Prolog

Grundlagen:

Prolog zählt zu den KI-Sprachen und orientiert sich an der mathematischen Logik. Es gibt Aussagen, die wahr oder falsch sind, und Aussageformen, die Variable enthalten und erst durch Ersetzen der Variablen durch konkrete Werte (**Atome**) zu wahren (**true**) oder falschen (**false**) Aussagen werden. Aussagen und Aussageformen nennt man auch **Prädikate**. Prädikate können durch mehrere Teilprädikate definiert werden, **Klauseln** genannt.

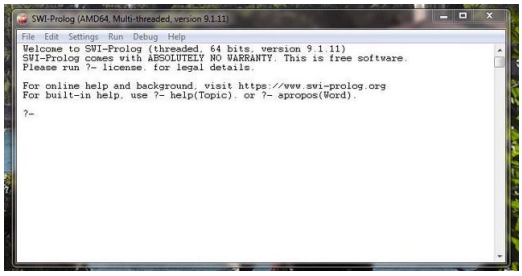
Klauseln bestehen aus dem Prädikatsnamen und gegebenenfalls aus einer in Klammern eingeschlossenen Parameterliste, zusammen dem **Kopf** der Klausel. Die Parameter können Atome, **Variable** (Platzhalter für Atome) oder _ sein, letzteres als Platzhalter für Atome ohne Wirkung.
Klauseln können weitere Prädikate aufrufen. In diesem Fall folgt auf dem Kopf nach **:-** die Liste dieser Prädikate, genannt der **Rumpf**. Nur wenn alle Prädikate des Rumpfes den Wert true haben, erhält auch die Klausel den Wert true.

Gestartet wird durch Aufruf eines Prädikats, bei dem evtl. Variable durch Atome ersetzt wurden. Der Rechner versucht dann, weitere Klauseln und Atome so zu einer Schlusskette zu kombinieren, dass das aufrufende Prädikat den Wert true erhält. Ausgegeben werden die benötigten Atome und der Wahrheitswert. Prolog ermöglicht, während des Kombinierens selbstständig neue Klauseln zu bilden, die dann zur Problemlösung genutzt werden.

Ursprünglich wurde beim Ablauf eines Prolog-Programms dieses von einem Interpreter stückweise analysiert und in Maschinensprache umgesetzt. Mittlerweile gibt es auch Compiler, die das gesamte Programm vor dem Ablauf in Maschinensprache übersetzen und so einen deutlichen Geschwindigkeitsgewinn bieten. Allerdings schränken Compiler die Struktur neuer Klauseln ein, die während des Ablaufs neu gebildet werden.

Gearbeitet werden soll hier SWI-Prolog bzw. SWISH-Prolog. SWI-Prolog muss auf dem eigenen Rechner installiert werden, während SWISH-Prolog als dessen Online-Version ohne Installation auskommt.

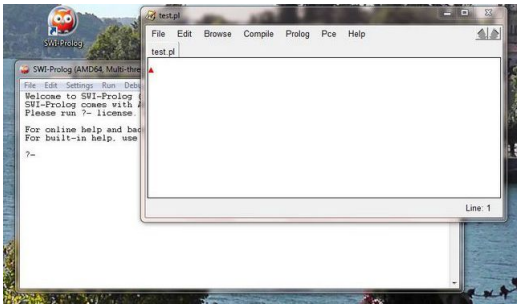
SWI-Prolog



Beim Start öffnet sich ein Fenster - das **Hauptfenster** -, das zur Organisation der Prolog-Umgebung dient, Meldungen des Compilers anzeigt und in dem man Abfragen eingeben kann.

Unter dem Menüpunkt **File** wird mit **New** angeboten, ein neues Programm zu erstellen oder ein altes mit **Edit** zu bearbeiten. **Consult** kompiliert ein fertiges Programm, dessen Name hierzu eingegeben werden muss.

Ruft man New oder Edit auf, so öffnet sich ein zweites Fenster mit einem Editor, in dem man ein Programm auf gewohnte Weise bearbeiten kann. Ein Programm kann gespeichert oder geöffnet werden. Wichtig ist unter dem Menüpunkt **Compile** die Option **Compile Buffer**,
mit der das angezeigte Programm kompiliert werden kann.

