daß in einigen Jahren Expertensysteme im kommerziellen und auch im häuslichen Bereich einen ähnlichen Durchbruch schaffen werden, wie heute PC und Homecomputer.

Im Zusammenhang mit Expertensystemen sind zwei weitere Begriffe aufgetaucht, die Shell und die Wissensbasis (Knowledge base).

Vereinfacht kann man sagen, daß beide zusammen ein Expertensystem bilden.

Unter Shell versteht man ein allgemeines Programm, das in der Lage ist, das in der Wissensbasis abgelegte Wissen zu verarbeiten. Das konkrete Wissen über das jeweilige Fachgebiet befindet sich in der Wissensbasis in Form von Regeln und Fakten. Im Prinzip besteht zwischen Shell und Wissensbasis ein ähnliches Verhältnis, wie zwischen PROLOG-System und PROLOG-Programm.

Deshalb läßt sich eine Shell auch in PROLOG leicht formulieren. Die eigentliche Schwierigkeit beim Bau von Expertensystemen besteht denn auch im Aufbau der Wissensbasis, der sogenannten Wissensakquisition. Erst muß ja das Wissen zusammengetragen werden, bevor es in die Wissensbasis abgelegt werden kann.

Wie eine Shell funktioniert, möchten wir Ihnen an dem Programm MINISHELL demonstrieren. MINISHELL ist eine recht einfache Shell. Sie besteht aus folgenden Einheiten:

1. Eine Hauptschleife.
   Diese organisiert den Dialogablauf.
2. Eine Inferenzmaschine (Schlussfolgerungsmaschine). Diese dient dazu, die Regeln der Wissensbasis anzuwenden. Im einzelnen kann diese Inferenzmaschine folgende logische Operatoren behandeln:

nicht	Verneinung von Aussagen.
und	Verknüpfung von Aussagen.
weil	Begründung von Aussagen.

   Die Regeln in der Wissenbasis können z.B. so aussehen:

```
Medikament hilft Patient
weil Medikament gegen Ursache
```

```
and Patient klagt_ueber Symptom
und Symptom ist_Symptom_von Ursache
und nicht Medikament ist_unpassend_fuer Patient.
```

3. Eine weitere Komponente der Shell ist für den Dialog mit dem Benutzer verantwortlich.
   So sind die Fakten zum Prädikat '`klagt_ueber`' nicht in der Wissensbasis abgelegt, sondern müssen vom Benutzer eingegeben werden. Zu diesem Zweck enthält die Wissenbasis das Faktum

   ```
   klagt_ueber ist_erfragbar.
   ```

   Die Inferenzmaschine erkennt das und leitet die Beantwortung von '`klagt_ueber`' der Dialogkomponente zu. Diese gibt die Frage aus und liest die Antwort ein. Die Antwort kann entweder '`ja`' oder '`nein`' sein, oder es kann auch ein Wert sein. Lautet die Frage beispielsweise:

   ```
   hans klagt_ueber Symptom ?
   ```

   so ist eine mögliche Antwort:

   ```
   fieber.
   ```

   oder auch:

   ```
   hans klagt_ueber fieber.
   ```

   Sie werden beim Studium des Shell-Programms bemerken, daß die Antworten auf die Fragen in der Datenbasis gespeichert werden. Das geschieht, damit nicht etwa mehrmals die gleiche Frage gestellt wird.

   Hat die Inferenzmaschine eine Lösung auf die ursprüngliche Frage gefunden, so wird sie ausgegeben, und der Benutzer wird gefragt, ob er damit zufrieden ist.

   Wenn nein, wird versucht eine andere Lösung zu finden.

4. Die Erklärungskomponente:

   Typisch an Expertensystemen ist, daß sie nicht nur Ergebnisse liefern, sondern auf Befragen erklären, wie sie zu der Lösung gekommen sind, bzw. warum eine Frage an den Benutzer gestellt wird.

   Auch MINISHELL besitzt eine solche Erklärungskomponente. Bei jeder Frage, die die Shell an die Benutzer stellt, kann dieser '`warum.`' eingeben. Die Shell antwortet dann mit einer Auflistung der erfolgreich durchlaufenen Regeln, bzw. mit der Aufzählung der für die Beantwortung der Benutzerfrage in Betracht gezogenen Regeln und Fakten.

   Um dieses zu erreichen, speichert sich die Shell bei den Schlußfolgerungen die angewandten Regeln und Fakten in zwei Listen.

   Die eine Liste, mit `T` bezeichnet, dient zur Erläuterung der Fragen der Shell an den Benutzer. Die andere, mit `S` bezeichnete, Liste erläutert das Ergebnis.

Wir sind am Ende unserer kleinen Einführung in Expertensysteme. In großen Expertensystemen angewandte Shells sind wesentlich komplexer. Ein wichtiger Teil dieser Systeme besteht darin, jeweils eine optimale Strategie auszuwählen, also welche Regeln zuerst angewendet werden sollen. Auch der Benutzerdialog ist wesentlich verfeinert. So gibt es Systeme, die mit natürlicher Sprache arbeiten und komplexe oder ungenaue Antworten zulassen. Auch werden Graphiken verwendet. Oder es wird erklärt, warum etwas nicht gilt.

Unser Ziel war es, Sie in die Technik der Expertensysteme einzuführen, und Sie anzuregen, vielleicht selbst ein kleines Expertensystem aufzubauen, oder auch die MINISHELL mit mehr Komfort zu versehen.

Ein praktisches Beispiel für die Anwendung der Minishell finden Sie in der Bibliothek '`clean`'. Diese enthält ein kleines Expertensystem für die Entfernung von Flecken auf verschiedenen Materialien.

16 Anhang

16.1 Der Lader

Im Lader können folgende Einstellungen vorgenommen werden:

`s$ (="")`: Startup-Frage.
Ist `s$` definiert, so wird der Inhalt von `s$` sofort nach dem Laden von PROLOG als Eingabe betrachtet. Beispiel:

```
s$ = "consult(expert),topexpert"
```

(kein Endpunkt !)
In diesem Falle bekommt man das PROLOG-System gar nicht mehr zu Gesicht — die Verarbeitung beginnt sofort mit dem in PROLOG geschriebenen Expertensystem.

Wollen Sie außerdem verhindern, daß Ihr System mit der RUN/STOP-Taste unterbrochen werden kann, so fügen Sie im Lader die Anweisung:
`poke 788,52`
ein.

`SC (=11)`: Rahmenfarbe

`BC (=0)`: Hintergrundfarbe

`CG (=5)`: Cursorfarbe

`EC (=4)`: Fehlerfarbe

`TC (=13)`: Tracefarbe

`RC (=138)`: Kontrollwort RS 232
Standardeinstellung 2400 Baud, 7 Bit ASCII, 2 Stopbits.

`RI (=224)`: Befehlswort RS 232
Standardeinstellung: keine Paritätsprüfung, 8. Datenbit=0, Vollduplex, 3-Draht-Handshake

5 DATA-Zeilen für die Klangregister 0 – 4.
Jede DATA-Zeile entspricht einer SID-Einstellung, wie es in Ihrem COMMODORE-Handbuch angegeben ist.
Das Register l jeder Stimme kann eine Zahl enthalten, die auf die Tonnummer addiert werden soll. Damit können Akkorde gespielt werden.
Das Register 0 jeder Stimme kann eine Zahl enthalten, die auf die ausge-

rechnete Frequenz addiert wird. Damit können Schwebungen bzw. breitere Klänge erzeugt werden.

16.2 Abweichungen vom Edinburgh-Standard

Integers nur bis 8191

Länge von Namen für Atome und Variable max. 123.

Max. 113 verschiedene Variablennamen in einer Session.

Max. 15 verschiedene Variablen innerhalb eines Ausdrucks.

In '`spy`' und '`nospy`' kann nur der Name eines Prädikats angegeben werden, nicht die Zahl der Argumente. Auch ist die Angabe von Listen nicht möglich.

Beim Tracen wird im Gegensatz zu C&M auch der wiederholte '`match`' auf eine Regel als '`redo`' angezeigt und nicht nur der '`redo`' nach einem fail. Längere '`redo`'-Wiederholung o.g. Art sind meist ein Anzeichen dafür, daß eine andere Organisation des Programms zu einer besseren Laufzeit führt.

Die Schreibweise `[ lib1,lib2,lib3 ]` anstelle mehrerer '`consult`'s ist nicht zulässig.

Operatorprioritäten nur bis 63.
Operatortypen nur fx, xf, xfy, yfx.

Konjunktionen werden intern als Listen behandelt.

Deshalb wird statt	`call((a,b,c))`	`call([a,b,c])`
und statt	`not((a,b,c))`	`not([a,b,c])`

geschrieben. Andererseits ist möglich:

```
append([a],[b],X),assertz(g :- X).
```

Das Zeichen 'backslash' (\) wird durch 'slash' (/) ersetzt. Das betrifft die Operatoren `/=` und `/==` .

Was in PROLOG 64 über den von C&M definierten Standard hinaus zusätzlich möglich ist, finden Sie im Kapitel 'Erweiterungen'.

16.3 Beispielsitzung

`LOAD "*",8` RETURN
SEARCHING PROLOG
LOADING
`RUN` RETURN
PROLOG 64 Titelbild erscheint. Das System wird geladen. Copyright-Zeile erscheint.
`?-likes(X,Y).` RETURN
no

```
?-consult(user). RETURN
likes(john,alfred). RETURN
likes(alfred,john). RETURN
likes(bertrand,john). RETURN
likes(david,bertrand). RETURN
likes(john,X) :- RETURN
    likes(X,bertrand). RETURN
▨ RETURN
```
yes
```
?-likes(john,bertrand). RETURN
```
no
```
?-likes(john,X). RETURN
  likes(john,alfred) ;
  likes(john,david) ;
```
no
```
?-likes(X,Y). RETURN
  likes(john,alfred) ;
  likes(alfred,john) ;
  likes(bertrand,john) ;
  likes(david,bertrand) ;
  likes(john,david) ;
```
no
```
?-likes(john,david). RETURN
```
yes.
```
?-clear. RETURN
```
Es erscheint die Copyright Zeile.
```
?-consult(graphics). RETURN
```
yes.
```
?-move(45,150),skip(13). RETURN
```
Es erscheint eine Diagonallinie.

RETURN
```
?-circle(50),skip(13). RETURN
```
Es erscheint ein Kreis um den Endpunkt der Linie.

RETURN
```
?-screen(12,8). RETURN
```
yes
```
?-repeat,draw(160,100),square(100),move(l,0),fail.RETURN
```
Es erscheint ein rotierendes Quadrat.

RUN/STOP

Es erscheint eine Tracezeile gefolgt von `?` .

a

abort
```
?-▨ RETURN
```
ready.

Der Rechner ist in BASIC.

RUN/STOP +RESTORE
Es erscheint die Copyrightzeile.
`?-user.` RETURN
ready.
`load "graphics",8` RETURN
searching graphics
loading
`list` RETURN
Die Bibliothek graphics wird gelistet.
RUN/STOP +RESTORE
Es erscheint die Copyright-Zeile.
.
.
.

16.4 Inhalt der Diskette PROLOG 64

`prolog`	Lader PROLOG-System
`demoboot`	spezieller Lader für Demo-Programm
`s to p`	Umsetzer sequentielle Datei in Programmdatei
`p`	PROLOG-Interpreter
`sprites`	Spritedefinitionen für Demo-Programm
`clean`	Expertensystem für Fleckentfernung (Beispielanwendung von 'minishell')
`corrector`	Korrigierhilfe für PROLOG-Programme
`demo`	Bibliothek für Demo-Programm
`editor`	syntaktischer Editor für PROLOG
`files`	Beispiel für Dateiverarbeitung
`gener`	Bibliothek mit `gensys` und `findall`
`grammar`	Bibliothek mit Grammatikregeln
`graphics`	Bibliothek mit graphischen Routinen
`inout`	Bibliothek mit komfortablen Ein- und Ausgaberoutinen
`libs`	Hilfe für Bibliothekbenutzung
`logic`	Bibliothek mit Übersetzer von Prädikatenlogik in Klauselform
`logiplan`	Netzplatechnik
`mapping`	Bibliothek mit `maplist`, `applist` und `sort`
`math`	Bibliothek mit verschiedenen mathematischen Programmen
`minishell`	Shell für Expertensysteme
`music`	Bibliothek mit Musikroutinen
`project`	Beispiel für Netzplantechnik
`search`	Bibliothek mit verschiedenen Suchverfahren
`set`	Bibliothek mit Mengenoperationen
`tcogrammar`	Bibliothek mit Precompiler für Grammatikregeln

17 Au istung der mitgelieferten Bibliotheken

17.1 Verzeichnis der Diskette

```
0 ■"PROLOG 64 V1.2  " V2 2A
9     "PROLOG"            PRG
8     "DEMOBOOT"          PRG
3     "S TO P"            PRG
49    "P"                 PRG
2     "SPRITES"           SEQ
15    "CLEAN"             PRG
19    "CORRECTOR"         PRG
19    "DEMO"              PRG
25    "EDITOR"            PRG
20    "FILES"             PRG
3     "GENER"             PRG
5     "GRAMMAR"           PRG
10    "GRAPHICS"          PRG
7     "INOUT"             PRG
8     "LIBS"              PRG
17    "LOGIC"             PRG
23    "LOGIPLAN"          PRG
10    "MAPPING"           PRG
16    "MATH"              PRG
20    "MINISHELL"         PRG
8     "MUSIC"             PRG
9     "PROJECT"           PRG
13    "SEARCH"            PRG
6     "SET"               PRG
7     "TCOGRAMMAR"        PRG
7     "ASM.EXAMPLE"       PRG
18    "ASM.DOCU"          PRG
3     "LIESMICH"          PRG
300 BLOCKS FREE.
```

