HU-Prolog Anwenderdokumentation

©1989-1993 Christian Horn, Mirko Dziadzka, Matthias Horn

verantwortlich für diese Ausgabe:
Mirko Dziadzka
(dziadzka@informatik.hu-berlin.de)

Version vom März 1993
für HU-Prolog 2.025

Abtract

HU-Prolog ist ein effizientes, portables und erweiterbares Interpretersystem für Prolog, das in quelltextkompatiblen Versionen auf PC's , Workstations und Mainframes unter den Betriebssystemen MS-DOS, UNIX und VMS[1] verfügbar ist.

HU-Prolog ist eine Implementation des Clocksin-Mellish Standards und entspricht DEC10-Prolog. Die von HU-Prolog bereitgestellten Erweiterungen betreffen Ausdrucksmittel für die prozedurale und funktionale Programmierung.

HU-Prolog entstand in den Jahren 1987 – 1990 an der Humboldt-Universität zu Berlin. Die Originalfassung der Sprachbeschreibung entsprechend Release 1.54 erschien Ende 1989 in der Zeitschrift **edv-aspekte** (8.Jahrgang, Heft 4, Seite 2-31).

Die Quellen von HU-Prolog sind von den Autoren der Öffentlichkeit übergeben worden und können für nichtkommerzielle Zwecke beliebig kopiert und genutzt werden, vorausgesetzt die Hinweise auf Urheberschaft und geistiges Eigentum werden nicht aus dem Programm und der Dokumentation gestrichen.

Prolog ist eine lebende Sprache und HU-Prolog eine lebende Implementation. Der vorliegende Text beschreibt die Implementation 'HU-Prolog 2.025' vom März 1993.

[1] Aus Mangel an Testhardware wird die VMS Version zur Zeit nicht mehr unterstützt

Inhaltsverzeichnis

1 Einleitung **5**

2 Syntax **10**
2.1 Zeichenvorrat 10
2.2 Terme 11
2.2.1 Atome 11
2.2.2 Zahlen 12
2.2.3 Variablen 12
2.2.4 Funktorterme 12
2.2.5 Operatorterme 13
2.2.6 Curlyterme 13
2.2.7 Listenterme 14
2.3 Programme 15

3 Ein- und Ausgabe **17**
3.1 Das Streamkonzept in HU-Prolog 17
3.1.1 Standardstreams 17
3.1.2 assign/2 17
3.1.3 Windows 18
3.2 Streamoperationen 18
3.2.1 open/1 und close/1 19
3.2.2 see/1, seen/0 und seeing/1 19
3.2.3 tell/1, told/0 und telling/1 19
3.2.4 seek/2 19
3.2.5 Fehlerbehandlung 20
3.3 Eingabeoperationen 20
3.3.1 read/1 und read/2 20
3.3.2 get0/1 und get/1 21
3.3.3 eof/0 und eoln/0 21
3.3.4 unget/0 21
3.3.5 skip/1 22
3.3.6 ask/1 22
3.4 Ausgabeoperationen 22
3.4.1 write/1, writeq/1 und display/1 22
3.4.2 put/1 23
3.4.3 nl/0 und tab/1 23
3.5 Bildschirmsteuerung 23
3.5.1 cls/0 und gotoxy/2 24

4 Termbehandlung **25**
4.1 Termklassifikation . . . 25
4.2 Termanalyse und -synthese . . . 26
4.2.1 =../2 . . . 26
4.2.2 functor/3 . . . 26
4.2.3 arg/3 . . . 27
4.3 Analyse und Synthese von Atomen . . . 27
4.3.1 current_atom/1 . . . 27
4.3.2 name/2 . . . 27
4.4 Termvergleich . . . 28
4.4.1 Unifizierbarkeit (=/2 und \=/2) . . . 28
4.4.2 Identität (==/2 und \==/2) . . . 28
4.4.3 Lexikographische Ordnung (@</2 ,@=</2 ,@=/2 ,@>=/2 ,@>/2 ,@\=/2) . . . 29

5 Syntax und Operatordeklarationen **30**
5.1 Standardoperatoren . . . 30
5.2 op/3 . . . 31
5.3 current_op/3 . . . 33

6 Manipulation der Datenbasis **34**
6.1 consult/1 und reconsult/1 . . . 35
6.2 Einfügen von Klauseln . . . 36
6.2.1 asserta/1 . . . 36
6.2.2 assert(z)/1 . . . 36
6.3 Entfernen von Klauseln . . . 36
6.3.1 retract/1 . . . 36
6.3.2 retractall/1 . . . 37
6.3.3 abolish/1 und abolish/2 . . . 37
6.4 Abfragen von Klauseln . . . 37
6.4.1 clause/2 . . . 38
6.4.2 current_predicate/1 . . . 38
6.4.3 listing/1 und listing/0 . . . 38

7 Arithmetik und funktionale Programmierung **39**
7.1 Standardfunktionen und -operationen . . . 40
7.2 is/2 . . . 40
7.3 Arithmetische Vergleiche (</2,=</2,=:=/2,>=/2,>/2,=\=/2) . . . 41
7.4 :=/2 . . . 42
7.5 Elementare symbolische Funktionen . . . 44
7.6 Nutzerdefinierte Funktionen . . . 44

8 Ablaufsteuerung **46**
8.1 Das Box-Modell . . . 46
8.2 ,/2 . . . 48
8.3 ;/2 . . . 48
8.4 ->/2 . . . 49
8.5 !/0 . . . 50
8.6 true/0 und fail/0 . . . 50
8.7 repeat/0 . . . 51
8.8 call/1 . . . 51
8.9 not/1 . . . 52

9 Globale Steuerung **53**
9.1 login/0 und logout/0 54
9.2 toplevel/0 und prompt/0 54
9.3 exit/1, halt/0 und end/0 55
9.4 abort/0 und restart/0 55
9.5 interrupt/0 55
9.6 error/2 55
9.7 unknown/1 57
9.8 ancestors/1 58
9.9 sys/1 58
9.10 dict/1 und sdict/1 58

10 Programmierumgebung **59**
10.1 Das Modul-Konzept von HU-Prolog 59
10.1.1 private/1 59
10.1.2 hide/1 59
10.1.3 ensure/3 60
10.2 Die Schnittstelle zur Systemumgebung 60
10.2.1 date/3, time/3 und weekday/1 60
10.2.2 timer/1 60
10.2.3 getenv/2 und putenv/2 60
10.2.4 system/1 60
10.2.5 argc/1 und argv/2 61
10.2.6 stats/0 und version/0 61
10.3 Testunterstützung 62
10.3.1 trace/0, trace/1 und notrace/0 62
10.3.2 spy/1 und nospy/1 64
10.4 Online Hilfssystem 64
10.5 Systemflags und Optionen 64

Kapitel 1

Einleitung

Prolog ist eine der faszinierenden Neuschöpfungen der Programmierkunst, eine Programmiersprache, die in schier unerschöpflicher Weise immer wieder verblüffend einfache und klare Lösungsvarianten eröffnet. Obwohl zur selben Zeit wie PASCAL entstanden, erscheint sie für den Praktiker heute jedoch immer noch unnahbar. Eine Ursache dafür liegt sicher darin, daß sie gleich mehrere qualitativ neue Elemente in die Programmierung einbringt: symbolische Manipulation, Backtracking, Rekursion und relationale Programmierung. Elemente, die sich zwar schon auf verschiedenen Gebieten als alternative Problemlösungsansätze bewährt haben, aber zuviele auf einmal, wenn man den im allgemeinen langsamen, evolutionären Akzeptanzprozeß von Programmiersprachen betrachtet.

Erst Anfang der achtziger Jahre begann international eine verstärkte Hinwendung zu Prolog, ausgelöst durch den Anwendungsdruck nach 'intelligenten' Problemlösungen, die sich mit klassischen Sprachen nur schwer realisieren ließen, durch eine Demystifizierung zumindest einiger Methoden der Künstlichen Intelligenz, die nun begannen, adäquater Bestandteil der normalen Programmierpraxis zu werden, und durch die Bereitstellung schneller und leistungsfähiger Implementierungen. Die aufsehenerregenden Ankündigungen der japanischen fünften Rechnergeneration taten ein Übriges: Nach diesen Plänen sollte Prolog bzw. eine geeignete Erweiterung als Kernsprache der neuen Rechnergeneration fungieren. Prolog bekam damit eine Rolle zugewiesen, die heute vielleicht im Hinblick auf ihre Universalität und Maschinennähe nur der Sprache C in UNIX-Systemen zukommt. Dabei zeigte sich bald ein Problem: Die einfachen, auf sehr hohem Abstraktionsniveau stehenden sprachlichen Mittel von Prolog reduzieren zwar deutlich den Programmieraufwand, verlagern aber die letztendlich immer notwendigen Implementationsentscheidungen auf die Ebene der Sprachimplementierung. Jede Implementierung von Prolog wird daher einen gewissen Programmierstil präjudizieren. Die abstrakte, von rein logischen Gesichtspunkten ausgehende Propagierung eines Prolog-Stils führt genau wie jedes Außerachtlassen der Eigenschaften eines Prolog-Systems zu ineffizienten, unnötige Resourcen beanspruchenden Programmen. Das führt zu ernsten Realisierungs- und Portierungspoblemen für Anwenderlösungen auf Prolog-Quelltextniveau insbesondere dann, wenn die Problemlösung die gegebenen technischen Möglichkeiten voll ausreizen muß.

Etwa 1987 begannen an der **H**umboldt-**U**niversität zu Berlin die Arbeiten an einer portablen Prolog-Implementierung. Ziel war die Bereitstellung eines bis auf konfigurierbare Leistungsparameter identischen Prolog-Systems für ein breites Spektrum von 16- und 32-Bit-Rechnern, das den von Clocksin und Mellish beschriebenen Sprachumfang vollständig überdeckt und darüber hinaus offen für Spracherweiterungen ist. Am Anfang stand dabei die Untersuchung verschiedener experimenteller Prolog-Systeme. Die Implementierung erfolgte schließlich in C und konnte sich auf Erfahrungen in der Gestaltung portabler Systemlösungen stützen. Dadurch ist das HU-Prolog-System heute nicht nur unter UNIX sondern auch unter MS-DOS und VMS; sowohl auf IBM-PC-kompatiblen (von 8086/8088 bis 80486)-Systemen, auf P8000 als auch auf VAX verfügbar[1].

[1] Die VMS und die P8000 Implementationen werden aus Mangel an vorhandener Hardware im Moment nicht mehr unterstützt

Bezüglich der Spracherweiterungen verfolgten wir unsprünglich die Strategie, dem Systemanwender Werkzeuge zur Verfügung zu stellen, mit denen er das Prolog-System um die für seine Anwendung relevanten Funktionen (wie z.B. 2D/3D- Grafik-Funktionen, Simulationsverfahren oder Datenbankschnittstellen) erweitern kann. Diese Vorgehensweise halten wir nach wie vor für relevant, doch erfordert sie beim Anwender ein tiefes Eindringen in die internen Strukturen des Prolog-Systems, was letztlich nur für wenige in Frage kommt. Wir haben uns daher entschlossen, das HU-Prolog-System standardmäßig um die Funktionen zu erweitern, die nach unseren Erfahrungen für die praktische Prolog-Programmierung besonders zweckmäßig sind, von international gebräuchlichen Prolog-Systemen zur Verfügung gestellt werden und/oder wesentlichen Entwicklungslinien bezüglich der Spracherweiterungen entsprechen. Dazu gehören neben einigen ergänzenden built-in-Prädikaten für die Ein- und Ausgabe und einem elementaren Modulkonzept in erster Linie das Konzept der globalen Zustandsvariablen sowie der verallgemeinerten funktionalen Auswertung von Termen. Grundsätzlich sind wir dabei so vorgegangen, daß die zusätzlichen Prädikate entweder im Sinne der logischen Geschlossenheit des Systems zwangsweise notwendig waren, oder daß sich diese Prädikate in Standard-Prolog explizit, wenn auch weniger effizient, definieren lassen.

Der vorliegende Beitrag gibt eine zusammenfassende Darstellung von HU-Prolog (ab Version 2.025 März 93) im Sinne einer Sprachbeschreibung. Es ist keine Einführung in die Prolog-Programmierung. Der interessierte Leser sei dafür auf die Vielzahl inzwischen auch in deutscher Sprache vorliegender Lehrtexte verwiesen. Der Beitrag soll dem zukünftigen, praktizierenden Prolog-Programmierer eine Hilfe sein. Wir haben uns daher ganz bewußt auf die Probleme konzentriert, die etwa 90% aller Prolog-Programmtexte ausmachen, aber kaum in Lehrbüchern behandelt werden: das sind Fragen der klassischen, prozeduralen und funktionalen Programmorganisation. Die „Glanzseiten“ von Prolog spielen kaum eine Rolle, darüber ist an anderer Stelle schon genug geschrieben worden. Wer sich überhaupt daran macht, Prolog zu benutzen, muß schon von den Vorteilen der Sprache überzeugt sein. Wir haben versucht, die Schattenseiten etwas aufzuhellen.

Ein Beispiel

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

```
% prolog
HU-Prolog (Public Domain Version)  Release 2.025 (last change: 16.03.93 14:53)
Author(s): 87-89  Christian Horn, Mirko Dziadzka, Matthias Horn
           90-93  Mirko Dziadzka (dziadzka@informatik.hu-berlin.de)

Type 'help.' for help

Ready

?-
```

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

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

Mit der erneuten Ausgabe des Prolog-Prompts zeigt der Interpreter an, daß er auf den nächsten Befehl wartet. Das Prolog-System bleibt ständig in dieser Interpreterschleife. Das Verlassen des Prolog-Interpreters ist nur durch die Abarbeitung eines speziellen Befehls (z.B. 'end') möglich oder durch Eingabe eines End-of-File-Zeichens.

Ein Prolog-Kommando, das vom Top-Level-Interpreter verarbeitet wird, kann aus mehreren, durch Komma getrennten Aufrufen bestehen, die nacheinander abgearbeitet werden. Das Kommando kann sich über mehrere Zeilen erstrecken, dabei darf aber nur die letzte Zeile mit Punkt und Enter abgeschlossen werden. Wenn das System aus irgendwelchen unersichtlichen Gründen mit der Abarbeitung eines Kommandos nicht beginnt, so ist eine der häufigsten Ursachen der vergessene Punkt am Ende der Eingabe. Das System nimmt in solcher Situation stets an, daß die Eingabe in der nächsten Zeile weiter geht. Das Einfügen von Leerzeichen oder Zeilenwechseln zur übersichtlicheren Gestaltung eines Prolog-Textes hat keinen Einfluß auf dessen Abarbeitung. Enthält ein Prolog-Kommando Variablen, so wird nach erfolgreicher Abarbeitung des Kommandos die Lösung des gestellten Problems in Form der Variablenbelegung ausgegeben. Das System wartet dann auf eine Nutzereingabe. Einfaches Drücken der Enter-Taste beendet den Abarbeitungsprozeß, das Prolog-System hat eine Lösung gefunden und antwortet mit 'yes'.

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

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

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

In Prolog wurde das Prozedurkonzept entsprechend verallgemeinert: eine Prozedur ist erfolgreich (und liefert dabei eine oder mehrere Lösungen) oder schlägt fehl. Beim erfolgreichen Verlassen einer Prozedur wird zunächst nur eine Lösung in Form von Variablenbindungen an die Umgebung weitergegeben. Das Fehlschlagen einer der nachfolgend aufgerufenen Prozeduren initiiert die Suche nach weiteren Lösungen. Dazu wird eine Rückverfolgung der Berechnungsspur (das sogenannte Backtracking) angestoßen. Der gesamte Berechnungsprozess wird auf den letzten Verzweigungspunkt zurückgesetzt, bei dem (noch) alternative Lösungen generiert werden können. Die Berechnung wird an diesem Punkt mit einer neuen Teillösung wieder aufgenommen und zu Ende geführt. Wir wollen nun versuchen, eine Wertetabelle für das logische Und zu erstellen. Ein Teilproblem ist dabei das Erzeugen der Wahrheitswertebelegungen für aussagenlogische Variablen. Hier benötigt man eine einstellige Prozedur *boole/1*, die bei ihrem Aufruf nacheinander zwei Lösungen liefert, 0 und 1. Eine solche Prozedur wird in Prolog am einfachsten durch zwei Fakten beschrieben:

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

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

```
?- [user]. <ET>
user> boole(0). <ET>
user> boole(1). <ET>
user> end. <ET>
yes
?-
```

Die Wahrheitswertebelegungen für X, Y bzw. X and Y erhalten wir nun nacheinander als Lösungen von:

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

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

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

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

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

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

```
?- zeile, fail. <ET>
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
no
?-
```

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

```
?- [user]. <ET>
user> tabelle:-zeile,fail. <ET>
user> tabelle. <ET>
user> end. <ET>
yes
?-
```

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

```
?- tabelle. <ET>
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
yes
?-
```

Kapitel 2

Syntax

2.1 Zeichenvorrat

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

```
Zeichen  = Buchstabe | Ziffer |
           Sonderzeichen | Einzelzeichen |
           Listenseparator | Klammer |
           Atombegrenzer | Stringbegrenzer |
           Kommentarzeichen | Leerzeichen .
Buchstabe = Grossbuchstabe | Kleinbuchstabe .
Grossbuchstabe = "A’ | ‘B’ | ... | ‘Z’ .
Kleinbuchstabe = "a’ | ‘b’ | ... | ‘z’ | ’$’.
Ziffer =         ’0’ | ’1’ | ’2’ | ... | ’9’ .
Sonderzeichen =  ’+’ | ’-’ | ’*’ | ’/’ | ’" | ’^’ |
                 ’<’ | ’>’ | ’=’ | ’‘’ | ’~’ | ’:’ |
                 ’.’ | ’?’ | ’@’ | ’#’ | ’&’ .
Einzelzeichen =  ’,’ | ’;’ | ’!’ .
Listenseparator = ’|’ .
Klammer =        ’(’ | ’)’ | ’[’ | ’]’ | ’{’ | ’}’ .
Atombegrenzer =  ’’’ .
Stringbegrenzer = ’"’ .
Kommentarzeichen = ’%’ .
Leerzeichen =     ’ ’ .
darstellbares_Zeichen = Zeichen | spezielles_Zeichen |
                        direkt_kodiertes_Zeichen .
spezielles Zeichen = ’" | ’\’’ | ’"’ | ’\n’ | ’\a’ |
                     ’\r’ | ’\b’ | ’\t’ | ’\f’ | ’\v’ .
direkt_kodiertes_Zeichen = ’\’ Oktalcode [ Oktalcode [ Oktalcode ]].
Oktalcode =       ’0’ | ... | ’7’ .
```

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

die von '/*' bis zum nächsten '*/' reichen. Wenn die erste Zeile einer Datei mit '#!' beginnt, so wird diese erste Zeile auch als Kommentar interpretiert. Innerhalb gequoteter Atome und Strings finden die von C bekannten Konventionen zur Zeichendarstellung Anwendung. Bei der expliziten Notation des Zeichenkodes 0 ist Vorsicht zu üben, da intern alle Zeichenketten entsprechend den C-Konventionen behandelt werden (wo 0 das Ende einer Zeichenkette markiert).