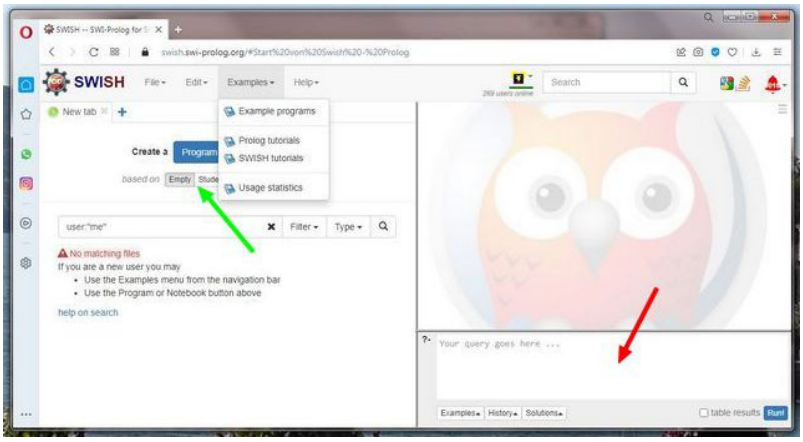


Programmfehler werden bereits beim Editieren farblich markiert, ebenso Fehler, die beim Compilieren entdeckt werden. Fehlermeldungen des Compilers werden als gesonderte Meldung angezeigt und ebenso im Hauptfenster.

Damit sind die für das Programmieren nötigen Funktionen für SWI-Prolog erklärt!

SWISH-Prolog

Die Online-Version von SWI-Prolog kann man über den Link **swish.swi-prolog.org** erhalten. Nach dem Start teilt sich der Bildschirm teilt sich in zwei Eingabe-Bereiche auf:



Rechts unten (roter Pfeil) findet man das **Hauptfenste**r, In ihm kann eine Abfrage eingeben werden, das durch den Button **RUN** am rechten unteren Fensterrand gestartet werden kann.

Links kann man den Programmtext eingeben und bearbeiten.
Dazu ist zunächst ist **Program** (grüner Pfeil) anzuklicken, damit sich das linke Teilfenster zu einem Editor wandelt.

Am linken oberen Bildschirmrand findet man ein Menü, das zusätzliche Aktionen wie Speichern und Laden von Dateien ermöglicht. Zum Aktivieren ist jeweils das kleine Dreieck rechts neben einem Menüpunkt anzuklicken.

Projekt 1 (Stammbaum):

Zur Einführung soll ein Stammbaum erstellt werden mit den zugehörigen Verwandschaftsverhältnissen. Vorgegeben sei ein Prädikat eltern, das als ersten Parameter den Vater, als zweiten die Mutter und als dritten das Kind nennt. Die Namen sind Atome und als solche klein zu schreiben oder in Hochkomma ‘ einzuschließen.

```
eltern(peter,kathi,conny).
eltern(peter,kathi,ulrich).
eltern(klaus,conny,bastian).
eltern(bastian,conny,corinna).
eltern(fritz,irene,lotti).
eltern(fritz,irene,alfred).
eltern(fritz,irene,christa).
eltern(ulrich,lotti,moritz).
eltern(ulrich,lotti,fabian).
eltern(ulrich,lotti,anna).
eltern(ulrich,lotti,theresa).
```

Nun kann man weitere Prädikate ergänzen:

```
vater(K,V):-eltern(V,_,K).
mutter(K,M):-eltern(_,M,K).
oma(K,O):-vater(K,V),mutter(V,O).
oma(K,O):-mutter(K,M),mutter(M,O).
```

Zur Erläuterung:
K,V,M sind Variable und daher mit großem Anfangsbuchstaben geschrieben. K steht für ein Kind, V für einen Vater und M für eine Mutter. In dem Prädikat vater interessiert die Mutter nicht, sie ist daher durch den Unterstrich _ ersetzt, ähnlich bei dem Prädikat mutter.
Bei oma gibt es zwei Möglichkeiten: Die Verwandschaft entsteht über den Vater oder die Mutter. Daher wird das Prädikat oma durch zwei Klauseln definiert entsprechend diesen Möglichkeiten.
Gibt man im Hauptfenster oma(anna,X). ein, so werden die Großmütter aufgelistet.

