Logisches Programmieren

Allgemeines zu Prolog:

1. Idee des „**automatischen Beweisens**“ von Fakten aus einer gegebenen Menge von Axiomen: (Eingeschränktes) **Resolutionsprinzip** als grundlegende Rechenmethode, sodass diese für Computer umsetzbar ist.
2. Ein Prolog-Programm ist dabei prinzipiell eine Sammlung von **Aussagen über logische Zusammenhänge eines Problems**.
3. Anwendung vor allem im Bereich „**künstliche Intelligenz**“ aber auch verstärkt als universelle Programmiersprache. [Nicht-numerische Mathematik, Übersetzerbau, Datenbanken, Verarbeitung natürlicher Sprache]
4. Imperative Programmiersprachen basieren auf der Idee der **von-Neumann-Architektur** [Programmieren als Manipulation von Speicherzellen]. Stattdessen basiert Prolog auf der **Prädikatenlogik 1. Stufe** [*Quantifizierung über Werte* mit Eigenschaften der 2. Stufe: *Quantifizierung über Prädikate*]. Der Unterschied zur **imperativen Programmierung**: Dort soll der Programmierer durch Angabe einer Reihe von Variablenmanipulationen als Algorithmus selbst beschreiben, wie ein Problem zu lösen ist. In Prolog soll hingegen lediglich die logische Struktur eines Problems durch den Programmierer erfasst werden, sodass Aussagen anschließend durch Regeln und Aussagen bewiesen werden können. [Prädikatenlogik beschreibt eine logische Struktur, in denen Sätze wie „X ist ein Tier“ *Prädikate* angeben, die bei auffüllen der Wahrheitswerte ergeben. Solche „Sätze“ werden dabei durch *Quantoren* verbunden.]

Prolog kennt die folgenden Kategorien von Zeichenelementen:

1. *Großbuchstaben*: A-Z
2. *Kleinbuchstaben*: a-z
3. *Ziffern*: 0-9
4. *Sonderzeichen*: Allerdings nicht alle ASCII-Sonderzeichen werden als Prolog-Sonderzeichen interpretiert wie insbesondere: % ! ( ) , ; [ ]
5. *Kommentar*: %-Zeichen, manchmal auch mehrzeilig: /*, */ - Ein Kommentar kann nicht innerhalb von Konstanten angegeben werden!

Grundsätzliche Ergebnisse in Prolog:

1. *yes*: Beweis gefunden, damit wahr.
2. *no*: Beweis nicht gefunden, nicht zwingen zu interpretieren als falsch!

Grundsätzliche Anwendung von Prolog-Systemen:

1. Bei Aufruf wird zumeist im Anfragemodus gearbeitet [?-]. Dann lässt sich:
    1. Sofern eine Datei mit Fakten und Regeln existiert durch Eingabe von „[rules.txt].“ diese Datei geladen werden.
    2. Andernfalls lässt sich durch Eingabe von „[user].“ eine Direkteingabe durchführen. Diese ist durch die Eingabe „end_of_file.“ zu beenden
2. Das System lässt sich durch Eingabe „halt.“ im Anfragemodus beenden.

Prolog kennt dabei folgende grundlegende Sprach-Elemente:

1. **Term** [einzige Datenstruktur in Prolog, auch **Objekt**, allerdings nicht zu verwechseln

mit der objektorientierten Programmierung]: Terme werden in Prolog nicht deklariert. Man bezeichnet gesondert **Grundterme** [ohne Variablen] und **Unterterme** [direkt oder indirekt in einem anderen Term enthalten] Es existieren im Detail folgende Typen von Termen:

1. **Konstante** [atomare, nicht-zusammengesetzte Terme]:
    1. **Zahl** [„123“]: Als Folge von Ziffern, oft auch Negativ- oder Gleitkommazahlen, wobei eine Wertobergrenze existiert. Allerdings ist Prolog nicht in erster Linie für arithmetische Operationen gedacht.
    2. **Atom** [„name“]: Als Folge von Zeichenelementen. Dabei muss ein Atom im Namen eines der folgenden Kriterien erfüllen:
        1. mit einem Kleinbuchstaben beginnen und dann nur Klein-, Großbuchstaben, Ziffern und Unterstriche enthalten.
        2. nur aus Sonderzeichen bestehen.
        3. Mit einem Apostroph beginnen und enden [Dazwischen escaped doppelt].
        4. Ein „Sonder“-Atom darstellen: , ; ! []
2. **Struktur**: Bestehend aus (a) einem Namen, dem **Funktor** – dieser Funktor muss ein Atom sein. Der Funktor identifiziert eine Struktur eindeutig. (b) Dazu in Klammern gefasst und durch Kommata getrennt, die **Komponenten** der Struktur. Die Zahl der Komponenten wird als **Stelligkeit** bezeichnet. Neben *normalen Strukturen* gibt es noch insbesondere:
    1. **Operatoren**: Grundsätzlich auch als normale Struktur einführbar, allerdings ist dies umständlich zu lesen. Daher ist auch die **Operatorschreibweise** zugelassen. Hier werden die Funktoren als Operatoren notiert wie unten dargestellt. Die vordefinierten, mathematischen Operatoren sind zueinander generell **linksassoziativ**. Allerdings gilt eine **Präzedenz** [niedrige Präzedenz wird bevorzugt, anders als in Mathematik, Wertbereich 1-1200], bei der Punkt- vor Strichrechung durchgeführt wird. Klammerung überlagert diese Reihenfolge. Man unterscheidet für Operatoren insbesondere:
        1. **Präfixoperator**: -2 → -(2)
        2. **Postfixoperator**: 3 fakultaet → fakultaet(3)
        3. **Infixoperator**: 2+3 → +(2,3)
    2. **Liste** [unbekannte *Länge* im Gegensatz zu allgemeinen Strukturen]: Werden in Prolog häufig zur Lösung von Problemen genutzt. Einer Liste kann ein **Listenkopf** mit **Listenrest** [Zahl von Kopf-Elementen = 1, aber auch mehr Elemente möglich] zugewiesen werden [Schreibweise: „<Listenkopf> | <Listenrest>“, „|“ kann bei mehreren Elementen auch nicht Listenkopf bezeichnen]. Eine Liste kann in verschiedenen vorgegebenen Schreibweisen realisiert werden [eigene Vorgabe möglich, aber Listen sind in Prolog effizient implementiert, daher Vorgabe empfohlen]:
        1. **Herkömmlich**: „.(e1, .(e2, []))“, wobei e die *Elemente* der Liste sind.
        2. **Leere Liste**: Durch das Atom „[]“, Synonym zur herkömmlichen Liste, da so der Term-Begriff nicht erweitert werden muss.
        3. **Listennotation**: „[e1, e2, e3]“ → „[e1 | [e2, e3]]“, → „[e1, e2 | [e3]]“ → „[e1, e2, e3 | []]“
        4. **String**: “Bla“: Automatische Umwandlung in Liste der ASCII-Werte.
3. **Variable** [„Name“]: Ein Objekt-Platzhalter wird anders als ein Objekt groß geschrieben oder beginnt mit einem Unterstrich. Darüber hinaus bestehen sie nur aus Klein-, Großbuchstaben, Ziffern und Unterstrichen. Anders als in anderen Programmiersprachen gilt die Variable allerdings als reiner Platzhalter (ähnlich der Mathematik), anstelle für einen Wert oder eine Speicheradresse.

    1. Variablen können dabei, je nach Verwendung, *implizit*:
        1. **All-quantifiziert** [Stellvertretend für *alle* möglichen Belegungen in *Klauseln*].
        2. **Existenz-quantifiziert**: [In *Anfragen* als (zu bestimmender) Platzhalter für alle *erlaubten* Belegungen].
    2. Eine Variable befindet sich dabei während der Bearbeitung in einem von zwei möglichen *Zuständen*: [*Globale Variablen* gibt es damit nie!]
        1. **Nicht-instanziert** [*ungebunden*]: Damit in Belegung irrelevant für Richtigkeit eines Beweises.
        2. **Instanziert** [*„gebunden“* ist ein wenig allgemeiner, falls eine Variable an eine Variable gebunden wird, spricht man nicht von einer Instanziierung]: Also an einen *konkreten* Term gebunden. Dieser Wert bleibt dann bis zum Ende einer Anfrage [außer beim Rückgängigmachen von Resolutionsschritten bzw. beim Ersetzen innerer Variablen] bestehen und kann nicht, wie in anderen Programmiersprachen, *zerstört* werden. [Ähnlich (nicht-Prolog) *Kostante*.] Solche Variablen bilden *Rückgabewerte* für Anfragen.
    3. **Anonyme Variable** [„_“]: Eine Platzhaltervariable, die für alles stehen kann und sich nicht paarweise gleichen muss. Dieses Konzept unterstützt in erster Linie die Lesbarkeit von Programmen und könnte substituiert werden.
2. **Klauseln** [Berechnungsvorschriften, keine Datenstrukturen!], eine (möglicherweise leere) Menge von Klauseln definiert **Prädikate / Relationen** von Objekten [z.B. weiblich(maria) bildet das Prädikat weiblich für das Objekt Maria, allgemein formulierbar als Anfrage-Funktionen, die mit ja oder nein beantwortet werden können. Die Objekte, die in diesen Funktionen vorkommen, sind **Argumente**, deren Zahl die **Stelligkeit**. Gleichnamige Klauseln mit unterschiedlicher Stelligkeit beschreiben dabei unterschiedliche Prädikate!] Klauseln bestehen allgemein aus Literalen, wobei Fakten ein grundsätzlich wahres und Klauseln ein durch andere **Literale** bedingtes Literal beschreiben.] Im Detail:
    1. **Faktum** [aussage(name, name2, ..).
]: Elementare *Tatsachen* über eines oder mehrere Prolog-Objekte bzw. *Regeln ohne Vorraussetzung* [a(e) :- true]. Die Reihenfolge der erwähnten Objekte ist dabei wichtig. Der *Name* eines Faktums muss selbst ein *Atom* sein. Die in Klammern stehenden Objekte dürfen jedem Term entsprechen. Variablen können dabei als Platzhalter für beliebige Werte gewählt werden.
    2. **Regel** [<Kopf/Konklusion,linke Seite> :- Rumpf/Prämisse/Voraussetzung/rechte Seite> → aussage(Name, ..) :- aussage2(Name2, ..), ..
]: Möglichkeit zur *Ableitung* / *Schlussfolgerung* neuen Wissens aus Tatsachen. Je nach Bezug kann eine Variable oder auch ein konkretes Objekt eingesetzt werden. Wieder ist die Reihenfolge wichtig. Zu Beachten ist, dass auf Grund der „Wenn-Dann-Logik“ je Regel nur eine Aussage bewiesen werden kann. Die rechtsstehenden, zu überprüfenden Aussagen werden allgemeiner als **Literale** bezeichnet, um hervorzuheben, dass diese nicht richtig sein müssen, der *Wahrheitsgehalt* vielmehr überprüft werden muss.
3. **Anfrage** [?- aussage(name, ..), aussage2(name2,..)]: Kann grundsätzlich in Endlosschleife resultieren.
    1. Sofern *keine Variablen* eigegeben werden, werden die Ergebnisse je nach Beweisbarkeit der Literale als „yes“ oder „no“ ausgegeben
    2. Sofern *Variablen* eingegeben werden, werden alle richtigen Aussagen als „Variable = objekt“ gelistet. Mit einem Semikolon „;“, wird die nächste Aussage

angezeigt. Sollte diese nicht existieren, endet die Anzeige mit einem „no“.

Für Prolog müssen zunächst folgende, inhaltlich äquivalente Regeln beachtet werden:

1. Grundlage für das Funktionieren logischer Programmiersprachen ist die **Abtrennungsregel** [*modus ponens*], die besagt, dass „*wenn L_1, …, L_n beweisbar sind, dann ist auch ein Literal L_0 :- L_1, …, L_n beweisbar*“. Fasst man ein Faktum als Literal mit leerer rechter Seite auf, so gibt diese *Schlußregel* dem Prolog-Programm (als Menge aus Fakten und Regeln) eine Bedeutung.
2. Aus dieser Tatsache heraus lässt sich dann das **Resolutionsprinzip** ableiten, mit dem Prolog arbeitet. [Umformulierung als Umkehrung: „*Ist L_0 :- L_1, …, L_n eine Regel, dann ist sie beweisbar wenn L_1 bis L_n beweisbar sind.*“]

=> Beide Aussagen unterscheiden sich in der **Sichtweise**: Mit Hilfe des Resolutionsprinzips lässt sich eine gegebene Aussage *systematisch mittels eines* Resolutionsbeweises beweisen. Ein solcher Beweis funktioniert wie folgt [wobei in jedem Schritt das **allgemeine Resolutionsprinzip** angewandt wird]:

1. Eine Anfrage „*A_1, … A_i, …, A_m*“ lässt sich *zurückführen* auf eine Anfrage: „*delta(A_1, …, L_1, …, L_n, …, A_m)*“ wobei *delta(L_0) und delta(A_i) identisch* sein müssen [folglich muss die Klausel L_0 auf das Anfrage-Literal A_i anwendbar sein].
2. Dabei muss die Funktion delta alle **Variablen** eines **Anfrage-Literals und der linken Seite einer passenden Klausel** mittels einer **Unifikation** in möglichst allgemeiner Form angleichen. Die Unifikation ist dabei formal gesehen eine *Funktion* [eine **Substitution**], die entsprechend ungleiche Variablen durch den passenden Term ersetzt, sofern möglich. Nicht-Variablen werden dabei niemals gelöscht. Die Funktion delta nennt man dann einen **Unifikator**. Der **allgemeinste Unifikator** ist dieser, der alle anderen Unifikatoren durch **Komposition** mit einer anderen beliebigen Substitution abbilden kann. [Dieser existiert in Prolog immer, falls zwei Terme überhaupt unifizierbar sind, *Algorithmus*: Schrittweises Möglichkeit prüfen und ersetzen von links nach recht, zukünftige Schritte berücksichtigen bisherig festgelegte Substitutionen, Strukturen werden rekursiv geprüft.] Dabei muss eigentlich auch immer ein **Vorkommenstest** durchgeführt werden, der bei der Substitution einer Variablen durch einen Term prüft, ob dieser Term die gleiche Variable enthält, was viele Systeme aber aus Effizienzgründen unterlassen.
3. Nach endlicher Wiederholung dieses Vorgangs sind alle Literale aufgelöst, womit die **letzte Anfrage keine Literale mehr enthält** und so endgültig mit „yes“ oder „no“ entschieden. Gilt dieses Ergebnis allerdings nur für bestimmte Bedingungen [falls die Anfrage selbst Variablen enthält], so wird diese **konditionale Entscheidung** für „yes“, falls eine solche überhaupt existiert, mit der entsprechenden Angabe versehen.
4. Prolog bedient sich dabei des **Backtracking**: Es werden die Literale der Anfrage von *links nach rechts* aufgelöst, dann die passenden Klauseln von *oben nach unten* in der Reihenfolge der Eingabe probiert. Sofern der Beweis in eine Sackgasse führt, wird der letzte Schritt [insbesondere *Unifikation*] rückgängig gemacht. [Anschaulich: *Baumstruktur*.] Aus der Theorie ist dabei bekannt, dass die Auswahlstrategie die **Vollständigkeit der Beweismethode** nicht beeinträchtigt. In Prolog kann eine Anfrage allerdings in einer **Endlosschleife** resultieren, da alle möglichen Klauseln sequentiell und **nicht parallel** geprüft werden. Damit ist diese Beweismethode nicht mehr *vollständig*! Damit gilt: Auch wenn Prolog-Programme mit unterschiedlicher Reihenfolge der Klauseln **logisch äquivalent** sind, führen sie zu unterschiedlichen Ergebnissen. Um Endlosschleifen zu vermeiden, sollten *Spezialfälle immer vor*

Klauseln stehen, die allgemeine Fälle behandeln.

