



Deutsches  
Forschungszentrum  
für Künstliche  
Intelligenz GmbH

**Document**  
D-91-01

**Relfun/X**  
**An Experimental Prolog Implementation of**  
**Relfun**

**Werner Stein      Michael Sintek**

**March 1991**

**Deutsches Forschungszentrum für Künstliche Intelligenz  
GmbH**

Postfach 20 80  
D-6750 Kaiserslautern  
Tel.: (+49 631) 205-3211/13  
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3  
D-6600 Saarbrücken 11  
Tel.: (+49 681) 302-5252  
Fax: (+49 681) 302-5341

# Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth  
Director

**Relfun/X**  
**An Experimental Prolog Implementation of Relfun**

**Werner Stein , Michael Sintek**

DFKI-D-91-01



© Deutsches Forschungszentrum für Künstliche Intelligenz 1990

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.



Relfun/X  
An Experimental Prolog Implementation of  
Relfun

Michael Sintek      Werner Stein

Universität Kaiserslautern  
W-6750 Kaiserslautern, F.R. Germany

March 14, 1991

**Abstract**

Relfun/X is an experimental implementation of Relfun, a relational and functional language developed by Harold Boley at Kaiserslautern University. It is totally implemented in Prolog; additionally, the Relfun/X programs are compiled into Prolog programs (i.e. "consulted" analogously to the ordinary consulting scheme of Prolog).

While Relfun/X does not provide all the features of the Lisp-based Relfun implementation, it is the first running version supporting Relfun's multi-footed clauses.





# Contents

<b>1</b>	<b>Introductory Examples</b>	<b>3</b>
<b>2</b>	<b>Additional Features</b>	<b>7</b>
<b>3</b>	<b>Differences in Syntax</b>	<b>7</b>
<b>4</b>	<b>Sequence Variables</b>	<b>7</b>
<b>5</b>	<b>Important Modules of the Relfun/X Implementation</b>	<b>8</b>
5.1	Flattener . . . . .	8
5.2	Smoother . . . . .	8
5.3	Sequence Variable Unification . . . . .	8
<b>6</b>	<b>Builtins</b>	<b>9</b>
<b>7</b>	<b>Distribution</b>	<b>9</b>
<b>A</b>	<b>File “README”</b>	<b>10</b>
<b>B</b>	<b>File “init.pro”</b>	<b>12</b>
<b>C</b>	<b>File “load-relfun.pro”</b>	<b>15</b>
<b>D</b>	<b>File “operators.pro”</b>	<b>16</b>
<b>E</b>	<b>File “builtins.pro”</b>	<b>17</b>
<b>F</b>	<b>File “relfun.pro”</b>	<b>21</b>
F.1	Compiler . . . . .	21
F.2	Run Time Functions . . . . .	29
F.3	Main Procedure . . . . .	32
<b>G</b>	<b>File “init.rfx”</b>	<b>35</b>
<b>H</b>	<b>The “demo*.rfx” Files</b>	<b>36</b>
H.1	File “demo.rfx” . . . . .	36
H.2	File “demo1.rfx” . . . . .	40
H.3	File “demo2.rfx” . . . . .	41
H.4	File “demo3.rfx” . . . . .	42
H.5	File “demo4.rfx” . . . . .	43
H.6	File “demo5.rfx” . . . . .	44
H.7	File “demo6.rfx” . . . . .	45
H.8	File “demo7.rfx” . . . . .	46
H.9	File “demo8.rfx” . . . . .	47



## 1 Introductory Examples

We think that introducing a new language is best done by providing some simple examples that show you some of its most significant features.

The following examples are included in the demo files provided with the Relfun/X system so you can test them all by yourself. Refer to the 'README' file to find out how to start the Relfun/X system.

For further Relfun examples and a general introduction see [1] and the "demo\*.rfx" files in the appendix.

### Example 1 *Relatives*

We start with the standard example found in more or less every introductory book on Prolog: facts and simple rules concerning relatives. But instead of representing the facts and rules as relations (as in `mother(john,helen)`) they have been transformed to – sometimes non-deterministic – functions (the prefix `&` precedes the return value, backquotes mark constants).

```
father('john) :- & 'graham.
mother('john) :- & 'helen.
brother('helen) :- & 'fred.
brother('helen) :- & 'bert.
parent(X) :- & father(X).
parent(X) :- & mother(X).
uncle(X) :- & brother(parent(X)).
```

```
! &- uncle('john).
fred
yes ;
bert
yes ;
no
```

### Example 2 *Fibonacci - relational and functional*

This example shows how the translation process could take place: the internal Prolog representation of the functional and the relational version is nearly the same, i.e. the return value can easily be implemented via an additional parameter.



```

% relational version
% -----

fibrel(0,'s(0)).
fibrel('s(0),'s(0)).
fibrel('s('s(N)),F) :- fibrel(N,X), fibrel('s(N),Y), plusrel(X,Y,F).

plusrel(0,N,N).
plusrel('s(M),N,P) :- plusrel(M,'s(N),P).

% functional version
% -----

fibfun(0)          :- & 's(0).
fibfun('s(0))      :- & 's(0).
fibfun('s('s(N)))  :- & plusfun(fibfun(N),fibfun('s(N))).

plusfun(0,N)       :- & N.
plusfun('s(M),N)   :- & plusfun(M,'s(N)).

! &- fibrel('s('s('s('s('s(0))))),X) , X.
s(s(s(s(s(0))))))
yes

! &- fibfun('s('s('s('s('s(0)))))).
s(s(s(s(s(s(s(0))))))))
yes

! &- fibfun(X).
s(0)
yes ;
s(0)
yes ;
s(s(0))
yes ;
s(s(s(0)))
yes ;
s(s(s(s(0))))
yes

```



**Example 3** *Greatest Common Divisor*

```
gcd(X,Y) :-  
    less(X,Y),  
    & gcd(X,sub(Y,X)).
```

```
gcd(X,Y) :-  
    less(Y,X),  
    & gcd(Y,sub(X,Y)).
```

```
gcd(X,X) :- &X.
```

```
! &- gcd(26,585).
```

```
13
```

```
yes
```

**Example 4** *Multi-Footed Clauses*

Relfun/X supports multi-footed clauses, i.e. a clause may return more than just one value at a time. Furthermore it is possible for a clause to have more than one arity (for input as well as output) as the following example demonstrates:<sup>1,2</sup>

```
multiple(X) :- &null.  
multiple(X) :- &X.  
multiple(X) :- &(X,X).  
multiple(X) :- &(X,X,X).
```

```
! &- multiple(5).
```

```
()
```

```
yes ;
```

```
5
```

```
yes ;
```

```
5 , 5
```

```
yes ;
```

```
5 , 5 , 5
```

```
yes ;
```

```
no
```

