Prolog-68

Benutzerhandbuch
(Vorläufige Version vom 23. Februar 1992)

Jens Kilian

Inhaltsverzeichnis

Vorwort

Dies ist die vorläufige Dokumentation zu Prolog-68, einem frei kopierbaren Prolog-System für den Atari ST. Die endgültige Fassung der Dokumentation wird noch einige Zeit auf sich warten lassen.

Ich bin ab Januar '92 unter den folgenden Adressen zu erreichen:

Pest:	Jens Kilian
	Holunderstraße 19
D-W-7033	Herrenberg-Gültstein
Internet:	`jensk@hpbbrn.bbn.hp.com`
MausNet:	`Jens Kilian @ BB`

Lassen Sie sich bitte nicht davon abhalten, mir Ihre positiven (hoffentlich) oder negativen (hoffentlich nicht) Eindrücke mitzuteilen; ich bin dankbar für jede Reaktion.

Noch eins: Fühlen Sie sich nicht dazu verpflichtet, mir Geld zu schicken.[1] Seit ich 'GNU Emacs' und 'X' kenne, bringe ich es nicht mehr übers Herz, für meine kümmerlichen Machwerke die Hand aufzuhalten. Wenn Sie der Meinung sind, Sie müßten dennoch etwas für mich tun, dann schreiben Sie eigene Programme und geben Sie sie genauso öffentlich frei, wie ich das mit Prolog-68 tue.

Apologies to all those who don't speak German. I tried to bring the system as close to the standard as possible, so you should be able to use it if you have had some Prolog experience, even if you can't read this document (you're not missing much ...).

Was ist Prolog-68 ?

Prolog-68 ist mein Versuch, auf dem Atari ST ein Prolog-System zu implementieren, das auch für anspruchsvolle Anwendungen zu gebrauchen ist. Meine vier wichtigsten Entwurfskriterien sind dabei

1. Geschwindigkeit,
2. Geschwindigkeit,
3. Geschwindigkeit und
4. Kompatibilität zu Workstation-Prologs.

Prolog-68 basiert auf der „Warren Abstract Machine"; es enthält einen Compiler, der die Programmklauseln in einen Zwischencode übersetzt. Der Zwischencode wird als 'direct threaded code' gespeichert und von einem Simulator abgearbeitet. Das Programm erreicht so eine Geschwindigkeit von 12 kLIPS unter dem `nrev`-Benchmark, was für Prolog auf dem Atari ST einen durchaus annehmbaren Wert darstellt.

Die meisten eingebauten Prädikate sind inzwischen implementiert, mit Ausnahme zweier Gruppen:

[1] "Don't want money. Got money. Want admiration."

- Datenbankoperationen (`assert`, `retract`, ...)
- `setof` und `bagof`

An den Datenbankoperationen arbeite ich (obwohl man ohne sie auskommen kann – wer's nicht glaubt, sollte „The Craft of Prolog“ lesen[2]). Für `setof` gibt es eine Implementation in der Edinburgh-Bibliothek, die ich vermutlich verwenden werde; ich habe im Moment nur keine Zeit, um den Code an Prolog-68 anzupassen.

Wer die Vorabversion von letztem Jahr schon hat, fragt sich vielleicht, was ich in den vergangenen 12 Monaten gemacht habe. Nun ja.

1. War ich in den USA (Dank an J.L. Encarnaçaõ, J.D. Foley, C. Hornung und die FhG);
2. habe ich meine Diplomarbeit gemacht;
3. habe ich einen Job gesucht und gefunden; und
4. bin ich umgezogen.

Die Erweiterungen an Prolog-68 halten sich also in Grenzen. Es gibt jetzt (1) einen fast vollständigen Debugger, (2) eine verbesserte Fehlerbehandlung, (3) einen etwas korrekteren Compiler, (4) neue built-ins und (5) kaum Dokumentation zu allen diesen Änderungen.

[2] Alle anderen auch.

Allgemeine Nutzungserlaubnis

Dies ist eine übersetzte Version der 'GNU General Public License' der Free Software Foundation in der Version 2 vom Juni 1991. Der Autor dieses Dokuments unterhält keine Beziehungen zur FSF, unterstützt aber deren Ziele.

Die vorliegende Übersetzung ist *nicht* von der FSF autorisiert und kann daher nicht als Ersatz für die Lizenzen der 'echten' GNU-Software dienen. (Richard M. Stallman hat mir auf eine Anfrage hin mitgeteilt, daß eine autorisierte Übersetzung von einem Rechtsanwalt überprüft werden müßte, der sich im Urheberrecht gut auskennt – sowas kann ich mir im Moment nicht leisten.)

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Präambel

Die Lizenzen der meisten Software-Produkte sind so angelegt, daß sie Ihnen die Freiheit nehmen, diese zu teilen oder zu verändern. Im Gegensatz dazu soll die 'GNU General Public License' Ihre Freiheit garantieren, freie Software auszutauschen und zu verändern – um sicherzustellen, daß die Software für alle ihre Benutzer frei ist. Diese Allgemeine Nutzungserlaubnis erstreckt sich auf den größten Teil der Software der Free Software Foundation und auf jedes andere Programm, dessen Autoren sie benutzen. (Ein kleinerer Teil der FSF-Software wird stattdessen durch die 'GNU Library General Public License' abgedeckt.) Auch Sie können sie für Ihre Programme nutzen.

Wenn wir von freier Software reden, beziehen wir uns auf die Freiheit, nicht auf den Preis. Unsere Allgemeine Nutzungserlaubnis (ob GPL oder GLPL) wurde entworfen, um sicherstellen, daß Sie Kopien von freier Software verbreiten dürfen (und für diese Dienstleistung Gebühren verlangen dürfen, wenn sie das wünschen), daß Sie Quellcode erhalten oder ihn beziehen können, wenn Sie ihn haben wollen, daß Sie die Software ändern oder Teile daraus in neuen freien Programmen verwenden können; und daß Sie wissen, daß Sie diese Dinge tun dürfen.

Um Ihre Rechte zu schützen, müssen wir Einschränkungen machen, die jedem verbieten, Ihnen diese Rechte vorzuenthalten oder Sie aufzufordern, auf diese Rechte zu verzichten. Diese Einschränkungen erlegen Ihnen eine bestimmte Verantwortung auf, wenn Sie Kopien der Software verbreiten, oder wenn Sie sie verändern.

Wenn Sie zum Beispiel Kopien der Software weitergeben, ob gratis oder gegen eine Gebühr, müssen Sie den Empfängern alle Rechte gewähren, die auch Sie besit-

[3] Ich habe mich bemüht, das Original so wortgetreu wie möglich zu übersetzen. Fußnoten (so wie diese) stammen von mir und dienen nur zur Erläuterung. —jjk

zen. Sie müssen sicherstellen, daß auch die Empfänger den Quellcode erhalten oder beziehen können. Außerdem müssen Sie ihnen diese Lizenz zeigen, damit sie ihre Rechte erfahren.

Wir schützen Ihre Rechte in zwei Schritten: Erstens stellen wir die Software unter urheberrechtlichen Schutz, zweitens gewähren wir Ihnen diese Nutzungserlaubnis, die Ihnen das Recht gibt, die Software zu kopieren, zu verbreiten und/oder zu verändern.

Zum Schutz aller Autoren und unser selbst wollen wir außerdem sicherstellen, daß jedem klar ist, daß für diese freie Software keine Gewährleistung besteht. Wird die Software von jemand anderem verändert und weitergegeben, so sollen die Empfänger erfahren, daß sie nicht das Original erhalten haben, so daß Probleme, die von anderen verursacht wurden, nicht das Ansehen des ursprünglichen Autors schmälern.

Zu guter Letzt: jedes freie Programm wird ständig durch Software-Patente bedroht.[4] Wir wollen der Gefahr vorbeugen, daß die Verbreiter eines freien Programms individuelle Patente darauf anmelden und damit das Programm im Endeffekt zu ihrem Eigentum machen. Um dies zu vermeiden, haben wir klargemacht, daß für jedes Patent entweder für jedermann frei Lizenzen vergeben werden müssen, oder daß dafür überhaupt keine Lizenzen vergeben werden dürfen.

Es folgen die genauen Bedingungen für die Vervielfältigung, Verbreitung und Veränderung.

Bedingungen für die Vervielfältigung, Verbreitung und Veränderung

1. Diese Lizenzvereinbarung bezieht sich auf jedes Programm oder andere Werk, das einen vom Urheber angebrachten Vermerk enthält, der besagt, daß es unter den Bedingungen dieser Allgemeinen Nutzungserlaubnis verbreitet werden darf. Jedes solche Programm oder Werk wird im folgenden als das „Programm“ bezeichnet, und ein „auf dem Programm basierendes Werk“ bezeichnet entweder das Programm oder jedes laut Urheberrecht daraus abgeleitete Werk: das heißt ein Werk, das das Programm oder Teile daraus enthält, entweder wörtlich oder mit Änderungen und/oder in eine andere Sprache übersetzt. (Im nachfolgenden Text ist die Übersetzung ohne Einschränkung im Ausdruck „Veränderung“ miteinbegriffen.) Jede Lizenznehmerin/Jeder Lizenznehmer wird als „Sie“ angesprochen.

   Andere Tätigkeiten als das Vervielfältigen, Verbreiten und Verändern werden durch diese Erlaubnis nicht abgedeckt; sie sind nicht in deren Geltungsbereich. Die Ausführung des Programms wird nicht eingeschränkt, und die vom Programm erzeugte Ausgabe wird nur dann von dieser Nutzungserlaubnis geschützt, wenn ihr Inhalt ein vom Programm abgeleitetes Werk darstellt (unabhängig davon, ob sie durch die Ausführung des Programms erzeugt wurde). Ob das der Fall ist, hängt von der vom Programm erfüllten Aufgabe ab.[5]

2. Sie dürfen wörtliche Kopien des Quellcodes des Programms, so wie Sie ihn erhalten haben, auf einem beliebigen Datenträger anfertigen und verbreiten, vorausgesetzt, daß Sie jede Kopie mit einem deutlichen und angemessenen Urhebervermerk und einem Widerruf der Gewährleistung versehen, daß Sie alle Vermerke in Bezug auf diese Allgemeine Nutzungserlaubnis und auf das Fehlen einer Gewährleistung unverändert beibehalten, und daß Sie jedem anderen

[4] Verwerfliche, krankhafte Abscheulichkeiten. Bekämpft Software-Patente! Schließt Euch der Liga für Programmierfreiheit an!

[5] Diese Klausel bezieht sich auf Programme wie den Parsergenerator ‘Bison’.

Empfänger des Programms ein Exemplar dieser Allgemeinen Nutzungserlaubnis zukommen lassen.

Sie dürfen für den physischen Vorgang des Anfertigens einer Kopie eine Gebühr erheben, und Sie dürfen, falls gewünscht, Gewährleistungsschutz gegen Zahlung einer Gebühr anbieten.

3. Sie dürfen Ihr (e) Exemplar (e) des Programms oder Teile davon verändern, wodurch Sie ein vom Programm abgeleitetes Werk herstellen, und solche Änderungen oder solch ein Werk unter den Bedingungen von Absatz 2 vervielfältigen und verbreiten, vorausgesetzt, daß Sie außerdem die folgenden Bedingungen erfüllen:

   (a) Sie müssen die geänderten Dateien mit deutlichen Vermerken versehen, aus denen hervorgeht, daß und wann Sie die Dateien verändert haben.

   (b) Sie müssen veranlassen, daß jedem Dritten ohne Erhebung einer Gebühr unter den Bedingungen dieser Allgemeinen Nutzungserlaubnis die Nutzung jedes Werks gestattet wird, das Sie verbreiten oder veröffentlichen und das im Ganzen oder in Teilen das Programm oder Teile davon enthält oder davon abgeleitet ist.

   (c) Falls das Programm im Normallfall interaktiv Kommandos einliest, wenn es gestartet wurde, müssen Sie veranlassen, daß es, wenn es auf die einfachste und gebräuchlichste Weise für eine solche interaktive Benutzung gestartet wird, eine Meldung ausgibt oder anzeigt, die einen angemessenen Urhebervermerk und einen Hinweis darauf enthält, daß keine Gewährleistung besteht (oder aber daß Sie Gewährleistung einräumen) und daß Benutzer das Programm unter den vorliegenden Bedingungen weiterverbreiten dürfen. Die Meldung muß dem Benutzer außerdem angeben, wie er ein Exemplar dieser Allgemeinen Nutzungserlaubnis einsehen kann. (Ausnahme: falls das Programm selbst interaktiv ist, aber im Normalfall keine solche Meldung ausgibt, wird auch von einem davon abgeleiteten Werk nicht verlangt, eine solche Meldung auszugeben.)

   Diese Bedingungen beziehen sich auf das gesamte veränderte Werk. Wenn identifizierbare Teile dieses Werks nicht vom Programm abgeleitet sind, und wenn sie vernünftigerweise selbst als unabhängige und eigenständige Werke betrachtet werden können, beziehen sich diese Nutzungserlaubnis und ihre Bedingungen nicht auf diese Teile, wenn Sie sie als eigenständige Werke verbreiten. Wenn aber dieselben Teile als Teil eines Ganzen verbreitet werden, das ein vom Programm abgeleitetes Werk darstellt, muß die Verbreitung dieses Ganzen unter den Bedingungen der vorliegenden Nutzungserlaubnis stattfinden, so daß die dadurch anderen Lizenznehmern gewährten Rechte sich auf das vollständige Ganze in allen seinen Teilen beziehen, unabhängig von deren Urhebern.

   Mit diesem Absatz ist daher nicht beabsichtigt, Rechte auf ein vollständig von Ihnen geschaffenes Werk anzumelden oder Ihre Rechte daran einzuschränken; die Absicht ist vielmehr, das Recht auf Kontrolle der Verbreitung von Werken auszuüben, die vom Programm abgeleitet sind oder es enthalten.

   Das reine Vorhandensein eines anderen, nicht vom Programm abgeleiteten Werks auf demselben Speichermedium oder Datenträger wie das Programm (oder ein davon abgeleitetes Werk) bringt das andere Werk nicht in den Geltungsbereich dieser Nutzungserlaubnis.

4. Sie dürfen das Programm (oder ein davon abgeleitetes Werk, siehe Absatz 3) in Maschinencode oder ausführbarer Form unter den Bedingungen der obigen

Absätze 2 und 3 vervielfältigen und verbreiten, vorausgesetzt, daß Sie

(a) den gesamten dazugehörigen maschinenlesbaren Quellcode beifügen, der unter den Bedingungen der obigen Absätze 2 und 3 auf einem üblicherweise für den Austausch von Software benutzten Medium verbreitet werden muß; oder daß Sie

(b) ein schriftliches, mindestens drei Jahre gültiges Angebot beifügen, gegen Zahlung von nicht mehr als Ihrer eigenen Kosten für den physischen Vorgang des Kopierens und Versands an jeden Dritten eine vollständige maschinenlesbare Kopie des zugehörigen Quellcodes abzugeben, der dann unter den Bedingungen der obigen Absätze 2 und 3 auf einem üblicherweise für den Austausch von Software benutzten Medium verbreitet werden muß; oder daß Sie

(c) die Informationen beifügen, die Sie selbst in Bezug auf das Angebot für die Verbreitung des zugehörigen Quellcodes erhalten haben. (Diese Möglichkeit ist nur für die nichtkommerzielle Nutzung und nur für den Fall erlaubt, daß Sie selbst das Programm nur als Maschinencode oder in ausführbarer Form mit einem entsprechenden Angebot erhalten haben, wie in Unterabsatz 4b erläutert.)

„Quellcode für ein Werk“ bezeichnet die bevorzugte Form des Werks, um daran Veränderungen vorzunehmen. Für eine ausführbare Datei bezeichnet „vollständiger Quellcode“ den gesamten Quellcode für alle Module, die sie enthält, einschließlich aller zugehörigen Schnittstellendefinitionen und aller Steuerdateien für die Übersetzung und Installation des ausführbaren Programms. Als Ausnahme braucht der verbreitete Quellcode keine Dateien zu enthalten, die normalerweise (ob im Quellcode oder in Binärformat) mit den einzelnen Bestandteilen des Betriebssystems verbreitet werden, auf dem die ausführbare Form des Programms abläuft, es sei denn, daß die entsprechenden Bestandteile dem ausführbaren Programm beiliegen.

Erfolgt die Verbreitung des Maschinencodes oder ausführbaren Programms dadurch, daß das Kopieren von einem festgelegten Platz angeboten wird,[6] so zählt das Angebot des Kopierens des Quellcodes von demselben Platz als Verbreitung des Quellcodes, auch wenn Dritte dadurch nicht gezwungen werden, den Quellcode zusammen mit dem Maschinencode zu kopieren.

5. Sie dürfen das Programm **nicht** vervielfältigen, verändern, verbreiten, übertragen oder an Dritte Lizenzen dafür vergeben, soweit dies unter dieser allgemeinen Nutzungserlaubnis nicht ausdrücklich gestattet ist. Jeder anderweitige Versuch, das Programm zu vervielfältigen, zu verändern, zu verbreiten, zu übertragen oder Lizenzen dafür an Dritte zu vergeben, ist nichtig und beendet automatisch die Ihnen durch diese Nutzungserlaubnis gewährten Rechte. Dritte, die von Ihnen unter dieser Allgemeinen Nutzungserlaubnis Kopien oder Rechte erhalten haben, behalten davon unberührt ihre Lizenzen, solange sie die vorliegenden Bedingungen vollständig einhalten.

6. Sie sind nicht zur Anerkennung dieser Nutzungserlaubnis gezwungen, da Sie keinen Lizenzvertrag unterzeichnet haben. Allerdings gibt Ihnen nichts anderes das Recht, das Programm oder die davon abgeleiteten Werke zu verändern oder zu verbreiten. Diese Tätigkeiten sind gesetzlich verboten, falls Sie diese Nutzungserlaubnis nicht anerkennen. Daher erklären Sie durch die Verbreitung oder Veränderung des Programms (oder jedes auf dem Programm basierenden Werks) Ihr Einverständnis mit dieser Erlaubnis und mit all ihren

[6] Beispielsweise von einer Mailbox.

Bedingungen für die Vervielfältigung, Verbreitung oder Veränderung des Programms oder davon abgeleiteter Werke.

7. Sooft Sie das Programm (oder jedes auf dem Programm basierende Werk) weitergeben, erhält der Empfänger automatisch vom ursprünglichen Lizenzgeber die Erlaubnis zur Vervielfältigung, Verbreitung oder Veränderung des Programms unter Einhaltung der vorliegenden Bedingungen. Sie dürfen dem Empfänger keine weiteren Einschränkungen für die Ausübung der hier gewährten Rechte auferlegen. Sie sind nicht verpflichtet, Dritte zur Einhaltung dieser Bedingungen zu zwingen.

8. Falls Ihnen als Folge eines Gerichtsurteils, einer Patentrechtsverletzung oder aus irgendeinem anderen (nicht auf Patentsachen begrenzten) Grund Bedingungen auferlegt werden (ob durch Anordnung eines Gerichts, durch Zustimmung zu einem Vergleich oder auf andere Weise), die den Bedingungen der vorliegenden Allgemeinen Nutzungserlaubnis widersprechen, so befreien erstere Bedingungen Sie nicht von der Befolgung der letzteren. Falls Sie bei der Verbreitung des Programms nicht gleichzeitig ihre Verpflichtungen unter dieser Nutzungserlaubnis und irgendwelche anderen auf Sie zutreffenden Verpflichtungen erfüllen können, so dürfen Sie als Folge davon das Programm überhaupt nicht verbreiten. Wenn zum Beispiel eine Patentlizenz nicht die gebührenfreie Weitergabe des Programms durch alle Personen erlaubt, die direkt oder indirekt von Ihnen Kopien erhalten haben, dann wäre der einzige Weg, auf dem Sie gleichzeitig diesen Lizenzbedingungen und der vorliegenden Nutzungserlaubnis genügen können, ganz von der Verbreitung des Programms abzusehen.

   Falls irgendein Teil dieses Absatzes unter irgendwelchen Umständen für ungültig oder undurchführbar befunden wird, so ist der restliche Inhalt des Absatzes dennoch anzuwenden; unter allen anderen Umständen ist der Absatz als Ganzes anzuwenden.