Symbol	Code	Bedeutung
`\177`	—	oktale Darstellung beliebiger Zeichencodes zwischen 0 und 255
`\\`	0x5c	ermöglichen die direkte Notation von
`\'`	0x22	Backslash sowie der jeweiligen Atom-
`"`	0x27	bzw. String-Quotes
`\a`	0x07	Alarmzeichen (Bell)
`\b`	0x08	Löschzeichen (Backspace)
`\t`	0x09	Tabulatorzeichen
`\n`	0x0a	Zeilenendezeichen entsprechend UNIX-Konventionen, wird im DOS in der Regel durch eine Zeichenkombination mit `\r` dargestellt
`\v`	0x0b	vertikaler Tabulator
`\f`	0x0c	Filemarke
`\r`	0x0d	Wagenrücklaufzeichen (Carriage Return)

2.2 Terme

```
Term = einfacher_Term | strukturierter_Term .
einfacher_Term = Atom | Zahl | Variable .
strukturierter_Term = Funktorterm |
                      Operatorterm |
                      Curlyterm |
                      Indexterm |
                      Listenterm .
```

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

2.2.1 Atome

```
Atom =           einfaches_Atom | gequotetes_Atom .
einfaches_Atom = Bezeichner |
                 Spezialatom |
                 Einzelzeichen .
Bezeichner =     Kleinbuchstabe { Buchstabe | Ziffer | '_'  } .
Spezialatom =    Sonderzeichen { Sonderzeichen } .
gequotetes_Atom = ''' { darstellbares_Zeichen } ''' .
```

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

2.2.2 Zahlen

```
Integerzahl = Ziffernfolge .
Ziffernfolge = Ziffer { Ziffer } .
Realzahl = Ziffernfolge '.' Ziffernfolge [ Exponent ] |
           Ziffernfolge Exponent .
Exponent = ( 'E' | 'e' ) [ '+' | '-' ] Ziffernfolge .
```

Zahlen sind die elementaren numerischen Daten von Prolog. HU-Prolog unterscheidet syntaktisch zwischen Integerzahlen und Realzahlen. Integerzahlen sind ganze Zahlen zwischen -2^{31} und $2^{31}-1$ (entsprechend dem Zahlentyp long des zugrundeliegenden C-Dialekts). Realzahlen liegen zwischen -0.13e309 und 0.13e309 (entsprechend dem Zahlentyp double des zugrundeliegenden C-Dialekts). Integerzahlen entsprechend dem Typ int[1] des zugrundeliegenden C-Dialekts werden direkt abgespeichert. Arithmetische Operationen über diesem Zahlenbereich sind damit auch die schnellsten und überaus speichereffektiv. Große Integer-, sowie Realzahlen erfordern intern eine Sonderbehandlung. Arithmetische Operationen über diesen Zahlenbereichen sind deshalb langsamer und speicheraufwendiger. Es sind Versionen von HU-Prolog ohne real-Arithmetik, mit unsigned-Arithmetik bzw. mit arbitrary-precision-Arithmetik im Umlauf, diese Versionen lassen sich jedoch relativ einfach durch Testaufrufe unterscheiden.

2.2.3 Variablen

```
Variable = ( Grossbuchstabe | '_' ) { Buchstabe | Ziffer | '_' } .
```

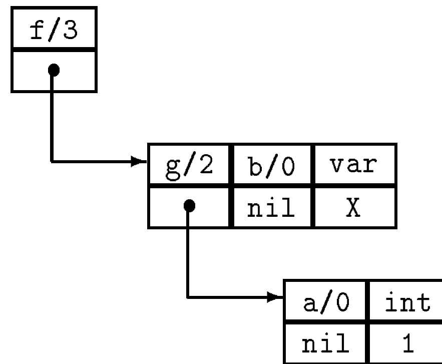
Prolog ist eine typfreie Sprache. Alle Daten haben die Struktur von Termen, die jedoch unterschiedlich groß ausfallen können. Die Variablen nehmen in Prolog daher nur Verweise auf Terme auf. Variablenbindungen entstehen bei der Unifikation. Das ist der elementare Prozeß der Angleichung zweier Term(muster)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

2.2.4 Funktorterme

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

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

[1] Auf 32 Bit Systemen gilt in der Regel long = int

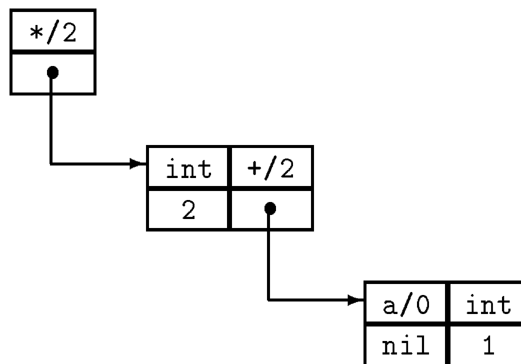


Interne Struktur des Funktorterms `f(g(a,1),b,X)`

2.2.5 Operatorterme

```
Operatorterm = Operator Term |
               Term Operator Term |
               Term Operator |
               '(' Term ')' .
```

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



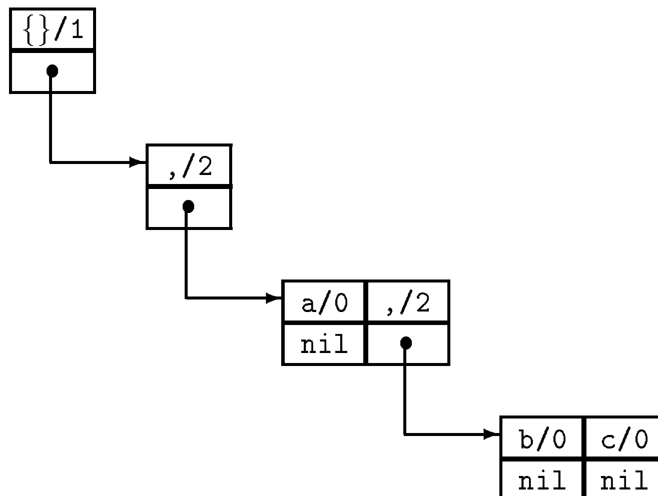
Interne Struktur des Operatorterms `2*(a+1)`

2.2.6 Curlyterme

```
Curlyterm = '{' '}' |
            '{' Term '}' .
```

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

entspricht, dargestellt. Die Kommata innerhalb eines Curlyterms werden als zweistellige Operatoren interpretiert.

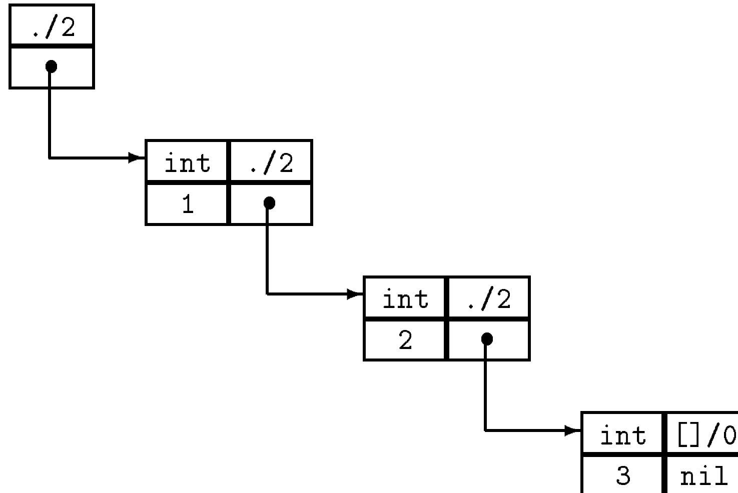


Interne Struktur des Curlyterms {a,b,c}

2.2.7 Listenterme

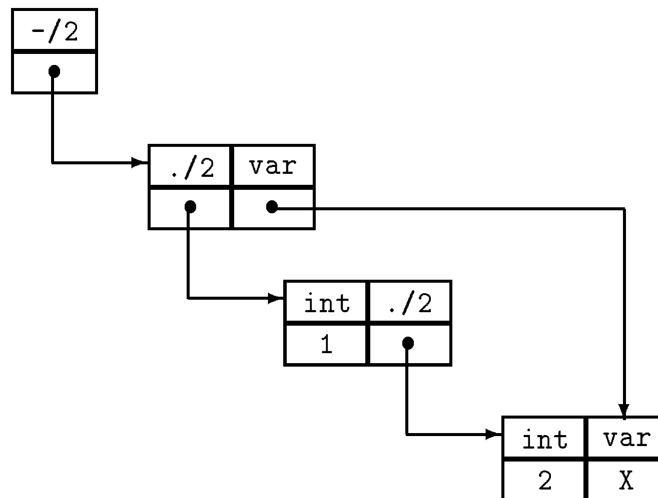
```
Listenterm = leere_Liste |
             Standardliste |
             offene_Liste |
             String .
leere_Liste = '[' ']' .
Standardliste = '[' Termfolge ']' .
offene_Liste = '[' Termfolge '|' Listenrest ']' .
Listenrest = Term .
Termfolge = Subterm { ',' Subterm } .
String = '"'  { darstellbares_Zeichen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

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



Interne Struktur der Liste `[1,2,3]`

Eine offene Liste ist eine Liste, die aus einem Term und einer freien Variable bzw. einer offenen Liste durch Anwendung des Funktors `./2` entsteht. Offene Liste sind besonderes interessant, weil sie die direkte Erweiterung einer Liste an ihrem Ende ermöglichen. Die häufigste Anwendung offener Listen findet man im Zusammenhang mit Differenzlisten:



Interne Struktur der Differenzliste `[1,2|X]-X`

2.3 Programme

```
Program = { Programmeinheit '.' }
Programmeinheit = Klausel |
                  Frage .
Klausel = Fakt |
          Regel .
Regel = Regelkopf ':-' Disjunktion .
Fakt = Regelkopf .
Regelkopf = Atom |
```

```
            strukturierter_Term .
Disjunktion = Konjunktion { ';' Konjunktion } .
Konjunktion = Aufruf { ',' Aufruf } .
Aufruf = Atom |
         strukturierter_Term |
         Variable .
Frage = '?-'  Disjunktion |
        ':-'  Disjunktion .
```

Ein Prolog-Programm besteht aus einem oder mehreren Quellfiles, die beim Systemstart bzw. über `consult/1` geladen werden. Jedes Prolog-Quellfile besteht aus einer Folge von Prolog-Termen, von denen jeder mit '.' und Leerzeichen bzw. Zeilenende abgeschlossen wird. Prolog-Quellfiles werden sequentiell mittels `read/1` gelesen. Fragen werden im Moment des Einlesens abgearbeitet. Für die Abarbeitung einer Frage stehen also auch nur die bis dahin schon geladenen Prozeduren zur Verfügung. Sie können also sowohl zur Initialisierung von Datenstrukturen als auch zur Steuerung des weiteren Einlesevorgangs benutzt werden. Nach dem Einlesen eines Programms (bzw. unmittelbar nach dem Programmstart, wenn man mit einem kompilierten Prolog-Programm arbeitet) besteht die Datenbasis aus genau den bisher eingelesenen oder während des Lesevorgangs generierten Prozeduren. Jede Prozedur besteht aus einer strikt geordneten Liste von Klauseln. Mit dem Systemstart geht die Systemsteuerung an den Prolog-Supervisor, der in Abhängigkeit von gesetzten (Kommandozeilen-)Optionen, dem aktuellen Start-up-File[2] bzw. vom Vorhandensein oder Nichtvorhandensein gewisser Prozeduren (wie z.B. `login/0`, `logout/0`, `prompt/0`, `toplevel/0`, `interrupt/0` oder `error/2`) im interaktiven Modus oder im Batchmodus läuft.

[2] unter U*IX: `.prologrc` bzw. `$HOME/.prolog`, unter DOS: `prolog.rc` bzw. `$HOME/prolog.rc`

Kapitel 3

Ein- und Ausgabe

HU-Prolog lehnt sich in Syntax und Semantik wesentlich an Edinburgh-Prolog an, das vor allem durch das Buch von Clocksin und Mellish zum de-facto-Standard geworden ist. Das Ein- und Ausgabekonzept von Prolog wurde dabei wesentlich durch die damalige Implementation auf DEC10 beeinflußt und wirkt dadurch stellenweise antiquiert. In HU-Prolog wurde dieses Konzept mit Blick auf modernere Betriebssysteme und die Mindestanforderungen an ein zeichenorientiertes Bildschirminterface (CUI) leicht modernisiert, wobei eine strikte Aufwärtskompatibilität gewahrt blieb.

3.1 Das Streamkonzept in HU-Prolog

HU-Prolog kennt zu jedem Zeitpunkt genau einen aktuellen Eingabestream und einen aktuellen Ausgabestream. Streams als logische Objekte können entweder *Files*, den Standard-Ein- bzw. Ausgabestreams des Interpreters oder *Windows* zugeordnet sein. Ein- bzw. Ausgabeoperationen beziehen sich immer auf den jeweils aktuellen Ein- bzw. Ausgabestream. Die Zuordnung der Streams zu den physischen Objekten (Files oder Windows) wird durch das Prädikat `assign/2` hergestellt. Die Umschaltung zwischen den verschiedenen Streams erfolgt mit Hilfe von Streamoperationen.

3.1.1 Standardstreams

Um eine sinvolle Arbeit mit dem Prologsystem zu ermöglichen, sind eine Reihe von Streams vordefiniert, welche ohne weiteres benutzt werden können. Das sind die Streams `stdin`, `stdout`, `stderr`, `stdwarn`, `stdtrace` und `stdhelp` . Streams werden durch Namen (Atome) beschrieben. `stdin/0`, `stdout/0` bzw. `stderr/0` bezeichnen die Standardeingabe, -ausgabe bzw. -fehlerausgabe des Prologsystems entsprechend den Betriebssystemkonventionen. Über den Stream `stdtrace/0` erfolgt die Kommunikationen beim Debugging, `stdwarn/0` ist fu"r die Warnungen zuständig und über `stdhelp/0` wird das Online-Help bedient.

3.1.2 assign/2

Die Zuordnung von Streams zu physischen Objekten ist in HU-Prolog durch das Prädikat `assign/2` möglich. Streamnamen wirken dabei wie logische Filenamen.

`asign(Streamname,Objekt)` ordnet dem Stream `Streamname` ein physisches oder logisches Objekt zu. Dies kann entweder ein File, ein Window oder wiederum ein Stream sein. `assign/2` speichert die Zuordnung lediglich ab. Erst bei tatsächlicher Bezugnahme auf einen Stream, d.h. im Moment des ein- oder ausgabeseitigen Umschaltens auf diesen Stream, wird der logische Streamname soweit wie möglich dereferenziert. Endet die Kette logischer Referenzen bei einem physischen Objekt, so wird dieses genommen. Endet die Kette auf einem logischen (Nicht-Standard-) Streamnamen, so wird dieser Streamname (entsprechend der Edinburgh-Prolog-Konvention) als Filename interpretiert.

Streamnamen wirken wie logische Filenamen und erlauben die Einhaltung gewisser Schnittstellen zwischen der Programmlogik und der Einbindung des Programms in die Betriebssystemumgebung. Wenn für einen Stream keine Vereinbarung mit `assign/2` getroffen wurde, verweist der Stream auf sich selbst und wird als File interpretiert. Eine Vereinbarung mit `assign/2` wird aufgehoben, indem man mit `assign(Stream,Stream)` einen Stream auf sich selbst verweisen läßt.

Beispiel 1 (`assign/2`)

```
:-assign(logfile,'/tmp/prolog.log').
:-assign(debugfile,logfile).
:-assign(testmessages,logfile).
```

Nach Abarbeitung dieser Zuweisungen beziehen sich alle Lese- und Schreibanweisungen auf die Streams `logfile`, `debugfile`, `testmessages` *auf ein und dasselbe File:* `/tmp/prolog.log`.

3.1.3 Windows

Windows werden in HU-Prolog als ein Term der Form

`window(XPos,YPos,XLen,YLen,Name,AtList)`

beschrieben. Dabei stehen

- `XPos` für die horizontale bzw. `YPos` für die vertikale, absolute Bildschirmposition des ersten *nutzbaren* Zeichens in der linken oberen Ecke des Windows;
- `XLen` und `YLen` für die horizontale bzw. vertikale Ausdehnung des nutzbaren (Innen-) Bereichs des Windows, wenn rund um das Window noch Platz ist, wird ein Rahmen gezeichnet; so daß die von einem Window verdeckte Fläche in jeder Richtung ein Zeichen größer ist;
- `Name` für ein Atom, den Titel des Windows, das in der Mitte des oberen Rahmen dargestellt wird; und
- `AtList` ist eine Liste, in der (hardwareabhängige) Attribute des Windows angegeben werden können.

Mit dem Prädikate `window/0` kann dynamisch getestet werden, ob in der aktuell laufenden Prolog-Konfiguration Windows verfügbar sind.

Beispiel 2 (`.prologrc`)

Das folgende ist ein Beispiel für ein Start-up-File mit dem Windows für die interaktive Arbeit definiert werden.