---

<sup>1</sup>(v1,v2,...) denotes a sequence variable

<sup>2</sup>unlike standard Relfun, Relfun/X allows every variable to be bound to a sequence





```

! &- multiple(1,2,3).
()
yes ;
1 , 2 , 3
yes ;
1 , 2 , 3 , 1 , 2 , 3
yes ;
1 , 2 , 3 , 1 , 2 , 3 , 1 , 2 , 3
yes ;
no

```

**Example 5 Higher-Order Functions**

Relfun/X does not yet implement the full higher-order functions found in [1] but it provides you with the standard 'call' mechanism of Prolog and with an additional infix function '@' that acts like ":" in Backus' FP or funcall in Lisp.

```
reduction(Function,Baseelement,[]) :- &Baseelement.
```

```
reduction(Function,Baseelement,[H|T]) :-
    & (Function @ (H,reduction(Function,Baseelement,T))).
```

```
count(A,B) :- &add(B,1).
```

```
! &- reduction('count',0,[1,4,3,2])
```

```
10
yes
```

```
! &- reduction('count',0,[1,4,3,2]).
4
yes
```



## 2 Additional Features

For reasons of symmetry we have allowed active calls everywhere in a clause; this even includes the head, the left-hand side of the ‘is’ builtin, structures and lists (even when they are backquoted). This implies having to backquote all constant structures and non-numerical atoms.

Programmers familiar with languages like Hope will thus be able to use conjunctive pattern matching as in<sup>3</sup>

```
hope(List is [Head|Tail]) :- ... ,
```

where ‘is’ is a function which unifies both arguments with the input value (comparable to **&** in Hope).

## 3 Differences in Syntax

As a consequence of using the Prolog builtin ‘read’ to consult Relfun/X programs, there are some (insignificant) differences between Relfun/X and the Relfun syntax in [1]:

1. Because **&** acts like any other function, it must be preceded by a ‘,’ or a space depending on the place where it is used. (Remark: **&** is implemented as an assignment of the form **Return-Variable is Return-Value**; consequently **&** can be used anywhere in a clause, not only as its last component (see [1]) which is very useful in combination with ‘;’<sup>4</sup>).
2. All terms of the form **functor[args]** (see [1]) have to be transcribed to ‘**functor(args)**’.
3. The empty sequence variable **()** has to be written as ‘**()**’.

## 4 Sequence Variables

In Relfun/X, variables are implemented as sequence variables, i.e. a variable may have an arbitrary ‘arity’. It is thus possible for functions to return more than one value using just one return variable (see example 4: ‘Multi-Footed Clauses’ and subsections 5.1 and 5.2)<sup>5</sup>.

---

<sup>3</sup>see “demo8.rfx” in appendix H

<sup>4</sup>the meaning of ‘;’ is the same as in Prolog

<sup>5</sup>it is still problematic to treat infix operators correctly in the context of sequence variables: it is difficult to keep both arguments apart, thus problems may arise when using “is” and other infix operators), but we cannot give an example here



## 5 Important Modules of the Relfun/X Implementation

### 5.1 Flattener

The flattener unpacks nested structures, e.g.

`Z is f(g(X))` becomes `Y is g(X), Z is f(Y)`.

For more details see [1].

### 5.2 Smoother

Internally, sequence variables are represented as Prolog structures of the form `#[V1,V2,...]`. It is necessary to smooth these structures from time to time because at runtime different sequence variable structures may denote the same value, e.g. `(1,2,3)` is equivalent to `#[1,2,3]`, `#[1,#[2,3]]`, and to `#[#[1,2],3]`. The smoother maps all these various forms to `#[1,2,3]` or, generally, to the unnested internal representation<sup>6</sup>.

### 5.3 Sequence Variable Unification

In this version, sequence variables are simply unified by applying the ordinary Prolog unification to the smoothed variables<sup>7</sup>. This implies that sequence variables of different 'arities' cannot be unified yet<sup>8</sup>.

---

<sup>6</sup>unary sequences (numbers and variables) are not transformed to `#` structures; nullary structures are represented as functor`#` `[]` to simplify unification

<sup>7</sup>full string unification – equality unification under associativity – seems far to costly

<sup>8</sup>it even seems to be necessary to implement two different kinds of variables to overcome this dilemma, as in Boley's FIT or in other pattern matching languages



## 6 Builtins

The Relfun/X system provides you with the following builtins<sup>9,10</sup>:

<b>is</b>	: $T^2 \rightarrow T$	: (infix operator)
<b>null</b>	: $\emptyset \rightarrow \emptyset$	
<b>add</b>	: $\mathcal{N}^2 \rightarrow \mathcal{N}$	
<b>sub</b>	: $\mathcal{N}^2 \rightarrow \mathcal{N}$	
<b>times</b>	: $\mathcal{N}^2 \rightarrow \mathcal{N}$	
<b>quot</b>	: $\mathcal{N}^2 \rightarrow \mathcal{N}$	
<b>rem</b>	: $\mathcal{N}^2 \rightarrow \mathcal{N}$	
<b>less</b>	: $\mathcal{N}^2 \rightarrow \emptyset$	
<b>show</b>	: $T \rightarrow \emptyset$	: display a term
<b>showln</b>	: $T \rightarrow \emptyset$	: display a term and start a new line
<b>cr</b>	: $\emptyset \rightarrow \emptyset$	: start a new line
<b>getch</b>	: $\emptyset \rightarrow \mathcal{N}$	: read a character (ASCII code)
<b>++</b>	: $\mathcal{A}^2 \rightarrow \mathcal{A}$	: concatenate atoms (infix operator)
<b>recons</b>	: $\mathcal{A} \rightarrow \emptyset$	: reconsult a Relfun/X file
<b>srecons</b>	: $\mathcal{A} \rightarrow \emptyset$	: reconsult a Relfun/X file silently
<b>goalcall</b>	: $T \rightarrow T$	: call a goal and return its value
<b>call</b>	: $T \rightarrow T$	: call(Goal) is equivalent to goalcall(& Goal)
<b>funccall</b>	: $T \rightarrow T$	: same as call
<b>@</b>	: $T^2 \rightarrow T$	: same as ":" in FP or funccall in Lisp (see example 5) (infix operator)
<b>naf</b>	: $T \rightarrow \emptyset$	: negation as failure
<b>!</b>	:	: same as in Prolog
<b>fail</b>	:	: same as in Prolog
<b>;</b>	:	: same as in Prolog