17.2 Prolog-Beispiele

17.2.1 clean

```
1   /*-----------------------------------
2      clean
3   -----------------------------------*/
5 ?-nl,write('Haben Sie einen Fleck ?'),
6 nl,write('Geben Sie ein: Was hilft.'),
7 nl,write('(vorher  run.  eingeben)').
90 /*-------------------------------*/
100 ?-reconsult minishell.
190 /*-------------------------------*/
200 ?-op(22,yfx,entfernt).
210 ?-op(22,xf,hilft).
220 ?-op(22,yfx,ist_befleckt_mit).
230 ?-op(22,xf,ist_farbschaedlich).
235 ?-op(22,xf,ist_eingefaerbt).
240 ?-op(22,yfx,besteht_aus).
250 ?-op(22,yfx,nicht_bei).
260 ?-op(22,yfx,ist_ungeeignet_fuer).
290 /*-------------------------------*/
300 chlorbleiche entfernt blumensaft.
```

```
301 chlorbleiche entfernt gras.
302 chlorbleiche entfernt obst.
303 chlorbleiche entfernt pflanzenfarben.
310 kochendes_wasser entfernt blumensaft.
311 warmes_seifenwasser entfernt fett.
312 warmes_seifenwasser entfernt oel.
313 warmes_seifenwasser entfernt fleischbruehe.
314 warmes_seifenwasser entfernt schmutz.
315 warmes_seifenwasser entfernt schweiss.
316 warmes_seifenwasser entfernt schuhcreme.
317 warmes_seifenwasser entfernt gras.
318 warmes_seifenwasser entfernt milch.
320 warmes_wasser entfernt kaffee.
321 warmes_wasser entfernt kakao.
322 warmes_wasser entfernt wein.
323 warmes_wasser entfernt essig.
324 warmes_wasser entfernt zitronensaft.
325 wasser entfernt bier.
330 verduennter_salmiakgeist entfernt
331   blut.
332 verduennter_salmiakgeist entfernt
333   fett.
334 verduennter_salmiakgeist entfernt
335   oel.
336 verduennter_salmiakgeist entfernt
337   harn.
338 verduennter_salmiakgeist entfernt
339   milch.
340 verduennter_salmiakgeist entfernt
341   wein.
342 verduennter_salmiakgeist entfernt
343   essig.
344 verduennter_salmiakgeist entfernt
345   zitronensaft.
346 verduennter_salmiakgeist entfernt
347   schmutz.
348 verduennter_salmiakgeist entfernt
349   schweiss.
350 verduennter_salmiakgeist entfernt
351   schuhcreme.
360 benzin entfernt fleischbruehe.
361 benzin entfernt milch.
362 benzin entfernt oelfarbe.
363 benzin entfernt firnis.
364 benzin entfernt harz.
365 benzin entfernt schmutz.
366 weinsaeure entfernt rost.
367 weinsaeure entfernt tinte.
368 oxalsaeure entfernt rost.
369 oxalsaeure entfernt tinte.
370 zitronensaft entfernt rost.
371 zitronensaft entfernt tinte.
380 terpentin entfernt oelfarbe.
381 terpentin entfernt firnis.
382 terpentin entfernt harz.
390 spiritus entfernt kopierstift.
391 spiritus entfernt kugelschreiber.
400 kochendes_wasser ist_farbschaedlich.
401 chlorbleiche ist_farbschaedlich.
402 zitronensaft ist_farbschaedlich.
403 oxalsaeure ist_farbschaedlich.
410 kochendes_wasser nicht_bei seide.
411 kochendes_wasser nicht_bei kunstfaser.
412 chlorbleiche nicht_bei seide.
413 chlorbleiche nicht_bei kunstfaser.
420 verduennter_salmiakgeist nicht_bei
421   kunstfaser.
422 weinsaeure nicht_bei kunstfaser.
490 /*--------------------------------*/
500 Fleckentferner hilft
502    weil untergrund ist_befleckt_mit
503                      Substanz
504    und Fleckentferner entfernt
505                       Substanz
506    und nicht Fleckentferner
507     ist_ungeeignet_fuer untergrund.
510 Fleckentferner
511     ist_ungeeignet_fuer untergrund
512    weil Fleckentferner
513         ist_farbschaedlich
514    und untergrund ist_eingefaerbt.
520 Fleckentferner
521     ist_ungeeignet_fuer untergrund
522    weil Fleckentferner
523                  nicht_bei Faser
524    und untergrund besteht_aus Faser.
790 /*--------------------------------*/
```

```
800 ist_eingefaerbt ist_erfragbar.
810 besteht_aus ist_erfragbar.
820 ist_befleckt_mit ist_erfragbar.
990 /*-------------------------------*/
```

17.2.2 corrector

```
1 /*-----------------------------------
2     Corrector
3  -----------------------------------
4  Mit Corrector koennen PROLOG-
5  Programm auf Schreibfehler geprueft
6  werden. Aufruf:
7 corr(P,N).
8 P = Name des zu pruefenden Praedikats
9 N = Anzahl seiner Argumente.
10     z.B. corr(append,3).
12 -----------------------------------
13 Corrector prueft, ob alle Praedikate
14 die auf der rechten Seite einer
15 Regel stehen, definiert oder
16 Primitive sind. Im letzteren Falle
17 wird zusaetzlich geprueft, ob die
18 Argumente des Primitivs auch richtig
19 sind, oder ob ein 'primitive error'
20 zu erwarten ist.
21 -----------------------------------
22 Corrector prueft nicht nur das ange-
23 gebene Praedikat, sondern auch alle
24 Praedikate, die von diesem mittel-
25 bar oder unmittelbar benutzt werden.
26 -----------------------------------*/
50 corr(P,D) :- corr(P,D,[]).
90 corr(P,N,D) :- member(c(P,N),D),!.
100 corr(P,N,D) :-
101   functor(S,P,N),
102   clause(S,G),
103   corrs(P,G,[c(P,N)|D]),
104   fail.
110 corr(_,_,_).
120 corrs(P,[G|L],D) :-
121   corrs(P,G,D),
122   corrs(P,L,D),!.
130 corrs(P,G,_) :-
131   functor(G,F,_),
132   corrp(F,T),
133   corrc(G,T),!.
140 corrs(P,G,D) :-
141   clause(G,S),!,
142   functor(G,F,N),
143   corri(G,D).
150 corrs(P,G,_) :-
151   write('Praedikat '),put("␣"),
152   write(G),nl,
153   write(' in Prozedur '),put("␣"),
154   write(P),nl,
155   write(' ist weder definiert noch'),nl,
156   write(' ein gueltiges Primitive.   '),
157   put("␣"),write('RETURN'),nl,
158   skip(13),nl.
160 corrc(G,T) :- !,G =.. [_|L],
161   corra(L,T).
170 corra([],[]) :- !.
180 corra([A|A1],[C|C1]) :-
181   corri(C,A),!,
182   corra(A1,C1).
185 corri(C,A) :- (var(A);var(C)),!.
186 corri(C,A) :- integer(C),!,
187   integer(A),A =< C.
188 corri(C,A) :- G =.. [C,A],call(G),!.
190 ariexp([X]) :- ariexpr(X),!.
192 ariexp(X) :- ariexpr(X).
194 ariexpr(X) :- var(X),!.
196 ariexpr(X) :- integer(X),!.
200 ariexp(X) :- X =.. [F,Y,Z],
205   ariop(F),ariexpr(Y),ariexpr(Z).
210 ariop(+).
215 ariop(-).
220 ariop(*).
225 ariop(/).
230 ariop(mod).
250 nolist(X) :- not islist(X).
```

```
infile(X) :- atom(X).
infile(X) :- X =.. [F,G],
  atom(F),integer(G).
outfile(user).
outfile(X) :- X =.. [F,G],
  atom(F),integer(G).
outfile(X) :- X =.. [F,G,append],
  atom(F),integer(G).
class(xf).
class(fx).
class(xfy).
class(yfx).
corr1([A|L],D) :-
  !,functor(A,F,N),
  corr(F,N,D),!,
  corr1(L,[c(F,N)|D]),!.
corr1(A,D) :-
  functor(A,F,N),
  corr(F,N,D).
corrp(true,[]).
corrp(fail,[]).
corrp(repeat,[]).
corrp(!,[]).
corrp(repeat,[]).
corrp(nl,[]).
corrp(seen,[]).
corrp(told,[]).
corrp(trace,[]).
corrp(notrace,[]).
corrp(debugging,[]).
corrp(nodebug,[]).
corrp(dir,[]).
corrp(clear,[]).
corrp(user,[]).
corrp(eof,[]).
corrp(error,[]).
corrp(np,[]).
corrp(var,[_]).
corrp(nonvar,[_]).
corrp(atom,[_]).
corrp(atomic,[_]).
corrp(integer,[_]).
corrp(islist,[_]).
corrp(call,[_]).
corrp(not,[_]).
corrp(read,[_]).
corrp(write,[_]).
corrp(display,[_]).
corrp(listing,[atom]).
corrp(asserta,[nolist]).
corrp(assertz,[nolist]).
corrp(retract,[nolist]).
corrp(get0,[255]).
corrp(get,[255]).
corrp(skip,[ariexp]).
corrp(put,[ariexp]).
corrp(tab,[ariexp]).
corrp(see,[infile]).
corrp(seeing,[infile]).
corrp(consult,[infile]).
corrp(reconsult,[infile]).
corrp(tell,[outfile]).
corrp(telling,[outfile]).
corrp(spy,[atom]).
corrp(nospy,[atom]).
corrp(seed,[ariexp]).
corrp(pause,[ariexp]).
corrp(pen,[ariexp]).
corrp(';',[_,_]).
corrp('.',[_,_]).
corrp(=,[_,_]).
corrp(==,[_,_]).
corrp(/=,[_,_]).
corrp(/==,[_,_]).
corrp(=..,[_,islist]).
corrp(>,[ariexp,ariexp]).
corrp(<,[ariexp,ariexp]).
corrp(=<,[ariexp,ariexp]).
corrp(>=,[ariexp,ariexp]).
corrp(=:=,[ariexp,ariexp]).
corrp(=/=,[ariexp,ariexp]).
corrp(name,[atom,islist]).
corrp(is,[integer,ariexp]).
corrp(random,[ariexp,integer]).
corrp(poke,[ariexp,ariexp]).
corrp(screen,[ariexp,ariexp]).
corrp(move,[ariexp,ariexp]).
```

```
1630 corrp(draw,[ariexp,ariexp]).
1640 corrp(plot,[ariexp,ariexp]).
1650 corrp(clause,[nolist,_]).
1660 corrp(last,[_,islist]).
1670 corrp(rev,[islist,islist]).
1680 corrp(sublist,[islist,islist]).
1690 corrp(member,[_,islist]).
1700 corrp(functor,[_,atom,15]).
1710 corrp(arg,[15,_,_]).
1720 corrp(op,[atom,class,63]).
1730 corrp(sound,[ariexp,ariexp,4]).
1740 corrp(append,[islist,islist,islist]).
1750 corrp(nextto,[_,_,islist]).
1760 corrp(efface,[_,islist,islist]).
1770 corrp(delete,[_,islist,islist]).
1800 corrp(subst,[_,islist,_,islist]).
9999 %%
```