   Der Zweck dieses Absatzes ist nicht, Sie zur Verletzung irgendwelcher Patente oder anderer Rechtsansprüche anzuhalten oder die Gültigkeit solcher Ansprüche in Frage zu stellen; die einzige Absicht ist es, die Integrität des Systems zur Verbreitung freier Software zu schützen, das durch allgemein vergebene Lizenzen realisiert wird. Viele Autoren haben großzügige Beiträge zu dem großen Angebot an Software gemacht, die durch dieses System verbreitet wird, wobei sie auf die konsistente Anwendung dieses Systems vertrauten; es ist die Angelegenheit des Autors, zu entscheiden, ob er oder sie Software über irgendein anderes System verbreiten will, und ein Lizenznehmer kann diese Entscheidung nicht erzwingen.

   Der Sinn dieses Absatzes ist, vollständig klarzumachen, was die Folge des Rests dieser allgemeinen Nutzungserlaubnis sein soll.

9. Falls die Verbreitung und/oder Benutzung des Programms in bestimmten Ländern durch Patente oder geschützte Schnittstellen eingeschränkt ist, so kann der Urheber, der das Programm unter diese Nutzungserlaubnis stellt, eine explizite geographische Verbreitungseinschränkung hinzufügen, die diese Länder ausschließt, so daß die Verbreitung nur innerhalb oder zwischen Ländern erlaubt ist, die nicht auf diese Weise ausgeschlossen sind. In solch einem Fall wird eine solche Einschränkung ein Teil der Allgemeinen Nutzungserlaubnis, als ob sie dort explizit niedergelegt wurde.

10. Die Free Software Foundation kann von Zeit zu Zeit revidierte und/oder neue Versionen der Allgemeinen Nutzungserlaubnis veröffentlichen. Solche neuen

Versionen werden im Geist der vorliegenden Version entsprechen, können aber in Einzelheiten von dieser abweichen, um neue Probleme oder Interessen abzudecken.

Jede Version erhält eine kennzeichnende Versionsnummer. Wenn das Programm eine Versionsnummer der Nutzungserlaubnis angibt, die sich auch auf „irgendeine spätere Version“ erstreckt, liegt es in Ihrem Belieben, entweder den Bedingungen der angegebenen Version oder denen irgendeiner späteren von der Free Software Foundation herausgegebenen Version zu folgen. Falls das Programm keine Versionsnummer der Nutzungserlaubnis angibt, dürfen Sie jede jemals von der Free Software Foundation herausgegebene Version wählen.

11. Falls Sie Teile des Programms in andere freie Programme aufnehmen wollen, die unter anderen Bedingungen verbreitet werden sollen, schreiben Sie an den Autor, um dafür Erlaubnis zu erbitten. Für Software, deren Urheberrechte bei der Free Software Foundation liegen, schreiben Sie an die Free Software Foundation; manchmal machen wir dafür Ausnahmen. Unsere Entscheidung wird von den zwei Zielen geleitet werden, den freien Status aller Abkömmlinge unserer freien Software zu erhalten und allgemein den Austausch und die Wiederbenutzung von Software zu unterstützen.

Ausschluss der Gewährleistung

12. Weil das Programm kostenlos genutzt werden darf, wird für das Programm keinerlei Gewährleistung eingeräumt, soweit dies gesetzlich zulässig ist. Falls nicht anderweitig schriftlich angegeben, stellen die Urheber und/oder Dritte das Programm „so wie es ist“ zur Verfügung, ohne Gewährleistung jedweder Art, eingeschlossen die Gewähr zur Erreichung eines bestimmten Verwendungszwecks, aber nicht beschränkt auf diese. Die Benutzung erfolgt auf eigene Gefahr; das gesamte Risiko in Bezug auf Qualität und Leistung des Programms liegt bei Ihnen. Sollte sich das Programm als fehlerhaft erweisen, so tragen Sie alle Kosten für anfallende Wartungs-, Reparatur- oder Korrekturarbeiten.

13. Soweit gesetzlich zulässig, haften die Urheber oder Dritte, die das Programm wie oben gestattet verändern und/oder weiterverbreiten, für keinerlei Schäden (einschliesslich irgenwelcher unmittelbaren Schäden, mittelbaren Schäden, Folgeschäden und Drittschäden), die aus der Benutzung oder der Unmöglichkeit der Benutzung des Programms entstehen (einschliesslich des Verlusts oder der Verfälschung von Daten, irgendwelcher materiellen Verluste, die Ihnen oder Dritten entstehen, oder des Versagens des Programms, mit irgendeinem anderen Programm zusammenzuarbeiten, aber nicht beschränkt auf diese), sogar in dem Fall, dass besagten Urhebern oder Dritten die Möglichkeit der Entstehung solcher Schäden bekannt war oder bekannt gemacht wurde.

Anhang: Wie Sie diese Bedingungen auch auf Ihre neuen Programme anwenden können

Wenn Sie ein neues Programm entwickeln, das der Öffentlichkeit von größtmöglichem Nutzen sein soll, dann ist der beste Weg, das zu erreichen, der, das Programm öffentlich freizugeben, so daß jedermann es unter Einhaltung dieser Bedingungen weiterverbreiten und verändern kann.

Um das zu erreichen, sollten Sie die folgenden Vermerke in Ihrem Programm anbringen. Es ist am sichersten, sie am Anfang jeder Quelldatei einzufügen, um das Fehlen einer Gewährleistung möglichst deutlich zu machen; jede Datei sollte zumindest die „Copyright"-Zeile und einen Verweis darauf enthalten, wo der vollständige Vermerk gefunden werden kann.

```
<eine Zeile mit dem Programmnamen und einer kurzen Beschreibung>
Copyright (C) 19yy  <Name des Autors>

Dieses Programm ist freie Software; Sie duerfen es
weiterverbreiten und/oder veraendern, solange Sie die Bedingungen
der 'GNU General Public License' einhalten, die von der 'Free
Software Foundation' herausgegeben wird; gueltig ist entweder
Version 2 oder (nach Ihrem Belieben) irgendeine spaetere Version.

Dieses Programm wird in der Hoffnung verbreitet, dass es von
Nutzen ist, aber OHNE JEDE GEWAEHRLEISTUNG; sogar ohne die
Gewaehr zur ERREICHUNG EINES BESTIMMTEN VERWENDUNGSZWECKS.
Bitte lesen Sie die 'GNU General Public License', wenn Sie
genaueres erfahren wollen.

Zusammen mit diesem Programm sollten Sie ein Exemplar der
'GNU General Public License' erhalten haben; falls nicht,
schreiben Sie an die Free Software Foundation, Inc.,
675 Mass Ave, Cambridge, MA 02139, USA.
```

Fügen Sie auch Informationen bei, wie man Sie durch elektronische und Papier-Post erreichen kann.

Wenn das Programm interaktiv ist, lassen Sie es eine kurze Meldung ausgeben (wie die folgende), wenn es in einem interaktiven Modus gestartet wird:

```
Gnomovision version 69, Copyright (C) 19yy <Name des Autors>
Gnomovision wird OHNE JEDE GEWAEHRLEISTUNG verbreitet; geben Sie
'show w' ein, um Naeheres zu erfahren.
Dies ist freie Software, und Sie dürfen sie weiterverbreiten,
solange Sie bestimmte Bedingungen einhalten; fuer mehr
Informationen geben Sie bitte 'show c' ein.
```

Dabei würden die hypothetischen Kommandos '`show w`' und '`show c`' die entsprechenden Teile der Allgemeinen Nutzungserlaubnis ausgeben. Sie können natürlich andere als diese Namen verwenden, oder Mausclicks oder die Auswahl aus einem Menü – was auch immer in Ihrem Programm angemessen erscheint.

Wenn Sie als Programmierer arbeiten, sollten Sie außerdem Ihren Arbeitgeber dazu bringen, schriftlich auf Rechte an dem Programm zu verzichten.[7]

[7] Im Original war hier ein Beispiel angegeben, das aber vermutlich nur in Amerika rechtsverbindlich ist.

Die vorliegende Allgemeine Nutzungserlaubnis erlaubt es nicht, Teile Ihres Programms in kommerzielle ('unfreie') Programme zu übernehmen. Falls es sich bei Ihrem Programm um eine Bibliothek von Unterprogrammen etc. handelt, kann es nützlicher sein, die Einbindung in kommerzielle Anwendungen zu erlauben. Wenn Sie das wollen, sollten Sie die 'GNU Library General Public License' verwenden.

Leerseite aus satztechnischen Gründen

Einführung 1

> **Logik,** *die* — Die Kunst, in strikter Übereinstimmung mit den Beschränkungen und Unfähigkeiten der menschlichen Nichterkenntnis zu denken und zu argumentieren. [...] —*Des Teufels Wörterbuch [4]*

In diesem Kapitel finden Sie zunächst einen kleinen Überblick über die Geschichte der logischen Programmierung und der Programmiersprache Prolog. Danach folgen eine (sehr oberflächliche) Einführung in den Teil der formalen Logik, der die Basis von Prolog bildet, und eine Erläuterung der entsprechenden programmiersprachlichen Konstrukte. Den Abschluß bildet die Erläuterung des Beweisverfahrens, das Prolog verwendet.

Dieses Kapitel ist *keine* Einführung in das Programmieren in Prolog. Es gibt dafür eine Reihe ausgezeichneter Lehrbücher, von denen hier nur die von Clocksin und Mellish [9, 10], Sterling und Shapiro [38, 39] und O'Keefe [31] genannt seien; im Literaturverzeichnis finden sich noch einige andere ([6, 8, 11, 18, 20, 21, 26, 27, 30, 34, 35, 36, 37, 42]). Darunter sind einerseits Lehrbücher für absolute Prolog-Anfänger (wie „Programming in Prolog“ von Clocksin und Mellish), andererseits solche, aus denen auch alte Prolog-Hasen noch das eine oder andere lernen können („The Craft of Prolog“ von Richard O'Keefe, trotz einiger Schlampereien eine wahre Fundgrube für Programmiertechniken).

Leser, die sich nicht für die theoretischen Hintergründe interessieren, können dieses Kapitel überschlagen.

1.1 Die Geschichte von Prolog

Die logische Programmierung basiert auf der Theorie automatischer Beweise, die bereits seit langer Zeit erforscht wird. Robinson [33] gelang auf diesem Gebiet ein Durchbruch mit der Entdeckung der Resolutionsregel und der Unifikation. Bereits im Jahre 1969 entwickelte Hewitt [16] am MIT einige Grundprinzipien der logischen Programmierung, und es wurden auch Programmiersprachen (z.B. PLANNER) entwickelt, denen logische Prinzipien zugrundelagen. Diese Sprachen waren aber so ineffizient und schwer zu kontrollieren, daß dieser Weg in den USA bald als Sackgasse betrachtet wurde.

In Europa (wo man der Entwicklung wie üblich hinterherhinkte) wurden Anfang der siebziger Jahre die Grundlagen für die moderne logische Programmierung geschaffen. Kowalski [24, 23, 22] erkannte, daß man den Beweis einer logischen Formel in Hornklauselnotation (siehe unten) als Ausführung eines Programms betrachten konnte. Colmerauer [12] entwickelte darauf aufbauend die Urform von Prolog für die Verarbeitung natürlicher Sprache, und van Emden und Kowalski [13] formalisierten die Semantik der neuen Programmiersprache.

Anfangs wurde die logische Programmierung (wie sie jetzt mit Recht genannt werden konnte) nur zögernd akzeptiert. Das Scheitern der oben erwähnten amerikanischen Projekte machte wenig Mut, in diese Richtung weiter zu forschen, und in den USA selbst hatte man nur ein müdes Lächeln für die Europäer übrig. In dieser Situation schlug der von Warren [41] entwickelte Compiler für Prolog wie eine Bombe ein. Prolog war auf einmal so schnell wie die damals schnellsten LISP-Systeme! Außerdem war der Compiler selbst beinahe vollständig in Prolog geschrie-

ben, womit die allgemeine Brauchbarkeit der Sprache bewiesen war. Ein Übriges tat die Ankündigung des japanischen 'Fifth Generation Project' im Jahre 1981, für das Prolog als bevorzugte Programmiersprache gewählt wurde. Seitdem ist die logische Programmierung endgültig den Kinderschuhen entwachsen.

1983 wurde, wiederum von Warren, ein Konzept entwickelt, das beinahe allen seitdem entwickelten Prolog-Systemen zugrundeliegt, nämlich die 'Warren Abstract Machine' [40, 14]. Neuerdings sprießen compilerbasierte Prolog-Versionen wie die Pilze aus dem Boden, und auch auf kleinen Rechnern werden annehmbare Rechenleistungen damit erzielt. Dabei werden ständig Erweiterungen und Verbesserungen für das ursprüngliche Modell entwickelt; viele dieser Erweiterungen sind in das vorliegende Programm eingegangen, das ebenfalls auf der WAM basiert.

Heute konzentriert sich die Forschung darauf, noch leistungsfähigere Programmierparadigmen zu entwickeln. Die Hauptrichtungen dabei sind die Parallelisierung von Prolog (z.B. Concurrent Prolog, FCP, GHC, ...; man kann kaum alle aufzählen) und das 'Constraint Logic Programming'[1] (z.B. CLP, Prolog III). Man darf also durchaus noch Neues von der logischen Programmierung erwarten.

Wie sehr die logische Programmierung heute akzeptiert wird, erkennt man an der Tatsache, daß inzwischen von amerikanischen Autoren ([1]) behauptet wird, das Ganze habe ja in Wirklichkeit mit der Arbeit von Hewitt begonnen, und die Europäer hätten nur versucht zu verstehen, was er eigentlich gemeint hat...

1.2 Begriffe der formalen Logik

Dieser Abschnitt soll nur einen groben Überblick über die Theorie geben, die hinter der logischen Programmierung steht. Wer sich näher für dieses Gebiet interessiert, der sollte zunächst das herrliche Buch von Hofstadter [17] lesen, in dem eine leichtverständliche Einführung in die Aussagenlogik gegeben wird. Außerdem behandelt dieses Buch die auch in der Logik wichtigen Fragen der Unvollständigkeit und Unentscheidbarkeit von formalen Systemen.

Wer tiefer in die Logik einsteigen will, der muß sich mit der Prädikatenlogik befassen. Eine gründliche, aber sehr anspruchsvolle Darstellung der Prädikatenlogik und ihrer Zusammenhänge mit der logischen Programmierung findet man bei Lloyd [25]; dieses Buch ist wohl das Standardwerk über die theoretischen Grundlagen von Prolog.[2]

Prolog arbeitet nach dem logischen Kalkül der *Resolutionsmethode*. Eine andere Art, automatische Beweise zu führen, kann man bei Bibel [7] finden. Diese *Konnektionsmethode* hat viele Gemeinsamkeiten mit der Resolution und wird wie diese praktisch eingesetzt. Darüber hinaus werden in dem Buch von Bibel verschiedene Erweiterungen der einfachen Beweismethoden angegeben.

1.2.1 Die Sprache der Prädikatenlogik 1. Stufe

In der Prädikatenlogik[3] werden Aussagen über *Objekte* gemacht. Diese Aussagen werden aus verschiedenen sprachlichen Einheiten zusammengesetzt, nämlich

[1] „Logische Programmierung mit Bedingungserfüllung"; man muß ja nicht alles übersetzen, sonst landet man am Ende wieder beim Viertopf-Zerknall-Treibling.

[2] Wer Gemeinsamkeiten zwischen diesem Buch und den nächsten Unterkapiteln entdeckt, braucht sich nicht zu wundern; meine diesbezüglichen Kenntnisse stammen aus einer Vorlesung, der dieses Buch zugrundelag.

[3] Ab sofort wird der Begriff *Prädikatenlogik* immer dann verwendet, wenn eigentlich *Prädikatenlogik 1. Stufe* gemeint ist. Logiken höherer Stufe sind für uns ohne Bedeutung, weil sie in Prolog nur am Rande in Erscheinung treten.

- *Variablen* $x,\ y, z,\ \ldots$, die stellvertretend für Objekte stehen. Es gibt (abzählbar) unendlich viele Variablen, die notfalls über Indizes unterschieden werden (z.B. y_{42}).
- *Funktionszeichen* $f^m, g^n,\ \ldots$. Es gibt (abzählbar) unendlich viele Funktionszeichen, notfalls wieder über Indizes unterscheidbar.

  Der obere Index (eine natürliche Zahl) gibt die *Stelligkeit* eines Funktionszeichens an. Funktionszeichen mit Stelligkeit 0 sind nichts anderes als Konstanten; normalerweise schreiben wir bei Konstanten die Stelligkeit aber nicht extra hin, sondern wir benutzen dafür die Buchstaben $a,\ b, c,\ \ldots$.
- *Prädikatszeichen* $p^m, q^n,\ \ldots$. Für sie gilt das bereits über die Funktionszeichen Gesagte. Prädikatszeichen mit Stelligkeit 0 sind die *Aussagenvariablen* der Aussagenlogik.
- *Junktoren* $\neg,\ \wedge, \vee, \rightarrow, \leftarrow,\ \leftrightarrow$, die die logischen Beziehungen zwischen Teilen einer Formel (s.u.) angeben.
- *Quantoren* $\forall,\ \exists$, mit denen Aussagen der Art „für alle x gilt P“ und „es gibt x, für das gilt P“ gebildet werden können.
- Hilfszeichen wie ‘(’, ’)’ und ‘,’.

Aus diesen syntaktischen Bausteinen können wir nach folgenden Regeln größere Einheiten konstruieren:

Definition 1 (Terme) *1. Jede Variable ist ein* Term.

2. Ist $n \geq 0$ und f^n ein n-stelliges Funktionszeichen, und sind $t_1, \ldots, t_n$ Terme*, so ist auch $f^n(t_1, \ldots, t_n)$ ein* Term.

Definition 2 (Atomformeln) *Ist $n \geq 0$ und p^n ein n-stelliges Prädikatszeichen, und sind $t_1, \ldots, t_n$ Terme, so ist $p^n(t_1, \ldots, t_n)$ eine* Atomformel.

Definition 3 (Formeln) *1. Jede Atomformel ist eine* Formel.

2. Sind F, G Formeln *und ist x eine Variable, so sind die folgenden Konstrukte ebenfalls* Formeln*:*

$\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftarrow G)$, $(F \leftrightarrow G)$, $(\forall x F)$, $(\exists x F)$

Um Klammern zu sparen, werden Konstanten wie $a()$ und Aussagenvariablen wie $p^0()$ ohne Klammern geschrieben: a, p^0. Außerdem vereinbaren wir für die Junktoren, daß einige davon stärker binden als andere, so daß man auch in komplizierteren Formeln Klammern weglassen kann. In der folgenden Aufstellung bindet $\neg$ am stärksten, $\leftrightarrow$ am schwächsten:

$$\neg \quad \begin{matrix}\forall \\ \exists\end{matrix} \quad \wedge \quad \vee \quad \begin{matrix}\rightarrow \\ \leftarrow\end{matrix} \quad \leftrightarrow$$

In Formeln wie $(\forall x F)$ und $(\exists x F)$ *binden* die Quantoren $\forall, \exists$ die Variablen, die hinter ihnen stehen. Der Geltungsbereich einer solchen Bindung ist die Formel F.

Definition 4 (Gebundene und freie Variablen) *Ein Vorkommen einer Variablen x in einer Formel F heißt* gebunden (in F)*, wenn es im Geltungsbereich eines Quantors $\forall x G$, $\exists x G$ liegt (also in der Formel G). Es heißt* frei (in F)*, wenn es nicht (in F) gebunden ist. Die Formel F heißt* geschlossen*, wenn keines der in ihr auftretenden Variablenvorkommen frei ist.*

Definition 5 (Allabschluß, Existenzabschluß) *Sei F eine Formel und seien $x_1, x_2, \ldots, x_n$ genau die Variablen, die in F frei vorkommen.*

1. *Der* Allabschluß $\forall(F)$ *von F ist die Formel*

$$\forall x_1 \forall x_2 \ldots \forall x_n F.$$

2. *Der* Existenzabschluß $\exists(F)$ *von F ist die Formel*

$$\exists x_1 \exists x_2 \ldots \exists x_n F.$$

Damit all diese Begriffe ein wenig klarer werden, folgen hier ein paar Beispiele:

- Eine vollständig geklammerte Formel:

$$\left(\exists x \left(p^1(x) \rightarrow \left(\forall y \left(\exists z q^2 \left(f^2(x,z), g^1(z)\right)\right)\right)\right)\right)$$

- Dieselbe Formel, überflüssige Klammern weggelassen:

$$\exists x \left(p^1(x) \rightarrow \forall y \exists z q^2 \left(f^2(x,z), g^1(z)\right)\right)$$

- Eine Formel, in der x gebunden vorkommt, y und z frei:

$$F \equiv \forall x(x \wedge y \rightarrow x \vee z)$$

- Die Abschlüsse von F:

$$\begin{aligned} \forall(F) &\equiv \forall y \forall z \forall x(x \wedge y \rightarrow x \vee z) \\ \exists(F) &\equiv \exists y \exists z \forall x(x \wedge y \rightarrow x \vee z) \end{aligned}$$

1.2.2 Interpretationen und Wahrheitswerte

Mit den gerade definierten Formeln kann man leider noch nicht viel anfangen. Wir wissen zwar, was die verschiedenen Junktoren etc. bedeuten *sollen*, aber bisher haben wir nur *syntaktische* Strukturen konstruiert, die auf alle möglichen Arten gedeutet werden können (auch auf völlig „unsinnige“, da wir den „Sinn“ der Formeln noch nicht kennen). Um das zu ändern, müssen wir unsere Formeln *interpretieren* können.