## 7 Distribution

You may redistribute this program freely. Report any bugs or improvements via email to: real-fun@dfki.uni-kl.de

---

<sup>9</sup>see the files "builtins.pro" and "init.rfx", which you can easily extend by yourself

<sup>10</sup> $\mathcal{N}$  denotes the set of numbers,  $\mathcal{A}$  the set of atoms,  $T$  the set of terms





## A File "README"

Some hints to run Relfun/X on your system:

-----  
- your current directory should contain at least the following files:

```
load-relfun.pro
init.pro
operators.pro
builtins.pro
relfun.pro
```

```
init.rfx
demo.rfx
demo?.rfx
```

- load Relfun/X by "reconsult('load-relfun.pro')."
- the file "init.pro" contains some standard prolog procedures which are not available on all prolog systems (member, reverse, ...)  
(modify this file (and "operators.rfx") to run Relfun/X on your prolog system)
- there must be the file "init.rfx", which usually contains the definitions of "null", "()" and some other basic functions that must be available to run your Relfun/X programs properly
- the file "operators.pro" contains operator declarations
- there should be a file "demo.rfx" containing a procedure "demo"; this file will be reconsulted and then started if you enter "\*demo." at the Relfun/X prompt



- if you are running the Relfun/X system under sepia prolog, add the following lines to your ".sepiarc" file:

```
:- cprolog.  
  
correct_prompt :-  
    set_prompt(input, '', output).  
  
:- reconsult('load-relfun.pro')
```

- if you use any other prolog, add "correct\_prompt." to the "init.pro" file

- after invoking the Relfun/X system (by typing "relfun." at the prolog prompt), a Relfun/X file can be loaded by

```
recons('filename').
```

where filename must be enclosed in single quotes (') if it contains any special characters like . or /, e.g.

```
recons('test.rfx').
```

(You may type "recons('user')" to enter procedures directly;  
quit by typing ^D)

- at the Relfun/X prompt, \*func switches to function mode which treats entered goals like funcall; analogously, \*goal turns on goal mode which acts like goalcall



## B File "init.pro"

```
% -----
% "init.pro" : some general procedures
% -----

% modify this file for your local prolog system

% standard procedures
% -----

/*
append( [],L,L).
append([FL1|RL1],L2,[FL1|L3]) :- append(RL1,L2,L3).

member(X,[X|_]).
member(X,[_|Rest]) :-
    member(X,Rest).
*/

reverse([],[]).
reverse([H|T],L) :-
    reverse(T,Trev),
    append(Trev,[H],L).

% some additional procedures
% -----

% correct_prompt.      % use this on all prolog systems except sepia

/*
abort:-          % abort the prolog system
    ???
*/

/*
systemcall(Command) :-
    unix(system(Command)).
*/
```



```

systemcall(Command) :- % Command must be an atom
    system(Command).      % sepia prolog

/* use this if your 'system/1' expects a string as a list of numbers
   (as in cprolog)

systemcall(Command) :-
    name(Command,SCommand),
    system(SCommand).

*/

readchar(C) :- % read next char from current input stream (including CR)
    get0(C).      % get_char(C) on the KCM

listp([]).

listp([_|_]).

% global variables (via assert and abolish)
% -----

del_global(Var) :- abolish(Var,1).
del_global(Var).

set_global(Var,Value) :-
    del_global(Var),
    Clause =.. [Var,Value],
    asserta(Clause).

get_global(Var,Value) :-
    Clause =.. [Var,Value],
    call(Clause).

```





```
% list and structure conversion
% -----

l2s([X],X) :- !.

l2s([X|Rest],(X,SRest)) :-
    l2s(Rest,SRest).

s2l(Var,[Var]) :-
    var(Var), !.

s2l((X,Y),[X|YY]) :- !,
    s2l(Y,YY).

s2l(X,[X]).
```



## C File "load-relfun.pro"

```
% -----  
% "load-relfun.pro" : load all Relfun/X files  
% -----  
  
?- reconsult('init.pro').  
  
?- reconsult('operators.pro').  
  
?- reconsult('builtins.pro').  
  
?- reconsult('relfun.pro').  
  
?-      nl,  
        write('+-----+'),nl,  
        write('| Relfun/X   --   Version 0.2   January 1991 |'),nl,  
        write('|      (c) Michael Sintek & Werner Stein      |'),nl,  
        write('+-----+'),nl,  
        nl,nl,  
        write('(Type "relfun." to start)'),  
        nl,nl.
```



## D File “operators.pro”

```
% -----  
% Relfun/X operator declarations  
% -----  
  
?- op(10,fy,#).  
?- op(10,fy,*).  
  
?- op(10,fy,').  
?- op(11,xfx,@).  
?- op(12,fx,&).  
  
?- op(101,xfx,:=).  
?- op(102,xfx,<=>).  
?- op(103,xfy,++).  
?- op(104,xfx,===).  
?- op(105,xfx,&).
```



## E File "builtins.pro"

```
% -----  
% Relfun/X builtins (prolog imports)  
% -----  
  
recons(EV,#[]) :-  
    smooth(EV,Atom),  
    get_atom(Atom,Filename),  
    rf_reconsult(Filename,verbose).  
  
srecons(EV,#[]) :-  
    smooth(EV,Atom),  
    get_atom(Atom,Filename),  
    rf_reconsult(Filename,silent).  
  
true(#[],#[]).  
  
add(EV,#[RV]) :-  
    smooth(EV,EV1),  
    EV1 = #[X,Y],  
    RV is X + Y.  
  
sub(EV,#[RV]) :-  
    smooth(EV,EV1),  
    EV1 = #[X,Y],  
    RV is X - Y.  
  
times(EV,#[RV]) :-  
    smooth(EV,EV1),  
    EV1 = #[X,Y],  
    RV is X * Y.  
  
quot(EV,#[RV]) :-  
    smooth(EV,EV1),  
    EV1 = #[X,Y],  
    RV is X // Y.
```





```

rem(EV,#[RV]) :-
    smooth(EV,EV1),
    EV1 = #[X,Y],
    RV is X mod Y.

less(EV,#[]) :-
    smooth(EV,EV1),
    EV1 = #[X,Y],
    X < Y.

is_(EV,V1) :-
    (EV = #[EV1,EV2]; EV = # [#[EV1,EV2]]),
    smooth(EV1,V1),
    smooth(EV2,V2),
    V1 == V2.

% don't use call or is inside call/funccall/goalcall!

funccall(EV,RV) :-
    smooth(EV,Goal),
    unsmooth(Goal,RealGoal),
    relfun_call(RealGoal,RV,func).

goalcall(EV,RV) :-
    smooth(EV,Goal),
    unsmooth(Goal,RealGoal),
    relfun_call(RealGoal,RV,goal).

syscall(EV,#[]) :-
    smooth(EV,EV1),
    get_atom(EV1,Command),
    systemcall(Command). % defined in "init.pro"