17.2.3 editor

```
1 /*-------------------------------------
2   Editor
3 -------------------------------------
4 Dieser Editor kann anstelle von
5   consult  user
6 verwendet werden.
7 Aufruf:
8   edit(P,N). P = Praedikat
9              N = Anzahl Argumente
10  z.B. edit(hanoi,1).
11 -------------------------------------
12 Folgende Funktionen sind vorhanden:
13 -------------------------------------
14 Taste          Funktion
15 -------------------------------------
16 Cursor down    Eine Regel abwaerts
17 Cursor up      Eine Regel aufwaerts
18 Cursor right   naechstes Praedikat
19 Cursor left    letztes Praedikat,
20                bzw. Regelkopf
21 HOME           auf die erste Regel
22 DEL       auf  Regelkopf:
23                Regel wird geloescht
24           auf  Praedikat
25                Praedikat wird
26                          geloescht
27 INS       auf  Regelkopf:
28                Regel einfuegen
29           auf  Praedikat:
30                Praedikat einfuegen
31 Leertaste auf Regelkopf:
32                Regel aendern
33            auf Praedikat:
34                Praedikat aendern
35 RETURN    auf  Praedikat:
36                betreffendes Praedikat
37                editieren.
38           auf  Regelkopf:
39                Editierung des lfd.
40                Praedikats beenden.
41                Zurueck zum vorher
42                editierten Praedikat
43                oder Editor beenden.
50 -------------------------------------*/
100 edit(Proc,N) :-
110   asserta(error :- skip(13),np,edit,!),
116   collectclause(Proc,N),!,
130   np,edit,!.
200 edit :-
210   retract(proc(L,R,S)),
220   printproc(L,R,S),
230   get0(X),
235   process(X,L,R,S,L1,R1,S1),
245   asserta(proc(L1,R1,S1)),
250   L1=[],retract(proc(_,_,_)),
260   retract(error :- _ ),!.
490 /*-------------------------------*/
500 process(17,L,A,_,L,B,B) :- nextto(A,B,L),!,home.
540 /*-------------------------------*/
550 process(29,L,R,[A|S],L,R,S) :- !,home.
590 /*-------------------------------*/
600 process(145,[A,B|L],S,_,[A,B|L],A,A) :- B==S,!,home.
```

```
610 process(145,[A],[],_,[A],A,A) :- !,home.
620 process(145,[A|L],R,_,[A|L],S,S) :- !,process(145,L,R,_,L,S,S).
640 /*-------------------------------*/
650 process(157,L,R,S,L,R,S) :- S==R,!,home.
660 process(157,L,R,S,L,R,[X|S]) :-
670    append(_,[X|S],R),!,home.
690 /*-------------------------------*/
700 process(20,[eof,true],_,_,[eof,true],[],[]) :- !,np.
705 process(20,[Y|L],R,S,[Y|L1],Y,Y) :-
706   R==S,last(R,L),efface(R,L,L1),!,np.
710 process(20,L,R,S,L1,Y,Y) :- R==S,
720   nextto(R,Y,L),efface(R,L,L1),!,np.
730 process(20,L,[X,Y],[S],L1,[X,true],[X,true]) :- Y==S,
740   subst([X,Y],L,[X,true],L1),!,np.
750 process(20,L,R,[X|S],L1,R1,S) :-
760   append(Y,[X|S],R),
770   append(Y,S,R1),
780   subst(R,L,R1,L1),!,np.
790 /*-------------------------------*/
800 process(13,L,R,S,L1,R1,S1) :- R==S,
805   modify(L),(retract(proc(L1,R1,S1));L1=[]),!,np.
810 process(13,L,R,[Y|S],L1,R1,S1) :-
815   asserta(proc(L,R,[Y|S])),
817   functor(Y,F,N),
820   collectclause(F,N),
830   retract(proc(L1,R1,S1)),!,np.
835 /*-------------------------------*/
840 process(19,[R|L],_,_,[R|L],R,R) :- !,home.
845 /*-------------------------------*/
850 process(148,L,R,S,L1,X,X) :- R==S,
855   getclause(X,proc(L,R,R)),
860   append(A,[R|Y],L),
865   append(A,[X,R|Y],L1),!,np.
870 process(148,L,R,S,L1,R1,[X|S]) :-
875   getarg(' ',X,proc(L,R,S)),
880   append(A,S,R),append(A,[X|S],R1),
885   subst(R,L,R1,L1),!,np.
900 /*-------------------------------*/
920 process(32,L,R,[X|S],L1,R1,[X1|S]) :-
925   getarg(X,X1,proc(L,R,[X|S])),
930   append(A,[X|S],R),append(A,[X1|S],R1),
935   subst(R,L,R1,L1),!,np.
950 /*-------------------------------*/
999 process(_,L,R,S,L,R,S) :- home.
1000 /*-------------------------------*/
1100 printout([],[]) :- put("▮"),put("▣"),put("."),put("▧"),nl,!.
1105 printout([],_) :- put("▮"),put("."),nl,!.
1110 printout(Y,Y) :-
1120   Y=[A|R],highlight(A),
1130   put(","),printout(R,0),!.
1140 printout([A|R],L) :-
1150   write(A),put(","),
1160   printout(R,L),!.
1190 /*-------------------------------*/
1200 highlight(A) :-
1210   put("▣"),write(A),put("▧").
1290 /*-------------------------------*/
1300 printrule([Hd,true],[Hd,true],_) :-
1310   highlight(Hd),put("."),nl,!.
1320 printrule([Hd,true],_,_) :-
1330   write(Hd),put("."),nl,!.
1400 printrule(R,S,L) :- R==S,S==L,
1405   R = [Hd|Rhs],
1410   highlight(Hd),tab(1),
1420   write(:-),nl,tab(3),
1430   printout(Rhs,0),!.
1450 printrule(R,S,L) :-  R==S,
1455   R = [Hd|Rhs],
1460   write(Hd),tab(1),
1470   write(:-),nl,tab(3),
1480   printout(Rhs,L),!.
1500 printrule([Hd|Rhs],_,_) :-
1510   write(Hd),tab(1),
1520   write(:-),nl,tab(3),
1530   printout(Rhs,0).
1590 /*-------------------------------*/
1600 printproc([],_,_) :- !.
1630 printproc([A|L],R,S) :-
1640   printrule(A,R,S),
1650   printproc(L,R,S),!.
1660 printproc(_,_,_).
1690 /*-------------------------------*/
1700 collectclause(F,N) :-
1710   asserta(found(mark)),
1715   functor(Hd,F,N),
1720   clause(Hd,Rhs),norm(Rhs,R),
1730   asserta(found([Hd|R])),fail.
```

```
1740 collectclause(_,_) :-
1750   collect([[eof,true]],[R|L1]),
1760   asserta(proc([R|L1],R,R)),!.
1800 collect(S,L) :- getnext(X),!,
1810   collect([X|S],L).
1820   asserta(proc([C|L])),!.
1850 collect(L,L).
1855 getnext(X) :- retract(found(X)),!,
1856   X /== mark.
1860 norm([],[]) :- !.
1863 norm(B,[B]) :- not islist(B),!.
1865 norm([A1|A],[A1|B]) :- !,norm(A,B).
1890 /*-----------------------------*/
1900 getarg(0,X,S) :-
1910   nl,nl,write(0),put("."),put("▯"),
1920   readsave(X,S).
1990 /*-----------------------------*/
2000 getclause(X,S) :-
2010   nl,nl,
2020   readsave(Y,S),getcl1(Y,X).
2030 getcl1((Hd :- Rhs),[Hd|R]) :-
2040   norm(Rhs,R),!.
2050 getcl1(Hd,[Hd,true]).
2060 readsave(Y,S) :- asserta(S),
2070   read(Y),retract(S).
2090 /*-----------------------------*/
2200 modify(L) :- L = [[Hd|_]|_],
2201   functor(Hd,F,N),functor(H,F,N),
2202   retractall(H),assert(L),!.
2210 assert([[eof,true]]) :- !.
2220 assert([[Hd,true|L]) :-
2221   assertz(Hd),assert(L),!.
2230 assert([[Hd|R]|L]) :-
2231   assertz(Hd :- R), assert(L).
2300 home :- put("▯"),!.
```

17.2.4 files

```
1 /*-----------------------------------
2          files
3 -----------------------------------
4 Aufruf:  run
5 -----------------------------------
12 Files verwaltet Fakten ueber
13 Objekte und ist ein Beispiel fuer
14 die Dateiverarbeitung mit PROLOG 64.
15 Sie koennen neue Objektklassen
16 definieren, Objekte anlegen und
17 abfragen.
18 Die Daten werden auf der Floppy
19 sequentiell gespeichert.
20 -----------------------------------
21 Bei Definition und Abfrage haben
22 Sie folgende Moeglichkeiten:
23 a) Angabe einzelner Werte
24 b) Angabe mehrerer Werte mit
25    Oder-Verknuepfung ';' .
26 b) Angabe von Zahlenbereichen mit
27    '-' , z.B. 30-40 .
28 b) Angabe von 'Wert unbestimmt'
29    mit '?' .
30 -----------------------------------*/
100 run :- retractall(error),
110        asserta(error :- run),
120        repeat,np,
130        put("▯"),tab(17),
140        write('FILES'),
150        tab(18),nl,nl,
160        write('Welche Objekte wollen Sie bearbeiten?'),nl,
170        read(Name),
180        getframe(Name,Frame),
190        process(Name,Frame),fail.
195 /*
196   Objektklasse vorhanden ?
197 */
200 getframe(Name,Frame) :-
250        File =.. [Name,8],
280        Errex =
281          (error :- addframe(Name)),
290        asserta(Errex),
300        see(File),read(f(Frame)),
310        seen,retract(Errex),!.
```

```
395 /*
396   Neue Objektklasse definieren.
397 */
400 addframe(Name) :-
410         retract(error :- addframe(Name)),
415         np,put("■"),write('DEFINITION'),tab(30),n1,
420         n1,write('Definieren Sie die Eigenschaften von'),n1,
430         write(Name),n1,
440         write('(EOF-Zeichen fuer Ende)'),
450         n1,n1,defframe(Frame),
460         File =.. [Name,8],
470         tell(File),display(f(Frame)),
480         put("."),n1,told,!,
490         process(Name,Frame),run.
595 /*
596   Eingaben neue Objektklasse.
597 */
600 defframe(L) :-
610         read(A),defframe(A,L).
620 defframe(eof,[]) :- !.
630 defframe(B,[B|L]) :- read(A),
640         defframe(A,L).
795 /*
796   Vorhandene Objektklasse
797   verarbeiten.
798   (Neues Objekt/Abfrage Objekt)
799 */
800 process(Name,Frame) :- repeat,np,
810   write(' n - Neues Objekt '),n1,
820   write(' a - Abfrage'),n1,
830   write(' e - Ende'),n1,
840   prompt("nae",X),
850   process(Name,Frame,X),
860   X is "e",!.
900 process(Name,Frame,X) :- X is "e",!.
910 process(Name,Frame,X) :- repeat,np,title(X),
920   write('Definieren Sie die Eigenschaften als'),n1,
930   write('Atome oder Integerzahlen.'),n1,n1,
935   write('Beispiele fuer Ausdruecke'),n1,
940   write('Zahlenbereiche: 1-10'),n1,
950   write('Oder-Bedingung: a;b'),n1,
960   write('Unbestimmt:     ?'),n1,n1,
970   interview(Frame,Data),Data /= [],
980   !,access(Frame,Name,Data,X),!.
985 /*
986   Titelzeile ausgeben.
987 */
990 title(X) :- X is "a", put("■"),write('ABFRAGE'),tab(33),n1,!.
991 title(X) :- X is "n", put("■"),write('NEUEINTRAG'),tab(30),n1,!.
995 /*
996   Objektpraedikate anhand des
997   Frames festlegen.
998 */
1000 interview([],[]) :- !.
1010 interview([Item|Rest],[Answer|Data]) :-
1020   ask(Item,Answer),
1030   interview(Rest,Data).
1095 /*
1096   Eingabe eines Praedikats.
1098 */
1100 ask(Item,Answer) :-
1110   repeat,write(Item),write('?  '),
1120   read(Answer),check(Answer).
1195 /*
1196   Eingabe pruefen.
1198 */
1200 check(Answer) :- atomic(Answer),!.
1210 check(X-Y) :- integer(X),integer(Y),!.
1220 check(X;Y) :- check(X),check(Y).
1295 /*
1296   Funktionscode eingeben.
1298 */
1300 prompt(L,X) :- repeat,get0(X),cprompt(L,X),!.
1310 cprompt([X|L],X) :- !.
1320 cprompt([_|L],X) :- cprompt(L,X).
1395 /*
1396   Neues Objekt anspeichern.
1398 */
1400 access(_,Name,Data,X) :- X is "n",!,
1410   File =.. [Name,8,append],
1420   tell(File),display(Data),
1430   put("."),n1,told.
1445 /*
1446   Objekt in Datei suchen.
1448 */
1450 access(Frame,Name,Data,_) :-
1460   File =.. [Name,8],see(File),
```

```
1470    repeat,
1480    read(Object),
1490    compare(Data,Object),
1500    see(user),output(Frame,Object),
1510    see(File),Object = eof,!,seen.
1595 /*
1596    Objekt anzeigen.
1598 */
1600 output(_,eof) :-
1610    write('Keine (weiteren) Objekte'),output([],[]),!.
1620 output([],_) :- nl,write('RETURN'),nl,nl,skip(13),!.
1630 output([F|Frame],[O|Object]) :-
1640    write(F),write(':     '),
1650    write(O),nl,output(Frame,Object).
1695 /*
1696    Gelesenes Objekt mit Eingaben
1697    vergleichen.
1698 */
1700 compare(_,eof) :- !.
1710 compare([],_).
1720 compare([D|Data],[O|Object]) :-
1730    compel(D,O),compare(Data,Object).
1735 /*
1736    Einzelnes Praedikat vergleichen.
1738 */
1740 compel(X,X) :- !.
1750 compel(?,_) :- !.
1760 compel(_,?) :- !.
1770 compel((X;Y),O) :- compel(X,O) ;
1780    compel(Y,O) ).
1790 compel(D,(X;Y)) :- compel(D,X) ;
1800    compel(D,Y) ).
1810 compel(X-Y,A-B) :- !,
1820    A =< X, B >= Y.
1830 compel(D,A-B) :-
1840    A =< D, B >= D.
1999 %%
```

17.2.5 gener

```
1 /*  gensym
2     erzeugt durchnumerierte atome
3 */
10 gensym(Root,Atom) :-
20      getnum(Root,Num),
30      name(Root,Name1),
40      intname(Num,Name2),
50      append(Name1,Name2,Name),
60      name(Atom,Name).
99 /* laufende nummer holen */
100 getnum(Root,Num) :-
110     retract(currentnum(Root,Num1)),
120     !,Num is Num1+1,
130     asserta(currentnum(Root,Num)).
140 getnum(Root,1) :-
150     asserta(currentnum(Root,1)).
199 /* integer in digits zerlegen */
200 intname(Int,List) :-
201     intname(Int,[],List).
210 intname(I,Sofar,[C|Sofar]) :-
220     I < 10,!,
221     C is I+48 .
230 intname(I,Sofar,List) :-
240     Tophalf is I/10,
250     Bothalf is I mod 10,
260     C is Bothalf+48 ,
270     intname(Tophalf,[C|Sofar],List).
999 %%
```

17.2.6 grammar

```
1 /*  GRAMMATIKREGELN
2     fuer einfache englische Saetze
3 */
4 ?-consult tcogrammar /*Uebersetzer*/.
40 /* --------------------------------
44 Phrase pruefen, z.B.
```

```
45 phrase(sentence,[the,man,eats])
46 ---------------------------------- */
50 phrase(P,L) :-
51       Goal =.. [P,L,[]],call(Goal).
94 /* ----------------------------------
95 Einzelne Grammatikregeln
96 ---------------------------------- */
110 sentence --> sentence(X).
115 sentence(X) -->
120 nounphrase(X),verbphrase(X).
130 nounphrase(X) -->
131      determiner(X), noun(X).
140 nounphrase(plural) -->
141      noun(plural).
150 verbphrase(X) --> verb(X).
160 verbphrase(X) -->
161      verb(X),nounphrase(Y).
170 determiner(_) --> [the].
171 determiner(singular) --> [a].
175 noun(singular) --> [boy].
176 noun(singular) --> [man].
177 noun(singular) --> [peach].
180 noun(plural) --> [boys].
181 noun(plural) --> [men].
182 noun(plural) --> [peaches].
200 verb(singular) --> [eats].
201 verb(singular) --> [sells].
205 verb(plural) --> [eat].
206 verb(plural) --> [sell].
900 %%
```

