Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-03

RFM Manual: Compiling RELFUN into the Relational/Functional Machine

(Second, Revised Edition)

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

July 1993

**Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH**

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
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 and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, 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
- ❏ Computer Linguistics
- ❏ Programming Systems
- ❏ Deduction and Multiagent Systems
- ❏ Document Analysis and Office Automation.

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.

Friedrich J. Wendl
Director

RFM Manual:
Compiling RELFUN into the Relational/Functional Machine
(Second, Revised Edition)

Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause

DFKI-D-91-03

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

RFM Manual: Compiling **RELFUN** into the Relational/Functional Machine

Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause

Universität Kaiserslautern
Fachbereich Informatik
Erwin-Schrödinger-Str.
67663 Kaiserslautern
Germany

Second Edition

July 28, 1993

Abstract

RELFUN's classifier produces a declarative clause language; its code generator optimizes target code for an underlying WAM emulator, called NyWAM. The parts are glued together by **RELFUN**'s user interface. All intermediate steps use explicit LISP S-expression representations, which can be displayed. The software is part of a LISP-based compilation laboratory for relational/functional languages.

Contents

1 Introduction

This document describes **RELFUN**'s classifier, code generator, and the underlying WAM emulator, called **NyWAM**. It is assumed that the reader is familiar with WAM architectures ([War83]) and **RELFUN**([Bol90], [BEH+93]). The software described herein is part of a compilation laboratory used in projects of the German Research Center for Artificial Intelligence. The projects are aimed at expert system development in the domain of mechanical engineering and materials engineering.
Let us give an idea of the compilation laboratory for those unfamiliar with the other documents ([Bol90], [Kra90], [Hei89], [Els90]).
This LISP-based laboratory consists of an interpreter and a 3-pass compiler well-suited for testing all compilation steps. The compiler is divided into an optional 'horizontal' transformer employing program-transformation techniques, a classifier producing an intermediate clause language, and a 'vertical' code generator producing target code for the **NyWAM** emulator. The parts are glued together by a command-line-oriented user interface. All steps produce intermediate output in a user-oriented LISP S-expression form.
Section 2 will explain the classified clauses. Section 3 will describe the code generator. In Section 4 we will reveal **NyWAM**'s([Nys]) internal structures.

2 The classifier

The code generator (vertical compiler) needs much information about the clauses and variables of a program (database), in order to generate RFM (WAM) code efficiently. The declarative intermediate language Classified Clauses represents this information explicitly; for this the classifier extends normal **RELFUN** source clauses with numerous declarations on different levels of description. The following short introduction is based on the implementation status of the Classified Clauses from November 1990. A more detailed introduction of an earlier version is presented (in German) in [Kra90]. This Section briefly describes the Classified Clauses; in section 2.6 the description grammar in an EBNF syntax is given.
In Classified Clauses we distinguish six levels of description, namely the database, procedure, clause, chunk, literal, and term levels. A knowledge base consists of an unordered set of procedures each consisting of an ordered set of clauses. All the clauses of procedures have the same name and arity. Name and arity yield the procedure name 'name/arity'. For example the clause `(hn (foo _v _w))` belongs to the procedure `foo/2`.
The Classified Clauses for a **RELFUN** program (database) are accordingly defined as follows:
classified_database ::= (**db**[1] {classified_procedure}*)

2.1 Procedure level

Syntax:

classified_procedure ::= (**proc** procedure_name clause_count {clause_classification}$^+$)

Description:

proc Each description of a procedure starts with the tag **proc**.

procedure_name The name and the arity of clauses yield the procedure name.

clause_count Clause_count gives the number of clauses belonging to the procedure.

Example:

Source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) (bar _p _q))
...
```

Classified Clauses:

```
(db (proc foo/2 2
          clause_classification
          clause_classification)
    ...)
```

[1]The db, temp, perm- tags are omitted in the current implementation

Remark:
It is planned for the future to extend the description of a procedure by information about the modes of the arguments in all feasible calls to the procedure. In this way it should be possible that, on the one hand, the user can declare the modes and, on the other hand, a mode interpreter can compute the modes automatically. Thus the mode interpreter could check the consistency of the modes generated by the user in exactly the same way.

2.2 Clause level

Syntax:

clause_classification	::=	(clause_type perm_var_list temp_var_list chunk_sequence)
chunk_sequence	::=	head_chunk_fact | head_chunk_rule body_chunk_list
perm_var_list	::=	(**perm** {global_perm_var_descr}*)
temp_var_list	::=	(**temp** {global_temp_var_descr}*)
global_perm_var_descr	::=	(variable perm_descr)
global_temp_var_descr	::=	(variable temp_descr)
perm_descr	::=	(Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr	::=	(X-reg_nr use_head use_premise)

Description:

clause_type The clause_type describes the kind of clauses, which are distinguished in **rel0**, **fun1den**, **fun1eva**, **fun*den**, **fun*eva**. We give the type **rel0** to a hn-clause without any body literal. Thus **rel0** tags an ordinary fact, as known from PROLOG. The "1" in the types **fun1den** and fun1eva indicates that the clause contains only one **chunk**. Hence "*" means the clause contains two or more **chunks**. "den" stands for denotative foot and "eva" for evaluative foot. It should be noted that an hn-clause with an evaluative last body literal still is a "den"-like clause, because hn-clauses implicitly return the value true and not the value of their last premise

perm_var_list (Global information about the permanent variables of the clause) An element of the perm_var_list is a pair of the form: (variable perm_descr). The perm_descr is a 3-tuple describing a.) where the variable has to be located in the local environment in order to make optimum environment trimming, b.) the occurrences in the head literal (a list of argument positions), and c.) the last occurrence (the last **chunk** and the last literal in this **chunk**) of the variable in the clause.

temp_var_list (Global information about the temporary variables in the clause) The temp_var_list describes which register (or X-reg_nr) has to be assigned to the temporary variable for register optimization on the machine level, and furthermore the occurrence in the head literal (or use_head) and the call literal (or use_premise). A temporary variable occurs only in one **chunk** by definition, in this way the call literal is unique and it is possible that neither use_head nor use_premise are different from the empty list nil.

Example:

Source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) (bar _p _q))
...
```

Classified Clauses:

```
(db (proc foo/2 2
          (rel0                 ; hn-clause without body goals
                (perm)          ; there are no  permanent variables
                (temp)          ; there are no temporary variables
                head_chunk_fact)


          (fun*eva              ; the ft-clause (foo _t ...). The
                                ; clause contains two small chunks
                                ; and an evaluative foot bar/2
                (perm (_p (1 nil (2 1)))); Permanent variable _p.
                                ; _p is assigned to the Y-reg 1 in the
                                ; local environment. _p doesn't occur
                                ; in the head. Its last occurrence is
                                ; in the second chunk and as the first
                                ; literal in the chunk.

                (temp (_t (1 (1) (1))) ; The temporary variable _t.
                                 ; _t is assigned to the X-reg 1. It
                                 ; has an occurrence in the head and
                                 ; call literal in the chunk at
                                 ; in the argument position 1.
                      (_q (2 nil (2)))) ; _q is assigned to register 2
                                 ; because its occurrence in the call
                                 ; literal is at argument position 2.
                head_chunk_rule
                body_chunk ))
     ... )
```