```



```

Q(EV,RV) :-
    smooth(EV,Goal),
    alpha1(Goal,RV).

alpha1(#[],#[]) :- !.           % empty function

alpha1(#[Function|Args], RV) :- !,
    alpha2(Function,Args,RV).

alpha1(Function,RV) :- !,
    alpha2(Function,[],RV).

alpha2(SAtom,Args,RV) :-
    get_atom(SAtom,Atom), !,
    Goal =.. [Atom,# Args,RV],
    call(Goal).

alpha2(Error,Args,RV) :- !,
    write('illegal call: '),
    write(Error),
    write(' (')', write(Args), write(')'),nl,
    !,fail.

show(EV,#[]) :-
    smooth(EV,EV1),
    unsmooth(EV1,EV2),
    write(EV2).

showln(EV,#[]) :-
    smooth(EV,EV1),
    unsmooth(EV1,EV2),
    write(EV2), nl.

cr(EV,#[]) :-
    smooth(EV,#[]),
    nl.

getch(EV,RV) :-
    smooth(EV,#[]),
    readchar(RV).

```



```
++(EV,RV) :-
    smooth(EV,[SAtom1,SAtom2]),
    get_atom(SAtom1,Atom1),
    get_atom(SAtom2,Atom2),
    name(Atom1,StringAtom1),
    name(Atom2,StringAtom2),
    append(StringAtom1,StringAtom2,StringAtom),
    name(Atom,StringAtom),
    RVAtom =.. [Atom,#[]],
    RV == RVAtom.
```



## F File "relfun.pro"

```
% -----  
% 'relfun.pro' : Relfun/X to Prolog Compiler  
% (c) real-fun -- Werner Stein & Michael Sintek  
%                Kaiserslautern University  
%                Germany  
% -----
```

### F.1 Compiler

```
%%%%%%%%%%%%  
% COMPILER %  
%%%%%%%%%%%%
```

```
fctrename(call,funcall) :- !.  
fctrename(is,is_) :- !.  
fctrename(F,F).
```

```
% flat - level 1  
% -----
```

```
% flat(Goal,GoalList) :  
%   Goal is an ordinary goal  
%  
%
```

```
flat(Term,[]) :-  
    (number(Term); listp(Term)), !,  
    write(Term),  
    write(' is not a goal. '), nl,  
    fail.
```

```
flat('Term,_) :- !,  
    nl,  
    write('),  
    write(Term),  
    write(' is not a goal. '), nl,  
    fail.
```





```

flat(Call,NewGoalList) :-
    Call =.. [call|LGoal], !,
    NewCall =.. [funcall|LGoal],
    flat(NewCall,NewGoalList).

flat(Goal,[Goal]) :-
    atom(Goal), !.

flat(Var is FunctionCall,NewGoalList):-
    var(Var),
    nonvar(FunctionCall),
    FunctionCall =.. [Function|Args],
    atom(Function),
    Function \== '.',
    Function \== ',',
    Function \== '[]',
    Function \== ''', !,
    flat1(Args,GoalList,Args1),
    fctrename(Function,NewFunction),
    FunctionCall1 =.. [NewFunction|Args1],
    append(GoalList,[Var := FunctionCall1],NewGoalList).

flat(Goal,NewGoalList) :-
    Goal =.. [Functor|LGoal], !,
    flat1(LGoal,GoalList,LGoal1),
    Goal1 =.. [Functor|LGoal1],
    append(GoalList,[Goal1],NewGoalList).

% flat - level 2
% -----

flat1(Term,[],Term) :-
    (var(Term);
    number(Term);
    Term == []),
    !.

flat1(Term,GoalList,Term1) :-
    Term =.. [Functor|Args],
    Functor \== '.', !,
    flat1([Term],GoalList,[Term1]).

```



```

flat1([Term|Rest],GoalList,[Term|LRestGoal]) :-
    var(Term), !,
    flat1(Rest,GoalList,LRestGoal).

flat1([Term|Rest],GoalList,[Term|LRestGoal]) :-
    (number(Term); Term = []), !,
    flat1(Rest,GoalList,LRestGoal).

flat1([[Head|Tail]|Rest],GoalList,[List|LRestGoal]) :- !,
    flat1([Head|Tail],GoalList1,List),
    flat1(Rest,GoalList2,LRestGoal),
    append(GoalList1,GoalList2,GoalList).

flat1(['X|Rest],GoalList,[X|LRestGoal]) :-
    (var(X) ; number(X)), !,
    flat1(Rest,GoalList,LRestGoal).

flat1(['[H|T]|Rest],GoalList,[Term1|LRestGoal]) :- !,
    Term = [H|T],
    Term =.. [Functor|LTerm], !,
    flat1(LTerm,GoalList1,LTerm1),
    Term1 =.. [Functor|LTerm1],
    flat1(Rest,GoalList2,LRestGoal),
    append(GoalList1,GoalList2,GoalList).

flat1(['Term|Rest],GoalList,[Term1|LRestGoal]) :-
    Term =.. [Functor|LTerm], !,
    flat1(LTerm,GoalList1,LTerm1),
    Term1 =.. [Functor,# LTerm1],
    flat1(Rest,GoalList2,LRestGoal),
    append(GoalList1,GoalList2,GoalList).

flat1([(Term1,Term2)|Rest],GoalList,[Term|LRestGoal]) :- !,
    flat1([Term1],GoalList1,[FTerm1]),
    flat1([Term2],GoalList2,[FTerm2]),
    connect(FTerm1,FTerm2,Term),
    append(GoalList1,GoalList2,GoalList3),
    flat1(Rest,GoalList4,LRestGoal),
    append(GoalList3,GoalList4,GoalList).

```



```

flat1([Term|Rest],GoalList,[NewVar|LRestGoal]) :- !,
    Term =.. [Functor|LTerm],
    flat1(LTerm,GoalList1,LTerm1),
    fctrename(Functor,NewFunctor),
    Term1 =.. [NewFunctor|LTerm1],
    append(GoalList1,[NewVar := Term1],GoalList2),
    flat1(Rest,GoalList3,LRestGoal),
    append(GoalList2,GoalList3,GoalList).

% strind : structural induction
% -----

strind(Var,_,Function,Paras,T) :-
    var(Var), !,
    Goal =.. [Function,Var,Paras,T],
    call(Goal).

strind(S,FunctionSet,Function,Paras,T) :-
    S =.. [Functor,Arg1,Arg2],
    member(Functor,FunctionSet), !,
    strind(Arg1,FunctionSet,Function,Paras,Arg1s),
    strind(Arg2,FunctionSet,Function,Paras,Arg2s),
    T =.. [Functor,Arg1s,Arg2s].

strind(S,_,Function,Paras,T) :- !,
    Goal =.. [Function,S,Paras,T],
    call(Goal).

% add_exitpoint(Body,NewBody)
% -----

find_exitpoint(Term) :-
    var(Term), !, fail.

find_exitpoint(Term) :-
    Term =.. [&|_], !.

find_exitpoint((Term1,Term2)) :- !,
    (find_exitpoint(Term1);
    find_exitpoint(Term2)).

```



```

find_exitpoint((Term1;Term2)) :- !,
    (find_exitpoint(Term1);
     find_exitpoint(Term2)).

find_exitpoint(Goalcall) :-
    Goalcall =.. [goalcall|Goallist], !,
    l_find_exitpoint(Goallist).

find_exitpoint(_) :- !, fail.

l_find_exitpoint([]) :- !, fail.

l_find_exitpoint([H|T]) :-
    find_exitpoint(H);
    l_find_exitpoint(T).

add_exitpoint(Goal,Goal) :-
% goal with exitpoints: no path without exitpoint allowed
    find_exitpoint(Goal), !.

add_exitpoint(Goal,(Goal,&true)).

% tg : transform goal
% -----

tg(Var := Goal,RV,TGoal) :- !,
    Goal =.. [Functor|Args],
    TGoal =.. [Functor,# Args,Var], !.

tg(Var1 is Var2,RV,Var1 == Var2) :- !.

tg(!,RV,! ) :- !.

tg(fail,RV,fail) :- !.

tg(Goal,RV,TGoal) :- !,
    Goal =.. [Functor|Args],
    TGoal =.. [Functor,# Args,_], !.

```





```

ls_tg([Goal],RV,TGoal) :- !,
    tg(Goal,RV,TGoal), !.

ls_tg([Goal|Rest],RV,(TGoal,TRest)) :- !,
    tg(Goal,RV,TGoal),
    ls_tg(Rest,RV,TRest), !.

trans(Var,_,_) :-
    var(Var), !,
    nl,
    write('A variable is not an allowed goal.'), nl,
    fail.

trans(Goal,RV,TGoal) :-
    Goal =.. [&|Args], !,
    l2s(Args,SArgs),
    flat(RV is SArgs,LTGoal),
    ls_tg(LTGoal,_,TGoal), !.

trans(Goal,RV,TGoal) :- !,
    flat(Goal,GoalList),
    ls_tg(GoalList,RV,TGoal), !.

transform(Body,RV,TBody) :-
    strind(Body,[' ',';'],trans,RV,TBody).

% th : transform head
% -----

th(Head,RV,THead,(smooth(EV,# []))) :-
    atom(Head), !,
    THead =.. [Head,EV,RV].

th(Head,RV,THead,(smooth(EV,EV1),TGoal)) :-
    Head =.. [Functor|Args],
    THead =.. [Functor,EV,RV],
    l2s(Args,SArgs),
    flat(EV1 is SArgs,LTGoal),
    ls_tg(LTGoal,_,TGoal).

```



```

% rf_reconsult
% -----

rf_abolish(Clause) :-
    abolish(Clause,2),
    abolish(Clause,3).

rf_abolish(Clause).

rf_test(Functor) :-
    get_global(clauses,[Functor|_]), !.

rf_test(Functor) :-
    get_global(clauses,Clauses),
    member(Functor,Clauses), !,
    write('Warning: '),
    write(Functor),
    write('" is not contiguously defined.'),
    nl.

rf_test(Functor) :- !,          % first time
    rf_abolish(Functor),
    get_global(clauses,Clauses),
    set_global(clauses,[Functor|Clauses]), !.

rf_assert(Clause) :-
    Clause =.. [(:-),Head,Body], !,
    th(Head,RV,THead,TGoal),
    add_exitpoint(Body,Body1),
    transform(Body1,RV,TBody),
    THead =.. [Functor|_],
    rf_test(Functor),
    assertz((THead :- (TGoal,TBody))), !.

rf_assert(Fact) :-
    th(Fact,# [],THead,TBody),
    Fact =.. [Functor|Args],
    rf_test(Functor),
    assertz((THead :- TBody)), !.

```



```

rf_reconsult(File,Mode) :- !,
    set_global(clauses,[]),
    see(File),
    repeat,
    read(Clause),
    rf_reconsult1(File,Clause,Mode), !.

rf_reconsult1(File,end_of_file,silent) :- !,
    seen,
    del_global(clauses).

rf_reconsult1(File,end_of_file,verbose) :- !,
    seen,
    write('reconsult file '),
    write(File),
    write(' reconsulted, procedures:'), nl,
    get_global(clauses,Clauses),
    reverse(Clauses,RClauses),
    write(RClauses), nl,nl,
    del_global(clauses).

rf_reconsult1(File,Clause,_) :-
    (rf_assert(Clause);
    nl,
    write('Error: '), nl,
    write(Clause), nl,
    write('is not a valid clause.'),nl,nl),
    !, fail.

```



## F.2 Run Time Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% run time functions %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% connect (used in smooth)
% -----

connect(T1,T2,T) :-
    connect1(T1,T2,NT),
    ((NT = [X], !, T = X) ;
     T = # NT).

connect1(T1,T2,List) :-
    nonvar(T1), nonvar(T2),
    T1 = # List1, T2 = # List2, !,
    append(List1,List2,List).

connect1(T1,T2,[T1|List2]) :-
    nonvar(T2), T2 = # List2, !.

connect1(T1,T2,List) :-
    nonvar(T1), T1 = # List1, !,
    append(List1,[T2],List).

connect1(T1,T2,[T1,T2]) :- !.