5. Insgesamt gilt, dass Prolog bei paralleler Verarbeitung immer mit „yes“ antworten könnte, falls ein Beweis existiert. Dies ist aber nicht effizient zu implementieren. Allgemein gilt aber, da die **Prädikatenlogik erster Stufe aber nicht entscheidbar** ist, dass kann kein Beweissystem immer eindeutig ja oder nein liefern, ohne dass man die Macht der Sprache zu sehr reduziert. Letzteres ist aber nicht von Interesse, da die Sprache sonst nicht mehr vielseitig genug wäre, bspw. wäre keine Implementierung eines Interpreters für andere Programmiersprachen mehr möglich.]

=> Jedes Prolog-System kennt dabei die Klausel „=*(X,X)*“, die prüft, ob zwei Literale sich entsprechen, also unifizierbar sind. [Achtung =(3,1+2) → falsch, da verschiedene Literale!]

Prolog hat zwei Sichtweisen auf seine **Semantik**:

1. **Operationale Semantik**: Wie werden Programme *abgearbeitet*.
2. **Deklarative Semantik**: Was ist die *logische Bedeutung* des Programmes.

=> Wie oben beschrieben in Prolog nicht identisch, da sequentiell und nicht parallel vorgegangen wird.

Typische Programmiertechniken in Prolog:

1. **Programmieren mit Akkumulator**: Eine ursprüngliche Anfrage enthält eine Akkumulator-Variable [oft: „A“], die für das Ergebnis wahr sein muss. Dazu wird ein *Eingabe- und ein Ausgabespeicher* deklariert, wobei der Eingabespeicher zunächst die gesamte Eingabe enthält und der Ausgabespeicher leer ist. Dann wird *schrittweise* ein Element vom Eingabe- in den Ausgabespeicher *verschoben*. Wenn der Eingabespeicher leer ist, wird der Akkumulator durch gleichsetzen mit dem Ausgabespeicher zurückgegeben.
2. **Generate-and-test: Suchen nach allen möglichen Lösungen** [loesung(L) :- moeglicheLoesung(L), korrekteLoesung(L)]: eine erste Klausel findet eine beliebige Kombination, die formal gesehen eine Lösung sein könnte, die zweite Klausel überprüft diese dann auf Richtigkeit. Durch Backtracking wird dann jede mögliche Kombination ausprobiert und im Fall deren Richtigkeit wiedergegeben. Dieses Prinzip wird vor allem dann angewandt, falls *kein Wissen über ein systematisches Lösungsverfahren* vorhanden ist, die Lösung aber *nicht zu viele mögliche Ausprägungen* hat. [Beispiel: N-Damen-Problem, Sekt*Bier-Rätsel]
    1. **generate-and-test** Verfahren: Prüfen aller möglichen Lösungen. [ineffizient, dafür durch die explizite Aufteilung von Generieren und Testen sehr elegant]
        1. Evtl. Wertebereich der Ausprägungen definieren.
        2. Alle Möglichkeiten der Kombinationen definieren.
        3. Richtige Kombination der Ausprägungen definieren.
        4. „loesung(L) :- potentielleLoesung(L), richtigeLoesung(L)“ definieren.
    2. **Standard-Backtracking:**
        1. Prüft **passive Bedingungen**, Verzahnen von Generieren und Testen: Die potentielle Lösung wird nicht für jede vollständige Ausprägung einer Lösungs-Möglichkeit überprüft, sondern es wird für jeden möglichen Präfix einer Lösung geprüft, ob diese Teillösung evtl. bereits für sich einen Verstoß gegenüber einer korrekten Lösung darstellt.
        2. **Eleganz** stark davon abhängig, wie leicht sich bestehende Teillösungen überprüfen lassen, im N-Damen-Problem beispielsweise auf Grund der ausschließlich binären Bedingungen vergleichsweise leicht. Bei der Kartenfärbung allerdings nicht mehr gegeben.

    3. Oft **Änderung der Datenrepräsentation** nötig, komplexer zu implementieren, Änderungen am Code erfordern dann oft mehr Aufwand.
    4. Dafür deutliche **Effizienzsteigerung** für größere Lösungsräume.
3. **Forward Checking:**
    1. Prüft **aktive Bedingungen**, also nicht nur bisherige Teillösung im Nachhinein, sondern prüft auch im Voraus, ob für die verbleibenden Belegungen dazu überhaupt noch lösende Werte möglich sind.
    2. Durch den zusätzlichen Verwaltungsaufwand steigt die **Laufzeit** dieses Vorgehens für kleine Lösungsräume, was sich für große Lösungsräume aber positiv auszahlt. Dies ist insbesondere so weil:
        1. **Wiederholte Prüfung gültiger Lösungen**: Das Standard-Backtracking prüft jeweils die Gültigkeit identischer Teillösungen, die als Suffix durch die Belegung für neue Präfix-Lösungen entstehen.
        2. **Spätest Auffinden von Fehlern** und **nutzlose Generierung** von Belegungen / **zu später Rücksetzpunkt**: Führt eine Variablenbelegung dazu, dass eine erst sehr viel später belegte Variable keinen gültigen Wert mehr erhalten kann, so wird dies durch Standard-Backtracking erst entdeckt, wenn diese Variable auch „an der Reihe“ ist. In der Zwischenzeit werden „zu viele“ Variablen belegt, um dies zu bemerken und der „richtige“ Backtrackingpunkt ist entsprechend bereits durch weitere verdeckt, so dass kein effizienter Rücksprung möglich ist.
    3. Oft starker **Programmieraufwand** und Erschwerung von Modifikationen.

3. **Musterorientierte Programmierung**: Anhand der Argumente der Klauselköpfe [es ist zu jedem Zeitpunkt nur ein Klauselkopf unifizierbar, man nennt diese Argumente auch *Muster*] gibt es zu jedem Zeitpunkt *nur eine Möglichkeit* um fortzufahren, weshalb hier meistens *nur eine Lösung* gibt. [Damit handelt es sich hier gewissermaßen um das *Gegenteil* zur Suche nach allen möglichen Lösungen.] Das macht die Bearbeitung sehr effizient. Allerdings kann diese Technik in großen Programmen die *Wartbarkeit* reduzieren, da bei einer Veränderung der Anfrage-Argumente, diese Veränderung auch konsequent im Programm umgesetzt werden muss. Andernfalls können sehr subtile Fehler entstehen.
4. **Relationale Programmierung** [eher allgemeines Vorgehen]: Grundsätzlich kann jede Anfrage beliebig wählen, welche Parameter mit Grundtermen und welche mit Variablen belegte werden sollen. Damit stellt jede Funktion gleichzeitig ihre *Gegenfunktion* zur Verfügung. Daher lohnt es sich, *Prädikate möglichst allgemein zu formulieren*, sodass diese weitreichender nutzbar sind. *Es soll also immer versucht werden die logischen Zusammenhänge zwischen Argumenten zu beschreiben, nicht einen funktionalen Ablauf darzustellen.* Auch sollte überlegt werden, ob eine neue Relation nicht als Spezialfall aus bereits vorhandenen Relationen beschreibbar ist. *Es soll also immer in Relationen, nicht in Funktionen gedacht werden*.

=> Diese Verfahren sollten vor allem deshalb genutzt werden, da Übersetzer ihre Anwendung erwarten und deshalb **effizient umsetzen**. Darüber hinaus werden noch folgende Standardverfahren angewandt:

1. **Divide-and-Conquer**: Aufteilen einer Klausel in zwei weitere Klauseln, die auf „unterster Stufe“ durch ein Faktum einfach zu beantworten sind. [Zerlegen in Datenstrukturen in kleinere, wie auch in anderen Programmierkonzepten üblich.]
2. **Information hiding**: Bei musterorientierter Programmierung für noch unsichere Datenstrukturen anzuwenden. Dabei wird eine Struktur, bspw. person(Vorname,Nachname), immer durch eine Variable, hier Person, identifiziert.

An einer anderen Stelle wird dann das benötigte Attribut mit der zusätzlichen „Abfrage-Klausel“ wie z.B. vorname(Person, Vorname) „geladen“, immer wenn die Variable Vorname benötigt wird, wobei ein zusätzliches Faktum „vorname(p(Vorname,Nachname),Vorname)“ bzw. „nachname(p(Vorname, Nachname),Nachname)“ definiert sein muss, um solche Abfragen zu ermöglichen.

Auch da ein Prologbeweis nicht zwingend das richtige Ergebnis ergibt [z.B. wegen Endlosschleifen], folgt es **nicht ausschließlich den Vorgaben der Prädikatenlogik** [*Abweichungen* und *Auslassungen*]. Der Grund ist die Konzeption von Prolog für aktuelle Rechnerarchitekturen auf denen das parallele Auswerten noch nicht effizient umzusetzen ist. Um die praktische Programmierbarkeit in Prolog zu gewährleisten, sind insgesamt folgende nicht-logische Konzepte in Prolog implementiert:

1. **Prädikatenlogik**: Nicht-paralleles Auswerten der Anfragen kann in Endlosschleifen resultieren und so eine richtige Lösung übersehen, die rechts von dieser Endlosschleife steht [wie bereits angesprochen].
2. **Beweisstrategien**: Für Effizienzgewinne in der Auswertung.
3. **Beschneidung des Suchraums / Cut** [!]: Durch Cuts wird der zur Lösung eines Beweises existierende Suchbaum an der Stelle, an der das Ausrufezeichen steht abgeschnitten. Die bisherige Bindung von Variablen bzw. die bisherige Auswahl von Prädikaten wird dadurch fixiert.
    1. **Wirkung im Detail:** Der Cut bezieht sich nicht auf den gesamten Suchbaum sondern nur auf das aktuelle Prädikat! Ein Cut verhindert, dass für die aktuelle Klausel [dazu auch gehörend die linke Seite], die vor dem Operator stehen Alternativen versucht werden. Der Cut gilt also insbesondere *nicht transitiv* für Schachtelungen von Klauseln.
    2. **Technisch** wird dies durch das Löschen der entsprechenden auf dem *Stack* [Kellerspeicher] abgelegten *Backtrackingpunkte* realisiert. [Es gibt auf Grund der Baumstruktur zu jedem Zeitpunkt im Allgemeinen genau einen solchen Punkt mit der Ausgangslage als Wurzel.]
    3. **Beweisbarkeit**: Der Cut ! Ist selbst trivialerweise immer einmal beweisbar.
    4. **Abschneiden richtiger Lösungen**: Durch die Anwendung können allerdings auch richtige Lösungen abgeschnitten werden, weshalb sich durch Verwendung von Cut die Lücke zwischen der logischen Bedeutung eines Programms und dessen Ergebnis im schlimmsten Fall vergrößert. [Sowohl die Reihenfolge der Literale einer Klausel sowie die der Klausel selbst spielt für den Einsatz von Cut evtl. eine Rolle.]
    5. **Fehler in allgemeinen Anfragen**: Die Verwendung von Cut ist insbesondere dann problematisch, falls durch Variablen in der Anfrage alle möglichen Lösungen ausgegeben werden sollen. Mit Cut wird deshalb insbesondere dann gearbeitet, falls nur *eine mögliche Lösung* existiert. [So kann ein allgemeines Literal als Alternative für zuvor stehende, spezielle genutzt werden. Aber auch hierdurch weicht man von der logischen Bedeutung des Programms ab. Siehe auch Beispiel mit Vokalen.]
    6. Ein Cut wird in Prolog im Allgemeinen aus den folgenden *Gründen* eingesetzt:
        1. **Effizienzsteigerung** [lein Unterschied im Ergebnis]: Ein alternativer Lösungszweig kann evtl. keine Lösung mehr enthalten, wenn ein Prädikat bereits auf bestimmte Weise erfolgreich gebunden wurde. Bspw. männlich, weiblich Prädikate: Wurde weiblich für eine Eingabe erfolgreich gebunden, sind Prädikate, die männlich erfordern immer falsch und können abgeschnitten werden. Auch kann so ein *Überlaufen* des Stacks verhindert