2.3 Chunk level

Syntax:

head_chunk_fact	::=	(**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule	::=	(**chunk** (head_literal {chunk_guard}* first_premise_literal}) chunk_descr)
body_chunk_list	::=	{body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk	::=	(**chunk** ({chunk_guard}* call_literal) chunk_descr)
call_literal	::=	literal_classification | lispcall_classification
chunk_guard	::=	builtin | passive_term
chunk_descr	::=	(lu_reg ({(variable permvar_uselit_list)}*))

permvar_uselit_list ::= ({arg_nr}$^+$)

Description:

body_chunk A chunk is a 2-argumented structure composed of the tag chunk, a list of denotative literals called chunk_guards with an additional evaluative literal called call_literal as the last element, and some information about the chunk called chunk_-descr.

head_chunk_fact If there are no call_iterals in the body of the clause, then the clause contains only one **chunk** ending with a denotative literal. We call this kind of **chunk** head_chunk_fact.In fact, all clauses with type **rel0** or **fun1den** are constructed with only the head_chunk_fact.

head_chunk_rule If there is at least one call literal in the clause, then the first **chunk** ends with an call literal (first_premise_literal). All clauses with types different from

```
                    (2 ((_p (2))) ); lu_reg = 2 because of the arity
                                   ; of the first_premise_literal. The
                                   ; permanent variable _p occurs at
                                   ; position 2 in the call_literal.
                (chunk             ; The tag for the second chunk.
                    (call_literal) ; There is only an call literal.
                    (2 ((_p (1)))))) ; _p occurs at position 1
                                       ; in the call_literal.
   ...)
```

2.4 Literal level and argument level

Syntax:

literal_classification	::=	(**usrlit** (functor arglist_classification) literal_descr)
lispcall_classification	::=	(lispcall_type (lisp-builtin arglist_classification) lispcall_descr)
builtin	::=	**unknown** | is_primitive | (**refl-Xreg** lhs_term)
arglist_classification	::=	{term_classification}*
term_classification	::=	constant_classification | variable_classification | structure_classification
is_primitive	::=	(**is** lhs_term rhs_term)
lhs_term	::=	constant_classification | variable_classification
rhs_term	::=	term_classification
constant_classification	::=	constant_name
variable_classification	::=	(variable local_var_descr)
structure_classification	::=	‘(functor arglist_classification) | (**inst** (functor arglist_classification))
local_var_descr	::=	(occurrence saveness var_class)
literal_descr	::=	(arity env_size arg_seq)
lispcall_descr	::=	(arity env_size arg_seq)

Description:

term_classification A term is a denotative literal. The inst_op (“‘” or “inst”) indicates that a literal is a denotative (sometimes called passive) one.

local_var_descr A variable is locally described (with respect to all its occurrences in the

clauses) by the local_var_descr. It is a list of three elements (occurrence saveness var_class). The occurrence can be first, nonfirst, or reuse. While the meaning of first and nonfirst is intuitively clear, reuse means that the classifier has assigned a register to more than one temporary variable. If a variable occurs first it gets the information reuse (instead of first) when the register was assigned to an other temporary variable heap), and unsafe (a possible reference to the local environment). The information var_class tells the code generator whether the variable is temp or perm.

literal_descr The arity gives the number of arguments in the literal.

env_size denotes how many permanent variables have to be survive the all to the literal. The Y-register assignment in the permvar_list has been done in a way that the env_size is as small as possible.

arg_seq is a list that tells the code generator in which order the argument positions have to be represented by RFM instructions. It's possible that some arguments need no instructions. A missing argument position in arg-seq indicates such a case.

Example:

Source:

```
(hn (foo alpha beta))
(ft (foo _t gamma) (bar _t _p) (bar _p _q))
...
```

Classified Clauses:

```
(db (proc foo/2 2
         (rel0 (perm) (temp)
              (chunk
                   ((usrlit (foo alpha beta)
                          (2 0 (1 2)))) ; The literal foo has 2
                                        ; arguments. The env_size is
                                        ; 0. Generate code for alpha
                                        ; first.
                   nil))                ; No chunk description needed
         (fun*eva
              (perm (_p (1 nil (2 1))))
              (temp (_t (1 (1) (1))) (_q (2 nil (2))))
              (chunk
                   ((usrlit (foo (_t (first safe temp)); _t occur
                                        ; first and is safe because
                                        ; it has a reference to the
                                   gamma); caller environment
                          (2 1 (2))); _t needs no instruction!!
                    (usrlit (bar (_t (nonfirst safe temp))
                                 (_p (first unsafe perm)))
                                        ; _p is potentially unsafe
                          (2 1 (2)))); As above!
                                        ; No instruction for _t
                   (2 ((_p (2)))) )
              (chunk
```

```
                          ((usrlit (bar (_p (nonfirst unsafe perm))
                                        (_q (first unsafe temp)))
                           (2 0 (1 2))))
                          (2 ((_p (1))))))
      ...)
```

Remark:
Further information about the meaning of the Classified Clauses is described in paragraph 3, where an introduction to the code generator is given. The code generator takes as input the Classified Clauses for **RELFUN** and produces the RFM code. Therefore, in paragraph 3 you can find more detailed information on how the added descriptions are used for code generation.

2.5 An example with structures

We consider the example "demostruc.rf" which is also used in the following paragraphs. In this example we can study in which way structures are represented in the Classified Clauses.

Source:

```
(ft (foo _2 _2 b)
    (is _w '(g _2))
    (is _2 '(f b))
    (bar b _w) )
(hn (bar _r _s))
...
```

Classified Clauses:

```
((proc bar/2  1
   (rel0                         ; bar/2 is an hn-fact
        (perm)                   ; No permanent variables
        (temp (_r (1 (1) nil)) ; 2 temporary variables
              (_s (2 (2) nil)))
        (chunk
              ((usrlit (bar (_r (first safe temp))
                            (_s (first safe temp)))
                     (2 0 (1 2)) )); Poposed instructions for position 1 and
              nil))))            ; 2, but the code generator will make it better

; Start of the description of the next procedure

(db (proc foo/3  1
      (fun1eva                   ; A one-chunk rule with an evaluative foot
           (perm)     ;
           (temp (_2 (1 (2 1) nil)) ; the variable _2 has no occurrence
                                    ; in the call_literal of its chunk
```

```
            (_w (2 nil (2))))
      (chunk
            ((usrlit (foo (_2 (first safe temp))
                          (_2 (nonfirst safe temp))
                          b) ; A constant gets no further description
                     (3 0 (3 1 2)) ) ; Generate code for the constant first!
             (is            ; All is-primitives are used denotatively
                 (_w (first unsafe temp)) ; in the Classified Clauses
                 '(g (_2 (nonfirst safe temp))) ) ; The structure g/2
                                                  ; beginning with ``''
             (is (_2 (nonfirst global temp))
                 '(f b))     ; A chunk guard gets no further description
             (usrlit (bar b
                          (_w (nonfirst unsafe temp)))
                     (2 0 (1)))); No instruction for _w necessary because
                                ; the register 2 is assigned to it
            (3 nil))))          ; lu_reg = 3, because of the literal foo/3
...)
```