17.2.7 graphics

```
95 /*---------------------------------
96             circle
97  circle(R) zeichnet einen Kreis mit
98  Radius R um die Turtle.
99 */
100 circle(R) :-
101     S is R * 35 / 112,
110     pen(P),pen(0),
111     move(0,R),move(81,0),
112     pen(P),sekante(20,S),
120     pen(0),move(99,R),
121     move(180,0),pen(P),!.
190 sekante(0,S) :- !.
200 sekante(N,S) :-
201     move(18,S),
202     N1 is N-1,sekante(N1,S).
285 /*---------------------------------
286             triangle
287  triangle(H) zeichnet einen Drei-
298  eck mit Hoehe H in Turtlerichtung.
299 */
300 triangle(H) :-
301     S is H * 37/32,
310     move(270,S/2),
320     move(120,S),move(120,S),
321     move(120,(S+1)/2),
330     move(90,0),!.
385 /*---------------------------------
386             histogram
387  histogram(b,L) zeichnet eine
388  Balkengrafik in Turtlerichtung.
389  Die Balkenhoehen werden in Liste
390  L angegeben, die Breite in b.
394 */
395 histogram(B,L) :- histogram(B,L,0).
400 histogram(B,[],X) :-
410      move(270,X),move(90,0),!.
500 histogram(B,[A|L],X) :-
510      position(X,A,B),
511      histogram(B,L,A).
600 position(X,A,B) :-
601      X>A,move(270,X-A),move(90,B),!.
610 position(X,A,B) :-
611      move(90,A-X),move(270,B),!.
685 /*---------------------------------
686              square
687  square(L) zeichnet ein Quadrat mit
688  Seitenlaenge L und Mittelpunkt auf
689  der Turtle.
```

```
690 */
700 square(R) :-
701     pen(P),
710     pen(0),move(180,R/2),pen(P),
720     move(90,R/2),
721     move(90,R),move(90,R),move(90,R),
722     move(90,(R+1)/2),
730     pen(0),move(90,R/2),pen(P),!.
785 /*------------------------------
786              man
787  man zeichnet einen Strichmann mit
788  Mittelpunkt des Kopfes in Turle-
789  position.
790 */
800 man :- kopf,arme,leib,beine,!.
810 kopf :-
811     circle(5),
812     pen(X),pen(0),move(0,5),
813     pen(X),move(0,3).
820 arme :-
821     move(30,15),move(180,15),
822     move(120,15),move(180,15),
823     move(210,0).
830 leib :- move(0,15).
840 beine :-
841     move(20,15),move(180,15),
842     move(140,15),move(180,15),
843     move(200,0).
845 /*------------------------------
846              people
847  people zeichnet einen Haufen Leute.
848 */
850 people :-
851     poke(12*256+4,Z),seed(Z),
852     screen(2,7),
855     move(270,0),pen(P),
856     repeat,
857     random(320,X),random(200,Y),
860     pen(0),draw(X,Y),pen(P),
861     man,
862     fail.
999 eof.
```

17.2.8 inout

```
1   /*
2   Inout enthaelt die Praedikate
3   readin und printout.
4   readin(X) liest einen satz ein und
5   wandelt ihn in Listenformat.
6   printout(X) druckt eine Liste im
7   Satzformat aus.
8   */
19 /*  Satz  */
20 readin([W|Ws]) :-
21     getc(C),readword(C,W,C1),
22     restsent(W,C1,Ws).
39 /*  Satzrest  */
40 restsent(W,_,[]) :- lastword(W),!.
50 restsent(W,C,[W1|Ws]) :-
51     readword(C,W1,C1),
52     restsent(W1,C1,Ws).
69 /*  Wort  */
70 readword(C,W,C1) :-
71     singlechar(C),!,
72     name(W,[C]),getc(C1).
80 readword(C,W,C2) :-
90     inword(C,NewC),!,
100    getc(C1),
110    restword(C1,Cs,C2),
120    name(W,[NewC|Cs]).
130 readword(C,W,C2) :- getc(C1),readword(C1,W,C2).
139 /*  Restwort  */
140 restword(C,[NewC|Cs],C2) :-
150     inword(C,NewC),!,
160     getc(C1),
170     restword(C1,Cs,C2).
180 restword(C,[],C).
199 /*  Interpunktionen  */
200 /* , */   singlechar(44).
210 /* ; */   singlechar(59).
220 /* : */   singlechar(58).
```

```
240 /* ? */   singlechar(63).
250 /* ! */   singlechar(33).
260 /* . */   singlechar(46).
279 /*  Wortzeichen  */
280 /*a...z*/ inword(C,C) :- C>64,C<91.
290 /*A...Z*/ inword(C,L) :- C>192,C<219,L is C-128.
300 /*1...9*/ inword(C,C) :- C>47,C<58.
310 /*  '  */ inword(39,39).
320 /*  -  */ inword(45,45).
339 /*  Satzendezeichen  */
340 lastword('.').
350 lastword('!').
360 lastword('?').
398 /*  Zeichen lesen und anzeigen  */
400 getc(C) :- get0(C),put(C).
497 /*
498           printout
499 */
500 printout([]) :- !,tab(1).
510 printout([C|C1]) :-
520    name(C,[A|L]),printspace(A),
525    printword([A|L]),printout(C1).
530 printspace(A) :- singlechar(A),!.
540 printspace(_) :- tab(1).
550 printword([]) :- !.
560 printword([A|L]) :-
561    put(A),printword(L).
999 %
```

17.2.9 libs

```
1 /*-----------------------------------
2              libs
3 -----------------------------------
4 libs bringt eine Uebersicht ueber die
5 mitgelieferten Bibliotheken. Per
6 Tastendruck kann eine Bibliothek
7 ausgewaehlt werden.
8 -----------------------------------*/
100 ?-np,put("■"),tab(7),
101   write('PROLOG 64   Bibliotheken'),
102   tab(9),nl,
110   write('Taste  Bibliothek  Kommentar'),
115   nl,write('----------------------------').
117 ?-nl,write(' 0     clean       Experte fuer Flecke').
118 lib(48,clean).
120 ?-nl,write(' 1     corrector   Fehlersuche').
125 lib(49,corrector).
130 ?-nl,write(' 2     editor      syntaktischer Editor').
135 lib(50,editor).
140 ?-nl,write(' 3     files       Karteikasten').
145 lib(51,files).
150 ?-nl,write(' 4     gener       gensym').
155 lib(52,gener).
160 ?-nl,write(' 5     grammar     Englische Grammatik').
165 lib(53,grammar).
170 ?-nl,write(' 6     graphics    Grafikbaukasten').
175 lib(54,graphics).
180 ?-nl,write(' 7     inout       Ein- und Ausgabe').
185 lib(55,inout).
190 ?-nl,write(' 8     logic       Praedikatenlogik').
195 lib(56,logic).
200 ?-nl,write(' 9     logiplan    Netzplanprogramm').
205 lib(57,logic).
210 ?-nl,write(' a     mapping     sort,maplist').
215 lib(65,mapping).
220 ?-nl,write(' b     math        hoehere Mathematik').
225 lib(66,math).
230 ?-nl,write(' c     minishell   Expertenshell').
235 lib(67,minishell).
236 ?-nl,write(' d     music       Musik nach Noten').
237 lib(68,music).
238 ?-nl,write(' e     project     Netzplan').
239 lib(69,project).
240 ?-nl,write(' f     search      Suchprogramme').
245 lib(70,search).
250 ?-nl,write(' g     set         Mengenlehre').
255 lib(71,set).
260 ?-nl,write(' h     tcogrammar  Grammatikregeln').
265 lib(72,tcogrammar).
290 ?-nl,write('----------------------------').
300 ?-repeat,seeing(L),see(user),
```

```
301   get(X),see(L),lib(X,Y),
302   retractall(lib(_,_)),consult(Y),
303   nl,write(Y),write(' consulted.').
```

17.2.10 logic

```
1 /*          translate
2 uebersetzt Ausdruecke der Praedi-
3 katenlogik in Klauselform.
4 Logische Operatoren:
9 */
10 ?-op(8,fx,¬)      /*  Negation    */.
20 ?-op(25,xfy,#)    /*  Oder        */.
30 ?-op(25,xfy,&)    /*  Und         */.
40 ?-op(37,xfy,->)   /*  Implikation */.
50 ?-op(37,xfy,<->) /*  Aequivalenz
60                       all
70                       exists      */.
88  ?-reconsult(gener).
100 translate(X) :-
110   implout(X,X1),
120   negin(X1,X2),
130   skolem(X2,X3,[]),
140   univout(X3,X4),
150   conjn(X4,X5),
160   clausify(X5,Clauses,[]),
170   pclauses(Clauses).
198 /* implikationen und aequivalenzen
199   aufloesen                        */
200 implout((P <-> Q),((P1 & Q1) # ( ¬P1 & ¬Q1))) :- !,
210   implout(P,P1),
220   implout(Q,Q1).
230 implout((P -> Q),( ¬P1 # Q1 )) :-
240   !,implout(P,P1),
250   implout(Q,Q1).
260 implout(all(X,P),all(X,P1)) :-
270   !,implout(P,P1).
280 implout(exists(X,P),exists(X,P1)) :-
281   !, implout(P,P1).
290 implout((P & Q),(P1 & Q1)) :-
300   !, implout(P,P1),implout(Q,Q1).
310 implout((P # Q),(P1 # Q1)) :-
320   !, implout(P,P1),implout(Q,Q1).
330 implout((¬P),(¬P1)) :-
331   !,implout(P,P1).
340 implout(P,P).
398 /* negation nach innen bringen   */
400 negin((¬P),P1) :- !,neg(P,P1).
410 negin(all(X,P),all(X,P1)) :-
411   !,negin(P,P1).
420 negin(exists(X,P),exists(X,P1)) :-
421   !, negin(P,P1).
430 negin((P & Q),(P1 & Q1)) :- !,
440 negin(P,P1),negin(Q,Q1).
450 negin((P # Q),(P1 # Q1)) :- !,
460 negin(P,P1),negin(Q,Q1).
470 negin(P,P).
500 neg((¬P),P1) :- !,negin(P,P1).
510 neg(all(X,P),all(X,P1)) :-
511   !,neg(P,P1).
520 neg(exists(X,P),exists(X,P1)) :-
521   !, neg(P,P1).
530 neg((P & Q),(P1 # Q1)) :-
540   !, neg(P,P1),neg(Q,Q1).
550 neg((P # Q),(P1 & Q1)) :-
560   !, neg(P,P1),neg(Q,Q1).
570 neg(P,(¬P)).
598 /* skolemisierung               */
600 skolem(all(X,P),all(X,P1),Vars) :-
601   !,skolem(P,P1,[X|Vars]).
610 skolem(exists(X,P),P2,Vars) :-
620   !, gensym(f,F), Sk =.. [F|Vars],
630   subst1(X,Sk,P,P1),
640   skolem(P1,P2,Vars).
650 skolem((P & Q),(P1 & Q1),Vars) :-
660   !, skolem(P,P1,Vars),
661   skolem(Q,Q1,Vars).
670 skolem((P # Q),(P1 # Q1),Vars) :-
680   !, skolem(P,P1,Vars),
681   skolem(Q,Q1,Vars).
690 skolem(P,P,_).
695 subst1(V1,V2,F1,F2) :-
```

```
696   F1 =.. [F|A1],
697   subst(V1,A1,V2,A2),
698   F2 =.. [F|A2].
699 /* all operatoren entfernen     */
700 univout(all(X,P),P1) :-
701   !,univout(P,P1).
710 univout((P & Q),(P1 & Q1)) :-
720   !,univout(P,P1),univout(Q,Q1).
730 univout((P # Q),(P1 # Q1)) :-
740   !,univout(P,P1),univout(Q,Q1).
750 univout(P,P).
798 /* konjunktive normalform bilden */
800 conjn((P # Q),R) :-
810   !,conjn(P,P1),conjn(Q,Q1),
820   conjn1((P1 # Q1),R).
830 conjn((P & Q),(P1 & Q1)) :-
840   !,conjn(P,P1),conjn(Q,Q1).
850 conjn(P,P).
900 conjn1(((P & Q) # R),(P1 & Q1)) :-
910   !,conjn1((P # Q),P1),
911   conjn1((Q # R),Q1).
920 conjn1((P # (Q & R)),(P1 & Q1)) :-
930   !,conjn1((P # Q),P1),
931   conjn1((P # R),Q1).
940 conjn1(P,P).
999 /* in klauselform bringen       */
1000 clausify((P & Q),C1,C2) :-
1010   !,clausify(P,C1,C3),
1011   clausify(Q,C3,C2).
1020 clausify(P,[c1(A,B)|Cs],Cs) :-
1030   inclause(P,A,[],B,[]),!.
1040 clausify(_,C,C).
1100 inclause((P # Q),A,A1,B,B1) :-
1110   !,inclause(P,A2,A1,B2,B1),
1120 inclause(Q,A,A2,B,B2).
1130 inclause((¬P),A,A,B1,B) :-
1140   !,not member(P,A),putin(P,B,B1).
1150 inclause(P,A1,A,B,B) :-
1160   not member(P,B),putin(P,A,A1).
1200 putin(X,[],[X]) :- !.
1210 putin(X,[X|L],L) :- !.
1220 putin(X,[Y|L],[Y|L1]) :-
1221   putin(X,L,L1).
1298 /* klauseln ausdrucken          */
1300 pclauses([]) :- !,n1,n1.
1310 pclauses([c1(A,B)|Cs]) :-
1311   pclause(A,B),n1,pclauses(Cs).
1320 pclause(L,[]) :-
1321   !,pdisj(L),write('.').
1330 pclause([],L) :-
1331   !,write(':- '),
1332   pconj(L),write('.').
1340 pclause(L1,L2) :-
1341   pdisj(L1), write(' :- '),
1342   pconj(L2),write('.').
1350 pdisj([L]) :- !,write(L).
1360 pdisj([L|Ls]) :-
1361   write(L),write('; '),pdisj(Ls).
1370 pconj([L]) :- !,write(L).
1380 pconj([L|Ls]) :-
1381   write(L),write(', '),pconj(Ls).
1999 %%
```