```
:- window,
     assign(stdtrace,window(1,17,80,7,' Trace ',[])),
     assign(stderr ,window(10,10,60,10,' Error ',[paging])),
     assign(stdwin, window(1,2,80,14,' HU-Prolog ',[])),
     assign(stdwarn, window(2,18,78,6,' Warnings ',[paging])),
     assign(stdhelp, window(2,2,78,22,' HELP ',[])),
     assign(stdin, stdwin),
     assign(stdout,stdwin)
   ;true.
```

3.2 Streamoperationen

Die Streamoperationen in HU-Prolog dienen dem Festlegen des aktuellen Ein- oder Ausgabestream. Damit werden also die Streams definiert, auf der die Standardmässigen Ein- Ausgabeoperationen stattfinden.

3.2.1 open/1 und close/1

Das Prädikat `open/1` eröffnet einen Stream zum Lesen und zum Schreiben. Wenn dieses nicht möglich ist, so entsteht ein Fehler. Ist dem Stream ein Window zugeordnet, so erscheint dieses auf dem Bildschirm.

Das Prädikat `close/1` schliesst einen Stream physisch ab. Wenn der Stream aktueller Eingabestream oder Ausgabestream ist, wird implizit ein `seen/0` bzw. ein `told/0` abgearbeitet. Ein Fehler entsteht, wenn `close/1` auf einen nicht eröffneten Stream angewendet wird. Wenn dem Stream ein Window zugeordnet ist, so wird dieses vom Bildschirm entfernt, und eventuell verdeckte Windows werden wieder sichtbar.

3.2.2 see/1, seen/0 und seeing/1

Das Prädikat `see/1` dient dem temporären Deklarieren eines Streams als aktueller Eingabestream. Ist der Stream noch nicht eröffnet, so wird er implizit zum Lesen eröffnet. Ein Fehler tritt auf, wenn das Eröffnen zum Lesen nicht möglich oder der Stream nur zum Schreiben eröffnet ist. Der Aufruf von see(user) ist äquivalent zu see(stdin). Wenn dem Stream ein Window zugeordnet ist, so wird es als oberstes Window auf dem Bildschirm plaziert.

Das Prädikat `seen/0` schliesst den Stream, welcher aktueller Eingabestream ist, physisch ab und definiert stdin als aktuellen Eingabestream. Wenn dem Stream ein Window zugeordnet ist, so wird dieses vom Bildschirm entfernt, und eventuell verdeckte Windows werden wieder sichtbar.

Das Prädikat `seeing/1` unifiziert sein Argument mit dem Namen des dem aktuellen Eingabestream, das heißt dem Atom, das beim letzten Aufruf von `see/1` als Parameter angegeben wurde. Wenn `see/1` mit dem Argument user aufgerufen wurde, unifiziert `seeing/1` sein Argument mit stdin.

3.2.3 tell/1, told/0 und telling/1

Das Prädikat `tell/1` dient dem temporären Deklarieren eines Streams als aktueller Ausgabestream. Ist der Stream noch nicht eröffnet, so wird er implizit zum Schreiben eröffnet. Ein Fehler tritt auf, wenn das Eröffnen zum Schreiben nicht möglich oder der Stream nur zum Lesen eröffnet ist. Der Aufruf von tell(user) ist äquivalent zu tell(stdout). Wenn dem Stream ein Window zugeordnet ist, so wird es als oberstes Window auf dem Bildschirm plaziert.

Das Prädikat `told/0` schließt den Stream, welcher aktueller Ausgabestream ist, physisch ab und definiert stdout als aktuellen Ausgabestream. Wenn dem Stream ein Window zugeordnet ist, so wird dieses vom Bildschirm entfernt, und eventuell verdeckte Windows werden wieder sichtbar.

Das Prädikat `telling/1` unifiziert sein Argument mit dem Namen des dem aktuellen Ausgabestream, das heißt dem Atom, das beim letzten Aufruf von `tell/1` als Parameter angegeben wurde. Wenn `tell/1` mit dem Argument user aufgerufen wurde, unifiziert `telling/1` sein Argument mit stdout.

3.2.4 seek/2

Das Prädikat `seek/2` dient dem Abfragen und Setzen des Schreib/Lese-Zeigers im File. Zu jedem eröffneten physischen File gehört ein Zeiger, der die aktuelle Position des Files beschreibt. Dieser Zeiger dient, solange vom File gelesen wird, als Lesezeiger. Er beschreibt dann die Position, von der aus die nächste Leseoperation beginnt, und wird von dieser implizit weitergesetzt. Wird auf das File geschrieben, so dient die gleiche Position als Schreibzeiger. Er beschreibt die Position auf die das nächste Zeichen geschrieben werden soll und wird bei der Ausgabe weitergestellt. Die Fileposition bleibt dabei an das File gebunden und wird von Veränderungen in der Zuordnung eines Files zu Streams nicht betroffen.

Das erste Argument von `seek/2` muß der Name eines eröffneten Streams sein, der einem physischen File zugeordnet ist. Das zweite Argument kann entweder eine Variable, eine ganze Zahl oder das Atom `end/0` sein. Wenn es eine Variable ist, so wird diese mit der aktuellen Position des Schreib/Lese- Zeigers (als Byte-Offset zum Fileanfang) unifiziert. Ist es eine nichtnegative Zahl, so wird der Schreib/Lese-Zeiger auf diese Position relativ zum Fileanfang gesetzt. Ist das zweite Argument das Atom `end/0` wird auf das Fileende positioniert, eine negative Zahl wird als Byteoffset zum Fileende interpretiert.

Wenn der angegebene Stream nicht eröffnet ist, oder nicht einem File zugeordnet ist, wird ein Fehler ausgelöst.

3.2.5 Fehlerbehandlung

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

3.3 Eingabeoperationen

Die im folgenden beschriebenen Eingabeprädikate beziehen sich alle auf den aktuellen Eingabestream. Jedem dieser Prädikate kann man den Präfix tty voranstellen. Sie beziehen sich dann auf den Standardeingabestream des Interpreters, unabhängig davon, ob dieser gerade der aktuelle Eingabestream ist oder nicht.

3.3.1 read/1 und read/2

Das Prädikat `read/1` liest einen Term vom aktuellen Eingabestream und unifiziert diesen anschließend mit seinem Argument. Der Term muß mit einem Punkt abgeschlossen sein, dem ein Leerzeichen, Tabulator oder ein Zeilenendezeichen folgt. Wenn der Term syntaktisch nicht korrekt ist, wird eine Fehlermeldung generiert, die die fehlerhafte Position genau markiert und die laufende Abarbeitung abgebrochen. Wenn das Streamende erreicht ist, liefert `read/1` das Atom `end/0` zurück. Wird Prolog als Stapelverarbeitungssystem genutzt, so hat das Fileende also die gleiche Funktion wie die Eingabe von '`end.`' im interaktiven Modus. An Stelle von '`end.`' kann man im interaktiven Modus auch versuchen, das Fileendekennzeichen des jeweiligen Betriebssystems direkt einzugeben.

Das Prädikat `read/2` arbeitet analog zu `read/1`, unifiziert aber zusätzlich das zweite Argument mit einer Liste, aus der die Variablennamen hervorgehen. Die einzelnen Elemente der Liste haben die Form:

Variable = Variablenname

Mit Hilfe eines einfach zu implementierenden Prädikates `write/2` ist es dadurch möglich, Terme mit ihren originalen Variablennamen auszugeben.

Beispiel 3 (`write/2`)

```
:- private([unify]).

unify([]).
unify([Head|Tail]) :-
   (Head ; true ),
   !,
   unify(Tail).

write(Term,Varlist) :-
   unify(Varlist),
   write(Term),
   fail;true.

:- hide([unify]).

?-write('>>'),read(Term,Varlist),call(Term),write(Term,Varlist),nl.
>>a(X,Y) =.. Z.
a(X, Y) =.. [a, X, Y]

Term = (a(_1, _2) =.. [a, _1, _2])
Varlist = [X = _1, Y = _2, Z = [a, _1, _2]]
```

3.3.2 get0/1 und get/1

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

Semantik (`get/1`)

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

3.3.3 eof/0 und eoln/0

Die Prädikate `eof/0` und `eoln/0` sind wie in PASCAL definiert und testen, ob das Streamende bzw. das Zeilenende erreicht wurde. Diese Prädikate haben keine Seiteneffekte. Das Prädikat `eof/0` wird genau dann wahr, wenn das Streamende erreicht ist. Das ist der Fall, wenn das letzte Zeichen eines Files eingelesen wurde. Wenn der Stream ein Terminal ist, wird `eof/0` erst nach Lesen des Fileendekennzeichens erfüllt. Das Prädikat `eoln/0` wird genau dann wahr, wenn das nächste Zeichen ein Zeilenendezeichen ist bzw. wenn `eof/0` wahr wird.

Beispiel 4 (`copy/0`)

copy/0 ist ein Prädikat, das den Rest des aktuellen Eingabefiles einliest und unverändert auf den Ausgabestream ausgibt.

```
copy:-eof.
copy:-repeat,get0(X),(eof ; put(X) , fail).
```

3.3.4 unget/0

Das Prädikat `unget/0` sorgt dafür, daß das zuletzt eingelesene Zeichen noch einmal eingelesen werden kann, unabhängig vom eventuellen Umschalten des Eingabestream. Wiederholtes `unget/0` hat keinen Ef-

fekt. `unget/0` wird immer wahr. `unget/0` erlaubt die relativ effiziente Implementierung von Algorithmen, die mit einem Zeichen Vorausschau arbeiten.

3.3.5 skip/1

Das Prädikat `skip/1` dient zum Überspringen von Zeichen auf dem aktuellen Eingabestream. Das Argument von `skip/1` muß ein auswertbarer arithmetischer Ausdruck sein, der eine Integerzahl ergibt. `skip/1` liest solange Zeichen vom aktuellen Eingabestream, bis ein Zeichen mit dem ASCII-Wert dieser Integerzahl gelesen oder das Streamende erreicht wurde. Dieses Prädikat wird meistens zum Überlesen bis zum Zeilenende in der Form skip(10) eingesetzt. Damit lassen sich die bei der interaktiven Arbeit beim Abschließen einer Eingabe zwangsläufig entstehenden Zeichen am einfachsten übergehen.

Semantik (`skip/1`)

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

3.3.6 ask/1

Mit dem Prädikat `ask/1` läßt sich das erste Zeichen einer Nutzerantwort bzw. das erste Zeichen einer Zeile abtesten. Das Argument von `ask/1` muß ein auswertbarer arithmetischer Ausdruck sein, der eine Integerzahl ergibt. `ask/1` liest eine Zeile vom aktuellen Eingabestream. Das Prädikat ist erfolgreich, wenn das erste Zeichen dieser Zeile einen ASCII-Wert hat, der dieser Integerzahl entspricht.

Semantik (`ask/1`)

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

3.4 Ausgabeoperationen

Die im folgenden beschriebenen Ausgabeprädikate beziehen sich alle auf den aktuellen Ausgabestream. Jedem dieser Prädikate kann man den Präfix tty voranstellen. Sie beziehen sich dann auf den Standardausgabestream des Interpreters, unabhängig davon, ob dieser gerade der aktuelle Ausgabestream ist oder nicht.

3.4.1 write/1, writeq/1 und display/1

Diese Prädikate schreiben ihr Argument als Prolog-Term auf den aktuellen Ausgabestream. `write/1` ist die Standardausgabeform und benutzt eine für den Nutzer gut lesbare Schreibweise, die die aktuellen Operatordeklarationen nutzt. Atome werden nicht gequotet. Ungebundene Variablen werden als Unterstrich, gefolgt von einer Zahl dargestellt. `writeq/1` arbeitet wie `write/1`, jedoch werden Atome, wenn nötig, gequotet. Mit `writeq/1` ausgegebene Terme können daher später immer wieder eingelesen werden (vorausgesetzt es gelten beim Lesen dieselben Operatordeklarationen wie beim Schreiben des Terms). `display/1` dagegen nutzt die Operatorschreibweise von Prolog nicht. Alle Terme werden in

ihrer Standardnotation als Funktorterme ausgegeben. Dies ist ein Hilfsmittel, um die Interpretation von Prologtermen, die in Operatorschreibweise vorliegen, zu testen.

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

3.4.2 put/1

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

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

3.4.3 nl/0 und tab/1

Das Prädikat `nl/0` schreibt das Zeichen bzw. die Zeichenfolge für das Zeilenende auf den aktuellen Ausgabestream. Grundsätzlich ist zu beachten, daß HU-Prolog aus der UNIX-Tradition stammt. Die Umsetzung des Zeilenendezeichens wird entsprechend C-Konventionen vom Low-Level-I/O-System gemacht. Das kann aber an einigen Stellen zur Verwirrung führen.

Semantik (`nl/0`)

```
nl:- put("\n").
```

Das Prädikat `tab/1` schreibt eine Anzahl von Leerzeichen auf den Ausgabestream. Das Argument von `tab/1` muß ein auswertbarer arithmetischer Ausdruck sein, der eine nichtnegative Integerzahl ergibt. Eine entsprechende Anzahl Leerzeichen wird auf den aktuellen Ausgabestream geschrieben.

Semantik (`tab/1`)

```
?- private(tabbing).
tab(Value):-N is Value, tabbing(N).
tabbing(0).
tabbing(I):-put(" "),J is I-1,tabbing(J).
?- hide(tabbing).
```

3.5 Bildschirmsteuerung

Für die korrekte Funktionsweise des zeichenorientierten Nutzerinterface (CUI) ist HU-Prolog in der DOS-Version auf den ANSI-Treiber angewiesen. Die Bildschirmoperationen werden durch spezielle built-in-Prädikate angestoßen, ggf. auf dem eingebauten Window-System simuliert und in die Steuersequenzen für die direkte Bildschirmsteuerung umgesetzt. Die gesamte Bildschirmsteuerung erfolgt nach außen über die Standardausgabe des Prologsystems, so daß die Steuersequenzen auch über Pipes oder Ausgabeumlenkung weitergereicht werden können. Die direkte Ausgabe von ANSI-Steuersequenzen mittels `put/1` oder `write/1` wird durch das Window-System weggefiltert.

3.5.1 cls/0 und gotoxy/2

Das Prädikat `cls/0` sendet das Zeichen bzw. die Zeichenfolge zum Löschen des Bildschirms auf den aktuellen Ausgabestream. Wenn der aktuelle Ausgabestream ein Window ist, wird nur dieses Window gelöscht.

Ein Aufruf von `gotoxy(XVal,YVal)` sendet die Zeichenfolge zur Cursorpositionierung auf den aktuellen Ausgabestream. Die Argumente von `gotoxy/2` müssen auswertbare ganzzahlige, arithmetische Ausdrücke sein. Der erste Parameter beschreibt die X-Koordinate (Spalte), der zeite Parameter die Y-Koordinate (Zeile). Die zulässigen Wertintervalle ergeben sich aus den zulässigen Bildschirm- bzw. Windowgrößen. Der Cursor wird auf die Spalte XVal und die Zeile YVal positioniert. Wenn die Werte von XVal und YVal nicht im vorgegebenen Bereich des Bildschirms liegen, ist das Ergebnis undefiniert. Die linke obere Ecke hat die Koordinaten (1,1).

Kapitel 4

Termbehandlung

4.1 Termklassifikation

HU-Prolog stellt eine Reihe von Testprädikaten für die Termklassifikation zur Verfügung. Diese Prädikate haben ein deterministisches Verhalten und keine Seiteneffekte. Die folgende Tabelle zeigt, welche Bedingungen die Argumente der Prädikate erfüllen müssen, damit ein Aufruf des jeweiligen Prädikats erfolgreich ist:

Prädikat	ist erfolgreich, falls:
atom(X)	X ein Atom ist
integer(X)	X eine Integerzahl ist
real(X)	X eine Realzahl ist
number(X)	X eine Zahl (Integer oder Real) ist
atomic(X)	X ein Atom oder eine Zahl ist
var(X)	X eine ungebundene Variable ist
nonvar(X)	X keine ungebundene Variable ist
ground(X)	X keine ungebundene Variable enthält
compound(X)	X ein strukurierter Term ist
list(X)	X eine Liste ist
string(X)	X ein String ist

Diese Bedingungen sind rein syntaktischer Natur, es spielt also z.B. keine Rolle, ob eine Real-Zahl einen ganzzahligen Wert repräsentiert. Die folgende Übersicht zeigt für einige Beispielterme, wie sich die Prädikate verhalten. Dabei bedeutet ○, daß der Aufruf fehlschlägt, und ● , daß er wahr wird.

Objekt	atom	integer	real	number	atomic	var	nonvar	invar	ground	compound	list	string
`X`	○	○	○	○	○	●	○	●	○	○	○	○
`_`	○	○	○	○	○	●	○	●	○	○	○	○
`98`	○	●	○	●	●	○	●	○	●	○	○	○
`9.8e+1`	○	○	●	●	●	○	●	○	●	○	○	○
`'98'`	●	○	○	○	●	○	●	○	●	○	○	○
`"98"`	○	○	○	○	○	○	●	○	●	●	●	●
`[]`	●	○	○	○	●	○	●	○	●	○	●	●
`[x,pi/2]`	○	○	○	○	○	○	●	○	●	●	●	○
`hallo`	●	○	○	○	●	○	●	○	●	○	○	○
`f(X,a)`	○	○	○	○	○	○	●	●	○	●	○	○
`sin(phi)`	○	○	○	○	○	○	●	○	●	●	○	○

Semantik (`Testprädikate`)

Die Prädikate `var/1`, `atom/1`, `real/1` und `integer/1` sind elementar und widerspiegeln die interne Struktur der Prolog-Terme. Dagegen lassen sich die anderen in Prolog definieren.

```
number(X):-integer(X); real(X).
atomic(X):-atom(X); number(X).
compound(X):-not var(X),not atomic(X).

ground(X):-var(X),!,fail.
ground(X):-atomic(X),!.
ground([X|L]):-!,ground(X),ground(L).
ground(X):-X=..[_|L],ground(L).

nonvar(X):-not var(X).

list(X):-var(X),!,fail.
list([]).
list([_|L]):-list(L).

string(X):-var(X),!,fail.
string("").
string([C|L]):-integer(C),0=<C,C=<255,string(L).
```

4.2 Termanalyse und -synthese

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

4.2.1 =../2

Das sogenannte univ-Prädikat `=../2` dient der Umwandlung zwischen beliebig strukturierten Termen und Listen. Die zu einem Term äquivalente Liste ist entsprechend der Termform definiert:

- zu einer Zahl ist die einelementige Liste, bestehend aus ebendieser Zahl, äquivalent
- zu einem Atom ist die einelementige Liste, bestehend aus diesem Atom, äquivalent
- zu einem strukturierten Term der Form f(t1,...,tn) ist die (n+1)-elementige Liste bestehend aus dem Hauptfunktor f des Terms gefolgt von den einzelnen Argumenten äquivalent.

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

4.2.2 functor/3

Die Funktion des Prädikates `functor/3` wird durch das erste Argument bestimmt. Handelt es sich dabei um eine freie Variable, so synthetisiert das Prädikat `functor/3` aus den anderen beiden Argumenten, die in diesem Fall vollständig instantiiert sein müssen, einen neuen Term und bindet die Variable im ersten Argument an diesen Term. Ist das erste Argument keine freie Variable, so wird die Struktur dieses Terms

analysiert. Die weiteren Argumente werden dann mit den Parametern der Termstruktur, das heißt dem Funktor und der Stelligkeit unifiziert. Der Aufruf ist erfolgreich, wenn beide Unifizierungen erfolgreich sind, andernfalls schlägt er fehl. Die Termsynthese erzeugt den allgemeinsten Term, dessen anschliessende Analyse genau die vorgegebenen Parameter liefern würde.

Semantik (functor/3)

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

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

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

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

4.2.3 arg/3

`arg(N,T,A)` unifiziert das `N`-te Argument des strukturierten Terms `T` mit `A`. Falls `T` kein strukturierter Term ist oder falls `N` keine ganze Zahl zwischen `1` und der Stellenzahl des strukturierten Terms `T` ist oder falls die Unifikation des `N`-ten Arguments von `T` mit `A` nicht möglich ist, schlägt `arg/3` fehl.

Semantik (arg/3)

Mit Hilfe eines Hilfsprädikats `n_th/3`, das bei einem Aufruf n_th(N,L,A) das Argument A mit dem N-ten Argument der Liste L unifiziert, läßt sich die Semantik von `arg/3` auf `=../2` zurückführen:

```
?- private(n_th).
     n_th(1,[A|_],A).
     n_th(N,[_|T],A) :- N > 1, M is N - 1, n_th(M,T,A).
     arg(N,T,A) :- T =.. [F|L],n_th(N,L,A).
?- hide(n_th).
```

4.3 Analyse und Synthese von Atomen

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

4.3.1 current_atom/1

Das Prädikat `current_atom/1` generiert (beim Backtracking) in lexikographischer Reihenfolge Namen und Stellenzahl aller dem System bekannten Atome und unifiziert sein Argument mit Termen der Form `Name/Arity`.

4.3.2 name/2

`name/2` konvertiert Atome und Zahlen in Strings, das heißt Listen von Integerzahlen, die die ASCII-Codes der Zeichen des Atoms repräsentieren und umgekehrt ASCII-Listen in Atome. Falls bei einem Aufruf `name(A,L)` das erste Argument `A` ein Atom oder eine Zahl ist, wird eine Liste der ASCII-Werte des Names des Atoms oder der Zeichenkettenrerpräsentation der Zahl erzeugt und mit `L` unifiziert. Falls

`A` eine Variable ist, muß `L` an eine ASCII-Liste gebunden sein. Aus dieser Liste wird ein Atom mit den Zeichen entsprechend der ASCII-Codes konstruiert. Anschliessend wird `A` mit dem konstruierten Atom unifiziert.

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

Beispiel 5 (`filename/2`)

filename/2 ist ein Prädikat, das Filenamen in der aus der MSDOS-üblichen Notation (als Prolog-Term) in ein Atom übersetzt, das z.B. beim Aufruf von assign/2 benutzt werden kann