2.6 EBNF syntax for Classified clauses

classified_database	::=	(**db** {classified_procedure}*)
classified_procedure	::=	(**proc** procedure_name clause_count {clause_classification}+)
clause_classification	::=	(clause_type perm_var_list temp_var_list chunk_sequence)
chunk_sequence	::=	head_chunk_fact | head_chunk_rule body_chunk_list
head_chunk_fact	::=	(**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule	::=	(**chunk** (head_literal {chunk_guard}* first_premise_literal}) chunk_descr)
body_chunk_list	::=	{body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk	::=	(**chunk** ({chunk_guard}* call_literal) chunk_descr)
chunk_descr	::=	(lu_reg ({(variable permvar_uselit_list)}*))
head_literal	::=	literal_classification
first_premise_literal	::=	call_literal
call_literal	::=	literal_classification | lispcall_classification
chunk_guard	::=	builtin | passive_term
passive_term	::=	term_classification
permvar_uselit_list	::=	({arg_nr}+)
literal_classification	::=	(**usrlit** (functor arglist_classification) literal_descr)
lispcall_classification	::=	(lispcall_type (lisp-builtin arglist_classification) lispcall_descr)
builtin	::=	**unknown** | is_primitive | (**refl-Xreg** lhs_term)
arglist_classification	::=	{term_classification}*
term_classification	::=	constant_classification | variable_classification | structure_classification
is_primitive	::=	(**is** lhs_term rhs_term)
lhs_term	::=	constant_classification | variable_classification
rhs_term	::=	term_classification
constant_classification	::=	constant_name
variable_classification	::=	(variable local_var_descr)
structure_classification	::=	'(functor arglist_classification) | (**inst** (functor arglist_classification))
perm_var_list	::=	(**perm** {global_perm_var_descr}*)
temp_var_list	::=	(**temp** {global_temp_var_descr}*)
literal_descr	::=	(arity env_size arg_seq)
lispcall_descr	::=	(arity env_size arg_seq)
global_perm_var_descr	::=	(variable perm_descr)
global_temp_var_descr	::=	(variable temp_descr)
perm_descr	::=	(Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr	::=	(X-reg_nr use_head use_premise)
local_var_descr	::=	(occurrence saveness var_class)
clause_type	::=	**rel0** | **fun1den** | **fun1eva** | **fun*den** | **fun*eva**
lispcall_type	::=	**cl-func** | **cl-pred** | **cl-extra**
Y-reg_nr	::=	reg_nr
X-reg_nr	::=	reg_nr

last_chunk	::=	chunk_nr
last_chunkliteral	::=	lit_nr
use_head	::=	({reg_nr}*)
use_premise	::=	({reg_nr}*)
arg_seq	::=	({arg_nr}*)
lu_reg	::=	reg_nr
occurrence	::=	**first** | **nonfirst** | **reuse**
saveness	::=	**global** | **safe** | **unsafe**
var_class	::=	**perm** | **temp**
variable	::=	_name | (**vari** name)
procedure_name	::=	name/arity
functor	::=	name
lisp-builtin	::=	lisp-fcts | lisp-preds | lisp-extras
lisp-fcts	::=	;;;;; **RELFUN** supported LISP functions
lisp-preds	::=	;;;;; **RELFUN** supported LISP predicates
lisp-extras	::=	;;;;; **RELFUN** supported LISP functions with side effects
constant_name	::=	name
clause_count	::=	cardinal
arg_nr	::=	cardinal
reg_nr	::=	cardinal
chunk_nr	::=	cardinal
lit_nr	::=	cardinal0
env_size	::=	cardinal0
arity	::=	cardinal0
name	::=	letter {letter | digit0}*
cardinal	::=	digit {digit0}*
cardinal0	::=	**0** | cardinal
letter	::=	**a** | **b** | ... | **z**
digit	::=	**1** | **2** | ... | **9**
digit0	::=	**0** | digit

2.7 The user interface and the code generator

The code generator produces WAM code from classified clauses and it is invoked after the classifier by typing **verti** at the RFM prompt. The idea of the classified clauses is to make the implicit structures of the compiler explicit in a declarative manner, thus allowing its output to be used in debugging sessions, for educational purposes, or further knowledge-based compilation steps. At the time of the **verti** command all **tups** must have been transformed into **cns** structures using the **untup** command. It is also assumed that flat clauses are in the database; flattening is performed by typing **flatter**. The **horizon** command comprises mainly these program transformations. See [Kra91] for further possible precompilation steps. Although this can be done automatically using **compile**, one might be interested in the individual source-to-source transformations the **horizon** command is performing.

The **verti** command collects all clauses starting with the same name and arity, and groups them together on the property list of the symbol determined by the procedure name, using the tag `'clauses`. This is necessary, because the basic entity in the WAM is a set of clauses with the same name and arity, a procedure.

Then the classifier and the code generator are called for each set of clauses on a procedure's property list. The target code is also stored on the property list, under the tag `'procedure`. It is possible to pretty print the code by typing **listcode**. The classified clauses are not stored on property lists, but can be simply reproduced by the **listclass** command.

The **compile** command can be called with an extra argument for compiling a single procedure, thus allowing procedure-based incremental compilation.

2.8 The user interface and the NyWAM

The user interface has two prompts: "`rfi>`" is displayed when the queries are sent to the interpreter and the interpreter database, while "`rfe>` " shows that the query, which possibly is a conjunction of literals, is compiled. The code obtained is stored under the name `main`, the datastructures for the variables in the query are created and their names and locations are memoized to get the variable names when the goal succeeds. Finally the emulator is called producing variable bindings or failures. When a goal succeeds the bindings are output and the user is asked whether he wants more results, giving him the opportunity to cause a failure and initiate backtracking so that the next solution may be computed. When **spy** is enabled, the query's compilation is output and the **NyWAM** is set into the debugger mode. With **nospy** this feature is turned off.

3 The code generator

The basic idea of the code generator is to keep it as simple as possible to allow an easy replacement of the **NyWAM** emulator by another abstract machine. The classified clauses should be considered as a 'machine-independent' representation of **RELFUN** procedures. It should be easy to modify the code generator to produce code for a C-based emulator.