17.2.11 logiplan

```
1 /* ------------------------------------
2            logiplan
3 ------------------------------------
4  logiplan ist ein allgemeines System
5  fuer Netzplantechnik.
6  Sie koennen Anfragen stellen, z.B. :
7    Was ist_kritischer_Pfad.
8  oder :
9    Was ist_Aktivitaet,Was
10     beginnt_fruehestens um X.
11 ------------------------------------
12 Ein Beispiel fuer die Anwendung und
13 die Definition von Netzplaenen
14 finden Sie in Bibliothek 'project'.
15 Im folgenden finden Sie die Defini-
16 tion der einzelnen Anfragen.
17 ------------------------------------
```

```
18 Operatorprioritaeten
19 ---------------------------------- */
20 ?-op(21,yfx,dauert).
25 ?-op(21,yfx,vor).
30 ?-op(22,xf,ist_Aktivitaet).
35 ?-op(22,xf,ist_Startaktivitaet).
40 ?-op(22,xf,ist_Schlussaktivitaet).
45 ?-op(22,xf,beginnt_fruehestens_um).
50 ?-op(22,xf,beginnt_spaetestens_um).
55 ?-op(22,xf,endet_fruehestens_um).
60 ?-op(22,xf,endet_spaetestens_um).
65 ?-op(22,xf,dauert_das_Projekt).
70 ?-op(22,xf,ist_kritisch).
75 ?-op(22,xf,ist_kritischer_Pfadteil).
80 ?-op(22,xf,ist_kritischer_Pfad).
85 ?-op(22,yfx,ist_frei_verschieblich_um).
90 ?-op(22,yfx,ist_insgesamt_verschieblich_um).
95 /* ---------------------------------
96 Auflistung der Anfragen
97 ---------------------------------*/
100 X ist_Aktivitaet :-
101   X dauert Y.
105 /* ---------------------------- */
110 X ist_Startaktivitaet :-
111   X ist_Aktivitaet, not _ vor X.
115 /* ---------------------------- */
120 X ist_Schlussaktivitaet :-
121   X ist_Aktivitaet, not X vor _.
125 /* ---------------------------- */
130 X beginnt_fruehestens_um 0 :-
131   X ist_Startaktivitaet,nonvar(X),
134   asserta(X beginnt_fruehestens_um 0
135     :- !).
140 X beginnt_fruehestens_um Y :-
141   findall(X1,[Y1 vor X,
142     Y1 endet_fruehestens_um X1],Z),
143   max(Y,Z),nonvar(X),
144   asserta(X beginnt_fruehestens_um Y
145     :- !).
147 /* ---------------------------- */
150 X endet_fruehestens_um Y :-
151   X beginnt_fruehestens_um Z,
152   X dauert T, Y is Z+T,nonvar(X),
153   asserta(X endet_fruehestens_um Y
154     :- !).
155 /* ---------------------------- */
160 X dauert_das_Projekt :-
161   findall(Z,[X1 ist_Schlussaktivitaet,
162      X1 endet_fruehestens_um Z],Y),
163   max(X,Y),nonvar(X),
164   asserta(X dauert_das_Projekt :- !).
165 /* ---------------------------- */
170 X endet_spaetestens_um Y :-
171   X ist_Schlussaktivitaet,
172   Y dauert_das_Projekt,
173   asserta(X endet_spaetestens_um Y
174     :- !),!.
180 X endet_spaetestens_um Y :-
181   findall(X1,[X vor Y1,Y1
182     beginnt_spaetestens_um X1],Z),
183   min(Y,Z),nonvar(X),
184   asserta(X endet_spaetestens_um Y
185     :- !).
187 /* ---------------------------- */
190 X beginnt_spaetestens_um Y :-
191   X endet_spaetestens_um Z,
192   X dauert T, Y is Z-T,nonvar(X),
193   asserta(X beginnt_spaetestens_um Y
194     :- ! ).
195 /* ---------------------------- */
200 X ist_kritisch :-
201   X ist_Aktivitaet,
202   X beginnt_fruehestens_um Y,
203   X beginnt_spaetestens_um Y.
205 /* ---------------------------- */
300 [X] ist_kritischer_Pfadteil :-
331   X ist_Schlussaktivitaet,
332   X ist_kritisch.
340 [X,Y|Z] ist_kritischer_Pfadteil :-
341   X ist_kritisch,
342   X vor Y,
343   [Y|Z] ist_kritischer_Pfadteil.
345 /* ---------------------------- */
350 [X|Y] ist_kritischer_Pfad :-
351   X ist_Startaktivitaet,
352   [X|Y] ist_kritischer_Pfadteil.
355 /* ---------------------------- */
```

```
360 X ist_insgesamt_verschieblich_um T :-
361   X beginnt_fruehestens_um Z,
362   (X beginnt_spaetestens_um Y),
363   (T is Y-Z).
365 /* ---------------------------- */
370 X ist_frei_verschieblich_um T :-
371   X ist_Schlussaktivitaet,
372   Z dauert_das_Projekt,
373   X endet_fruehestens_um Y,
374   T is Z-Y.
380 X ist_frei_verschieblich_um T :-
381   findall(X1,[X vor Y1,
382     Y1 beginnt_fruehestens_um X1],Z),
383   min(Z1,Z),
384   X endet_fruehestens_um T1,
385   T is Z1 - T1.
395 /* ---------------------------- */
396 /*        Hilfsroutinen         */
397 /* ---------------------------- */
410 min(A,[X|Y]) :- min(A,Y),A =< X,!.
420 min(X,[X|Y]).
440 max(A,[X|Y]) :- max(A,Y),A >= X,!.
450 max(X,[X|Y]).
500 findall(X,G,_) :-
501   asserta(found(mark)),
502   call(G),
503   asserta(found(X)),fail.
510 findall(_,_,L) :-
511   collectfound([],M),!,L=M.
520 collectfound(S,L) :-
521   getnext(X),!,
522   collectfound([X|S],L).
530 collectfound(L,L).
540 getnext(X) :- retract(found(X)),!,
541   X /== mark.
999 %%
```

17.2.12 mapping

```
1 /*   maplist(p,L,Z)
2      wendet Praedikat p auf Liste X
3      an und liefert das Ergebnis in Z.
4 */
10 maplist(_,[],[]).
20 maplist(P,[X|L],[Y|M]) :-
30    Q =.. [P,X,Y],
40    call(Q),maplist(P,L,M).
41 /*---------------------------------
42     applist(p,L)
43     wendet Praedikat p auf Liste X
44     an.
45 */
50 applist(_,[]).
60 applist(P,[X|L]) :-
70   Q =.. [P|X],
80   call(Q),
90   applist(P,L).
91 /*---------------------------------
92       change ist nur ein Beispiel
93       fuer maplist(change,L,Z).
94 */
100 change(du,ich).
110 change(bist,bin).
120 change(franzose,deutscher).
130 change(ein,kein).
140 change(X,X).
150 /*---------------------------------
155 Verschieden Sortierverfahren:
156 Praktisch verwendbar ist nur der
157 der Quicksort (Geschwindigkeit).
195 ---------------------------------
196    sort sortiert natuerlich:
197    alle Kombinationen erzeugen und
198    pruefen, ob sie sortiert sind.
199 */
200 sort(L1,L2) :-
201        permutation(L1,L2),
202        sorted(L2),!.
209 /* alle kombinationen erzeugen */
210 permutation(L,[H|T]) :-
211        append(V,[H|U],L),
212        append(V,U,W),
```

```
213         permutation(W,T).
220 permutation([],[]).
229 /* testen ob sortiert */
230 sorted(L) :- sorted(0,L).
240 sorted(_,[]).
250 sorted(N,[H|T]) :-
251         order(N,H),sorted(H,T).
297 /*-------------------------------
298    order ist das Sortierkriterium
299   --------------------------------*/
300 order(X,Y) :- X =< Y.
394 /*-------------------------------
395     busort ist der 'bubble sort'.
396     Benachbarte Elemente werden so-
397     lange getauscht, bis alles
398     alles sortiert ist.
399 */
400 busort(L,S) :-
401         append(X,[A,B|Y],L),
402         order(B,A),
403         append(X,[B,A|Y],M),
404         busort(M,S).
405 busort(L,L).
491 /*-------------------------------
492   quisort ist der 'quick sort':
493   a)  Liste splitten in zwei Listen
494         A  (alle Elemente <= H )
495         B  (alle ELemente > H  ) .
496    b) Beide Teillisten rekursiv
497         sortieren.
498    c) Die Listen verketten.
499 */
500 quisort([H|T],S) :-
501         split(H,T,A,B),
502         quisort(A,A1),
503         quisort(B,B1),
504         append(A1,[H|B1],S).
510 quisort([],[]).
520 split(H,[A|X],[A|Y],Z) :-
521         order(A,H),
522         split(H,X,Y,Z).
530 split(H,[A|X],Y,[A|Z]) :-
531         order(H,A),
532         split(H,X,Y,Z).
540 split(_,[],[],[]).
999 %%
```

17.2.13 math

```
1 /*          primes
2    primes(1,X) liefert in X alle
3    Primzahlen im Bereich 1 - 1.
4 */
10 primes(Limit,Ps) :-
11   integers(2,Limit,Is),
12   sift(Is,Ps).
20 integers(Low,High,[Low|Rest]) :-
30   Low =< High ,!,
31   M is Low+1,
32   integers(M,High,Rest).
40 integers(_,_,[]).
50 sift([],[]).
60 sift([I|Is],[I|Ps]) :-
61   remove(I,Is,New),
62   sift(New,Ps).
70 remove(P,[],[]).
80 remove(P,[I|Is],[I|Nis]) :-
90   not( 0 is I mod P),!,
91   remove(P,Is,Nis).
100 remove(P,[I|Is],Nis) :- 0 is I mod P,
101   !,remove(P,Is,Nis).
190 /*-------------------------------
191          gcd
192    gcd(X,Y,Z) liefert den groessten
193    gemeinsamen Teiler von X und Y
194    in Z.
195 */
200 gcd(I,0,I) :- !.
210 gcd(I,J,K) :-
211   R is I mod J,
212   gcd(J,R,K).
220 lcm(I,J,K) :-
```

```
221   gcd(I,J,R),
222   K is (I*J)/R.
300 /*-------------------------------
301                d
302     d(f,x,Z) differenziert den
303     Ausdruck f nach x und bringt das
304     Ergebnis nach Z.
305 */
310 ?- op(9,fx,**).
320 d(X,X,1) :- !.
325 d(C,X,0) :- atomic(C).
330 d(U+V,X,A+B) :- d(U,X,A),d(V,X,B).
335 d(U-V,X,A-B) :- d(U,X,A),d(V,X,B).
340 d(C*U,X,C*A) :-
341     atomic(C),C /= X ,d(U,X,A),!.
345 d(U*V,X,B*U+A*V) :-
346     d(U,X,A),d(V,X,B).
350 d(U**C,X,C*U**(C-1)*W) :-
351     atomic(C), C /= X , d(U,X,W).
355 d(U/V,X,(Y*V-W*U)/V**2) :-
356     d(U,X,Y),d(V,X,W).
360 d(-U,X,-V) :- d(U,X,V).
365 d(sin(U),X,V*cos(U)) :- d(U,X,V).
370 d(cos(U),X,-V*sin(U)) :- d(U,X,V).
375 d(tan(U),X,V/sin(U)**2) :- d(U,X,V).
380 d(cot(U),X,-V/cos(U)**2) :- d(U,X,V).
385 d(ln(U),X,-V/U) :- d(U,X,V).
396 /*-------------------------------
397             simplify
398   simplify(X,Z) vereinfacht Ausdruck X.
399 */
400 simplify(X,Z) :-
410       simp(X,Y),X/=Y,!,simplify(Y,Z).
420 simplify(X,X).
450 simp(E,E) :- atomic(E), ! .
455 simp(E,F) :-
460       E =.. [Op,La,Ra],
465       simp(La,X),
470       simp(Ra,Y),
475       s(Op,X,Y,F),!.
480 simp(E,F) :-
485       E =.. [Op,Ra],
490       simp(Ra,Y),
495       F =.. [Op,Y].
500 /* Vereinfachungen fuer +  */
510 s(+,X,0,X).
520 s(+,0,X,X).
525 s(+,X,-Y,X-Y).
526 s(+,-X,Y,Y-X).
530 s(+,X+Y,W,X+Z) :-
531          integer(Y),
532          integer(W),
533          Z is Y+W .
534 s(+,X-Y,W,X+Z) :-
535          integer(Y),
536          integer(W),
537          W>=Y,Z is W-Y .
538 s(+,X-Y,W,X-W) :-
539          integer(Y),
540          integer(W),
541           W<Y, Z is Y-W .
542 s(+,X+Y,W,Y+Z) :-
543          integer(X),
544          integer(W),
545          Z is X+W .
550 s(+,X,Y,Z) :-
551          integer(X),
552          integer(Y),
553          Z is X + Y.
560 s(+,X,Y,X+Y) /* catchall  */ .
565 /* Vereinfachungen fuer *  */
570 s(*,_,0,0).
580 s(*,0,_,0).
590 s(*,1,X,X).
600 s(*,X,1,X).
610 s(*,X*Y,W,X*Z) :-
611           integer(Y),
612           integer(W),
613           Z is Y*W .
620 s(*,X,Y,X*Y) /*  catchall */ .
700 /* Vereinfachungen fuer -  */
710 s(-,X,0,X).
720 s(-,X-Y,W,X-Z) :-
721         integer(Y),
722         integer(W),
723         Z is X+W .
730 s(-,X-Y,W,Y-Z) :-
```

```
731         integer(X),
732         integer(W),
733         X>=W,
734         Z is X - W .
740 s(-,X,Y,Z) :-
741         integer(X),
742         integer(Y),
743         X>=Y,
744         Z is X-Y .
750 s(-,X+Y,W,X+Z) :-
751         integer(Y),
752         integer(W),
753         X>=W,
754         Z is X-W .
760 s(-,X+Y,W,X-Z) :-
761         integer(Y),
762         integer(W),
763         X<W,
764         Z is W-Z .
790 s(-,X,Y,X-Y) /*  catchall  */ .
795 /* Vereinfachungen fuer /  */
800 s(/,X,1,X).
810 s(/,X*Y,X,Y).
820 s(/,Y*X,X,Y).
830 s(/,X,Y*X,1/Y).
840 s(/,X,X*Y,1/Y).
850 s(/,X*Y,X*Z,Y/Z).
860 s(/,X*Y,Z*X,Y/Z).
870 s(/,X,Y,W/Z) :-
871          integer(X),
872          integer(Y),
873          gcd(X,Y,G),
874          W is X/G,
875          Z is Y/G .
880 s(/,X,Y,X/Y) /*  catchall  */ .
890 /* Vereinfachungen fuer **   */
900 s(**,X,1,X).
910 s(**,_,0,X).
920 s(**,X**Y,W,X**Z) :-
921          integer(Y),
922          integer(W),
923          Z is Y*W .
930 s(**,X,Y,X**Y) /*  catchall */ .
999 %
```