Definition 6 (Interpretation) *Ein Tripel $I = (D, I_F, I_P)$ heißt* Interpretation, *wenn*

- *D (der* Individuenbereich*) eine nichtleere Menge ist;*
- *I_F eine Funktion ist, die jedem n-stelligen Funktionszeichen f^n eine n-stellige Funktion $I_F(f^n) : D^n \rightarrow D$ zuordnet;*
- *I_D eine Funktion ist, die jedem n-stelligen Prädikatszeichen p^n eine n-stellige Relation $I_D(p^n) \subseteq D^n$ zuordnet.*

Definition 7 (Variablenbelegung) *Eine* Variablenbelegung *über einem Individuenbereich D ist eine Funktion V von der Menge aller Variablen nach D.*

Ist V eine Variablenbelegung*, x eine Variable und $d \in D$, so ist $V(x\backslash d)$ eine neue* Variablenbelegung*, für die gilt:*

$$(V(x\backslash d))(y) = \begin{cases} d, & \text{falls } y \equiv x, \\ V(y) & \text{sonst.} \end{cases}$$

F'	G'	$(F \wedge G)'$	$(F \vee G)'$	$(F \rightarrow G)'$	$(F \leftarrow G)'$	$(F \leftrightarrow G)'$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$
$\mathcal{F}$	$\mathcal{W}$	$\mathcal{F}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{W}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{W}$	$\mathcal{F}$	$\mathcal{W}$	$\mathcal{F}$
$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}$

Tabelle 1.1: Wahrheitswerte von Formeln

Mit diesen Begriffen können wir festlegen, was die vorhin definierten Strukturen bedeuten sollen.

Definition 8 (Wert eines Terms) *Sei I eine Interpretation, V eine Variablenbelegung. Der* Wert *eines Terms (bezüglich I und V) ist folgendermaßen definiert:*

1. *Der* Wert *einer Variablen x ist $V(x)$.*
2. *Seien $t'_1, \ldots, t'_n$ die* Werte *der Terme $t_1, \ldots, t_n$; dann ist der* Wert *von $f^n(t_1, \ldots, t_n)$ gleich $(I_F(f^n))(t'_1, \ldots t'_n)$.*

Definition 9 (Wahrheitswert einer Formel) *Der* Wahrheitswert *einer Formel (bezüglich einer Interpretation I und einer Variablenbelegung V) ist folgendermaßen definiert:*

1. *Sei F eine Atomformel, $F \equiv p^n(t_1, \ldots, t_n)$. Seien $t'_1, \ldots, t'_n$ die Werte der Terme $t_1, \ldots, t_n$. Dann ist der* Wahrheitswert *von F gleich*

$$F' = \begin{cases} \mathcal{W}, & \text{wenn } (t'_1, \ldots, t'_n) \in I_P(p^n), \\ \mathcal{F} & \text{sonst.} \end{cases}$$

2. *Sei F eine Formel, F' ihr* Wahrheitswert*. Der* Wahrheitswert *von $\neg F$ ist*

$$(\neg F)' = \begin{cases} \mathcal{W}, & \text{wenn } F' = \mathcal{F}, \\ \mathcal{F}, & \text{wenn } F' = \mathcal{W}. \end{cases}$$

3. *Seien F, G Formeln und F', G' deren* Wahrheitswerte*. Die* Wahrheitswerte *der daraus aufgebauten Formeln ergeben sich aus Tabelle 1.1.*
4. *Der* Wahrheitswert *von $\forall x F$ ist $\mathcal{W}$ genau dann, wenn für alle $d \in D$ der* Wahrheitswert *von F bezüglich I und $V(x\backslash d)$ gleich $\mathcal{W}$ ist.*
5. *Der* Wahrheitswert *von $\exists x F$ ist $\mathcal{W}$ genau dann, wenn es ein $d \in D$ gibt, für das der* Wahrheitswert *von F bezüglich I und $V(x\backslash d)$ gleich $\mathcal{W}$ ist.*

Ist der Wahrheitswert *von F bezüglich I und V gleich $\mathcal{W}$, so sagen wir, daß F* wahr ist bezüglich I und V*. Ist er gleich $\mathcal{F}$, so sagen wir, daß F* falsch ist bezüglich I und V*.*

Der Wahrheitswert (bezüglich I und V) einer Formel F, in der die Variable x nicht frei vorkommt, hängt nicht davon ab, welchen Wert $V(x)$ besitzt. Ist F geschlossen, so hängt sein Wahrheitswert daher nur von I, nicht aber von V ab. Wir reden in diesem Fall vom *Wahrheitswert von F bezüglich I*.

1.2.3 Erfüllbarkeit, Gültigkeit und Folgerungen

Definition 10 (Erfüllbarkeit, Gültigkeit) *Sei F eine Formel, I eine Interpretation.*

1. *F heißt* erfüllbar in *I, wenn $\exists(F)$ wahr ist bezüglich I.*

2. *F heißt* gültig in *I, wenn $\forall(F)$ wahr ist bezüglich I.*

3. *F heißt* unerfüllbar in *I, wenn $\exists(F)$ falsch ist bezüglich I.*

4. *F heißt* nicht gültig in *I, wenn $\forall(F)$ falsch ist bezüglich I.*

Definition 11 (Modell) *Eine Interpretation I heißt* Modell *einer geschlossenen Formel F, wenn F wahr ist bezüglich I. Wir schreiben dafür $I \models F$.*

I heißt Modell *einer Menge S von geschlossenen Formeln ($I \models S$), wenn I* Modell *jeder Formel $F \in S$ ist.*

Definition 12 (Allgemeingültigkeit, Erfüllbarkeit) *Eine Formel F heißt* allgemeingültig *(geschrieben $\models F$), wenn sie gültig ist in jeder Interpretation I. F heißt* erfüllbar*, wenn sie ein Modell besitzt. F heißt* unerfüllbar*, wenn sie in keiner Interpretation gültig ist.*

Entsprechendes definiert man auch für Formelmengen S.

Inzwischen sind wir an einem Punkt angelangt, an dem wir mit den Formeln etwas anfangen können. *Allgemeingültige* und *unerfüllbare* Formeln (und Mengen von Formeln) können unabhängig von speziellen Interpretationen und Variablenbelegungen betrachtet werden; deshalb sind diese Formeln erheblich interessanter als alle anderen. Am liebsten wäre es uns natürlich, wenn wir einen Weg hätten, um diese Formeln von den anderen zu unterscheiden; aber bevor wir zu den *Beweisverfahren* kommen, die genau diese Unterscheidung ermöglichen, hier noch ein paar Grundlagen:

Satz 1 *Eine Formel F ist genau dann allgemeingültig, wenn $\neg F$ unerfüllbar ist.*

(Der Beweis dieses Satzes ist so trivial, daß er noch nicht einmal als Übung taugt.)

Definition 13 (Semantische Folgerung) *Seien F und G geschlossene Formeln und S eine Menge geschlossener Formeln. Wir sagen „G* folgt semantisch aus *F“ (geschrieben $F \models G$), wenn jedes Modell von F auch Modell von G ist, und „G* folgt semantisch aus *S“ (bzw. $S \models G$), wenn jedes Modell von S auch Modell von G ist.*

Satz 2 *Sei S eine Menge von geschlossenen Formeln und F eine geschlossene Formel. Dann gilt $S \models F$ genau dann, wenn $S \cup \{\neg F\}$ unerfüllbar ist.*

Dieser Satz (das *Deduktionstheorem*) gibt bereits ein einfaches Beweisverfahren an. Jeder, der einmal einen Widerspruchsbeweis geführt hat, hat ihn schon benutzt. Der Beweis ist einfach und wird dem Leser zur Übung überlassen.[4]

1.2.4 Beweisverfahren

Die Allgemeingültigkeit (oder Unerfüllbarkeit) einer Formel mithilfe des semantischen Folgerungsbegriffs zu beweisen, ist relativ schwierig, da man dazu die (unendlich große) Menge aller möglichen Interpretationen untersuchen muß. Ideal wäre ein Beweisverfahren, das aufgrund der syntaktischen Struktur der Formel feststellt, ob sie allgemeingültig ist. Ein solches Beweisverfahren arbeitet nicht mit semantischen, sondern mit syntaktischen Folgerungsregeln (sogenannten *Schlußregeln*). Man schreibt z.B. $F \vdash_{\mathcal{R}} G$, um auszudrücken, daß G aus F syntaktisch folgt unter

[4] Wie überhaupt alle Beweise in diesem Kapitel!

Anwendung der Schlußregel $\mathcal{R}$, und $\vdash_{\mathcal{B}} F$, wenn F durch das Beweisverfahren $\mathcal{B}$ als allgemeingültig erkannt wird.

Es gibt eine ziemlich große Zahl von Beweisverfahren. Dazu gehören z.B. die *Resolutionsmethoden* (von denen wir eine genauer erläutern werden), die *Konnektionsmethoden* und einige weniger gebräuchliche wie Tableaux, der Sequenzenkalkül und das Maslov-Verfahren (ein zur Resolution duales Verfahren). In der Prädikatenlogik werden hauptsächlich die Resolutions- und Konnektionsmethoden verwendet.

Definition 14 (Korrektheit, Vollständigkeit) *Ein Beweisverfahren $\mathcal{B}$ heißt*

- korrekt*, wenn für alle Formeln F gilt $\vdash_{\mathcal{B}} F \implies \models F$.*
- vollständig*, wenn für alle Formeln F gilt $\models F \implies \vdash_{\mathcal{B}} F$.*
- Entscheidungsverfahren*, wenn es immer nach endlich vielen Schritten abbricht.*

Die genannten Beweisverfahren sind alle vollständig und korrekt, aber sie sind nur dann Entscheidungsverfahren, wenn man sie auf aussagenlogische Formeln anwendet. Für die Prädikatenlogik 1. Stufe und für alle Logiken höherer Stufen existieren keine Entscheidungsverfahren.[5]

1.3 Die logische Basis von Prolog

Nachdem wir nun die grundlegenden Begriffe der formalen Logik kennen, wollen wir sie auf den eigentlichen Gegenstand unseres Interesses anwenden, nämlich die Programmiersprache Prolog.

1.3.1 Programme und Anfragen

Prolog-Programme sind logische Formeln in einer eingeschränkten Form, der *Hornklauselform*. Diese wird folgendermaßen definiert:

Definition 15 (Literale) *Sei F eine Atomformel. Dann sind F und $\neg F$* Literale*. F heißt* positives*, $\neg F$* negatives Literal*.*

Definition 16 (Klauseln, Hornklauseln) *Eine* Klausel *ist der Allabschluß einer Disjunktion von Literalen.*

Eine Hornklausel *ist eine Klausel, die höchstens ein positives Literal enthält. Eine* Programmklausel *ist eine Klausel, die genau ein positives Literal enthält; eine* Zielklausel *ist eine Klausel, die nur negative Literale enthält.*

Einige Beispiele für Klauseln:

- $\forall x \forall y \forall z \left(p^3(x,y,z) \vee q^0\right)$
- $\forall \left(p^2(x,z) \vee \neg q^2(x,y) \vee \neg q^2(y,z)\right)$
- $\neg p^0 \vee \neg q^0$

[5] Diese Behauptungen gehören eigentlich in einen **Satz**, aber dazu müßte ich alle diese Beweisverfahren formal definieren, was den Rahmen dieses Kapitels endgültig sprengen würde. Außerdem sind die Beweise dieser Sätze so ziemlich das Kompliziertesteste, das die Logik zu bieten hat.

Um die Notation zu vereinfachen, schreibt man eine Klausel gewöhnlich unter Ausnutzung der Identität $F \leftarrow G \iff F \vee \neg G$ und der de Morganschen Gesetze folgendermaßen:

$$\begin{aligned} & \forall (A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n) \\ \iff \quad & \forall (A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n) \end{aligned}$$

Zur weiteren Abkürzung läßt man meistens auch noch den Allquantor weg und schreibt Kommata statt der Junktoren $\vee$ und $\wedge$. Damit sehen die Beispielklauseln von oben so aus:

- $p^3(x, y, z), q^0 \leftarrow$
- $p^2(x, z) \leftarrow q^2(x, y), q^2(y, z)$
- $\leftarrow p^0, q^0$

Jedes Prolog-*Programm* ist eine Konjunktion von Programmklauseln. Eine *Anfrage*, die der Benutzer an das System stellt, ist eine Zielklausel (siehe das letzte Beispiel). Das System nimmt diese Klausel zu den Programmklauseln hinzu und versucht, aus der gesamten entstehenden Formel die *leere Klausel* ($\leftarrow$ oder $\square$) herzuleiten. Gelingt das, so ist das Programm in Verbindung mit der Anfrage unerfüllbar.[6]

Für den Benutzer stellt sich dieser Vorgang wie ein Widerspruchsbeweis dar, da er seine Anfrage *positiv* interpretiert. Die Anfrage $\forall x \neg p^1(x)$ heißt aus seiner Sicht $\exists x p^1(x)$, und das System versucht zu beweisen, daß diese Anfrage in Verbindung mit den Programmklauseln allgemeingültig ist. Laut Satz 1 ist diese Betrachtungsweise absolut legal.

1.3.2 Terme und das Herbrand-Universum

Definition 17 *Ein* Grundterm *ist ein Term, in dem keine Variablen vorkommen. Eine* Grund-Atomformel *ist eine Atomformel, in der keine Variablen vorkommen.*

Das Herbrand-Universum U_H *ist die Menge aller Grundterme. Die* Herbrand-Basis B_H *ist die Menge aller Grund-Atomformeln.*[7]

Definition 18 (Herbrand-Interpretation) *Sei $I = (D, I_F, I_P)$ eine Interpretation. I heißt* Herbrand-Interpretation*, wenn gilt:*

1. $D = U_H$
2. $I_F(f^n) = \lambda t_1 \ldots \lambda t_n . f^n(t_1, \ldots, t_n)$

Eine Herbrand-Interpretation $I = (D, I_F, I_P)$ ist eindeutig bestimmt durch I_P, d.h. durch die Menge der Grund-Atomformeln, die bezüglich I den Wahrweitswert $\mathcal{W}$ besitzen (eine Teilmenge von B_H). Man kann daher Herbrand-Interpretationen und Teilmengen der Herbrand-Basis miteinander identifizieren.

Die Besonderheit der Herbrand-Interpretationen ist, daß die Funktionszeichen nur noch rein syntaktische Funktion haben. Genau dasselbe passiert in Prolog, wo der Term '3 + 4' nicht gleichbedeutend mit dem Term '7' ist. Prolog geht sogar noch einen Schritt weiter und unterscheidet Prädikatszeichen und Junktoren nicht von Funktionszeichen; damit sind Formeln in Prolog ebenfalls Terme und können vom Programm selbst manipuliert werden. Diese Möglichkeit interessiert uns hier nicht weiter, da sie innerhalb der Prädikatenlogik 1. Stufe nicht ausdrückbar ist.

[6] Das müssen wir aber erst noch beweisen!

[7] Strenggenommen muß man für jede Sprache L 1. Stufe ein eigenes Herbrand-Universum U_L und eine eigene Herbrand-Basis B_L definieren. Wir haben bisher immer die größtmögliche Sprache angenommen, und U_H bzw. B_H sind das dazugehörige Herbrand-Universum bzw. die Herbrand-Basis.

1.4 Das Beweisverfahren von Prolog

In diesem Abschnitt wird das von Prolog verwendete Beweisverfahren, die *SLD-Resolution*, erläutert. Dazu müssen wir zunächst einen grundlegenden Algorithmus, die *Unifikation* kennenlernen. Für die Programmiersprache Prolog hat die Unifikation die Bedeutung eines Parameterübergabeverfahrens (wie z.B. *call-by-value* oder *call-by-name*). Sie hat aber auch Anwendungen außerhalb der logischen Programmierung, z.B. bei Datenbanken oder im Compilerbau.

1.4.1 Substitutionen und Unifikatoren

Definition 19 (Ausdrücke) *Ein* einfacher Ausdruck *ist entweder ein Term oder eine Atomformel. Ein* Ausdruck *ist ein Term, ein Literal oder eine Konjunktion oder Disjunktion von Literalen.*

Definition 20 (Substitution) *Sei $\mathcal{V}$ die Menge aller Variablen, $\mathcal{T}$ die Menge aller Terme. Eine* Substitution *ist eine Funktion $\sigma : \mathcal{V} \to \mathcal{T}$, für die gilt*

$$\{x \in \mathcal{V} | \sigma(x) \neq x\} \quad \text{ist endlich.}$$

Die Anwendung *einer Substitution σ auf Ausdrücke ist wie folgt definiert:*

$$\begin{aligned}
x \in \mathcal{V} &\Rightarrow x\sigma \stackrel{\text{def}}{=} \sigma(x) \\
t \equiv f^n(t_1, \ldots, t_n) &\Rightarrow t\sigma \stackrel{\text{def}}{=} f^n(t_1\sigma, \ldots, t_n\sigma) \\
F \equiv p^n(t_1, \ldots, t_n) &\Rightarrow F\sigma \stackrel{\text{def}}{=} p^n(t_1\sigma, \ldots, t_n\sigma) \\
G \equiv \neg F &\Rightarrow G\sigma \stackrel{\text{def}}{=} \neg(F\sigma) \\
G \equiv F \wedge E &\Rightarrow G\sigma \stackrel{\text{def}}{=} (F\sigma) \wedge (E\sigma) \\
G \equiv F \vee E &\Rightarrow G\sigma \stackrel{\text{def}}{=} (F\sigma) \vee (E\sigma)
\end{aligned}$$

Wir schreiben $\sigma = \{x_1 \backslash t_1, \ldots, x_n \backslash t_n\}$ für die Substitution σ, für die gilt

$$\sigma(x) = \begin{cases} t_i, & \text{wenn } x \equiv x_i \text{ für ein } i \in \{1, \ldots, n\}, \\ x, & \text{wenn } x \notin \{x_1, \ldots, x_n\}. \end{cases}$$

Definition 21 *Seien σ und τ Substitutionen. Die* Komposition *$\sigma\tau$ von σ und τ ist definiert durch $x(\sigma\tau) \stackrel{\text{def}}{=} (x\sigma)\tau$. Die Erweiterung auf beliebige Ausdrücke geschieht wie in Definition 20.*

Definition 22 (Permutation) *Eine Substitution π heißt* Variablenumbenennung *oder* Permutation *genau dann, wenn*

1. *für alle Variablen x auch $x\pi$ eine Variable ist, und*
2. *π eine Bijektion auf der Menge aller Variablen darstellt.*

Definition 23 (Allgemeinheit) *Eine Substitution σ heißt* allgemeiner *als eine Substitution τ, wenn es eine Substitution ρ gibt mit $\tau = \sigma\rho$.*

Definition 24 (Unifikator, mgu) *Sei S eine Menge einfacher Ausdrücke und σ eine Substitution. Dann heißt σ* Unifikator von *S, wenn $S\sigma$ nur ein einziges Element enthält.*

σ heißt allgemeinster Unifikator (most general unifier, mgu) von *S, wenn*

1. *σ Unifikator von S ist und*

Seien t_1, t_2 die zu unifizierenden Terme; sei S eine Menge von Gleichungen der Form $s = t$, wobei s und t Terme sind; sei σ eine Substitution.