% smooth(X,Y)
% -----

smooth(X,Y) :-
    var(X), var(Y), !,
    X = Y.

smooth(X,Y) :-
    var(X), !,
    smooth(Y,Y1),
    X = Y1.
```





```

smooth(# [X], X1) :- !,
    smooth(X,X1).

smooth(# [],# []) :- !.

smooth(# [Term|Rest], List) :- !,
    smooth(Term,STerm),
    smooth(# Rest,SRest),
    connect(STerm, SRest, List).

smooth(Term,STerm) :-
    Term =.. [Functor|Rest], !,
    l_smooth(Rest,SRest),
    STerm =.. [Functor|SRest].

smooth(X,X).

l_smooth([], []).

l_smooth([Head|Tail],[SHead|STail]) :-
    smooth(Head,SHead),
    l_smooth(Tail,STail).

% unsmooth : convert s(#[...]) to s(...)
% -----

unsmooth(Var,Var) :-
    var(Var), !.

unsmooth(# [], '()') :- !.

unsmooth(# [H|T], Structure) :- !,
    unsmooth(H,H1),
    unsmooth(T,T1),
    l2s([H1|T1],Structure).

unsmooth([], []) :- !.

unsmooth([H|T],[H1|T1]) :- !,
    unsmooth(H,H1),
    unsmooth(T,T1).

```



```

unsmooth(Structure, Functor) :-
    nonvar(Structure),
    get_atom(Structure, Functor), !.

unsmooth(Structure, Structure1) :-
    Structure =.. [Functor|X],
    X == [# Args], !,
    unsmooth(Args, Args1),
    Structure1 =.. [Functor|Args1].

unsmooth(Structure, Structure1) :-
    Structure =.. [Functor|Args], !,
    unsmooth(Args, Args1),
    Structure1 =.. [Functor|Args1].

unsmooth(X, X).

% get_atom(SAtom, Atom)
% -----

get_atom(SAtom, Atom) :- % (SAtom must be 'smoothed')
    SAtom =.. [Atom, X],
    X == #[] .

% === : partial "string-variable" unification
% -----

Term1 === Term2 :-
    smooth(Term1, Eq), smooth(Term2, Eq).

```



### F.3 Main Procedure

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% main procedure %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

?- set_global(mode,[func,'! &- ']).

relfun :-
    nl,
    rf_reconsult('init.rfx',verbose),
    write('(Enter "*help." for help)'),nl,nl,
    repeat,
    correct_prompt, % e.g. for sepia prolog
    get_global(mode,[Mode,Prompt]),
    write(Prompt),
    read(Goal),
    ((Goal == end_of_file,
     nl, nl, write('Good bye'), nl, abort);
     (command(Goal),
      prolog_goal(Goal),
      relfun_goal(Goal,Mode),
      fail))).

% command
% -----

command(Var) :-
    var(Var), !,
    write('Please be a little bit more specific'), nl,
    !,fail.

command(* func) :-
    set_global(mode,[func,'! &- ']),
    !,fail.

command(* goal) :-
    set_global(mode,[goal,'! ?- ']),
    !,fail.
```



```

command(* demo) :-
    rf_reconsult('demo.rfx',silent), !,
    demo(_,_),
    !,fail.

command(* help) :-
    nl,
    write('Available comands are'), nl,nl,
    write('  ^D          -- abort Relfun/X session'), nl,nl,
    write(' *func.       -- switch to function mode'),nl,
    write(' *goal.        -- switch to goal mode'), nl,
    write(' *demo.        -- run a demo'), nl,
    write(' *help.        -- show this help information'), nl,nl,
    write(' #<prolog-cmd>. -- execute prolog comand, e.g.:'), nl,
    write(' #listing(<f>). -- show compiled Relfun/X function <f>'),nl,
    nl,nl,
    !,fail.

command(_).

% prolog_goal
% -----

prolog_goal(# Goal) :- !,
    ((call(Goal), !, nl, write(#yes)); (!, nl, write(#no))), nl,
    fail.

prolog_goal(_).

% relfun_goal
% -----

read_key(Key) :- !,
    readchar(Key),
    read_key2(Key), !.

read_key2(10) :- !.

read_key2(_) :- !,
    readchar(Key),
    read_key2(Key), !.

```





```
relfun_call(Goal,RV,goal) :-  
    add_exitpoint(Goal,Goal1),  
    transform(Goal1,RV,Goal2),  
    call(Goal2).
```

```
relfun_call(Goal,RV,func) :-  
    transform(& Goal,RV,Goal1),  
    call(Goal1).
```

```
relfun_goal(Goal,Mode) :-  
    relfun_call(Goal,RV,Mode),  
    show(RV),  
    write('yes '),  
    read_key(Key),  
    Key \== 59,  
    nl, !.
```

```
relfun_goal(Goal,Mode) :-  
    write(no),  
    nl, nl, !.
```

```
show(RV) :-  
    smooth(RV,RVO),  
    unsmooth(RVO,RV1),  
    write(RV1), nl, !.
```



## G File "init.rfx"

```
null.
```

```
'()'.
```

```
naf(Goal) :- goalcall(Goal), !, fail.  
naf(Goal).
```



## H The "demo\*.rfx" Files

### H.1 File "demo.rfx"

```
% -----  
% "demo.rfx" : demo Relfun/X file, loaded by "*demo."  
% -----  
  
demo :-  
    cr,  
    showln('-----'),  
    showln('Relfun/X Demo      (c) Werner Stein & Michael Sintek'),  
    showln('-----'),  
    cr,cr,  
    demo1.  
  
demo1 :-      % relatives  
    showfile('demo1.rfx'),  
    recons('demo1.rfx'),  
    wait_for_cr,  
    showln('You may now enter questions like'),  
    showln('  brother('helen)'),  
    showln('  uncle('john)'),  
    cr,  
    showln('To invoke backtracking, type ";" at the "yes" prompt. '),  
    cr,  
    showln('Continue the demo by typing "demo2.". '),  
    cr,  
    showln('(If an error has stopped the Relfun/X session,)',  
    showln(' resume it by typing "relfun.")'),  
    cr.  
  
demo2 :-      % slow_sort  
    cr,  
    showfile('demo2.rfx'),  
    recons('demo2.rfx'),  
    wait_for_cr,  
    showln('You may now enter questions like'),  
    showln('  slow_sort(['s('s(0)),0,'s(0)]). '),  
    cr,  
    showln('Continue the demo by typing "demo3.". '),cr.
```



```

demo3 :-          % fib
    cr,
    showfile('demo3.rfx'),
    recons('demo3.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  fibrel(n2s(7),X), s2n(X).'),
    showln('  s2n(fibfun(n2s(5))).'),
    cr,
    showln('Continue the demo by typing "demo4.".'),cr.

demo4 :-          % math and lists
    cr,
    showfile('demo4.rfx'),
    recons('demo4.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  gcd(26,585).'),
    showln('  app([1,2,3],[4,5]).'),
    cr,
    showln('Continue the demo by typing "demo5.".'),cr.

demo5 :-          % lisp
    cr,
    showfile('demo5.rfx'),
    recons('demo5.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  last([1,2,3]).'),
    showln('  nth([1,2,3],2).'),
    cr,
    showln('Continue the demo by typing "demo6.".'),cr.

demo6 :-          % palindrome
    cr,
    showfile('demo6.rfx'),
    recons('demo6.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  palindrome(['s(Y,'b),'a,Y,X]),X,Y.'),
    cr,
    showln('Continue the demo by typing "demo7.".'),cr.