Aufgabe: Prädikate onkel, tante und vetter, cousine definieren

Dass Prolog Prädikate rekursiv aufrufen kann, verwenden wir, um **Primzahlen** zu finden. Dabei soll zunächst untersucht werden, ob eine Zahl A durch einen Teiler T zwischen 2 und A-1 ohne Rest teilbar ist. Ist das der Fall, so ist die Zahl eine Primzahl. Dies geschieht mit dem Prädikat teiler(A,T):

teiler(A,T):-E is A/T,integer(E).

*A/T wird berechnet (**is**), der Wert ersetzt die Variable E. Ist der Wert eine natürliche Zahl, so ist T Teiler von A.*

Ist T kein Teiler von A, so ist die auf T folgende Zahl zu untersuchen:

teiler(A,T:-B is T+1,B<A,teiler(A,B).

Diese Klausel wird aufgerufen, wenn T kein Teiler ist. B=T+1 ersetzt dann T in teiler() .

teiler(A,2) liefert somit den Wert false, wenn auch die zweite Klausel ohne Ergebnis abbricht, also bei B=A. teiler() insgesamt liefert true, wenn es einen Teiler zwischen 2 und A-1 gibt, und false, wenn A eine Primzahl ist. Man kann daher definieren:

prim(A):-not(teiler(A,2)),write(A),nl.

not kehrt den Wert eines Prädikates um. Ist teiler(A,2) falsch, also A eine Primzahl, so ist not(teiler(..)) wahr. Der Rechner überprüft die Prädikate write und nl, die immer wahr sind. write schreibt auf den Bildschirm, nl erzeugt einen Zeilenvorschub.
Ist teiler(A,2) wahr, also A keine Primzahl, so ist not(teiler(..)) falsch und der Rechner bricht die Untersuchung des Wertes von prim vorzeitig mit der Meldung false ab.

Ein Programm, das alle Primzahlen bis 1000 berechnet, kann dann so aussehen:

```
teiler(A,T) :- E is A/T, integer(E).
teiler(A,T) :- B is T+1, B<A, teiler(A,B).
prim(A) :- not(teiler(A,2)), write(X), write(' ').

alle(X) :- prim(A), fail.
alle(X) :- X<1000, Z is X+1, alle(Z).
```

Die letzte Klausel bewirkt eine Schleife, die von einem Startwert X(=3) bis 1000 durchlaufen wird. Das Prädikat prim kann nicht in dieser Klausel benutzt werden, da prim den Wahr heitswert wahr (Primzahl) oder falsch (keine Primzahl) haben kann und damit die letzte Klausel vorzeitig abbricht, wenn eine Zahl gefunden wird, die keine Primzahl ist. Daher steht vor der letzten Klausel eine weitere, in der prim aufgerufen wird. Wird alle() aufgerufen, so testet Prolog, ob die erste Klausel zu alle() richtig ist. Ist dies der Fall (bei einer Primzahl), so führt der Rechner die letzte Klausel nicht mehr aus, also somit nicht die Schleife. Umd dies zu verhindern, muss die erste alle()-Klausel immer den Wert falsch liefern. Dazu wird zum Ende der Klausel das Prädikat fail aufgerufen, das immer falsch ist. Der Rechner testet also in der ersten alle()-Klausel, ob X eine Primzahl ist, meldet dann aber immer false und geht daher zu der zweiten alle()-Klausel, die dann alle() rekursiv mit einer neuen Zahl aufruft.

Das Programm ist kurz, aber trickreich. Die Sprache Prolog ist einfach gehalten. Allerdings muss man sich an einigen Stellen Tricks wie oben einfallen lassen, damit das Programm das Gewünschte tut. Um nochmals die Arbeitsweise von Prolog zu veranschaulichen, versuchen wir, logische Schaltungen mit Prolog zu simulieren. Bekanntermaßen läßt sich jede **logische Schaltung** aus AND-, OR- und NOT-Gattern zusammensetzen. NOT in Prolog haben wir bereits kennen gelernt. Es fehlen noch AND und OR. Argumente von AND und OR sind jeweils zwei Wahrheitswerte, die in konkreten Schaltungen durch 0 (false) und 1 (true) dargestellt werden. Die fehlenden Klauseln sind:

```
and(A,B):- A,B.
or(A,_):-A.          or(A,B) ist wahr, wenn A wahr ist.
or(_,B):-B.          or(A,B) ist auch wahr, wenn B wahr ist.
```

Damit lassen sich komplexe logische Ausdrücke simulieren, wie sie bei der Entwicklung von Schaltungen entstehen: Beispiel:

```
expr(A,B):- or(and(A,not(B)),and(not(A),B)).      exklusiv-Oder, Halbaddierer
```

Das folgende Programm ist komplizierter, zeigt aber Probleme, die die Programmierung stark beeinträchtigen können: Das Programm soll (einfache) **Funktionen differenzieren**. Dazu werden zunächst die bekannten Ableitungsregeln formuliert:

```
abl(X,0):-integer(X).                  Ableiten von Konstanten  (ganze Zahlen)
alb(X,0):-float(X).                                             (Dezimalzahlen)
abl(x,1).                              Ableiten von f(x)=x
abl(F**N,A*N*F**M):-M is N-1,abl(F,A). Ableiten von Potenzen, Kettenregel
abl(F+G,A+B):-abl(F,A),abl(G,B).       Summenregel (Summen)
abl(F-G,A-B):-abl(F,A),abl(G,B).                    (Differenzen)
abl(F*G,F*B+G*A):-abl(F,A),abl(G,B).   Produktregel
```

Ganzrationale Funktionen werden einwandfrei abgeleitet. Das Programm lässt sich einfach erweitern, so dass es auch u.a. trigonometrische Funktionen bearbeitet. Allerdings sind die Ausgaben kompliziert, da das Programm noch nicht Terme zusammen fassen kann. Wir versuchen dies durch das Prädikat zus(A,N) zu erreichen. Dabei wird man verschiedene Klauseln formulieren, die einen Term A zu N vereinfachen. Wenn sich A nicht vereinfachen lässt, findet der Rechner keine Klausel für die Behandlung von A und meldet false. Damit unterbleibt auch eine Ausgabe des Terms. Um dies zu verhindern, muss die letzte Klausel zu zus() zus(A,A) sein, so dass der Rechner immer eine Klausel zus() findet, die true meldet, auch wenn A nicht verändert wird.

Man könnte nun wie oben einfach die bekannten Regeln (Kommutativ-, Assoziativ-, Distributivgesetz) formulieren wie zum Beispiel:

```
zus(A*(B*C),H):-zus((A*B)*C,H).
zus((A*B)*C,H):-zus(A*(B*C),H).
```

Man produziert hierduch allerdings eine Endlosschleife, aus A*(B*C) wird zunächst (A*B)*C gemacht, dann durch die zweite Klause wieder A*(B*C) usw. . Um dies zu vermeiden, kann man rechts in einer Klausel den Cut "!" einschieben:

```
zus(A*(B*C),H):-!,zus((A*B)*C,F).
```

Der Cut verhindert, das das Programm nach Überschreiten wieder nach links des Cuts zurückkehren kann. Der Wirkung des Cuts ist nicht einfach zu überblicken und kann zu unerwünschten Effekten führen.

Die zweite Möglichkeit, um Enlosschleifen zu vermeiden, besteht darin, dass man die bekannten Gesetze in mehrere Klauseln aufspaltet, durch die eine Bearbeitungsrichtung festgelegt wird, wenn eine bestimmte Situation auftritt.

```
zus(A*(B*C),H):-zus(A*B,F),zus(F*C,H).  Assoziativgesetz für Produkte
```

```
zus(A+B+C,H):-zus(A+B,F),zus(F+C,H).    Assoziativgesetz für Summen
```

Wir müssen dem Rechner noch beibringen, dass er sich mit den einzelnen Summanden bzw. Faktoren beschäftigt, bevor er die Summe bzw. das Produkt bildet:

```
zus(A+B,F+G):-zus(A,F),zus(B,G).        Zusammenfassen von Summanden
zus(A*B,F*G):-zus(A,F),zus(B,G).        Zusammenfassen von Faktoren
```

Zum Schluss formulieren wir Klauseln, mit der einfache Rechnungen bzw. Umformungen durchgeführt werden:

```
zus(A+0,B):-zus(A,B).                   Addition von 0 verändert A nicht
zus(0+A,B):-zus(A,B).
zus(0*_,0).                             Multiplikation mit 0 ergibt 0
zus(_*0,0).
zus(1*A,B):-zus(A,B).                   Multiplikation mit 1 verändert A nicht
zus(A*1,B):-zus(A,B).
zus(A**1,B):-zus(A,B).                  Exponent 1 verändert A nicht
zus(A*B,D):-integer(A),integer(B),D is A*B.   Produkt von Zahlen berechnen
zus(A+B,D):-integer(A),integer(B),D is A+B.   Summe von Zahlen berechnen
zus(A,A).                               nichts tun, falls nichts anderes geht

dif(F,H):-abl(F,G),zus(G,H).            Differenzieren, indem zunächst die Ableitung durch abl()
                                        bestimmt und diese dann durch zus() vereinfacht wird
```

Das Programm liefert für die Funktion $f(x)=(4*x^2 + x)^5$ die erwartete Ableitung, bei anderen nicht unbedingt optimale Terme. Man kann sich überlegen, wie man die Klauseln zu zus(A,B) erweitern kann, damit weitere Ableitungen optimal zusammengefasst werden.

Als letztes Beispiel, bevor wir uns eine KI-Anwendung vornehmen, behandeln wir die Verwaltung einer **Adressdatenbank** Eine Datenbank muss neue Daten aufnehmen und nach Daten suchen können. Daten als solche kennt Prolog nicht, sondern nur Prädilate und die zugehörigen Klauseln.. Daher muss eine Adresse innerhalb einer Klausel zu einem Prädikat gespeichert werden. Ein Feature von Prolog ist, dass ein Programm selbst neue Klauseln erzeugen und an das Programm anhängen kann. Dies geschieht durch das Prädikat **asserta(A)**, wobei A für eine neue Klausel steht. Prolog versucht immer, durch Kombination von Klauseln richtige Aussagen zu finden. Um das Suchen von Klauseln zu steuern, kann man das Prädikat clause(A,B) einsetzen, wobei A der Name der zu suchenden Klausel ist und B einer der Wahrheitswerte true oder false: clause() sucht die erste Klausel nach dem letzten Fundstelle mit dem angegebenen Wahrheitswert. Der Wahrheitswert von clause richtet sich danach, ob eine entsprechende Klause gefunden werden konnte oder nicht.
Zum Einlesen von neuen Daten verwenden wir die Klausel readln(A), die als A eine Tastatur-eingabe nach Betätigen der Enter-Taste speichert.
Wenn wir die Adresse als Klausel dat(N,Z) speichern, wobei N der Name und Z zusätzliche Angaben sind, kann eine Adressdatenbank so aussehen:

```
daten(N):-clause(dat(N,Z),true),write(N),write(' : '),write(Z),nl.
daten(N):-write(N),write(' Zusatz:'),readln(Z),asserta(dat(X,Z)).
```

Die erste Klausel sucht einen Namen X und gibt diesen mit Zusatzangaben aus.
Findet der Rechner keine wahre dat-Klausel zu dem Namen N. Findet der Rechner keine dat-Klausel, so gibt er den Namen N aus, erfragt Zusantzangeben und fügt dann N und Z in einer neuen dat-Klause dem Programm hinzu. Zum Start ruft man daten mit einem Namen als Parameter auf

(Texte in Hochkomma ’ engeschlossen!). Die Ausgaben sind dann zwar sehr einfach, lassen sich aber durch weitere Klauseln komfortbler gestalten.

Expertensystem

Ein Expertensystem ist ein Programm, das aufgrund einer Datenbasis an eine Person nacheinander Fragen, die von den Antworten der Person abhängen, um dann zum Schluss eine Diagnose zu erstellen. Anwendungen derartiger System findet man als kommerzielle Programme in der Medizin, aber auch bei der Analyse von Defekten an Fahrzeugen.
Wenn das System dann auch noch selgstständig neue Fragen und Diagnosen lernen kann, so rechnet man es zum Bereich KI (künstliche Intelligenz). Wir werden ein System in Prolog programmieren, das auf optische Effekte verzichtet. Das Programm wird nicht viel größer als eine Datenbank (vgl. oben).
Es sind zu speichern: zu einer Frage eine Antwort und die nächste Frage. Diagnosen können wie neue Fragen behandelt werden, müssen aber kenntlich gemacht werden, damit keine weiteren Antworten und Fragen gestellt bzw. erwartet werden. Wir machen dies, indem wir als folgende Frage ein # angeben, Fragen sind in Hochkomma einzuschließen, wenn sie aus mehreren Wörtern bestehen oder mit Großbuchstaben beginnen.
Das Programm kann dann so aussehen:

```
frage(X):-clause(dat(X,_,#'),true),write(X).
```
Ausgabe einer Diagnose

```
frage(X):-write(X),write(':'),readln(Y),clause(dat(X,Y,Z),true),frage(Z).
```
Ausgabe einer Frage, Tastatureingabe einer Antwort,
Suchen eines Datensatzes zu dieser Frage (X) und dieser Antwort (Y),
Ausgabe der passenden neuen Frage/Diagnose (Z)

```
frage(X):-write(X),write(' - Antwort unbekannt, Anwort wiederholen:'),
```
readln(Y),nl,write('neue Frage/Diagnose:'),readln(Z),asserta(dat(X,Y,Z)).
Falls die ersten beiden Klauseln nicht zutreffen, also keine Diagnose vorliegt und die Antwort nicht bekannt ist, wird diese Klausel aufgerufen. Allerdings kann die in der zweiten Klausel eingegebene Antwort nicht an diese Klausel übergeben werden, so dass die Antwort nochmals erfragt wird.
Als Reaktion auf die Antwort wird eine neue Frage oder eine Diagnose angefordert, die dann zusammen mit der alten Frage und der Antwort in der Klausel dat(X,Y,Z) gespeichert wird.

Man benötigt noch eine Startfrage, die entweder bei jedem Programmstart neu eingegeben wird oder in einem neuen Prädikat fest gehalten wird:

```
start():-frage(‚Fühlst du dich krank’).
```

Das Programm arbeitet nun folgendermaßen:
Es stellt eine Frage, man gibt eine Antwort ein. Falls die Antwort bekannt ist, folgt eine weitere Frage, Sonst wird die Eingabe einer neuen Frage oder Diagnose erwartet, die dann anschließend auf dem Bildschirm ausgegeben werden.
Bei einer neuen Diagnose wird ebenfalls eine Antwort erfragt. Diese hat keine Bedeutung und kann leer bleiben. Als darauf folgende Frage ist dann # als Kennzeichen einer Diagnose einzugeben. Wird die Diagnose nach einem Fragendurchlauf erneut ausgegeben, so wird nicht mehr eine Antwort erwartet, sondern das Programm beendet.

Die Ein- und Ausgaben sind primitiv. SWI-Prolog bietet eine Schnittstelle zu Windows, mit der sich

diese Unzulänglichkeit beheben lässt. Die Schnittstelle verbindet zwei unterschiedliche Programmierkonzepte, das von Prolog und das von C. Dadurch ergeben sich einige Besonderheiten:

Prolog - Windows

Um die Besonderheiten zu verstehen, sollte man wissen, wie Windows mit Programmen kommuniziert: Programme und Windows tauschen miteinander Daten und Statusmeldungen in Form von „Botschaften“ aus, die in einer Warteschlange zwischen gelagert werden. Durch diese Pufferung ist es nicht klar, wann eine Botschaft dann verarbeitet werden kann. Hierdurch ergeben sich besondere Effekte bei der Programmierung.
Prolog arbeitet mit Klauseln, die nicht direkt miteinander direkt Daten austauschen können. Das bedeutet für das Erstellen einer grafischen Oberfläche, dass alles innerhalb einer Klausel geschehen muss. Sollen Daten verarbeitet werden, so muss man die Daten als Parameter an andere Prädikate übergeben, die in dieser Klausel aufgerufen werden. Die Prädikate können aber keine Daten an die aufrufende Klausel zurück geben, die dann dort für die grafischen Oberfläche genutzt werden können.

Soll Windows eine Aktion ausführen, so sendet der Befehl **send(D,append,*Botschaft*)** eine entsprechende Botschaft an die Warteschlange, die zu einem Fenster D gehört, und hängt diese an das Ende der Schlange an (append).
Sendet Windows eine Botschaft an das aufrufende Programm, so ist mit dem Prädikat message zu bestimmen, was mit der Botschaft geschehen soll. message wartet auf eine Botschaft und ruft beim Eintreffen ein Prolog-Prädikat auf und übergibt ihm Daten als Parameter.
Im Einzelnen:

new(F,dialog(*'Titel'*)) *produziert ein* **neues Fenster** *mit 'Titel' als Kopfzeile mit einem Link F zu diesem Fenster.*
send(F,append,text('*Inhalt*')) **zeigt den Text** *'Inhalt' im Fenster F an.*

send(F,append,text_item('*Sujet*',X)), *zeigt den Text ,Sujet an, danach ein* **Eingabefeld** *an mit dem durch X übergebenen Inhalt*
send(F,append,button('*Text*',message(..))) *erzeugt einen* **Button** *mit der Aufschrift ,Text'.*