1. Setze $S = \{t_1 = t_2\}$, $\sigma = \emptyset$.
2. Falls eine der folgenden Regeln anwendbar ist, führe diese Regel aus.
   (a) Ist $S = S' \cup (x = x)$, wobei x eine Variable ist, so setze $S = S'$.
   (b) Ist $S = S' \cup (t = x)$, wobei x eine Variable ist und t ein Term, der keine Variable ist, so setze $S = S' \cup (x = t)$.
   (c) Ist $S = S' \cup (f^n(s_1, \ldots, s_n) = f^n(t_1, \ldots, t_n))$, so setze $S = S' \cup \{(s_1 = t_1), \ldots, (s_n = t_n)\}$.
   (d) Ist $S = S' \cup (x = t)$, wobei x eine Variable ist, die nicht in t vorkommt, so ersetze σ durch $\sigma \cup \{x\backslash t\}$ und setze $S = S'\{x\backslash t\}$.
3. Wiederhole den vorigen Schritt, bis keine Regel mehr anwendbar ist.
4. Falls $S = \emptyset$, so ist σ mgu von t_1 und t_2. Ansonsten sind t_1 und t_2 nicht unifizierbar.

Abbildung 1.1: Ein einfacher Unifikationsalgorithmus für Terme

2. σ allgemeiner ist als jeder Unifikator τ von S.

Ein allgemeinster Unifikator ist eindeutig bis auf Variablenumbenennung, d.h. sind σ, τ mgus von S, dann gibt es eine Permutation π mit $\tau = \sigma\pi$.

Algorithmen zur Berechnung allgemeinster Unifikatoren gibt es einige. Der einfachste ist der ursprünglich von Robinson [33] angegebene. Martelli und Montanari [28] geben ebenfalls einen einfachen Algorithmus an, der dann schrittweise erweitert wird; der effizienteste Unifikationsalgorithmus überhaupt stammt von Paterson und Wegman [32]. Abbildung 1.1 zeigt das Grundprinzip dieser Algorithmen.

Die meisten Prolog-Systeme benutzen einen vereinfachten, dafür aber schnelleren Algorithmus, der leider in manchen Situationen Fehler hevorruft. Beispielsweise dürften die Terme x und $f^1(x)$ im Schritt 2d des Algorithmus' von Abbildung 1.1 nicht unifiziert werden, da x in $f^1(x)$ vorkommt. Bei der vereinfachten Unifikation fehlt die Überprüfung, ob eine solche Situation vorliegt (der sogenannte '*occurrence check*'). Das führt dazu, daß ein unendlich großer Term[8] für x substituiert wird, was später zu Endlosschleifen führen kann. Es gibt auch Prolog-Systeme (beispielsweise Prolog-II), in denen diese unendlichen Terme legal sind.

Ein paar Beispiele zur Unifikation:

- $S = \{x, y, z\}$
  $\sigma = \{x\backslash z, y\backslash z\}, S\sigma = \{z\}$
  Da mgus nur bis auf Variablenumbenennung eindeutig sind, sind auch die folgenden Substitutionen mgus von S: $\sigma_1 = \{x\backslash y, z\backslash y\}, \sigma_2 = \{y\backslash x, z\backslash x\}$.
- $S = \{f^2(a, x), f^2(y, y)\}$
  $\sigma = \{x\backslash a, y\backslash a\}, S\sigma = \{f^2(a, a)\}$
- $S = \{f^2(a, x), g^2(y, y)\}$
  Nicht unifizierbar, da f^2 nicht durch Substitution zu g^2 verändert werden kann.

[8] In den meisten Systemem werden Terme intern als gerichtete Graphen dargestellt. Diese sind normalerweise azyklisch; bei der Substitution von x durch $f^1(x)$ entsteht ein zyklischer Graph, der als endliche Darstellung eines unendlich großen Terms betrachtet werden kann.

- $S = \left\{f^2(x,x), f^2(a,b)\right\}$
  Nicht unifizierbar, da x nicht gleichzeitig durch die Konstanten a und b substituiert werden kann.
- $S = \left\{f^2\left(g^2\left(y, h^1\left(x\right)\right), a\right), f^2\left(g^2\left(x,y\right), z\right)\right\}$
  Nicht unifizierbar, da ansonsten x durch $h^1(x)$ substituiert werden müßte. In Prolog-Systemen ohne *occurrence check* wird ein unendlich großer Term erzeugt.

1.4.2 SLD-Resolution

Die Schlußregeln, die in Resolutionsverfahren benutzt werden, sind die *Resolutionregel* und die *Faktorisierungsregel*. Beide Regeln werden benutzt, um aus bereits bekannten Klauseln eine neue Klausel abzuleiten.

Definition 25 (Komplement) *Sei L ein Literal. Das* Komplement $\overline{L}$ *von L ist*

$$\overline{L} \equiv \begin{cases} \neg F, & \text{wenn } L \text{ positiv und } L \equiv F, \\ F, & \text{wenn } L \text{ negativ und } L \equiv \neg F. \end{cases}$$

Seien $c \equiv \forall\left(K_1 \vee \ldots \vee K_k\right)$ und $d \equiv \forall\left(L_1 \vee \ldots \vee L_l\right)$ zwei Klauseln. Sei π eine Permutation, so daß c und $d\pi$ keine Variablen gemeinsam haben. Es gebe außerdem i, $1 \leq i \leq k$ und j, $1 \leq j \leq k$, so daß K_i und $\overline{L_j}\pi$ unifizierbar sind mit dem mgu σ. Dann kann man mit der *Resolutionsregel* die folgende neue Klausel ableiten:

$$\begin{aligned} e \quad \equiv \quad & \forall\left(\left(K_1 \vee \ldots \vee K_{i-1}\right.\right. \\ & \vee L_1\pi \vee \ldots \vee L_{j-1}\pi \vee L_{j+1}\pi \vee \ldots \vee L_l\pi \\ & \left.\left. \vee K_{i+1} \vee \ldots \vee K_k\right)\sigma\right) \end{aligned}$$

c und d heißen *Elternklauseln*, e heißt *Resolvente* von c und d.

Das Prinzip hinter der Resolutionsregel ist, daß die Literale $K_i\sigma$ und $L_j\pi\sigma$ *komplementär* sind, wenn K_i und $\overline{L_j}\pi$ unifiziert werden können; also ist eines dieser Literale wahr, das andere falsch. Da die Klauseln, aus denen K_i und L_j stammen, beide wahr sind, muß in derjenigen Klausel, die das falsche Literal enthält, mindestens ein wahres Literal vorkommen. Wenn man also *alle* verbleibenden Literale der beiden Klauseln zusammenfaßt, erhält man wieder eine wahre Klausel. Die beiden Literale K_i und L_j fehlen in der neuen Klausel, sie wurden aufgelöst (*resolviert*).

Für die *Faktorisierungsregel* benötigt man nur eine einzige Elternklausel $c \equiv \forall\left(K_1 \vee \ldots \vee K_k\right)$, für die es i und j gibt mit $1 \leq i,j \leq k$, so daß K_i und K_j unifizierbar sind mit dem mgu σ. Die daraus abgeleitete Klausel ist

$$d \equiv \forall\left(L_1 \vee \ldots \vee L_l\right),$$

wobei $\{L_1, \ldots, L_l\} = \{K_1, \ldots, K_k\}\sigma$. Wie wir später sehen werden, ist diese Regel für Prolog ohne Bedeutung.

Satz 3 *1. Sei e die Resolvente von c und d. Dann gilt $c, d \models e$.*

2. d sei die aus c durch Faktorisierung abgeleitete Klausel. Dann gilt $c \models d$.

Der erste Teil dieses Satzes ist oben (informell) bewiesen worden.

Definition 26 (Resolutionsableitung) *Sei S eine Menge von Klauseln. Eine* Resolutionsableitung *für S ist eine (endliche oder unendliche) Folge $(c_1, c_2, \ldots)$ von Klauseln, so daß für jedes c_i eine der folgenden Bedingungen gilt:*

1. $c_i \in S$

2. *Es gibt c_j, c_k mit $j, k < i$, so daß c_i Resolvente von c_j und c_k ist.*

3. *Es gibt c_j, $j < i$, so daß c_i aus c_j durch Faktorisierung abgeleitet wurde.*

Definition 27 (Resolutionswiderlegung) *Sei S eine Menge von Klauseln. Eine* Resolutionswiderlegung *von S ist eine endliche Resolutionsableitung $(c_1, \ldots, c_n)$ für S mit $c_n \equiv \square$. Gibt es eine Resolutionswiderlegung für S, so heißt S* durch Resolution widerlegbar.

Satz 4 (Korrektheit der Resolution) *Jede durch Resolution widerlegbare Klauselmenge ist unerfüllbar.*

Satz 5 (Vollständigkeit der Resolution) *Jede unerfüllbare Menge von Klauseln ist durch Resolution widerlegbar.*

Diese beiden Sätze zeigen, daß die Resolution ein Beweisverfahren darstellt. Das bisher besprochene Verfahren ist sehr allgemein, und Prolog verwendet nur eine eingeschränkte Version, die *SLD-Resolution*. Das Hauptmerkmal dieses Verfahrens ist, daß es nur Hornklauseln (oder *definite* Klauseln, daher das D) und nur die Resolutionsregel verwendet. Wie man sich leicht überlegen kann, ist die Resolvente zweier Hornklauseln auch wieder eine Hornklausel; außerdem ist die Resolvente einer Programm- und einer Zielklausel wieder eine Zielklausel. Eine SLD-Ableitung ist eine eingeschränkte Resolutionsableitung der Form

$$(Z_0, P_1, Z_1, P_2, \ldots),$$

wobei Z_0 die ursprüngliche Anfrage des Benutzers ist, P_i Klauseln aus dem Programm sind und Z_j jeweils die Resolvente von Z_{j-1} und P_j darstellt.

Das S in 'SLD' steht für eine *Auswahlfunktion* (engl. *selection function*). Diese Funktion wählt aus der aktuellen Zielklausel Z_n in einer Resolutionsableitung dasjenige Literal aus, das mit einem Literal aus P_{n+1} resolviert werden soll.[9] Die Auswahlfunktion heißt auch *Berechnungsregel*; daneben gibt es noch eine *Suchregel*, die die in einem Resolutionsschritt verwendete Klausel auswählt.[10] Eine grundlegende Eigenschaft der SLD-Resolution ist, daß das Verfahren unabhängig von den verwendeten Such- und Berechnungsregeln immer zum selben Ergebnis kommt.[11]

Eine Resolutionsableitung für eine Klauselmenge S kann als Baum dargestellt werden, dessen Blätter die Elemente von S sind. Für eine Resolutionswiderlegung ist die leere Klausel $\square$ die Wurzel des Baums. Eine SLD-Ableitung hat eine sehr einfache Form, wenn sie als Baum dargestellt wird:

$$\begin{array}{ccccccc} P_1 & & P_2 & & & & \\ & \searrow & & \searrow & & & \\ Z_0 & \to & Z_1 & \to & Z_2 & \to & \ldots \\ & \theta_1 & & \theta_2 & & & \end{array}$$

Das L von 'SLD' bezieht auf die lineare Form dieses Ableitungsbaums.

Im dargestellten Ableitungsbaum wurde für jeden Resolutionsschritt eine Substitution θ_i vermerkt. Diese Substitutionen sind die bei der Resolution verwendeten Unifikatoren. Für eine SLD-Widerlegung kann man aus den Unifikatoren eine *Antwortsubstitution* für die ursprüngliche Anfrage konstruieren.

[9] In Prolog wählt diese Funktion immer das erste Literal einer Zielklausel aus, weil Klauseln dort von links nach rechts abgearbeitet werden.

[10] Die erste ausgewählte Klausel liefert nicht immer das gewünschte Ergebnis. In einem solchen Fall macht man den letzten Resolutionsschritt rückgängig und wählt eine andere Klausel aus. In Prolog werden die Klauseln in der Reihenfolge betrachtet, in der der Benutzer sie eingegeben hat.

[11] Da die Resolution kein Entscheidungsverfahren ist, kann es aber sein, daß in *endlicher* Zeit keine Lösung gefunden wird.

C_1	$\equiv$	$f^2(a,i)$	Abraham ist Vater von Isaak.
C_2	$\equiv$	$f^2(i,j)$	Isaak ist Vater von Jakob.
C_3	$\equiv$	$g^2(x,z) \leftarrow f^2(x,y), f^2(y,z)$	X ist Großvater von Z, wenn X Vater von Y ist und Y Vater von Z ist.
Z_0	$\equiv$	$\leftarrow g^2(x,j)$	Wer ist Großvater von Jakob ?

Abbildung 1.2: Ein logisches Programm

Definition 28 (Antwortsubstitutionen) *Sei $G \equiv\leftarrow K_1, \ldots, K_n$ eine Zielklausel und P eine Menge von Hornklauseln. Eine* Antwortsubstitution für P und G *ist eine Substitution σ, so daß für alle Variablen x, die nicht in G vorkommen, gilt $x\sigma = x$.*

σ heißt korrekte Antwortsubstitution für P und G, *wenn*

$$P \models \forall\left((K_1 \wedge \ldots \wedge K_n)\,\sigma\right).$$

σ heißt berechnete Antwortsubstitution für P und G, *wenn es eine Resolutionswiderlegung für $P \cup \{G\}$ gibt, die die Unifikatoren $\theta_1, \ldots, \theta_k$ benutzt und wenn*

$$x\sigma = \begin{cases} x\theta_1 \ldots \theta_k, & \text{wenn } x \text{ in } G \text{ vorkommt,} \\ x & \text{sonst.} \end{cases}$$

Die Korrektheit der SLD-Resolution bewirkt, daß jede berechnete Antwortsubstitution eine korrekte Antwortsubstitution ist.

Zum Abschluß des Kapitels soll das logische Programm in Abbildung 1.2 mit Hilfe der Resolution bewiesen werden. Wir verwenden die Berechnungs- und Suchregel von Prolog.

1. Zielklausel ist $Z_0 \equiv\leftarrow g^2(x,j)$; die einzige Klausel, mit der resolviert werden kann, ist C_3.

   Zunächst werden die Variablen in C_3 umbenannt, so daß die neue Klausel $P_1 \equiv g^2(x',z') \leftarrow f^2(x',y'), f^2(y',z')$ entsteht. Die Unifikation von $g^2(x,j)$ und $g^2(x',z')$ ergibt die Substitution $\theta_1 = \{x'\backslash x, z'\backslash j\}$.

   Die neue Zielklausel ist die entstehende Resolvente:

   $$Z_1 \equiv\leftarrow f^2(x,y'), f^2(y',j)$$

2. Die Berechnungsregel wählt als nächstes Literal, mit dem resolviert werden soll, das erste Literal von Z_1 aus, also $f^2(x,y')$. Es gibt zwei Klauseln, die für die Resolution in Frage kommen, nämlich C_1 und C_2. Die Suchregel legt fest, daß zunächst C_1 benutzt wird.

   Es gilt $P_2 \equiv\leftarrow f^2(a,i)$, da keine Variablen umbenannt werden müssen. Unifikation unter $\theta_2 = \{x\backslash a, y'\backslash i\}$ und Resolution ergibt die neue Zielklausel

   $$Z_2 \equiv\leftarrow f^2(i,j).$$

3. Für den nächsten Resolutionsschritt kommen wieder die Klauseln C_1 und C_2 in Frage. Die Suchregel wählt wieder C_1 als erste zu betrachtende Klausel aus.

   Die Unifikation von Z_2 und $P_3 \equiv C_1$ schlägt fehl. Es muß also eine neue Klausel betrachtet werden; die Suchregel liefert C_2 als nächste (und letzte) Möglichkeit.

Jetzt ist $P_3 \equiv \leftarrow f^2(i,j)$. Die Unifikation gelingt, da beide betrachteten Literale gleich $f^2(i,j)$ sind. Der Unifikator ist $\theta_3 = \{\}$.

Die neue Zielklausel ist $Z_3 \equiv \leftarrow \equiv \square$. Damit ist bewiesen, daß die Klauselmenge $\{C_1, C_2, C_3, Z_0\}$ unerfüllbar ist. Die berechnete Antwortsubstitution ergibt sich aus der Einschränkung der Komposition

$$\theta_1\theta_2\theta_3 = \{x\backslash a, x'\backslash x, y'\backslash i, z'\backslash j\}$$

auf die in Z_0 vorkommenden Variablen; das Ergebnis ist $\sigma = \{x\backslash a\}$.

Wir haben damit bewiesen, daß die Formel $\forall x \neg g^2(x,j)$ in Verbindung mit den Formeln C_1, C_2 und C_3 unerfüllbar ist. Wie bereits in Abschnitt 1.3.1 erwähnt, kann diese Aussage auch so interpretiert werden, daß die Formel $\exists x g^2(x,j)$ allgemeingültig ist.[12] Nach Satz 2 haben wir also gezeigt, daß $\{C_1, C_2, C_3\} \models \exists x g^2(x,j)$.

Wir beenden hier die Einführung in die logischen Grundlagen von Prolog. Der Rest dieses Handbuchs beschäftigt sich nur mehr mit der *Programmiersprache* Prolog, oder vielmehr mit einem ihrer zahlreichen Dialekte.

[12] Unser Verfahren liefert in der Antwortsubstitution σ sogar einen Wert für x zurück.

Formale Syntax von Prolog-68

2

> Gewöhnlich glaubt der Mensch, wenn er nur Worte hört,
> Es müsse sich dabei doch auch was denken lassen. —*Faust [15]*

Dieses Kapitel beschreibt die *Syntax* von Prolog-68. Eine Besonderheit von Prolog (wie auch von anderen nicht-prozeduralen Sprachen wie LISP oder Smalltalk) ist, daß programmiersprachliche Konstrukte syntaktisch nicht besonders hervorgehoben werden. Eine typische Syntaxregel von Pascal ist z.B.

⟨*if statement*⟩ ⟶ `if` ⟨*expression*⟩ `then` ⟨*statement*⟩
| `if` ⟨*expression*⟩ `then` ⟨*statement*⟩ `else` ⟨*statement*⟩
□

Diese Regel definiert eine syntaktische Form für ein Konstrukt, das nur in dieser Form geschrieben werden darf. Außerdem darf kein anderes Konstrukt dieselbe Form besitzen; man kann in Pascal noch nicht einmal die Bezeichner `if`, `then` oder `else` vereinbaren.

In den oben erwähnten Sprachen werden solche Konstrukte *einheitlich* geschrieben, z.B. als Liste (LISP), Botschaft (Smalltalk) oder Prädikat. Die besondere Bedeutung ergibt sich erst aus der Semantik, die der Liste `(IF` ⟨*expr*⟩ ⟨*expr*⟩`)` oder der Botschaft Boolean ifTrue: aBlock ifFalse: aBlock zugeordnet ist. Diese Zuordnung erfolgt sogar meistens erst zur Laufzeit, wogegen in Pascal die Bedeutung des ⟨*if statement*⟩ mit der Sprachdefinition ein für allemal festgelegt wurde.

In Prolog sind alle Daten und alle Teile eines Programms *Terme*. Diese bestehen aus *Variablen*, *Konstanten* und *Strukturen*, wobei sich letztere in kleinere Einheiten zerlegen lassen, die wieder Terme sind. Die ziemlich komplizierte Syntax von Prolog verschleiert diesen einfachen Aufbau, weil sie viele Abkürzungen und Schreibweisen für bestimmte Sorten von Termen (z.B. Listen) zuläßt. Der folgende Abschnitt beschreibt die verschiedenen Termarten und Schreibweisen.

Die genaue Syntax der Terme wird in Abschnitt 2.3 definiert. Abschnitt 2.4 beschreibt, wie ein Prolog-Programm aus einzelnen Termen aufgebaut ist. Das Kapitel wird abgeschlossen durch Abschnitt 2.5.1, der bestimmte Einschränkungen für die Bildung von Termen erläutert. In diesem Abschnitt erfolgt außerdem ein Vergleich mit der Syntax einiger anderer Prolog-Dialekte.

2.1 Datenstrukturen in Prolog

Wie gesagt sind alle Datenstrukturen in Prolog *Terme*. Diese entsprechen ziemlich genau den Termen aus der formalen Logik, die im vorigen Kapitel verwendet wurden (siehe Definition 1).

Es gibt verschiedene Unterarten von Termen. Zunächst einmal ist zu unterscheiden zwischen *einfachen* und *zusammengesetzten* Termen. Zusammengesetzte Terme entsprechen in etwa den „records“ von Pascal oder den „structures“ von C; sie können im Prinzip beliebig tief verschachtelt werden, so daß sich sehr komplexe Strukturen bilden lassen.