```





```

demo7 :-          % divide
    cr,
    showfile('demo7.rfx'),
    recons('demo7.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  divide(divide(11,3)).'),
    showln('  (which yields: divide(3,2) --> (1,1))'),
    cr,
    showln('Continue the demo by typing "demo8.".'),cr.

demo8 :-          % curiosities
    cr,
    showfile('demo8.rfx'),
    recons('demo8.rfx'),
    wait_for_cr,
    showln('You may now enter questions like'),
    showln('  hope([1,2,3,4]).'),
    showln('  hope([1,2,3]).'),
    cr,
    showln('  reduction('add,0,[1,2,3,4]).'),
    showln('  reduction('count,0,[1,2,3,4]).'),
    cr,cr,
    showln('(End of demo)'),
    cr.

demo9 :-
    cr,
    showln('game over'),
    cr.

```



```
% some general procedures
% -----

showfile(X) :-
    syscall('more ++ ' ' ' ++ X).

wait_for_cr :-
    show('Please press <CR> to continue: '),
    getcr,
    cr.

getcr :-
    10 is getch, !.

getcr :-
    _ is getch,
    getcr.
```



## H.2 File "demo1.rfx"

```
% "demo1.rfx" : relatives
% -----

father('john) :- & ('john_the_first).
mother('john) :- & 'helen.
brother('helen) :- & 'fred.
brother('helen) :- & 'bert.
parent(X) :- & father(X).
parent(X) :- & mother(X).
uncle(X) :- & brother(parent(X)).
child(parent(X)) :- &X.
```



### H.3 File "demo2.rfx"

```
% "demo2.rfx" : slow_sort
% -----

slow_sort(X) :- & sorted(perm(X)).

sorted([]) :- & [].
sorted([X]) :- & [X].
sorted([X,Y|Z]) :- <=(X,Y), & cons(X,sorted([Y|Z])).

perm([]) :- & [].
perm([X|Y]) :- & cons(U,perm(del(U,[X|Y]))).

del(X,[X|Y]) :- & Y.
del(X,[Y|Z]) :- & cons(Y,del(X,Z)).

<=(0,X) :- numberp(X).
<=('s(X),'s(Y)) :- <=(X,Y).

cons(X,Y) :- & [X|Y].

numberp(0).
numberp('s(N)) :- numberp(N).
```





#### H.4 File "demo3.rfx"

```
% "demo3.rfx" : fib
% -----

% relational version
% -----

fibrel(0,'s(0)).
fibrel('s(0),'s(0)).
fibrel('s('s(N)),F) :- fibrel(N,X), fibrel('s(N),Y), plusrel(X,Y,F).

plusrel(0,N,N).
plusrel('s(M),N,P) :- plusrel(M,'s(N),P).

% functional version
% -----

fibfun(0)          :- & 's(0).
fibfun('s(0))      :- & 's(0).
fibfun('s('s(N)))  :- & plusfun(fibfun(N),fibfun('s(N))).

plusfun(0,N)       :- & N.
plusfun('s(M),N)   :- & plusfun(M,'s(N)).

% conversion of "structure numbers" :
% -----

n2s(0)             :- &0.
n2s(N)             :- & 's(n2s(sub(N,1))).

s2n(0)             :- &0.
s2n('s(X))        :- & add(s2n(X),1).
```



## H.5 File "demo4.rfx"

```
% "demo4.rfx" : ordinary functions
% -----

gcd(X,Y) :-
    less(X,Y),
    & gcd(X,sub(Y,X)).

gcd(X,Y) :-
    less(Y,X),
    & gcd(Y,sub(X,Y)).

gcd(X,X) :- &X.

fac(0) :- !, &1.

fac(N) :- &times(N,fac(sub(N,1))).

app([],L) :- &L.

app([H|T],L) :- & ([H|app(T,L)]).

rev([]) :- &[].

rev([H|T]) :- &app(rev(T),[H]).

mem(X,[X|R]) :- &[X|R].

mem(X,[_|R]) :-
    &mem(X,R).
```



## H.7 File "demo6.rfx"

```
% "demo6.rfx" : palindrome  
% -----
```

```
palindrome([]) :- &true.
```

```
palindrome([Center]) :- &true.
```

```
palindrome([First_and_Last | Rest]) :-  
    apprel(Middle, [First_and_Last], Rest),  
    &palindrome(Middle).
```

```
apprel([], L, L).
```

```
apprel([H|T], L, [H | T1]) :- apprel(T, L, T1).
```



## H.8 File "demo7.rfx"

```
% "demo7.rfx" : string variables  
% -----
```

```
divide(N,D) :- & (quot(N,D), rem(N,D)).
```





## H.9 File "demo8.rfx"

```
% "demo8.rfx" : curious procedures
% -----

% Hope style pattern matching
% -----

% a pattern of the form "P1 is P2" binds the input to both P1
% and P2, e.g.:

hope(List is [Head | Tail is [Head1|Tail1]]) :-
    & (List,Head,Tail,Head1,Tail1).

% if the list [1,2,3,4] is provided to hope, it will be bound to
% List as well as to [Head | Tail]

% "Higher Order Functions"
% -----

reduction(Function,Baseelement,[]) :- &Baseelement.

reduction(Function,Baseelement,[H|T]) :-
    & (Function @ (H,reduction(Function,Baseelement,T))).

count(A,B) :- &add(B,1).
```



## References

- [1] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.