The idea is to have associated with each nonterminal symbol a function returning code for that specific construct. The returned code is then appended to the other already existing code. This concept ensures a (more or less) functional structure of the code generator. But an append wastes time and cons-cells. Therefore, every call to **append** is done via the macros **doappend** and **addcode**. If runtime problems with the code generator should occur, these macros may be modified to expand to **nconc**.
The functions and macros will be introduced in the following. The descriptions of the function's parameters will not be given, so the reader should consult the source code, although the variable names should be self-explaining.
The source of the code generator has been written in a very functional style using only a small subset of COMMON LISP, having in mind a simple reimplementation of the code generator in **RELFUN**. Thus, we make extensive use of CONDs instead of using ecase, jump tables, and other specialities COMMON LISP is offering.

3.1 Software interface

The code generator has two access function from the outside (in the view of software modules). `(code-gen-proc classified_procedure)` is used to generate WAM code from a classified procedure. This is the function we use from the outside to compile a procedure incrementally.
In the future, the compilation of a single clause may become important for dynamic asserts and retracts. The appropriate function to produce WAM code for a single classified clause is `(code-gen-cc clause_classification)`.
If extensions to the code generator are made, one should ensure that this interface does not change.
In the following, functions for code generation are described. Nonterminals are used as input parameters representing the argument type. The rightarrows prefix the returned value of the system, which is often represented by nonterminal symbols. The symbols in bold case are the terminal symbols.

3.2 classified_procedure

classified_procedure ::= (**proc** procedure_name clause_count {clause_classification}$^+$)

- (s-cg-proc-id classified_procedure)
  → **proc**
- (s-cg-procedure_name classified_procedure)
  → procedure_name
- (s-cg-clause_count classified_procedure)
  → clause_count
- (s-cg-clause_classifications classified_procedure)
  → list of clause_classification(s)
- (code-gen-proc classified_procedure)
  → **NyWAM** code for the procedure. This procedure is responsible for generating try/retry/trust instructions.

3.3 clause_classification

clause_classification	::=	(clause_type perm_var_list temp_var_list chunk_sequence)
chunk_sequence	::=	head_chunk_fact | head_chunk_rule body_chunk_list

- (s-cg-clause_typ clause_classification)
  → clause_type
- (s-cg-perm_var_list clause_classification)
  → perm_var_list
- (s-cg-temp_var_list clause_classification)
  → temp_var_list
- (s-cg-chunks clause_classification)
  → list of head_chunk_fact or list of head_chunk_fact or list of head_chunk_rule body_chunk_rule.
- (code-gen-cc clause_classification)
  → **NyWAM** code for a classified clause. This function has to cope with **rel0**, **fun1den**, **fun1eva**, **fun*den** and **fun*eva** and for setting up an appropriate environment.

3.4 head_chunk_fact, head_chunk_rule, body_chunk

head_chunk_fact	::=	(**chunk** (head_literal {chunk_guard}*) chunk_descr)
head_chunk_rule	::=	(**chunk** (head_literal {chunk_guard}* first_premise_literal}) chunk_descr)
body_chunk_list	::=	{body_chunk}* [(({chunk_guard}*) chunk_descr)]
body_chunk	::=	(**chunk** ({chunk_guard}* call_literal) chunk_descr)

Let **chnk** be an abbreviation for *head_chunk_fact*, *head_chunk_rule* or *body_chunk*.

- (s-cg-chunk_id chnk)
  → **chunk**
- (s-cg-chunk_descr chnk)
  → chunk_descr
- (s-cg-chunk_head_literal chnk)
  → head_literal
- (s-cg-chunk_hd_cgfpl head_chunk_rule)
  → list: ((chunk_guard/s) first_premise_literal)
  remark: cgfpl = **c**hunk **g**uard, **f**irst **p**remise **l**iteral
- (s-cg-chunk-bd_cgcl body_chunk)
  → ((chunks_guard/s) call_literal)
  remark: cgcl = **c**hunk **g**uard, **c**all **l**iteral

- (code-gen-hdchunk perms temps chunk callexeflg deallocflg chunknr)
  This function returns code for the first chunk in the clause. One may notice that this function is very similar to code-gen-chunk below, although further enhancements (indexing, global compilation) may result in a complete reformulation of that function, whereas code-gen-chunk is likely to keep the same.

- (code-gen-chunk perms temps chunk callexeflg deallocflg chunknr)
  Returns WAM code for a chunk to be found in the body.

3.5 chunk_descr

chunk_descr ::= (lu_reg ({(variable permvar_uselit_list)}*))

- s-cg-chunk_lu_reg (chk_descr)
  → lu_reg

- s-cg-chunk_vpul (chk_descr)
  → list of (variable permvar_uselit_list)

3.6 literal_classification

literal_classification ::= (**usrlit** (functor arglist_classification) literal_descr)

- (s-cg-usrlit_id literal_classification)
  → **usrlit**

- (s-cg-literal_descr literal_classification)
  → literal_descr

- (s-cg-fac_list literal_classification)
  → (functor arglist_classification)
  remark: fac = functor arglistclassification

- (s-cg-functor fac)
  → functor

- (s-cg-arglist_classification fac)
  → arglist_classification

- (code-gen-head perms temps fac arg_seq)
  Generates code for the first literal in the clause.
  - (code-gen-head-arg place temps arg)
    Generates code for an argument place in the first literal in the clause.
  - (code-gen-head-temp place temps arg)
    Generates code for an X-variable in the first literal of a clause.
  - (code-gen-head-perm place temps arg)
    Generates code for a Y-variable in the first literal of a clause.

- (code-gen-tail perms temps arity permcnt fac callexeflg deallocflg cnknr litnr arg_seq)
  Generates code for the literals except the first in the clause.
  - (code-gen-tail-arg place perms temps arg chknr litnr)
    Generates code for an argument place in the literals except the first in the clause.
  - (code-gen-tail-temp place temps arg)
    Generates code for an X-variable in the body literals of a clause.
  - (code-gen-tail-perm place perms arg chknr litnr)
    Generates code for the literals except the first in the clause.

3.7 variable_classification, local_var_descr

variable_classification	::=	(variable local_var_descr)
local_var_descr	::=	(occurrence saveness var_class)

- (s-cg-local-var-descr variable_classification)
  → local_var_descr
- (s-cg-local_var_occurrence variable_classification)
  → local_var_occurrence
- (s-cg-local_var_saveness variable_classification)
  → local_var_saveness
- (s-cg-local_var_class variable_classification)
  → local_var_class

3.8 Global variables

- Emulator-related variables
  - *user-variables*
    Contains the user's variables when a query is issued.
  - *registers*
    The `define-register` functions adds each register to this list, causing the debugger to output the variables of this list.
  - *read-mode*
    This is a global flag in the machine indicating the read/write status, which is used in the unify instructions.
  - *emu-debug*
    This flag determines whether the emulator is in a debugging state or will just run through the code.
- code generator-related variables
  - *lureg*
    This variable determines which X-registers can be used by the code generator without any interference with the classifier's allocations.