2.1.1 Einfache Terme

Einfache Terme lassen sich nicht mehr weiter zerlegen. Prolog kennt zwei Sorten einfacher Terme, *Variablen* und *Konstanten*.

Variablen dienen als Platzhalter für Terme, die unbekannt sind bzw. erst noch berechnet werden müssen. Sie entsprechen eher den Variablen, die in der Logik oder Algebra verwendet werden, als den aus imperativen Programmiersprachen her bekannten Programmvariablen. Prolog kennt keine Zuweisungsoperation; eine Variable kann nur ein einziges Mal einen Wert erhalten, indem sie bei einer Unifikation *gebunden* wird. Von diesem Zeitpunkt an ist sie identisch mit ihrem Wert, also nicht mehr als Variable erkennbar.

Jede in einem Term vorkommende Variable hat einen Namen. In Prolog-68 müssen Variablennamen mit einem Großbuchstaben oder dem Zeichen `_` beginnen; der Rest des Namens darf nur Buchstaben (groß oder klein), Ziffern und `_` enthalten. Beispiele für gültige Variablennamen sind

```
A   X1   _3141   NextCharacter   _A_Long_But_Valid_Variable_Name
```

Konstanten sind unveränderliche Daten. Es gibt in Prolog zwei Arten von Konstanten, *Zahlen* und *Atome*. Prolog-68 kennt nur Ganzzahlen; einige andere Prolog-Dialekte können auch Fließkommazahlen verarbeiten.

Zahlen können auf verschiedene Weise angegeben werden. Positive Zahlen werden ohne Vorzeichen geschrieben, negative Zahlen mit einem `-`. Die eigentliche Zahl kann in einer beliebigen Basis zwischen 2 und 36 geschrieben werden:

```
<Basis>'<Zahl>
```

Bei Dezimalzahlen braucht die Basis 10 nicht extra angegeben werden; in diesem Fall schreibt man nur die `<Zahl>`. Für Basen über 10 werden die Buchstaben `A`, `B`, `C` ... (oder die entsprechenden Kleinbuchstaben) als zusätzliche Ziffern verwendet. Einen Sonderfall bildet die „Basis“ 0. Wird sie verwendet, dann ist der Wert der Zahl gleich dem ASCII-Wert des ersten Zeichens unmittelbar hinter dem Apostroph.

Die folgenden Beispiele sind allesamt korrekte Schreibweisen für die Antwort auf die letzte Frage nach dem Leben, dem Universum und überhaupt allem [2]:

```
42   2'101010   6'110   21'K   0'*
```

Atome sind Zeichenketten, die als Einheit betrachtet werden. Sie entsprechen nicht ganz den Zeichenketten in anderen Programmiersprachen (die meistens Felder von einzelnen Zeichen darstellen). Es gibt verschiedene Möglichkeiten, solche Zeichenketten anzugeben:

- Ein *Name* muß mit einem Kleinbuchstaben beginnen und darf danach nur Buchstaben, Ziffern und das Zeichen `_` enthalten.
- Ein *Symbol* ist ein Atom, das nur aus folgenden Zeichen besteht:

  ```
  #   $   &   *   +   -   .   /   :
  <   =   >   ?   @   \   ^   `   ~
  ```

- Die folgenden speziellen Zeichen und Zeichenkombinationen sind Atome:

  ```
  ;   !   []   {}
  ```

- Beliebige andere Zeichenketten können als Atome verwendet werden, wenn sie in Apostrophen eingeschlossen werden. Beispiele dafür sind die Atome

```
'Atom'   '0'   'zweiundvierzig'   'What''s up, man ?'
```

Das letzte dieser Beispiele zeigt, daß im Innern einer solchen Zeichenkette auch Apostrophen vorkommen dürfen; sie müssen dort allerdings (wie in Pascal) verdoppelt werden.

Falls gewünscht, können auch die aus der Sprache C bekannten „Escape-Sequenzen“ in einer solchen Zeichenkette verwendet werden; siehe dazu Abschnitt 2.1.3.

2.1.2 Zusammengesetzte Terme

Zusammengesetzte Terme besitzen einen *Funktor* und ein oder mehrere Argumente. Der Funktor wird gekennzeichnet durch seinen *Namen* (immer ein Atom) und durch seine *Stelligkeit* (die Anzahl der Argumente); die Argumente dürfen beliebige Terme sein. Man kann einen zusammengesetzten Term als Element eines neuen Datentyps ansehen, bei dem der Name des Funktors den Typ angibt; der Term

```
circle(point(0, 0), 1)
```

entspricht unter dieser Interpretation der C-Struktur

```
typedef struct {
   int x, y;
} point;

typedef struct {
   point center;
   int radius;
} circle;

circle c = { { 0, 0 }, 1 };
```

Wenn von einem Funktor gesprochen wird, ohne daß aus dem Zusammenhang seine Stelligkeit ersichtlich wird, dann wird diese meistens an den Namen angehängt; beispielsweise enthält das obige Beispiel die Funktoren `circle/2` und `point/2`.[1]

Wie bereits bei den Atomen gibt es auch für zusammengesetzte Terme eine Vielzahl von Schreibweisen. Die einfachste ist die *Präfix-* oder *kanonische* Schreibweise, die im obigen Beispiel bereits verwendet wurde. Dabei wird der Funktor zuerst geschrieben; die einzelnen Argumente werden als Liste (eingeschlossen in runde Klammern und durch Kommata getrennt) angegeben. Weitere Beispiele für solche Terme sind:

```
f(a, g(X, b), X)   p(1)   .(0, .(1, .(2, [])))   +(X, Y)
```

Wegen der Notwendigkeit, zu jeder öffnenden auch eine schließende Klammer anzugeben, können komplexere Terme in dieser Schreibweise recht unübersichtlich werden (der bekannte LISP-Effekt). Daher kennen die meisten Prolog-Dialekte (natürlich auch Prolog-68) abkürzende Schreibweisen zur Einsparung von Klammern, nämlich die *Listen-* und die *Operatornotation*.

Listen sind ein Spezialfall zusammengesetzter Terme. Eine nichtleere Liste ist ein zusammengesetzter Term mit Funktor `'.'/2`. Das erste Argument ist der *Listenkopf*, das zweite der *Listenrumpf*. Der Rumpf einer Liste muß normalerweise ebenfalls eine Liste darstellen (es gibt spezielle Programmiertechniken, bei denen Ausnahmen von dieser Regel auftreten). Die leere Liste wird durch das weiter oben erwähnte spezielle Atom `[]` repräsentiert.

[1] Im Englischen ist dafür die Sprechweise “`circle` of 2 arguments” oder kürzer “`circle` of 2” gebräuchlich.

Da Listen in Prolog (wie in LISP und anderen KI-Sprachen) sehr häufig vorkommen, wurde eine besondere Notation dafür eingeführt. Die Listen

```
.(Listenkopf, Listenrumpf)   .(0, .(1, .(2, [])))
```

können in dieser Schreibweise wie folgt geschrieben werden:

```
[Listenkopf | Listenrumpf]   [0 | [1 | [2 | []]]]
```

Es ist leicht zu sehen, daß diese Schreibweise noch keine Verbesserung des Schreibaufwands oder der Verständlichkeit ergibt. Dazu führen erst die beiden folgenden Vereinfachungsregeln:

- Eine Liste der Form `[X1 | [X2 | ... [Xn | Y] ... ]]` darf auch in der einfacheren Form `[X1, X2, ..., Xn | Y]` geschrieben werden.
- In einer Liste der Form `[X1, X2, ..., Xn | []]` darf das „`| []`“ weggelassen werden; man schreibt dafür nur noch `[X1, X2, ..., Xn]`.

Damit schreibt sich zumindest das zweite obige Beispiel leichter, es wird zu der erheblich kürzeren Liste `[0, 1, 2]`. Die verschachtelten Klammern verschwinden, und es ergibt sich ein etwas vertrauterer Anblick.[2]

Ein weiterer sehr häufig vorkommender Spezialfall sind Listen, die auschließlich aus ASCII-Zeichen bestehen.[3] Diese können ebenfalls abgekürzt angegeben werden, indem man die entsprechende Zeichenkette in Anführungszeichen setzt. Die folgenden Terme sind damit identisch:

```
"Yow"   [0'Y, 0'o, 0'w]   .(89, .(111, .(119, [])))
```

Innerhalb einer solchen Zeichenkette vorkommende Anführungszeichen müssen verdoppelt werden. Auch „Escape-Sequenzen“ können angegeben werden (siehe 2.1.3).

Operatornotation ist die zweite Möglichkeit, beim Schreiben von Termen die Anzahl der Klammern zu reduzieren. Operatoren gibt es auch in anderen Programmiersprachen (wie Pascal oder C); dort werden sie hauptsächlich zum Schreiben von arithmetischen Ausdrücken benutzt. Prolog unterscheidet sich in zwei Punkten von solchen Sprachen:

- In Prolog kann der Benutzer seine eigenen Operatoren definieren. Auch die „eingebauten“ Operatoren der Sprache dürfen – bis auf wenige Ausnahmen – umdefiniert werden.
- In Prolog führt ein „Operator“ keine wie auch immer geartete Operation durch – keine Berechnung, gar nichts. Er dient einzig und allein zur Konstruktion eines zusammengesetzten Terms (wo er den Funktor darstellt), hat also rein syntaktische Bedeutung.

Die Operatorschreibweise ist erheblich flexibler (und komplizierter) als die Listenschreibweise. Es gibt zunächst einmal drei verschiedene *Typen* von Operatoren, nämlich *Präfix-*, *Infix-* und *Postfixoperatoren*. Präfix- und Postfixoperatoren besitzen jeweils ein einziges Argument, wobei Präfixoperatoren davor, Postfixoperatoren aber dahinter geschrieben werden. Infixoperatoren haben zwei Argumente und werden dazwischen gestellt. Beispiele für alle drei Typen sind:

[2] Für LISP-Kenner, die aber die Kommata für überflüssig halten und die häßlichen eckigen Klammern nicht mögen werden. Tatsächlich entspricht die Listennotation bis auf diese Kleinigkeiten genau der LISP-Notation, auch die Datenstruktur ist identisch. Sogar die „dotted pairs“ von LISP sind vorhanden: `(a b c . d)` ist identisch mit `[a, b, c | d]`.

[3] In Prolog werden sie etwas irreführend „Strings“ oder „Zeichenketten“ genannt; sie sind *keine* Atome, die zwar nach außen hin durch Zeichenketten repräsentiert werden, intern jedoch unteilbare Einheiten darstellen.

```
- X      3!      a + b
```

Kombiniert man mehrere Operatoren miteinander, so muß irgendwie entschieden werden, in welcher Reihenfolge sie zum Aufbau eines Terms herangezogen werden. Ohne zusätzliche Informationen ließe sich sonst z.B. der Term `1 + 2 * 3 - 4` auf folgende verschiedenen Arten interpretieren:

```
+(1, *(2, -(3, 4)))   +(1, -(*(2, 3), 4))   *(+(1, 2), -(3, 4))
-(+(1, *(2, 3)), 4)   -(*(+(1, 2), 3), 4).
```

In Prolog wird dieses Problem so gelöst wie in anderen Sprachen: jeder Operator erhält eine *Priorität*, anhand derer der Parser entscheidet, wie Kombinationen von Operatoren ausgewertet werden. Solche Prioritäten haben in Prolog Werte von 1 bis 1200; Terme, die nicht mit Operatoren gebildet werden, besitzen die Priorität 0. Beispielsweise haben im Normalfall die Operatoren `'+'/2`, `'*'/2` und `'-'/2` die Prioritäten 500, 400 und 500. Die Regel, nach der die Bindungsreihenfolge entschieden wird, ist einfach: Eine kleinere Priorität bedeutet, daß der entsprechende Operator einen kleineren Term konstruiert.[4] Im obigen Beispiel verbleiben damit nur noch die Interpretationen

```
+(1, -(*(2, 3), 4))   -(+(1, *(2, 3)), 4),
```

da in den anderen Fällen der `*`-Operator größere Terme bildet als `+` bzw. `-`, was aber mit den gegebenen Prioritäten nicht möglich ist.

Es bleibt noch die Frage, was mit Operatoren gleicher Priorität geschieht. Auch hier wird auf eine bekannte Lösung zurückgegriffen: neben der Priorität besitzt jeder Operator noch eine *Assoziativität*, die angibt, wie er sich zu Operatoren der gleichen Priorität verhält. Normalerweise sind die Operatoren `'+'/2` und `'-'/2` in Prolog *linksassoziativ*, so daß sich eine eindeutige Darstellung für den Term `1 + 2 * 3 - 4` ergibt:

```
-(+(1, *(2, 3)), 4)
```

Die oben gemachte Aussage, es gäbe in Prolog drei Operatortypen, muß unter Berücksichtigung der Assoziativitäten etwas erweitert werden. In Wirklichkeit kennt Prolog nämlich die folgenden sieben Arten von Operatoren, die durch die angegebenen Codebezeichnungen unterschieden werden:

fx Nichtassoziative Präfixoperatoren,

fy assoziative Präfixoperatoren,

xfx nichtassoziative Infixoperatoren,

xfy rechtsassoziative Infixoperatoren,

yfx linksassoziative Infixoperatoren,

xf nichtassoziative Postfixoperatoren und

yf assoziative Postfixoperatoren.

Die angegebenen Codes (`fx`, `xfy` etc.) dienen einmal zur Angabe des Typs bei der Definition eines neuen Operators, zum anderen geben sie auch eine Merkhilfe, indem sie den Aufbau eines Terms symbolisch darstellen. Der Buchstabe `f` (für „Funktor“) gibt die Stellung des Operators in Beziehung auf das oder die Argumente an, `x` und `y`

[4] Leider entspricht das dem genauen Gegenteil dessen, was man erwarten sollte; es bedeutet nämlich, daß ein Operator mit kleiner Priorität *stärker* bindet als ein solcher mit großer.

stehen jeweils für ein einzelnes Argument. Die Angabe von `x` bedeutet dabei, daß das Argument eine niedrigere Priorität haben muß als der Operator, bei `y` darf es auch dieselbe Priorität wie der Operator besitzen. Im Falle eines `yfx`-Operators mit Priorität 42 heißt das, daß sein linkes Argument eine Priorität im Bereich 0–42 haben darf, das rechte dagegen nur 0–41.

Bei scharfem Hinsehen erkennt man, daß die Kombination `yfy`, die ja theoretisch möglich wäre, in der Liste fehlt. Das hat den einfachen Grund, daß damit die Mehrdeutigkeiten, die die Assoziativitätsangabe beseitigen soll, doch wieder auftreten würden.

2.1.3 Sonderzeichen in Atomen und Strings

Im Normalfall wird bei der Angabe von Atomen oder Strings keine besondere Verarbeitung durchgeführt, so daß z.B. folgende Terme identisch sind:

```
"\n\\"   [0'\, 0'n, 0'\, 0'\]
```

Prolog-68 kann jedoch in einen besonderen Modus versetzt werden, in dem es bei der Ein- und Ausgabe von Termen „Escape-Sequenzen“ benutzt. Dazu dienen die Aufrufe

```
:- prolog_flag(character_escapes, _, 1).    % Escapesequenzen an

:- prolog_flag(character_escapes, _, 0).    % Normalzustand
```

Ist der „Escape-Modus“ aktiv, so werden beim Einlesen von Atomen, Strings und Zahlen in Basis 0 folgende Zeichenkombinationen in die angegebenen Sonderzeichen umgewandelt:

`\a` in das Zeichen „(Audible) Bell“ (ASCII 7);

`\b` in das Zeichen „Backspace“ (ASCII 8);

`\t` in das Zeichen „Horizontal Tab“ (ASCII 9);

`\n` in das Zeichen „Line Feed“ (ASCII 10);

`\v` in das Zeichen „Vertical Tab“ (ASCII 11);

`\f` in das Zeichen „Form Feed“ (ASCII 12);

`\r` in das Zeichen „Carriage Return“ (ASCII 13);

`\e` in das Zeichen „Escape“ (ASCII 27);

`\s` in das Zeichen „Space“ (ASCII 32);

`\d` in das Zeichen „Delete“ (ASCII 127);

`\^?` ebenfalls in das Zeichen „Delete“;[5]

`\^`*x* in das Zeichen *x* mod 32, wobei *x* ein beliebiges ASCII-Zeichen außer „?“ sein darf;

`\`*ooo* in das ASCII-Zeichen, dessen oktaler Code *ooo* ist; dabei dürfen 1–3 Ziffern angegeben werden.

[5] Aus Gründen der Kompatibilität; logisch ist das nicht!

Anstelle der Kleinbuchstaben `a`, `b`, `t`, ... können auch die entsprechenden Großbuchstaben benutzt werden. Alle anderen Kombinationen von `\` mit einem folgenden Zeichen werden durch eben dieses Folgezeichen ersetzt, mit drei Ausnahmen: Handelt es sich um ein Steuer-oder Leerzeichen (einschließlich „Delete“), so wird die Kombination ignoriert; die Sequenz `\c` wird ebenfalls ignoriert, wobei allerdings auch noch alle folgenden Steuerzeichen übersprungen werden. Damit können Atome und Strings auch über das Zeilenende hinaus geschrieben werden; beispielsweise sind die zwei folgenden Atome identisch:

```
'FORTRAN␣sucks␣!'
'FOR\
TRAN␣\c
␣␣␣␣␣␣␣␣␣␣␣sucks\␣␣!'
```

Die Kombination `\z` bzw. `\Z` schließlich wird durch das Dateiendezeichen (-1) ersetzt; ihre Verwendung ist nur bei der Angabe einzelner Zeichen sinnvoll.

Bei der Ausgabe von Atomen werden Sonderzeichen nur im Escape-Modus in ihrer \-Form ausgegeben; dabei wird die Oktalschreibweise benutzt, wenn für ein Zeichen keine spezielle Kombination existiert. Normalerweise werden außer `'` bei der Ausgabe keine Zeichen umgewandelt.

2.2 Notation

In den folgenden Abschnitten beschreiben wir die verschiedenen Elemente eines Prolog-Programms mit Regeln einer (nicht ganz kontextfreien) Grammatik, die folgendermaßen aufgebaut sind:

Definiertes Symbol $\longrightarrow$ 1. Alternative
| 2. Alternative
⋮ ⋮
| letzte Alternative
□

In den Alternativen können *terminale Symbole* und *nicht-terminale Symbole* auftreten. Terminale Symbole sind die Elemente, aus denen letztlich das definierte Konstrukt besteht und die der Benutzer eingibt. Sie werden umrahmt und in `Maschinenschrift` dargestellt; nichtdruckbare Zeichen wie ␣ werden durch ihre ASCII-Namen ersetzt. In der weiter oben angegebenen Grammatikregel von Pascal sind `if`, `then` und `else` terminale Symbole.

Nicht-terminale Symbole bezeichnen bestimmte Untereinheiten der Sprache wie z.B. Konstanten, Terme oder Klauseln. Wir schreiben sie in *Kursivschrift* und mit ⟨spitzen Klammern⟩. Das definierte Symbol einer grammatischen Regel ist ein nicht-terminales Symbol. In der obigen Pascal-Regel sind ⟨*if statement*⟩, ⟨*expression*⟩ und ⟨*statement*⟩ nicht-terminale Symbole. Das besondere nicht-terminale Symbol ε steht für eine leere Folge von Symbolen.

Zur Abkürzung vereinbaren wir, daß ein nicht-terminales Symbol *Parameter* erhalten kann, die als Subskripte an den Namen des Symbols angehängt werden. Ein Beispiel dafür sind die beiden (nicht besonders sinnvollen) Regeln

$\langle Test_n \rangle \longrightarrow$ `*` $\langle Test_{n-1} \rangle$
□

$\langle Test_0 \rangle \longrightarrow$ `#`
□

Die zweite Regel dient dazu, eine „Verankerung“ für die Rekursion zu bilden. Solche Regeln entsprechen den *Grammatikregeln* von Prolog.

Zur besseren Lesbarkeit verwenden wir außerdem einige Abkürzungen für besonders häufig vorkommende Fälle. Für das Konstrukt

⟨*optionales* X⟩ ⟶ X
| ε
□

dürfen wir überall dort, wo das nicht-terminale Symbol ⟨*optionales* X⟩ benutzt wird, stattdessen das Kürzel [X] schreiben. X ist dabei eine beliebige Folge von Symbolen.

Die beliebig häufige Wiederholung eines Symbols oder einer Symbolfolge wird formal dargestellt als

⟨*beliebig viele* X⟩ ⟶ X ⟨*beliebig viele* X⟩
| ε
□

Stattdessen schreiben wir { X }; auch hier ist X eine beliebige Symbolfolge.

Wir können die Konstrukte [. . .] und { . . . } auch ineinander verschachteln. So definiert z.B. die Regel

⟨*Beispiel*⟩ ⟶ a [b { c }]
□

die Sprache, die aus den Worten `a`, `ab`, `abc`, `abcc`, . . . besteht.

Kommentare zu einzelnen Regeln legen bestimmte einschränkende Bedingungen fest, wie im folgenden Beispiel:

⟨*Alle Zeichen außer* ∗⟩ ⟶ ⟨*Zeichen*⟩
– ⟨*Zeichen*⟩ muß ungleich ∗ sein.
□

Weitere Einschränkungen können im begleitenden Text erwähnt werden. In den meisten Fällen dienen solche Einschränkungen nur zur Vereinfachung der angegebenen Regeln.

2.3 Syntax von Termen

In diesem Abschnitt definieren wir die Syntax von Termen, also fangen wir auch mit der entsprechenden Regel an:

⟨*Term*⟩ ⟶ ⟨$Term_{1200}$⟩
– Alle Operatoren haben Prioritäten ≤ 1200.
□

Terme können in *Operatornotation* geschrieben werden. Die folgenden Regeln sind nicht ganz vollständig, weil derselbe Operator gleichzeitig mehrere Typen haben kann, und weil mehrdeutige Kombinationen von zwei oder mehr Operatoren möglich sind. Der Parser von Prolog-68 erkennt alle Fälle, die sich allein aus dem Vergleich zweier Operatoren entscheiden lassen; im Fall einer mehrdeutigen Kombination wird die Alternative gewählt, die am wahrscheinlichsten zu einem korrekten Ergebnis führt.[6]

⟨$Term_n$⟩ ⟶ ⟨$Term_{n-1}$⟩
| ⟨*Präfix-*$Term_n$⟩
| ⟨*Infix-*$Term_n$⟩
| ⟨*Postfix-*$Term_n$⟩
□

[6] Nach Ansicht des Autors.

$\langle \mathit{Präfix\text{-}Term}_n \rangle \longrightarrow \left\langle \mathit{Operator}_{\mathrm{fx},n} \right\rangle \langle \mathit{Term}_{n-1} \rangle$
| $\left\langle \mathit{Operator}_{\mathrm{fy},n} \right\rangle \langle \mathit{Term}_n \rangle$
□

$\langle \mathit{Infix\text{-}Term}_n \rangle \longrightarrow \langle \mathit{Term}_{n-1} \rangle \left\langle \mathit{Operator}_{\mathrm{xfx},n} \right\rangle \langle \mathit{Term}_{n-1} \rangle$
| $\langle \mathit{Term}_{n-1} \rangle \left\langle \mathit{Operator}_{\mathrm{xfy},n} \right\rangle \langle \mathit{Term}_n \rangle$
| $\langle \mathit{Term}_n \rangle \left\langle \mathit{Operator}_{\mathrm{yfx},n} \right\rangle \langle \mathit{Term}_{n-1} \rangle$
□

$\langle \mathit{Postfix\text{-}Term}_n \rangle \longrightarrow \langle \mathit{Term}_{n-1} \rangle \left\langle \mathit{Operator}_{\mathrm{xf},n} \right\rangle$
| $\langle \mathit{Term}_n \rangle \left\langle \mathit{Operator}_{\mathrm{yf},n} \right\rangle$
□

$\left\langle \mathit{Operator}_{T,n} \right\rangle \longrightarrow \langle \mathit{Atom} \rangle$
– $T \in \{\mathrm{fx}, \mathrm{fy}, \mathrm{xfx}, \mathrm{xfy}, \mathrm{yfx}, \mathrm{xf}, \mathrm{yf}\}, 1 \leq n \leq 1200$
Das $\langle \mathit{Atom} \rangle$ muß als Operator vereinbart worden sein.
□

Es gibt einige spezielle Operatoren, die nicht umdefiniert werden können (sie sind ein integraler Bestandteil der Sprache):

$\langle \mathit{Term}_{1100} \rangle \longrightarrow \langle \mathit{Term}_{1099} \rangle$ `|` $\langle \mathit{Term}_{1100} \rangle$
– Das Zeichen `|` ist eine alternative Schreibweise
für den Operator `';'/2` (mit Priorität 1100).
□

$\langle \mathit{Term}_{1000} \rangle \longrightarrow \langle \mathit{Term}_{999} \rangle$ `,` $\langle \mathit{Term}_{1000} \rangle$
– Der Operator `','/2` hat Priorität 1000.
□

Die folgende Regel definiert Terme, auf deren äußerster Ebene kein Operator verwendet wird:

$\langle \mathit{Term}_0 \rangle \longrightarrow$ `(` $\langle \mathit{Term} \rangle$ `)`
| `{` $\langle \mathit{Term} \rangle$ `}`
– Der Term `{ t }` ist gleich dem Term `'{}'( t )`.
| $\langle \mathit{Atom} \rangle \left[\text{`(`} \langle \mathit{Termliste} \rangle \text{`)`} \right]$
– Die Klammer muß *ohne Zwischenraum* auf das $\langle \mathit{Atom} \rangle$ folgen.
| $\langle \mathit{Liste} \rangle$
| $\langle \mathit{String} \rangle$
| $\langle \mathit{Ganzzahl} \rangle$
| $\langle \mathit{Variable} \rangle$
□

$\langle \mathit{Liste} \rangle \longrightarrow$ `[` [$\langle \mathit{Listeninhalt} \rangle$] `]`
□

$\langle \mathit{Listeninhalt} \rangle \longrightarrow \langle \mathit{Termliste} \rangle \left[\text{`|`} \langle \mathit{Term}_{999} \rangle \right]$
□

$\langle \mathit{Termliste} \rangle \longrightarrow \langle \mathit{Term}_{999} \rangle \left\{ \text{`,`} \langle \mathit{Term}_{999} \rangle \right\}$
□

Ein *String* ist eine Abkürzung für eine Liste von ASCII-Werten. Der String `"Gulp"` ist gleich der Liste `[71, 117, 108, 112]`. Die Stringnotation ist „syntaktischer Zucker“ und dient zur Erleichterung der Eingabe solcher Listen.

⟨*String*⟩ ⟶ `"` { ⟨*S-Zeichen*⟩ } `"`
□

⟨*S-Zeichen*⟩ ⟶ `""`
| ⟨ *Jedes Zeichen außer* `"` ⟩
□

Atome sind – wie der Name schon sagt – unteilbare Einheiten. Zusammen mit den Zahlen bilden die Atome die *Konstanten* von Prolog. Es gibt verschiedene Arten, ein Atom zu bilden:

⟨*Atom*⟩ ⟶ ⟨*Name*⟩
| ⟨*Beliebige Zeichenkette*⟩
| ⟨*Symbol*⟩
| ⟨*Einzelzeichen*⟩
| `[]`
| `{}`
□

⟨*Name*⟩ ⟶ ⟨*Kleinbuchstabe*⟩ { ⟨*Alphanumerisches Zeichen*⟩ }
□

⟨*Beliebige Zeichenkette*⟩ ⟶ `'` { ⟨*Q-Zeichen*⟩ } `'`
□

⟨*Q-Zeichen*⟩ ⟶ `''`
| ⟨ *Jedes Zeichen außer* `'` ⟩
□

⟨*Symbol*⟩ ⟶ ⟨*Symbolzeichen*⟩ { ⟨*Symbolzeichen*⟩ }
□

Die zweite Art von Konstanten in Prolog sind *Zahlen*. Prolog-68 kennt nur ganze Zahlen (es gibt andere Dialekte, die auch Fließkommazahlen verarbeiten können). Zahlen können in beliebiger Basis (zwischen 2 und 36) angegeben werden, ASCII-Werte auch durch direkte Angabe des entsprechenden Zeichens. Die Liste im obigen String-Beispiel läßt sich damit als `[0'G, 0'u, 0'l, 0'p]` darstellen.

⟨*Ganzzahl*⟩ ⟶ [`-`] ⟨*Positive Ganzzahl*⟩
– Der erlaubte Wertebereich ist $[-67108864 \ldots 67108863]$.
□

⟨*Positive Ganzzahl*⟩ ⟶ ⟨*Dezimalzahl*⟩
| ⟨$Basis_n$⟩ `'` ⟨$Positive\ Ganzzahl_n$⟩
| `0'` ⟨*Zeichen*⟩
– Der Wert der Zahl ist der ASCII-Wert des ⟨*Zeichens*⟩.
□

⟨*Dezimalzahl*⟩ ⟶ ⟨$Positive\ Ganzzahl_{10}$⟩
□

⟨$Basis_n$⟩ ⟶ ⟨*Dezimalzahl*⟩
– Der Wert der ⟨*Dezimalzahl*⟩ ist n mit $2 \leq n \leq 36$.
□

⟨*Positive Ganzzahl*$_n$⟩ ⟶ ⟨*Ziffer*$_n$⟩ { ⟨*Ziffer*$_n$⟩ }
□

⟨*Ziffer*$_n$⟩ ⟶ ⟨*Dezimalziffer*⟩
| ⟨*Kleinbuchstabe*⟩
| ⟨*Großbuchstabe*⟩
– Die Buchstaben `a`–`z` bzw. `A`–`Z` gelten als Ziffern 10–35.
Für ein gegebenes n sind nur Ziffern $< n$ erlaubt.
□

Variablen haben Namen, die mit einem Großbuchstaben oder mit `_` (Unterstrich) beginnen. Normalerweise bezeichnet ein in einem Term an verschiedenen Stellen vorkommender Variablenname an jeder dieser Stellen dieselbe Variable; es gibt aber den Sonderfall der *anonymen Variablen*, deren Namen nur aus einem einzelnen Unterstrich bestehen. In einem Term können mehrere solche Variablen auftauchen, wobei alle so definierten Variablen voneinander verschieden sind. Abgesehen von dieser Besonderheit unterscheiden sich anonyme Variablen in keiner Weise von normalen Variablen.

Variablennamen in verschiedenen Termen bezeichnen auch verschiedene Variablen, solange diese Terme nicht unifiziert oder auf andere Weise manipuliert werden. Die Namen der Variablen werden normalerweise nicht gespeichert, so daß sie bei der Ausgabe von Termen nicht rekonstruiert werden können. Für Fälle, in denen die Erhaltung der Variablennamen wichtig ist, stehen spezielle Operationen zur Verfügung.

⟨*Variable*⟩ ⟶ `_` { ⟨*Alphanumerisches Zeichen*⟩ }
| ⟨*Großbuchstabe*⟩ { ⟨*Alphanumerisches Zeichen*⟩ }
□

Die Gesamtheit der ASCII-Zeichen wird in verschiedene *Zeichenklassen* unterteilt, die in den obigen Regeln bereits benutzt wurden:

⟨*Dezimalziffer*⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
□

⟨*Kleinbuchstabe*⟩ ⟶ ⟨*Zeichen*⟩
– ⟨*Zeichen*⟩ ∈ {`a` … `z`}.
□

⟨*Großbuchstabe*⟩ ⟶ ⟨*Zeichen*⟩
– ⟨*Zeichen*⟩ ∈ {`A` … `Z`}.
□

⟨*Alphanumerisches Zeichen*⟩ ⟶ ⟨*Dezimalziffer*⟩
| ⟨*Kleinbuchstabe*⟩
| ⟨*Großbuchstabe*⟩
| `_`
□

⟨*Symbolzeichen*⟩ ⟶
`#` | `$` | `&` | `*` | `+` | `-` | `.` | `/` | `:`
| `<` | `=` | `>` | `?` | `@` | `\` | `^` | `` ` `` | `~`
□

⟨*Einzelzeichen*⟩ ⟶ `;` | `!`
□

Die unteilbaren Untereinheiten, aus denen sich komplexere Terme zusammensetzen, heißen *Tokens*.[7] Leerzeichen, Kommentare etc. dürfen zwischen einzelnen Tokens auftreten, aber im Inneren eines Tokens sind sie nicht erlaubt.

⟨*Token*⟩ ⟶ ⟨*Atom*⟩
| ⟨*Variable*⟩
| ⟨*Ganzzahl*⟩
| ⟨*String*⟩
| `(` | `)` | `[` | `]` | `{` | `}` | `|` | `,`
| ⟨*Ende-Punkt*⟩
– Jeder Term muß mit einem ⟨*Ende-Punkt*⟩ abgeschlossen werden.
□

⟨*Ende-Punkt*⟩ ⟶ `.` ⟨*Leerzeichen*⟩
– Das ⟨*Leerzeichen*⟩ ist zwingend erforderlich.
□

⟨*Freiraum*⟩ ⟶ ⟨*Leerzeichen*⟩ { ⟨*Freiraum*⟩ }
| ⟨*Kommentar*⟩ { ⟨*Freiraum*⟩ }
□

⟨*Leerzeichen*⟩ ⟶ ⟨*Alle Zeichen mit ASCII-Werten unter 32*⟩
| `SPC`
| `DEL`
□

⟨*Kommentar*⟩ ⟶ ⟨*P-Kommentar*⟩
| ⟨*C-Kommentar*⟩
□

⟨*P-Kommentar*⟩ ⟶ `%` ⟨*Alle Zeichen bis zum Zeilenende*⟩
□

⟨*Alle Zeichen bis zum Zeilenende*⟩ ⟶ { ⟨*Zeichen*⟩ } `LF`
– ⟨*Zeichen*⟩ ist ungleich `LF`.
□

⟨*C-Kommentar*⟩ ⟶ `/*` ⟨*C-Kommentar-Zeichen*⟩ `*/`
□

⟨*C-Kommentar-Zeichen*⟩ ⟶ { ⟨*Zeichen*⟩ }
– Die Zeichenkombination `*/` darf nicht auftreten.
□

[7] Auch hier könnte man einen deutschen Ausdruck verwenden, aber wer möchte schon dauernd "kleinste syntaktische Untereinheit" schreiben? „Lexem" ist ein Kunstwort, da ist mir das alte englische "token" schon lieber.

2.4 Syntax von Klauseln und Direktiven

Wie bei der SLD-Resolution (siehe Kapitel 1) unterscheiden wir zunächst zwischen Programmklauseln und Anfragen. Da die Benutzerschnittstelle von Prolog-68 verschiedene Sorten von Anfragen verarbeitet, bezeichnen wir diese hier mit dem Oberbegriff „Direktiven“.

⟨*Klausel*⟩ ⟶ ⟨*Programmklausel*⟩
| ⟨*Direktive*⟩
□

Programmklauseln sind Fakten, Regeln oder Grammatikregeln. Die Besonderheit der Programmklauseln ist, daß sie ein positives Literal besitzen, den *Klauselkopf*.

⟨*Programmklausel*⟩ ⟶ ⟨*Faktum*⟩
| ⟨*Regel*⟩
| ⟨*Grammatikregel*⟩
□

⟨*Faktum*⟩ ⟶ ⟨*Kopf*⟩
– Der äußerste Funktor darf weder `':-'/2` noch `'-->'/2` sein.
□

⟨*Regel*⟩ ⟶ ⟨*Kopf*⟩ `:-` ⟨*Rumpf*⟩
□

Der *Kopf* einer Klausel ist ein Literal. Literale sind alle Terme außer Variablen, Zahlen und Listen.

⟨*Kopf*⟩ ⟶ ⟨*Literal*⟩
□

⟨*Literal*⟩ ⟶ ⟨*Term*⟩
– Weder eine ⟨*Variable*⟩, noch eine ⟨*Ganzzahl*⟩, noch eine ⟨*Liste*⟩.
□

Der *Rumpf* einer Klausel ist im Prinzip ein gewöhnlicher Term, aber normalerweise haben Klauselrümpfe eine bestimmte Struktur, die durch folgende Regeln beschrieben wird:

⟨*Rumpf*⟩ ⟶ ⟨*Rumpf-Alternative*⟩ { `;` ⟨*Rumpf-Alternative*⟩ }
□

⟨*Rumpf-Alternative*⟩ ⟶ ⟨*Aufruf*⟩ { `,` ⟨*Aufruf*⟩ }
□

⟨*Aufruf*⟩ ⟶ `(` ⟨*Rumpf*⟩ `)`
| ⟨*Literal*⟩
| ⟨*Meta-Aufruf*⟩
□

⟨*Meta-Aufruf*⟩ ⟶ ⟨*Variable*⟩
□

Grammatikregeln sind eine abkürzende Schreibweise für die im Compilerbau verwendeten *definite clause grammars*. Hier zeigt sich Prolog von seiner elegantesten Seite: Eine in BNF formulierte Grammatik kann 1:1 nach Prolog übersetzt werden, und das entstehende Programm ist ein kompletter Parser.[8]

[8] In der Realität sieht die Sache etwas komplizierter aus, aber für LL(1)-Grammatiken ist es wirklich so einfach.

⟨*Grammatikregel*⟩ ⟶ ⟨*Linke Seite*⟩ `-->` ⟨*Rechte Seite*⟩
□

⟨*Linke Seite*⟩ ⟶ ⟨*Nicht-terminales Symbol*⟩ [`,` ⟨*Kontext*⟩]
□

⟨*Kontext*⟩ ⟶ ⟨*Terminale Symbole*⟩
□

⟨*Rechte Seite*⟩ ⟶ ⟨*Regel-Alternative*⟩ { `;` ⟨*Regel-Alternative*⟩ }
□

⟨*Regel-Alternative*⟩ ⟶ ⟨*Regel-Element*⟩ { `,` ⟨*Regel-Element*⟩ }
□

⟨*Regel-Element*⟩ ⟶ `(` ⟨*Rechte Seite*⟩ `)`
| ⟨*Nicht-terminales Symbol*⟩
| ⟨*Terminale Symbole*⟩
| ⟨*Bedingung*⟩
| ⟨*Meta-Phrase*⟩
□

⟨*Nicht-terminales Symbol*⟩ ⟶ ⟨*Literal*⟩
□

⟨*Terminale Symbole*⟩ ⟶ ⟨*Liste*⟩
– Nur „geschlossene“ Listen sind erlaubt.
| ⟨*String*⟩
□

⟨*Bedingung*⟩ ⟶ `!`
| `{` ⟨*Term*⟩ `}`
□

⟨*Meta-Phrase*⟩ ⟶ ⟨*Variable*⟩
□

Direktiven sind in Prolog-68 entweder Kommandos oder Anfragen. Beide unterscheiden sich dadurch, daß nach der Abarbeitung einer Anfrage die entstehenden Variablenbelegungen ausgegeben werden. Außerdem kann der Benutzer bei einer Anfrage nach alternativen Lösungen suchen lassen; Kommandos werden dagegen immer deterministisch ausgeführt, d.h. sie liefern höchstens einmal ein Ergebnis (wenn überhaupt).

⟨*Direktive*⟩ ⟶ ⟨*Kommando*⟩
| ⟨*Anfrage*⟩
□

⟨*Kommando*⟩ ⟶ `:-` ⟨*Rumpf*⟩
□

⟨*Anfrage*⟩ ⟶ `?-` ⟨*Rumpf*⟩
– Bei interaktiver Eingabe wird der Funktor `?-` weggelassen.
□

2.5 Besonderheiten

2.5.1 Restriktionen

Prolog besitzt eine sehr flexible Syntax – zu flexibel, denn sie enthält einige Mehrdeutigkeiten. In Prolog-68 wird ein Parser benutzt, der nur eine eingeschränkte Version dieser Syntax verarbeiten kann. Die Einschränkungen sind aber nicht allzu schwerwiegend:

- Zwischen dem Funktor eines in Standardnotation geschriebenen Terms und dessen öffnender Klammer dürfen keine Leerzeichen oder Kommentare auftreten, wenn dieser Funktor als Operator definiert wurde. Wird dagegen ein Präfixoperator auf ein Argument angewandt, das mit einer öffnenden Klammer beginnt, so muß dazwischen mindestens ein Leerzeichen stehen.

- Soll der Name eines Operators (ein Atom) als Argument für einen Operator niedrigerer Priorität verwendet werden, so muß ersterer in Klammern eingeschlossen werden. Beispiel: `(+)/2` für den Term `/(+, 2)`.

- Der Operator `','/2`, der für die Abgrenzung von Argumenten in Strukturen und Listen benutzt wird, hat die Priorität 1000. Deshalb müssen Terme, deren Funktoren Operatoren mit Prioritäten ≥ 1000 sind, im Innern einer Struktur oder Liste in Klammern eingeschlossen werden. Die wichtigsten dieser Operatoren sind `':-'`, `'-->'`, `'?-'`, `';'`, `'->'` und `','`.

- Wird der Präfixoperator `'-'/1` direkt vor einer Ganzzahl geschrieben, so betrachtet der Parser ihn als Vorzeichen. Um das zu verhindern, kann der gewünschte Term als `-(Zahl)` oder `- (Zahl)` geschrieben werden; das `-` ist in diesem Fall der Funktor eines zusammengesetzten Terms.

  Frühere Versionen des Parsers interpretierten `-` generell als Vorzeichen, auch wenn es als Infix-Operator verwendet wurde. Das Resultat waren regelmäßig Syntaxfehler. Mit der neuen Parserversion ist diese Restriktion (endlich !) verschwunden, so daß `1-1` jetzt ein legaler Term ist.

- Kombinationen von Operatoren werden nur dann richtig verarbeitet, wenn die korrekte Struktur sich allein aus dem Vergleich zweier Operatoren und maximal eines weiteren Tokens ableiten läßt. Fehlerhafte Kombinationen werden immer erkannt, aber wenn mehrere legale Interpretationen möglich sind, wird eine davon willkürlich ausgewählt (der Parser ist deterministisch). Diese Vorgehensweise kann dazu führen, daß syntaktisch korrekte Terme fälschlicherweise abgelehnt werden.

  In der Praxis treten solche mehrdeutigen Kombinationen selten auf. Es hat nicht viel Sinn, sich um die Auflösung dieser Situationen zu bemühen, da schon in einfachen Fällen wie z.B. dem Term `-2/3` verschiedene Prolog-Systeme verschiedene Antworten liefern. Prolog-68 interpretiert diesen Term als `/(-2, 3)`, einige andere Systeme dagegen als `-(/(2, 3))`. Da die Standardsyntax keine Priorität für das als Vorzeichen verwendete `-` angibt, kann man beide Versionen als korrekt betrachten.[9]

- Jeder Operator darf nur je einen Präfix-, Infix- und Postfix-Typ besitzen. Es ist also z.B. nicht möglich, einen Operator zu definieren, der gleichzeitig die Typen `yfx` und `xfy` besitzt. Die Prioritäten der drei verschiedenen Typen dürfen sich unterscheiden.

  Das ist immerhin ein großer Fortschritt gegenüber dem alten Parser, der nur je einen unären und einen binären Typ mit der gleichen Priorität erlaubte.

[9] Auch der Autor hat in diesem Fall seine Betrachtungsweise mehrmals geändert ...

- Steht das Zeichen `.` direkt vor einem Leerzeichen, so gibt es das Ende eines Terms an (es ist dann ein ⟨*Ende-Punkt*⟩). Will man `.` als Symbol benutzen (z.B. um den Listenkonstruktor `'.'/2` als Operator zu definieren), so muß man es an das nachfolgende Token anschließen lassen. Zu Schwierigkeiten kann es kommen, wenn das nachfolgende Token seinerseits ein Symbol ist, da das Zeichen `.` in diesem Fall als Teil des Symbols betrachtet wird.

- Ganzzahlen dürfen nur in Basen von 2 bis 36 angegeben werden. Die Basis 1 ist nicht erlaubt (es dürfte auch schwierig sein, in dieser Basis eine Zahl $\neq 0$ anzugeben).

- Ein Symbol darf nicht mit der Zeichenkombination `/*` beginnen, die als Anfang eines Kommentars interpretiert wird. Direkt vor einem Kommentar mit `/*` darf kein Symbol stehen, da der Anfang des Kommentars sonst nicht erkannt wird.

- Eine Liste ist kein ⟨*Literal*⟩, deshalb kann der Listenkonstruktor `'.'/2` nicht als Name eines Prädikats verwendet werden.

2.5.2 Unterschiede zu TOY Prolog

TOY Prolog ist die in [21] beschriebene Implementierung von Prolog, die vor einiger Zeit als TOY Prolog ST von mir auf den Atari ST portiert wurde. Prolog-68 ist *keine* Nachfolgeversion von TOY Prolog ST, sondern ein eigenständiges System, bei dessen Entwurf die Kompatibilität zu weiter verbreiteten Prolog-Dialekten[10] wie Prolog-10 oder Quintus Prolog im Vordergrund stand. Dabei wurden einige „elegante“ Eigenschaften von TOY Prolog aufgegeben, andere kamen hinzu:

- In der vorliegenden Version von Prolog-68 wurde ein völlig neu konzipierter Parser verwendet, der praktisch alle Nachteile des Parsers von TOY Prolog beseitigt. Bis auf die oben geschilderte Restriktion für Operatoren, die als Funktoren von Termen in kanonischer Schreibweise verwendet werden, ist die neue Syntax zur alten aufwärtskompatibel.

- In TOY Prolog werden Zeichen nicht als ASCII-Werte, sondern als Atome dargestellt. Der String `"Prolog"` beispielsweise entspricht in TOY Prolog der Liste `[P, r, o, l, o, g]`, in Prolog-68 dagegen repräsentiert er die Liste `[80, 114, 111, 108, 111, 103]`.

- In TOY Prolog dürfen Zahlen nur im Dezimalsystem angegeben werden, außerdem ist der Wertebereich sehr beschränkt.[11]

- Anonyme Variablen in TOY Prolog verhalten sich reichlich seltsam, da sie niemals gebunden werden. Deshalb schlägt das folgende Programmstück in Prolog-68 fehl, während es in TOY Prolog erfüllt werden kann:

```
test(X) :- X = 0, var(X).
:-test(_).
```

- In TOY Prolog ST dürfen alle Sonderzeichen des ST-Zeichensatzes benutzt werden, wobei z.B. deutsche Umlaute auch als Buchstaben erkannt werden (so daß beispielsweise `ätsch` ein normales Atom ist).

[10] Ja, ich weiß – es gibt mindestens einen Public-Domain-Versand, der mein Programm vollmundig als „Prolog 10“ anpreist, nur weil ich unvorsichtigerweise in einer Dokumentation erwähnt habe, daß TOY Prolog einigermaßen kompatibel zu Prolog-10 sein *sollte*.

[11] 16 Bit. Prolog-68 benutzt 27 Bit (für die Neugierigen: 32 Bit weniger 2 Bit für den *garbage collector* weniger 3 Bit Typkennzeichen.)

Prolog-68 arbeitet intern mit dem erweiterten ASCII-Zeichensatz nach der Norm ISO 8859-1. Bei der Ein- und Ausgabe von Textzeichen wird eine automatische Umkodierung zwischen der internen Repräsentation und dem Atari-Zeichensatz vorgenommen; bei binärer Ein- und Ausgabe findet keine Umwandlung statt. Dieses Verfahren hat den Nachteil, daß weder der gesamte Atari-Zeichensatz noch der vollständige Zeichensatz nach ISO 8859-1 zugänglich sind, weil sie disjunkte Mengen darstellen; es gewährleistet jedoch eine maximale Portabilität, da viele Prolog-Systeme unter Unix ebenfalls den ISO-Zeichensatz benutzen.

- In TOY Prolog gibt es nur die „normalen“ Kommentare mit `%`, die das Ende einer Zeile bilden. Prolog-68 kennt auch Kommentare im PL/1-Stil (`/* ... */` – wer hat da eben ‚C‘ gesagt ?)

- In Grammatikregeln dürfen jetzt Variablen als Elemente auftauchen, genau wie in Klauseln. Eine solche Variable wird automatisch in einen Aufruf des Prädikats `phrase/3` übersetzt, so wie eine Variable in einer Klausel in einen Aufruf von `call/1` verwandelt wird.

  Grammatikregeln werden anders übersetzt als in TOY Prolog. Dafür gibt es mehrere Gründe:

  – Das alte Übersetzungsschema kann fehlerhaften Code erzeugen, wenn ’cuts’ in einer Regel auftreten [31].

  – Das neue Schema ist flexibler [31].

  – Das neue Schema ist kompatibel zu verbreiteteren Dialekten.

  – Den Präprozessor von TOY Prolog wollte ich aus urheberrechtlichen Gründen nicht einfach übernehmen.

2.5.3 Unterschiede zu Standard-Dialekten

Heutige Prolog-Dialekte kann man – wenn man einmal von Exoten wie micro-Prolog absieht – in zwei große Gruppen einteilen, die „Edinburgh-“ und die „Marseille-Dialekte“. Beide sind nach ihren Herkunftsorten benannt: In Edinburgh wurde Prolog-10 entwickelt, in Marseille Prolog II. Am weitesten verbreitet sind wohl die Dialekte, die sich von Prolog-10 ableiten, darunter z.B. C-Prolog, Quintus Prolog und auch Prolog-68, dessen Syntax sich nur in folgenden Punkten von den größeren Systemen unterscheidet:

- Ein Term der Form `[X | Y]` (eine „offene“ Liste) darf nicht in der (veralteten) Schreibweise `[X ,.. Y]` angegeben werden.

- `%(` und `%)` können nicht als Ersatz für `{` bzw. `}` verwendet werden (auch diese Schreibweise ist ein Anachronismus).

- Prolog-10 verwendet die ASCII-Zeichen CTRL-? (31) und CTRL-Z (26) als Kennzeichen für das Ende einer Zeile bzw. einer Datei. In Prolog-68 wird das Zeilenende durch das Zeichen LF (10) dargestellt, das Dateiende durch die Zahl -1. Damit folgt Prolog-68 dem Beispiel vieler Implementationen auf Unix-Basis.

Leerseite aus satztechnischen Gründen

Arbeiten mit Prolog-68 3

Dieses Kapitel ist noch in Arbeit. Tschuldigung ...

3.1 Programmaufruf

... this needs work ...

3.1.1 Optionen in der Kommandozeile

Zulässig sind die folgenden Optionen:

`-help` Gibt eine Kurzbeschreibung der zulässigen Optionen aus.

`-debug` Schaltet einen primitiven Testmodus ein, in dem die einzelnen Aufrufe des Prolog-Programms ausgegeben werden. Der Nutzen ist begrenzt, weil weder Erfolg noch Fehlschlag der einzelnen Aufrufe angezeigt werden. In der jetzigen Form dürfte der Autor der einzige sein, der mit dieser Option etwas anfangen kann.

`-code` *N* Vergibt Speicher für den *Codebereich* der abstrakten Prolog-Maschine. In diesem Bereich werden alle langlebigen Datenstrukturen (z.B. Atome, Funktoren und Klauseln) gespeichert.

Der Parameter *N* gibt an, wieviel Speicher zugewiesen werden soll; er wird im nächsten Unterabschnitt genauer erläutert.

`-global` *N* Vergibt Speicher für den *globalen Bereich*, in dem alle während des Programmablaufs entstehenden Terme abgelegt werden. Außerdem liegt in diesem Bereich der *trail stack*, der benutzt wird, um Variablenbindungen rückgängig zu machen. Die Bedeutung des zusätzlichen Parameters *N* wird unten erläutert.

`-local` *N* Vergibt Speicher für den *lokalen Bereich*, der zur Kontrolle des Programmablaufs und zum Speichern von lokalen Variablen benutzt wird. *N* hat dieselbe Bedeutung wie bei den beiden vorigen Optionen.

`-reserve` *N* Gibt Speicher ans Betriebssystem zurück. Die Größe des Speicherbereichs wird mit *N* angegeben (siehe unten).

3.1.2 Speicheraufteilung

... this needs MUCH work ...

Der beim Programmstart verfügbare Speicher kann über die Optionen `-code`, `-global`, `-local` und `-reserve` auf die verschiedenen Speicherbereiche des Prolog-Systems verteilt werden. Diese Optionen dürfen in beliebiger Reihenfolge angegeben werden und auch mehrfach vorkommen (wobei dann die einzelnen Werte summiert werden).

Alle vier Optionen müssen von einem zusätzlichen Parameter gefolgt werden, der die Größe des zu vergebenden Speicherbereichs angibt. Er besteht aus einer

Dezimalzahl, die von einem der Zeichen '`k`', '`K`', '`m`', '`M`' oder '`%`' gefolgt werden kann. Bei Angabe eines der Buchstaben oder beim Fehlen des zusätzlichen Zeichens wird die entsprechende Anzahl Bytes (bzw. KBytes oder MBytes) reserviert; bei Angabe von '`%`' wird stattdessen der angegebene Prozentsatz des noch nicht vergebenen Speichers benutzt. Aufeinanderfolgende Prozentangaben beziehen sich auf denselben Gesamtwert.

Fehlen in der Kommandozeile Angaben zu einem oder mehreren der drei Bereiche, so wird der gesamte nach Abarbeitung der Kommandozeile verbleibende Speicher auf die fehlenden Bereiche verteilt, und zwar im Verhältnis $1 : 2 : 4$ für Codebereich, lokalen und globalen Bereich.

Bleibt nach der Aufteilung Speicher übrig, so wird dieser an das Betriebssystem zurückgegeben. Das ist insbesondere dann wichtig, wenn von Prolog aus andere Programme aufgerufen werden sollen. Die Option `-reserve` tut im Prinzip dasselbe, erlaubt aber die explizite Angabe der Größe des zurückgegebenen Speicherbereichs.

Es folgen ein paar Beispiele, die die Flexibilität dieses Verfahrens verdeutlichen sollen. Angenommen sei, daß beim Programmstart genau 2 MByte (2^{20} Byte) an Hauptspeicher zur Verfügung stehen:

- `-code 1M -global 50% -local 25%`
  Der Codebereich erhält zunächst 1M Byte. Vom verbleibenden Speicher geht die Hälfte (512 KByte) an den globalen Bereich, ein Viertel (also 256 KByte) an den lokalen Bereich. Es verbleibt ein Rest von 256 KByte, der ans Betriebssystem zurückgeht.

- `-code 25% -local 512K -code 25%`
  Von den ursprünglichen 2 MByte wird zunächst ein Viertel für den Codebereich reserviert. Weitere 512 KByte gehen an den lokalen Bereich. Es verbleiben 1 MByte, von denen wiederum ein Viertel dem Codebereich zugeschlagen wird. Der restliche Speicher wird für den globalen Bereich verwendet, da für diesen keine Angabe gemacht wurde.

- *Keine Angabe.*
  $\frac{1}{7}$ des Speichers geht an den Codebereich, $\frac{2}{7}$ gehen an den lokalen Bereich. Die restlichen $\frac{4}{7}$ werden für den globalen Bereich verwendet. Es wird kein Speicher ans Betriebssystem zurückgegeben.

Fehlermeldungen A

> Beetle studierte den Druck mit gerunzelter Stirn. „Ich werde wirklich nicht schlau daraus“, murmelte er. „Was soll das denn bedeuten?“
>
> —*Stalky & Co. [19]*

Fehler, die bei der Arbeit mit Prolog-68 auftreten können, fallen in mehrere Kategorien, die verschiedene Arten von Meldungen erzeugen:

- *Interne Fehler* werden durch Programmierfehler im Prolog-System verursacht und können vom Benutzer nicht behoben werden. Solche Fehler sollten überhaupt nicht auftreten.

  In diese Gruppe fallen auch alle Fehler, die sich nicht durch eine Fehlermeldung, sondern auf andere Weise (Bomben, Endlosschleifen etc.) bemerkbar machen.

- *Fatale Fehler* entstehen durch falsche Handhabung des Programms oder durch die Überschreitung der Ressourcen des Rechners. Sie können nicht abgefangen werden und führen zum Abbruch des gesamten Prolog-Systems.

- *Syntaxfehler* brauchen wohl nicht näher erläutert zu werden.

- *Übersetzungsfehler* werden vom Compiler erkannt, während Klauseln in die Datenbank aufgenommen werden.

- *Laufzeitfehler* treten – wie der Name schon sagt – zur Laufzeit des Prolog-Programms auf.

- *Warnungen* weisen auf mögliche Mißverständnisse oder Schreibfehler hin. Sie haben keinen Einfluß auf den Ablauf des Programms.

Die verschiedenen Fehlerkategorien werden in den folgenden Abschnitten einzeln besprochen. Dabei werden Übersetzungs- und Laufzeitfehler sowie Warnungen zusammen behandelt, da die Abgrenzung zwischen diesen Gruppen etwas unscharf ist. Einzelne Fehlermeldungen werden daher in Abschnitt A.4 zusätzlich gekennzeichnet:

Ü bezeichnet Übersetzungsfehler,

L Laufzeitfehler und

W Warnungen.

A.1 Interne Fehler

Falls es Ihnen gelingt, eine der folgenden Fehlermeldungen reproduzierbar zu erzeugen, würde der Autor gerne von Ihnen hören.

`Can't boot (some *.WAM file may be corrupted).`

Dieser Fehler kann durch Manipulation an den Dateien entstehen, die die fertig übersetzten Systemprogramme (Compiler, Benutzerschnittstelle etc.) enthalten. Ohne Backups dieser Dateien kann der Fehler nicht behoben werden.

`Can't read PROLOG.LST.`
Die Datei `PROLOG.LST` konnte nicht gelesen werden. Sie enthält normalerweise eine Liste der Namen von vorübersetzten Dateien, die vor dem Start des Prolog-Systems eingelesen werden müssen. Wird diese Datei versehentlich gelöscht oder beschädigt, so kann sie mit einem Texteditor rekonstruiert werden.

`Illegal database reference.`
Kann nicht vom Benutzer behoben werden.

`Illegal memory block used.`
Kann nicht vom Benutzer behoben werden.

`Illegal primitive called.`
Kann nicht vom Benutzer behoben werden.

`Parser jammed.`
Kann nicht vom Benutzer behoben werden.

`Sequencing error in memory manager.`
Kann nicht vom Benutzer behoben werden.

`System initialization failure (INIT.WAM may be corrupted).`
Dieser Fehler kann durch Manipulation an der Datei `INIT.WAM` entstehen. Diese Datei enthält die Informationen, die der nicht in Prolog geschriebene Teil von Prolog-68 benötigt und sollte **auf keinen Fall** verändert werden.

A.2 Fatale Fehler

Wie bereits erwähnt, führen die hier besprochenen Fehler zum Abbruch des Prolog-Systems. Zumindest einige davon werden aber in zukünftigen Programmversionen intelligenter behandelt werden (geplant sind u.a. ein zweiter *garbage collector* und ein *stack shifter*, die Speicherüberläufe so weit wie möglich vermeiden sollen).

`Can't realize this memory configuration.`
Der Benutzer hat ungültige Vorgaben für die Speicheraufteilung gemacht. Angaben zur Aufteilung des Hauptspeichers finden sich in Abschnitt 3.1.2 auf Seite 45.

`Code space overflow.`
Der *code space* ist ein Speicherbereich, in dem Atome, Funktoren, Klauseln und andere Datenstrukturen gespeichert werden. Ein Überlauf tritt ein, wenn zu viele Klauseln oder Terme in die Datenbank aufgenommen werden. Da es (noch) keinen *garbage collector* für diesen Speicherbereich gibt, führt ein Überlauf zum sofortigen Programmabbruch.

`Deep recursion (machine stack overflow).`
Überlauf des Maschinenstacks. Dieser Fehler kann z.B. durch die Ausgabe einer zyklischen Struktur verursacht werden.

`Garbage collection failure.`
Ein Überlauf des *global stack*, der alle während der Ausführung eines Prolog-Programms entstehenden Datenstrukturen aufnimmt, konnte nicht korrigiert werden, weil der *garbage collector* nicht genügend Speicher freimachen konnte.

`Global stack overflow.`
Eine vom Prolog-Programm angelegte Datenstruktur ist so groß, daß ein Überlauf des globalen Stacks auftrat, bevor der *garbage collector* aufgerufen werden konnte.

`Local stack overflow.`
Dieser Fehler wird durch eine Endlosrekursion im Prolog-Programm verursacht.

`Unification failure (possibly caused by cyclic structure).`
Zwei Terme sind zu groß, als daß sie unifiziert werden könnten. Ein solcher Fehler weist auf das Auftreten von zyklischen Strukturen hin.

`Usage: wam [options] [...]`
Der Benutzer hat das Programm mit unzulässigen Argumenten aufgerufen (siehe Abschnitt 3.1.1 auf Seite 45).

A.3 Syntaxfehler

Beim Auftreten eines Syntaxfehlers wird zunächst eine der folgenden Meldungen ausgegeben:

`Integer overflow.`
Der zulässige Wertebereich für Ganzzahlen wurde überschritten. Prolog-68 kann Zahlen von $-2^{26} = -67108864$ bis $2^{26} - 1 = 67108863$ verarbeiten.

`Internal memory overflow.`
Ein eingelesener Term ist zu groß für den Stack des Parsers (das ist eigentlich kein *Syntaxfehler*, wird aber genauso behandelt).

`Syntax error:`
Allgemeiner Syntaxfehler; diese Meldung wird gefolgt von einer näheren Angabe der Fehlerursache:

- `',' or ')' expected.`
  Die Argumentliste eines Terms ist fehlerhaft.
- '*token*' `has priority >= 1000, used in context with priority < 1000.`
  Ein Operator hoher Priorität wurde in der Argumentliste eines Terms, in einer Liste oder als Argument von `','/2` verwendet; in einem solchen Kontext sind aber nur Operatoren mit maximaler Priorität 999 erlaubt. Abhilfe: den zugehörigen Unterterm in Klammern einschließen.
- '*token*' `has priority >= 1100, used in context with priority < 1100.`
  Ein Operator hoher Priorität wurde als Argument von `'|'/2` verwendet; da dies ein Synonym für den Operator `';'/2` ist, sind in diesem Kontext aber nur Operatoren mit maximaler Priorität 1099 erlaubt. Abhilfe: den zugehörigen Unterterm in Klammern einschließen.
- '*token*' `is not a postfix operator, but used as such.`
- '*token*' `is not a prefix operator, but used as such.`
- '*token*' `is not an infix operator, but used as such.`
- '*token*' `is not an infix or postfix operator, but used as such.`
  Diese Meldungen werden von Operatoren verursacht, wenn sie falsch verwendet werden.
- `Illegal combination of operators.`
  Zwei Operatoren mit einander widersprechenden Prioritäten verursachen einen Konflikt.
- `Infix or postfix operator expected.`
- `Infix/postfix operator or ')' expected.`

- `Infix/postfix operator or ']' expected.`
- `Infix/postfix operator or '}' expected.`
  An der entsprechenden Stelle ist ein Unterterm zu Ende; entweder muß ein Operator folgen (um einen größeren Term zu bilden), oder der nächsthöhere Teilterm muß fortgesetzt werden.
- `Term expected; '`*token*`' can't start a term.`
  Das angegebene Symbol darf nicht am Anfang eines Terms stehen.

`Too many arguments for compound term.`
Der Benutzer hat einen Term mit mehr als 255 Argumenten eingegeben. Solche Terme sind in Prolog-68 unzulässig.

`Unrecognizable token.`
Das zuletzt gelesene Zeichen konnte nicht interpretiert werden.

Nach der Meldung wird der Kontext angezeigt, in dem der Fehler auftrat. Alle bereits gelesenen Symbole werden ausgegeben, gefolgt von dem Text bis zum nächsten ⟨*Ende-Punkt*⟩ (siehe Kapitel 2). Das fehlerhafte Symbol wird – falls möglich – markiert. Schließlich wird der Parser neu gestartet, um die mißlungene Leseoperation zu wiederholen.

A.4 Andere Fehler

Die meisten hier aufgezählten Meldungen werden zusammen mit einer weiteren Meldung ausgegeben, die die Fehlerursache genauer beschreibt. Bei Fehlern in eingebauten Prädikaten wird beispielsweise der fehlerhafte Aufruf mit angezeigt.

`'`*mode*`' is not a valid opening mode.` **L**
`open/3` wurde mit einem unzulässigen Öffnungsmodus aufgerufen. Die zulässigen Modi sind `read`, `write`, `append`, `read_binary`, `write_binary` und `append_binary`.

`Access to erased database reference.` **L**
Ein Datenbankverweis wurde mit `erase/1` gelöscht, aber zu einem späteren Zeitpunkt noch einmal verwendet. Im Gegensatz zu anderen Dialekten (C-Prolog) ist in Prolog-68 die Verwendung gelöschter Verweise verboten.

`Atom occurring in arithmetical expression.` **Ü**
Ein arithmetischer Ausdruck konnte nicht übersetzt werden, weil darin ein Atom auftauchte. In Prolog-68 dürfen arithmetische Ausdrücke keine Atome enthalten.

`Can't open file '`*file*`' in mode '`*mode*`'.` **L**
Die angegebene Datei konnte im angegebenen Modus nicht geöffnet werden. Mögliche Gründe:

- Die Datei sollte zum Lesen geöffnet werden, aber sie existiert nicht.
- Die Datei sollte zum Schreiben geöffnet werden, aber sie existiert und ist schreibgeschützt.
- Es wurden zu viele Dateien geöffnet (Prolog-68 kann bis zu 50 offene Dateien verwalten, aber ein Betriebssystem wie `MiNT` kann diese Zahl weiter einschränken).

`Can't open standard stream '`*stream*`'.` **L**
stream ist ein vordefinierter Datenstrom (z.B. `user` oder `midi_output`). Diese Datenströme sind ständig offen und können nicht mit `open/3` geöffnet werden.

`Clause not compiled.` **Ü**

Der Compiler konnte eine Klausel nicht übersetzen, entweder wegen eines Fehlers in der Klausel (der in einer eigenen Fehlermeldung beschrieben wurde) oder wegen eines internen Compilerfehlers (in diesem Fall würde der Autor gerne erfahren, welche Klausel den Fehler verursacht hat).

`Declaration failed.` **Ü**

Eine Direktive in einer Datei, die mit `consult/1` oder `compile/1` eingelesen wird, ist fehlgeschlagen.

`Division by zero.` **L**

Bei der Auswertung eines arithmetischen Ausdrucks wurde versucht, durch 0 zu dividieren.

`File '`*file*`' already open in mode '`*mode*`'.` **L**

Es wurde versucht, die Datei *file* zum Lesen (bzw. Schreiben) zu öffnen, während sie bereits zum Schreiben (bzw. Lesen) geöffnet war.

`Generic error in [argument` *n* `of]` *Goal* **L**

Beim Aufruf von *Goal* ist ein nicht näher klassifizierbarer Fehler aufgetreten. Falls der Fehler auf ein bestimmtes Argument zurückzuführen ist, wird dieses mit angegeben.

`Illegal left-hand side of is/2.` **Ü**

Die linke Seite eines Aufrufs von `is/2` ist weder eine Variable noch eine Zahl.

`Illegal list in arithmetical expression.` **Ü**

In einem arithmetischen Ausdruck kommt eine Liste vor, die nicht ausgewertet werden kann; zulässig sind nur Listen, die genau eine Zahl enthalten.

`Illegal stream position.` **L**

Ein Term, der keine gültige Positionsinformation darstellt, wurde als Argument eines Prädikats benutzt, das eine solche benötigt.

`Input/output error in [argument` *n* `of]` *Goal* **L**

Beim Aufruf von *Goal* ist bei einer Ein- oder Ausgabeoperation ein Fehler aufgetreten. Beispiel: Speichermedium voll.

`Invalid argument to close/1` **L**

Dem Prädikat `close/1` wurde ein Term übergeben, der weder ein offener Datenstrom noch der Name einer mit `see/1` oder `tell/1` geöffneten Datei noch der Name eines vordefinierten Datenstroms ist.

`Invalid entry in library_directory/1 table.` **L**

Die vordefinierten Prädikate `absolute_file_name/2`, `compile/1`, `consult/1` und `reconsult/1` können Dateien in einer Reihe von Bibliotheksverzeichnissen suchen. Das benutzerdefinierbare Prädikat `library_directory/1` legt fest, in welchen Verzeichnissen gesucht wird. Dieses Prädikate sollte nur eine Reihe von Fakten enthalten, die jeweils ein solches Verzeichnis definieren.

`Invalid grammar rule (illegal item in right-hand side)` **Ü,L**

`Invalid phrase (illegal item in right-hand side)` **L**

Auf der rechten Seite einer Grammatikregel bzw. im ersten Argument des Prädikats `phrase/[2,3]` kommt ein nicht übersetzbares Element (z.B. eine Zahl) vor.

`Invalid grammar rule (illegal left-hand-side terminals)` **Ü,L**

Der ⟨*Kontext*⟩ auf der linken Seite einer Grammatikregel ist keine Liste von terminalen Symbolen.

`Invalid grammar rule (illegal list of terminals)` **Ü,L**

`Invalid phrase (illegal list of terminals)` **L**
Auf der rechten Seite einer Grammatikregel bzw. im ersten Argument des Prädikats `phrase/[2,3]` kommt eine Liste vor, die nicht ausschließlich terminale Symbole enthält.

`Invalid grammar rule (rule head is a variable)` **Ü,L**
Der Kopf einer Grammatikregel ist eine Variable.

`Invalid grammar rule (rule head is not a non-terminal)` **Ü,L**
Der Kopf einer Grammatikregel ist kein nicht-terminales Symbol.

`Instantiation fault in [argument` *n* `of]` *Goal* **L**
Spezialfall eines Typfehlers: einem eingebauten Prädikat wurde eine Variable als Argument übergeben, das Prädikat braucht aber einen vollständigen Term. Das fehlerhafte Argument wird mit angegeben, falls es eindeutig bestimmbar ist.

`No code generated for assignment to void variable.` **W**
Mit `is/2` soll ein Wert an eine Variable zugewiesen werden, die in ihrer Klausel genau einmal auftaucht. Der gesamte Aufruf von `is/2` wurde ignoriert, da eine solche Zuweisung keine Auswirkungen auf den Rest der Klausel haben kann.

`Not a database reference.` **L**
Ein Term, der keinen gültigen Datenbankverweis darstellt, wurde als Argument eines Prädikats benutzt, das einen solchen Verweis benötigt.

`Not a file stream, can't change position.` **L**
Es wurde versucht, mit `stream_position/3` die Lese- bzw. Schreibposition auf einem Datenstrom zu verändern, der nicht zu einer Datei gehört. Die Änderung der Position ist nur auf Dateiströmen sinnvoll (und zulässig).

`Not a valid stream.` **L**
Ein Term, der keinen gültigen Datenstrom darstellt, wurde als Argument eines Prädikats benutzt, das einen Datenstrom benötigt.

`Not an input stream.` **L**
Ein Datenstrom, der nicht zum Lesen geöffnet ist, wurde als Argument eines Prädikats benutzt, das einen solchen Datenstrom benötigt.

`Not an internal database reference.` **L**
Ein Term, der keinen gültigen Verweis auf einen Eintrag in der internen Datenbank darstellt, wurde als Argument eines Prädikats benutzt, das einen solchen Verweis benötigt.

`Not an open stream.` **L**
Ein Datenstrom, der bereits geschlossen wurde, wurde als Argument eines Prädikats benutzt, das einen offenen Datenstrom benötigt.

`Not an output stream.` **L**
Ein Datenstrom, der nicht zum Schreiben geöffnet ist, wurde als Argument eines Prädikats benutzt, das einen solchen Datenstrom benötigt.

`Object doesn't exist in [argument` *n* `of]` *Goal* **L**
Als *Goal* aufgerufen wurde, waren nicht alle Vorbedingungen erfüllt. Beispiele: Versuch, eine nicht vorhandene Datei zu öffnen oder über das Ende einer Datei hinauszulesen.

`Out of memory error in [argument` *n* `of]` *Goal* **L**
Bei der Auswertung von *Goal* war zuwenig freier Speicher vorhanden. Die Operation konnte nicht ausgeführt werden.

`Path name too long.` **L**

Bei der Benutzung von `absolute_file_name/2` wurde versucht, einen Pfadnamen zu konstruieren, der die maximal zulässige Länge (ca. 200 Zeichen) überschreitet.

`Read past EOF on '`*file*`'.` **L**

Nach dem Erreichen des Endes der Datei *file* wurde versucht, weitere Zeichen oder Terme von dieser Datei zu lesen.

`Redefinition of predefined predicate` *Name*/*Arity* **W**

Das angegebene vordefinierte Prädikat wurde beim Übersetzen eines Programms überschrieben. In späteren Programmversionen wird dies nicht mehr erlaubt sein, so daß aus dieser Warnung eine „echte“ Fehlermeldung werden wird.

`Stream position out of range.` **L**

Beim Aufruf von `stream_position/3` wurde eine unzulässige Position angegeben.

`System error in [argument` *n* `of]` *Goal* **L**

Der Aufruf von *Goal* konnte wegen eines systembedingten Fehlers nicht durchgeführt werden. Falls möglich, wird das fehlerhafte Argument mit angegeben.

`Too many ".." 's in path name.` **L**

Bei der Benutzung von `absolute_file_name/2` wurde ein Pfadname konstruiert, der mehr Verweise auf übergeordnete Dateiverzeichnisse enthielt als wirkliche Verzeichnisse (Beispiel: `A:\..`).

`Too many open streams.` **L**

`open/3` konnte keinen neuen Datenstrom anlegen, da das interne Maximum von 50 gleichzeitig offenen Strömen überschritten wurde.

`Type mismatch in [argument` *n* `of]` *Goal* **L**

Ein Argument eines eingebauten Prädikats hat einen falschen Typ (siehe auch `Instantiation fault` ...). Das fehlerhafte Argument wird mit angegeben, falls es eindeutig bestimmbar ist.

`Unbound variable in arithmetical expression.` **Ü**

Ein arithmetischer Ausdruck konnte nicht übersetzt werden, weil darin eine Variable auftauchte, die zur Laufzeit nicht gebunden sein kann. In bestimmten Situationen kann der Compiler diesen Fehler *nicht* erkennen, so daß stattdessen ein Laufzeitfehler auftritt.

`Unknown expression.` **Ü**

Ein arithmetischer Ausdruck konnte nicht übersetzt werden, weil darin eine unbekannte Operation auftauchte.

`Value out of range in [argument` *n* `of]` *Goal* **L**

Ein Argument eines eingebauten Prädikats hat zwar den richtigen Typ (meistens eine Ganzzahl), aber der zulässige Wertebereich wurde überschritten. Falls möglich, wird das fehlerhafte Argument mit angegeben.

`Variable bound to non-integer in expression.` **L**

Zur Laufzeit wurde ein arithmetischer Ausdruck ausgewertet, in dem eine Variable vorkam, deren Wert zum Zeitpunkt der Auswertung keine Zahl war.

`Void variable in arithmetical expression.` **Ü**

Ein arithmetischer Ausdruck konnte nicht übersetzt werden, weil darin eine Variable auftauchte, die in der Klausel nur genau einmal vorkommt. Eine solche Variable kann zur Laufzeit nicht gebunden sein.

Leerseite aus satztechnischen Gründen

Literaturverzeichnis

[1] Abelson, H. und Sussman, G.J., mit Sussman, J., *Structure and Interpretation of Computer Programs*, The MIT Press, Massachusetts Institute of Technology, Cambridge (Mass.), 1985.

[2] Adams, D., *Per Anhalter durch die Galaxis*, z.B. bei Zweitausendeins.

[3] Aho, A.V., Sethi, R. und Ullman, J.D., *Compilers. Principles, Techniques, and Tools*, Addison-Wesley, Reading (Mass.), 1986.

[4] Bierce, A., *Des Teufels Wörterbuch*, Haffmans Verlag, Zürich, 1986.

[5] Badertscher, K., GEMDOS EXTENDED ARGUMENT (ARGV) SPECIFICATION, *INFO-ATARI16 Digest* **89**, Issue 595, 2. November 1989.

[6] Belli, F., *Einführung in die logische Programmierung mit Prolog*, Zweite Auflage, Bibliographisches Institut, Mannheim, 1988.

[7] Bibel, W., *Automated Theorem Proving*, Second, revised edition, Vieweg, Wiesbaden, 1987.

[8] Bratko, I., *Prolog Programmierung in der künstlichen Intelligen*, Addison-Wesley, Bonn, 1987.

[9] Clocksin, W.F. und Mellish, C.S., *Programmieren in Prolog*, Springer-Verlag, Heidelberg, 1989.

[10] Clocksin, W.F. und Mellish, C.S., *Programming in Prolog*, Third, revised and extended edition, Springer-Verlag, Heidelberg, 1987.

[11] Coelho, H. und Cotta, J., *Prolog by Example*, Springer-Verlag, Heidelberg, 1988.

[12] Colmerauer, A., Kanoui, H., Pasero, R. und Roussel, P., Un système de communication homme-machine en français, Rapport de recherche, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[13] van Emden, M.H. und Kowalski, R.A., The Semantics of Predicate Logic as A Programming Language, *Journal of the ACM* **23**, pp. 733–744, April 1979.

[14] Gabriel, J., Lindholm, T., Lusk, E.L. und Overbeek, R.A. A Tutorial on the Warren abstract machine for computational logic, Technical report ANL-84-84, Argonne National Laboratory, Argonne (Ill.), 1984.

[15] Goethe, J.W., *Faust. Der Tragödie erster Teil*, Philipp Reclam jun., Stuttgart, 1971.

[16] Hewitt, C.E., PLANNER: A language for proving therorems in robots, *Proc. of the International Joint Conference on Artificial Intelligence*, pp. 295–301, 1969.

[17] Hofstadter, D.R., *Gödel, Escher, Bach: ein Endloses Geflochtenes Band*, Klett-Cotta, Stuttgart, 1985.

[18] Johansson, A., Eriksson-Granskog, A. und Edman, A., *Prolog Versus You*, Springer-Verlag, Heidelberg, 1989.

[19] Kipling, R., *Stalky & Co*, Haffmans Verlag, Zürich, 1988.

[20] Kleine-Büning, H. und Schmitgen, S., *Prolog. Grundlagen und Anwendungen*, Zweite Auflage, B.G. Teubner, Stuttgart, 1988.

[21] Kluźniak, F. und Szpakowicz, S., mit Bień, J.S., *Prolog for Programmers*, Academic Press, London, 1985.

[22] Kowalski, R.A., Algorithm = Logic + Control, *Communications of the ACM* **22**, pp. 424–436, 1979.

[23] Kowalski, R.A., *Logic For Problem Solving*, North-Holland, Amsterdam, 1979.

[24] Kowalski, R.A., Predicate Logic as a Programming Language, Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, 1973.

[25] Lloyd, J.W., *Foundations of Logic Programming*, Second edition, Springer-Verlag, Heidelberg, 1987.

[26] Maier, D. und Warren, D.S., *Computing with Logic—Logic Programming with Prolog*, Benjamin/Cummings Publishing Company, Menlo Park (Cal.), 1988.

[27] Malpas, J., *Prolog. A Relational Language and Its Applications*, Prentice Hall, 1987.

[28] Martelli, A. und Montanari, U., An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems* **4**, pp. 258–282, April 1982.

[29] Moss, C., CUT & PASTE — defining the impure Primitives of Prolog, *Proc. of the Third International Conference on Logic Programming*, pp. 686–694, Juli 1986. (Lecture Notes in Computer Science 225, Springer-Verlag, Heidelberg)

[30] Nilsson, U. und Maluszynski, J., *Logic, Programming and Prolog*, Wiley, 1990.

[31] O'Keefe, R.A., *The Craft of Prolog*, The MIT Press, Massachusetts Institute of Technology, Cambridge (Mass.), 1990.

[32] Paterson, M.S. und Wegman, M.N., Linear Unification, *Journal of Computer and System Sciences* **16**, pp. 158–167, April 1978.

[33] Robinson, J.A., A Machine-Oriented Logic Based On The Resolution Principle, *Journal of the ACM* **12**, pp. 23–41, Januar 1965.

[34] Ross, P., *Advanced Prolog. Techniques and Examples*, Addison-Wesley, Reading (Mass.), 1989.

[35] Rowe, N. *Artificial Intelligence Through Prolog*, Prentice Hall, 1988.

[36] Saint-Dizier, P., *An Introduction to Programming in Prolog*, Springer-Verlag, Heidelberg, 1989.

[37] Schnupp, P., *Prolog. Einführung in die Programmierpraxis*, Carl Hanser Verlag, Stuttgart, 1986.

[38] Sterling, L. und Shapiro, E., *Prolog. Fortgeschrittene Programmiertechniken*, Addison-Wesley, Bonn, 1988.

[39] Sterling, L. und Shapiro, E., *The Art of Prolog. Advanced Programming Techniques*, The MIT Press, Massachusetts Institute of Technology, Cambridge (Mass.), 1986.

[40] Warren, D.H.D., An Abstract Prolog Instruction Set, Technical report, SRI International, Artificial Intelligence Center, August 1983.

[41] Warren, D.H.D., Implementing Prolog—Compiling Predicate Logic Programs, DAI Research Reports 39 and 40, University of Edinburgh, 1977.

[42] Young, R.J., *Practical Prolog*, Van Nostrand Reinhold, 1989.

Übersicht