Deutsches  
Forschungszentrum  
für Künstliche  
Intelligenz GmbH

DFKI  
-Bibliothek-  
PF 2080  
6750 Kaiserslautern  
FRG

## DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.

## DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.

### DFKI Research Reports

#### RR-90-01

*Franz Baader*: Terminological Cycles in KL-ONE-based Knowledge Representation Languages  
33 pages

#### RR-90-02

*Hans-Jürgen Bürckert*: A Resolution Principle for Clauses with Constraints  
25 pages

#### RR-90-03

*Andreas Dengel, Nelson M. Mattos*: Integration of Document Representation, Processing and Management  
18 pages

#### RR-90-04

*Bernhard Hollunder, Werner Nutt*: Subsumption Algorithms for Concept Languages  
34 pages

#### RR-90-05

*Franz Baader*: A Formal Definition for the Expressive Power of Knowledge Representation Languages  
22 pages

#### RR-90-06

*Bernhard Hollunder*: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems  
21 pages

#### RR-90-07

*Elisabeth André, Thomas Rist*: Wissensbasierte Informationspräsentation:  
Zwei Beiträge zum Fachgespräch Graphik und KI:

1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

#### RR-90-08

*Andreas Dengel*: A Step Towards Understanding Paper Documents  
25 pages

#### RR-90-09

*Susanne Biundo*: Plan Generation Using a Method of Deductive Program Synthesis  
17 pages

#### RR-90-10

*Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann*: Concept Logics  
26 pages

#### RR-90-11

*Elisabeth André, Thomas Rist*: Towards a Plan-Based Synthesis of Illustrated Documents  
14 pages

#### RR-90-12

*Harold Boley*: Declarative Operations on Nets  
43 pages

#### RR-90-13

*Franz Baader*: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles  
40 pages

#### RR-90-14

*Franz Schmalhofer, Otto Kühn, Gabriele Schmidt*: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories  
20 pages

#### RR-90-15

*Harald Trost*: The Application of Two-level Morphology to Non-concatenative German Morphology  
13 pages



**RR-90-16**

*Franz Baader, Werner Nutt*: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification  
25 pages

**RR-91-01**

*Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka* : On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations  
20 pages

**RR-91-02**

*Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt*: The Complexity of Existential Quantification in Concept Languages  
22 pages

**RR-91-03**

*B.Hollunder, Franz Baader*: Qualifying Number Restrictions in Concept Languages  
34 pages

**RR-91-04**

*Harald Trost*  
X2MORF: A Morphological Component Based on Augmented Two-Level Morphology  
19 pages

**RR-91-05**

*Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist*: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.  
17 pages

**RR-91-06**

*Elisabeth André, Thomas Rist*: Synthesizing Illustrated Documents  
A Plan-Based Approach  
11 pages

**RR-91-07**

Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures  
13 pages

**RR-91-08**

*Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist*  
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation  
23 pages

**RR-91-09**

*Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta*  
RATMAN and its Relation to Other Multi-Agent Testbeds  
31 pages

**RR-91-10**

*Franz Baader, Philipp Hanschke*  
A Scheme for Integrating Concrete Domains into Concept Languages  
31 pages

**RR-91-11**

*Bernhard Nebel*  
Belief Revision and Default Reasoning: Syntax-Based Approaches  
37 pages

**RR-91-13**

*Gert Smolka*  
Residuation and Guarded Rules for Constraint Logic Programming  
17 pages

---

**DFKI Technical Memos**
**TM-89-01**

*Susan Holbach-Weber*: Connectionist Models and Figurative Speech  
27 pages

**TM-90-01**

*Som Bandyopadhyay*: Towards an Understanding of Coherence in Multimodal Discourse  
18 pages

**TM-90-02**

*Jay C. Weber*: The Myth of Domain-Independent Persistence  
18 pages

**TM-90-03**

*Franz Baader, Bernhard Hollunder*: KRIS: Knowledge Representation and Inference System -System Description-  
15 pages

**TM-90-04**

*Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profflich*: Terminological Knowledge Representation: A Proposal for a Terminological Logic  
7 pages

**TM-91-01**

*Jana Köhler*  
Approaches to the Reuse of Plan Schemata in Planning Formalisms  
52 pages





**TM-91-02***Knut Hinkelmann*Bidirectional Reasoning of Horn Clause Programs:  
Transformation and Compilation

20 pages

**TM-91-03***Otto Kühn, Marc Linster, Gabriele Schmidt*

Clamping, COKAM, KADS, and OMOS:

The Construction and Operationalization  
of a KADS Conceptual Model

20 pages

**TM-91-04***Harold Boley*

A sampler of Relational/Functional Definitions

12 pages

**TM-91-05***Jay C. Weber, Andreas Dengel and Rainer**Bleisinger*

Theoretical Consideration of Goal Recognition

Aspects for Understanding Information in Business

Letters

10 pages

**D-90-06***Andreas Becker: The Window Tool Kit*

66 Seiten

**D-91-01***Werner Stein, Michael Sintek*

Relfun/X - An Experimental Prolog

Implementation of Relfun

48 pages

**D-91-03***Harold Boley, Klaus Elsbernd, Hans-Günther Hein,**Thomas Krause*

RFM Manual: Compiling RELFUN into the

Relational/Functional Machine

43 pages

**D-91-04**

DFKI Wissenschaftlich-Technischer Jahresbericht

1990

93 Seiten

---

**DFKI Documents****D-89-01***Michael H. Malburg, Rainer Bleisinger:*

HYPERBIS: ein betriebliches Hypermedia-

Informationssystem

43 Seiten

**D-90-01**

DFKI Wissenschaftlich-Technischer Jahresbericht

1989

45 pages

**D-90-02***Georg Seul: Logisches Programmieren mit Feature**-Typen*

107 Seiten

**D-90-03***Ansgar Bernardi, Christoph Klauck, Ralf**Legleitner: Abschlußbericht des Arbeitspaketes*

PROD

36 Seiten

**D-90-04***Ansgar Bernardi, Christoph Klauck, Ralf**Legleitner: STEP: Überblick über eine zukünftige**Schnittstelle zum Produktdatenaustausch*

69 Seiten

**D-90-05***Ansgar Bernardi, Christoph Klauck, Ralf**Legleitner: Formalismus zur Repräsentation von**Geo-metrie- und Technologieinformationen als Teil*  
*eines Wissensbasierten Produktmodells*

66 Seiten



\_\_\_\_\_

**Relfun/X - An Experimental Prolog Implementation of Relfun**  
**Werner Stein , Michael Sintek**

**D-91-01**  
Document