17.2.14 minishell

```
1 /*-----------------------------------
2    Minishell
3 -----------------------------------
4  Minishell ist eine einfache Shell
5  fuer Expertensysteme.
6  Dazu gehoert eine Wissensbasis,
7  die das Expertenwissen enthaelt.
8  Deren Regeln werden mit den Opera-
9  toren 'weil', 'und' und 'nicht' auf-
10 gebaut. Bestimmte Praedikate koennen
11 als 'ist_erfragbar' gekennzeichnet
12 werden.
20 -----------------------------------
21 Das jeweilige Expertensystem wird
22 mit 'run' gestartet.
23 Dann wird die Frage an den Experten
24 eingegeben.
25 Die Shell versucht dann mit Hilfe
26 der Regeln und Fakten der Wissens-
27 basis eine Antwort zu finden.
28 Fuer erfragbare Praedikate stellt
29 die Shell automatisch die notwen-
30 digen Fragen, falls diese nicht be-
31 reits beantwortet sind.
32 -----------------------------------
33 Sie haben die Moeglichkeit, auf die
34 Frage mit 'ja.' oder 'nein.' zu ant-
35 worten oder Werte einzugeben.
36 Sie koennen auch mit 'warum.' ant-
37 worten. Die Shell erklaert Ihnen
38 dann den Grund der Frage.
40 Am Schluss werden Sie gefragt, ob
41 Sie mit der Antwort zufrieden sind.
42 Falls Sie 'nein.' eingeben, versucht
43 die Shell eine andere Loesung zu
```

```
44 finden. Oder wenn Sie 'warum.' ein-
45 geben, erklaert Ihnen die Shell,
46 wie sie zu der Loesung gekommen ist.
49 -----------------------------------
50 ( siehe auch Kapitel Expertensysteme
51 im Handbuch.)
90 -----------------------------------
91 Operatoren der Wissensbasis :
92 -----------------------------------*/
100 ?-op(29,yfx,weil).
115 ?-op(27,xfy,und).
120 ?-op(26,fx,nicht).
130 ?-op(25,xf,ist_erfragbar).
137 /*-----------------------------------
138 Weitere Operatoren
139 -----------------------------------*/
140 ?-op(25,xf,wurde_bejaht).
150 ?-op(25,xf,wurde_verneint).
190 /*-----------------------------------
191 Hauptschleife
192 -----------------------------------*/
220 run :- repeat,nl,
221     retractall(_ wurde_bejaht),
222     retractall(_ wurde_verneint),
223     write('Bitte Frage eingeben!'),
224     nl,nl,read(X),nl,
226     givans(X),nl,
228     fail.
240 givans(X) :- process(X,[],S),
242     nl,write(X),nl,
244     ask('Mit der Antwort zufrieden',S),!.
250 givans(X) :-
252     write('Weiss ich auch nicht.').
290 /*-----------------------------------
291 Inferenzmaschine
292 -----------------------------------*/
300 process(nicht X,T,[X]) :-
302     process(X,T,S),!,fail.
304 process(nicht X,_,[nicht X]) :- !.
310 process(X und Y,T,S) :- !,
312     process(X,T,S1),process(Y,T,S2),
314     union(S1,S2,S).
320 process(X,T,S) :- functor(X,F,_),
322     F ist_erfragbar,!,
324     confirm(X,T,S).
340 process(X,_,[X]) :- call(X).
350 process(X,T,[X weil Y|S]) :-
352     X weil Y,
354     process(Y,[X weil Y|T],S).
390 /*-----------------------------------
391 Fragemechanismus
392 -----------------------------------*/
400 confirm(X,_,[X wurde_bejaht]) :-
402     X wurde_bejaht.
420 confirm(X,T,[X wurde_bejaht]) :-
422     not X wurde_bejaht,
424     not X wurde_verneint,
426     ask(X,T),
428     assertz(X wurde_bejaht).
430 confirm(X,_,[X wurde_verneint]) :-
432     not X wurde_bejaht,
434     not X wurde_verneint,
436     assertz(X wurde_verneint),
438     fail.
490 /*-----------------------------------
491 Frage stellen
492 -----------------------------------*/
500 ask(X,T) :-
502     write(X),write(' ?'),
504     read(A),checkans(X,A,T).
550 checkans(X,ja,_) :- !.
560 checkans(X,nein,_) :- !,fail.
570 checkans(X,warum,T) :-
571     write('Ich gehe von folgenden Annahmen aus:'),
572     nl,explain(T),!,
574     ask(X,T).
580 checkans(X,Y,_) :- assign(X,Y).
590 /*-----------------------------------
591 Erklaerungskomponente
592 -----------------------------------*/
600 explain([A]) :- explrule(A,0),!,nl.
610 explain([A|L]) :- explrule(A,0),
612     more,!,explain(L).
620 explain(_).
630 explrule(X weil Y,0) :-
632     nl,write(X),write(' weil '),
634     nl,write('1) '),!,
```

```
636     explrule(Y,2).
640 explrule(X und Y,N) :-
642     write(X),write(' und'),
644     nl,write(N),write(') '),!,
646     N1 is N+1, explrule(Y,N1).
650 explrule(nicht X,N) :-
654     write('nicht '),!,
656     explrule(X,N).
660 explrule(X,_) :-
662     write(X),nl.
670 more :-
672     nl,write('mehr ? (j/n)'),
674     nl,get(X),X is "j".
690 /*---------------------------------
691 Antwort auswerten
692 ---------------------------------*/
700 assign(X,X).
710 assign(X,Y) :- X =.. [_|L],
712     asslist(L,Y).
720 asslist([],[]) :- !.
730 asslist([X],X) :- !.
740 asslist([B|X],[B|Y]) :- !,
742     asslist(X,Y).
750 asslist([_|X],Y) :- asslist(X,Y).
790 /*---------------------------------
791 Hilfsprogramme
792 ---------------------------------*/
800 union([],X,X).
810 union([X|R],Y,Z) :-
812     member(X,Y),!,
814     union(R,Y,Z).
820 union([X|R],Y,[X|Z]) :-
822     union(R,Y,Z).
890 /* ------------------------------*/
```

17.2.15 music

```
1 /*---------------------------------
2             PLAY
3 play(L,K) spielt Liste L in Klang-
4 farbe K. L besteht aus Elementen
5 d(T,D) ( T=Tonnummer, D=Dauer )
6 oder aus Elementen P (P=Pausendauer)
8 */
10 play([],_) :- !.
20 play([d(Ton,Dauer)|Rest],K) :-
30 sound(Ton,Dauer,K),!,play(Rest,K).
40 play([Dauer|Rest],K) :- pause(Dauer),play(Rest,K).
94 /*---------------------------------
95             MAY
96 may(K) spielt 'Komm lieber Mai und
97 mache ...' von W.A. Mozart in Klang-
98 farbe K.
99 */
100 may(K) :-
110 play2([d(4,1),d(4,2),fis(4,1),
120 a(4,2),d(5,1),a(4,4),fis(4,1),
130 d(4,1),g(4,2),g(4,1),g(4,1),
140 a(4,1),g(4,1),fis(4,3),pause(2)],15,K),
150 play2([d(4,1),d(4,2),fis(4,1),
160 a(4,2),d(5,1),a(4,4),fis(4,1),
170 d(4,1),e(4,2),e(4,1),e(4,1),
180 fis(4,1),e(4,1),d(4,3),pause(2)],15,K),
190 play2([fis(4,1),g(4,2),dis(4,1),
200 e(4,1),fis(4,1),g(4,1),a(4,2),
210 fis(4,1),d(5,2),pause(1),d(5,1),d(5,1),
220 cis(5,1),h(4,1),h(4,1),a(4,1),gis(4,1),a(4,3),pause(2)],15,K),
230 play2([d(4,1),d(4,2),fis(4,1),
240 a(4,2),d(5,1),d(5,1),h(4,1),
250 g(4,1),e(4,2),h(4,1),a(4,1),
255 fis(4,1),a(4,1),g(4,1),
260 fis(4,1),e(4,1),d(4,3),pause(2)],15,K).
491 /*---------------------------------
492             PLAY2
493 play(L,K) spielt Liste L in
494 Klangfarbe K und Tempo T.
495 L besteht aus den ELementen
496 tonname(o,d) oder pause(d)
497 ( o = oktave, t = dauer ) .
498 */
500 play2([],_,_) :- !.
510 play2([A|Ai],To,K) :-
```

```
511     playn(A,To,K),play2(A1,To,K).
520 playn(pause(D),To,_) :-
521     pause(D*To),!.
530 playn(A,To,K) :-
531     functor(A,F,_),note(F,Fn),
540     arg(1,A,Ok),arg(2,A,D),
560     sound(Fn+Ok*12,To*D,K).
600 note(c,0).
610 note(cis,1).
620 note(d,2).
630 note(dis,3).
640 note(e,4).
650 note(f,5).
660 note(fis,6).
670 note(g,7).
680 note(gis,8).
690 note(a,9).
700 note(b,10).
710 note(h,11).
999 %%
```

17.2.16 project

```
1 /* ----------------------------------
2     project
3   ----------------------------------
4   project ist ein Beispiel fuer die
5   die Anwendung der Bibliothek
6   logiplan (Netzplantechnik).
7   Zunaechst wird logiplan konsultiert:
9   ----------------------------------*/
10 ?-consult logiplan.
11 /* ----------------------------------
12 Dann erfolgt die Definition der
13 einzelnen Aktivitaeten mit ihrer
14 Dauer.
15 (Das Praedikat 'dauert' ist in
16 logiplan als Operator definiert.)
17 Beachten Sie, dass die Schreibweise
18 der Aktivitaeten der Schreibweise
19 fuer Atome entsprechen muss.
20 ------------------------------------
21 Das Beispiel behandelt sehr ver-
22 einfacht einen Hausbau. Die Zeiten
23 sind in Tagen angegeben.
25 ----------------------------------*/
100 'Keller_ausschachten' dauert 5.
110 'Maurerarbeiten' dauert 29.
120 'Zimmererarbeiten' dauert 16.
130 'Dachdecken' dauert 8.
140 'Elektroinstallation' dauert 10.
150 'Sanitaerinstallation' dauert 14.
160 'Malerarbeiten' dauert 17.
170 'Schreinerarbeiten' dauert 11.
180 'Glaserarbeiten' dauert 9.
190 'Heizungsmontage' dauert 6.
200 'Einzug' dauert 6.
250 /* ----------------------------------
251 Dann wird definiert, welche Aktivi-
252 taeten voneinander abhaengig sind,
253 d.h. welche Aktivitaeten beendet
254 sein muessen, damit mit einer
255 neuen Aktivitaet begonnen werden
256 kann.
257 ----------------------------------*/
300 'Keller_ausschachten' vor
301   'Maurerarbeiten'.
310 'Maurerarbeiten' vor
311   'Zimmererarbeiten'.
312 'Maurerarbeiten' vor
313   'Elektroinstallation'.
314 'Maurerarbeiten' vor
315   'Sanitaerinstallation'.
316 'Maurerarbeiten' vor
317   'Schreinerarbeiten'.
318 'Maurerarbeiten' vor
319   'Heizungsmontage'.
320 'Zimmererarbeiten' vor
321   'Dachdecken'.
330 'Dachdecken' vor
331 'Einzug'.
342 'Elektroinstallation' vor
```

```
343 'Malerarbeiten'.
352 'Sanitaerinstallation' vor
353 'Malerarbeiten'.
360 'Malerarbeiten' vor
361 'Einzug'.
370 'Schreinerarbeiten' vor
371 'Glaserarbeiten'.
380 'Glaserarbeiten' vor
381 'Einzug'.
390 'Heizungsmontage' vor
400 'Malerarbeiten'.
999 %
```

17.2.17 search

```
1 /*          maze
2    maze(a,b,[])    sucht in einem
3    Irrgarten einen Weg von a nach b.
4 */
10 maze(X,X,T).
20 maze(X,Y,T) :-
21            (d(X,Z) ; d(Z,X)),
22            not(member(Z,T)),
23            maze(Z,Y,[Z|T]).
25 /*
26     Beispiel fuer Irrgarten
27 */
30 d(a,b).
40 d(b,e).
50 d(b,c).
60 d(d,e).
70 d(c,d).
80 d(e,f).
90 d(g,e).
195 /*--------------------------------
196            hanoi
197      Die Tuerme von Hanoi
198 */
210 hanoi(N) :-
211       moves(N,left,centre,right).
220 moves(0,_,_,_) :- ! .
230 moves(N,A,B,C) :-
240         M is N-1,
250         moves(M,A,C,B),
260         inform(A,B),
270         moves(M,C,B,A).
280 inform(X,Y) :-
281          write[move,a,disc,from,the,X,pole,to,the,Y,pole]),
282          nl.
294 /*--------------------------------
295            findall
296   findall(X,G,L)  findet alle
297   X, die Bedingung G genuegen
298   und legt sie in Liste L ab.
299 */
300 findall(X,G,_) :-
310          asserta(found(mark)),
320          call(G),
330          asserta(found(X)),
340          fail.
350 findall(_,_,L) :-
351          collectfound([],M),
352          !,L=M.
360 collectfound(S,L) :-
361          getnext(X),!,
362          collectfound([X|S],L).
370 collectfound(L,L).
380 getnext(X) :-
381          retract(found(X)),!,
382          X /== mark.
394 /*--------------------------------
395            go
396      go(a,b,Z)      findet einen
397      Weg von a nach b und liefert
398      das Ergebnis in Z.
399 */
400 go(Start,Dest,Route) :-
401  go1([[Start]],Dest,R),rev(R,Route).
405 go1([First|Rest],Dest,First) :-
406           First = [Dest| _ ].
410 go1([[Last|Trail]|Others],Dest,Route) :-
420      findall([Z,Last|Trail],legalnode(Last,Trail,Z),List),
```

```
430      append(Others,List,NewRoutes),
440      go1(NewRoutes,Dest,Route).
500 legalnode(X,Trail,Y) :-
501       (a(X,Y) ; a(Y,X)),
502       not(member(Y,Trail)).
595 /*-------------------------------
596        beispiel fuer Wegeplan
597 /*-------------------------------*/
600 a(newcastle,carlisle,58).
610 a(carlisle,penrith,23).
614 a(townB,townA,16).
620 a(darlington,newcastle,40).
624 a(townB,townC,10).
630 a(penrith,darlington,52).
634 a(workington,townC,5).
640 a(workington,carlisle,33).
644 a(darlington,townA,25).
650 a(workington,penrith,39).
660 a(X,Y) :- a(X,Y,Z).
694 /*-------------------------------
695              path
696     path(a,b,Z)       findet den
697     kuerzesten Weg von a nach b
698     und liefert das Ergebnis in Z.
699 */
700 path(Start,Dest,Route) :- go3([r(0,[Start])],Dest,R),rev(R,Route).
710 go3(Routes,Dest,Route) :-
720 shortest(Routes,Shortest,RestRoutes),
730 proceed(Shortest,Dest,RestRoutes,Route).
740 proceed(r(Dist,Route),Dest,_,Route) :- Route = [Dest|_].
750 proceed(r(Dist,[Last|Trail]),Dest,Routes,Route) :-
760 findall(r(D1,[Z,Last|Trail]),legalnode(Last,Trail,Z,Dist,D1),List),
770 append(List,Routes,NewRoutes),
780 go3(NewRoutes,Dest,Route).
790 shortest([Route|Routes],Shortest,[Route|Rest]) :-
800 shortest(Routes,Shortest,Rest),
810 shorter(Shortest,Route),!.
820 shortest([Route|Rest],Route,Rest).
830 shorter(r(W1,_),r(W2,_)) :- W1 < W2.
840 legalnode(X,Trail,Y,Dist,NewDist) :-
850 (a(X,Y,Z) ; a(Y,X,Z)), not(member(Y,Trail)),
860 NewDist is Dist + Z.
999 %%
```

17.2.18 set

```
1 /*  subset(X,Y)
2      ist wahr, wenn X eine Untermenge
3      von Y ist.
4 */
10 subset([A|X],Y) :-
11         member(A,Y),
12         subset(X,Y).
20 subset([],Y).
30 /*-------------------------------
31      intersection(X,Y,Z)
32      Z ist der Mengendurchschnitt von
33      X und Y.
34 */
40 intersection([],X,[]).
50 intersection([X|R],Y,[X|Z]) :-
60        member(X,Y),!,
61        intersection(R,Y,Z).
70 intersection([X|R],Y,Z) :-
71        intersection(R,Y,Z).
75 /*-------------------------------
76      union(X,Y,Z)
77      Z ist die Mengenvereinigung von
78      X und Y.
79 */
80 union([],X,X).
90 union([X|R],Y,Z) :-
91         member(X,Y),!,
92         union(R,Y,Z).
100 union([X|R],Y,[X|Z]) :-union(R,Y,Z).
105 /*-------------------------------
106      difference(X,Y)
107      difference ist wahr, wenn X und
108      Y kein gemeinsames Element haben.
109 */
110 difference(X,Y) :-
111     not([member(Z,X),member(Z,Y)]).
```

```
113 /*--------------------------------
114      flatten(X,Z)
115      Die mehrstufige Liste X wird
116      flach gemacht, z.B. :
117      [a,[b,c],[],[d]]  --> [a,b,c,d]
119 */
120 flatten([],[]) :- ! .
130 flatten([A|B],Z) :-
131         append(A,B,Y),!,
132         flatten(Y,Z).
140 flatten([A|B],[A|Z]) :-flatten(B,Z).
300 %
```

17.2.19 tcogrammar

```
10 /*    tco fuer Grammatikregeln
20 */
100 ?- op(63,xfy,-->) /* ableitung  */.
110 ?- op(59,fx,<<) /*anfang clause */.
120 ?- op(58,xf,>>) /* ende clause  */.
190 /* */
200 tco((P0-->Q0),(P:-Q)) :-
210   lefthandside(P0,S0,S,P),
220   righthandside(Q0,S0,S,Q).
295 /*
296   Kopf der Regel verarbeiten
297 */
300 lefthandside([NT|Ts],S0,S,P) :-
310   !,nonvar(NT),
311   islist(Ts),
312   tag(NT,S0,S1,P),
313   append(Ts,S0,S1).
320 lefthandside(NT,S0,S,P) :-
330   nonvar(NT),
332   tag(NT,S0,S,P).
370 righthandside(P,S0,S,true) :-
380 last(X,P),!,append(P,S,S0).
395 /*
396   Endteil der Regel verarbeiten
397 */
400 righthandside([X1|X2],S0,S,P) :-
410   righthandside(X1,S0,S1,P1),
411   righthandside(X2,S1,S,P2),
412   and(P1,P2,P).
420 righthandside((X1;X2),S0,S,(P1;P2))
421   :- !,
430   or(X1,S0,S1,P1),
431   or(X2,S1,S,P2).
440 righthandside(<< P >>,S,S,P) :- !.
450 righthandside(!,S,S,!) :- !.
460 righthandside(Ts,S0,S,true) :-
461   reallist(Ts),!,
462   append(Ts,S,S0).
470 righthandside(X,S0,S,P) :-
480   tag(X,S0,S,P).
495 /*
496   Oder - Verknuepfung verarbeiten
497 */
500 or(X,S0,S,P) :-
510   righthandside(X,S0a,S,Pa),
511   ( var(S0a),S0a==S,!,S0=S0a,P=Pa;
512     P=[S0=S0a|Pa] ).
595 /*
596   tag
597 */
600 tag(X,S0,S,P) :-
610   X =.. [F|A],
611   append(A,[S0,S],AX),
612   P =.. [F|AX] .
695 /*
696   Und - Verknuepfung
697 */
700 and(true,P,P) :- ! .
710 and(P,true,P) :- ! .
720 and(P,Q,[P|Q]).
795 /*
796   Echte Liste (Unterschied zu
797      Konjunktion )
798 */
800 reallist([]) :- !.
810 reallist([_|X]) :- reallist(X).
```

```
999 %
```

17.3 Programme auf der Diskette

17.3.1 liesmich

```
10 "Version 1.2 von PROLOG 64 enthaelt
20 "ausser einigen kleineren Fehler-
30 "korrekturen eine Erweiterung:
40 "Die Moeglichkeit Assemblerunter-
50 "programme in einem PROLOG Programm
60 "aufzurufen.
70 "Wie das geht, erfahren Sie in
80 " asm.docu.
90 "Ein Beispielprogramm finden Sie in
100 " asm.example.
110 "Wenn Sie selbst keine Unterprogramme
120 "schreiben wollen, koennen Sie den
130 "Primitive dafuer als Nullseiten-
140 "poke verwenden (@ anzuwenden wie
150 " poke).
160 "Die Nullseitenbelegung finden Sie
170 "auch in asm.docu.
```

17.3.2 asm.docu

```
10 data"Assemblerunterprogramme in
20 data"       PROLOG 64
30 data"-----------------------------------
40 data"Fuer den Aufruf von Assembler-
50 data"unterprogrammen von PROLOG 64 aus
60 data"steht das Praedikat '@' zur Ver-
70 data"fuegung. Standardmaessig ist dieses
80 data"Praedikat als Nullseitenpoke defi-
90 data"niert (aehnlich POKE).
100 data"Das zugehoerige Assemblerprogramm
110 data"ist im Lader als DATA abgelegt und
120 data"kann ausgetauscht oder erweitert
130 data"werden. Als Speicherplaetze fuer
140 data"Assemblerroutinen stehen folgende
150 data"Speicherplaetze zur Verfuegung:
160 data"1. Kassettenpuffer $33c-$3fb
170 data"2. RS232-Puffer bzw. EPSON-Inter-
180 data"   face-Bereich $9a90-$9fff
190 data"   Dieser Bereich weist eine
200 data"   Besonderheit auf. Wenn auf
210 data"   Adresse $9a90 der Assembler-
220 data"   befehl SEI ($78) steht, wird
230 data"   beim Start von PROLOG oder be
240 data"   STOP/RESTORE die dort stehende
250 data"   Routine durchlaufen.
260 data"Die Assemblerroutinen muessen im
270 data"Lader an die vorgesehenen Speicher-
280 data"plaetze gebracht werden. Die
290 data"Adresse der Routine muss in den
300 data"USR-Vektor ($311-$312) gepokt wer-
310 data"den.
320 data"Die Routine wird in PROLOG mit
330 data" @(arg1,arg2) aufgerufen.
340 data"Der Assemblerroutine werden die
350 data"Argumente in $22-23 bzw. $24-25
360 data"uebergeben.
370 data"Dies koennen natuerlich auch
380 data"Variable sein, denen ein Wert zu-
390 data"wiesen wurde. Um diesen Wert zu
400 data"erhalten, wird Routine $a000
410 data"aufgerufen.
420 data"Diese Routine erwartet den Eingabe-
430 data"wert im x- und a-Register (Lower
440 data"Byte in x). Der Ausgabewert steht
450 data"in $22-23.
460 data"Das x-Register enthaelt den Typ des
470 data"Ergebnisses:
480 data" 0 = integer
```

```
490 data" 1 = atom
500 data" 2 = functor
510 data" 3 = freie Variable
520 data" 4 = Liste
530 data"   integer Werte werden in einer
540 data"   speziellen Form gespeichert:
550 data"   $a000 = 0, $bfff = 8191.
560 data"Das Carry-bit ist aus, wenn das
570 data"Ergebnis eine freie Variable ist.
580 data"Ein Seiteneffekt der Routine $a000
590 data"ist die Veraenderung des frame
600 data"pointers auf $57-58.
610 data"Bei mehrfacher Verwendung von
620 data"Routine $a000 ist der frame pointer
630 data"ggf. zu sichern.
640 data"Bei der Rueckkehr aus der Benutzer-
650 data"routine gibt es drei Moeglich-
660 data"keiten:
670 data"1. Rueckkehr mit RTS
680 data"   bewirkt einen Match der Argu-
690 data"   mente auf $22-23 und $24-26.
700 data"   Das Ergebnis haengt vom Ausgang
710 data"   des Match ab.
720 data"2. Rueckkehr mit PLA,PLA,SEC,RTS.
730 data"   Es wird kein Match durchge-
740 data"   fuehrt, sondern die Routine
750 data"   wird mit Erfolg beendet.
760 data"3. Rueckkehr mit PLA,PLA,CLC,RTS.
770 data"   Es wird kein Match durchge-
780 data"   fuehrt, sondern die Routine
790 data"   wird mit 'fail' beendet.
800 data"Falls Sie von der Assemblerroutine
810 data"aus das Betriebssystem aufrufen
820 data"wollen, muessen Sie zunaechst den
830 data"ROM einschalten:
840 data"   lda #$e7
850 data"   sta 1
860 data"und danach wieder ausschalten:
870 data"   lda #$35
880 data"   sta 1
890 data"
900 data"Als Beispiel finden Sie auf der
910 data"Diskette das Assemblerquellprogramm
920 data"fuer den Nullseitenpoke, wie er im
930 data"Lader definiert ist.
940 data"
950 data"Folgende Speicherplaetze auf der
960 data"Nullseite werden von PROLOG 64
970 data"belegt:
1000 data" $3-4     working
1010 data" $5-6     stack top
1020 data" $7-8     heaptop
1030 data" $14-15   current frame
1040 data" $22-23   1. argument
1050 data" $24-25   2. argument
1060 data" $26-27   3. argument
1070 data" $39-3c   work
1080 data" $45-46   ancestor frame
1090 data" $47-4d   working
1100 data" $4e      line counter
1110 data" $4f      column counter
1120 data" $57-58   frame pointer
1130 data" $59-5a   work
1140 data" $5b-5c   pointer to variable frame
1150 data" $5d-5e   work
1160 data" $5f-60   frame pointer during exit
1170 data" $61-62   read pointer
1180 data" $63      work
1190 data" $64      step switch debug
1200 data" $65      prompt switch
1210 data" $66      current variable id
1220 data" $67-6a   work
1230 data" $6b      current state
1240 data" $6c-6d   current read pointer
1250 data" $6e      current primitive id
1260 data" $6f      top of stack
1270 data" $70-71   work
1280 data" $72      pen color
1290 data" $8b-8f   work garbage collection
1300 data" $216-225 work
1310 data" $240-241 output stream
1320 data" $242-243 input stream
1330 data" $24e-251 work
1340 data"zzz
1490 s=s+1:print"                                        ";s
1500 for i=1 to 20
1510 read x$:ifx$="zzz"then i=20:goto1530
```

```
1515 print"  ";x$
1520 next
1530 print"CTRL       druecken sie eine taste          "
1540 geta$:ifa$=""then1540
1550 ifx$ <> "zzz" then 1490
```

17.3.3 asm.example

```
1000 :exmpl  *=  $33c; kassettenpuffer
1010 :
1020 : ; das ist ein beispiel fuer ein
1030 : ; assembler unterprogramm .
1040 : ; es erlaubt zugriff zu der null-
1050 : ; seite und den systemparametern
1060 : ; von $00 - $3ff .
1070 :
1080 :       lda $57; frame pointer sichern
1090 :       pha
1100 :       lda $58;
1110 :       pha
1120 :       ldx $22; 1. argument holen
1130 :       lda $23
1140 :       jsr $a000; wert holen
1150 :       cpx #$00 ; integer
1160 :       bne exmf ; nein
1170 :       lda $23  ; oberes byte
1180 :       and #$1f ; int bits entfernen
1190 :       cmp #8   ; nicht nach adr
1200 :       bcs exmf ; > $7ff poken
1210 :       sta $27  ; nullseitenadr
1220 :       lda $22  ;  aufbauen
1230 :       sta $26
1240 :       pla      ; frame pointer
1250 :       sta $58  ; ruecksetzen
1260 :       pla
1270 :       sta $57
1280 :       ldx $24  ; 2. argument holen
1290 :       lda $25
1300 :       jsr $a000 ; wert holen
1310 :       bcc exmpl1 ; variable
1320 :       cpx #$00 ; integer
1330 :       bne exmfi; nein
1340 :       lda $23  ; oberes byte
1350 :       cmp #$a0 ; > 255
1360 :       bne exmfi; ja
1370 :       lda $22  ; unteres byte
1380 :       ldy #0
1390 :       sta ($26),y ; poken
1400 :       pla      ; kein match
1410 :       pla
1420 :       sec      ; erfolg signalis.
1430 :       rts      ; fertig
1440 :
1450 :exmpl1 ldy #$00
1460 :       lda ($26),y ; wert holen
1470 :       sta $22     ; nach arg 1
1480 :       lda #$a0    ; oberes int byte
1490 :       sta $23
1500 :       rts       ; match
1510 :
1520 :exmf   pla       ; stack saeubern
1530 :       pla
1540 :exmfi  pla       ; kein match
1550 :       pla
1560 :       clc       ; fail
1570 :       rts
1580 :       .end
```

17.3.4 prolog

```
1 S$ ="": REM  STARTUP-BEFEHL   ------
2 REM ---------------------------------
3 REM ----- RAHMENFARBE,ERRORFARBE,TRACEFARBE,RS232-KONTROLLREGISTER ----------
4 SC=3:EC=4:TC=5:BC=1:CC=6:RC=138
5 RC=138:RI=224:IFA=1THEN500
6 REM
7 REM ---- SOUND CONTROL 5 REGISTER ---
```

```
8 REM
9 REM REGISTER 0 -----------------
10 DATA 0,0,0,0,33,153,137,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15
11 REM REGISTER 1 ---------------
12 DATA 0,0,0,0,33,1,218,0,4,0,0,33,1,218,0,7,0,0,33,1,218,0,0,0,15
13 REM REGISTER 2 ---------------
14 DATA 0,0,0,128,65,153,137,0,0,0,128,65,153,137,0,0,0,0,17,170,154,0,0,0,15
15 REM REGISTER 3 ---------------
16 DATA 0,0,0,0,17,153,137,8,0,0,0,17,153,137,24,0,0,0,17,153,137,0,0,0,15
17 REM REGISTER 4 ---------------
18 DATA 0,7,0,0,129,144,141,0,0,0,0,0,0,0,0,0,0,1,65,31,0,0,0,0,15
20 FOR I=1TO125
21 READ X: POKE39424+I,X
22 NEXT
29 REM == ASM-SUBROUTINE ===============
30 DATA165,087,072,165,088,072,166,034,165,035,032,000,160,224,000,208
31 DATA060,165,035,041,031,201,004,176,052,133,039,165,034,133,038,104
32 DATA133,088,104,133,087,166,036,165,037,032,000,160,144,020,224,000
33 DATA208,029,165,035,201,160,208,023,165,034,160,000,145,038,104,104
34 DATA056,096,160,000,177,038,133,034,169,160,133,035,096,104,104,104
35 DATA104,024,096
36 FORI= 828TO 910:READX:POKEI,X:NEXT
40 POKE785,60:POKE786,3
89 REM ================================
90 A=1:POKE53281,11:POKE53280,11:PRINT"■":PRINT:PRINT CHR$(142)
100 PRINT"■   ┌─┐ ┌─┐ ┌─┐ ┌┐   ┌─┐ ┌─┐"
110 PRINT"■   │ │ │ │ │ │ ││   │ │ │ │"
120 PRINT"■   ├─┘ ├─┘ │ │ ││   │ │ │ ┌┘"
130 PRINT"■   │   │ \ │ │ └┘┐  │ │ │ │"
140 PRINT"■   │   │  \└─┘ └──┘ └─┘ └─┘"
150 PRINT
160 PRINT"■                    V1.2"
170 PRINT
200 PRINT"■              ┌─┐ │ │  "
220 PRINT"■              │ │ │ │  "
230 PRINT"■              ├─┐ └─┘  "
240 PRINT"■              │ │   │  "
250 PRINT"■              └─┘   │  "
260 PRINT
270 PRINT
280 PRINT
300 PRINT"■■              COPYRIGHT  1986"
310 PRINT
320 PRINT"■             DR. BERTHOLD DAUM"
350 LOAD "P",8,1
500 POKE 53247,PEEK(44)
515 POKE 53246,SC:POKE53245,EC:POKE53244,TC
516 POKE 659,RC:POKE660,RI
520 POKE51347,CC:POKE51359,CC:POKE51348,BC:POKE51360,BC
590 PRINT "■■"
600 IF S$=""THENPOKE38400,0:GOTO999
605 S$=S$+CHR$(13)
610 FOR I=1TOLEN(S$)
620 POKE 38400+I,ASC(MID$(S$,I,1))
630 NEXT:POKE38400,255
640 PRINT CHR$(14);CHR$(8);
999 SYS49152
```

17.3.5 demoboot

```
1 S$ ="CONSULT DEMO,TOPDEMO"
2 REM --------------------------------
3 REM ----- RAHMENFARBE,ERRORFARBE,TRACEFARBE,RS232-KONTROLLREGISTER ----------
4 SC=3:EC=4:TC=5:BC=1:CC=6:RC=138
5 RC=138:RI=224
6 REM
7 REM ---- SOUND CONTROL 5 REGISTER ---
8 REM
9 REM REGISTER 0 -----------------
10 DATA 0,0,0,0,33,153,137,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15
11 REM REGISTER 1 ---------------
12 DATA 0,0,0,0,33,1,218,0,4,0,0,33,1,218,0,7,0,0,33,1,218,0,0,0,15
13 REM REGISTER 2 ---------------
14 DATA 0,0,0,128,65,153,137,0,0,0,128,65,153,137,0,0,0,0,17,170,154,0,0,0,15
15 REM REGISTER 3 ---------------
16 DATA 0,0,0,0,17,153,137,8,0,0,0,17,153,137,24,0,0,0,17,153,137,0,0,0,15
17 REM REGISTER 4 ---------------
18 DATA 0,7,0,0,129,144,141,0,0,0,0,0,0,0,0,0,0,1,65,31,0,0,0,0,15
19 REM ================================
89 IF A=1 THEN 500
90 A=1:POKE53281,11:POKE53280,11:PRINT"■":PRINT:PRINT CHR$(142)
100 PRINT"■   ┌─┐ ┌─┐ ┌─┐ ┌┐   ┌─┐ ┌─┐"
```

```
110 PRINT"
120 PRINT"
130 PRINT"
140 PRINT"
150 PRINT
160 PRINT
170 PRINT
200 PRINT"
220 PRINT"
230 PRINT"
240 PRINT"
250 PRINT"
260 PRINT
270 PRINT
280 PRINT
300 PRINT"                COPYRIGHT  1986"
310 PRINT
320 PRINT"               DR. BERTHOLD DAUM"
350 LOAD "P",8,1
500 POKE 53247,PEEK(44)
515 POKE 53246,SC:POKE53245,EC:POKE53244,TC
516 POKE 659,RC:POKE660,RI
520 POKE51347,CC:POKE51359,CC:POKE51348,BC:POKE51360,BC
550 FOR I=1TO125
560 READ X: POKE39424+I,X
570 NEXT
590 PRINT "
600 IF S$=""THENPOKE38400,0:GOTO700
605 S$=S$+CHR$(13)
610 FOR I=1TOLEN(S$)
620 POKE 38400+I,ASC(MID$(S$,I,1))
630 NEXT:POKE38400,255
640 PRINT CHR$(14);CHR$(8);
700 OPEN1,8,2,"SPRITES,S,R"
710 FORI=0TO2:GET#1,X$:GET#1,X$
720 FORJ=1TO64:GET#1,X$:IFX$=""THENX$=CHR$(0)
730 POKE 831+I*64+J,ASC(X$)
740 NEXTJ:NEXTI:GET#1,X$:GET#1,X$
750 FORJ=1TO64:GET#1,X$:IFX$=""THENX$=CHR$(0)
760 POKE 703+J,ASC(X$)
770 NEXTJ:CLOSE1
780 POKE2040,13:POKE2041,14:POKE2042,15:POKE2043,11
999 SYS49152
```

17.3.6 demo

```
90 ?-consult graphics.
91 ?-consult music.
100 topdemo :- repeat,screen(2,7),
110             vorstellung,
120             edinburgh,
130             primitives,
140             graf,
150             sounds,
160             debug,
170             bibliotheken,
180             doku,
190             fail.
200 vorstellung :- np,nl,tab(15),
210   write('PROLOG 64'),nl,tab(9),
215   write('von Dr. Berthold Daum'),nl,line,nl,
220   put("
230   write(' PROLOG 64 ist die erste PROLOG-'),nl,
240   write(' Implementierung fuer COMMODORE C64.'),nl,nl,put("
250   write(' Die folgende Demonstration zeigt'),nl,
260   write(' Ihnen die Faehigkeiten von PROLOG 64.'),nl,nl,line,nl,
270   write(' PROLOG 64 kostet 289 DM.'),nl,line,nl,
280   write(' Ihre Bestellung richten Sie bitte an:'),nl,nl,
290   write('       BRAINWARE'),nl,
300   write('       Kirchgasse 24 '),nl,
310   write('       6200 WIESBADEN'),nl,nl,
320   line,pause(3000).
330   line :- write('_____________________________________________').
400 edinburgh :- np,nl,nl,
430   write('PROLOG 64 erfuellt im wesentlichen'),nl,
440   write('den Edinburgh-Standard.'),nl,nl,line,nl,put("
450   write('Beispiele:'),nl,line,nl,
460   listing append,line,nl,
470   listing member,line,nl,
480   put("
500 primitives :- np,
510   write('Eingebaute Funktionen von PROLOG 64:'),nl,put("
```

```
520    writlist([append,arg,asserta,assertz,atom,atomic,
530      call,clause,clear,consult,debugging,delete,dir,display,draw,
540      efface,eof,error,fail,functor,get,get0,integer,is,islist,last,
550      listing,member,mod,move,nextto,n1,nodebug,nospy,not,notrace,op,pause,
560      pen,plot,poke,put,random,read,reconsult,repeat,retract,retractall]),
570    put("■■"),
580    writlist[rev,screen,see,seed,seeing,seen,skip,sound,spy,
590      subst,sublist,tab,tell,telling,told,trace,true,user,var,write,
600      =..,!,=,/=,==,/==,=<,<,>,>=,=:=,=/=,+,-,*,/]),
601      n1,put("?"),put("-"),
602      curpos(10),put(":"),put("-"),curpos(20),
605      put(","),curpos(30),put(";"),n1,put("."),
610  put("■"),pause(3000).
650  writlist([]) :- !.
660  writlist[X]) :- n1,write(X),!.
670  writlist[X,Y]) :- n1,write(X),curpos(10),write(Y),!.
680  writlist[X,Y,U]) :- n1,write(X),curpos(10),write(Y),curpos(20),write(U),!.
690  writlist[X,Y,U,V|L]) :-
691    n1,write(X),curpos(10),write(Y),
692    curpos(20),write(U),curpos(30),write(V),writlist(L).
700  curpos(A) :- n1,put("■"),not cp(A).
710  cp(1) :- put("■"),!,fail.
720  cp(A) :- put("■"),B is A-1,cp(B).
800  graf :- np,n1,put("■"),
805    tab(16),write('GRAFIK'),n1,n1,put("■"),line,n1,n1,
810    write('PROLOG 64 unterstuetzt die grafischen'),n1,
820    write('Faehigkeiten des C64 mittels Turtle-'),n1,
825    write('grafik und erlaubt die Verwendung von'),n1,
826    write('Sprites.'),n1,n1,n1,line,sprites,
830    seed(555),move(270,0),repeat,pen(0),
840    random(320,X),random(200,Y),draw(X,Y),pen(7),man,X=276,Y=105,!,
850    pause(2000),screen(2,7).
860  sprites :-
861    poke(0,120),poke(1,200),poke(39,3),
862    poke(2,160),poke(3,240),poke(40,5),
863    poke(4,200),poke(5,240),poke(41,2),
864    poke(6,240),poke(7,220),poke(42,4),
865    poke(21,15),repeat,pause(8),
866    poke(0,A),poke(0,A-1),
867    poke(3,B),poke(3,B-2),
868    poke(5,C),poke(5,C-1),
869    poke(6,D),poke(6,D-1),
870    poke(7,E),poke(7,E-1),
871    A=0,poke(21,0),!.
900  sounds :- np,n1,put("■"),
905    tab(17),write('KLANG'),put("■"),n1,n1,line,n1,n1,
910    write(' PROLOG 64 unterstuetzt die Klangmoeg-'),n1,
920    write(' licheiten des C64.'),n1,n1,n1,line,may(3).
1000 debug :- np,n1,write('PROLOG 64 hat leistungsfaehige Test-'),n1,
1010       write('hilfen, z.B. den Trace:'),n1,put("■"),line,trace,
1020    append[a,b,c],[d],X),notrace,n1,put("■"),line,put("■"),pause(3000).
1100 bibliotheken :- np,n1,write('Folgende Beispielbibliotheken werden'),n1,
1110      write('mitgeliefert:'),n1,n1,line,put("■"),n1,
1120      write(' - Mathematische Problemloesungen'),n1,n1,
1130      write(' - Mengenlehre'),n1,n1,
1140      write(' - Suchverfahren, Netzplantechnik'),n1,n1,
1150      write(' - Praedikatenlogik'),n1,n1,
1160      write(' - Grammatik natuerliche Sprache'),n1,n1,
1170      write(' - Grafikbaukasten'),n1,n1,
1180      write(' - Musikprogramme'),n1,n1,
1190      write(' - Minishell - Expertensystem'),n1,put("■"),line,pause(3000).
1200 doku :- np,n1,tab(13),write('Dokumentation'),n1,n1,
1210      line,put("■"),n1,
1220      write('Die Dokumentation von PROLOG 64 ist in'),n1,
1225      write('deutscher Sprache und umfasst:'),n1,n1,
1230      write(' - Bedienungsanleitung'),n1,n1,
1240      write(' - Funktionsbeschreibung'),n1,n1,
1250      write(' - Kurztutorium'),n1,n1,
1260      write(' - Auflistung der Bibliotheken'),n1,put("■"),line,pause(3000).
```

17.3.7 s to p

```
5 PRINT "■DIESES PROGRAMM WANDELT PROLOG-ASCII-"
6 PRINT "DATEIEN IN EDITIERBARE PROGRAMMDATEIEN"
7 PRINT "UM.■"
10 INPUT"■NAME ASCII-FILE";I$
20 INPUT"■NAME PROGRAM-FILE";B$
30 IFI$=B$THENPRINT:PRINT"BITTE KEINE GLEICHEN NAMEN !":GOTO10
100 LN=100:PRINT:PRINT
105 OPEN15,8,15,"I"
110 OPEN1,8,2,I$+",S,R"
115 OPEN4,8,4,"@:"+B$+",P,W"
```

```
116 PRINT#4,CHR$(0);
117 GOSUB200:I=0
120 GET#1,A$:I=I+1
125 IFPEEK(144) > 0 THEN 160
126 IFASC(A$) <> 13 THEN 130
127 IF I=1THEN 120
128 GOTO117
130 PRINT#4,A$;:IF I<30THEN120
140 IFA$=" "ORA$=","ORA$=")"ORA$="["ORA$="["ORA$="("THENGOSUB198:PRINT#4,"  ";
150 GOTO120
160 PRINT#4,CHR$(0);CHR$(0);CHR$(0);:CLOSE1:CLOSE4:CLOSE15:END
198 I=2
200 PRINT#4,CHR$(0);CHR$(1);CHR$(8);
210 PRINT#4,CHR$(LN-256*INT(LN/256));CHR$(INT(LN/256));:LN=LN+10
220 PRINT"□";:PRINTLN-10
250 RETURN
2000 INPUT#15,EM,EM$,ES,ET:IFEM=0THENRETURN
2010 PRINT EM,EM$,ES,ET:GOTO160
```

18 Diskettenaufkleber