- y-x-usage-list
  An assoc-list mapping Y variables to X-registers.

3.9 perm_var_list, temp_var_list

perm_var_list	::=	(**perm** {global_perm_var_descr}*)
temp_var_list	::=	(**temp** {global_temp_var_descr}*)
global_perm_var_descr	::=	(variable perm_descr)
global_temp_var_descr	::=	(variable temp_descr)

- (s-cg-perm_var global_perm_var_descr)
  → variable
- (s-cg-perm_descr global_perm_var_descr)
  → perm_descr
- (s-cg-temp_var global_temp_var_descr)
  → variable
- (s-cg-temp_descr global_temp_var_descr)
  → temp_descr

3.10 perm_descr, temp_descr

perm_descr	::=	(Y-reg_nr use_head (last_chunk last_chunkliteral))
temp_descr	::=	(X-reg_nr use_head use_premise)

- (s-cg-perm_y_nr perm_descr)
  → Y-reg_nr
- (s-cg-perm_use_head perm_descr)
  → use_head
- (s-cg-perm_last_literal perm_descr)
  → last_chunkliteral
- (s-cg-temp_x_nr temp_descr)
  → X-reg_nr
- (s-cg-temp_use_head temp_descr)
  → use_head
- (s-cg-temp_use_premise temp_descr)
  → use_premise

3.11 literal_descr

literal_descr	::=	(arity env_size arg_seq)

- (s-cg-arity literal_descr)
  → arity

- (s-cg-env_size literal_descr)
  → env_size

- (s-cg-arg_seq literal_descr)
  → arg_seq

3.12 lispcall_type, lispcall_classification

lispcall_classification	::=	(lispcall_type (lisp-builtin arglist_classification) lispcall_descr)
lispcall_type	::=	**cl-func** | **cl-pred** | **cl-extra** | **cl-relf**

- (cg-lispcall-p lispcall_classification)
  → t, if it is an external LISP call, nil otherwise

- (cg-lispcall-fun lispcall_classification)
  → lisp-function

- (cg-lispcall-args lispcall_classification)
  → arglist_classification

3.13 arglist_classification, term_classification, constant_classification

arglist_classification	::=	{term_classification}*
term_classification	::=	constant_classification | variable_classification | structure_classification
constant_classification	::=	constant_name
variable_classification	::=	see 3.7
structure_classification	::=	'(functor arglist_classification) | (**inst** (functor arglist_classification))

- (cg-inst-p term_classification)
  → t, if argument is an instantiation operator, nil otherwise

- (cg-s-inst-functor term_classification) (already knowing term is inst-op)
  → functor

- (cg-s-inst-funargs term_classification) (already knowing term is inst-op)
  → arglist_classification

- (arg-var-p term_classification)
  → t, if argument is a variable_classification, nil otherwise

- (arg-nil-p arglist_classification)
  → t, if argument is an empty list, nil otherwise

- (arg-const-p arglist_classification)
  → t, if argument is a constant, nil otherwise

3.14 Getting global information on variables

When it is known that a variable with a local description occurs, it is useful to look up the global information. At this level of processing, it is assumed that the code generator already has stored the global X- and Y-variable information in a local variable further referred to as **perms** and **temps**.

- (get_perm_descr arg_var perms)
  get the global information of the permanent variable arg_var.

- (get_temp_descr arg_var perms)
  get the global information of the temporary variable arg_var.

3.15 Obtaining the procedure arity

When processing with a classified procedure, the arity is needed. This is coded in the proce-

3.17 Y-variable scoreboarding

The idea of Y-variable scoreboarding is to safe memory bandwidth by remembering which Y-variable was already loaded into an X-register. Every time a Y-variable is 'touched', the corresponding X-register is saved as a pair (Y-variable X-register) on an assoc-list named `y-x-usage-list`, which is a global variable meaning that the Y-variable can also be found in an X-register.
The following functions are dealing with Y-variable scoreboarding:

- **(is-y-in-x y-vari y-x-usage-list)**
  This function associates the Y-variable with its X-argument position. If the Y-variable is not in an X-register, the result is `nil`.
- **(add-y-x-list y-vari x-reg y-x-usage-list)**
  This function adds a (Y-variable X-register) pair to the scoreboard.
- **(d_yreg_assoc yreg y-x-usage-list)**
  This is used to eliminate a pair specified by its Y-variable.
- **(d_xreg_assoc xreg y-x-usage-list)**
  This is used to eliminate a pair specified by its X-variable.

4 The NyWAM

A LISP-based emulator was obtained from Sven-Olof Nystrøm[Nys], Uppsala University. The present **NyWAM** version was modified to work within our Complab approach. This implementation could be replaced by some WAM implementation in C[Els90]. But the flexibility would be lost and the turnaround times would increase. Thus the **NyWAM** is an ideal prototype implementation choice.

4.1 Terminology

'Global Stack' and 'heap' as well as 'local stack', 'stack' and 'runtime stack' are synonyms, an environment and a choice point are portions of the local stack, the push-down list (PDL) is a stack used temporarily by the unification procedure, but it is not needed within the NyWAM, since this is done recursively in LISP. In most publications the A-registers are assumed to be the same as the X-registers and for those authors assuming disjoint A and X sets of registers the A-regs can be mapped to a single X-register set. Therefore argument registers will be referred herein as X-registers.

4.2 The datastructures

The WAM model assumes a tagged memory model. This means that memory locations are 'typed', e.g. that it is possible to tell which datatype is in the memory location. Since registers have neither tags nor addresses, with these it is only possible to handle references (or at most constants) but it is impossible to represent free variables, structures or lists directly. The tagged memory is handled by the (LISP) structure WORD:

Tag	Value
empty	undefined
ref	a memory address
struct	a memory address
list	a memory address
const	constant symbol
fun	a list (function-name arity)
code	a list (procedure-name . rest-of-instruction-list)
trail	a list of references to bound variables

The memory layout is shown in table 1. At the top are the low addresses, increasing downwards.

4.2.1 The local stack

The local stack contains environment and choicepoint frames. An environment must be created in a clause (using the `allocate` instruction) as soon as local variables become necessary.

A choice point is needed if there is more than one clause in a procedure. If a recent goal failed, the next clause must be explored with all argument registers appropriately (re-)set and the variables bound later than the invocation of the current clause restored to an unbound state.

heap (address 0)	← start-of-heap
...	
heap (address n)	← HB
heap (address n+m)	← H
...	
...	
maximum heap address	← start-of-stack−1
local stack	← start-of-stack
...	
environment and choicepoint frames	
...	
local stack	← A
...	
...	
	← memory-size

Table 1: The memory layout of the local and global stacks

previous environment pointer (CE)	← new E
previous continuation pointer (CP)	
Y-variable$_1$	
...	
Y-variable$_n$	
	← new A

Table 2: The memory layout of an environment

4.2.2 The heap

The heap holds compound terms. These compound terms may be lists or structures. The H-register points to the top of the heap, whereas the register HB is the (redundant) heap backtrack register used for speeding up references to the old heap pointer.