werden.
    2. **Beschreibung funktionaler Abhängigkeiten von Argumenten** [Ergebnis durch Cut beeinflusst]: Der Programmierer kann so mitteilen, dass ein Prädikat keine andere Lösung hat. Bspw. mit Vokalen im Skript: Es muss kein Prädikat für jeden Konsonanten formuliert werden sondern lediglich ein Cut nach Vokalen. [Ersatz für if-else: *p :- q, !, r | p :- f => „p :- if q then r else f“* | if Vokal else Konsonant, vorheriges Beispiel auch so interpretierbar!]
    3. **Verhinderung von Laufzeitfehlern** bei bestimmten vordefinierten Prädikaten, bspw. bei arithmetischen Prädikaten [leichter *stack overflow*].
4. **Negation**: [\+ X, \+(X), dabei ist X ein Literal]: Keine Negation im Sinne der Logik. Das Prädikat spiegelt nicht wieder, ob X falsch ist, sondern, ob der Versuch X zu beweisen gescheiter ist.
    1. Das **realisiert** Prolog, indem zuerst versucht wird X zu beweisen und dann entsprechend das Gegenteil zurück geben wird.
    2. Dabei werden **Variablen in X aber in keinem Fall für spätere Schritte gebunden**! [Hierin ist die Differenz des Prädikats und der logischen Bedeutung begründet. Fortgeschrittene Ansätze teilen einen Beweis deshalb so weit in Teilbeweise in Form von *Koroutinen* auf, dass zu negierende Ausdrücke zunächst zurückgestellt werden, bis darin enthaltene Variablen gebunden sind. Dies erfolgt aber zu Lasten der Effizienz.] Aus diesem Grund sollten nur Literale mit *Grundtermen* negiert werden.
    3. Unter anderem deshalb funktioniert die **doppelte Negation** nicht sondern liefert im Gegenteil für bereits ein (einzelnes) beweisbares X, dass \+ \+ X für alle X beweisbar ist: \+\+ yes => \+ no => yes.
    4. Auf Grund dessen spielt die **Reihenfolge der Literale** auch wieder eine entscheidende Rolle, da diese darüber entscheidet, ob Variablen zuvor bereits gebunden sind.
    5. **Endlosschleifen** resultieren weiterhin in solchen, da zunächst X ohne Betrachtung der Negation zu beweisen versucht werden.
    6. Alternativ können negative Aussagen auch durch das nullstellige Prädikat **fail** ausgedrückt werden. Dieses ist als allgemein falsch unterstellt und wird häufig in Verbindung mit einem Cut als Möglichkeit zur Negation eingesetzt.
5. **Zyklische Terme**: Auf Grund des häufig nicht durchgeführten *Vorkommenstest* bei der Unifikation zweier Terme, sind insbesondere zyklische Terme problematisch. Hierin würde ein Beweisverfahren dazu führen, dass ein Prolog-System in eine Endlosschleife gerät, anstatt im Sinne der Logik die Unifikation abzubrechen.
6. **Arithmetische Ausdrücke**: Diese ließen sich grundsätzlich zwar durch logische Ausdrücke beschreiben, allerdings sind sie in Prolog auf Grund ihrer häufigen Anwendung und daraus resultierenden Effizienzgründen *nicht-logisch vordefiniert*.
    1. Ein **arithmetischer Ausdruck** ist ein Term, der nur aus *Zahlen*, die *Infixoperatoren* +,-,*,/,mod, die *Präfixoperatoren* +,- und *Variablen* enthält. Sind sämtliche Variablen an arithmetische Ausdrücke gebunden, spricht man von einem *vollständig instanzierten arithmetischen Ausdruck*.
    2. Der **vordefinierter Infixoperator X is Y** rechnet dabei Y aus und versucht das Ergebnis mit X zu unifizieren. [2*3 is 2*3 => 2*3 is 5, ist nicht unifizierbar, „is“ nicht zu verwechseln mit „gleich“!]; is ist deshalb rein logisch nicht beschreibbar, denn es
        1. ist ein **partielles Prädikat** [andere Ausdrücke als arithmetische, bzw. sogar vollständig instanzierte arithmetische Ausdrücke auf der rechten Seite führen zu einem Laufzeitfehler],

    2. in dem auch die **Reihenfolge** der Argumente eine Rolle spielt.
  3. *Weitere partielle Prädikate* [ab Prolog 6] sind: X =:= X, Wertgleichheit, X =\= Y, Wertungleichheit, X <,=<,>,>= Y; dazu Gleitkommaarithmetik. Alle Prädikate benötigen zwei vollständig instanzierte arithmetische Ausdrücke, die intern zunächst mittels „is“ berechnet werden.
7. **Ein- und Ausgabe**: Bisher waren Ein- und Ausgaben nur mittels Anfragen realisiert. Darüber hinaus existieren für die Kommunikation des Programmes mit dem Nutzer weitere, vorgegebene Prädikate. Diese haben neben ihrer *logischen Bedeutung* immer auch einen *Seiteneffekt*. Charakteristisch für einen Seiteneffekt ist, dass der Schritt *beim Backtracking nicht rückgängig* gemacht wird.
  1. *read(<X>)* → Wartet auf eine Nutzereingabe, versucht anschließend die Eingabe mit X zu unifizieren. Bei Erfolg der Unifikation ist das Literal genau einmal beweisbar. Ist X eine Variable, dann ist dieser Schritt trivial und gleicht einer Zuweisung, die allerdings auch beim Backtracking nicht rückgängig gemacht wird! [s.o.] Achtung: Die Eingabe muss natürlich der Prologsemantik folgen! [Die Eingabe hello world ist bspw. nicht gültig.]
  2. *write(<X>), writeq(<X>)* → Genau einmal beweisbar. Als Seiteneffekt wird der Inhalt X auf der Konsole ausgegeben, was auch bei Backtracking nicht rückgängig gemacht wird. [write(q) bevorzugt die Operatorschreibweise bzw. Listendarstellung; writeq behält bei Atomen die Anführungsstriche 'atom' bei, insbesondere nötig, falls Ausgabe später wieder von Prolog gelesen wird.]
  3. *Weitere vordefinierte Prädikate*:
    1. *get_char(N)* → Seiteneffekt: N wird mit dem nächsten Zeichen, dass der Nutzer eingibt unifiziert. Liefert true falls N der Eingabe entspricht. Mittels get_char(r) ließe sich z.B. r aus richtig als Eingabe prüfen.
    2. *get_code(X)* → Wie vorheriges Kommando, aber mit der Interpretation der Nutzereingabe als ASCII-Code.
    3. *skip_char(N)* → Für N als Zeichen ignoriert der Befehl so lange Nutzereingaben, bis das Zeichen eingegeben ist. (Auch als Teil einer längeren Eingabe. Für andere Eingaben liefert es false.
    4. *skip_code(N)* → Analog mit ASCII-Code.
    5. *put_char(N)* → schreibt den Buchstaben in die Konsole und liefert true zurück. Falls N kein Zeichen ist, ergibt das Literal eine Fehlermeldung.
    6. *put_code(N)* → Analog mit ASCII-Code.
    7. *nl* → Schreibt einen Zeilenvorschub auf die Konsole, ist dann immer true.
8. **Operatorendefinition** [nur Ein- und zweistellige Strukturen]: Insbesondere auch für die Darstellung von Ein- und Ausgaben interessant, da so die Darstellung auf der Konsole modifiziert werden kann. [Konsole wählt immer diese Darstellung.]
  1. *Formale Definition eines Operators* [op(P,T,A), selbst ein partielles Prädikat, deshalb nicht logisch]: Ist genau einmal beweisbar und definiert bzw. überschreibt als Seiteneffekt den dargestellten Operator. Dabei ist:
    1. *Präzedenz* [P, 1-1200]: Je geringer die Präzedenz eines Operators, desto vorrangiger wird er gegenüber anderen Operatoren zu einer Struktur gebündelt. [„Geringe Präzedenz bildet Unterstruktur hoher Präzedenz.“]
    2. *Assoziativität* [A, x: nicht assoziativ, y: Richtung der Assoziation]
    3. *Typ [A, <x>f<y>]:*
      1. Infixoperator: xfy, yfx, xfx
      2. Präfixoperator: xf, yf
      3. Postfixoperator: fx, fy
    4. Operator-Name [A]: Ein Atom, dass den Operator benennt.

2. *„Speichervorrang“ der Operatoren*: Bei bereits existierenden Operatoren werden abweichende Präzedenzen oder Assoziativitäten überschrieben. Operatoren gleicher Namen mit aber unterschiedlichen Typen können Koexistieren [Bsp: + als Infix- und als Prefixoperator]
3. *Nicht zu verwechseln mit arithmetischen Ausdrücken*: Diese gelten, unabhängig von der Schreibweise als Operator oder Struktur, nur für +,-,*,/,mod in Infix- sowie +,- in Prefixschreibweise. Operatoren dienen lediglich der Umwandlung von Strukturen in die Darstellung mit Operatoren!

9. **Dateien / Streams**: Das Dateikonzept von Prolog kennt zu jedem Zeitpunkt einen Eingabe- und einen Ausgabestream. Diese können eine Datei bzw. die Konsole [*Standardein- und -ausgabe*] sein [mittels „user“ bezeichnet, Dateien in Anführungsstrichen 'dir/file'].
   1. **Zustand**: Ein Stream kann entweder *geöffnet* oder *geschlossen* sein. Die Konsole [Standardein- und -ausgabe] ist grundsätzlich immer geöffnet. Dateien sind zu Beginn immer geschlossen.
   2. **Eingabe**: Einlesen von Inhalten.
      1. *see(D)* → Definiert D als Eingabestream. Ist D geschlossen, wird D geöffnet.
      2. *seeing(D)* → Unifiziert D mit dem derzeitigen Namen des Eingabestreams. [Ist D eine Variable, wird D an diesen Stream gebunden.]
      3. *seen* → Schließt den aktuellen Stream und öffnet die Standardeingabe.
   3. **Ausgabe**: Schreiben von Inhalten.
      1. *tell(D)* → Definiert D als Ausgabestream. Ist D geschlossen, wird die Datei falls existent überschrieben oder andernfalls angelegt. Ist D geöffnet, wird das Schreiben an der aktuellen Stelle fortgesetzt.
      2. *telling(D)* → Unifiziert D mit dem derzeitigen Namen des Ausgabestreams.
      3. *told* → Schließt den aktuellen Stream und öffnet die Standardausgabe.
   4. **Konsultieren** [als Eingabe]: Laden von Prologprogrammen.
      1. *consult(D)* → Alle Klauseln werden dem aktuellen Programm hinzugefügt.
      2. *reconsult(D)* → Alle Klauseln in D werden dem aktuellen Programm hinzugefügt. Dabei werden alle Prädikate aus D, die im bisherigen Programm auftreten gelöscht. Insbesondere für das Neueinlesen interesant.
      3. *[D1, …, -Dn]* → ruft consult(D1) bzw. reconsult(Dn) für alle Elemente der Liste auf.

10. **Programmieren mit beliebigen Termstrukturen**: Inhaltlich wieder näher an der logischen Programmierung. Dienen dazu, zu erkennen, ob ein Term eine bestimmte Form aufweist [z.B. ist eine Struktur, ist ein Atom, ist eine Variable]. Bisher wurde oft davon ausgegangen, dass eine bestimmte Struktur bereits vorliegt. Dies kann mit den folgenden Anweisungen aber explizit getestet werden.
    1. **Zersetzen und Aufbau beliebiger Terme** [T =.. L]: T steht für einen beliebigen Term und L für eine Zerlegung dieses Terms als Liste, wobei das erste Element den Funktor als Atom repräsentiert und anschließend die Elemente der etwaigen Struktur gelistet werden [Für Atome gilt demnach T =.. [T].]. Ein Aufruf mit beidseitig stehenden Variablen erzeugt eine Fehlermeldung. So können:
       1. Bestehende Terme in ihre Komponenten zerlegt werden.
       2. Neue Terme aus bestehenden aufgebaut werden.
    2. **Lesen des Prädikats** [functor(T,F,N)]: Gibt für einen Term T den Funktor F und die Stelligkeit N an [N=0 steht also für eine Konstante].
    3. **Lesen der Komponente einer Struktur** [arg(N,T,A)]: Liest für den Term T das Nte Argument A aus. [Achtung: Bei einer Liste ist der Listenrest ab dem Nten Element das Nte Argument.]

4. **Zerlegen eines Atoms in ASCII-Werte** [name(A,L)]: Gibt für ein Atom A die ASCII-Werte der einzelnen Zeichen in Liste L. [“xyz“ ist kein Atom, sondern eine Kurzschreibweise der Liste der einzelnen ASCII-Werte des Atoms xyz.]
5. **Testen auf nicht-instanzierte Variable** [var(X) bzw. nonvar(X)].
6. **Testen auf ein Atom** [atom(X)].
7. **Testen auf ein Integer** [integer(X)].
8. **Testen auf eine Konstante** [atomic(X)]: Falls das Prologsystem Gleitkommazahlen versteht, sind diese hier mit einbezogen.
9. **Vergleiche von Termen**: X =,\= Y → Unifizierbarkeit bzw. Negation von X und Y. X ==, \== Y, analog Gleichheit, X und Y sind an gleiche Terme, also inklusive gleicher Variablen, gebunden.

11. **Erweiterte Anfragen**: Eine Auflistung syntaktischer Elemente in Prolog, die eine festgelegte Bedeutung vordefiniert haben.
    1. *call(<X>) / X* [Interpretiert Daten (Terme) als Programme (Prädikate)] → Wertet den Inhalt von Call als Anfrage aus. [Terme können dabei als Funktor ','(A,B) zu Anfragen verknüpft werden. In Kombination von call und =.. können Anfragen so auch erst nach Erstellen des Programmes dynamisch eingebunden werden. So können durch call Anfragen auch auf beliebige Prädikate angewandt werden, was bereits in die *Prädikatenlogik zweiter Stufe* hineinreicht: Anwendung bspw. für eine Map, die auf jedes Element einer Liste ein Prädikat aufruft]. Steht eine Variable X in einer Anfrage ohne call(X), so ist dies ein synonymer Ausdruck.
    2. *true, fail* → Primitive Ausdrücke, die schlichtweg als nullstelliges Prädikat definiert bzw. nicht definiert sind.
    3. *repeat* → Anders als true ist repeat unendlich oft beweisbar. Als rekursive Struktur einfach ausdrückbar. Wird häufig für Prädikate mit Seiteneffekt im Sinne einer *failiure-driven-loop* benutzt: [fdl :- repeat, side_effect(X), end_condition(X), ! → Bricht durch Cut ab, sobald Endbedingung erreicht ist, aber Achtung bei im selben Ausdruck vorkommenden fails! [Rückkehr zu repeat durch Backtracking!] Ohne Seitenbedingung im Allgemeinen sinnlos, da so entweder Endlosschleife oder einmaliger Durchlauf gegeben ist.
    4. *Komma, Stichkomma* → Ausdruck von Und- bzw. Und-Oder-Beziehungen zwischen zwei Ausdrücken.
12. **Veränderung der Datenbank** [das Prolog-Programm, Interpretiert Programme als Date, mittels sogenannter „Datenbanksprädikate“; auf Grund der syntaktischen Äquivalenz von Termen und Klauseln möglich, Sammlung der Klauseln nennt sich dabei Datenbank]:
    1. **Dynamische Prädikate** [*:- dynamic name/n.*]: In machen Prolog-Systemen muss ein Prädikat, das zur Laufzeit verändert werden soll explizit deklariert werden.
    2. **Laden von Dateien** [*consult(<X>), reconsult(<X>)*]: Lädt Dateien mit Prologprogrammen an das Ende der Datenbank. [s.o.] Genau einmal beweisbar.
    3. **Laden in Anfang der Datenbank** [asserta(K)]: Dabei muss K einen gültigen Klauselausdruck enthalten. Genau einmal beweisbar. [Klammerung zur Vereinfachung möglich.]
    4. **Laden an das Ende der Datenbank** [assertz(K)]: Dabei muss K einen gültigen Klauselausdruck enthalten. Genau einmal beweisbar. [Erlaubt consult zu simulieren.]
    5. **Löschen von Klauseln** [retract(K)]: Löscht alle Prädikate, die mit K unifizierbar sind. Falls keine Unifikation stattfindet, ist das Prädikat falsch.
    6. **Löschen von Prädikaten** [absolish(X)]: Löscht ein Prädikat der Form name/n.

Genau einmal beweisbar, falls gefunden.

7. **Suchen von Klauseln** [clause(H,B)]: Unifiziert den Kopf einer Klausel H mit ihrem Rumpf [dabei haben Fakten den Rumpf true]. Die meisten Prolog-Systeme stellen darüber Hinaus das Prädikat listing/1, das alle Klausel zu einem passenden Prädikat name/n findet, und listing/0, das die gesamte Datenbank wiedergibt, zur Verfügung.

Praktische Anwendungsmöglichkeiten von Prolog:

1. **Denksportaufgaben**: Ein Rätsel kann in der Regel in Prolog beschrieben werden. Dazu ist folgendermaßen Vorzugehen:
   1. *Eigenschaften*: Zunächst sind „banale“ Fakten aus der Angabe zu extrahieren.
   2. *Mögliche Übergänge*: Dann sind alle möglichen Transitionen, die aus diesen Eigenschaften möglich sind, darzustellen.
   3. *Richtige Lösungen*: Abschließend ist zu definieren, wann eine Folge von Transitionen aus dem Ausgangszustand in den gewünschten Zustand erfolgt ist.
2. **Meta-Interpreter** [z.B. zur Realisierung von Expertensysteme]: Ein Meta-Interpreter ist ein Programm, das die Sprache interpretieren kann, in der es geschrieben ist. [Insbesondere kann es sich selbst ausführen. Im Allgemeinen reicht es aber auch, wenn es sich „ungefähr“ selbst versteht.]
   1. *Cut*: Die Verwendung des Cut-Operators ist auf Grund der nicht-lokalen Wirkung ein wenig aufwändiger für einen Meta-Interpreter zu interpretieren, wird hier deshalb nicht besprochen, ist aber grundsätzlich möglich.
   2. *Einfacher Interpreter* [prove/1]:
      1. prove(true) :-! → True ist immer wahr, kann deshalb so „stehen bleiben“.
      2. prove((A1,A2)) :- prove(A1), prove(A2) → Zwei durch Komma getrennte Literale können zerlegt werden. [Bei mehr als zwei Literalen sind die restlichen in A2 gesammelt, die rekursiv wieder zerlegt werden.]
      3. prove(A) :- clause(A,R), prove(R) → Finde für jedes Literal A eine Regel R mittels clause und werte diese aus. Für ein Faktum ist diese wahr [s.o.], andernfalls erfolgt ein entsprechender rekursiver Aufruf.
   3. *Vordefinierte Prädikate*: Soll ein predefiniertes Prädikat [z.B. read/1, write/1] mittels call/1 auf dem System bewältigt werden, so muss eine Regel zunächst prüfen, ob dieses Prädikat vom Nutzer so definiert wurde und dann bei Erfolg mittels cut die generelle Bearbeitung unterbinden. Das Prädikat kann dann durch call/1 exklusiv bearbeitet werden. [prove(A) :- functor(A,P,N), predef(P,N), !, call(A). predef als Liste von Fakten. Für Cut so nicht möglich.]
   4. *Lemma-Generierung*: Bereits positiv geprüfte Regeln (ohne Seiteneffekte) können generell als neues Faktum [am Anfang der Datenbank] abgespeichert werden. Das macht ein Programm effizienter, solange die Datenbank nicht überfüllt wird. Dies ist insbesondere bei sehr aufwändigen Prädikaten [z.B. Fibonacciberechnung, exponentielles Wachstum] oft sogar notwendig. Um zu entscheiden, wann ein Lemma generiert werden soll, kann entschieden werden:
      1. *Ab bestimmter Tiefe*: Überschreitet der rekursive Aufruf von prove/1 eine gegebene Grenze, so kann das Ergebnis als neues Faktum eingefügt werden. Diese Tiefe kann durch eine einfache Zählvariable überprüft werden.
      2. *Für bestimmte Prädikate* [Nutzereinschätzung i.d.R. Überlegen]: Realisierung einfach dadurch, dass nach abgeschlossenem Beweis [nur bei Erfolg erreicht], noch eine Prüfung erfolgt, ob das zugehörige Prädikat in einer Liste markierter Prädikate liegt. Ist dem so, wird das Faktum hinzugefügt. Dies kann einfach an ein Hilfsprädikat delegiert werden.

Neuere Entwicklungen in Prolog sind:

1. **Verbesserung der Negation** [Siehe Abschnitt zur Negation]: Negierte Ausdrücke werden zunächst aufgeschoben.
2. **Parallele logische Programmiersprachen**: Teils parallelisierte Auswertung von Literalen in Prolog oder neue Sprachen mit Konstanten zur Beschreibung paralleler Aktion.
3. **Modulkonzept**: Problem gerade bei großen Programmen, die viele Hilfsprädikate besitzen. Werden diese Prädikate am Schluss in ein großes Programm importiert, gibt es Konflikte, von denen der Programmierer aber nichts merkt, da diese Prädikate vereinigt werden. Darum bieten Prolog-Systeme oft einfache Module an, die es erlauben nur bestimmte Prädikate zu exportieren, die aber die internen Hilfsprädikate weiternutzen.
4. **Typkonzepte**: Viele Prädikate verlangen bereits einen festen Argument-Typen [is, name, append]. Das führt oft zu Programmierfehlern. Z.B. Turbo-Prolog hat deshalb ein Typkonzept ähnlich Pascal eingeführt, das für jede Variable einen festen Typen einführt. Dies geht aber zu Lasten der Flexibilität, der eigentlichen Stärke von Prolog. Daher müsste ein flexibleres, polymorphes Typkonzept gefunden werden.
5. **Interaktion mit anderen Sprachen**: Prolog eignet sich für manche Aufgaben nur eingeschränkt, insbesondere für grafische Anwendungen. Daher erlauben Prolog-Systeme oft, dass bestimmte Prädikate statt auf Literale auf externe Routinen zu verweisen. Dabei ist aber oft die Zulässigkeit der Argumente eingeschränkt [Atome, Zahlen, Variablen], was eine Vorarbeit in Prolog erfordert.

Eine noch nicht „standardisierte“ *Erweiterung* des logischen Programmierens ist das **logische Programmieren mit Constraints** [„Beschränkungen“].

1. **Constraints** sind dabei bestimmte *Sprachkonstrukte*, die *Bedingungen oder Einschränkungen* an die Bindung von Variablen ausdrücken. In Prolog werden als Constraints solche *Prädikate* [nicht alle Prädikate sind Constraints] bezeichnet, die:
   1. bei **variablenfreien Termen** als Argumente
   2. **entweder lösbar** sind oder **nach endlich vielen Schritten als nichtlösbar** geprüft werden können.
2. **CLP(X)**: Es lässt sich zeigen, dass bezogen auf die jeweilige Domäne der Einschränkungen, diese Domänen Speziallfälle eines allgemeinen Constraint-Schemas sind. Es existiert bisher trotzdem **kein allgemein akzeptierter Standard** zur Programmierung mit Constraints.
3. Die Idee der constraint-logischen Programmierung ist, den **Berechnungsbereich** über die Berechnung von Termen hinaus zu erweitern [Beispielsweise das Einführen von expliziten arithmetischen Operatoren]. Eine solche Erweiterung ist im besten Fall durch eine Erweiterung des *Unifikationsprozesses* zu realisieren. Im schlimmsten Fall würden solche Unifikationen aber *unentscheidbar* werden. Umgesetzt wurden solche Modifikationen bereich für:
   1. **Lineare arithmetische Terme**: Das Eliminationsverfahren nach Gauß bzw. das Simplex-Verfahren liefern effiziente Verfahren zur Lösung. [Nicht-lineare Arithmetik: Multiplikation zweier nicht-instanzierter Variablen, müssen verzögert werden]
   2. **Bool'sche Terme** [Auch in CLP(FD)]: Vereinfachung von bool'schen Termen, die nur true oder false enthalten können.
   3. **Terme über Finite Domains / Endliche Bereiche** [Standardfall im Kurs]: Beschränkung von Variablen über einen bestimmten, endlichen Wertebereich.

[CLP(FD), *finite domain*]

4. Um derartige Lösungen allerdings zuzulassen, muss zunächst der **Antwortbegriff** des normalen logischen Programmierens erweitert werden. Dazu sind:
   1. **Wertbereichseinschränkungen** [z.B. X in 1..5].
   2. Beliebige **weitere Constraints**, die Bedingungen an Variablen knüpfen [z.B. arithmetischer Constraint X+Y = Z].
5. Das **Lösen von CLP-Programmen** basiert weiterhin auf dem *Resolutionsprinzip* und der Unifikation, lässt sich dabei aber aus zwei Sichten betrachten:
   1. Das Lösen von Constraints ist ein **dritter Teil** des Lösungsverfahren, dass immer dann herangezogen wird, wenn Variablen mit Constraints involviert sind.
   2. Die Unifikation ist ein Teil des Constraint-Solvings, letzteres also eine **Verallgemeinerung** der Unifikation. [CLP-Berechnung: Lösen von Constraintsystemen.]
6. **Backtracking** wird dabei (auch) ausgelöst, sobald die Menge der Constraints zusammen genommen unerfüllbar wird. Was hingegen die **gelöste Form** eines Constrains ist, ist zunächt unklar und ist davon abhängig, welche Art von Constraints durch das System zugelassen sind. [Bsp: nur Werbereichseinschränkungen?]

Contraint-logische Programmierung mit **endlichen Bereichen** [CLP(FD)]:

1. Die *Berechnung* constraint-logischer Programme erfolgt mit Hinsicht:
   1. Ein **endlicher Bereich** ist eine *nicht-leere Menge von Konstanten* [Also Atome und Zahlen; B bezeichnet dabei i.d.R. die Menge aller endlichen Bereiche. Diese Menge ist *nach unten abgeschlossen*; alle Teilmengen aller Elemente in B sind also auch in B enthalten.]
   2. Ein **Herbrand-Universum**, die *Menge aller Grundterme* [Terme ohne Variablen], die mit diesen endlichen Bereichen gebildet werden können. [Alle Klauseln, die sich ausgehend von denen im Quelltext gegebenen generieren lassen.]
2. **Systeme**, die existieren:
   1. *Prolog 2*: Erster Ansatz mit einem Sprachkonstrukt, das erzwingt, dass zwei Variablen verschiedene Belegungen erhalten.
   2. *CLP(X)*: Entdeckung, dass der Unifikationsalgorithmus ein spezielle Form eines Algorithmus ist, der Gleichungen zwischen symbolischen Termen löst. In einer allgemeineren Form kann dieser dabei auch Constraints berücksichtigen.
      1. *CLP(R)*: Erlaubt arithmetische Constraints.
      2. *Prolog III*: Erlaubt Constraints in der 2-wertigen bool'schen Algebra, in linearen arithmetischen Constraints und über Strings.
      3. *CHIP / clp(FD)*: Bool'sche Algebra auch über Symbole, lineare Arithmetik über endlichen Zahlen. [Kommerziell auf CHIP basierend: *CHARME*, *ILOG*]
      4. *LOGIN / LIFE*: Kommerzielle Sprachen, die auch über sogenannte ordnungs-sortierte Feature-Strukturen [i.w.S gerichtete Graphen] funktionieren.
3. Weiterhin muss das **Variablenkonzept** verändert werden. Alle verbleibenden Sprachelemente können unverändert weiterverwendet werden. [Beide Typen von Variablen werden dabei gemeinsam weiterhin als *Variablen* bezeichnet]:
   1. Für jeden endlichen Bereich d in D, lassen sich **Domainvariablen** definieren [Durch hochgestellten Index bezeichnet]. Diese Variable ist auf die Konstanten in d beschränkt.
   2. Alle Variablen, die nicht auf einen einfachen Bereich beschränkt sind werden **einfache Variablen** genannt. [Kann auch als unbeschränkte Domainvariable interpretiert werden.]

4. Die **Unifikation von Domainvariablen** kann als Verallgemeinerung der bisherigen Methode interpretiert werden [einfache Variablen sind dabei in ihrem Wertebereich auf einen Wert reduzierte „Domainvariablen“]. Im Kurs wird dabei der Unifikationsalgorithmus erweitert, wohl aus Effizienzgründen. Dabei sind die folgenden Fälle also neu zu berücksichtigen:
   1. Eine *Domainvariable* über Bereich d kann nur mit einer *Konstante* x unifiziert werden, wenn x in d liegt. Das Ergebnis der Unifikation ist dann die Konstante.
   2. Wird eine *Domainvariable* mit einer *freien Variable* unifiziert, so nimmt die freie Variable die Domain der Domainvariable an.
   3. Werden *zwei Domainvariablen* unifiziert, so werden beide Variablen auf den Durchschnitt der beiden Domains beschränkt. Ist dieser leer, schlägt die Unifikation fehl.
5. Das **Prinzip der Resolution** lässt sich auf constraint-logische Programme übertragen. [Durch Verwendung der obigen, abgeänderten *Unifikationsregel*.] Allerdings sind noch zusätzliche *Inferenzregeln* [Arten der Schlussfolgerung] möglich. Einige spezielle Inferenzregeln sind:
   1. **Forward-Checking Inferenzregel** [*FCIR*, *Forward Checking in CLP*, Beispiel *not/2* auf Domainvariable und Grundterm]: Übertragen des bereits vorgestellten Forward-Checking Prinzips auf CLP.
      1. Ein **Literal** ist dabei dann **vorwärts-überprüfbar**: Das zugehörige Prädikat ist ein *Constraint* und das Literal empfängt genau ein Argument als Domainvariable [*forward-variable*, *Vorwärts-Variable*] und die restlichen als Grundterme.
      2. Dabei wird die **Domäne der Vowärts-Variablen** auf diejenigen Werte eingeschränkt, die es erlauben, dass das Literal beweisbar ist. Ist dies ein einzelner Wert, wird die Domänenvariable mit diesem Wert instanziert.
      3. Auf Grund der geforderten **Constraint-Eigenschaft** des zugehörigen Prädikats, kann dies durch jeweiliges „Einsetzen“ konkreter Werte in gleiches Prädikat erfolgen, was dann in endlicher Zeit zu einer Lösung führt.
      4. Der **Suchraum wird dadurch im Voraus** reduziert. Nach Beschränkung der Domäne ist das beschränkende Prädikat *implizit erfüllt*, es kann nach entsprechender Unifikation der Domänenvariable also in der folgenden Resolution ignoriert werden. **Backtracking ist nicht mehr notwendig**, es muss also kein Backtrackingpunkt angelegt werden.
      5. Als **Auswahlregel** lohnt es sich deshalb, die Forward-Checking Inferenzregel bevorzugt anzuwenden, sobald dies möglich ist.
   2. **Lookahead-Inferenzregel** [*LAIR*, *Looking Ahead in CLP*, Beispiele *min/3*, *max/3*]: Prüft wechselseitige Bedingungen für verschiedene Domänenvariablen.
      1. Ein **Literal** ist dann **vorausschauend-überprüfbar** [*lookahead-checkable*]: Das zugehörige Prädikat ist ein *Constraint* und mindestens ein Argument ist eine Domänenvariable [*Lookahead-Variablen*], die restlichen Grundterme.
      2. Dabei werden die **Domänen der Lookahead-Variablen** jeweils darauf geprüft, ob eine spezielle Belegung der Variablen bei [beliebiger, aber passender] Instanzierung der restlichen Domänenvariablen noch lösbar wäre. Ist dies nicht der Fall, wird die Domäne dieser Variablen entsprechend eingeschränkt.
      3. Da jeweils nur Grundterme „probiert“ werden, stellt die **Constraint-Eigenschaft** des Prädikats wiederum sicher, dass diese Lösung in endlicher Zeit berechnet werden kann.
      4. Werden alle Lookahead-Variablen außer maximal einer Instanziert, so ist die

Bedingung des Constraints durch die durchgeführte **Unifikation** bereits implizit erfüllt. Das Anfrage-Literal kann in der folgenden Betrachtung also vernachlässigt werden. Ist dies nicht der Fall, muss das Prädikat allerdings beibehalten werden und bei zugewonnener Information erneut überprüft werden.

6. **Spezielle Prädikate** in diesem Bereich [da *kein Standard* existiert, sind verschiedene Ausprägungen, Namen und Implementierungen geläufig, analog zu Parametertypen werden dabei die *Präfixe* +,-,? eingeführt, die anzeigen, ob eine Variable bei Aufruf mit einem nicht-instanzierten Term instanziert sein soll, uninstanziert sein soll oder beides sein kann; hier wird das Packet *library(clpfd)* in Augenschein genommen, das auch im Kurs verwendet wird.]:
   1. **Spezifikation endlicher Bereiche** [Oft nur eine Methode]:
      1. **Für ein Prädikat** p\n [domain p(d_1, …, d_n)]: Hierbei wird für den Aufruf dieses Prädikat grundsätzlich eine Domäne für das jeweilige Element gefordert. Ist ein Argument eine *Liste*, so gilt die Anforderung dabei für jedes Element der Liste. Bedingungen sind dabei:
         1. *h* → Steht für einen beliebigen Term des Herbrand-Universums [h: Herbrand]
         2. *[a_1, …, a_m] / a..b* → Steht für Domäne aus Elementen der Liste oder für die Zahlen a bis b [a < b].
      2. **Explizit für eine Variable** [*clpfd*, *domain(-X,+Lb,+Ub), domain(+Xs,+Lb, +Ub)*]: Weist einer Variable X bzw. einer Liste von Variablen Xs den Wertebereich der Zahlen Lb bis Ub zu. Alternativ [*clpdf,* in a..b]: Hierbei kann a auch inf und b sup sein, wobei dann keine Iteration über die Werte mehr möglich ist.
      3. **Zugriff auf Domänen**:
         1. *fd_min(?X,?N)* → Gibt das Minimum N aus Domäne X aus.
         2. *fd_max(?X,?N)* → Gibt das Maximum N aus Domäne X aus.
         3. *fd_size(?X,?N)* → Gibt die Zahl N der Elemente in X aus.
         4. *fd_dom(?X, ?Dom)* [*clpfd*] → Weißt X eine Domäne Dom zu.
   2. **Arithmetische Constraints**: Wie bereits dargestellt. [In *clpfd* wird den Zeichen jeweils ein # vorgestellt. Verwechslungsgefahr besteht insbesondere bei „=“, das ohne Hashtag eine gültige Unifikation repräsentiert.] Es können dabei auf beiden Seiten auch mathematische Ausdrücke bestehend aus Variablen, Zahlen und Operatoren stehen. [Achtung: Keine Übergabe von nicht-atomaren Ausdrücken als Argument eines Prädikats, da die Methode der Variable, diese dann mit dem Ausdruck und nicht der hier repräsentierten, eingeschränkten Domäne unifiziert.]
      1. *Lineare arithmetische Constraints* [<,>,<=,>=,=].
      2. *Sonstige arithmetische Constraints* [min,max].
   3. **Auswahlprädikate** [*indomain(?X)*]: Bei X als Konstante ist der Aufruf einfach Erfolgreich. Ist X eine Domänenvariable, so gibt das Prädikat nach und nach alle Elemente der Domäne als X aus [analog zu member\2, wobei hier als zweites Element noch eine explizite Liste angegeben werden muss].
      1. **First-Fail-Prinzip** [*deleteff(?X,+L,?R)*]: Durch vorrangige Auswahl derjenigen Domänenvariable mit der kleinsten Domäne [am meisten eingeschränkt] begünstigt schnellen Erfolg.
      2. **Aufteilen der Domäne** [*split(?X)*]: Anstatt direkt konkrete Werte einer Domäne auf X einzusetzen, um deren jeweilige Gültigkeit zu prüfen, kann es sich lohnen, einen Backtrackingpunkt anzulegen, der jeweils die Hälfte der

Domäne enthält. Diese Hälften können evtl. für sich effizienter geprüft werden.
    3. **Generelles Prädikat** [labeling(?Opt,?Vars)]: In *clpfd* erlaubt dieses Prädikat die Verwendung verschiedener Belegungsoptionen inklusive einer vollständigen Belegung der Variablen in der Liste Vars. Als Option kann dabei z.B. ff für First-Fail stehen.
4. **Symbolische Constraints**:
    1. *element(?I,+L,?V)* → Beziehung zwischen einer Liste L ganzer Zahlen und Element V an Index I.
    2. *all_different(+L) / all_distinct(+L)* → Generiert alle Constraints der Form X != Y für alle Domänenvariablen in Liste L. [(2) nur in *clpfd*.]
    3. *atmost(+N,+L,+V)* → Beschränkt die Domänen-Variablen in Liste L so, dass maximal N Variablen die Ausprägung V besitzen.

Programmiertechniken bei CLP:
1. **Forward-Checking**: Basiert auf Domänen für alle Variablen. Es wird dabei jeweils abwechselnd ein freier Wert für eine Variable belegt und die Domänen der restlichen Variablen dann entsprechend eingeschränkt.
2. **constrain-and-generate** [*verallgemeinertes* Forward-Checking]: Dieses Verfahren funktioniert analog zweistufig zu *generate-and-test*, mit dem Unterschied, dass zunächst alle Variablen einzuhaltende Constraints zugewiesen bekommen und erst anschließend versucht wird, Werte zuzuordnen. Da die Constraints dabei gelten, werden die Werte dabei automatisch entsprechend auf gegenseitige Kompatibilität überprüft. [Typischer Einsatz: Kombinatorische Probleme.] Typischerweise bleiben zwei weitere Möglichkeiten zur effizienten Lösung bei Zuordnung der Werte:
    1. **Auswahl der zuerst zu belegenden Variable**: Variablen, die eine geringen (verbleibende) Domäne haben, sollten im Idealfall zuerst belegt werden, da sich der Suchbaum hierdurch verkleinert [*First-Fail-Prinzip*].
    2. **Auswahl des hier zuerst zu belegenden Wertes**: Evtl. ist es effizienter gleich einen Teilbereich der möglichen Belegungen zu betrachten [split] statt einzelner Werte. Kann hierdurch gezeigt z.B. gezeigt werden, dass keiner oder alle der Werte für eine Lösung in Frage kommen, bleibt einiger Arbeitsaufwand erspart.

Funktionales Programmieren

Gründe für den zunehmenden Erfolg funktionaler Sprachen:
1. In der Regel **kürzerer Quelltext**, Programme sind **abstrakter** aber besser **versteh- und modifizierbar** [im Vergleich zu imperativen Sprachen].
2. Basieren auf **einfacheren mathematischen Modellen** und sind damit einfacher formal **analysier- und verifizierbar**.
3. **Inhärent parallel** ausführbar [insbesondere in Hinblick auf billigere parallele Hardware nützlich].

Scheme ist ein Dialekt von Lisp, wobei
1. Lisp eine der **ältesten** Programmiersprachen überhaupt ist (1965) und in der Praxis nicht mehr verwendet wird. Scheme wird benutzt, da es **statisch bindet** und einen **klaren funktionalen Kern** besitzt.

2. Lisp-Interpreter arbeiten als **read-eval-print-loop** [Lesen-Auswerten-Schreiben-Schleife], ähnlich MATLAB.
3. Scheme ist eine **symbolverarbeitende Sprache** [wesentliches Merkmal: Symbole sind *eigener Datentyp*, ähnlich Prolog]: Der Interpreter werdet dabei Ausdrücke wie folgt aus: Zahlen und Strings werden unverändert zurückgegeben. Bei Symbolen wird der gebundene Wert zurückgegeben. Applikationen werden ausgewertet.
4. **Kommentare** [„; <Text>“] werden in Scheme durch einen Semikolon für den Rest der Zeile gekennzeichnet.
5. Basiert auf dem **Lamda-Kalkül**: Ein Funktionskalkül aus der mathematischen Logik, der Basis für Syntax und Semantik von Scheme darstellt.
6. Funktionen sind ein **gleichberechtigter Datentyp**.
7. *Alle Ausdrücke* in Scheme sind zunächst **Daten**. Ausdrücke, die *zusätzlichen Kriterien* folgen sind dabei **Programme**. [Ähnlichkeit von Programm und Daten bereits aus Prolog bekannt.]
8. Der „Treibstoff“ von Scheme ist der **Funktionsaufruf mit Rückgabewert**. Die Parameter werden dabei grundsätzlich nicht verändert [Funktionsaufruf erfolgt ja auch in neuer Umgebung]. Stattdessen sind Ergebnisse immer **Kopien**. [*Vorteil*: Sicherheit, da keine Seiteneffekte; *Nachteil*: Zeit- und Speicherplatzverbrauch.]
9. Automatische **Garbage collection**: Jedes Symbol wird anhand seiner Zeiger verfolgt und so erreichte Objekte im Heap als erreichbar markiert. [Auch Listen] Anschließend werden nicht erreichbare Objekte [„Garbage“] freigegeben.

Folgende **Ausdrücke** [symbolic expression] existieren in Scheme:

1. **Atome / elementare Ausdrücke** [*elementare Datentypen,* Literale]:
   1. **Strings** [Zeichenketten, in Anführungszeichen, Escaping]
   2. **Zahlen** [in vielen Implementierungen beliebige Länge]
   3. **Symbole [Variablen]**: Namen zur Benennung von „Gegenständen“ [Atome, Funktionen] in Scheme. Diese enthalten nie *“,(,),.,;, Leerzeichen oder Zeilenumbrüche*. Dabei sind die Namen keine reinen Zahlen, Groß- / Kleinschreibung wird nicht unterschieden. [Kleinbuchstaben aber Konvention.] Man sagt auch, dass ein Symbol einen [zu definierenden] Wert *bindet*.
   4. **Bool'sche Konstanten** [#t, #f]: Jeder von #f verschiedene Wert kann als true verwendet werden, wo ein bool'scher Ausdruck gefordert ist.
2. **Klammerausdruck / Funktion / Form** [„(*Operator* <Liste von *Operanden*>)“] Dabei sind sowohl Operator als auch Operand schachtelbar. Anders als in der Mathematik sind Funktionen durch *Präfixnotation* gekennzeichnet: Schlechter Lesbar, aber syntaktisch einfacher.]
   1. **Applikation** [*einfache Form*]: Applikation ist eine sprachlich leicht missverständliche Bezeichnung für einen Funktions*aufruf*, welcher durch einen Klammerausdruck dargestellt wird und der keinen vom Standard abweichenden Auswertungsregeln folgt. Die Auswertung von Applikationen erfolgt dabei von **Innen nach Außen**, bis alle Ausdrücke „vollständig“ ausgewertet wurden. Dann wird der links stehende Ausdruck als *Operator*, der Rest als *Operanden* eingesetzt. [„*eager evaluation, innermost evaluation, applicative-order evaluation*“] Ausdrücke terminieren nicht, wenn ein Operator / Operand nicht terminiert [nur *strikte Funktionen* erlaubt].
      1. **Primitive** [+,-,*,/]: Vordefinierte Applikationen.
         1. **Prädikate** [*<,>, =, <=, >=, and, or, not*, nicht alle Applikationen]: Funktionen mit bool'schen Rückgabewert. Werden per Konvention mit „?“ am Ende benannt, wenn sie vom Nutzer definiert werden.

        1. *Vergleich*: Zweistellig, erwarten Zahlargumente.
        2. *eq?*: Testet Symbole auf Gleichheit.
        3. *not*: Negiert einen Ausdruck.
    2. *error* [kein Standard in Scheme]: Gibt Fehlermeldung bestehend aus Argumenten aus.
2. **Makro** [*spezielle Form*, auch durch Nutzer definierbar, hier aber nicht explizit behandelt, damit alles Primitive; *define-syntax*]: Klammerausdrücke, die nicht der konventionellen Auswertungsregel folgen. Makros werten insbesondere nicht alle Argumente aus.
    1. **Quotierter Ausdruck** [außerdem *Literal*, „'s“, „(quote s)“, insbesondere für Symbole interessant, da die Quotierung immer die Auswertung des *folgenden Ausdrucks* verhindert, also nicht nur für Atome interessant ist]: Für Atome, die kein Symbol sind irrelevant. Leere Liste „()“ muss quotiert werden. Für Symbole wird die Auswertung verhindert.
    2. **Prädikate** [beenden Auswertung ggf. vorzeitig]:
        1. *or*: Alle Argumente werden von links nach rechts ausgewertet, bis einer nicht #f ergibt. Der Rest wird nicht mehr ausgewertet. Ist kein positives Argument vorhanden, wird #f rückgegeben.
        2. *and*: Die Argumente werden von links nach rechts ausgewertet, das erste falsche wird dabei zurückgegeben und der Rest nicht ausgewertet. Sind alle Argumente richtig, wird #r rückgegeben.
    3. **Bedingte Ausdrücke** / **Fallunterscheidungen**: [Werten nur für die Ergebnisfindung relevante Ausdrücke aus]
        1. *cond* [(cond (<Prädikat> <Ausdrucksfolge>) … (else <Ausdruck> …),]: Die Prädikate werden der Reihe nach ausgewertet, bis eines nicht #f ergibt. Falls eine Ausdrucksfolge folgt, wird das Ergebnis des letzten Ausdrucks zurückgegeben, falls diese nicht folgt, der Wert des Prädikats. Der Rest des cond-Ausdrucks wird nicht ausgewertet. Gibt jedes Prädikat #f zurück, ist der Wert des cond-Ausdrucks undefiniert außer es folgt ein else-Ausdruck.
        2. *if* [(if <Prädikat> <Ausdruck wahr> <Ausdruck false>)]: Wobei der Ausdruck für false auch fehlen darf.
    4. **Symboldefinition** [(define <Symbol> <Wert>), wertet <Symbol> nicht aus]: *Bindet* ein Symbol an einen Wert. Soll ein anderes Symbol an ein Symbol gebunden werden, so wird dieses wiederum quotiert. Eine *Redefinition* ist jederzeit möglich.
    5. **Funktionsdefinition** [mit Symbolzuweisung, *Indiana style*: (define <Symbol> <Lambda>): abkürzend, aber syntaktisch eigentlich nicht richtig *MIT style*: (define (<Symbol> <Formale Parameter>) <Ausdrücke>), wertet <Symbol> nicht aus]: Das gebundene Symbol kann anschließend immer dann eingesetzt werden, wenn ein Operator erwartet wird.
        1. **Lamda-Ausdruck** [(lambda (<Formale Parameter>) <Ausdrücke>), wertet keinen Ausdruck aus sondern generiert ein neues *Abschlußobjekt: Funktionsobjekt mit Abschluß*]: Rückgabe ist ein in das System von Scheme *übersetzter*, *namenloser* Ausdruck. Sofern dieser nur einmal verwendet werden soll, kann ein Lambda-Ausdruck auch direkt eingesetzt werden.
            1. **Formale Parameter** [<Symbol> … <Symbol>]: Kann auch leer sein. Werden sequentiell abgearbeitet.
            2. **Rumpf der Funktion** [<Ausdruck> … <Ausdruck>]: Mindestens ein

Ausdruck. Letzter Ausdruck bestimmt *Funktionsergebnis*. Die vorherigen Ausdrücke können auch als *lokale Definitionen* [Vergleich zu anderen Sprachen] interpretiert werden.

2. **„Temporäre Funktion“** [(let ((<Symbol> <Ausdruck>) …) <Ausdrücke>), Realisiert als Funktion höherer Ordnung von Lambda]: Legt eine Funktion an, in der vorübergehend Symbole an Ausdrücke gebunden werden, um die anschließende Ausdrucksfolge dann mit diesen Bindungen auszuwerten. [Generiert damit einen eigenen Sichtbarkeits-Block.] Die Symbole und werden dabei nicht ausgewertet! Wird insbesondere auch dann eingesetzt, wenn teuer zu berechnende Zwischenergebnisse zwischenzulagern sind.

Beim **Aufruf einer Funktion** [vorkommen eines Klammerausdrucks außer *define*] wird jeweils eine neue **Umgebung** erzeugt. [*Sichtbarkeitsbereich, Block*] Eine solche Umgebung enthält:

1. Einen neuen **Rahmen** [Tabelle mit *Symbolbindungen*]. Enthalten sind:
   1. **Gebundene Variablen** [*Parameterbindung*]: Bindung der formalen Parameter an übergebene aktuelle Parameter.
   2. **Freie Variablen**: Bindung von Symbolen im Rumpf der Funktion mittels *define*. Solche Bindungen werden immer in den aktuell geltenden Rahmen eingetragen. [Achtung: Geschachteltes *define* bildet neue Umgebungen!]
2. Einen *Zeiger* auf die **Vorgänger-Umgebung** [*zeitlicher Vorgänger*] in der dieser Aufruf stattfindet. Auf oberster Ebene steht dabei die **globale Umgebung** [*Interpreter-Eingabeebene*].

Wichtig ist, dass Scheme [andere Lisp-Dialekte verhalten sich anders] **statisch bindet** [*lexikalische Bindung*].

1. Somit ist es nicht möglich, Definitionen zuvor undefinierter Symbole später „nachzutragen“ oder abzuändern. [Konkret: Symbole werden beim Binden an andere Symbole direkt an den repräsentierten, *aktuellen* Wert gebunden und nicht an das Symbol. [Werden Symbole im aktuellen Block redefiniert, die auch in umfassenden Blöcken gelten, so wird eine Fehlermeldung erzeugt, falls zuvor auf die äußeren Werte zugegriffen wurde. Dies geschieht um Schreibfehler zu verhindern.]
2. Dies gilt nicht für Rümpfe von Funktionsdefinitionen, da hier eine Auflösung erst stattfindet, wenn die Funktion auch aufgerufen wird. [Somit ist auch *verschränkte Rekursion* möglich.]

Beim **Definieren einer Funktion** [Lambda-Ausdruck] enthält ein Funktionsobjekt immer auch einen Verweis auf die Umgebung, in der es definiert wurde. Diese Kombination aus Lambda-Ausdruck e und Abschluß U wird als **Abschlussobjekt** [*closure*] bezeichnet.

Für die **Auswertung eines Ausdrucks** in Umgebung U gilt zusammenfassen:

1. **Konstanten**: Alle Literale werden als die zugehörige Konstante ausgewertet.
2. **Symbole** [e] werden entlang der verketteten Umgebungen beim ersten Auftreten einer Bindung als Wert dieser Bindung ausgegeben. Gibt es keinen Rahmen, der eine Bindung für e enthält, so endet die Auswertung mit einem Fehler. Eine *tautologische Definition* [s = s] ist ebenfalls möglich.
3. **Sequenz** [e = e_1 … e_n)]: Auswertung erfolgt von links nach rechts, das Ergebnis von e_n ist auch Ergebnis von e.

4. **Applikation** [e = (e_0 … e_n)]: Zunächst werden in beliebiger Reihenfolge Operator e_0 und, falls existent, die Operanden e_1 bis e_n ausgewertet. Dann wird e_0 als Funktion der restlichen Ausdrücke ausgewertet. Diese Auswertung findet in einer neuen Umgebung U" statt, die als Vorgängerumgebung auf den Abschluss U' der Funktion e_0 verweist. Diese Funktion wird dann mit e_1, … e_n als aktuelle Parameter ausgewertet. Ist dies nicht möglich [e_0 kein Funktionsobjekt oder falsche Parameterzahl] so wird ein Fehler ausgegeben.
5. (*Allgemeine*) **Makros**: Je nach den zugehörigen Regeln. Im speziellen:
    1. **Definition** [e = (define x e')]: Zunächst wird e' [Ausgehend von U] ausgewertet. Dann erfolgt eine Bindung [auch Überschreibung möglich] des Ergebnisses für das Symbol im *aktuellen Rahmen*.
    2. **Lambda-Ausdruck** [e = lambda (x) e']: Resultat ist ein neues Abschlussobjekt mit U als Abschluss und e als neuer Lambda-Ausdruck [Funktionsobjekt].

Prinzip der **funktionalen Abstraktion**: Für das funktionale Programmieren ist es wichtig, dass Funktionen möglichst abstrahiert geschrieben werden. Dies erfolgt auch in Form höherer Funktionen. Dazu sind Funktionen konzeptionell immer:

1. Mit festen **Aufgabenbereichen** zu erzeugen. Dabei soll einem Nutzer jeweils die Bedeutung [*Semantik*] von Funktionen bekannt sein, die Implementierung allerdings egal.
2. So ist unter anderem eine **Modularisierung** leicht möglich. [Beispiel: Newton-Methode durch Baum an Teil-Funktionen repräsentierbar, die dann autauschbar sind.]

Zwei Aspekte sind beim Schreiben mit funktionalen Programmiersprachen insbesondere unterschiedlich zu bspw. dem imperativen Programmieren:

1. Erzeugte **Rechenprozesse** bei Übersetzung von Funktionen und dort auftretende **Rekursion:** Es ist durch geeignete Code-Formulierungen ein *rekursiver Prozess* in einen **iterative Prozess** transformierbar. Dies erledigt der Kompilierer automatisch, wenn es sich um eine **endständig rekursive Funktione** handelt.
    1. **Lineare rekursive Rechenprozesse** [bspw. durch *Akkumulatoren*, wird auch zur Emulation von *Schleifen* genutzt]: Ein einfacher, „innerer“ rekursiver Aufruf sollte an das Ende verschoben werden. Sonst hat dieser einen *linear wachsenden Speicherverbrauch*. [z.B. Berechnung der Fakultät]
    2. **Baumrekursion** [*baumrekursiver Prozess*, Umwandlung bspw. durch *finite differencing*]: Mehrere Rekursive Aufrufe pro Funktionsaufruf führen im schlimmsten Fall sogar zu exponentiellem Speicherverbrauch. [z.B. Fibonacci]
2. **Funktionen höherer Ordnung**: Dadurch können *gleichartige Probleme* noch weiter abstrahiert werden und auch so Rechenvorschriften sehr allgemein wiederverwendet werden. [„Kleiner Code“ auch für vielschichtige Probleme.]
    1. Funktionen, die auch (andere) *Funktionen als Parameter* zulassen. [Beispiele: Summenbildung über Bereich einer Funktion, auch mit individueller Inkrementierung, Approximation Pi] Werden genutzt, um sonst bereits abgeschlossene Rechenvorgänge zu komplettieren.
    2. Funktionen, die *Funktionen als Ergebnis* liefern. Wichtig ist hierbei, zu verstehen, dass die Rückgegebenen Funktionen neue Umgebungen bilden, die ursprüngliche Funktion also unberührt bleibt. [Beispiel: Betragsbildung von Funktionen.] So kann eine Evolution von Rechenprozessen stattfinden. Eine typische Technik zur Entwicklung von Funktionen, die das ausnutzen ist die *sequentielle Dekomposition*. [Aufteilen einer Aufgabe auf zwei Funktionen, die

dann leicht verkettet werden können.]

Wie andere höhere Programmiersprachen, kennt auch Scheme **strukturierte Daten**: Basis aller solcher Daten sind **Zweikomponentige Objekte** [*dotted-pair*, „(cons a b)“]: Zum Erstellen beliebiger binärbaumartiger Strukturen geeignet. Durch Zeiger [auch *cons-Zellen*] im Speicher realisiert. Die leere Liste „()“ gilt als Nullelement. [Vorsicht: „.“ auch für Gleitkommazahlen in Benutzung. Verwechslungsgefahr!]

1. **(Echte) Listen** [(e_1 … e_n)]: Binärbaum mit Nicht-Listen als Elemente. Letztes Element ist dabei eine leere Liste. Eine solche Liste kann generiert werden:
    1. Indirekt mittels *cons*: Verschachteln von zweielementigen Paaren, wobei zweites Element wieder eine solche Liste ist. letztes Element wird mit leerer Liste verbunden.
    2. *list*: Abkürzende Schreibweise. Listenelemente werden einfach gereiht.
    3. *Quotierung*: Quotierte Klammer wird als Liste der enthaltenen Elemente interpretiert.
2. **dotted-list** [(e_1 … e_(n-1) . e_n)]: „Liste“ ohne leere Liste als letztes Element [damit keine Liste im eigentlichen Sinne].
3. **Bäume**:
    1. *„Einfache“ Bäume*: Sofern die inneren Knoten und die Wurzel keine Informationen enthalten sollen, können solche Bäume durch rekursive Listen angewandt werden.
        1. *depth* [kein Standard]: Berechnet Tiefe des Baumes.
        2. *fringe* [kein Standard]: Sammelt Blätter als Liste.
        3. Standardrekursion: cond mit drei Argumenten [atom?, null?, else → rekursiv].
    2. *Markierte Bäume*: Muss als dreielementiges Tupel selbst realisiert werden. Im Kurs sind folgende Funktionen in Gebrauch. Dabei erfolgt im Idealfall eine Fallunterscheidung für Nicht-Listen, um den Speicherplatzverbrauch gering zu halten.
        1. *make-tree*: Konstruiert einen Baum mit Knoten [erstes Argument] und Liste der Teilbäume [zweites Argument]. Erzeugt Element-Nachfolger-Beziehung. Für Speicherplatzschonung sollte Tupel nur erzeugt werden, wenn Nachfolger nicht die leere Liste ist. Achtung: Ist das erste Element eine Liste muss diese doppelt umklammert werden, da sonst keine Unterscheidung möglich.
        2. *subtrees*: Liefert Liste mit Unterbäumen von t [erstes Argument]. Falls optimiert, muss für Atome manuell die leere Liste ausgegeben werden.
        3. *root*: Liefert Wurzelknoten. Falls optimiert, muss für Atome lediglich die Eingabe ausgegeben werden.

Paare können wieder aufgeteilt werden mittels [Namesgebung von erstmaliger Lisp-Implementierung auf IBM 704]:

1. **car**: Gibt erstes Element des Paares wieder.
2. **cdr** [sprich: *kodder*]: Gibt zweites Element des Paares wieder.
3. Kombinationen [*c(a^n)(d^m)r*]: Wertet erst m cdr und dann n car Aufrufe aus.
4. *ith* [kein Standard]: Gibt i-tes Element wieder. Indizierung im Kurs immer ab 0.
5. *length* [kein Standard]: Gibt Länge der Liste aus.
6. *append* [kein Standard]: Fügt Element an das Ende der Liste an.
7. **apply**: Wendet die Funktion im ersten Argument auf jedes Listenelement im zweiten Argument an.

=> Für nicht-leere Listen ist dabei charakteristisch, dass cdr wieder eine [evtl. auch leere]

Liste ergibt.

Für die Untersuchung, um welchen Datentyp es sich bei einem Lisp-Objekt handelt, gibt es folgende Prädikate:

1. **null?**: Prüft, ob sein Argument eine leere Liste ist.
2. *atom?* [kein Standard]: Prüft, ob Zahl, [quotiertes] Symbol, String, bool'sche Konstante.
3. **number?**: Prüft, ob Nummer.
4. **symbol?**: Prüft, ob Symbol.
5. **string?**: Prüft, ob String.
6. **boolean?**: Prüft, ob bool'sche Konstante.

Ein Anwendungsbeispiel für markierte Bäume sind **Syntaxbäume** [Ableitungsbäume] für eine kontextfreie Sprache. Dazu muss in der funktionalen Programmierung:

1. Ein **Prädikat** [istype?] je Produktion der Grammatik definiert werden: So wird überprüft, welcher Konstruktor als nächstes angewandt werden soll.
2. Ein **Konstruktor** [make-simple] je Produktion der Grammatik definiert werden: So wird der nächste Vereinfachungs-Schritt für den Syntaxbaum festgelegt, falls ein solcher vorgesehen ist.
3. Das *eigentliche Programm* geschrieben werden, dass Prädikate und Konstruktoren entsprechend kombiniert.

=> Für *arithmetische* Syntaxbäume wird dabei die 0 aus Additionen entfernt und die 0,1 aus Multiplikationen. [Beispiel im Skript.] Dies ist ein Indiz, dass es in der funktionalen Programmierung vergleichsweise leicht ist, Programme zu schreiben, die andere Programme als Daten nehmen und interpretieren.

Durch gute **Datenabstraktion** [konkrete Implementierung der Daten wird nach Außen verborgen und kann deshalb geändert werden] kann man in Scheme leicht **generische Softwarepakete** generieren. [Generisch im Sinne: Unabhängigkeit der Operationen von den Typen der Argumente. Auch durch *Überladen* der Operationen realisiert.] Zwei Techniken erleichtern das:

1. **Datengerichtete Programmierung**: Für das Überladen eines Operators [bspw. +] kann auf Grund der Typen der Parameter entschieden welche speziellere Funktion aufgerufen werden soll. [Bspw. auch in streng typisierten Sprachen wie Pascal gebräuchlich. Dort kann aber bereits zur Kompilierung statisch gebunden werden. In Scheme steht allerdings zur Kompilierung einer Funktion nicht fest, an welchen Datentyp ein Symbol gebunden ist, weshalb erst bei Anwendung unterschieden werden kann.] Dazu werden Elemente mit *Typsymbolen* etikettiert.
   1. *Naiv*: Verzweigung mittels Hauptfunktion, die jeweilige überladene Methoden als Subroutinen enthält.
   2. Mittels *imperativer Elemente*: Neue Funktionen und Typen können bei Verzweigungsfunktion angemeldet werden.
2. **Botschaftenorientierte Programmierung** [wichtiger Bestandteil der *objektorientierten Programmierung*, die auch in vielen Lisp-Dialekten Einzug gefunden hat, allerdings Smalltalk-80 Vorreiter: In Scheme beispielsweise keine explizite Sichtbarkeit der Vererbung]: Daten haben eine *aktive Rolle* [sind *Datenobjekte*, denen eine *(strukturierte) Nachricht* gesendet werden kann]. Solche Objekte sind Funktionen, denen in jedem Fall ein Symbol enthalten ist, das die Nachricht beschreibt und eventuell weitere Objekte [ggf. 'dummy, falls irrelevant].
   1. *dispatcher-Funktion*: Nötig, falls rekursiver Aufruf stattfindet.

2. *Direkt*: Nur kosmetisch einfacher.

Ströme [streams, Datenstruktur] sind endliche oder unendliche Folgen von Daten, auf die insbesondere sequentiell zugegriffen wird. [Anders als beispielsweise Arrays mit wahlfreiem Speicherzugriff.]

1. **Idee von Strömen**: Zum Austausch von Daten zwischen *verschiedenen Modulen* einer Software, für die insbesondere im Vornherein unbekannt ist, wie lange die Ausgabe-Daten sind. [Auch für *parallele Prozesse* wichtig: Einrichtung einer parallel genutzten Pipe.] Beispielsweise *Unix*, wo jeder einfache Prozess eine Standardein- und -ausgabe für Zeichen hat. Komplexere Programme entstehen dabei durch Hintereinanderschalten von solchen Prozessen, die jeweils einen Strom entgegennehmen und als Ergebnis auch wieder einen Strom liefern. In der Regel terminiert ein Modul, sobald es der Eingabestrom endet.
2. **Implizite Rekursion**: Es soll versucht werden, Rekursion grundsätzlich zu vermeiden. Allerdings kann diese in primitiven Bausteinen als implizite Rekursion [*Fixpunkt* vorhanden, in der Regel das Enden eines Eigabestroms, kein externes Terminieren der Rekursion nötig] vorkommen. Die Stromverarbeitenden Funktionen sollen hingegen immer mit dem Ende des Stroms terminieren. Vorteil ist dabei, dass einfache algebraische Gesetze zur vollständigen Beschreibung des Systems ausreichen.
3. **Allgemeine Definition von Strömen**: Ströme sind durch eine Menge *algebraischer Eigenschaften* definiert, die allgemein durch eine Vorgabe von Funktionen repräsentierbar ist. Im Kurs ist eine einfache algebraische Struktur gegeben [Element-Datentyp E, Strom-Datentyp S, hier zunächst durch *Listen* beschrieben, aber auch *andere Repräsentationen* möglich]:
   1. the-empty-stream: S
   2. singleton-stream: E → S
   3. append-streams: S x S → S
   4. cons-stream: E x S → S [auch: append-streams (singleton E) x S]
   5. head: S → E
   6. tail: S → S
   7. empty-stream?: S → Bool
4. **Homomorphismus** [*strukturerhaltende Abbildung* / Funktion]: Durch Verwendung homomorphistischer Funktionen soll sichergestellt werden, dass die *elementweise Verarbeitung* eines Stromes funktioniert. Das Resultat der Verarbeitung eines *Eingabestroms* S soll dabei wieder ein [evtl. einelementiger] *Ausgabestrom* S' sein.
   1. **Formal** ist ein Homomorphismus also eine Funktion *h : S → S'* [dazu evtl. auch neuer Elementtypen *E'*] mit:
      1. *h(the-empty-stream) = initial*: Ein leerer Strom wird auf einen anzugebenden **Initial-Strom** [quasi ein neutrales Element in S'] abgebildet.
      2. *h(cons-stream(e,s)) = combine(e,h(s))*: Ein Strom, der aus einem Element und einem „Reststrom“ besteht, wird durch eine anzugebende Funktion combine auf das erste Element E und das Ergebnis des homogenisierten Reststroms berechnet. Daher kann die Berechnung des Homomorphismus als eine **Reduktion von rechts nach links interpretiert** werden. [Deshalb auch nur auf endliche Ströme sinnvoll anwendbar.]
   2. **Berechnung von Strömen**: Ein Homomorphismus ist allgemein als Funktion *right-reduce* definierbar, wobei als Argumente zusätzlich zum *Strom s* ein Symbol *inital* und eine Berechnungs-Funktion *combine* übergeben werden müssen. Daraus lässt sich dann aus der allgemeinen Vorgabe ein konkreter

Homomorphismus konstruieren. Welche Art von Ergebnis ein Homomorphismus liefert ist dabei egal, es zählt einzig und allein das Vorgehen bei der Berechnung. Ergebnisse von Homomorphismen:
    1. Können **einelementig** sein [Bspw. Verarbeitung von Zahlen zu *Summen*].
    2. Können wiederum ein **Strom** sein [Bspw. *Spiegelung* eines Stroms bzw. *map*: Homomorphismus, der Funktion entgegennimmt, die auf jedes Element einzeln angewandt wird, um die Ergebnisse in einen Ausgabestrom zu reihen]
    3. **Konkatenation** [S x S → S]: Für festen zweiten Strom auch als Homomorphismus auffassbar.
    4. Können **Filter** sein: Für jedes Element wird ein *Prädikat* angewendet, das überprüft, ob ein Element im Ausgabestrom noch vorkommen soll. Dazu muss zunächst ein Strom erstellt werden, der mittels map für jedes Element einen einelementigen Stream zurück gibt, falls ein Element den Filter passiert und einen leeren Stream falls nicht.
    5. **Flache Ströme**: Verschachtelte Ströme werden dabei in einen Elementstrom konvertiert.
5. **Probleme von Listenimplementierung**: Vergleichsweise ineffizient. [Beispiel ist das Berechnen des Quadrates der zweiten Primzahl eines Stromes. Dies wird realisiert, indem zunächst alle Primzahlen gefiltert werden (statt den ersten beiden) und dann das jeweilige Quadrat berechnet wird (statt nur das zweite). Dies wäre mit einem iterativen Prozess einfacher zu implementieren.]
    1. **Laufzeit**: Oftmals fließen nicht viele Elemente eines Stroms in ein Endergebnis ein, dann werden für lange Ströme *zu viele Zwischenergebnisse* berechnet, die auf späterer Stufe ohnehin gefiltert werden.
    2. **Speicherplatz**: Komplexe Programme zerfallen oft in zu viele Mini-Funktionen. Die Listen der Zwischenergebnisse sind dann aber jeweils *vollständig im Speicher*.
6. **Ströme mit verzögerter Auswertung**: Eine Applikation wertet zunächst alle Argumente vollständig aus, was aber nicht bei allen Applikationen für das Berechnen des Ergebnisses notwendig ist. Um die Auswertung zu verzögern:
    1. **Parameterlose Lambda-Ausdrücke** [*(lambda () e)*]: Statt den Wert des Ausdrucks e zu berechnen, entspricht dies mehr einem *Versprechen* dies, sobald nötig, zu tun. Durch Auswerten des Ausdrucks wird das Ergebnis dann realisiert.
    2. **Lazy evalutation: force / delay**: Mittels (force x) kann ein mittels (delay e) verzögerter Ausdruck ausgewertet werden. delay ist dabei in eigentlich allen Scheme-Dialekten durch ein Makro vorgegeben, da e ja eben nicht ausgewertet werden soll.
    3. **cons-stream**: Entsprechend kann in Scheme eine *verzögerte Listenverarbeitung* inszeniert werden. Dies geschieht durch ein Makro, in dem der Listenrest jeweils immer mittels *delay* für eine spätere Berechnung abgelegt, wird. Wenn dieser abgefragt (*tail*) wird, muss dies also mittels *force* geschehen. [Die verzögerten Listen sind dabei immer elementweise geschachtelt verzögert → Wird aber mehrmals auf das selbe Element zugegriffen, muss dieses für jeden Zugriff mit force *neuberechnet* werden. Dies ist durch objektorientierte Ansätze verhinderbar.]
7. **Unendliche Ströme**: Durch Verzögerung können nun auch unendliche Ströme konstruiert werden, da das jeweils nächste Element des Stroms nur bei Zugriff [mittels force] berechnet wird. Dies war durch die zuvorige Berechnung von „*rechts*

nach links“ nicht möglich [Endlosschleife, daher Listendarstellung unzureichend]. Geichsam können unendliche Ströme durch rekursive Funktionsberechnungen ausgegeben werden.

Zustandorientiertes Programmieren:

1. *Idee*:
    1. Bisher **eingeschränkt** möglich, indem mittels *define* Symbole überschrieben werden können, sodass sich die Funktionsweise von Programmen nachträglich verändert.
    2. Manchmal will man modellieren, dass der **Zustand der Welt** sich permanent verändert hat. [Glaubenskrieg: Mit abnehmender Intensität geführt.]
    3. Für **Effizienz** vorteilhaft: Wenn Werte vergessen oder überschrieben werden können, da nicht ständig (evtl. hinfällige) Zwischenergebnisse beibehalten werden müssen.
2. **Zuweisung** [*(set! x e), Marko*]: Bindet das Ergebnis von e an das erste gefundene Symbol x, in dem Rahmen in dem dieses Symbol bereits zuvor definiert war. Andernfalls liefert set! eine Fehlermeldung.
3. **Zyklische Strukturen** / Zuweisung in Listen [*set-cdr!, set-car!, keine Makros*]: Wurde eine Liste so definiert, dass erstes und zweites Element durch das gleiche Symbol repräsentiert wurden, so bedeutet hier eine Änderung eines Symbols auch die Änderung des anderen. So lassen sich insbesondere zyklische Strukturen erzeugen. [Befehl dazu ist: *(set-cdr! q q).*] Damit dies Sinn ergibt, muss man Scheme-Objekte dabei auch als Zeiger auf Werte verstehen. [Gleichheitsprüfung mittels *eq?*]
4. **Memo-Funktionen** [In Haskell mittels *Monaden* ein eleganterer Ansatz vorhanden]: Mittels *set!* lassen sich nun die Information, ob das Ergebnis bereits berechnet wurde sowie das Ergebnis selbst in zwei Variablen speichern, die sich anschließend ändern lassen. Solche Memo-Funktionen sind leicht automatisiert zu erstellen. [So ließen sich die Abrufe von force organisieren, sodass die mehrfache Auswertung der gleichen Berechnung verhindert wird, da nach dem ersten Aufruf der Memo-Wert zur Verfügung steht.]
5. **Objektorientierte Programmierung**: Durch das botschaftenorientierte Konzept lässt sich mittels set! nun leicht auf eine Reihe von Symbolen in tieferen Rahmen zugreifen, die mittels define nicht überschreibbar waren, da die dispatcher-Methode auf diese nur lesend, aber nicht schreibend zugreifen konnte. Diese Symbole können nun als Instanzvariablen gebraucht werden. [Jede neue „Instanz“ hat dabei ihre eigenen Variablen. Der Zugriff erfolgt dabei nur mittels Methoden, wie in gutem objektorientiertem Stil üblich.]

Weitere Aspekte:

1. **Modularisierung**: Erst im R6RS-Standard möglich, Programme in Bibliotheken zu untergliedern, die mittels import / export in einen Namensraum eingebunden werden können.
2. **Typisierung**: Anwendung von primitiven Funktionen wie die der Addition kann auf falsche Werte angewandt einen Laufzeitfehler verursachen. Gleichzeitig will man aber kein zu enges Typengerüst anlegen, um eine gewisse Flexibilität beizubehalten. Dies wird durch polymorphe Typen erreicht.
3. **Verzögerte Auswertung ohne Makros** [Auswertung in *Normalordnung*]: Sprachen wie Haskell verzögern die Auswertung von Funktionen grundsätzlich, bis tatsächlich auf das Ergebnis zugegriffen wird. Zuvor wird lediglich die Struktur gelinkt. So

werden auch nur solche Parameter ausgewertet, die wirklich benötigt werden. [Eine Auswertung wird beispielsweise durch primitive Funktionen wie die Addition angestoßen.] So sind unendliche Datenstrukturen [Ströme] und nicht-strikte Funktionen [Funktion, die auch dann ein wohl-definiertes, terminierendes Resultat haben kann, wenn die Auswertung einzelnen Parameter nicht terminiert] leicht auch ohne Makros umsetzbar, da eine Verzögerung immer bereits durch das System stattfindet. [Terminiert eine Auswertung in *applikativer Reihenfolge* (bisher), dann terminiert sie auch immer in *Normalordnung*. Die Ergebnisse sind immer gleich. Damit ist die Normalordnung die eigentlich grundsätzlich bevorzugte Berechnungsvorschrift.]

4. **Haskell** [ähnlich *Miranda*, das nicht weiterentwickelt wurde]: Modernste funktionale Sprache. Features:
    1. Polymorphes *Typenkonzept*.
    2. Auswertung in *Normalordnung*.
    3. Funktionsdefinitionen mit *Gleichungen* und *Parametermustern*.
    4. *Elementare Daten* vereinfacht.
    5. *Monaden* erlauben Funktionen mit Seiteneffekten direkt in das funktionale Programmier-Konzept einzugliedern.

Integration logischer und funktionaler Programmierung

Logisches und funktionales Programmieren sind die wichtigsten Ausprägungen der **deklarativen Programmierung**.

1. Im **Gegensatz zur imperativen Programmierung** ist diese in seiner reinen Form von Zustand von Variablen abstrahiert, das von Wertzuweisungen absieht [Ausnahmen bestehen, beispielsweise (!set …) in Scheme].
2. Die **jeweilig wesentlichen charakteristischen Eigenschaften** sind:
    1. im *logischen Programmieren*:
        1. *logische Variablen* [Als Sammelbegriff verschiedener Werte]
        2. *partielle Datenstrukturen* [Datenstrukturen unter Verwendung von Variablen]
        3. *eingebaute Suche* [zur Lösung von Anfragen]
    2. im *funktionalen Programmieren*:
        1. *verschachtelte Ausdrücke* [in Klammerausdrücken, Funktionen]
        2. *Funktionen höherer Ordnung* [Funktionen als Datentyp]
        3. *lazy evaluation* [verzögerte Auswertung von Elementen]
3. Mögliche **Vorteile der Kombination** der Paradigmen [*funktional* / *logisch*]:
    1. **Funktionen / Relationen**: Funktionen sind grundsätzlich nur spezielle Relationen, wobei erstere oft aber praktischer anzuwenden sind. Im Idealfall sollten daher beide Prinzipien anwendbar sein.
    2. **Elemente höherer Ordnung**: In funktionalen Sprachen mittels Funktionen höherer Ordnung bereits berücksichtigt. In logischen Sprachen [meta-call] nur außerhalb des Paradigmas integriert. [In jedem Fall: Eine Entscheidung, Elemente höherer Ordnung zu integrieren oder dies nicht zu tun, hätte starken Einfluss auf die Programmiertechniken des jeweiligen Paradigmas.]
    3. **Suche und Ablaufsteuerung**: Das logische Programmieren stützt sich auf ein internes, nicht-deterministisches Suchverfahren zur Ermittlung von Lösungen. Dieser Nicht-Determinismus wird erst durch ein eine Implementierung durch eine entsprechende Tiefensuche mit Backtracking aufgelöst. Sollen *Algorithmen*

umgesetzt werden, muss hingegen oft auf nicht-deterministische Programmelemente zurückgegriffen werden. Hierzu eigenen sich funktionale Sprachen oft besser.

4. **Partielle Datenstrukturen**: Logische Sprachen sehen grundsätzlich vor, dass Datenstrukturen mit Variablen integriert werden können. Die hier durchgeführte Suche, kann Aussagen im Idealfall also ohne Variablenbindungen beweisen oder widerlegen und muss keine entsprechenden Backtrackingpunkte für einzelne Variablen-Werte anlegen. In funktionalen Sprachen ist dies hingegen nicht vorgesehen. Sollen verschiedene Werte für Ein- und Ausgabe berücksichtigt werden, muss hingegen auf Listen zurückgegriffen werden.
5. **Datenfluss**: Der Datenfuss ist in funktionalen Sprachen durch formale Parameter und Rückgabewerte klar gegeben. In logischen Sprachen kann dieser jedoch in beliebiger Richtung erfolgen [sofern nur rein-logische Elemente verwendet werden].
6. **Implementierungstechniken**: Logische Sprachen sind mächtiger, aber Programme damit oft auch komplexer zu implementieren. So könnte für funktional charakterisierte Programmelemente idealerweise die effiziente Implementierungsmöglichkeit funktionaler Sprachen ausgenutzt werden.
7. **Programmierkonzepte** [Typkonzepte, Modulkonzepte, Funktionen und Logiken höherer Ordnung]: Eine Kombination logischer und funktionaler Programmierung sollte es erlauben, die Verwendung dieser Konzepte auf ihre Vorteile hin besser abzuwägen und die jeweils beste Lösung auszuwählen.

4. Die **wichtigsten Gesichtspunkte**, die logische und funktionale Sprachen unterscheiden und die Zwecks einer Zusammenführung betrachtet werden müssen:
   1. **Logische Variablen / Unifikation**: In funktionalen Sprachen nicht vorhanden.
   2. **Nicht-Determinismus**: Eng zusammenhängend mit (1), widerspricht dem Prinzip der Ablaufsteuerung in funktionalen Sprachen.
   3. Können Funktionen / Relationen **höherer Ordnung** in einer Kombination eingebracht werden?: Hier nicht diskutiert.

Konkrete Möglichkeiten, funktionale und logische Sprachen **zu integrieren** [hier ohne Rücksicht auf Elemente höhere Ordnung] sind:

1. **Kombination bestehender Sprachen** in gemeinsamer Umgebung:
   1. Gegenseitiger Aufruf von Sprachen am Beispiel von *LOGLISP*:
      1. *Prolog aus Lisp* [(ALL (X_1, …, X_r) C_1, …, C_n]: Stellt Anfrage an Prolog mit Literalen C_i und Variablen X_j. Als Ergebnis der Funktion ALL wird eine Liste der Lösungstupel zurückgegeben.
      2. *Lisp aus Prolog*: Alle LISP-Funktionen stehen automatisch als Literal in Prolog zur Verfügung.
   2. Zwei wichtige *Nachteile*:
      1. **Konvertierung von Datenstrukturen** kostet Zeit.
      2. **Komplexe Semantik**, da Sprachen nicht in eine gemeinsame Syntax überführt werden.
2. **Beschränkung des Gebrauchs logischer Variablen** und des hiermit verbundenen **Suchraums** [*Mode-Deklaration*]:
   1. Für jede Variable eines Prädikats wird angegeben, ob es sich um eine *Eingabe- oder Ausgabevariable* handelt. Bei Eingabevariablen muss es sich dann um *voll-instanzierte Werte* handeln, bei Ausgabe-Variablen um *nicht-instanzierte Variablen*, die als Resultat der „Prädikats-Funktion“ ein Ergebnis zugewiesen bekommen.

    2. Als Konsequenz dieser Einschränkung ist **keine vollständige Unifikation** mehr notwendig, sofern ein Prädikat mittels einer Mode-Deklaration beschränkt ist. Stattdessen findet eine **einseitige Unifikation** statt, auch **Pattern Matching** genannt, bei der das Binden von Variablen an Werte zwingend nur *in eine Richtung* stattfindet. [Es erfolgt also ein parameterabhängiger Aufruf analog zum *musterorientierten Programmieren* in den logischen Sprachen.]
    3. Als Vorteil zur klassischen funktionalen Programmierung sind so auch **mehrelementige Rückgabewerte** von Funktionen möglich, ohne dass diese in Form von Tupeln als eigenständiger Datentyp zusammengefasst werden müssen. [Das Mode-System muss hier zusätzlich sicherstellen, dass solche Prädikate nicht mit identischen Variablen für verschiedene Rückgabewerte aufgerufen werden.]
3. **Erweiterung funktionaler Sprachen um logische Variablen** und des hiermit verbundenen **Suchraums** [am allgemeinen Beispiel]:
    1. Im Kern lässt sich jede funktionale Sprache auf **Definitionsgleichungen** zurückführen, nach denen Umwandlungen vollzogen werden [t_1 → t_2].
    2. Die **operationale Auswertung** wird dann durch **Reduktionsschritte** vollzogen.
        1. Voraussetzung dabei ist, dass der zu reduzierende Term [quasi die „Eingabe“ in eine solche Definition] **keine Variablen** enthält.
        2. Die Auswahl der jeweils passenden Definitonsgleichung erfolgt dabei durch **Pattern Matching** der Eingabe mit der linken Seite der Definitionsgleichungen [*einseitige Unifikation*, da nur Variablen in der Definitionsgleichung gebunden werden, nie aber in der Eingabe]. Man bezeichnet diese spezielle Form von Unifikatoren dabei auch als *Matcher von s auf t* [s: Definitionsgleichung, t: Eingabe].
    3. Ein Satz von Definitionsgleichungen kann aber bestimmte *Eigenschaften* aufweisen, die bei Reduktion einer Eingabe auf seine **Normalform** [nach endlich vielen Schritten geltende „Endform“] gelten:
        1. **Konfluenz**: Die Normalform ist von *Reihenfolge und Auswahl der Reduktionsschritte* unabhängig, die jeweils durch die Definitionsgleichungen erlaubt sind.
        2. **Terminierung**: Nach *endlich vielen Reduktionsschritten* ist keine Reduktion mehr möglich.
    4. **Narrowing**: Ersetzt man nun das Pattern Matching durch einen *allgemeinen Unifikator*, lassen sich Lösungen durch das Narrowing-Verfahren lösen. So sind Variablen in Ein- und Ausgabetermen möglich.
        1. Dabei wird durch **schrittweise (allgemeine) Unifikation** versucht, zwei anzugleichende Terme auf die gleiche Form zu transformieren, wobei beide Definitions-Richtungen erlaubt sind.
        2. Allerdings entsteht hierdurch schnell ein vergleichsweise **großer Suchbaum**, der durchsucht werden muss, was sich in der Laufzeit nachteilig niederschlägt.

Eine funktional-logische Sprache **bietet** dann im Idealfall:

1. Sowohl rein logisches als auch rein funktionale Möglichkeit zum Programmieren. [Und damit auch die jeweilig spezifischen **Sprachkonstrukte**.]
2. **Effizientere Auswertungsmechanismen**. Beispiel nicht-logische Elemente:
    1. **Deterministische Berechnungen** mittels Funktionen, sodass der Einsatz des Cut-Operators zum expliziten Beschränken unnötiger Suchzweige vermieden werden kann.

    2. Keine **Simulation Funktionen höherer Ordnung** mehr nötig [in Prolog mittels *call*].
    3. **Unendliche Datenstrukturen** mittels *lazy evaluation*.
3. Für die **operationale Semantik** werden bisher zwei Möglichkeiten diskutiert [im Idealfall würde eine Kombination aus beiden Verfahren zugelassen, sodass die Vorteile beider Möglichkeiten genutzt werden können]:
    1. **Narrowing**: Vollständiges Berechnungsverfahren, dass alle Lösungen auch bei nicht instanzierten Variablen durch allgemeine Unifikation findet. Allerdings ist das Erfahren *ineffizient*.
    2. **Residuation**: Funktionsaufrufe werden verzögert, bis alle Argumente instanziert sind und deterministisch ausgewertet werden können. Allerdings ist das Verfahren durch das Warten *unvollständig*, da so nicht alle Anfragen bearbeitet werden können, falls entsprechende Variablen nicht instanziert werden.