Will man **Daten von Eingabefeldern** abrufen, so benötigt man eine **Kennung** K, die Windows dem Feld zuteilt und über das es identifiziert werden kann. Dies geschieht über den Befehl

send(F,append,new(K,button(..)) bzw. send(F,append,new(K,text_item(..))

K?selection *übergibt Daten, die in dem Feld K eingegeben wurden.*

message() kann in zwei Weisen genutzt werden:

message(F?frame,free) **schließt das Fenster** *F z.B. nach dem Betätigen eines Button*

message(@prolog,ausw,K?selection,L?selection,P...) *ruft das Prädikat ausw auf und übergibt ihm als Parameter die Daten der Felder K und L und den Prolog-Paramter P*

Die Position der Grafik-Elemente richtet sich nach der Reihenfolge, in der sie erzeugt werden, die Größe wird von Windows bestimmt (in Java bekannt als Flow-Layout).
Um **Fenster F anzuzeigen**, muss als letztes Prädikat der erzeugenden Klausel eingegeben werden:

send(F,open)
Als Beispiel werden wir die früher erstellte Datenbank mit einer grafischen Oberfläche versehen:

```
-use_module(library(pce)).          /*bindet die Windows-Schnittstelle ein*/

/*Menü:*/
anf:-new(D,dialog('Eingabe')),
     send(D,append,button('Neu',message(@prolog,ein,save))),        /*Bemerkung 1 (siehe unten)*/
     send(D,append,button('Ausgabe',message(@prolog,ein,read))),    /*Bemerkung 2 (siehe unten)*/
     send(D,append,button('Ende',message(D?frame,free))),
     send(D,open).

/*Fenster für Dateneingabe mit variabler Aktion F:*/
ein(F):-new(D,dialog('Eingabe')),
     send(D,append,new(N,text_item('Name'))),
     send(D,append,new(Z,text_item('Zusatz'))),
     send(D,append,button('OK',message(@prolog,F,N?selection,Z?selection))),
     send(D,open).

/*Ausgabefenster für die Daten N,Z:*/
aus(N,Z):-new(D,dialog('Ausgabe')),
     send(D,append,text_item('Name',N)),
     send(D,append,text_item('Zusatz',Z)),
     send(D,append,button('OK',message(D?frame,free))),
     send(D,open).

save(N,Z:-asserta(daten(N,Z)).      /*neuen Datensatz anlegen*/
read(N,_):-dat(N,Z),aus(N,Z).       /*Datensatz suchen, in neuem Fenster ausgeben*/
```

Bemerkung 1:
message(@prolog,ein,save) ruft in Prolog das Prädikat ein mit Parameter save auf, also ein(save). ein erzeugt ein Eingabefenster, über das nach Betätigen des Button ’Ok‘ save aufgerufen wird mit den eingegebenen Texten als Parametern N und Z. save(N,Z) legt die Texte in einer neuen Klausel dat(N,Z) ab. Das von ein() erzeugte Fenster bleibt allerdings geöffnet und kann zu weiteren Eingaben genutzt werden.

Bemerkung 2:
message(@prolog,ein,read) ruft in Prolog das Prädikat ein mit Parameter read auf, also ein(read). ein erzeugt ein Eingabefenster, in dem ein Namen eingegeben werden kann, dessen Daten ausgegeben werden sollen. Betätigt man in ein() den Button ’Ok‘, so wird read() aufgerufen mit den eingegebenen Texten als Parameter N und Z, also read(N,Z). read(N,Z) sucht einen zu N passenden Datensatz dat(N,D), dabei werden Eingaben zu ’Zusatz‘ ignoriert. read(N;D) ruft dann das Prädikat aus(N,D) auf, das die Daten in einem neuen Fenster ausgibt.

Das Programm ist so lauffähig und kann in SWI-Prolog kopiert werden.