4.2.3 The trail

Contrary to other implementations the trail is realized as a LISP list. This is possible since no random access may happen on that structure. Either a reference is pushed on the trail (when a binding occurs) or the information is popped sequentially (when backtracking to a certain point occurs).

4.3 The registers

A register defined by `define-register` can be set using `(set-reg register value)` and referenced using `(reg register)`. Currently, there are 64 X-registers defined in the array.

X-register$_1$ ... X-register$_n$	
previous environment pointer (BCE)	
previous continuation pointer (BCP)	
previous choice point (B1)	
next clause pointer (BP)	
trail pointer (TR1)	
heap pointer (H1)	← new B
	← new A

Table 3: The memory layout of a choicepoint (backtrack point)

Register	Description	points to	Definition
P	program counter	program code	define-register
CP	continuation pointer	program code	define-register
E	last environment	local stack	define-register
B	last choicepoint	local stack	define-register
A	top of stack	local stack	define-register
TR	trail list		define-register
H	top of heap	heap	define-register
HB	heap backtrack point	heap	define-register
S	structure pointer	heap	define-register
X_i	registers	heap,stack	array

4.4 The instructions

The instructions are written in a LISP-like manner. The indexes of X and Y variables start with the index 1. Structures are coded by a list `(fun arity)`. The list structures are coded as nestings of the structure `(cns car cdr)` on the classified clauses representation [illegible]

- (put_structure F X_{to})
- (put_list X_{to})

4.4.2 GET-instructions

- (get_variable_temp Xn Ai)
- (get_variable_perm Yn Ai)
- (get_value_temp Xn Ai)
- (get_value_perm Yn Ai)
- (get_nil Xi)
- (get_constant C Xi)
- (get_structure F Xi)
- (get_list Xi)

4.4.3 UNIFY-instructions

- (unify_variable_temp X_i)
- (unify_variable_perm Y_i)
- (unify_void n)
- (unify_value_temp X_i)
- (unify_value_perm Y_i)
- (unify_local_value_temp X_i)
- (unify_local_value_perm Y_i)
- (unify_nil)
- (unify_constant C)

4.4.4 Indexing instructions

- (switch_on_type Lvarunbound Linteger Lsymbol Llist Lstruct Lnil Lother)
- (switch_on_constant Len Table Default)
- (switch_on_structure Len Table Default)

4.4.5 Procedural instructions

- (try L n)
- (retry L n)
- (trust L n)
- (try_me_else L n)
- (retry_me_else L n)
- (trust_me_else_fail n)
- (allocate n)
- (deallocate)
- (proceed)
- (execute proc/n)
- (call proc/n envsize)

4.4.6 Special instructions

- (has-succeeded)
- (has-failed)

4.4.7 Special builtins - Cuts and Metacall

- (save_cut_pointer)
  This instruction must be generated if there is a cut occurring in the clause except in the first chunk. This implies that there is more than one chunk and an environment must be existent.
- (first_cut)
  This instruction is used when the cut is in the first chunk and the first chunk is no pseudochunk. It contains a call to another procedure and thus is not the only subgoal in the clause.
- (lonely_cut)
  This instruction stands for a clause with a cut at the end of the first and only chunk. (So a call to another procedure is not present.)
- (last_cut)
  last_cut is to be used in a clause, which has a chunk (and hence a call to a procedure) and a cut at the very end of the last (pseudo)-chunk.
- (cut n)
  This instruction represents a cut occurring in a chunk except the first and the last chunk. The parameter n indicates the size of the environment used (for trimming).

- (mcall X_i)
  This a metacall where X_i references a structure (not a list!) representing the call to be invoked.

4.4.8 LISP interface

Only ground arguments (not variables) can be converted to LISP. The LISP functions are not allowed to return structures (nor variables). All NyWAM-LISP interface instructions convert *arity* argument registers into a LISP list and apply the function *fun* to this list. Only **RELFUN** tups - but not structures - can be converted.

- (cl-func fun arity)
  This function returns the value obtained from LISP to the argument register X1.

- (cl-pred fun arity)
  This instruction generates a failure if the returned value is `nil`. [2]

- (cl-extra fun arity)
  This instruction is used for side-effect LISP calls. [3]

4.5 User interface of the NyWAM

The user may define a procedure using the `definstr` macro. Queries are dynamically compiled by flattening, classifying and generating code for a procedure named '`main/arity`'. The arity of this procedure is determined by the number of variables originally found in the user query.

4.5.1 The debugger control commands

The debugging behavior of the NyWAM can be controlled by the variable `*emu-debug*`, which is normally set to `nil` to just run through the WAM code. If the user wishes to have WAM debugging information, this global variable may be set to `t` by the RFE-command `spy`.

```
All control commands consist of one character.

E,e             Terminate and go to LISP.
F,f             Generate a fail. (Sometimes this command may
                cause trouble.)
?               Output this Help-Menu.
X,x             Execute until program succeeds.
S,s,newline     Single step execution.
V,v             Output values before single step.
```

[2]In the interpreter a `false` is produced, which generates a failure if used as a body premise.
[3]X1 will not be changed.

4.5.2 The debugger display commands

This mode will be enabled by typing `v` in the control mode.

```
All display commands consist of one character.

?              Output this Help-Menu.
X,x            Output n (to be read) argumentregisters X(1)..X(n).
H,h            Output Heap.
R,r            Output all registers except argumentregisters.
S,s            Output stack.
```

5 A sample session

We consult and compile the well known naive reverse benchmark, run an nrev-query and then demonstrate the usage of the debugger using a simple append-query.

```
rfi> emul
rfe> consult "exa/bench"
rfe> l
(hn (app nil _l _l))
(ft (app (cns _h _l1) _l2 (cns _h _l3))
    (app _l1 _l2 _l3) )
(hn (nrev nil nil))
(ft (nrev (cns _h _l1) _l3)
    (nrev _l1 _l2)
    (app _l2 '(cns _h nil) _l3) )
```

The database has been consulted and listed. In the following we do some horizontal transformations and list the result.

```
rfe> horizon
rfe> l
(hn (app nil _l _l))
(ft (app _1 _l2 _2)
    (is _2 '(cns _h _l3))
    (is _1 '(cns _h _l1))
    (app _l1 _l2 _l3) )
(hn (nrev nil nil))
(ft (nrev _1 _l3)
    (is _1 '(cns _h _l1))
    (nrev _l1 _l2)
    (is _2 '(cns _h nil))
    (app _l2 _2 _l3) )
```