```
filename(A,A):-atom(A).
filename(A,X):-integer(A),A>=0,name(A,L),name(X,L).
filename(A.B,X):-name(A,La),filename(B,BB),name(BB,Lb),
 append(La,[46|Lb],Lx),name(X,Lx).
filename(A\B,X):-filename(A,AA),name(AA,La),filename(B,BB),name(BB,Lb),
 append(La,[92|Lb],Lx),name(X,Lx).
```

4.4 Termvergleich

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

4.4.1 Unifizierbarkeit (=/2 und \=/2)

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

Semantik (`=/2 und \=/2`)

```
X=X.
X\=Y:-X=Y,!,fail.
```

4.4.2 Identität (==/2 und \==/2)

Die Prädikate `==/2` und `\==/2` vergleichen ihre Argumente auf Identität. Sie verhalten sich deterministisch und erzeugen beim Backtracking keine weiteren Lösungen.

Die Identitätsrelation zweier Terme ist wie folgt definiert:

- Zahlen sind mit Zahlen identisch, die vom gleichen Typ (Integer bzw. Real) sind und denselben Wert repräsentieren. Mit anderen Termen sind Zahlen nicht identisch.
- Atome sind nur mit sich selbst identisch.

- Variablen sind nur mit Variablen identisch, mit denen sie bereits vorher unifiziert wurden, das heißt wenn sie nach Dereferenzieren physisch auf die gleiche Termstruktur verweisen. Mit anderen Termen sind Variablen nicht identisch.
- Strukturierte Terme sind nur mit strukturierten Termen identisch, die denselben Hauptfunktor und dieselbe Stelligkeit besitzen und deren Argumente paarweise identisch sind.

Das Prädikat `==/2` testet seine Argumente auf Identität. Es wird genau dann erfolgreich, wenn die Argumente identisch sind. Das Prädikat `\==/2` ist die Umkehrung von `==/2` und wird genau dann erfolgreich, wenn die Argumente nicht identisch sind.

4.4.3 Lexikographische Ordnung (`@</2` ,`@=</2` ,`@=/2` ,`@>=/2` ,`@>/2` ,`@\=/2`)

Die Prädikate `@</2`, `@=</2` , `@>/2`, `@>=/2`, `@=/2` und `@\=/2` vergleichen zwei Terme in Bezug auf ihre lexikographische Ordnung. Sie verhalten sich deterministisch und erzeugen beim Backtracking keine weiteren Lösungen. Die lexikographische Ordnung zwischen Termen ist wie folgt definert:

- Variablen sind kleiner als alle anderen Terme und untereinander gleich.
- Zahlen sind kleiner als alle sonstigen nichtvariablen Terme und sind untereinander nach ihrem numerischen Wert sortiert. Dabei spielt der Zahlentyp (Integer oder Real) keine Rolle.
- Atome sind größer als Zahlen und Variablen und kleiner als alle anderen Terme. Untereinander sind Atome nach ihrer lexikographischen Ordnung (entsprechend dem ASCII-Code) sortiert.
- Strukturierte Terme sind in der Ordnung die größten Terme. Untereinander sind sie nach den Hauptfunktoren und, falls diese gleich sind, nach den Argumenten gemäß ihrer lexikographischen Ordnung sortiert.

$X \circ Y$	ist erfolgreich, wenn in der lexikographischen Ordnung...
X `@<` Y	... X kleiner als Y ist
X `@=<` Y	... X kleiner oder gleich Y ist
X `@>` Y	... X größer als Y ist
X `@>=` Y	... X größer oder gleich Y ist
X `@=` Y	... X gleich Y ist bzw.
X `@\=` Y	X ungleich Y ist

Kapitel 5

Syntax und Operatordeklarationen

Die abstrakte Syntax von Prolog unterscheidet drei Termformen: *Variablen*, *Zahlen* und *Funktorterme*, die aus einem *Atom*, charakterisiert durch *Namen* und Stellenzahl (*Arität*), sowie einer Liste von Argumenttermen bestehen, wobei die Anzahl der Argumentterme genau der Stelligkeit des Atoms entspricht).

Obwohl dieses Termkonzept unter rein algorithmischem Aspekt vollkommen ausreichend ist, wird der praktische Gebrauch von Prolog durch die Möglichkeit der freien und dynamischen Operatordeklaration wesentlich bestimmt.

Zusätzlich zu einer, wie in jeder Programmiersprache, vordefinierten Menge von Standardoperatoren gibt es in Prolog die Möglichkeit, zur Programm*ausführungszeit* (als Spezialfall davon z.B. beim Programmladen) die Syntax der Sprache zu modifizieren. Das erfolgt über dynamische Operatordeklarationen mit dem `op/3`-Prädikat.

5.1 Standardoperatoren

In Prolog werden Operatoren durch ihre *Priorität* und *Assoziativität* beschrieben. Die Priorität eines Operators ist eine ganze Zahl aus dem Bereich 0...1200, wobei (widersinnigerweise) ein Operator mit kleinerer Priorität stärker bindet als ein Operator mit höhererer Priorirät. Zum Beispiel hat die Addition (`+/2`) die Priorität 500 und die Multiplikation (`*/2`) die Priorität 400. Normalerweise geht man davon aus, daß Operatoren eine Priorität zwischen 0 und 999 haben. Das liegt an der Doppelrolle des Kommas in Prolog. Einerseits trennt es die Argumente eines mehrstelligen strukturierten Terms, andererseits ist `,/2` ein zweistelliger Operator, der die sequentielle (konjunktive) Verknüpfung von Termen (Prolog-Aufrufen) beschreibt. Dieses Problem hat man dadurch gelöst, daß das Komma eine sehr hohe Priorität, nämlich 1000, erhielt, während die "normalen" nutzerdefinierbaren Operatoren eine Priorität echt kleiner als 1000 haben sollten. Dadurch trennt das Komma (unabhängig von der Lesart) eindeutig die Argumente eines strukturierten Terms.

Die globale Syntax eines Prolog-Programms wird durch die Priorität von Pseudo-Operatoren definiert: Eine Prolog-Klausel besteht aus Kopf und Körper, getrennt durch `:-/2`. Der Operator `:-/2` hat die absolut höchste Priorität. Der Körper einer Klausel ist eine Disjunktion von Konjunktionen. Das wird dadurch erreicht, daß die Priorität des Disjunktionsoperators `;/2` (1100) genau zwischen der Priorität von `:-/2` (1200) und der Priorität des Konjunktionsoperators `,/2` (1000) liegt, der seinerseits die sequentielle Verkettung von Prolog-Aufrufen beschreibt. Eine Prolog-Klausel der Form "`p:-a,b; c,d,e.`" wird also unter Beachtung der Prioritäten als "`:-(p,;(,(a,b),(c,(d,e)))).`" gelesen.

Während man in der Mathematik gemeinhin sagt, Klammern dürfen (nur) bei assoziativen Operationen weggelassen werden, weil die beiden möglichen Interpretationen einer ungeklammerten Formel $a \circ b \circ c$, nämlich $a \circ (b \circ c)$ und $(a \circ b) \circ c$ beide gleich sind, ist es in der Informatik übliche Praxis, die Klammern auch bei nichtassoziativen Operationen wegzulassen. Kaum jemand nimmt Anstoß an der Formel "`a-b-c-d`", weil allgemein unterstellt wird, daß "`((a-b)-c)-d`" gemeint ist. Umgekehrt wird bei

der sequentiellen Konjunktion (z.B. in C) “`a && b && c && d`” als “`a && (b && (c && d))`” gelesen.

In einer Sprache mit der Möglichkeit freier Operatorvereinbarungen muß man nun Vorkehrungen treffen, um die genaue Lesart solcher nicht geklammerter Terme zu definieren. In Prolog benutzt man dazu spezielle Atome (`xfx/0`, `xfy/0`, `yfx/0`, `fx/0`, `fy/0`, `xf/0` und `yf/0`), die die *Assoziativität* und zusätzlich die *Stellung* (Infix, Prefix bzw. Postfix) eines Operators beschreiben:

Assoziativität	Stellung	Verwendung
`xfy`	Infix	$a \circ b \circ c = a \circ (b \circ c)$
`yfx`		$a \circ b \circ c = (a \circ b) \circ c$
`xfx`		$a \circ b \circ c$ ist verboten
`fy`	Prefix	$\circ \circ a = \circ(\circ(a))$
`fx`		$\circ \circ a$ ist verboten
`yf`	Postfix	$a \circ \circ = \circ(\circ(a))$
`xf`		$a \circ \circ$ ist verboten

Die genaue Lesart der assoziativitätsbeschreibenden Atome ergibt sich aus der mnemotechnischen Konvention, daß `f` für den Operator, “`x`” für einen Term echt niedrigerer Priorität und “`y`” für einen Term unter Umständen gleicher Priorität steht.

Die Standardoperatoren von HU-Prolog lassen sich nun in der nachfolgenden Tabelle zusammenfassen:

Priorität	Assoziativität	Operatoren	Verwendung
1200	`xfy`	`:-`	Programmstruktur
1100	`xfy`	`;`	Disjunktion
1050	`xfy`	`->`	Subjunktion
1000	`xfy`	`,`	Konjunktion
900	`fy`	`not` , `\+`	metalogische Negation
700	`xfx`	`:=` , `is`	Wertzuweisung
		`=`, `\=`	Unifikation
		`==`, `\==`	Identität
		`=:=`, `=\=`, `<`, `=<`, `>=`, `>`	numerischer Vergleich
		`@=`, `@\=`, `@<`, `@=<`, `@>=`, `@>`	lexikographischer Vergleich
		`=..`	univ (Termanalyse)
650	`xfy`	`\`, `\\` , `&` , `&&`	bitweise und logische Verknüpfungen
650	`fy`	`‘`	Quote-Operation
600	`xfy`	`>>` , `<<`	Links- und Rechtsverschiebung
500	`yfx`	`+` , `-`	arithmetische Operationen
400	`yfx`	`*` , `/` , `//` , `mod`	
350	`xfy`	`**`	
300	`xfy`	`.`	cons (Listenoperation)
300	`fy`	`-`	arithmetische Negation (Minus)
		`~` , `/`	bitweise und logische Negation

5.2 op/3

`op/3` macht dem Interpreter einen oder mehrere neue Operatoren mit gleicher Priorität und Assoziativität bekannt. Bei einem Aufruf der Form `op(P,A,N)` ist `P` die Priorität, `A` die Assoziativität und N der Name bzw. eine Liste von Namen der zu definierenden Operatoren.

Es ist nicht möglich, einen Operatornamen gleichzeitig als Infix und Postfix bzw. als Prefix und Postfix zu vergeben, da dieser Fall syntaktisch bzw. semantisch nicht eindeutig auflösbar ist. Ein Fehler tritt auf, wenn eines der Argumente von `op/3` einen unzulässigen Wert hat. Ein Aufruf von `op(0,A,N)` löscht eine eventuell vorhandene Operatordeklaration. Ein neuerlicher Aufruf von `op/3` für den gleichen Operator überschreibt die bisherige Priorität und Assoziativität. Die Standard-Operatoren können umdefiniert

werden, es ist jedoch zweckmäßig, diese Umdefinition erst dynamisch beim Programmstart bzw. nach dem Einlesen aller Quellfiles vorzunehmen, weil sonst u.U. Fehler in der Interpretation des Quellcodes auftreten.

Die "Unsichtbarkeit" der eigentlichen Termstruktur ist häufig ein Problem im Debugging-Prozeß: wenn der Fehler nämlich nicht im Algorithmus, sondern in der Kodierung der Grammatik liegt. Für Debugging-Zwecke gibt es daher das `display/1`-Prädikat, das den Argumentterm in vollständig geklammerter Funktorschreibweise ausgibt. Nur so läßt sich die Wirkung der Operatordeklarationen tatsächlich testen:

```
?- display(a + b * c).
+(a, *(b, c))
yes
?- display(a - b - c).
-(-(a, b), c)
yes
?- display([a,b,c]).
.(a, .(b, .(c, [])))
yes
```

Beispiel 6 (`op/3`)

Mit Hilfe von Operatordeklarationen lassen sich relativ einfach Wissensrepräsentations- oder Datenbankabfragesprachen definieren:

```
?- op(100,xfx,von).
yes
?- op(50,xfy, und).
yes
?- op(700, xfx, ist).
yes
?- display(anton ist vater von klaus).
ist(anton, von(vater, klaus))
yes
?- display(heidi ist schwester von karl und anna).
ist(heidi, von(schwester, und(karl, anna)))
yes
```

Die aktuell gültigen Operatordeklarationen werden bei der termorientierten Ein- und Ausgabe (mittels `read/1` und `write/1` sowie deren Derivaten) automatisch angewendet. Das erlaubt eine sehr flexible Gestaltung der Ein- und Ausgabesprache. Es ist z.B. möglich große Teile der Syntax von PASCAL oder C durch Operatordeklarationen zu beschreiben, so daß "intelligente" Programmierwerkzeuge relativ einfach in Prolog entwickelt werden können.

Beispiel 7 (`PL/0`)

Als Beispiel betrachten wir die Sprache PL/0 (Wirth, Niklaus: Compilerbau. Teubner, Stuttgart 1977, S.36). Die nachfolgenden Operatordeklarationen beschreiben die Syntax der Sprache vollständig (Das lokale Umschalten in den Systemmodus ist notwendig, weil Standard-Deklarationen überschrieben werden):

```
?-sysmode(on).
?-op(900,xfy, ';').
?-op(850,fx,[const,var,procedure]).
?-op(800,xfy, ',').
?-op(900,fy,begin).
?-op(899,yf,end).
?-op(900,xfy,[then,do]).
?-op(800,fx,[if,while]).
?-op(700,fx,odd).
?-op(100,fx,call).
?- sysmode(off).
```

Nun kann mit `read(X)` *ein komplettes PL/0-Programm eingelesen und dann als Prolog-Term weiterverarbeitet werden, zu beachten ist jedoch, daß die von* `read/1` *bereitgestellte Termstruktur noch nicht die abstrakte Syntax und damit die Semantik des PL/0-Programms wiederspiegelt, hier müßte zunächst eine Termtransformation vorgenommen werden.*

```
const m=7, n=85;
var x,y,z,q,r;
procedure multiply;
  var a,b;
  begin a:=x; b:=y; z:=0;
        while b>0 do
          begin if odd b then z:=z+1; a:=2*a; b:=b/2 end
  end;
begin
  x:=m; y:=n; call multiply; write(z)
end.
```

5.3 current_op/3

Mit `current_op/3` lassen sich alle aktuell definierten Operatoren abfragen. Ein Aufruf von `current_op(P,A,N)` unifiziert `P`, `A` und `N` mit der Priorität, der Assoziativität und dem Namen eines dem System bekannten Operators. Beim Backtracking werden alle möglichen Lösungen (einschließlich der Standardoperatoren) in lexikographisch geordneter Reihenfolge generiert.

Beispiel 8 (`list_op/0`)

Das folgende Programm schreibt alle dem System aktuell bekannten Operatordefinitionen in einer solchen Form auf ein File, daß dieses File direkt wieder eingelesen werden kann.

```
list_op(File) :-
    tell(File), current_op(V,A,N), writeq((?-op(V,A,N)),nl, fail.
list_op:-
    told.
```

Kapitel 6

Manipulation der Datenbasis

Dieser Abschnitt beschreibt die Prädikate zur Modifikation und Abfrage der Datenbasis, die sowohl Programmcode als auch statische Daten enthält.

HU-Prolog basiert auf dem Konzept einer einheitlichen internen Datenbasis, die Programme und Daten in komprimierter Form enthält, ohne zwischen statischem bzw. dynamischem Programmcode und Daten zu unterscheiden.

Da die Klauseln beim Einfügen in die Datenbasis in eine Art "Normalform" gebracht werden, kann es zwischen dem Einspeichern in die Datenbasis und der Abfrage zu leichten Veränderungen im Regelkörper kommen. Das betrifft aber ausschließlich das Einfügen bzw. Streichen von *Leeraufrufen* (`true/0`).

Für die externe Darstellung von Datenbasiseintragungen wird einheitlich das Klauselformat benutzt, d.h. eine Repräsentation der Fakten und Regeln als Prolog-Terme. Bei der Termdarstellung ist zu beachten, daß die Pseudo-Operatoren "`:-`", "`,`" und "`;`", die zur Abteilung von Regelkopf und Regelkörper bzw. zur Verknüpfung der einzelnen Aufrufe im Regelkörper dienen, eine sehr hohe Priorität besitzen, so daß Regeln, die als Parameter an andere Prädikate übergeben werden sollen, immer zusätzlich geklammert werdenm müssen!

Die Datenbasis zerfällt logisch in Prozeduren. Das sind Mengen von Klauseln mit dem nach Namen und Stellenzahl gleichem Hauptfunktor. Innerhalb einer Prozedur sind die Klauseln streng sequentiell geordnet. Die interne Ordnung entspricht zunächst genau der Reihenfolge des Ladens und damit der Reihenfolge im Quelltext. Klauseln können aber während der Abarbeitung des Programmes dynamisch erzeugt und gestrichen werden.

Die einfachste und schnellste Form des Zugriffs auf die Datenbasis ist der Aufruf eines Prädikats, allerdings wird dabei das Prädikat *abgearbeitet*, was wiederum Seiteneffekte haben kann. Im Unterschied zu klassischen Programmiersprachen, bei denen es eine klare funktionale Trennung zwischen Datenstrukturen und Programmstrukturen gibt, vermischen sich in Prolog beide Aspekte. Anstelle der expliziten Speicherung einer Zustandsvariable kann die implizite Änderung des von ihre beeinflußten Programmcodes stehen, und umgekehrt.

Grundsätzlich können Programmcode oder Daten nur klauselweise eingefügt, gestrichen oder ausgetauscht werden. Die lokale Manipulation innerhalb einzelner Klauseln ist nicht möglich.

6.1 consult/1 und reconsult/1

`consult/1` ist die ursprüngliche Form des Programmladens in Prolog. Durch einen Aufruf von `consult(`*Filename*`)` wird das File mit dem angegebenen Namen eröffnet und termweise mit `read/1` sequentiell eingelesen. Für die Abgrenzung der Terme gelten dabei die üblichen Prolog-Konventionen (Punkt gefolgt von einem Leerzeichen oder Zeilenendezeichen). Die eingelesenen Terme werden interpretiert:

- Terme mit dem Hauptfunktor `?-/1` oder `:-/1` werden als Aufrufe abgearbeitet;
- Der Term `end/0`, der normalerweise von der Eingaberoutine bei Erreichen des physischen Fileendes generiert wird, dient als logisches Fileendekennzeichen und beendet den Einleseprozeß, unabhängig, ob tatsächlich das Fileende erreicht wurde;
- Terme mit dem Hauptfunktor `:-/2` werden als *Regel* interpretiert und an das Ende der Datenbasis angefügt.
- Alle anderen Terme werden als *Fakten* interpretiert und ebenso an das Ende der Datenbasis angefügt.

Wird `consult/1` mit einem Streamnamen aufgerufen, der aktuell einem Window zugeordnet ist, so kann über dieses Window interaktiv Prolog-Quelltext eingegeben werden. Das mehrfache "Konsultieren" eines Files bewirkt, daß die entsprechenden Regeln mehrfach abgespeichert werden. Das kann, insbesondere bei mehrmaligem Durchlaufen des Run-Debug-Edit-Zyklus inneralb einer Prolog-Sitzung, zu erheblichem Frust führen. Für das wiederholte Laden eines Quelltextfiles, z.B. nach Korrekturen, gibt es daher das `reconsult/1`-Prädikat. Es wirkt wie `consult/1`, doch werden beim Einlesen neuer Klauseln für ein bereits definiertes Prädikat zunächst alle alten Klauseln als zur vorangehenden Programmversion gehörig aus der Datenbasis gestrichen.

Semantik (`consult/1`)

Der Steuerablauf beim Konsultieren eines Files läßt sich in Prolog beschreiben:

```
?-private([consult_file,consult_term]).
consult(F):-see(F),consult_file,seen.

consult_file:-
   repeat, read(X), consult_term(X), X=end.
consult_term(end).
consult_term((?- Call)):-!,Call,!.
consult_term((:- Call)):-!,Call,!.
consult_term(Clause):-assert(Clause).
?-hide([consult_file,consult_term]).
```

Da das (re)konsultieren von Quellfiles zu den häufigsten Nutzerinteraktionen gehört, gibt es eine Kurznotation: Der direkte Aufruf einer Liste von Filenamen bewirkt das Konsultieren aller angegebenen Files von links nach rechts. Ein Minus-Zeichen (`-/1`) vor dem Filenamen erzwingt `reconsult/1`. Ein Aufruf von `?-[f1,-f2].` enspricht also `?-consult(f1),reconsult(f2).`

Semantik (`./2`)

```
[F] :- atom(F),consult(F).
[-F] :-atom(F),reconsult(F).
[F,FF|L] :- [F],[FF|L].
```

6.2 Einfügen von Klauseln

Für das Einfügen neuer Klauseln in die Datenbasis ist die Prädikatfamilie `assert` verantwortlich. `assert`[a] fügt am Anfang der Datenbasis ein, `assert`[z] am Ende.

Mit dem Einfügen einer neuen Klausel wird diese aktiviert, d.h. der nächste Aufruf eines so modifizierten Prädikats bezieht sich unmittelbar auf die erweiterte Klauselmenge. Wird eine bereits aktivierte Prozedur modifiziert, so wird die Änderung des Codes jedoch nur wirksam, wenn noch ein Backtrack-Point für diese Prozedur aktiv ist. Der Prolog-Abarbeitungsmechanismus arbeitet mit Vorausschau, um das unnötige Anlegen von Verzweigungspunkten zu vermeiden. Wenn daher einmal festgestellt wurde, daß eine Prozedur keine weitern Verzweigungspunkte besitzt, ist auch ein nachträgliches Generieren eines solchen Backtrackpunkts wirkungslos. Darauf sollte man sich aber im Interesse portabler Programme nicht verlassen.

6.2.1 asserta/1

Das Prädikat `asserta/1` dient dem Einfügen neuer Klauseln am Anfang der jeweiligen Prozedur. Das Argument ist die einzufügende Klausel in Termdarstellung, wobei für Fakten lediglich der Klauselkopf anzugeben ist, während Regeln mit Hilfe des Funktors `:-/2` gebildet werden (und zusätzlich in Klammern einzuschließen sind). Der Klauselkopf darf dabei weder eine freie Variable noch eine Zahl sein, noch darf der Hauptfunktor in Name und Arität mit einem Systemprädikat übereinstimmen.

Beispiel 9 (`debug/1`)

Ein wirksames, selektives Testhilfsmittel ist das Einfügen einer "Pre-Fetch"-Klausel zu einem Prädikat, die z.B. nichts weiter macht, als den Aufruf zu protokollieren und anschließend fehlschlägt. Eine solche Klausel ist für den Inferenzprozeß transparent.
Wenn wir ein Prädikat `p/1` *definiert haben, so kann es durchaus angebracht sein, als erste Klausel* `p(X):-nl,write('Call: '),write(p(X)),fail.` *einzufügen, die dann jeden Aufruf mit seinen aktuellen Parametern protokolliert.*

```
debug(Name/Arity):-
  functor(X,Name,Arity),asserta((X:-nl,write('Call: '),write(X),fail)).
```

6.2.2 assert(z)/1

Dieses Prädikat dient zum Einfügen neuer Klauseln am Ende der jeweiligen Prozedur. Werden Klauseln mit `assertz` in die Datenbasis eingefügt, so entspricht die spätere Reihenfolge in der Datenbasis genau der Reihenfolge des Erzeugens. Das Prädikat `assert/1` ist vollständig synonym zu `assertz/1`.

6.3 Entfernen von Klauseln

Für das Entfernen von Klauseln ist die Prädikfamilie `retract` verantwortlich. `retract/1` ist ein backtrackbares Prädikat, das nacheinander alle auf die aktuellen Aufrufparameter passenden Klauseln aus der Datenbasis entfernt und als Terme in Klauselform mit den Aufrufparametern unifiziert. Die Information wird also aus der Datenbasis gelöscht, geht aber nicht unmittelbar verloren. `retractall/1` löscht alle auf ein gegebenes Muster passende Klauseln. `abolish/1` löscht eine bzw. alle gleichnamigen Prozeduren vollständig. Mit dem Löschen von Datenbasiseintragungen steht so gewonnener Speicherplatz wieder für neue Eintragungen zur Verfügung und wird vorrangig genutzt. Die Freigabe des Speicherbereichs kann sich aber verzögern, wenn die freizugebenden Bereiche aktiven Code enthalten.

6.3.1 retract/1

Das Prädikat `retract/1` löscht die erste Klausel der Datenbasis, deren Kopf bzw. Kopf und Körper mit dem Argument von `retract/1` unifizierbar ist. Beim Backtracking werden die weiteren mit dem

Argument unifizierbaren Klauseln gelöscht. Nach dem Löschen einer Klausel ist das Argument an einen Term gebunden, der genau der gestrichenen Klausel entspricht. Das Argument von `retract/1` muß mindestens soweit instantiiert sein, daß der Name und die Stellenzahl des Prädikats feststehen. `retract/1` schlägt fehl, wenn keine Klausel existiert, die unifiziert werden kann. Ein Fehler tritt auf, wenn versucht wird, ein Systemprädikat zu löschen.

6.3.2 retractall/1

Das Prädikat `retractall/1` löscht alle Klauseln aus der Datenbasis, deren Kopf sich mit dem Argument von `retractall/1` unifizieren läßt. Diese Unifizierung wird jedoch nicht ausgeführt, d.h. der Parameter ist nach dem Aufruf von `retractall/1` unverändert. Das Prädikat `retractall/1` wird immer wahr, unabhängig davon, ob und wieviele Klauseln gestrichen wurden.

Semantik (`retractall/1`)

```
retractall(X):-
    ( retract(X)        % streichen aller Fakten
    ; retract((X :- _)) % streichen aller Regeln
    ), fail.
retractall(_).
```

6.3.3 abolish/1 und abolish/2

Das Prädikat `abolish/2` erwartet als erstes Argument den Namen und als zweites Argument die Stelligkeit eines Prädikats und löscht alle zu diesem Prädikat gehörenden Klauseln aus der Datenbasis

`abolish/1` erwartet als Argument einen Namen und löscht alle Prädikate dieses Namens aus der Datenbasis.

Semantik (`abolish/1`)

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

Beispiel 10 (`stack`)

Mit `assert/1` *und* `retract/1` *lassen sich auch klassische abstrakte Datentypen realisieren. Die Grundidee besteht in der Verwendung eines lokalen Prädikats unter dessen Namen dynamisch Informationen abgelegt und wieder gestrichen werden. Wesentlich ist allerdings die korrekte Initialisierung der Datenstruktur, weil sonst die Prolog Fehlerbehandlung zu unerwünschten Effekten führen kann.*

```
?- private([stack,bottom]).
init_stack:-abolish(stack),assert(stack(bottom)).
empty_stack:-stack(X),!,X=bottom.
push(X):-asserta(stack(X)).
pop:-not empty_stack,retract(stack(X)).
top(X):-not empty_stack,stack(Y),!,X=Y.
?- init_stack.
?- hide([stack,bottom]).
```

6.4 Abfragen von Klauseln

Die Abfrage einzelner Klauseln erfolgt über die Prädikatfamilie `clause`. Ein Aufruf von `clause/2` unifiziert die Argumente beim Backtracking nach und nach mit allen Klauseln eines Prädikats. Mit

`current_predicate/1` kann hingegen die Existenz von Klauseln für eine durch Namen und Stelligkeit gegebene Prozedur getestet werden.

6.4.1 clause/2

Das Prädikat `clause/2` dient der expliziten Abfrage einzelner Klauseln aus der Datanbasis. Beim Aufruf von `clause(Head,Body)` wird eine Klausel gesucht, deren Kopf mit `Head` und deren Körper mit `Body` unifizierbar ist. Diese Unifizierung wird ausgeführt. Für Fakten wird als Klauselkörper `true/0` eingesetzt. Beim Backtracking werden, wenn möglich, alle weitere Lösungen erzeugt. Aus dem ersten Argument von `clause/2` müssen Name und Stellenzahl des Prädikats hervorgehen.

6.4.2 current_predicate/1

`current_predicate/1` ist ein Generator, der sein Argument mit dem Namen und der Stelligkeit aller momentan definierten Prädikate unifiziert.

6.4.3 listing/1 und listing/0

Das Prädikat `listing/0` gibt alle vom Nutzer, genauer gesagt, alle nicht im System-Mode definierten Klauseln auf den aktuellen Ausgabestream aus[1]. Das Prädikat `listing/1` spezifiziert die auszugebenen Klauseln näher. `listing(all)` ist äquivalent zu `listing/0`. `listing(Name)` gibt die Klauseln aller Prozeduren mit dem angegebenen Namen aus. `listing(Name/Arity)` gibt alle Klauseln der spezifizierten Prozedur aus.

[1] Die aktuelle Version unterdrückt auch die Ausgabe aller mit `ensure/` geladenen Prädikate

Kapitel 7

Arithmetik und funktionale Programmierung

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

- Die von C bekannten und bewährten Operationen wurden, soweit sie mit den Prolog-Grundkonzepten verträglich sind, einschließlich Syntaxregeln, Priorität und Assoziativität übernommen. Die Standardfunktionen von HU-Prolog entsprechen den gleichnamigen Funktionen der C-Standardbibliothek.
- Die funktionale Auswertung von Prädikaten sowie die Verwendung globaler, strukturierter Variablen werden unterstützt.
- Operationen und Funktionen können auf beliebige Termstrukturen angewendet werden und sind nicht auf numerische Werte beschränkt.

Die (arithmetische) Auswertung von Termen ist aus der Sicht der logischen Programmierung ein notwendiges Übel und daher grundsätzlich auf ausgewählte built-in-Prädikate beschränkt. In Edinburgh-Prolog sind dies das is/2-Prädikat, das einen arithmetischen Ausdruck auswertet und das Resultat mit einer Variablen unifiziert, und die arithmetischen Vergleichsprädikate (</2, =</2, >/2, >=/2, =:=/2 sowie =\=/2), die jeweils zwei Terme arithmetisch auswerten und die Ergebnisse miteinander vergleichen.

Die Auswertung des Ausdrucks erfolgt wie in klassischen Programmiersprachen: Bei Standardfunktionen und -operationen werden zuerst alle Argumente in der Reihenfolge ihres Auftretens von links nach rechts ausgewertet. Die Funktion selber wird anschließend auf die Argumentwerte angewendet. Der Resultattyp der Funktion ist dabei im allgemeinen von den Argumenttypen abhängig. Eine Typanpassung von *int* nach *real* erfolgt automatisch. Falls die Argumenttypen nicht mit den geforderten Typen übereinstimmen, wird ein Fehler ausgelöst.

In HU-Prolog wurde der is/2-Operator um einige Standardfunktionen und -operationen entsprechend den C-Konventionen erweitert. Die *funktionale und prozedurale Komponente* wird in HU-Prolog durch einen zusätzlichen Wertzuweisungsoperator :=/2 sowie eine Erweiterung der arithmetischen Vergleichsoperatoren realisiert.

Die Grundidee beim Wertzuweisungsoperator :=/2 besteht darin, das Ergebnis der Auswertung eines Terms einer globalen Variablen zuzuweisen, d.h. einen entsprechenden Eintrag in der Datenbasis vorzunehmen, und globale Variablen bei der Auswertung eines Terms über call/1 zu dereferenzieren. Falls die Datenbankeintragung nicht nur auf einen Fakt, sondern auf eine oder mehrere Prolog-Regeln verweisen sollte, so wird der Regelkörper wie bei einem Funktionsaufruf abgearbeitet. Bei nicht definierten Variablen oder Fehlschlagen des Regelaufrufs wird der Teilausdruck selber als Ergebnis zurückgeliefert.

7.1 Standardfunktionen und -operationen

Konstanten		
Name	Typ	Bedeutung
maxint	→ int	größte darstellbare ganze Zahl (2147483647)
minint	→ int	kleinste darstellbare ganze Zahl (-2147483648)
e	→ real	Eulersche Zahl (2.7182818284)
pi	→ real	Kreiszahl Pi (3.1415926535)
maxarity	→ int	maximale Funktor-Stellenzahl

1-stellige Funktionen		
Name	Typ	Bedeutung
`-x`	int → int real → real	arithmetische Negation (Minus)
`/x` `~x`	int → int	logische Negation bitweise Negation
`random(n)`	int → int	Zufallszahl im Bereich $[0..n-1]$
`real(x)` `entier(x)`	int → real real → int	Typkonvertierung

`exp(x)`
`ln(x)`
`log10(x)`
`sqrt(x)`
`sin(x)`
`cos(x)`
`tan(x)`
`asin(x)`
`acos(x)`
`atan(x)`
`floor(x)`
`ceil(x)`
	real → real	Exponentialfunktion (Basis e)
		Logarithmusfunktion (Basis e)
		Logarithmusfunktion (Basis 10)
		Quadratwurzel
		Sinusfunktion
		Cosinusfunktion
		Tangensfunktion
		Arcussinusfunktion
		Arcuscosinusfunktion
		Arcustangensfunktion
		nächstkleinere ganze Zahl
		nächstgrößere ganze Zahl

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

7.2 `is/2`

Ein Aufruf "V `is` E" wertet den Ausdruck E aus und unifiziert das Ergebnis mit dem Term V. Je nachdem, ob diese Unifikation möglich ist oder nicht, ist `is/2` erfolgreich oder schlägt fehl.

Beispiel 11 (`genint/1`)

Das Prädikat genint/1 erzeugt beim Backtracking fortlaufend wachsende ganze Zahlen 0,1,2,... .

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

Beispiel 12 (`for/3`)

Ein Aufruf des Prädikats for(X,I,J) bindet die Variable X nacheinander an die Werte I, I+1, I+2, ..., J und schlägt dann fehl.

```
for(_,I,J):- I>J, !, fail.
for(X,I,_):- X is I.
for(X,I,J):- B is I+1, for(X,B,J).
```

Beispiel 13 (`fak/2`)

Ein Aufruf von fak(F,N) berechnet die Fakultat F von N, vorausgesetzt, N ist als numerischer Wert gegeben.

```
fak(1,0) :- !.
fak(F,N):-
    N1 is N - 1, fak(F1,N1), F is F1 * N.
```

Beispiel 14 (`nullstelle/4`)

Das Prädikat nullstelle/4 berechnet die Nullstelle einer Funktion. Als Parameter werden der Name F der Funktion, sowie Anfangswerte A und B mit $F(A) < 0$ *und* $F(B) > 0$ *übergeben. Funktionen werden als zweistellige Prädikate aufgefaßt, deren zweiter Parameter mit dem Argumentwert belegt wird, und die auf dem ersten Parameter den Funktionswert zurückliefern.*

```
nullstelle(X,F,A,B):-
    C is (A+B)/2, Call=..[F,Z,C], Call,
    (is_zero(Z), X=C;
     Z<0, nullstelle(X,F,C,B);
     Z>0, nullstelle(X,F,A,C)
    ),!.
is_zero(Z):-Z>=0.0, Z<1.0e-10.
is_zero(Z):-Z<0, Z> -1.0e-10.
f(Y,X):- Y is X**2+2*X-3.

?-nullstelle(X,f,0.6,1.8),write(X),nl.
```

7.3 Arithmetische Vergleiche (`</2,=</2,=:=/2,>=/2,>/2,=\=/2`)

HU-Prolog stellt sechs Prädikate (`</2`, `=</2`, `>/2`, `>=/2`, `=:=/2` und `=\=/2`) zum Vergleich arithmetischer Ausdrücke zur Verfügung. Alle Vergleichsprädikate sind zweistellig und werden als Infixoperator mit gleicher Priorität notiert. Bei Aufruf eines Vergleichsprädikates werden beide Terme wie bei `:=/2` ausgewertet und ihr Ergebnis numerisch verglichen. Wenn die Auswertung der Terme keine numerischen Werte liefert, führt das zum Abbruch der Abarbeitung: [1]

[1] Hinweis: die Standard-Vergleichsoperatoren sind in Prolog gerade so definiert, daß pfeilähnliche Konstrukte, wie `=>`, nicht vorbelegt sind und so dem Anwender zur freien Verfügung stehen.

$a \circ b$	ist erfolgreich, wenn der numerisch Wert von ...
`a=:=b`	... a gleich b ist,
`a=\=b`	... a ungleich b ist,
`a<b`	... a kleiner als b ist,
`a=<b`	... a kleiner oder gleich b ist,
`a>= b`	... a größer oder gleich b ist,
`a>b`	... a größer als b ist,

7.4 :=/2

Während der Prolog-Inferenzmechanismus auf *strenger Datenlokalität* und dem *single-assignment*-Prinzip beruht, sind für viele Anwendungen globale Ablauf- und Steuerinformationen wesentlich. Dies können Schaltervariablen sein, die globale Optionen beschreiben oder auch Attributinformationen für den Bildschirmaufbau oder den Zugriff zum Filesystem. Für die Einbettung derartiger Informationen gibt es in Prolog normalerweise zwei Wege.

Die "saubere" Prolog-Lösung besteht darin, die Attribut-Informationen als zusätzliche Parameter durchzureichen. Das erfordert jedoch die nachträgliche Modifikation der Schnittstellen praktisch aller Prozeduren und geht mit den Konzepten der *schrittweisen Verfeinerung* im Programmentwurf nur bedingt konform. Dennoch ist diese Herangehensweise vorteilhaft, wenn die Attributbestimmung selber als Teil des Berechnungsprozesses betrachtet werden soll, d.h. wenn Attributwerte nicht nur von außen nach innen, sondern auch umgekehrt weitergegeben oder geeignete Parametersetzungen (z.B. für das Layout beim Bildschirmaufbau) mit Hilfe des Backtracking-Mechanismus gefunden werden sollen.

Der typische Prolog "Hacker-Trick" benutzt dynamische Modifikationen der Datenbasis. Die Prädikate `assert/1`, `retract/1` und `clause/2` realisieren dabei das Hinzufügen, Löschen und Abfragen der Informationen aus der Datenbasis. Die Werte bleiben über Klauselgrenzen hinweg verfügbar und beim Backtracking erhalten.

Bei sorgfältigem (und trickreichem) Gebrauch dynamischer Prädikate kann man sehr effektive Programme schreiben. Diese von Standard-Prolog diktierte Herangehensweise vermischt jedoch die notwendige dynamische Modifikation des Programmcodes, eine wesentliche Voraussetzung für den Aufbau intelligenter Lösungen, mit dem einfachen Zugriff auf globale Daten und führt letztlich zu schwer lesbaren und fehleranfälligen Programmkonstruktionen.

HU-Prolog stellt für diese Anwendungsfälle globale Variablen bereit, die einen Kompromiß zwischen schneller Behandlung und sauberer Abgrenzung des Zugriffs auf globale Daten darstellen. Von ihrer äußeren Form her sind globale Variablen *Atome* oder beliebige, *strukturierte Terme*. Globale Variablen können beliebige Terme als Werte annehmen, die dann als Datenbasiseinträge über Klauselgrenzen hinweg erhalten bleiben. Die Grundidee besteht darin, den Wert einer n-stelligen globalen Variable als *Fakt* für ein gleichnamiges, $(n+1)$-stelliges Prädikat zu speichern, indem der aktuelle Wert der Variablen als erster Parameter zusätzlich eingefügt wird.

Der Operator `:=/2` stellt die Schnittstelle zur funktionalen und prozeduralen Komponente von HU-Prolog dar. Ein Aufruf "V := E" bewirkt zunächst die Auswertung des Ausdrucks E und anschließend die Zuweisung des Ergebnisses an V, wenn V eine globale Variable ist, oder die Unifikation des Ergebnisses mit V, falls V eine ganz normale Prolog-Variable ist.

Die Wertzuweisung `a:=0` erzeugt eine Datenbasiseintragung `a(0)`. Bei einem anschließenden `a:=a+1` wird zunächst das `a` auf der rechten Seite mit einem `call(a(X))` ausgewertet, was zu der Variablenbindung `X=0` führt. Mit dieser Variablenbindung wird die rechte Seite `X+1` ausgewertet und liefert als Resultatwert eine `1`. Nun wird die alte Datenbasiseintragung für `a` gelöscht und der neue Wert in der Form `a(1)` eingetragen.

```
?- a := 0.
yes
?- listing(a/1).
a(0).
yes
?- a := a + 1.
yes
?- listing(a/1).
a(1).
yes
?- X:= 2*a+1.
X = 3
yes
?-
```

Der Wert einer globalen Variablen ist bis zu dem Zeitpunkt, in dem die erste Wertzuweisung an sie stattfindet, undefiniert. Die Auswertung einer solchen globalen Variablen ergibt dann, ähnlich wie in LISP, die Variable selber.

Ähnlich verhält sich der Assign-Operator bei der Auswertung numerischer Standardfunktionen mit falschen Argumenten. Das Ergebnis einer solchen Auswertung ist der Funktionsaufruf selbst. Teilausdrücke, deren Berechnung möglich ist, werden jedoch normal ausgewertet.

```
?- X := 2 * 3 + cot(phi).
X = 6 + cot(phi)
yes
?-
```

Dieser Ansatz lässt sich einfach auf strukturierte globale Variable übertragen. Damit sind z.B. Variablenfelder oder Datenstrukturen realisierbar. Der Wert einer n-stelligen globalen Variablen wird in mehreren (n+1)-stelligen Fakten in der Datenbasis abgelegt. Und zwar genaugenommen für jede belegte Komponente mit einem Fakt: Die Wertzuweisung `f(1,2):=5` bewirkt z.B. das Streichen aller alten Datenbasiseintragungen für `f(_,1,2)` und das Eintragen des (neuen) Faktes `f(5,1,2).` am Anfang der Datenbasis. Die Werte der globalen Variablen werden immer im ersten Argument der entsprechenden $(n+1)$-stelligen Fakten gespeichert.

```
?- stueckzahl(schraube(m6)):= 3.
yes
?- stueckzahl(mutter(m6)):= 4.
yes
?- listing(stueckzahl).
stueckzahl(4,mutter(m6)).
stueckzahl(3,schraube(m6)).
yes
?-
```

Auch der Zugriff auf strukturierte Variablen verläuft analog. Der Aufruf `X:=f(a1,...,aN)` bewirkt zunächst die Konstruktion eines Aufrufterms `f(Y,a1,...,aN)`, der über `call/1` ausgewertet wird. Ist der Aufruf erfolgreich, so geht `Y` in die weitere Berechnung ein, anderenfalls wird die Teilformel `f(a1,...,aN)` ungeändert übernommen.

Zusammenfassend kann man globale Variablen in HU-Prolog wie folgt charakterisieren:

1. Globale Variablen der Form $f(x_1, ..., x_n)$ werden in Klauseln der Form $f(y, x_1, ..., x_n)$ gespeichert. $y, x_1, .., x_N$ sind dabei beliebige Terme.
2. Der Zugriff auf eine globale Variable $f(x_1, ..., x_n)$ erfolgt durch einen gewöhnlichen Prolog-Aufruf mittels `call`$(f(Y, x_1, ..., x_n))$, wobei die Belegung von Y als Funktionswert zurückgegeben wird.

3. Eine globale Variable ist genau dann undefiniert, wenn der zugehörige Aufruf fehlschlägt. Der Wert der Variablen ist dann die Variable selbst.

4. Bei Wertzuweisung an eine globale Variable werden alle auf diese Variable *genau passenden* Datenbasiseintragungen gestrichen und eine die neue Belegung widerspiegelnde Eintragung am Anfang der Datenbasis hinzugefügt.

Beispiel 15 (`e/2`)

Durch die Verwendung von freien Variablen als Selektor ist die Einfügung von Default-Werten für strukturierte Variablen möglich, wie z.B. bei der Abspeicherung schwachbesetzter "verunreinigter" Einheitsmatrizen:

```
?- e(X,Y):=0.
yes
?- e(X,X):=1.
yes
?- e(2,3):=3.141592.
yes
?- listing.
e(0.3141592e+01,2,3).
e(1,X,X).
e(0,_,_).
yes
?-
```

7.5 Elementare symbolische Funktionen

HU-Prolog unterstützt neben den in C-Bibliotheken üblichen numerischen Funktionen die Verarbeitung symbolischer Terme. Doch damit treten altbekannte Probleme auf: Wie unterscheidet man zwischen einer Wertzuweisung, die einer Variablen `x` den Term "`n`" zuweist, und einer Wertzuweisung, die `x` den aktuellen Wert von "`n`" zuweist? In Anlehnung an LISP wurden in HU-Prolog daher die Funktion Quote (`` `/1 ``) eingeführt, die ihr Argument unverändert weitergibt, d.h. die weitere Auswertung des "gequoteten" Teilterms unterdrückt.

```
?- X:= 2+3.
X = 5
yes
?- X:= `(2+3).
X = 2+3
yes
?-
```

7.6 Nutzerdefinierte Funktionen

Für die Auswertung von Funktionen und globalen Variablen ist es unerheblich, ob die entsprechenden Klauseln durch eine Wertzuweisung, `assert/1` oder `consult/1` in die Datenbasis gelangt sind. Der synthetisierte Aufruf muß lediglich als erstes Argument einen Term zurückliefern, der in die weitere Auswertung eingeht. Das bietet einen einfachen Mechanismus zur Definition von Funktionen in HU-Prolog. Wesentlich ist dabei, daß für diese Art nutzerdefinierter Funktionen die Art und Reihenfolge der Parameterauswertung vollkommen offen ist. Sie wird vom Nutzer mit der Definitionen der Funktion bzw. des charakteristischen Prädikats explizit angegeben.

Bei einem *call-by-value*-Parameterübergabeschema werden zunächst alle aktuellen Argumentwerte ausgewertet und die Ergebnisse verknüpft:

Beispiel 16 (`cadr/1`)

In Anlehnung an LISP definiert man die Funktionen car/1 und cdr/1 durch folgendes Prädikat:

```
car(Y, X):- Z:= X, Z=[Y|_].
cdr(Y, X):- Z:= X, Z=[_|Y].
```

Beispiel 17 (`!/2`)

n! liefert den Wert der Fakultätsfunktion an der Stelle n. Bei einem Aufruf von n! wird zunächst n berechnet, was im allgemeinen eine Zahl N liefern wird. Mit dieser Zahl startet die eigentliche Auswertung des Fakultätsprädikats, dessen Ergebnis als Funktionswert zurückgegeben wird. (Für die Definition des Prädikats fak/2 siehe Seite 41)

```
!(Y,X):- Z:=X, fak(Y,Z).

?- op(50,yf,!).

?- n:=2.
yes
?- X:= (2*n+1)! .
X = 120
yes
?- X:= 3!!.
X= 720
yes
```

Für manche Funktionen ist eine *call-by-need*-Strategie wesentlich sinnvoller. Hier werden Parameterwerte nur ausgewertet, wenn sie effektiv benötigt werden.

Beispiel 18 (`if/3`)

if(c,x,y) beschreibt einen bedingten Ausdruck analog zu (c ? x : y) in der Sprache C; in Abhängigkeit von der Bedingung `c` *wird entweder* `x` *oder* `y` *ausgewertet.*

```
if(Z,C,X,_):-C,!,Z:=X.
if(Z,_,_,Y):-Z:=Y.
```

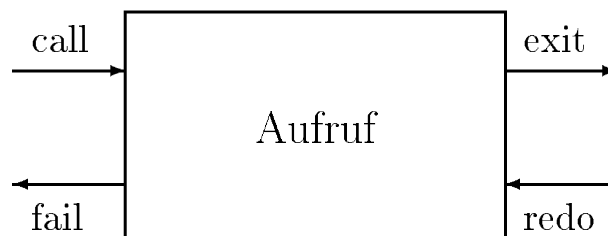
Kapitel 8

Ablaufsteuerung

Die Ablaufsteuerung eines Prolog-Programms kann man sich am besten anhand des Box-Modells, einer Verallgemeinerung der bekannten Steuerflußbilder auf Sprachen mit *backtracking*, veranschaulichen. Dieses Box-Modell liegt dabei auch wesentlich den *trace*-Hilfsmitteln für den dynamischen Programmtest zugrunde.

8.1 Das Box-Modell

Ein Prolog-Prädikataufruf verallgemeinert den klassischen Prozeduraufruf mit einem Eintrittspunkt und einem Austrittspunkt: Es gibt auch für ein Prolog-Prädikat einen *Haupt*eintrittspunkt (*call-port*), durch den die Abarbeitung beim Prädikataufruf beginnt und einen endgültigen *Haupt*austrittspunkt (*fail-port*), der allerdings nur beim definitiven Fehlschlagen des Prädikats passiert wird. Interessant wird es dazwischen, wenn es um die Implementierung nichtdeterministischer Suchalgorithmen mittels Backtracking geht. Prolog-Prädikate besitzen für diesen Fall einen temporären *Neben*ausgang (*exit-port*), mit dessen Passieren *eine* Lösung des Suchproblems durch Variablen-Instantiierung nach außen gegeben wird. Wenn diese Teillösung vom nachfolgenden Aufruf verworfen wird, so erfolgt ein *Backtracking* durch den *Neben*eingang (*redo-port*) zurück in das immernoch aktive Prädikat. Hier wird nun unter Umständen eine weitere Lösung generiert, die durch erneutes Verlassen des Prädikates über den *exit-port* zur Prüfung nach außen weiter gegeben wird, bis dann nach mehrmaligen Durchlaufen der Backtrack-Schleife keine alternativen Lösungen generiert werden können. In diesem Fall schlägt das Prädikat endgültig fehl, das Prädikat wird über den *Haupt*ausgang, den *fail-port*, verlassen.

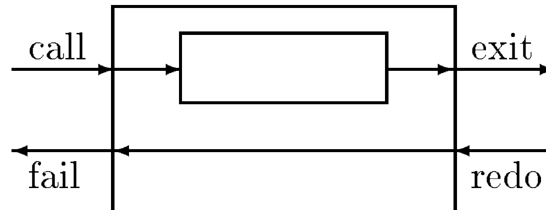


Beim Vergleich mit klassischen Sprachen ist es also wesentlich, davon auszugehen, daß beim "normalen" Verlassen eines Prolog-Prädikats über den *exit-port* das Prädikat aktiv bleibt. Alle lokalen Variablen und der lokale Berechnungszustand werden im Kellerspeicher abgelegt. Erst beim endgültigen Verlassen, d.h. also beim Fehlschlagen des Prädikats, wird der Kellerspeicher bereinigt, so wie wir es beim Verlassen einer klassischen Prozedur kennen.

Aus prozeduraler Sicht lassen sich im Box-Modell folgende Spezialfälle unterscheiden:

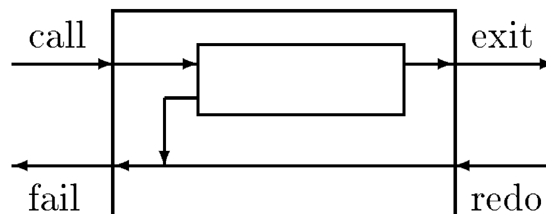
- **Operationen**

  Operationen sind Prädikate, die immer erfolgreich sind und jeweils *genau eine* Lösung generieren, so daß sie für das Backtracking transparent sind. Häufig ist der Seiteneffekt das eigentlich Wesentliche an einer Operation, wie z.B. bei der Ein- und Ausgabe.



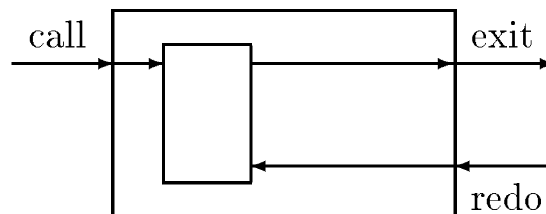
- **Tests**

  Tests sind Prädikate, die für jeden Aufruf höchstens eine Lösung liefern, häufig sogar ohne weitere Instantiierung beteiligter Variablen. Für das Backtracking sind Tests transparent.



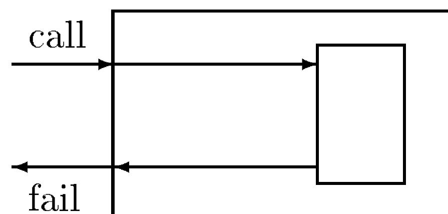
- **Generatoren**

  Generatoren sind Prädikate, die bei jedem Aufruf unendlich viele Lösungen generieren. Charakteristisch für Generatoren ist, daß sie auch beim Backtracking immer wieder erfolgreich sind. Der *fail-port* wird also niemals benutzt.



- **Terminatoren**

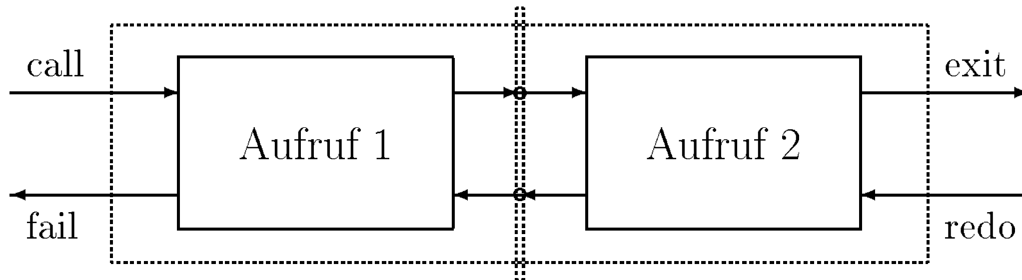
  Terminatoren sind Prädikate, die niemals eine Lösung liefern und stets fehlschlagen. Sie bilden den Abschluß von Aufrufketten und erzwingen über Backtracking die Generierung aller Lösungen der vorangehenden Prädikate. Im Zusammenspiel mit Generatoren dienen sie der Organisation unendlicher Schleifen.



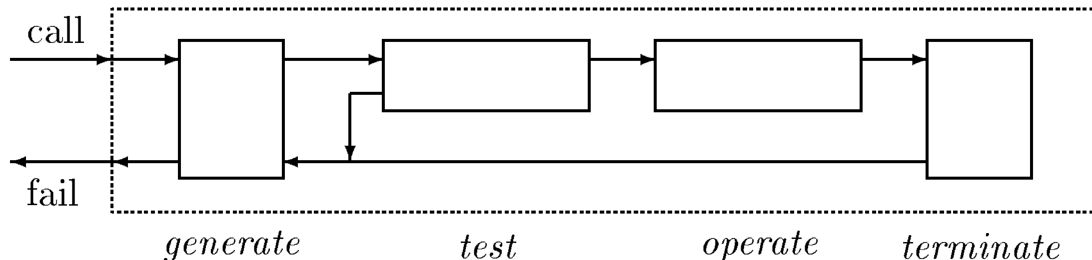
8.2 ,/2

Prolog-Aufruffolgen entstehen durch sequentielle Verknüpfung mit dem `,/2`-Operator[1]. Das Prädikat `,/2` entspricht damit logisch der *Konjunktion*. Es tritt normalerweise nur im Regelkörper auf, manchmal aber auch bei der Metaprogrammierung. Das Prädikat `,/2` ist transparent bezüglich des Cut-Operators, ein Cut innerhalb einer Sequenz bezieht sich immer auf das umgebende Prädikat.

Bei der sequentiellen Verknüpfung zweier Aufrufe, beschrieben durch das Prädikat `,/2`, werden die beiden Boxen ähnlich einem Vierpol zusammengeschaltet:



Auf diese Weise entstehen die für die Prolog-Programmierung typischen *generate-and-test* Programmstrukturen: Der erste Aufruf generiert Lösungskandidaten, der nachfolgende Aufruf filtert die tatsächlich in Frage kommenden Kandidaten heraus. Aufruffolgen, die mit einem Terminator abgeschlossen werden, verhalten sich wie Prozeduraufrufe in klassischen Sprachen: Wenn das Problem vollständig gelöst ist, wird der Laufzeitkeller zurückgesetzt und die Prozedur verlassen. Würde man die ganze Aufruffolge als eine Box betrachten, so verhielte sie sich nach außen wie ein Terminator. Ein Schlüssel für die *inkrementelle Transformation* einer *Prototyplösung* in ein *Produkt*, das industriellen Anforderungen genügen soll, liegt häufig in der Umwandlung der *top-level* Prädikate in Terminatorform:

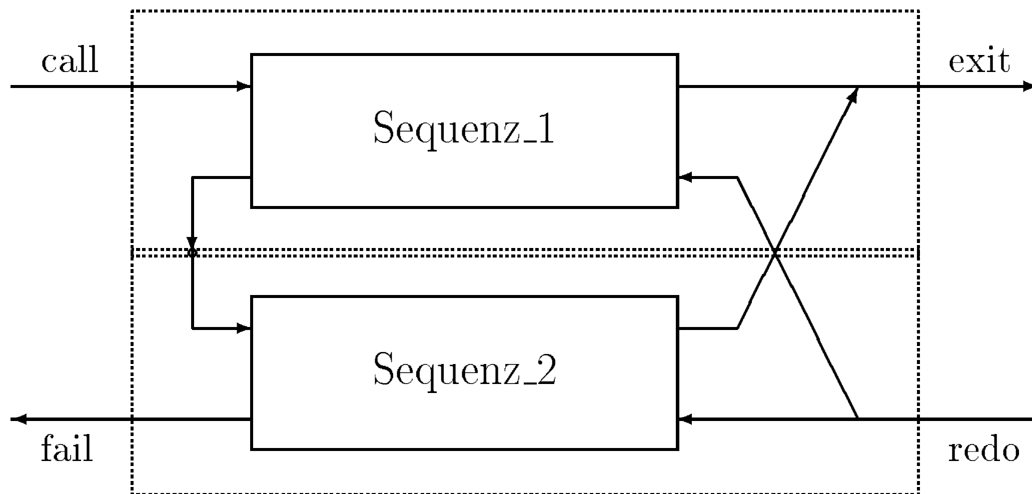


8.3 ;/2

Das Prädikat `;/2` beschreibt Alternativen in der Abarbeitungsfolge und entspricht damit logisch der *Disjunktion*. Der Operator `;/2` tritt normalerweise nur im Regelkörper auf sowie bei der Gestaltung von Schleifen mit Hilfe des Backtracking.

Zunächst werden die links vom Semikolon stehenden Aufrufe abgearbeitet. Sind diese erfolgreich, so ist damit bereits eine Lösung gegeben. Beim Backtracking werden dann zunächst alle Lösungen des linken Zweiges generiert. Erst wenn der linke Zweig keine weiteren Lösungen liefert, werden die Aufrufe rechts vom Semikolon abgearbeitet. Das Prädikat `;/2` ist transparent bezüglich des Cut-Operators. Ein Cut innerhalb einer Alternative bezieht sich immer auf das umgebende Prädikat.

[1] Das separierende Komma zwischen den Argumenten eines Funktorterms ist aber kein Operator. Um die syntaktische Eindeutigkeit zu gewährleisten, ist festgelegt, daß die Argumente eines Funktorterms kleinere Priorität haben müssen als der Kommaoperator. Das heißt, Operatorterme mit größerer Priorität müssen explizit geklammert werden.



Beispiel 19 (`;-Elimination`)

Gerade das rapid-prototyping verleitet zur extensiven Verwendung des `;/2`*-Operators. Es ist jedoch als schlechter Programmierstil anzusehen und häufig ein Indiz unnötig komplizierter Programmstruktur. Man sollte daher im Zuge der schrittweisen Transformation eines Programms from-prototype-to-product versuchen, den Operator* `;/2` *soweit als möglich zu eliminieren. Dazu werden zunächst eventuell eingeklammerte Aufrufe von* `;/2`*, sofern sie keine Cut-Aufrufe enthalten, durch ein Hilfsprädikat in disjunktiver Normalform ersetzt:*

```
q:-x,(a,b;c,d,e),y.
```

wird zu

```
q:-x,p,y.
p:-a,b; c,d,e.
```

Das Prädikat `p` *wird dann wiederum äquivalent ersetzt durch:*

```
p:-a,b.
p:-c,d,e.
```

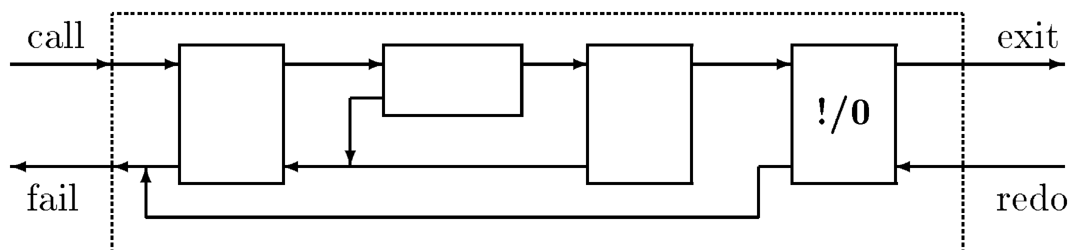
8.4 `->/2`

Das `->/2` Prädikat beschreibt eine eingeschränkte Form der logischen Folgerung. Wenn die Bedingung links vom `->` erfüllt ist, dann wird die Aktion rechts vom `->` abgearbeitet, wenn die Bedingung nicht erfüllt ist, schlägt das Prädikat fehl. In Verbindung mit dem `;/2`-Operator gestattet `->/2` die Konstruktion bedingter Aufrufe `(Cond -> Then ; Else)`: Die Bedingung `Cond` wird zunächst ausgewertet und entsprechend dem Ergebnis weiter verzweigt. Ist die Bedingung erfolgreich, so wird die Abarbeitung im `Then`-Zweig fortgesetzt, anderenfalls im `Else`-Zweig. Beim Backtracking werden dann jeweils alle Lösungen des `Then`- bzw. des `Else`-Zweiges generiert, bevor der bedingte Aufruf insgesamt fehlschlägt. Ein Backtracking in die Bedingung hinein bzw. vom `Then`- in den `Else`-Zweig ist nicht möglich.

Die if-then-else-Konstruktion wird häufig während des Debugging-Prozesses eingesetzt, als "*schneller Hack*", ist jedoch potentiell sehr fehleranfällig, weil sie konzeptionell mit dem Berechnungsmodell von Prolog nicht verträglich ist. De facto sucht man länger Folgefehler, als man durch den Hack eingespart hat. Obendrein gibt es ehrhebliche Auslegungsunterschiede beim `->/2`-Prädikats zwischen verschiedenen kommerziellen Prolog-Systemen. Der Autor empfielt daher, den Operator `->/2` strikt zu vermeiden.

8.5 !/0

Der Cut-Operator `!/0` ist das wichtigste Hilfsmittel zur Steuerung des Backtrackverhaltens von Prolog-Prädikaten. Durch den Aufruf von '!' innerhalb eines Prädikats werden mögliche alternative Lösungen dieses Prädikats, die aus dem bisherigen Berechnungsprozess resultieren, "abgeschnitten". Dadurch kann zum Beispiel im Laufzeitkeller der Speicherplatz für die Zustandsvariablen von Aufrufen eingespart werden, die nun über das Backtracking nicht mehr erreichbar sind. Im Box-Modell stellt sich das so dar, daß der *fail-port* von Cut immer direkt in den *fail-port* des umgebenden Prädikats führt. Der Cut-Operator ist also bezüglich des normalen Aufrufs transparent, wirkt aber beim Backtracking wie eine Abkürzung (*engl. short cut*).



Der Cut-Aufruf läßt sich damit zur Effektivierung von Prolog-Programmen einsetzen, ohne daß sich deren Semantik ändert, wenn die abgeschnittene Lösungsmenge leer ist: eine Eigenschaft, die sich nicht automatisch herleiten läßt. Ein Cut-Aufruf kann aber auch die Semantik eines Prädikats wesentlich prägen. Als letzter Aufruf eines Prädikats angewandt, erzwingt Cut das deterministische Verhalten des Prädikats. In Verbindung mit `fail/0` erlaubt der Cut-Aufruf das zwangsweise Fehlschlagen eines Prädikats. Die Prädikate `,/2`, `;/2` und `->/2` sind transparent bezüglich Cut-Anwendung, d.h. ein Cut in einem Zweig von `,/2`, `;/2` bzw. `->/2` wirkt auf den umgebenden Aufruf.

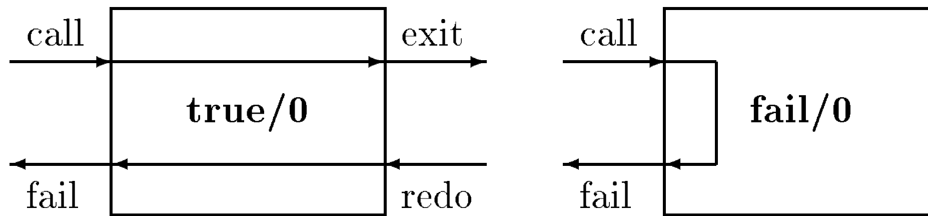
Beispiel 20 (`!/1`)

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

```
?- private([exec,solution]).
!(X):-exec(X),fail.
!(X):-retract(solution(X)),!.
exec(X):-X,!,asserta(solution(X)).
?- hide([exec,solution]).
```

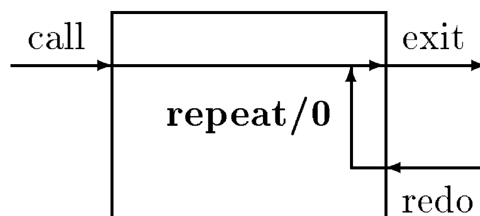
8.6 true/0 und fail/0

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



8.7 repeat/0

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



Beispiel 21 (`ask/0`)

Das Prädikat ask/0 erwartet eine Nutzerentscheidung (y/n). Es ist erfolgreich, wenn der Nutzer 'y' eingegeben hat, schlägt fehl, wenn der Nutzer 'n' eingegeben hat, und wiederholt seine Eingabeaufforderung bei allen anderen Nutzerantworten. Die hier verwendete Programmstruktur (three-way-branching) ist ein Beispiel für den notwendigen und zweckmäßigen Gebrauch von ;/2. Hier ist der ;/2-Operator die natürliche Problemlösung:

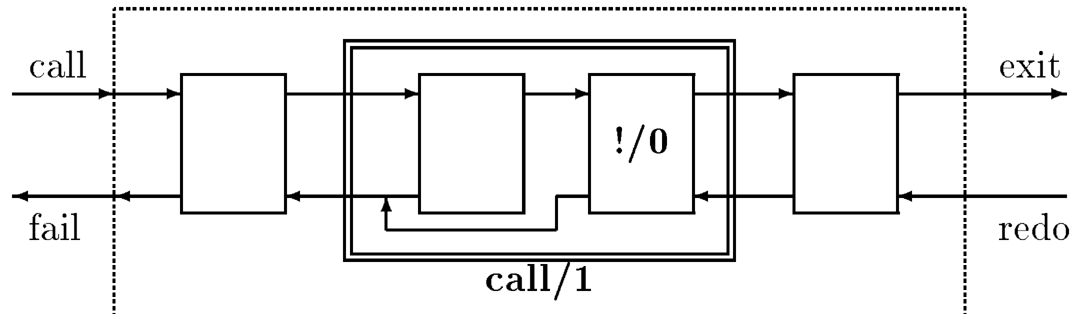
```
ask:-repeat,
       write(' (y/n) '),
       get(X),
       ( [X] = "y" ,!;
         [X] = "n" ,!,fail;
         write('please reply '),fail
       ).
```

8.8 call/1

Aufrufe innerhalb eines Prolog-Prädikats können dynamisch während der Abarbeitung erzeugt werden. Dafür gibt es zwei Wege: die Verwendung einer Prolog-Variablen in der Aufruffolge, die zur Laufzeit mit einem ausführbaren Aufruf instantiiert sein muß, und der Metaaufruf mittels `call/1`. Der direkte Aufruf einer Prolog-Variablen ist dem (nur selten korrekt implementierten) *call-by-name* in klassischen Sprachen vergleichbar. Der Aufrufterm wird zur Ausführungszeit in dem konkreten Kontext interpretiert und kann die Ablaufsteuerung innerhalb des rufenden Prädikats grundsätzlich beeinflussen, beispielsweise wenn die Variable an `!` oder `(!,fail)` gebunden wird.

Der Metaaufruf `call(X)` hingegen wirkt so, als ob temporär ein neues Prädikat (mit den aktuellen Parametern `X` als Körper) definiert und sofort abgearbeitet würde. Bei normalen Anwendungen ist der Unterschied in der Semantik nicht spürbar. Der Vorteil der Verwendung von `call/1` liegt jedoch darin, daß die Ablaufsteuerung des rufenden Prädikats nicht gestört wird: Ein Metaaufruf kann nur erfolgreich sein oder fehlschlagen. Die Wirkung des Cut wird durch den Metaaufruf 'gekapselt'. Einige

kommerzielle Systeme lassen daher das direkte Auftreten von Variablen im Körper einer Klausel nicht zu oder interprerieren es immer als `call(Variable)`.



Semantik (`call/1`)

```
call(X):-X.
```

8.9 not/1

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

Semantik (`not/1`)

```
not X :- call(X), !, fail.
not _ .
```

Beispiel 22 (`?/1`)

Einen Test, ob ein Aufruf erfolgreich ist, ohne die aus diesem Aufruf resultierenden Unifizierungen tatsächlich auszuführen, ergibt sich mit Hilfe der doppelten Negation:

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

Kapitel 9

Globale Steuerung

Die Arbeitsweise des HU-Prologsystems läßt sich durch folgenden Algorithmus beschreiben:

1. Laden und Starten des Programms `prolog`. Zu diesem Zeitpunkt sind nur die in dieser Dokumentation beschriebenen built-in-Prädikate sowie eventuell eincompilierte nutzerdefinierte Prädikate bekannt.

2. Wenn in der Kommandozeile mit der `-S-Option` ein Quellfile angegeben wurde, so wird dieses im Systemmodus konsultiert. Anderenfalls prüfe, ob daß bei der Übersetzung von HU-Prolog angegeben globale `prologrc`-File vorhanden ist und lade dieses. Unter U*IX ist das in der Regel '/usr/local/lib/huprolog/prologrc'.

3. Wenn in der Kommandozeile mit der `-s-Option` ein Quellfile angegeben wurde, so wird dies im Systemmodus konsultiert. Anderenfalls prüfe, ob im *aktuellen Directory* ein `prologrc`-File existiert, wenn ja, konsultiere dieses File. Anderenfalls prüfe, ob sich dieses File in dem durch die Environmentvariable $HOME bezeichnetem Verzeichnis befindet.

4. Wenn in der Kommandozeile ein weiteres Quellfile angegeben wurde, so wird dieses jetzt geladen.

5. Wenn ein `login/0`-Prädikat definiert ist, generiere einen einmaligen Aufruf von `login/0`.

6. Solange das System nicht explizit mit einem Aufruf von `end/0`, `halt/0` oder `exit/0` verlassen wird, bleibe in der Hauptsteuerschleife:

   (a) Wenn ein Prädikat `toplevel/0` definiert ist, rufe es auf und kehre anschließend zur Hauptsteuerschleife zurück.

   (b) Wenn ein Prädikat `prompt/0` definiert ist, rufe es auf, ansonsten schreibe das Systemprompt "`?- `" auf den aktuellen Ausgabestrom und verfahre weiter entsprechend dem Prolog-Standardtoplevel: Lese einen Term vom aktuellen Eingabestrom und interpretiere ihn mit `call/1`.

   (c) Wenn eine Lösung gefunden wurde, protokolliere diese und warte auf Nutzerantwort. Bei Eingabe von "`;`" versuche eine neue Lösung zu generieren. Bei Eingabe von `<ET>` akzeptiere die Lösung.

   (d) Wenn die Lösung als endgültig akzeptiert wurde oder (trotz positiven Ausgangs) keine Variablenbelegung generiert wurde, antworte mit "`yes`" und kehre zur Hauptsteuerschleife zurück.

   (e) Wenn keine (weitere) Lösung existiert, antworte mit "`no`" und kehre zur Hauptsteuerschleife zurück.

   (f) Wenn während der Systemarbeit Fehler oder Interrupts auftreten, so kann man diese durch nutzerdefinierte Fehlerbehandlungs- bzw. Interruptprädikate behandeln.

7. Wenn das Prädikat `logout/0` definiert ist, generiere einen einmaligen Aufruf von `logout/0`.
8. Verlasse das System unter Rückgabe des eventuell über `exit/1` gesetzten Returncodes.

Durch das Definieren bzw. gewollte Nicht-Definieren einiger Prädikate kann man das Systemverhalten in weiten Grenzen ausgestalten, so daß sich komplette Anwendungslösungen in Prolog definieren lassen. In diesem Kapitel werden die globalen Systemsteuerungs-Prädikate beschrieben.

9.1 login/0 und logout/0

Die Prädikate `login/0` und `logout/0` werden, falls sie definiert sind, jeweils genau einmal aufgerufen und erlauben die Systeminitialisierung bzw. -terminierung auf Prolog-Niveau.

9.2 toplevel/0 und prompt/0

HU-Prolog besitzt ein eingebautes, interaktives Toplevel das die Kommunikation mit dem Nutzer regelt. Dieses wurde in der Einleitung zu dieser Dokumentation beschrieben. Das Prädikat `toplevel/0` erlaubt nun dem Nutzer, sein eigenes Toplevel zu schreiben. Wenn in der Datenbasis Klauseln mit dem Hauptfunktor `toplevel/0` vorhanden sind, werden diese immer dann aufgerufen, wenn das normale Toplevel von Prolog aktiviert werden würde.

Wenn für die Systemsteuerung das Standard-Toplevel benutzt wird, so kann man zumindest mit dem `prompt/0`-Prädikat sein eigenes Prompt ausgeben.

Semantik (`toplevel/0`)

Mit folgenden Klauseln kann man das Standard-Toplevel nachbilden:

```
?- private([process,write_prompt,solutions]).
toplevel :-
    write_prompt,
    warn(Oldwarn),      % warn-Flag merken
    warn(off),          % Warnungen ausschalten
    read(Term,Varlist), % Nutzereingabe einlesen
    warn(Oldwarn),      % warn-Flag zuruecksetzen
    process(Term,Varlist).

process(end,_):-exit(0).
process(Term,[]):-call(Term),write(yes),nl,restart.
process(Term,[X=Val|VarList]):-
    call(Term),solutions([X=Val|VarList]),write(yes),nl,restart.
process(_,_):-write(no),nl,restart.

write_prompt:-current_predicate(prompt/0),call(prompt),!.
write_prompt:-current_predicate(prompt/0),!.
write_prompt:-write('?- ').

solutions([X=Val]):-write(X=Val),!,not ask(59).
solutions([X=Val|VarList]):-write(X=Val),nl,solutions(VarList).
?- private([process,write_prompt,solutions]).
```

9.3 exit/1, halt/0 und end/0

Das Prädikat `exit/1` führt zum direkten Verlassen des Prolog-Systems. Dabei wird an die aufrufende Shell der Exit-Code zurückgegeben, der im Argument von `exit/1` angegeben wird. Dieses Argument muß ein arithmetischer Ausdruck sein, der eine ganze Zahl zwischen 0 und 255 liefert. Ein Aufruf von `halt` ist äquivalent zu `exit(0)`. Nach Abarbeitung des Prädikats `end/0` wird nur noch der aktuelle Aufruf zu Ende abgearbeitet. Sowie die Hauptsteuerschleife des Prolog-Systems die Steuerung zurückerhält, wird das System mit `halt/0` verlassen. `end/0` bewirkt also ein verzögertes Verlassen des Prolog-Interpreters. Zu beachten ist, daß die `end/0` Behandlung im ggf. im nutzerdefinierten `toplevel/0` selber vorgenommen werden muß.

9.4 abort/0 und restart/0

Das Prädikat `abort/0` erzeugt eine Fehlermeldung und beendet die laufende Abarbeitung. Der Interpreter kehrt auf das Toplevel zurück. Das Prädikat `restart/0` beendet ebendso die laufende Abarbeitung, erzeugt aber keine Fehlermeldung. Dies ist zum Beispiel bei einer eigenen Fehlerbehandlung oder nutzerdefiniertem `toplevel/0` sinnvoll.

9.5 interrupt/0

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

Beispiel 23 (`interrupt/0`)

```
interrupt :-
    ttynl, ttywrite(‘interrupt:’),ttynl,
    repeat,
      ttywrite(’[a]bort/[t]race/[c]ommand :’),
      ttyget(Char),
      ( [Char] == "a’,abort
      ; [Char] == ‘t’,ttyskip(10),trace(on)
      ; [Char] == ‘c’,ttyread(Command),call(Command),fail
      ),!.
```

9.6 error/2

Die meisten Fehlermeldungen, die zum Abbruch des laufenden Programms führen, können in einem Anwendungsprogramm abgefangen werden. Ausgenommen sind Fehlermeldungen wegen Speicherplatzproblemen des Prolog-Systems.

Wenn in der Datenbasis Klauseln mit dem Namen `error/2` definiert sind, wird das Prädikat `error/2` bei einem Fehler aufgerufen. Als erstes Argument erhält es den Aufruf, bei dem der Fehler auftrat, als zweites die Fehlernummer. Wenn `error/2` erfolgreich abgearbeitet wurde, wird danach die Abarbeitung normal fortgesetzt. Anderenfalls wird ein Backtracking ausgelöst.

Beispiel 24 (`error/2`)

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

Fehler-Nr.	Fehlernachricht
1	`‘execution aborted’`
2	`’unsuitable argument(s) to system predicates’`
3	`’out of atom space’`
4	`’arity of functor out of range’`
5	`‘probably malformed ’,..’’`
6	`‘character value out of range’`
7	`‘closing bracket missing’`
8	`‘malformed expression’`
9	`’unmatched closing bracket’`
10	`‘bad numerical argument type ’`
11	`’unsuitable arguments to ‘call’’`
12	`’unterminated comment’`
13	`‘nesting too deep probably cyclic term’`
14	`‘division or mod by zero’`
35	`‘floating point error’`
15	`’unexpected end of file’`
16	`’out of frame space’`
17	`‘I/O error’`
18	`’out of local stack space’`
19	`‘infix or postfix operator expected’`
20	`‘closing quote expected’`
21	`’operand or prefix operator expected’`
22	`‘bad number syntax’`
23	`’out of variable table space’`
24	`’operator has unsuitable precedence’`
25	`‘goal failed during program input’`
26	`‘nesting too deep in input’`
27	`‘read stack overflow’`
28	`‘function called with wrong argument(s)’`
29	`’accessing or modifying system procedures’`
30	`’out of trail space’`
31	`’undefined function in expression’`
32	`’out of variable name space’`
33	`‘illegal character in input’`
34	`’out of string space’`
36	`‘can‘t create file’`
37	`‘can‘t open file’`
38	`‘file is not open’`
39	`‘file is a tty’`
40	`‘to many files’`
41	`‘file is current outputfile’`
42	`‘file is current inputfile’`
43	`‘file is only open for output’`
44	`‘file is only open for input’`
48	`‘bad magic number’`
49	`‘checksum error’`

9.7 unknown/1

Wenn in Prolog ein Prädikat abgearbeitet werden soll, das weder ein eingebautes noch ein vom Nutzer definiertes Prädikat ist, schlägt dieser Aufruf normalerweise fehl. Wenn allerdings vom Nutzer das Prädi-

kat `unknown/1` definiert ist, wird stattdessen dieses aufgerufen. Als Argument erhält `unknown/1` den ursprünglichen Aufruf. Damit kann sich der Nutzer die Reaktion auf nicht vorhandene Prädikate selbst definieren, zum Beispiel ein Konzept des dynamischen Nachladens von Prädikaten.

Beispiel 25 (`unknown/1`)

```
library('./').                              % Spezifikation von
library('/home/dziadzka/lib/huprolog/'). % Prologbibliotheken, die
library('/usr/local/bin/huprolog/').     % durchsucht werden
unknown(Call) :-
    functor(Call,Name,Arity),
    library(LIB),
    ensure(LIB,Name,Arity),
    % es wird versucht, Klauseln nachzuladen
    % siehe Modulkonzept
    !,
    Call.
unknown(Call) :-
    functor(Call,Name,Arity),
    ttywrite('Warning: no clause for relation '),
    ttywrite(Name / Arity),
    ttynl,
    !,fail.
```

9.8 ancestors/1

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

9.9 sys/1

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

9.10 dict/1 und sdict/1

Ein Aufruf von `sdict/1` unifiziert das Argument mit der Liste aller Namen von Systemprädikaten in der Form `Atom/Arity`. `dict/1` liefert dazu komplementär die Liste aller nutzerdefinierten sowie der implizit benutzten Funktoren.

Kapitel 10

Programmierumgebung

10.1 Das Modul-Konzept von HU-Prolog

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

10.1.1 private/1

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

10.1.2 hide/1

Dieses Prädikat erwartet als Argument ein Atom oder eine Liste von Atomen. `hide/1` kennzeichnet das Ende des Sichtbarkeitsbereiches dieser Atome. Die bisher sichtbaren angegebenen Atome erhalten einen neuen Namen, und die vorher mit `private/1` verdeckten Atome werden wieder sichtbar.

Wenn `hide/1` auf ein Atom auf dem obersten Deklarationsniveau angewendet wird, wird auch dieses versteckt. Es ist also möglich, Systemfunktionen umzudeklarieren, indem man das vordefinierte Prädikat zuerst mit `hide/1` versteckt und anschliessend neu definiert.

Beispiel 26 (`rand/1`)

Das Prädikat rand(X) ist ein Pseudozufallszahlengenerator, der über Backtracking Zufallszahlen im Bereich $0..2^{15}-1$ *liefert. Der Algorithmus entspricht der UNIX-C-Funktion rand(3C). Der Modul enthält eine versteckte, lokale Zustandsvariable seed.*

```
?-private(seed).
?-seed :=  1.
random(X) :-
   repeat,
     seed := seed * 1103515245 + 12345,
     X := (seed // 65536) mod 32768.
?- hide(seed).
```

10.1.3 ensure/3

Dieses Prädikat ermöglicht das sinnvolle Aufbauen von Bibliotheken, indem es das mehrfache Laden von Files verhindert. Das Prädikat `ensure/3` erwartet als Argumente zwei Atome und eine Zahl. Das erste Argument muß der Namen eines Verzeichnisses sein, das zweite der Name eines Atoms und das dritte die Stellenzahl des Atoms. Wenn bei einem Aufruf `ensure(Lib,Name,Arity)` auf das zweite und dritte Argument bereits einmal erfolgreich angewendet wurde, so ist `ensure/3` ohne weiteres erfolgreich. Sonst wird aus den drei Argumenten ein Filename der Form `'Lib/Name.Arity'` gebildet. Wenn dieses File nicht existiert, schlägt `ensure/3` fehl. Sonst wird das File mit `reconsult/1` eingelesen, und `ensure/3` wird wahr.

`ensure/3` liest also ein bestimmtes File höchstens einmal ein.

10.2 Die Schnittstelle zur Systemumgebung

10.2.1 date/3, time/3 und weekday/1

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

10.2.2 timer/1

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

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

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

10.2.3 getenv/2 und putenv/2

Das Prädikat `getenv/2` erlaubt den lesenden Zugriff zu Umgebungsvariablen. Das zweite Argument muß der Name der Variable sein. Das erste Argument wird mit dem aktuellen Wert der Umgebungsvariable unifiziert. Das Prädikat `putenv/2` erlaubt den schreibenden Zugriff zu den Umgebungsvariablen. Beide Argumente müssen mit Atomen belegt sein. `putenv/2` setzt die durch sein zweites Argument bezeichnete Umgebungsvariable auf den durch das erste Argument spezifizierten Wert.

```
?- getenv(Editor,'EDITOR').
Editor = /usr/bin/vi
yes
?-
```

10.2.4 system/1

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

Beispiel 27 (`edit/1`)

Aufruf des durch die Environment Variable 'EDITOR' spezifizierten Editors (z.B. vi) und anschließendes Laden des Files. Beim Aufruf ohne Argument wird das zuletzt editierte File noch einmal editiert.

```
?-private(file).
edit(Filename) :-
    abolish(file,1),
    assert(file( Filename )),
    gentenv(Editor,'EDITOR'),
    system([Editor, Filename ]),
    reconsult( Filename ).
edit :- retract(file( Filename )), edit(Filename).
?-hide(file).
```

10.2.5 argc/1 und argv/2

Diese Prädikate erlauben den Zugriff auf die Kommandozeilenargumente des Interpreters, ahnlich wie in C. Das Prädikat `argc/1` unifiziert sein Argument mit der Anzahl der Kommandozeilenparameter. Dabei wird, wie in C üblich, der Kommandoname selbst mitgezählt. Das Prädikat `argv/2` erwartet als zweites Argument eine Zahl. Es unifiziert dann sein erstes Argument mit dem entsprechenden Argument der Kommandozeile.

Beispiel 28 (`echo/0`)

echo/0 Ausgabe der Kommandozeile, mit der Prolog gerufen wurde

```
echo :-
    i:=0,
    repeat,
      I:=i, argv(X,I), write(X), tab(1), i:=i+1,
    i>argc,
    nl,!.
```

10.2.6 stats/0 und version/0

Das Prädikat `stats/0` ermittelt die Speicherauslastung des Prologinterpreters. In einer kurzen Nachricht auf der Standardausgabe werden die Auslastungen der einzelnen Speicherbereiche ausgegeben.

Der Interpreter verwaltet fünf tabellenartig organisierte Speicherbereiche (Nodes, Atome, Characters, Trail und Environments). Terme werden im Node-Speicher abgelegt. Atome, Variablen und Integerzahlen belegen darin genau ein Element. Strukturierte Terme belegen die Elemente, die für die Argumente nötig sind, zuzüglich eines Elements für den Hauptfunktor. Klauseln werden in kompakter Form als Terme abgespeichert, denen noch drei weitere Node-Elemente zugeordnet sind, die Verwaltungsinformationen enthalten. Funktoren werden durch einen Verweis auf einen Eintrag im Atomspeicher dargestellt. Gleichnamige Atome beziehen sich dabei auf ein und dieselbe Zeichenfolge im Char-Speicher. Im Trail-Speicher werden alle Variablen vermerkt, die während des Backtrackings wieder freigegeben werden müssen. Im Environment-Speicher werden Verweise auf alle aktiven Klauseln vermerkt.

Die Node-, Atom- und Char-Speicher untergliedern sich jeweils in zwei Bereiche. Im Heap-Bereich werden alle Elemente abgespeichert, die für Terme der Datenbasis benötigt werden. Im Stack-Bereich sind alle dynamisch erzeugten Strukturen abgespeichert, die beim Backtracking wieder freigegeben werden.

Die Ausgabe von `stats/0` schlüsselt die Auslastung jedes einzelnen Speicherbereiches auf und gibt zusätzlich deren maximale Größe und die prozentuale Auslastung aus.

```
?-stats.

Prolog Execution Statistics:

Nodes: 48000 Stack: 10 (0%) Heap: 431 (0%) Released: 0 (0%)
Atoms: 10000 Stack: 0 (0%) Heap: 282 (2%)
Strings: 32000 Stack: 0 (0%) Heap: 1544 (4%)
Environments: 10000 Used: 3 (0%)
Trail: 10000 Used: 0 (0%)

yes
```

Das Prädikat `version/0` gibt die Copyright-Meldung, die aktuelle Versionsnummer sowie die Konfigurationsparameter auf den aktuellen Ausgabestream aus.

```
?-version.
HU-Prolog (Public Domain Version)  Release 2.025 (last change: 16.03.93 14:53)
Author(s): 87-89  Christian Horn, Mirko Dziadzka, Matthias Horn
           90-93  Mirko Dziadzka (dziadzka@informatik.hu-berlin.de)

system:  UNIX BIT32 POINTERMODE
limits:  MAXDEPTH = 5000
 MAX_NAME_LEN = 8192
 MAXARITY = 127
 MAXPREC = 2047
 MAXVARS = 200
 VARLIMIT = 1000
 WRITEDEPTH = 250
 WRITELENGTH = 250
 READSIZE = 2000
 READDEPTH = 1000
arithmetik:  LONGARITH REALARITH
other options:  WINDOWS USER QUICK_BUT_DIRTY SAVE
specials:  HIGHER_ORDER
paths:  STDPLGRC = /usr/local/lib/huprolog/prologrc
 STD_HELP_FILE = /usr/local/lib/prolog.hlp

yes
```

10.3 Testunterstützung

Zur Unterstützung der Programmentwicklung mit dem System stellt der Interpreter `trace`- und `spy`-Funktionen zur Verfügung, die auf dem Box-Modell der Prolog-Abarbeitung basieren. Damit ist es möglich, den Ablauf der Interpretation vollständig bzw. selektiv zu verfolgen.

10.3.1 trace/0, trace/1 und notrace/0

Das Verfolgen des Ablaufes mittels Trace wird über ein Flag gesteuert. Das `trace`-Flag wird mit `trace/0` eingeschaltet und mit `notrace/0` wieder ausgeschaltet. Solange das `trace`-Flag gesetzt ist, wird das Betreten oder Verlassen eines jeden Prädikats über einen der vier Ports protokolliert. Als Protokollfile wird der Stream `stdtrace` benutzt. Neben `trace/0` und `notrace/0` kann das trace-Flag mit dem `trace/1`-Prädikat abgefragt bzw. explizit gesetzt werden. Die Standardeinstellung ist off.

Bei gesetztem trace-Flag erhält man folgende Ausschriften:

- Wird ein Prädikat aufgerufen, so erscheint die Ausschrift:

```
(n)  CALL: p(...)     [sanft?]
```

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

```
(n)  REDO: p(...)     [sanft?]
```

  In beiden Fallen erwartet der Interpreter danach eine Eingabe eines der Zeichen "`sanft?`" :

  - **[f]ail** Der Interpreter löst an dieser Stelle Backtracking aus.
  - **[s]kip** Das Prädikat p(...) wird aufgerufen, jedoch dabei die Ablaufverfolgung ausgeschaltet. Als nachtste Testausschrift erscheint die Information über das Verlassen des Prädikates term.
  - **[a]bort** Die Interpretation wird abgebrochen und der Interpreter kehrt in das Toplevel zurück.
  - **[n]otrace** Das trace-Flag wird auf off gesetzt und das Prädikat p(...) aufgerufen.
  - **[t]race** Das trace-Flag wird auf on gesetzt und das Prädikat p(...) aufgerufen.
  - ? Es wird eine Hilfsinformation über die Funktion der Testoptionen ausgegeben.
  - Alle anderen Eingaben wirken wie `<ET>`: die Abarbeitung des Prädikats beginnt.

- Beim erfolgreichen Verlassen eines Prädikates erscheint die Ausschrift:

```
(n)  EXIT:  p(...)
```

- Beim Fehlschlagen eines Prädikates erscheint die Ausschrift:

```
(n)  FAIL:  p(...)
```

10.3.2 spy/1 und nospy/1

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

10.4 Online Hilfssystem

Für HU-Prolog wurde ein Online Hilfssystem entwickelt. Dieses wird über die Prologprädikat `help/0` oder `help/1` aufgerufen. Das Online Help ist in der Regel aktueller, aber nicht so ausführlich wie diese Dokumentation.

Um mehr über die Funktioneweise des Online Hilfssystems zu erfahren, rufe man `help(help).` auf.

10.5 Systemflags und Optionen

Das Ausgabeverhalten von HU-Prolog wird über die Flags `echo`, `warn` und `log` gesteuert. Jedes dieser Flags kann die Zustande `on` (gesetzt) oder `off` (nicht gesetzt) annehmen. Mit Hilfe der gleichnamigen Prädikate `echo/1`, `warn/1` und `log/1` können die Zustande der Flags geändert und abgefragt werden. Werden die Prädikate mit dem Argument on oder off aufgerufen, so wird das entsprechende Flag auf diesen Wert gesetzt. Ist das Argument eine freie Variable, so wird diese mit dem aktuellen Zustand

des entsprechenden Flags unifiziert. Jedes andere Argument führt zu einem Fehler. Jedes Flag hat bei Systemstart eine Standardeinstellung, die jedoch durch Kommandozeilenparameter (Optionen) geändert werden kann.

- Mit dem `echo`-Flag wird die Art und Weise des Einlesens von Termen gesteuert. Falls das Flag gesetzt ist, werden Eingaben im Moment des Einlesens auf die Standardausgabe ausgegeben. Dieser Mechanismus ist nicht auf das Einlesen im Toplevel beschränkt, sondern erstreckt sich auf alle Eingabeoperationen. Die Standardeinstellung ist `off`. Die Kommandozeilenoption `-e` schaltet das echo-Flag ein.

- Mit dem `warn`-Flag wird die Ausgabe von Warnungen aus- bzw. eingestellt. Die Standardeinstellung ist `on`. Die Kommandozeilenoption `-w` schaltet das `warn`-Flag ein.

- Mit dem `log`-Flag wird die Erzeugung eines vollständigen Ein-/Ausgabeprotokolls gesteuert. Es gibt in HU-Prolog die Möglichkeit alle Ein- und Ausgaben des Interpreters in einer Datei zu protokollieren. Dazu muß beim Aufruf des Interpreters der Filename der Protokolldatei mit der `l-` bzw. `L-`Option spezifiziert werden. Wenn das `log`-Flag gesetzt ist, werden alle Ein- und Ausgaben in der spezifizierten Datei protokolliert, wenn es nicht gesetzt ist, nicht. Die Standardeinstellung ist `off`. Wenn beim Systemstart keine Protokolldatei spezifiziert wurde, hat das `log`-Flag keine Wirkung.

  `log/0` ist eine Abkürzung für log(on) und `nolog/0` eine Abkurzung für log(off).

  Die Optionen `-l` *filename* und `-L` *filename* dienen der Angabe des Dateinamens für das Protokollfile. Der Dateiname muß dabei als folgendes Kommandozeilenargument angegeben werden. Die Option `-l` läßt dabei die Standardeinstellungen der Flags unverändert, während die Option `-L` das log-Flag setzt.

- Das `sysmode`-Flag beschreibt den aktuellen Systemzustand: ist das Flag gesetzt (`on`), so befindet sich das System im priveligierten Systemmodus. In diesem Zustand ist ein Umdefinieren der built-in-Prädikate und -Operatoren möglich. Im Anwendermodus ist aus Sicherheitsgründen das Überschreiben von Systemprädikaten verboten. Die Standardeinstellung ist `off`.

  Die Option -s, gefolgt von einem Dateinamen, spezifiziert eine Prolog- Quelltextdatei, die vor dem Interpreterstart eingelesen und interpretiert wird. Alle Prädikate, die in dieser Datei definiert sind, erhalten den Status von Systemprädikaten.

Beim Systemstart ist eine Umlegung der Standardeingabe- und ausgabestrome möglich. Mit Hilfe von Prolog lassen sich daher Anwenderprogramme schaffen, die sich in die allgemeine Betriebssystemphilosophie einordnen. Prolog-Programme können so z.B. in Pipes eingebaut werden oder auf Betriebssystemebene mit anderen Bausteinen gekoppelt werden.