The horizontal transformations are followed by the vertical transformations into WAM code. The resulting code is shown by the `listcode` command. If you want to see classified clauses, type `listclass`.

```
rfe> verti
rfe> listcode app/3
((try_me_else 0 3)
 (get_nil 1)
 (get_value_temp 2 3)
 (put_constant true 1)
 (proceed)
 0
 (trust_me_else_fail 3)
 (get_list 3)
```

```
 (unify_variable_temp 4)
 (unify_variable_temp 5)
 (get_list 1)
 (unify_value_temp 4)
 (unify_variable_temp 6)
 (put_value_temp 6 1)
 (put_value_temp 5 3)
 (execute app/3) )

rfe> listcode nrev/2
((try_me_else 0 2)
 (get_nil 1)
 (get_nil 2)
 (put_constant true 1)
 (proceed)
 0
 (trust_me_else_fail 2)
 (allocate 3)
 (get_variable_perm 3 2)
 (get_list 1)
 (unify_variable_perm 2)
 (unify_variable_temp 3)
 (put_variable_perm 1 2)
 (put_value_temp 3 1)
 (call nrev/2 3)
 (put_list 2)
 (unify_value_perm 2)
 (unify_nil)
 (put_unsafe_value_perm 1 1)
 (put_value_perm 3 3)
 (deallocate)
 (execute app/3) )
```

We are now finished compiling the database. Next we perform an nrev-query.

```
rfe> (nrev '(cns 1 (cns 2 (cns 3 nil))) _x)
true
(_x = (tup 3 2 1))

More solutions?  (y or n) y

unknown
```

Now we are interested in obtaining a trace of a simple query, displaying the internal structures when something interesting happens. The query is compiled and then the debugger is invoked.

```
rfe> spy
rfe> (app '(cns 1 nil) '(cns 2 nil) _x)
((proc
  main/1
  1
  (funleva
   nil
   ((_x (3 (1) (3))) (_1 (4 nil (1))) (_2 (2 nil (2))))
   (chunk
    ((usrlit (main (_x (first safe temp))) (1 0 (1)))
     (is (_1 (first unsafe temp)) '(cns 1 nil))
     (is (_2 (first unsafe temp)) '(cns 2 nil))
     (usrlit
      (app
       (_1 (nonfirst unsafe temp))
       (_2 (nonfirst unsafe temp))
       (_x (nonfirst safe temp)) )
      (3 0 (1 3)) ) )
    (4 nil) ) ) ))
((get_variable_temp 3 1) (put_list 4) (unify_constant 1) (unify_ nil)
 (put_list 2) (unify_constant 2) (unify_nil) (put_value_temp 4 1)
 (execute app/3))
```

The following is a debugger trace.

```
 P = [code  :    (TRY PROC 0) in TOP-LEVEL]  :v
 Value of?s
mem[20000]=[???   :          ??????]  <== E <== B
```

Initially there is not much on the stack. Registers E and B point to the beginning of the stack. The next instruction creates a choicepoint and the registers are set appropriately.

[illegible]

This is the standard choicepoint which is responsible for the output of unknown/success messages, having the next clause entry pointing to code causing the output of the user's variables.

```
 P = [code  :    (try proc 0) in top-level]  :s

 P = [code  :            proc in top-level]  :v
Value of?s
mem[20000]=[???   :          ??????]  <== E
mem[20001]=[ref   :           20000]
mem[20002]=[code  : (has-succeeded) in top-level]
mem[20003]=[ref   :           20000]
mem[20004]=[code  :  (trust fail 0) in top-level]
mem[20005]=[trail :             nil]
mem[20006]=[ref   :               1]  <== B
```

```
P = [code  :              PROC in TOP-LEVEL]  :s

P = [code  : (CALL MAIN/1 0) in TOP-LEVEL]  :s

P = [code  :(GET_VARIABLE_TEMP 3 1) in MAIN/1]  :s

P = [code  :    (PUT_LIST 4) in MAIN/1]  :s

P = [code  :(UNIFY_CONSTANT 1) in MAIN/1]  :s

P = [code  :     (UNIFY_NIL) in MAIN/1]  :s

P = [code  :    (PUT_LIST 2) in MAIN/1]  :s

P = [code  :(UNIFY_CONSTANT 2) in MAIN/1]  :s

P = [code  :     (UNIFY_NIL) in MAIN/1]  :s

P = [code  :(PUT_VALUE_TEMP 4 1) in MAIN/1]  :s

P = [code  : (EXECUTE APP/3) in MAIN/1]  :v
Value of?a
Type number of argumentregisters to output:3

A(1 ) = [list  :            2]
A(2 ) = [list  :            4]
A(3 ) = [ref   :            1]
P = [code  : (EXECUTE APP/3) in MAIN/1]  :v
Value of?h

mem[  0]=[???   :        ??????]  <== S
mem[  1]=[ref   :             1]  <== HB
mem[  2]=[const :             1]
mem[  3]=[const :           NIL]
mem[  4]=[const :             2]
mem[  5]=[const :           NIL]  <== H
P = [code  : (EXECUTE APP/3) in MAIN/1]  :s
```

The code above allocates the structures for the query in the data space and sets the argument registers accordingly. Register X1 points to a list at memory locations 2 and 3, representing the list `(1 . nil)`, and register X2 points to the list at memory locations 4 and 5. The third argument (X3) is a reference to memory location 1, whose contents points to the same location. This is the representation of a free variable.

```
P = [code  :(TRY_ME_ELSE 0 3) in APP/3]  :s

P = [code  :     (GET_NIL 1) in APP/3]  :s
```

Please note that the get_nil fails and jumps to label 0 in app/3 to continue. The choice-point can now be removed since the next clause is the last clause in the procedure. So **trust_me_else_fail** removes the choicepoint.

```
 P = [code  :              0 in APP/3]  :s

 P = [code  :(TRUST_ME_ELSE_FAIL 3) in APP/3]  :v
Value of?s

mem[20000]=[???   :          ??????]  <== E
mem[20001]=[ref   :           20000]
mem[20002]=[???   :          ??????]
mem[20003]=[ref   :           20000]
mem[20004]=[code  :  (TRUST FAIL 0) in TOP-LEVEL]
mem[20005]=[trail :             NIL]
mem[20006]=[ref   :               1]
mem[20007]=[ref   :               1]
mem[20008]=[list  :               4]
mem[20009]=[list  :               2]
mem[20010]=[ref   :           20000]
mem[20011]=[code  : (HAS-SUCCEEDED) in TOP-LEVEL]
mem[20012]=[ref   :           20006]
mem[20013]=[code  :               0 in APP/3]
mem[20014]=[trail :             NIL]
mem[20015]=[ref   :               5]  <== B
 P = [code  :(TRUST_ME_ELSE_FAIL 3) in APP/3]  :s

 P = [code  :    (GET_LIST 3) in APP/3]  :vs
Value of?
mem[20000]=[???   :          ??????]  <== E
mem[20001]=[ref   :           20000]
mem[20002]=[???   :          ??????]
mem[20003]=[ref   :           20000]
mem[20004]=[code  :  (TRUST FAIL 0) in TOP-LEVEL]
mem[20005]=[trail :             NIL]
mem[20006]=[ref   :               1]  <== B
```

In the following the next procedure invocation of app/3 is prepared.

```
 P = [code  :    (GET_LIST 3) in APP/3]  :s

 P = [code  :(UNIFY_VARIABLE_TEMP 4) in APP/3]  :s

 P = [code  :(UNIFY_VARIABLE_TEMP 5) in APP/3]  :s

 P = [code  :    (GET_LIST 1) in APP/3]  :s
```

```
 P = [code  :(UNIFY_VALUE_TEMP 4) in APP/3]  :s

 P = [code  :(UNIFY_VARIABLE_TEMP 6) in APP/3]  :s

 P = [code  :(PUT_VALUE_TEMP 6 1) in APP/3]  :s

 P = [code  :(PUT_VALUE_TEMP 5 3) in APP/3]  :s

 P = [code  : (EXECUTE APP/3) in APP/3]  :v
Value of?a
Type number of argumentregisters to output:3

 A(1 ) = [const :           NIL]
 A(2 ) = [list  :             4]
 A(3 ) = [ref   :             7]
 P = [code  : (EXECUTE APP/3) in APP/3]  :v
Value of?h

mem[  0]=[???   :        ??????]
mem[  1]=[list  :             6]
mem[  2]=[const :             1]
mem[  3]=[const :           NIL]  <== S
mem[  4]=[const :             2]
mem[  5]=[const :           NIL]  <== HB
mem[  6]=[const :             1]
mem[  7]=[ref   :             7]  <== H
 P = [code  : (EXECUTE APP/3) in APP/3]  :s
```

Now app/3 is called with the following arguments: X1 is nil, X2 is (2.nil) and X3 is a free variable. Clearly, the first clause of app/3 must be applied.

```
 P = [code  :(TRY_ME_ELSE 0 3) in APP/3]  :s

 P = [code  :     (GET_NIL 1) in APP/3]  :s

 P = [code  :(GET_VALUE_TEMP 2 3) in APP/3]  :s

 P = [code  :(PUT_CONSTANT TRUE 1) in APP/3]  :s

 P = [code  :       (PROCEED) in APP/3]  :s

 P = [code  : (HAS-SUCCEEDED) in TOP-LEVEL]  :v
Value of?s

mem[20000]=[???   :        ??????]  <== E
mem[20001]=[ref   :         20000]
```

```
mem[20002]=[???   :          ??????]
mem[20003]=[ref   :           20000]
mem[20004]=[code  :  (TRUST FAIL 0) in TOP-LEVEL]
mem[20005]=[trail :             NIL]
mem[20006]=[ref   :               1]
mem[20007]=[ref   :               7]
mem[20008]=[list  :               4]
mem[20009]=[const :             NIL]
mem[20010]=[ref   :           20000]
mem[20011]=[code  : (HAS-SUCCEEDED) in TOP-LEVEL]
mem[20012]=[ref   :           20006]
mem[20013]=[code  :               0 in APP/3]
mem[20014]=[trail :[ref   :              1]]
mem[20015]=[ref   :               7]  <== B
 P = [code  : (HAS-SUCCEEDED) in TOP-LEVEL1]  :s
true

(_x = (tup 1 2))

More solutions?   (Y or N) y
```

In the following some other possibilities are tested, but fail. Finally the **unknown** message is generated due to the failure pointer in the very first choicepoint entry.

```
 P = [code  :              0 in APP/3]  :s

 P = [code  :(TRUST_ME_ELSE_FAIL 3) in APP/3]  :s

 P = [code  :    (GET_LIST 3) in app/3]  :s

 P = [code  :(UNIFY_VARIABLE_TEMP 4) in APP/3]  :s

 P = [code  :(UNIFY_VARIABLE_TEMP 5) in APP/3]  :s

 P = [code  :    (GET_LIST 1) in app/3]  :s

 P = [code  :  (TRUST FAIL 0) in TOP-LEVEL]  :s

 P = [code  :            FAIL in TOP-LEVEL]  :s

 P = [code  :    (HAS-FAILED) in TOP-LEVEL]  :s
 unknown
rfe> lisp
```

References

[BEH+93] Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, and Werner Stein. RELFUN Guide: Programming with Relations and Functions Made Easy. Document D-93-12, DFKI GmbH, July 1993.

[Bol90] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.

[Els90] Klaus Elsbernd. Effizienzvergleiche zwischen einer LISP- und C-codierten WAM. SEKI Working Paper SWP-90-03, Universität Kaiserslautern, Fachbereich Informatik, Juni 1990.

[Hei89] Hans-Günther Hein. Adding WAM-Instructions to support Valued Clauses for the Relational/Functional Language RELFUN. SEKI Working Paper SWP-90-02, Universität Kaiserslautern, Fachbereich Informatik, December 1989.

[Kra90] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. SEKI Working Paper SWP-90-04, Universität Kaiserslautern, Fachbereich Informatik, Mai 1990.

[Kra91] Thomas Krause. Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN. Diplomarbeit, DFKI D-91-08, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, März 1991.

[Nys] Sven Olof Nystrøm. Nywam - a WAM emulator written in LISP.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
67608 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-35
Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to H-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster:
WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi:
Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-53
Werner Stephan, Susanne Biundo:
A New Logical Framework for Deductive Planning
15 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
13 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
19 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist:
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi:

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Juergen Buerckert:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen:
Equational and Membership Constraints for Infinite

RR-93-17
Rolf Backofen:
Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder:
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
Manfred Meyer, Jörg Müller:
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy:
Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel:
Document Highlighting —
Message Classification in Printed Business Letters
17 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger:
Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

DFKI Technical Memos

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh:
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference

DFKI Documents

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.): Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann: Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01
Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

D-93-03
Stephan Busemann, Karin Harbusch(Eds.): DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1992
194 Seiten

D-93-05
Elisabeth André, Winfried Graf, Jochen Heinsohn, Bernhard Nebel, Hans-Jürgen Profitlich, Thomas Rist, Wolfgang Wahlster: PPP: Personalized Plan-Based Presenter
70 pages

D-93-06
Jürgen Müller (Hrsg.): Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-07
Klaus-Peter Gores, Rainer Bleisinger: Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch: Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

D-93-09
Hans-Ulrich Krieger, Ulrich Schäfer: TDL ExtraLight User's Guide
35 pages

D-93-12
Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, Werner Stein: RELFUN Guide: Programming with Relations and Functions Made Easy
86 pages

D-91-03
Document

RFM Manual: Compiling RELFUN into the Relational/Functional Machine
(Second, Revised Edition)

Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause