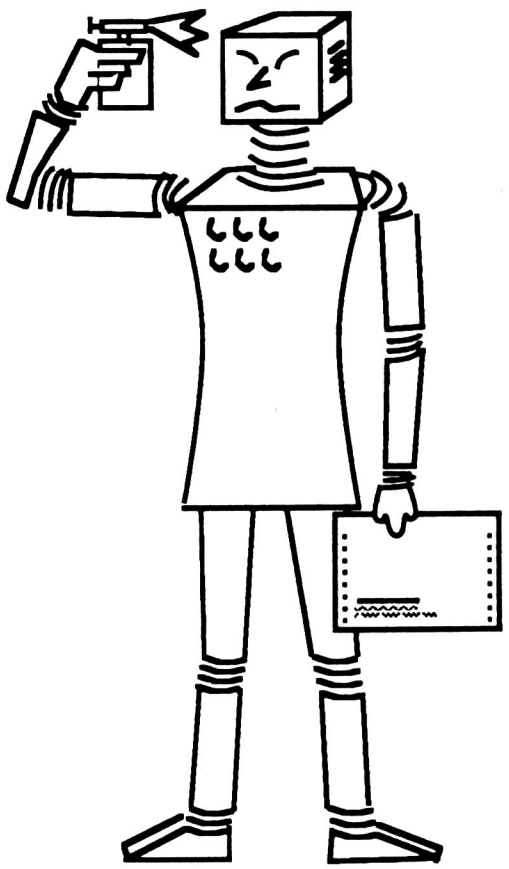
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT



Adding WAM-instructions to support Valued Clauses for the Relational/Funcional Language RELFUN

HAns-Günther Hein

SEKI Working Paper SWP-90-02

Adding WAM instructions to support Valued Clauses for the Relational/Functional Integration Language RELFUN

Hans-Günther HEIN

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1
West Germany

Projektarbeit
(Betreuer: Dr. Harold Boley)

December 1989

Abstract:
The integration of functional and relational programming languages is mainly based on interpreter systems. This work shows how to add instructions to the abstract PROLOG machine WAM, introduced by David H. D. Warren [11], for compiling the integration language RELFUN. In the relational view of computation the binding of variables is one of the basic concepts, whereas the functional view is based on the returning of values. RELFUN is built around valued clauses, i.e. the last call of a clause determines the value of the clause.
In this paper "VALREG" instructions are introduced to support valued clauses in a WAM based machine, developing a proposal originally made by Harold Boley [2] for the Prolog machine model (WPE) of David Maier and David S. Warren [5]. It is shown that these instructions are integrated easily into the existing WAM model and that the compilation of RELFUN programs is straightforward. The added instructions were implemented in LISP for the Sven-Olof Nystroem based WAM model ([12]) and for a WAM extension, the Joachim Beer based model ([9],[10],[13]).

Acknowledgements
I have to thank Harold Boley for papers, discussions and his support, Michael Tepp, who helped me to handle the Beer/Bürckert/Tepp WAM compiler and emulator, Ralph Scheubrein and Thomas Krause, who added the extensions to the Beer/Bürckert/Tepp compiler and Sven-Olof Nystroem, who emailed his WAM emulator to us. Moreover, I want to thank Prof. Dr. M.M. Richter for providing an exciting AI research environment for logic programming in Kaiserslautern.

1. Introduction

RELFUN is a declarative programming language trying to combine the logic programming style (as in PROLOG) and functional programming style (as in LISP). PROLOG users perform their computations by binding variables, whereas a LISP user formulates his algorithm by defining a function returning the computed value.
The RELFUN approach permits programming in a functional and logic (relational) style by the concept of value returning clauses. The definitional interpreter for RELFUN has been implemented by Harold Boley[1]. For efficient evaluation he set up the RFM (Relational/Functional Machine) project based on compilers and low level machine models (emulators).
This work presents the first emulator implementation of the RFM project.
The Warren Abstract Machine (WAM) is the standard for Prolog compiler technology and it has been extended to support efficiently the usage of value returning clauses.

RELFUN's original syntax is LISP-based, because the interpreter has been implemented in LISP and the reader of LISP makes a scanner and parser unnecessary. In [3] a Prolog-based RELFUN syntax was introduced. Since the development of the valued clause WAM additions started with a PROLOG compiler, both PROLOG-like syntax (with a built-in operator to indicate valued clauses) and LISP-like syntax will be used in the following.

1.1 Basics of RELFUN

A valued clause can be written as a LISP list containing at least one operator application:

```
( (p1 ..) (p2 ..) .. (pn ..) )
```

The first element of this list is the head of the clause, the rest (cdr) of the list is the body representing the premises of the clause.

Example (nullary operators):

```
PROLOG-like syntax:  p :- q,r.
LISP-like syntax:    ( (p) (q) (r) )
```

A fact can be written as a list with only one element.

```
PROLOG-like syntax:  p.
LISP-like syntax:    ( (p) )
```

An operator application is a non-nil list. The first element of the list usually is a LISP atom denoting the operator name, the rest of the list are the arguments of the operator application.

The arguments may be atoms, variables, lists or backquoted lists. Backquoted LISP lists represent RELFUN lists or structures (as both known from PROLOG). Normal lists are interpreted as embedded operator calls to other value returning clauses and constitute one of the extensions to PROLOG. We speak of passive (backquoted) and active (normal) lists. The value of a clause is determined by the last operator application in a clause. A clause may return a value explicitly by a backquote operator.

A variable is prefixed with an underscore. The primitive "is" is extended to arbitrary right hand sides, whereas in PROLOG it has mainly arithmetic character.
RELFUN operator calls containing embedded calls are flattened syntactically. The effect can be compared intuitively with the call-by-value function evaluation in LISP.

Since there is no difference between structures and lists in RELFUN (yet), there is no one-to-one mapping between the LISP-like syntax of RELFUN and the syntax of PROLOG enhanced with value returning clauses.

Pure RELFUN employs two major primitives:

The built-in "inst" or "backquote" operator instantiates its argument. The built-in "is" operator unifies its first argument with its evaluated second argument. The description in 1.1 will now be clarified by a number of examples.

1.2 RELFUN in 8 examples

```
1)      LISP-like syntax:            ((likes john mary))
        Remark: Value true
        PROLOG-like syntax:      likes(john,mary).
                If someone asks: ?- (likes _X _Y)        one gets:
                        -> _X = john
                           _Y = mary
                           Value = TRUE

2)      LISP-like syntax:      ((likes john _x) (likes _x wine))
        Remark: Value computed by (likes _x wine)
    (+) PROLOG-like syntax:      likes(john,X) :- likes(X,wine).
        Add the fact:          ((likes gaby wine))
        If someone asks: ?- (likes john gaby)   one gets:
                        -> Value = TRUE

3)      Fac(0) returns 1
        LISP-like syntax:        ((fac 0) 1)
        PROLOG-like syntax:      fac(0) :- backquote(1).

4)      Fac(N) is calculated by N times the fac of N-1.
        LISP-like syntax: ((fac _n)(is _m (fac (- _n 1)))(* _m _n))
        PROLOG-like syntax: fac(N) :- is(M,fac(minus(N,1))),
                                      times(N,M).
        The flattening process results in  (_1 is a variable) :
                                fac(N) :- is(_1,minus(N,1)),
                                          is(M,fac(_1)),
                                          times(N,M).

        If someone asks: ?- (fac 3)              one gets:
                        -> Value = 6

5)      Consider the rule: ((likes gaby wine) `always )
                           and the rule (+)
                      likes(gaby,wine) :- backquote(always).
        If someone asks: ?- (likes john gaby)   one gets:
                        -> Value = always
```

6) Fred is older than mary, return the structure (years 5)

```
LISP-like syntax:    ((older fred mary) `(years 5))
PROLOG-like syntax: older(fred,mary) :-
                      backquote([years,5]).
                    or: older(fred,mary) :-
                      backquote(years(5)).
If someone asks: ?- (older fred mary)   one gets:
                 -> Value = (years 5)
```

7) A is older than B, if the age of A > the age of B.
Return difference in a dimensional list.

```
LISP-like syntax:       ((older _A _B)
                         (> (age _A) (age _B))
                         (list years (- (age _A) (age _B)))
                         )
PROLOG-like syntax: older(A,B) :-
                      greater(age(A),age(B)),
                      list(years,minus(age(A),age(B))).
The flattening process results in:
(_1,_2,_3,...  are variables)
                    older(A,B) :- is(_1,age(A)),
                                  is(_2,age(B)),
                                  greater(_1,_2),
                                  is(_4,age(A)),
                                  is(_5,age(B)),
                                  is(_3,minus(_4,_5)),
                                  list(years,_5).
Consider the following rules:
          (*)      ((age fred) 45)
                   age(fred) :- backquote(45).
                   ((age mary) 40)
                   age(mary) :- backquote(40).
If someone asks: ?- (older _X _Y)       one gets:
                 -> _X = fred
                    _Y = mary
                    Value = [years,5]
```

8) A is older than B if the age of A > the age of B, return a list with the two ages.

```
LISP-like syntax:       ((older _A _B)
                         (> (age _A) (age _B))
                          `(exceeds (age  _A) (age _B)))
                         )
PROLOG-like syntax: older(A,B) :-
                      greater(age(A),age(B)),
                      backquote(exceeds(age(A),age(B))).
If someone asks with the rules (*) above:
                  ?- (older _X _Y)       one gets:
                 -> _X = fred
                    _Y = mary
                    Value = (exceeds (age 45) (age 40))

If someone asks with the rules (*) above:
 ?- ( is (exceeds (age 45) (age _x)) (older fred _o) )
    -> _X = 40       ( Remark: LISP converts lowercase to
       _O = mary       uppercase: _o --> _O )
    Value = (exceeds (age 45) (age 40))
```

Note that the terms "structure" and "list" have been used almost interchangeably when discussing both syntaxes in parallel.

2. The Nystroem based WAM Model "NyWAM"

The design used in the NyWAM was influenced directly by David Warren's model (1983) of the abstract WAM machine [12]. The NyWAM is implemented in LISP. The primary goal of this implementation is the possibility of easy modification. It was written by Sven-Olof Nystroem, Uppsala University[12].

2.1 The main data structures

It is assumed that the reader is familiar with the memory managing techniques in WAMs as described in [5] and [11].
"Global Stack" and "Heap" as well as "Local Stack" and "Run Time Stack" are synonyms, respectively, the environment and the choice points are a portion of the Local Stack.

Opposed to other implementations, the NyWAM has a divided memory layout: Normally the code area, the trail, the global and local stack are assumed to be in a common memory, starting at different addresses.
In the NyWAM a different approach is used: The code area is split up in procedures which are stored on the property list under a LISP identifier constructed from the predicate name and the predicate arity. No label resolving ("linking") is done, so a reference to another procedure is not represented by a numeric address, but by its symbolic name and its arity, resulting in very readable and modifyable WAM Code.
The Local Stack and the Heap share the same tagged memory starting at different locations. The memory is an array of defstruct elements "word", which consist of a tag and a value field. This sharing eases the lookup of references: Environment locations may point into the heap and these references need not be handled by an extra routine.

The following tags of words are possible:

Tag	Meaning
'empty	undefined.
'ref	a pointer to a memory address.
'struct	a pointer to a memory address where a structure starts.
'list	a pointer to a memory address where a list starts.
'const	the value of the constant (must be an atom).
'fun	a list (functor arity), used as the first word in a structure.
'code	a list (procedure-name/arity . instruction-list) .
'trail	a list of references to bound variables.

The tags are stored symbolically.

The trail list represents a stack of variables (memory locations) which are to be unbound down to a certain entry when a failure occurs and backtracking must be done.
Unbound variables are represented by a reference (ref) to themselve.

The following registers are used:

```
(*)     P       program counter (points to executable code)
(*)     CP      continuation pointer (points to executable code)
(*)     E       last environment (points to local stack)
(*)     B       last choice point (backtrack point)
                (points to local stack)
(*)     A       top of local stack
(*)     TR      top of trail (TR trail is a list,
                the CAR is the top)
(*)     H       top of heap
(*)     HB      heap backtrack point
(*)     S       structure pointer to the heap
(+)     A1,A2.. argument registers
(+)     X1,X2.. temporary registers equivalent to A1,A2,..
        Y1,Y2.. permanent variable locations in environment
```

(*) = defined by define-register, Value on propertylist attribute, reg-value, represented by structure WORD
(+) = defined by the array "argument-regs"

2.2. Environments

An environment is created if the procedure needs to have local variables. An environment has the following structure:

```
                FIELD                          OFFSET
g    ¦      previous environment                 CE     <-- new E
r    ¦      continuation pointer                 CP
o    ¦      Y-variable 1                         (+ Y 0)
w    ¦        ...
i    ¦      Y-variable n                         (+ Y (1- n))
n    ¦                                           <--------- new A
g    ¦
     ¦
     ¦
A    v
```

2.3. Choice points

A choice point is needed if there is more than one clause for a goal. If a recent clause fails, the next clause is to be explored with all the argument registers appropriately set and the variables bound later than the invocation of the current clause restored (so they are unbound).

```
                FIELD                          OFFSET
g    ¦      A-Argument n                         (- A (1- n))
r    ¦      ...
o    ¦      A-Argument 1                         (- A 0)
w    ¦      environment                          BCE
i    ¦      continuation pointer                 BCP (code pointer)
n    ¦      previous choice point                B1
g    ¦      next clause                          BP
     ¦      trail point                          TR1
A    v      heap point                           H   <- new B
                                                 <----- new A
```

3. The Beer based WAM Model BBT (Beer/Bürckert/Tepp)

The design used in the Beer based WAM [13] is an enhencement of the machine described in [11]. Beer's specification additionally supports efficiently the compilation and code-generation for the cut-operator. It was implemented in LISP by Hans-Jürgen Bürckert, Michael Tepp et al. and extended to support many-sorted resolution [8], iterative deepening and the occur-check. Since their WAM was aimed at physical realization, their LISP implementation has been on a very low "machine level" optimized for speed (vast usage of macros and arrays). Furthermore Michael Forster has implemented a compiler for a subset of that model, (cut, lists, iterative deepening and many-sorted resolution have been left out).

3.1 The main data structures

In the following the term WAM refers to the BBT model.
The WAM instructions of a "program" are generated by the compiler and stored in a file. The WAM-emulator reads the instructions into the array 'code' resolving any symbolic label to the appropriate array index. Execution is an evaluation of the code array entry, index by the program counter 'wpm-ds*pc'.
The A/X-Registers are represented by the array 'wpm-ds*reg', where the tag of register i is stored at location 2*i and its value is stored at location 2*i+1.
The heap and the environment/choice point structures are held in the array 'wpm-ds*memory', pointed to by 'wpm-ds*gpos' (heap) and 'wpm-ds*lpos' (local stack).
The trail uses the extra stack 'wpm-ds*trail-stack' and the top of stack is referenced by 'wpm-ds*tpos'.

The following tags are possible:

Tag sym.	Code	Meaning
unbound	0	unbound variable. The value field should point to itself unless it is not a normal PROLOG variable. A sorted variable points to its sort.
const	1	the value is a constant.
structure	2	the value of the cell is a list (f arity) Remark: the functor / arity information is not split to different memory cells.
list	3	the value of the cell is a pointer to the car element.
ref	4	the value of the cell is a pointer to another object. Should be dereferenced to obtain its value.
s-structure	5	this indicates a pointer to a sorted structure in the heap (sort and structurename).

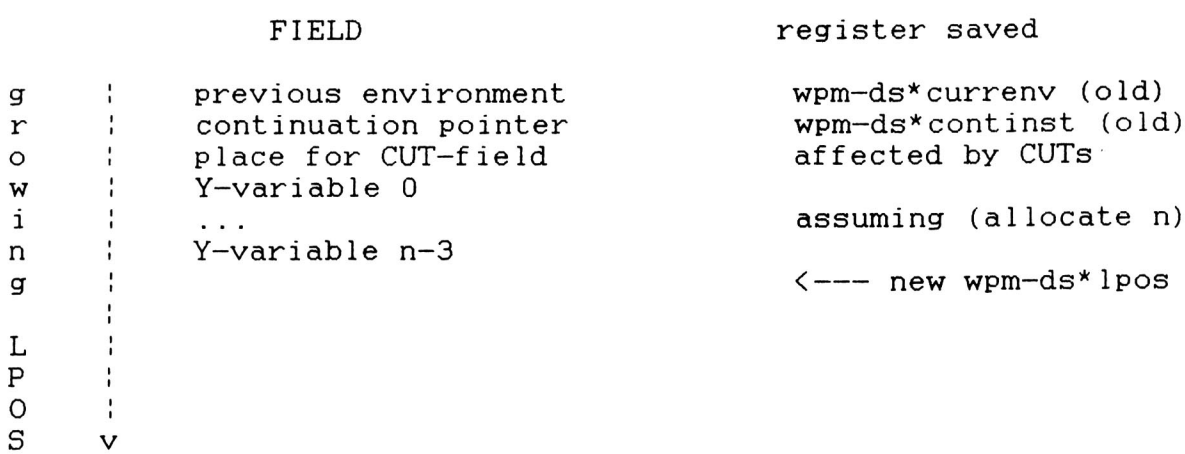
Code and trail need no extra tag since they use different arrays.

The following registers are used:
(The terminology of [5] and [11] have been added).

wpm-ds*pc	Program counter. IC [5], P [11]
wpm-ds*continst	If goal succeeded, control goes to this address. Resembles Return Address of Subroutine calls in conventional languages. CP [11], RETREG[5]
wpm-ds*currenv	Points to the base of the current environment. E [9], ENVREG [5]
wpm-ds*last_chpnt	Pointer to the current choicepoint. B [11], BREG [5]
wpm-ds*cut_pointer	Pointer to the last choice point of the parent goal.
wpm-ds*old_gpos	Top of the heap when the choicepoint was created. Used for efficient trailing. HB [11],([5] uses CTRREG in choice point)
wpm-ds*lpos	A [11]
wpm-ds*gpos	HREG [5] , H [11]
wpm-ds*tpos	TRREG [5], TR [11] (list!)
wpm-ds*nextclause	Codeaddress to the next alternative, representing the next OR-branch, i.e. the clause to be tried next. NEXTCL [5] CP [11]
wpm-ds*nextarg	pointer into structured objects SREG [5] S-Register [11]
wpm-ds*reg	holds the Ai/Xi-Registers at (tag,value) = ([2*i],[2*i+1])

3.2. Environments

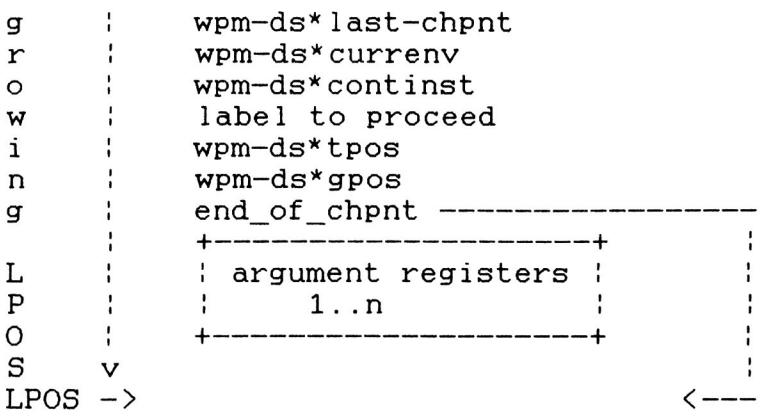
The allocation of an environment leaves the following structure on the local stack:



LPOS is an abreviation for wpm-ds*lpos

3.3 Choice points

The format of a choice point is:



4. Implementing Value-returning WAM-instructions

Within this paragraph the implementation of the WAM instructions will become obvious. A source transformation of RELFUN to normal PROLOG will be discussed. When using WAM instructions the NyWAM model version will be taken.

4.1 Source transformation of RELFUN to PROLOG

The differences between RELFUN and PROLOG are the value concept and the processing of active terms. These items and their "costs", expressed in memory usage and runtime behaviour, will be discussed in the following.

4.1.1 The flattening concept

Active terms may be seen as function calls, producing returned values, each "replacing" an active term. The solution for handling these active terms has been presented in [1], which is done by static or dynamic flattening. The active terms are processed, their results assigned to unique variables and the active calls are replaced by these unique variables. The result is **"flat"** RELFUN with no active terms in the calls. Without loss of generality flat RELFUN may be assumed as a basic representation. I will call the newly generated variables "flattening variables".

4.1.2 The cost of flattening variables

Flattening variables do occur in flat RELFUN in at least two premises, so these variables are used two times. So they have to be stored in the local environment as an Y-register.
The flattening-variables used are potentially unsave, since they do not occur in the head: Their first occurrence binds them to the value of an active term, which can point to a free variable in the current binding frame. The second and last time they are

referenced, they may be globalized onto the heap with a "put_y_unsafe_variable" instruction, consuming space and leaving work for the garbage collector.

Example:

```
( (f _r) (p _z) (g (h _z) _r) ) is flattened to:
remark: _1 is a flattening variable.
((f _r)
 (p _z)
 (is _1 (h _z))
 (g _1 _r)
)
((h _s) _s)
```

The procedure f/1 has 3 Y-variables: _r _z and _1.
_r cannot become unsafe, since it occurs in the head. Assume that p/1 does not bind _z. Then _1 is bound to _z by the h/1 procedure. So _1 has a reference to _z which is in the same environment. This environment will be trimmed before g/2 is called, so _1 must be made safe by moving it on the heap.

4.1.3 The value returning clauses concept

RELFUN clauses are valued, and the RELFUN to PROLOG conversion described will affect **all** clauses.
This is done by adding an extra argument to all predicates, which is a variable distinct from the other variables in the clause. The last premise is an assignment (actually a unification) to that variable. I will call these variables for the value binding "value returning variables".

Please note that I suggest instantiating the "result" variable at the end of the clause. If the instantiation was not done at the end, terms or lists may be (expensively) constructed on the heap, but one of the following calls may fail, resulting in instantiated value returning variables never used or worse, leaving garbage on the heap when constructing structures (runtime and memory waste).

Example:

1. a) `ari(X,RESULT) :- RESULT is t(t(t(Y))),p(X,Y).`
   assume p(..) fails and RESULT is uninstatiated.
   - the terms for the RESULT-variable are allocated on the heap although it is never used.

   b) `ari(X,RESULT) :- p(X,Y),RESULT is t(t(t(Y))).`
   space on the heap is only used if it is really needed.

Note that when using the second alternative, final literal optimisation is no longer possible.

4.1.4 The cost of the extra argument

A disadvantage of the transformation is the use of an extra argument register, which is a direct consequence of the added argument. If there is more than one clause in the procedure, it costs an extra memory location in the choice point for the argument register (see above) and extra time to restore the A-

register upon failure. Despite the fact that extra code is needed for every call preparation, the variable must be held in the Y-area when more than one premise is to be processed.

The extra argument Ai in the head of the clause is to be processed with a "get_variable_perm Yj,Ai" {1} instruction and the value is unified with a "get_value_perm Yj,Ak" {2} (Ak is the argument register, which references the value to be returned; Yj is the location, where the value returning variable is stored).

(The numbers in the brackets "{}" above reference the examples below.)

```
Example:              likes(mary,john) :- backquote(much).
is transformed to:    likes(john,mary,much).
Example:              likes(mary,_x) :- likes(_x,father(john)),
                                        backquote(john).
is transformed to:    likes(mary,_x,_1) :- father(john,_2),
                                           likes(_x,_2,_),
                                           _1 = john.
is compiled to:       likes/3   ;assuming deterministic procedure
                          allocate                3
                          get_constant            mary,A1
                          get_variable_perm       Y1,A2  ; _x
                          get_variable_perm       Y2,A3  ; _1 {1}
                          ; --- ---- ----         head processed
                          put_constant            john,A1
                          put_variable_perm       Y3,A2  ; _2
                          call                    3,father/2
                          put_value_perm          Y1,A1
                          put_unsafe_value_perm   Y3,A2
                          put_variable_temp       A3,A3
                          call                    2,likes/3
                          put_constant            john,A1
                          get_value_perm          Y2,A1 ; _1  {2}
                          deallocate
                          proceed
```

I will call potentially unsave Y-variables "U-Y-variables".
The flattening variable "_2" is a U-Y-variable.

4.1.5 Compilation of transformed RELFUN

In the following flat RELFUN is considered. We will proceed by examples.
Six cases will be distinguished. The separation of clauses with one and n premises is not obvious, but will be needed when the compiled form is discussed.

a) Facts
are true by definition. The added argument must be true.

```
Example transformation: likes(john,mary).
                  -->   likes(john,mary,true).

Generated code: likes/3 ; deterministic procedure assumed
                        get_constant        john,A1
                        get_constant        mary,A2
                        get_constant        true,A3
                        proceed
```

b) Denotative rules with one premise are rules returning a specified value. The added argument is the specified value. The value may be a variable (possibly anonymous), a constant, a list or a passive term.

```
Example transformation:  likes(john,mary) :- backquote(much).
              -->        likes(john,mary,much).

Costs: (*):extra argument (memory,time upon failure)

Generated code: likes/3 ; deterministic procedure assumed
                        get_constant        john,A1
                        get_constant        mary,A2
                        get_constant        much,A3
                        proceed
```

c) Evaluative rules with one premise
The added variable occurring in the head as an additional argument is passed to the premise.

```
Example transformation:  likes(john,X) :- likes(X,wine).
              -->        likes(john,X,R) :- likes(X,wine,R).

Costs: (*) argument
          moving the value variable if it is not in the same
          argument location in the head and the premise.
          this is the "cheapest" variable:
           it can be held in an A/X-Variable thus wasting no extra
           memory in the environment. (but possibly in the
           choice point if the procedure is not deterministic)

Generated code: likes/3  ; deterministic procedure assumed
                       get_constant        john,A1
                       get_variable_temp   A1,A2
                     ;  R need not be handled since place is ok
                       put_constant        wine,A2
                       execute             likes/3
```

d) Denotative rules with two premises

The added variable is unified by an "is" operation to the result of the clause. The result may be an (anonymous) variable, a constant, a list or a passive term.

```
Example tranformation:   likes(john,X) :- drinks(X,wine),
                                          backquote(X).
              -->        likes(john,X,R) :- drinks (X,wine,_),
                                          R is X.

Costs: (*) argument
           -result of the first premise must be possibly held
           in the heap since the result may not be needed;
           otherwise it had to be stored in a U-Y-variable.
```

```
Generated code: likes/3      ; assuming deterministic procedure
                        allocate              2
                        get_constant          john,A1
                        get_variable_perm     Y2,A2 ; X
                        get_variable_perm     Y1,A3 ; R
                        put_value_perm        Y2,A1 ; X
                        put_constant          wine,A2
                        put_variable_temp     A3,A3 ; heap var.(!)
                        call                  2,drinks/3
                        put_value_perm        Y2,A1
                        get_value_perm        Y1,A1
                        deallocate
                        proceed
```

e) Evaluative rules with n premises.

The added variable occurs in the head and in the last premise, since the last call determines the result.
Other value-returning premises are processed before and they may have anonymous variables, i.e. those generated by the value-returning concept (some of them may be held on the heap if their value is not needed).

```
Example transformation:  p(X) :- q(X,Y), r(X,Z).
                 -->     p(X,R) :- q(X,Y,_) , r(X,Z,R).
```

```
Costs: (*) argument
           -result of a premise may be held in the
           heap since the result may not be needed; (see above)
           -the result variable must be stored in a Y-register.
```

```
Generated code: p/2      ; assuming deterministic procedure
                        allocate              2
                        get_variable_perm     Y1,A1 ; X
                        get_variable_perm     Y2,A2 ; R
                        put_variable_temp     A2,A2 ; Y
                        put_variable_temp     A3,A3 ; _
                        call                  2,q/3
                        put_value_perm        Y1,A1 ; X
                        put_variable_temp     A2,A2 ; Z
                        put_value_perm        Y2,A3
                        deallocate
                        execute               r/3
```

f) Denotative rules with n premises

The added head variable is unified by an is operation in the last premise. Again, the other value returning calls may leave a result in their extended argument variable for later processing or it may be discarded.

```
Example   transformation:  p(X) :- q(X,Y), S is o(X,Y),
                                 backquote([f(S)]).
                remark: o(X,Y) is an active call and not a structure
                -->     p(X,R) :- q(X,Y,_), o(X,Y,S), R is [f(S)].
```

Costs: (*) argument
- every value returning variable not used must be allocated on the heap.
- every used value returning variable must be allocated in the environment.

```
Generated code: p/2       ; assuming deterministic procedure
                          allocate                4
                          get_variable_perm       Y4,A1 ; X
                          get_variable_perm       Y2,A2 ; R
                          put_variable_perm       Y3,A2 ; Y
                          put_variable_temp       A3,A3 ; _
                          call                    4,q/3
                          put_value_perm          Y4,A1 ; X
                          put_unsafe_value_perm   Y3,A2 ; Y
                          put_variable_perm       Y1,A3 ; S
                          call                    2,o/3
                          put_structure           f/1,A1
                          unify_value_perm        Y1
                          get_value_perm          Y2,A1
                          deallocate
                          proceed
```

Such transforming can be characterized as yielding a runtime and space expensive additional argument. Values not used are saved on the heap although they are never needed, wasting space and producing memory garbage. Value variables may also be allocated in the environment but may become potentially unsafe.
Flattening variables are even worse: they are potentially unsafe since they cannot occur in the head. In section 4.2 we will therefore proceed to another implementation technique.

4.2 Prolog Computation Model and Valued Clauses

WAM models such as WPE[5] or those considered here can be extended for valued clauses by introducing a register, VALREG, that holds returned values and adding put/get-like instructions that return values to VALREG and is unify values from it [2].
It has been argued in [2] that VALREG will always ultimately contain the RELFUN value of a main call, because it is overwritten by the rightmost subprocedure call and the successful backtrack branch. In the following I try to sketch the VALREG returning correctness on the basis of an extended PROLOG model.

A Prolog program represents an AND/OR graph, which is explored depth-first, left-to-right.

OR-Nodes can be found in a Prolog program in a procedure where the different clauses represent the OR-Node arcs.
In the WAM this information is statically stored in the procedure's code with the try/retry/trust instructions and dynamically, in the choice-points.

An AND-Node represents the set of premises of a clause together with the clause's (local) variables.
This information is to be found in the sequence of procedure calls in the WAM code and the instantiations in the environments.

A fact is a degenerated OR-node having the AND-Node TRUE as a successor. A query is an AND-Node which is the root of the AND/OR Tree.

Value returning terms are special forms of AND-nodes passing values to the upper OR-Nodes, especially from the facts, which are true, or denotative rules, which give back an arbitrary value. The **is primitive** unifies the local variables in the AND-node with VALREG.

Proceessing in the WAM model means that an AND-Node has been satisfied. So the AND-Node calculates either a value from its OR-"inputs" (a valued rule) or passes the value of the rightmost OR-tree-arc.

The value of an OR-Node is the value of the currently active AND-Node.

The value of an AND-Node is
a) TRUE, if it is a fact.
b) the backquote term, if it is a denotative rule.
c) the value of the last AND-node-subtree if it is an evaluative rule.

A global register, named VALREG, is sufficient for storing the value of a success branch in an AND/OR-tree.

Facts bind VALREG to TRUE, 1-premise denotative rules bind VALREG to the value specified, evaluative rules bind VALREG to the value bound to it by their rightmost premise subtree.

The value of a query is the value of the active AND-tree.
In the WAM-model the query is satisfied if a proceed to the query CALL is executed. Proceeds only happen in an AND-tree, so the value is either constructed in the AND-node or comes from the underlying OR-Node son, so it is correct to leave VALREG untouched. Since RELFUN's operational semantics specifies the rightmost son of the AND-tree as the query's value and the tree is explored from left to right, the rightmost son is the last to bind VALREG.

4.3 Addition of VALREG-affecting instructions and their usage.

Instructions are needed to construct the value of a clause, others are needed to implement the "is" primitive, basicly a unification between an object (variable,constant,list,term) referenced by VALREG and an object on the left hand side of the "is" primitive.

4.3.1 Data structures added to the WAM

To support valued clauses, a little modification is necessary to the register model. The VALREG register has to be added. In the NyWAM this is easily done by the generic definition function

```
(define-register VALREG)
```

building the register on the property list and adding the register to the list of data structures to be dumped when debugging. The other thing to be done was to define a front end for handling calls of the form "is <term> <active goal>".

In the BBT Model the VALREG register must be represented as an array holding 2 elements. The first element is the tag of the register and the second element is the actual value or pointer. Initialisation and debugging support had to be directly coded in the source, since no abstract constructors are supplied.

4.3.2 VALREG instructions returning values in clauses.

Facts have their value TRUE by defintion. Therefor VALREG must be bound to TRUE when a fact is about to succeed. Without any additional code, this can be achieved by a slightly modified version of "PROCEED", called "PROCEED_TRUE", which binds the global register VALREG to TRUE before returning. The semantics of PROCEED have not been changed. The only difference is in the compilation of normal facts. The assignment of a global register costs no memory access time and is to be considered very cheap.
In the following the added NyWAM and BBT-Wam instructions will be introduced together with some examples. The NyWAM instructions will be given first, followed by the BBT instructions. The difference of the names is for the sake of similarity with the instructions of the two underlying WAM machines.

If a clause returns a value, final literal optimisation is no longer possible, because after the last operator application the value to be returned must be moved to the VALREG register. This is similar to the method described in 4.1.3, where final literal optimisation is neither possible.

Valued clauses can have a variety of returned objects:

a) return a constant.

valreg_write_constant(c)
wpm-valreg_write_constant(c)

```
PROLOG-like syntax:          ... :- ..., backquote(c).
LISP-like syntax:               ((..) (..) .. `c) or
                             ((..) (..) .. c)
```

The clause returns a constant by binding VALREG to that constant.

valreg_write_nil()
wpm-valreg_write_nil()

```
PROLOG-like syntax:          ... :- ..., backquote(nil).
LISP-like syntax:               ((..) (..) .. `nil) or
                             ((..) (..) .. nil)
```

This is a special case of "valreg_write_constant" and it has been introduced mainly for symmetry reasons.

b) return an anonymous variable

valreg_write_variable_temp(x)
wpm-valreg_write_variable(x)

```
PROLOG-like syntax:          ... :- ..., backquote(X).
LISP-like syntax:            ((..) (..) .. `_x)  or
```

((..) (..) .. _x)

Assumimg X does not occur in the rest of the clause, the aim of this construct is to return an anonymous variable. This anonymous variable must be kept on the heap.

```
Example:  ((fatherof fred) _any)
          get_constant        fred,A1
          valreg_write_variable_temp
          proceed
```

c) return a variable which is in an X-register

valreg_write_value_temp(Xn)
wpm-valreg_write_x_value(Xn)

```
PROLOG-like syntax              ... :- ..., backquote(Xn).
LISP-like  syntax:              ((..) ... `_xn)     or
                                ((..) ... _xn)
```

Xn is a variable occurring in an X-register of a clause. This is an instruction which will be mainly used in denotational rules, where the return value can be found in an argument register of the clause.

```
Example:  ((add _x 0) _x)
          get_constant 0,A2
          valreg_write_value_temp A1
          proceed
```

d) return a variable which is in a Y-register and Y cannot be unsafe.

valreg_write_value_perm(Yn)
wpm-valreg_write_y_value (Yn)

```
Prolog like syntax:             ... :- ..., backquote(Yn).
LISP-like syntax:               ((..)(..) .. `_yn) or
                                ((..)(..) .. _yn)
```

Yn is a variable held in the Y-area of the local stack. Yn is said to be safe if that variable has been used in the head of the clause or it has been used on the heap (when constructing lists or structures).

e) return a variable which is in a Y-register which can be potentially unsafe.

valreg_write_value_unsafe_perm(Yn)
wpm-valreg_write_y_unsafe_value(Yn)

```
PROLOG-like syntax:       ... :- ..., backquote(Yn).
LISP-like syntax:         ((..) (..) ... `_yn)      or
                          ((..) (..) ... _yn)
```

Yn is a variable held in the Y-area of the local stack, but the variable may become unsafe. This may happen, if it is neither used in the head of the clause nor in a list or structure. Thus Yn must be "globalized" before the local environment is deallocated, otherwise there would be dangling references to variables released.

f) return a list.

valreg_write_list()
wpm-valreg_write_list()

```
PROLOG-like syntax:       ... :- ..., backquote([ ... ]).
LISP-like syntax:         ((..) (..) ... `( .. ))
```

The value to be returned is a list. It is stored on the heap (as usual) and the list can be constructed with the ordinary unify instructions.

Example: ((older john mary) `(age 3 4)) ; list return
older(john,mary) :- backquote([age,3,4]).
Please remember that in original RELFUN there is no difference between lists and structures.

```
older/2   get_constant        john,A1
          get_constant        mary,A2
          put_list            A3
          unify_constant      4
          unify_nil
          put_list            A4
          unify_constant      3
          unify_value_temp    A3
          valreg_write_list
          unify_constant      age
          unify_value_temp    A4
          proceed
```

g) return a structure
valreg_write_structure(f/arity)
wpm-valreg_write_structure(f/arity)
The value to be returned is a structure. It is stored on the heap and the arguments can be constructed with the ordinary unify instructions.

Example: ((older john mary) `(age 3 4)) ; structure(!)return
older(john,mary) :- backquote(age(3,4)).

Compiled version:

```
older/2   get_constant              john,A1
          get_constant              mary,A2
          valreg_write_structure    age/2
          unify_constant            3
          unify_constant            4
          proceed
```

4.3.3 VALREG instructions supporting the is-primitive

The is-primitive has the general structure "is <X1> <X2>", where <X1> is the passive part and <X2> is the active call of the is-primitive.
If an active call is to be unified with a structure/list, the active call is to be handled first, leaving its value in VALREG. Then VALREG must be processed with instructions unifying the structure/list.
The "is <variable> <active call>" is done in the following way: first the active call is processed leaving a returned object in the VALREG register. Then it must be unified with <variable> using the family of the read_valreg_<register> instructions, where <register> depends on the type (X or Y variable) and on the occurrence ("value","variable","unsafe").
Without loss of generality, deterministic procedures are assumed in the examples.
The following cases must be distinguished:

a) The value of the active term in VALREG is to be bound to an uninstantiated X-variable.
The Syntax is "<uninstatiated X-variable> is
<active term>"

valreg_read_variable_temp(Xn)
wpm-valreg_read_x_variable (Xn)

```
Prolog-like syntax:     s2(X) :- Y is succ(X), backquote(f(Y)).
LISP-like syntax:       ((s2 _X) (is _Y (succ _X)) `(f _Y))
could be used in the following manner:
              s2/2 call      0,succ/1
                   valreg_read_variable_temp X1
                   valreg_write_structure    f/1
                   unify_value_temp          X1
                   proceed
```

In the above example a code-optimization has been used which sees the variable _Y as a temporary variable. (There is only one "chunk" [14].)

b) The value of the active term in VALREG is to bound to an uninstantiated Y Variable in the local environment.
The Syntax is "<uninstantiated Y-Variable> is
<active term>"

valreg_read_variable_perm(Yn)
wpm-valreg_read_y_variable (Yn)

```
Prolog-like syntax:     s2(X) :- Y is s(X), q(Y), p(Y).
LISP-like syntax:       ((s2 _X) (is _Y (s_X)) (q _Y) (p _Y))

s2/2      allocate                  1
          call                      1,s/1
          valreg_read_variable_perm Y1
          put_value_perm            Y1,A1
          call                      1,q/1
          put_value_unsafe_perm     Y1,A1
          deallocate
          execute                   p/1
```

c) The value of the active term in VALREG is to be unified with an instantiated X-variable.
The Syntax is "<instantiated X-Variable> is
<active term>"

valreg_read_value_temp(Xn)
wpm-valreg_read_x_value (Xn)

This instruction can be used in optimized code-generation and if you have built-in predicates affecting X-register Xn to be bound to a certain value without touching other variables. (arithmetic, etc.)

d) The value of the active term in VALREG is to be unified with an instantiated Y-variable in the local environment. The variable is not occurring the last time or if it is the last time, it is not unsafe.
The Syntax is "<instantiated Y-Variable> is

<active term>"

valreg_read_value_perm(Yn)
wpm-valreg_read_y_value (Yn)

```
Prolog-like syntax:      s(X,Y) :- Y is succ(X).
LISP-like syntax:        ((s _X _Y) (is _Y (succ _X)) )

s/2       allocate                  1
          get_variable_perm         Y1,A2 ; _Y
          call                      1,succ/1
          valreg_read_value_perm    Y1
          deallocate
          proceed
```

e) The value of the active term in VALREG is to be unified with a constant. If the constant is nil, a special instruction is provided.

The Syntax is "<constant> is
<active term>"

valreg_read_constant(c)
wpm-valreg_read_constant (c)
valreg_read_nil()
wpm_valreg_read_nil()

```
Example:  ((odd _x) (is 1 (remainder x 2)) )

odd/1     put_constant          2,A2
          call                  0,remainder/2
          valreg_read_constant  1
          proceed
```

f) The value of the active term in VALREG is to be unified with a structure. The structure pointer must be set appropriately to allow the use of subsequent unify instructions.

The Syntax is "f(...) is <active term>"

valreg_read_structure(f)
wpm-valreg_read_structure(f)

```
Prolog-like syntax:      s(X) :- older(3,4) is age(X).
LISP-like syntax:        ((s _X) (is (older 3 4) (age _X)) )

s/1            call                      0,age/1
               valreg_read_structure     older/2
               unify_constant            3
               unify_constant            4
               proceed
```

g) The value of the active term in VALREG is to be unified with a list. The structure pointer must be set appropriately to allow the use of subsequent unify instructions.

The Syntax is "[...] is <active term>"

valreg_read_list()
wpm-valreg_read_list()

```
PROLOG-like syntax:    s(X) :- [3,4] is age(X).
LISP-like syntax:      ((s _X) (is `(3 4) (age _X)) )
remark:  (3 4) should now be interpreted as a list.

s/1             call                      0,age/1
                valreg_read_list
                unify_constant            3
                unify_variable_temp       A1
                get_list                  A1
                unify_constant            4
                unify_nil
                proceed
```

4.3.4 Compilation examples compared to transformed RELFUN

The examples in 4.1.5 will be compared to the VALREG-compiled form. "VALREG-Compilation methods" as introduced in 4.3.2 and 4.3.3 will be slightly better than "Transformational methods" as described in 4.1. In general, the VALREG-Compilation methods do not ommit the flattening variables, but a space optimisations can be applied when a flattening variable is unified with an operator application's value and directly used afterwards. So, it must not be stored in an U-Y-variable.

```
Example:          ... :- g(h(X)), ...
is flattened to:  ... :- _1 = h(X), g(_1), ...
compiled to:          ...
                      call h/1,n          ; Value of h/1 directly
                      valreg_write_value_temp X1 ; stored in
                      call g/1,m          ; argument place
                      ...
```

a) Facts are true by definition.
Deterministic procedure assumed; no extra time compared to conventional PROLOG is used when ommitting the time for setting VALREG to true. (This could be made parallel in a microcoded version !)

```
                      get_constant        john,A1
                      get_constant        mary,A2
                      proceed_true
```

b) 1-premise denotative rules are rules immediately returning a specified value. The added argument is the specified value. Since flat RELFUN is considered, the value may be a variable (anonymous), a constant, a list or a passive term.

```
likes(john,mary) :- backquote(much).
```

Costs: no extra memory in choice points and no time consumed upon failure and restore.

```
Generated code: likes/3 ; deterministic procedure assumed
                      get_constant          john,A1
                      get_constant          mary,A2
                      valreg_write_constant much
                      proceed
```

c) Evaluative rules with one premise.

Example transformation: likes(john,X) :- likes(X,wine).
Absolutely no difference in compiling with conventional PROLOG can be seen.

```
Generated code: likes/3  ; deterministic procedure assumed
                         get_constant          john,A1
                         get_variable_temp     A1,A2
                         put_constant          wine,A2
                         execute               likes/2
```

d) Denotative rules with two premises

```
Example:   likes(john,X) :- drinks(X,wine),
                                        backquote(X).
```

Costs:
Local Environment is smaller. The operator application "drinks" leaves a value in VALREG, but since nothing is done with it, it is discarded. This version leaves no heap variables !

```
Generated code: likes/3      ; assuming deterministic procedure
                         allocate              1
                         get_constant          john,A1
                         get_variable_perm     Y1,A2 ; X
                         put_value_perm        Y1,A1 ; X
                         put_constant          wine,A2
                         call                  1,drinks/2
                         valreg_write_perm     Y1
                         deallocate
                         proceed
```

e) Evaluative rules n premises.

```
  p(X) :- q(X,Y), r(X,Z).
```

Again, this version is a normal PROLOG clause, and the additions are totally free.

```
Generated code: p/2      ; assuming deterministic procedure
                         allocate              2
                         get_variable_perm     Y1,A1 ; X
                         put_variable_temp     A2,A2 ; Y
                         call                  2,q/2
                         put_value_perm        Y1,A1 ; X
                         put_variable_temp     A2,A2 ; Z
                         deallocate
                         execute               r/2
```

f) Denotative rules with n premises

```
 p(X) :- q(X,Y), S is o(X,Y),
                              backquote([f(S)]).
               remark: O(X,Y) is an active call and not a structure
               -->       p(X,R) :- q(X,Y,_) , o(X,Y,S), R is [f(S)].

Generated code: p/2      ; assuming deterministic procedure
                         allocate              3
                         get_variable_perm     Y1,A1 ; X
```

```
put_variable_perm     Y2,A2 ; Y
call                  2,q/2
put_value_perm        Y1,A1
put_value_perm        Y2,A2
call                  2,o/2
valreg_read_variable_temp X1
valreg_write_list
unify_structure       f/1
unify_value_temp      X1
deallocate
proceed
```

If the transformed version would have been used, 4 Y-variables would have been needed. When the result is to be bound to R, the chain into deeper environments must be followed. This chain can be rather long, depending on the depth of the computation. The longer the chain, the more space is wasted due to unnecessary links and the more runtime is consumed.

Generated code: p/2

```
; assuming deterministic procedure
allocate              4
get_variable_perm     Y4,A1 ; X
get_variable_perm     Y2,A2 ; R
put_variable_perm     Y3,A2 ; Y
put_variable_temp     A3,A3 ; _
call                  4,Q/3
put_value_perm        Y4,A1 ; X
put_unsafe_value_perm Y3,A2 ; Y
put_variable_perm     Y1,A3 ; S
call                  2,O/3
put_structure         f/1,A1
unify_value_perm      Y1    ; S
get_value_perm        Y2,A1 ; R
deallocate
proceed
```

- extra arguments cost memory and restorage time.

5. Conclusions

It was shown that the added instructions save time and memory for computations returning values. When evaluative rules are used as in RELFUN's pure PROLOG subset, no extra memory is necessary; the only overhead is in RELFUN's facts, which are to leave VALREG bound to TRUE. However, the time used to set a register (VALREG) to a constant (TRUE) incurs minimal overhead; it should be even non-measureable when the time-consuming unification algorithm comes into play. (Move-register-to-register and move-constant-to-register instructions are said to be the most quickest in a general purpose processor, and in a processor dedicated to Prolog execution they can be performed in parallel).

Value returning clauses are supported efficiently by the introduced instructions. Indeed many (standard) PROLOG programs are coded to return values by an extra argument. However it is a matter of style not to recode a function as a relation, but leave it explicitly as a function. Programs thus become more readable.

6. References

[1] RELFUN: A Relational/Function Integration with Valued Clauses, Harold Boley, SIGPLAN Notices 21(12), December 1986, pp. 87-98

[2] An Extension of the WPE-WAM Instructions for RELFUN's Return Values, Harold Boley, RFM Discussion Paper, University of Kaiserslautern, Postfach 3049, D-6750 Kaierslautern, March 1989

[3] A Functional Extension of Logic Programming orthogonal to Parallel Evaluation, Harold Boley, Gigalips Workshop Proceedings, Stockholm, SICS, April 1989 (updated for HP Bristol talk, October 1989)

[4] How to Invent a Prolog Machine, Peter Kursawe, New Generation Computing 5 (1987), pp. 97-114, OHMSHA , LTD and Springer Verlag

[5] Computing with Logic, David Maier/David S. Warren, Logic Programming with Prolog, Benjamin and Cummings Publ. Com. 1988

[6] A Tutorial on the Warren Abstract Machine for Computational Logic; John Gabriel,Tim Lindholm,E.L. Lusk, R.A. Overbeek; Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439

[7] A Tutorial on Prolog Implementation, Dr. Klaus Estenfeld, Siemens AG, ZTI INF 21, Otto-Hahn-Ring 6, D-8000 München 83

[8] Extending the Warren Abstract Machine to Many-sorted PROLOG, Hans-Jürgen Bürckert, July 85, Universität Kaiserslautern, FB Informatik, Postfack 3049, D-6750 Kaierslautern, Joint Research Project ITR-8501 A with NIXDORF Computer

[9] A Critique of Warren's abstract Prolog instruction set, Joachim Beer, GMD FIRST/TU Berlin

[10] Comments on Compiling Prolog-Programs using Warren's abstract instruction set, Version 2.0, Joachim Beer, Nov. 85, GMD-FIRST / TU Berlin

[11] An Abtract PROLOG Instruction Set, David H.D. Warren, Artificial Intelligence Center, SRI International, Technical Note 309, Oct. 83, SRI Project 4776

[12] NyWAM written in LISP by
Sven-Olof Nystroem, Uppsala University, Sweden.
uunet: SVEN-OLOF@aida.csd.uu.se

[13] Beer WAM written in LISP by
H.-J. Bürckert, Michael Tepp, Peter Forster, C. Decker.
University of Kaiserslautern, Postfach 3049,
D-6750 Kaiserslautern, Sonderforschungsbereich 314

[14] Register Allocation in a Prolog Machine, Saumya K. Debray, Departement of Computer Science, State University of New York at Stony Brook, NY 11794, Symposium on Logoc Programming, 1986, Salt Lake City, Utah.

Appendix A: Standard "append" - Compilation example in PROLOG
and in RELFUN

```
; standard PROLOG:
; append([],S,S).
; append([H|T],S,[H|R]):- append(T,S,R).

(defprocedure append/3
        (try L1 3)
        (trust L2 3)
L1
        (get_nil 1)
        (get_value_temp 2 3)
        (proceed)
L2
        (get_list 1)
        (unify_variable_temp 4)
        (unify_variable_temp 1)
        (get_list 3)
        (unify_value_temp 4)
        (unify_variable_temp 3)
        (execute append/3)
)

; cons is a built-in function: [A|B] === cons(A,B) !!
; cons-f is a defined Relfun-Function.
;
; append-f(nil,S) :- backquote(S).
; append-f(cons(H,T),S) :- cons-f(H,append-f(T,S)).
;  === append-f(cons(H,T),S) :- _1 is append-f(T,S), cons-f(H,_1).
; cons-f(H,T) :- backquote(cons(H,T)).
;

(defprocedure append-f/2
        (try L1 2)
        (trust L2 2)
L1      (get_nil 1)
        (valreg_write_variable_temp 2)   ; Argument 2 is the
        (proceed)                        ; result of the clause.
L2      (allocate 1)
        (get_list 1)
        (unify_variable_perm 1)          ; Y1 = H
        (unify_variable_temp 1)          ; X1 = T
        (call append-f/2 1)
        (put_value_perm 1 1)             ; Y1 --> A1
        (valreg_read_variable_temp 2)          ; valreg --> A2
        (deallocate)
        (execute cons-f/2)
)

(defprocedure cons-f/2
        (valreg_write_list)
        (unify_value_temp 1)             ; this is a simple cons
        (unify_value_temp 2)
        (proceed)
)
```

Appendix B
Listing of the NyWAM

```
; A WAM emulator in Common Lisp
;
; Author: Sven-Olof Nystroem
; Summer 1985 and May 1989
;
; Copyright (c) May 1989 by Sven-Olof Nystroem and Uppsala University.
; Permission to copy all or part of this material is granted, provided that
; the copies are not made or redistributed for resale, and that the copyright
; notice and reference to the source file appear.

; Note: There is no prolog compiler; you must supply your own WAM code.

; Some documentation is in Swedish, but I think you'll understand the code
; anyway.

; The emulator only handles the 'pure' prolog instructions given in Warren's
; paper.
```

;;; Part 1. Primitive operations

```
(defstruct (word (:print-function printword))
  (tag 'empty)
  (value nil))

(defmacro definstr (name param-list &body body)
  "Defines an instruction."
  `(setf (get',name 'instruction)
       #'(lambda ,param-list . ,body)))


(defmacro define-offset (name value)
  (if (not (numberp value))
      (error "Value of offset not a number"))
  `(setf (get ',name 'offset-value) ,value))

(defmacro offset (name)
  `(get ',name 'offset-value))

(defvar *registers* nil)

(defmacro define-register (name)
  (setq *registers* (delete-duplicates (cons name *registers*)))
  `(setf (get ',name 'reg-value)
         (make-word)))

(defmacro reg (name)
  `(get ',name 'reg-value))

(defmacro set-reg (name value)
  `(setf (reg ,name) ,value))

(defparameter memory-size 20000)

(defparameter start-of-heap 0)

(defparameter start-of-stack 10000)

(defvar memory (make-array memory-size
                           :initial-element (make-word)))

; easy optimisations: some defuns may be coded as a macro
(defun mem (adr)
  (aref memory (address adr)))

(defun setmem (adr val)
  (setf (aref memory (address adr)) val))
```

```
(defparameter number-of-argument-regs 16)

(defvar argument-regs (make-array  number-of-argument-regs))

(defun argument-reg(n)
  (aref argument-regs  (1- n)))

(defun set-argument-reg (n value)
  (setf (aref argument-regs  (1- n)) value))


(defun perm-variable(n)
  (ref-plus (reg E) (offset Y)  (1- n)))


(defun constant (c)
  (make-word :tag 'const
             :value c))

(defun constant-nil()
  (constant nil))

(defun functor (f)
  (make-word :tag 'fun
             :value f))

(defmacro defprocedure(name &body krop)
  `(setf (get ',name 'procedure) ',krop))

(defun label (label)
  (let* ((active-proc (car (word-value (reg p))))
         (temp1 (member label (get active-proc 'procedure)))
         (temp2 (if (numberp label) nil (get label 'procedure))))
    (when (and (null temp1) (null temp2))
          (error "Label ~a not found" label))
    (if (null temp1)
       (make-word :tag 'code :value (cons label temp2))
       (make-word :tag 'code :value (cons active-proc temp1)))))

(defun proc (proc)
  (let ((c  (get proc 'procedure)))
    (if (null c) (error (format nil "~a is not a known procedure" proc))
       (make-word :tag 'code
                  :value (cons proc c)))))

(defun neq(x y)
  (not (eq x y)))

(defun word-equal(x y)
  (and (equal (word-tag x) (word-tag y))
       (equal (word-value x) (word-value y))))

(defun address (val)
  (if (member (word-tag val) '(ref struct list))
      (word-value val)
      (error (format nil "Word ~a does not contain an address" val))))

(defun ref-plus (ref &rest numbers)
  (make-word :tag 'ref
             :value (apply #'+ (cons (address ref) numbers))))

(defun ref-lessp(ref1 ref2)
  (< (address ref1) (address ref2)))

(defun next-term-S ()
  (mem (set-reg S (ref-plus (reg S) 1))))

(defun new-value (val)
  (set-reg H (ref-plus (reg H) 1))
  (setmem (reg H) val))

(defun new-variable ()
```

```
  (set-reg H (ref-plus (reg H) 1))
  (setmem (reg H) (reg H)))

(defun new-struc (S)
  (set-reg H (ref-plus (reg H) 1))
  (setmem (reg H) S)
  (make-word :tag 'struct
             :value (word-value (reg H))))

(defun new-list-cell ()
  (make-word :tag 'list
             :value (word-value (ref-plus (reg H) 1))))

(defun deref (term)
  (if (and (eq (word-tag term) 'ref)
           (neq (mem term) term))
      (deref (mem term))
      term))

(defun unify(ref1 ref2)
  (cond  ((word-equal ref1 ref2))
         ((and (varp ref1)(varp ref2))
          (if (ref-lessp ref1 ref2)
              (bind ref2 ref1)
              (bind ref1 ref2)))
         ((varp ref1)
          (bind ref1 ref2))
         ((varp ref2)
          (bind ref2 ref1))
         ((not (eq (word-tag ref1)(word-tag ref2)))
          (fail))
         (t (case (word-tag ref1)
              ((const)
               (if (equal (word-value ref1)
                          (word-value ref2))
                   nil
                   (fail)))
              ((struct)
               (unify-structures (mem ref1) (mem ref2) ref1 ref2))
              ((list)
               (unify (deref (mem ref1))
                      (deref (mem ref2)))
               (unify (mem (ref-plus ref1 1))
                      (mem (ref-plus ref2 1))))))))

(defun unify-structures (fun1 fun2 ref1 ref2)
  (if (not (equal (word-tag fun1) (word-tag fun2)))
      (fail))
  (ecase (word-tag fun1)
    ((fun)
     (if (equal (word-value fun1)(word-value fun2))
         (let ((arity (second (word-value fun1))))
           (do ((i arity (1- i)))
               ((zerop i))
             (unify (deref (ref-plus ref1 i))
                    (deref (ref-plus ref2 i))))))))))

(defun varp (term)
  (and (eq (word-tag term) 'ref)
       (word-equal term (mem term))))

(defun bind (ref val)
  (setmem ref val)
  (if (or (not (ref-lessp (reg HB) ref))
          (and
           (<  start-of-stack (address ref))
           (not (ref-lessp (reg B) ref))))
      (trail ref)))

(defun trail (ref)
  (set-reg TR
   (make-word :tag 'trail
```

```
            :value (cons ref (word-value (reg TR))))))

(defun fail ()
  (let ((temp (reg TR)))
    (set-reg TR (mem (ref-plus (reg B) (offset TR1))))
    (set-reg H  (mem (ref-plus (reg B) (offset H1))))
    (set-reg E  (mem (ref-plus (reg B) (offset BCE))))
    (set-reg CP (mem (ref-plus (reg B) (offset BCP))))
    (set-reg P  (mem (ref-plus (reg B) (offset BP))))
    (set-reg A  (reg B))
    (unwind-trail temp (reg TR)))
  (throw 'fail nil))

(defun unwind-trail (r1 r2)
  (uwtr (word-value r1)(word-value r2)))

(defun uwtr(r1 r2)
  (cond
   ((neq r1 r2)
    (setmem (first r1) (first r1))
    (uwtr (rest r1) r2))))

(defun save-argument-regs (n)
  (dotimes (i n)
    (setmem (ref-plus (reg A) (offset A) (- i))
            (argument-reg (1+ i)))))


(defun restore-argument-regs (n)
  (dotimes (i n)
    (set-argument-reg (1+ i) (mem (ref-plus (reg A) (offset A) (- i))))))

(defun set-read-mode()
  (setq *read-mode* t))

(defun set-write-mode()
  (setq *read-mode* nil))

(defun read-mode()
  *read-mode*)

(defun lessp(x y)
  (< x y))
```

;;; Part 2: Some definitions.

```
;def av register

(define-register P  )
(define-register CP )
(define-register E  )
(define-register B  )
(define-register A  )
(define-register TR )
(define-register H  )
(define-register HB )
(define-register S  )


;def av offset i enviroment

(define-offset CE +1)
(define-offset CP +2)
(define-offset Y +3)

;define av offset i choice point

(define-offset BCE -5)
(define-offset BCP -4)
(define-offset B1  -3)
```

```
(define-offset BP  -2)
(define-offset TR1 -1)
(define-offset H1   0)
(define-offset A   -6)

(defconstant regs-in-choicepoint 6)

(defconstant regs-in-environment 2)
```

;;; Part 3. The instructions.

```
;**********  Put instructions **********

(definstr put_variable_perm (Yn Ai)
  (setmem (perm-variable Yn) (perm-variable Yn))
  (set-argument-reg Ai (perm-variable Yn)))

(definstr put_variable_temp (Xn Ai)
  (set-argument-reg Ai
                (set-argument-reg Xn (new-variable))))

(definstr put_value_perm (Yn Ai)
  (set-argument-reg Ai (mem (perm-variable Yn))))


(definstr put_value_temp (Xn Ai)
  (set-argument-reg Ai (argument-reg Xn)))

(definstr put_unsafe_value_perm (Yn Ai)
  (let ((temp (deref (perm-variable Yn))))
    (if (and (varp temp)
             (ref-lessp (reg E) temp))
        (bind temp (set-argument-reg Ai (new-variable)))
        (set-argument-reg Ai temp))))

(definstr put_constant (C Ai)
  (set-argument-reg Ai (constant C)))

(definstr put_nil (Ai)
  (set-argument-reg Ai (constant-nil)))

(definstr put_structure (F Ai)
  (set-argument-reg Ai (new-struc (functor F)))
  (set-write-mode))

(definstr put_list (Ai)
  (set-argument-reg Ai (new-list-cell))
  (set-write-mode))

;**********  Get instructions **********


(definstr get_variable_temp (Xn Ai)
  (set-argument-reg Xn (argument-reg Ai)))

(definstr get_variable_perm (Yn Ai)
  (setmem (perm-variable Yn) (argument-reg Ai)))

(definstr get_value_temp (Xn Ai)
  (let ((temp1 (deref (argument-reg Xn)))
        (temp2 (deref (argument-reg Ai))))
    (unify temp1 temp2)
    (set-argument-reg Xn temp1)))

(definstr get_value_perm (Yn Ai)
  (let ((temp1 (deref (perm-variable Yn)))
        (temp2 (deref (argument-reg Ai)))))
```

```
    (unify temp1 temp2)))

(definstr get_nil (Ai)
  (let ((temp (deref (argument-reg Ai))))
    (cond
     ((varp temp) (bind temp (constant-nil)))
     ((word-equal temp (constant-nil)))
     (t (fail)))))

(definstr get_constant (C Ai)
  (let ((temp (deref (argument-reg Ai))))
    (cond
     ((varp temp) (bind temp (constant C)))
     ((word-equal temp (constant C)))
     (t (fail)))))

(definstr get_structure (F Ai)
  (let ((temp (deref (argument-reg Ai))))
    (cond
     ((varp temp)
      (bind temp (new-struc (functor F)))
      (set-write-mode))
     ((and (eq (word-tag temp) 'struct)
           (word-equal (mem temp) (functor F)))
      (set-reg S temp)
      (set-read-mode))
     (t (fail)))))

(definstr get_list (Ai)
  (let ((temp (deref (argument-reg Ai))))
    (cond
     ((varp temp)
      (bind temp (new-list-cell))
      (set-write-mode))
     ((eq (word-tag temp) 'list)
      (set-reg S (ref-plus temp -1))
      (set-read-mode))
     (t (fail)))))



;**********  Unify instructions **********

(definstr unify_variable_temp (Xn)
  (if (read-mode)
      (set-argument-reg Xn (next-term-S))
      (set-argument-reg Xn (new-variable))))

(definstr unify_variable_perm (Yn)
  (if (read-mode)
      (setmem (perm-variable Yn) (next-term-S))
      (setmem (perm-variable Yn) (new-variable))))


(definstr unify_void(n)
  (if (read-mode)
      (dotimes (? n) (next-term-S))
      (dotimes (? n) (new-variable))))

(definstr unify_value_temp (Xn)
  (if (read-mode)
      (let ((temp1 (deref (argument-reg Xn)))
            (temp2 (deref (next-term-S))))
       (unify temp1 temp2)
       (set-argument-reg Xn temp1))
      (new-value (argument-reg Xn))))

(definstr unify_value_perm (Yn )
  (if (read-mode)
      (let ((temp1 (deref (perm-variable Yn)))
            (temp2 (deref (next-term-S))))
       (unify temp1 temp2))
```

```
      (new-value (mem (perm-variable Yn)))))

(definstr unify_local_value_temp (Xn)
  (if (read-mode)
      (let ((temp1 (deref (argument-reg Xn)))
            (temp2 (deref (next-term-S))))
       (unify temp1 temp2)
       (set-argument-reg Xn temp1))
      (let ((temp (deref (argument-reg Xn))))
       (if (and (varp temp)
                (< start-of-stack (address temp)))
           (set-argument-reg Xn
                             (bind temp (new-variable)))
           (new-value temp)))))

(definstr unify_local_value_perm (Yn)
  (if (read-mode)
      (let ((temp1 (deref (perm-variable Yn)))
            (temp2 (deref (next-term-S))))
       (unify temp1 temp2))
      (let ((temp (deref (perm-variable Yn))))
       (if (and (varp temp)
                (< start-of-stack (address temp)))
           (bind temp (new-variable))
           (new-value temp)))))

(definstr unify_nil ()
  (if (read-mode)
      (let ((temp (deref (next-term-S))))
       (cond
        ((varp temp) (bind temp (constant-nil)))
        ((word-equal temp (constant-nil)))
        (t (fail))))
      (new-value (constant-nil))))

(definstr unify_constant (C)
  (if (read-mode)
      (let ((temp (deref (next-term-S))))
       (cond
        ((varp temp) (bind temp (constant C)))
        ((word-equal temp (constant C)))
        (t (fail))))
      (new-value (constant C))))

;********** Indexings-instructions **********

(definstr try (L n)
  (set-reg A
           (ref-plus
            (reg A) n  regs-in-choicepoint))
  (save-argument-regs n)
  (setmem (ref-plus (reg A) (offset BCE)) (reg E))
  (setmem (ref-plus (reg A) (offset BCP)) (reg CP))
  (setmem (ref-plus (reg A) (offset B1)) (reg B))
  (setmem (ref-plus (reg A) (offset BP)) (reg P))
  (setmem (ref-plus (reg A) (offset TR1)) (reg TR))
  (setmem (ref-plus (reg A) (offset H1)) (reg H))

  (set-reg HB (reg H))
  (set-reg B  (reg A))
  (set-reg P  (label L)))


(definstr retry (L n)
  (restore-argument-regs n)
  (setmem (ref-plus (reg B) (offset BP)) (reg P))
  (set-reg P  (label L)))


(definstr trust (L n)
  (restore-argument-regs n)
  (set-reg P (label L))
```

```
  (set-reg A
         (ref-plus
          (reg A)
          (- n)
          (- regs-in-choicepoint)))
  (set-reg B  (mem (ref-plus (reg B) (offset B1)))))

;**********  added instructions ********** HGH

 (definstr try_me_else (L n)
   (set-reg A
         (ref-plus
          (reg A) n  regs-in-choicepoint))
   (save-argument-regs n)
   (setmem (ref-plus (reg A) (offset BCE)) (reg E))
   (setmem (ref-plus (reg A) (offset BCP)) (reg CP))
   (setmem (ref-plus (reg A) (offset B1)) (reg B))
   (setmem (ref-plus (reg A) (offset BP)) (label L))
   (setmem (ref-plus (reg A) (offset TR1)) (reg TR))
   (setmem (ref-plus (reg A) (offset H1)) (reg H))
   (set-reg HB (reg H))
   (set-reg B  (reg A))
 )


 (definstr retry_me_else (L n)
   (restore-argument-regs n)
   (setmem (ref-plus (reg B) (offset BP)) (label L))
 )

 (definstr trust_me_else_fail (n)
   (restore-argument-regs n)
   (set-reg A
         (ref-plus
          (reg A)
          (- n)
          (- regs-in-choicepoint)))
   (set-reg B  (mem (ref-plus (reg B) (offset B1)))))


(definstr switch_on_type( Va In Sy Ls St Ni Ot)
  (let ((temp (deref (argument-reg 1))))
    (case (word-tag temp)
      ((ref)  (set-reg P (label Va)))
      ((const)       (cond
                ((null (word-value temp)) (set-reg P (label Ni)))
                ((integerp (word-value temp)) (set-reg P (label In)))
                (t (set-reg P (label Sy)))))
      ((list)        (set-reg P (label Ls)))
      ((struct) (set-reg P (label St)))
      (T      (set-reg P (label Ot))))))


(definstr switch_on_constant(Len Table Default)
  (let* ((temp (deref (argument-reg 1)))
        (dest (second (assoc (word-value temp) Table :test #'equal))))
    (if dest (set-reg P (label dest))
       (set-reg P (label Default)))))

(definstr switch_on_structure(Len Table Default)
  (let* ((temp (deref (argument-reg 1)))
        (dest (second (assoc (word-value (mem temp)) Table :test #'equal))))
    (if dest (set-reg P (label dest))
       (set-reg P (label Default)))))

;**********  Controll instructions **********

(definstr allocate (enviroment-size)
  (let ((temp (reg E)))
    (set-reg E (reg A))
    (setmem (ref-plus (reg E) (offset CP)) (reg CP))
    (setmem (ref-plus (reg E) (offset CE)) temp)
```

```
    (set-reg A
            (ref-plus (reg A) enviroment-size regs-in-environment))))

(definstr deallocate ()
  (if (<= (address (reg B)) (address (reg E)))
      (set-reg A (reg E)))
  (set-reg CP (mem (ref-plus (reg E) (offset CP))))
  (set-reg E  (mem (ref-plus (reg E) (offset CE)))))

(definstr proceed ()
  (set-reg P (reg CP)))

(definstr execute(proc)
  (set-reg P  (proc proc)))

(definstr call(proc k)
  (set-reg CP (reg P))
  (set-reg P  (proc proc)))

;*********  Utility instructions *********

(definstr choice_temp(Xn)
  (set-argument-reg Xn (reg B)))

(definstr commit_temp(i Xn)
  (set-reg B (argument-reg Xn))
  (set-reg  TR
           (make-word
            :tag 'reference
            :value (compact-trail
                   (word-value (reg TR))
                   (word-value (mem (ref-plus (reg B) (offset TR1))))
                   (word-value (reg B))))))


(definstr commit-perm(i Yn)
  (set-reg B (permanent-variable Yn))
  (set-reg  TR
           (make-word
            :tag 'reference
            :value (compact-trail
                   (word-value (reg TR))
                   (word-value (mem (ref-plus (reg B) (offset TR1))))
                   (word-value (reg B))))))

(defun compact-trail (TR TR1 B)
  (cond
   ((eq TR TR1) TR)
   ((< B (word-value (car TR)))
    (compact-trail (cdr TR) TR1 B))
   (t (cons (car TR) (compact-trail (cdr TR) TR1 B)))))

;*********  Special instructions **********

(definstr has-succeded()
  (do ((var *user-variabels* (rest var)))
      ((null var))
    (format *standard-output*
          "~% ~a = ~a " (car (first var)) (show-term (cdr (first var)))))
  (if (y-or-n-p "More solutions?")
      (fail)
      (throw 'halt nil)))

(definstr has-failed()
  (format *standard-output*
        "No (more) solutions")
  (throw 'halt nil))

(definstr escape-to-lisp(fun)
  (funcall
   (eval fun)))
```

```
;(has-failed)
;
;
;(escape-to-lisp #`(lambda ()(.. )))
;
```

;;; Part 4. User Interface.

```
(defun interpret ()
  (catch 'fail
        (let ((instr (second (word-value (reg P)))))
          (set-reg P (increment (reg P)))
          (exec instr)))
  (if nil (print-changed-registers)))

(defun exec (instr)
  (if (not (atom instr))
      (let ((instr-body (get (first instr) 'instruction)))
       (if (null instr-body)
           (error (format nil "~a not a defined instruction" (first instr)))
           (apply instr-body (rest instr))))))

(defun increment (ref)
  (when (not (eq (word-tag ref) 'code))
       (error "argument to increment is not code-pointer"))
  (make-word :tag 'code
             :value (cons (car (word-value ref))
                          (rest (rest (word-value ref))))))

(defun call (anruf)
  (init)
  (setq  *user-variabels* nil)
  (let* ((name      (first anruf))
         (arg-list  (rest anruf))
         (len       (length arg-list))
         (carg-list (map 'list #'construct-term arg-list))
         (name-len
          (intern (format nil "~a/~d" name len))))
    (dotimes (i len)
      (set-argument-reg (1+ i) (nth  i carg-list)))
    (let
     ((code `((try proc 0)
              (trust fail 0)
              proc
              (call ,name-len 0)
              (has-succeded)
              fail
              (has-failed))))
     (setf (get 'top-level 'procedure) code)
     (set-reg P (make-word :tag 'code
                           :value (cons 'top-level code)))))
  (setq  *user-variabels* (reverse *user-variabels*))
  (stepp))


(defvar  *user-variabels* nil)


(defun init()
  (set-reg E (make-word :tag 'ref
                        :value start-of-stack))
  (set-reg B (make-word :tag 'ref
                        :value start-of-stack))
  (set-reg A (make-word :tag 'ref
                        :value start-of-stack))
  (set-reg H (make-word :tag 'ref
                        :value start-of-heap))
  (set-reg HB (make-word :tag 'ref
```

```
                    :value start-of-heap))
  (set-reg TR (make-word :tag 'trail
                    :value nil))
  (set-reg S (make-word :tag 'ref
                   :value start-of-heap)))

(defun show-list (term-list)
  (let ((head (show-term (mem term-list)))
        (tail (show-term (mem (ref-plus  term-list 1)))))
    (if (eq (first tail) 'list)
       (cons head (rest tail))
       (cons head tail))))

(defun printword(word &optional stream depth)
  (let ((tag (word-tag word))
        (value (word-value word)))
    (ecase tag
     ((ref struct list)
      (format stream "[~(~6a~):~16@a]" tag value))
     ((const)
      (format stream "[~(~6a~):~16@a]" tag value))
     ((fun)
      (format stream "[~(~6a~):~14@a/~2a" tag (first value) (second value)))
     ((empty)
      (format stream "[???   :          ??????]"))
     ((code)
      (format stream "[code  :~16@a in ~a]" (second value)(first value)))
     ((trail)
      (format stream "[trail :~16@a]" (first value))))))

(defun printmseg(loc1 loc2 &optional (stream *standard-output*))
  (do ((i loc1 (1+ i)))
      ((> i loc2))
    (printmloc i stream)
    (if (eql i (address (reg E))) (format stream " <== E"))
    (if (eql i (address (reg B))) (format stream " <== B"))
    (if (eql i (address (reg H))) (format stream " <== H"))
    (if (eql i (address (reg HB)))(format stream " <== HB"))
    (if (eql i (address (reg S))) (format stream " <== S"))))

(defun printmloc (i &optional (stream  *standard-output*))
  (format stream "~%mem[~3d]=~a " i (aref memory i)))

(defun printreg(name)
  (format *debug-io*
         "~% Register ~2a = ~a "
         name (get name 'reg-value)))


; Warning : THIS ROUTINE MAY BE MACHINE DEPENDENT
(defun step-get-char ()
 (do ((char (read-char *debug-io*) (read-char *debug-io*) ))
     ((neq char #\Newline) (make-char char))
     nil
 )
)

;;; orginal code:
;;; (defun step-get-char ()
;;;  (prog1 (make-char
;;;       (princ (read-char *debug-io*) *debug-io*))
;;;      (terpri *debug-io*)))


(defun stepp()
  (with-open-stream
   (*debug-io* (open "/dev/tty" :direction :io ))
;;;;;;;;;;;;;;;   (*debug-io* (open "TTY:" :direction :io))    ;;; comment out
   (catch
    'halt
    (loop (format *standard-output* "~% P = ~a  :" (get 'p 'reg-value))
          (case (step-get-char)
```

```
          ((#\? #\H #\h)
           (print-stepp-help))
          ((#\Newline #\Space #\S #\s)(interpret))
          ((#\R #\r)
           (loop (interpret)))
          ((#\F #\f)
           (catch 'fail (fail)))
          ((#\E #\e)
           (throw 'halt nil))
          ((#\V #\v)
           (show-value-of-something))
          ((#\x #\X)
           (princ "abc-test-debug-message-later-to-kill"))
          (t (princ "Unknown input , Type ? for help " *debug-io*)))))))
          ; ^ this is the otherwise case

(defun print-stepp-help()
  (princ
"      All commands consist of one character.

       E,e           Terminate and go to LISP.
       F,f           Generate a fail. (Sometimes this command may
                     cause trouble.)
       H,h,?         Output this Help-Menu.
       R,r           Execute until program succeeds.
       S,s,newline   Single step execution.
       V,v           Output values before single step.
" *debug-io*))

(defun show-value-of-something()
  (princ "Value of?" *debug-io*)
  (case (step-get-char)
    ((#\?)(value-help)
          (show-value-of-something))
    ((#\A #\a)
    (princ "Type number of argumentregisters to output:" *debug-io*)
     (dotimes (i (read))
       (format *standard-output* "~% A(~2a) = ~a" (1+ i) (argument-reg (1+ i)))))
    ((#\H #\h)
     (printmseg start-of-heap (address (reg H))))
    ((#\S #\s)
     (printmseg start-of-stack (address (reg A))))
    ((#\R #\r)
     (dolist (x *registers*)
       (format *debug-io* "~%   Reg ~2a = ~16a" x (get x 'reg-value)))
     (terpri *debug-io*))
    (t (princ "Unknown input , Type ? for help " *debug-io*))))


(defun value-help()
  (princ
"      All commands consist of one character.

       ?             Output this Help-Menu.
       A,a           Output n (to be read) argumentregisters
                     A(0)..A(n-1).
       H,h           Output Heap.
       R,r           Output all registers exept argumentregisters.
       S,s           Output stack.

" *debug-io*))


(defun bracket-reader (&optional (stream *standard-input*) char)
  "Called by read if '[' is scanned."
  "Prologlists have the form [e1 e2 .. en] or [e1 e2 .. a . b]"
  "with n>=0 and ei and a and b are s-expr.  "

  (let ((object (read )))
    (if (eql object #\])
        nil
        (make-cons :car object :cdr (bracket-reader2 stream)))))
```

```
; Helpfunction for bracket-reader.

(defun bracket-reader2(stream )
  (let ((char (peek-char t stream)))
    (cond
     ((eql char #\]) (read-char stream) nil)
     ((eql char #\.)
      (read-char)
      (prog1 (read)
             (if (not (eql (read-char) #\]))
                 (error "] expected"))))
     (t (make-cons :car (read) :cdr (bracket-reader2 stream))))))

(set-macro-character #\[ #'bracket-reader)
(set-macro-character #\] #'(lambda (stream char) #\]))

(defstruct (cons (:print-function printcons))
  (car nil)
  (cdr nil))

(defun printcons(c &optional (stream *standard-output*) (depth nil))
  (cond
   ((and *print-level* (<= *print-level* depth))
    (princ "#" stream))
   (t (princ "[" stream)
      (write (cons-car c) :stream stream :level (1+ depth))
      (printrestcons (cons-cdr c) stream depth))))

(defun printrestcons(c stream depth)
  (cond
   ((null c) (princ "]" stream))
   ((and *print-level* (eql *print-level* depth))
    (princ " .. ]" stream))
   ((not (cons-p c))
    (princ " . " stream)
    (write c :stream stream :level (1+ depth))
    (princ "]" stream))
   (t
    (princ " " stream)
    (write (cons-car c) :stream stream :level (1+ depth))
    (printrestcons (cons-cdr c) stream  depth))))



(defun construct-term (term)
  (cond
   ((null term)
    (constant-nil))
   ((cons-p term)
    (construct-list term))
   ((atom term)
    (constant term))
   ((eq (first term) 'var)
    (construct-variable (second term)))
   (t
    (let ((temp (map 'list #'construct-term (rest term)))
          (ref
           (new-struc (functor (list (first term) (length (rest term)))))))
     (mapc #'new-value temp)
     ref))))

(defun construct-variable (var)
  (cond ((numberp var)
         (new-value (make-word :tag 'ref
                               :value var)))
        ((assoc var *user-variabels*)
         (new-value (cdr (assoc var *user-variabels*))))
        (t
         (let ((temp (new-variable)))
           (setq  *user-variabels* (cons (cons var temp) *user-variabels*))
```

```
        temp))))

(defun construct-list (term)
  (let ((head (construct-term (cons-car term)))
        (tail (construct-term (cons-cdr term)))
        (ref (new-list-cell)))
    (new-value head)
    (new-value tail)
    ref))



(defun show-term (term)
  (ecase (word-tag term)
   ((const) (word-value term))
   ((ref) (if (varp term) (list 'var (word-value term))
                  (show-term (mem term))))
   ((list) (make-cons :car (show-term (mem term))
                      :cdr (show-term (mem (ref-plus term 1)))))
   ((struct)
    (ecase (word-tag (mem term))
     ((fun)
      (let ((name (first (word-value (mem term))))
            (arity (second (word-value (mem term))))
            (ref   (ref-plus term 1)))
       (cons name (show-terms ref arity))))))))

(defun show-terms (ref arity)
  (if (zerop arity) nil
      (cons (show-term (mem ref))
            (show-terms (ref-plus ref 1) (1- arity)))))
```

Appendix C
Listing of the added instructions in the NyWAM

```
(define-register VALREG )

;**********  VALREG_WRITE_*_* INSTRUCTIONS **********

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_constant ( C )                                   ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the constant c into register VALREG

(definstr valreg_write_constant (C)
  (set-reg VALREG (constant C)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_nil ()                                           ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the special constant nil into register VALREG.
;  NIL represents an empty list

(definstr valreg_write_nil ()
  (set-reg VALREG (constant-nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_variable                                         ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  creates an unbound variable on the global stack
;  puts a reference to it into the register VALREG

(definstr valreg_write_variable ()
  (set-reg VALREG (new-variable)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_value_temp ( an )                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the value of register An into register VALREG

(definstr valreg_write_value_temp (Xn)
  (set-reg VALREG (argument-reg Xn)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_value_perm ( Yn )                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; puts the variable Yn  into VALREG

(definstr valreg_write_value_perm (Yn)
  (set-reg VALREG (mem (perm-variable Yn))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   valreg_write_value_unsafe_perm (Yn)                           ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  the instruction dereferences Yn and puts the result in VALREG.
;  If Yn dereferences to a variable in the current environment, that variable
;  is bound to a new global variable.

(definstr valreg_write_value_unsafe_perm (Yn)
  (let ((temp (deref (perm-variable Yn))))
    (if (and (varp temp)
```

```
            (ref-lessp (reg E) temp))
        (bind temp (set-reg VALREG (new-variable)))
        (set-reg VALREG temp))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                  valreg_write_list ()                                        ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  creates a list at the current top of the global stack
;  a reference to the beginning of the list is put into register VALREG
;  execution proceeds in write mode

(definstr valreg_write_list ()
  (set-reg VALREG (new-list-cell))
  (set-write-mode))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                   valreg_write_structure (f)                                 ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  pushes the functor f for the structure onto the global stack.
;  Remark: functor f is (name arity)
;  Puts the corresponding structure pointer into register VALREG.
;  Execution proceeds in write mode

(definstr valreg_write_structure (F)
  (set-reg VALREG (new-structure (functor F)))
  (set-write-mode))


;******************  VALREG_READ_*_* INSTRUCTIONS *****************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                   valreg_read_variable_temp(Xn)                              ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Represents a left hand side of an is-primitive that is an unbound X-variable.
;  The instruction simply gets
;  the value of register VALREG and stores it in Xn.

(definstr valreg_read_variable_temp (Xn)
  (set-argument-reg Xn (reg VALREG)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                   valreg_read_variable_perm (Yn)                             ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Represents a left hand side of an is-primitive that is an unbound Y-variable.
;  The instruction simply gets
;  the value of register VALREG and stores it in yn

(definstr valreg_read_variable_perm (Yn)
  (setmem (perm-variable Yn) (reg VALREG)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                   valreg_read_value_temp (Xn)                                ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is an bound variable.
;  The instruction gets
;  register VALREG and unifies that with the contents of variable Xn.

(definstr valreg_read_value_temp (Xn)
  (let ((temp1 (deref (argument-reg Xn)))
        (temp2 (deref (reg VALREG))))
    (unify temp1 temp2)
    (set-argument-reg Xn temp1)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                    valreg_read_value_perm (Yn)                               ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is an bound variable.
;  The instruction gets
;  register VALREG and unifies that with the contents of variable Yn.

(definstr valreg_read_value_perm (Yn)
  (let ((temp1 (deref (perm-variable Yn)))
        (temp2 (deref (reg VALREG))))
    (unify temp1 temp2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                    valreg_read_constant (C)                                  ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is a constant.
;  The instruction gets VALREG and dereferences it. If the result is a
;  reference to a variable, that variable is bound to the constant C and the
;  binding is trailed if neccesary. Otherwise the result is compared
;  with the constant C; if the two values are not idendical,backtracking occurs

(definstr valreg_read_constant (C)
  (let ((temp (deref (reg VALREG))))
    (cond
     ((varp temp) (bind temp (constant C)))
     ((word-equal temp (constant C)))
     (t (fail)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                    valreg_read_nil()                                         ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive which is the constant nil.

(definstr valreg_read_nil ()
  (let ((temp (deref (reg VALREG))))
    (cond
     ((varp temp) (bind temp (constant-nil)))
     ((word-equal temp (constant-nil)))
     (t (fail)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                    valreg_read_structure (f)                                 ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Marks the beginning of a structure occurring in a lhs of an is-primitive.
;  The instruction gets the value of register VALREG and dereferences it.
;  If the result is a reference to a variable that variable is bound to a new
;  structure pointer, pointing at the top of the global stack and the binding
;  is trailed if necessary. Functor f is pushed onto the global stack, and
;  execution proceeds in write mode. Otherwise if the result is a structure
;  and its functor is identical to f. The register S is set to point to
;  the arguments of the structure and execution proceeds in read mode.

(definstr valreg_read_structure (F)
  (let ((temp (deref (reg VALREG))))
    (cond
     ((varp temp)
      (bind temp (new-structure (functor F)))
      (set-write-mode))
     ((and (eq (word-tag temp) 'struct)
           (word-equal (mem temp) (functor F)))
      (set-reg S temp)
      (set-read-mode))
     (t (fail)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    valreg_read_list                                            ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  this instruction is a special case of the read_valreg_structure.
;  the structure in question is in this case a list.

(definstr valreg_read_list ()
  (let ((temp (deref (reg VALREG))))
    (cond
     ((varp temp)
      (bind temp (new-list-cell))
      (set-write-mode))
     ((eq (word-tag temp) 'list)
      (set-reg S (ref-plus temp -1))
      (set-read-mode))
     (t (fail)))))


; *-----------------------------------CONTROL-------------------------------*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    proceed_true                                                ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; proceed_true sets the VALREG-register to T (LISP constant TRUE)
(definstr proceed_true()
  (set-reg VALREG (constant T))
  (set-reg P (reg CP)))


; This function is called, when a goal is satisfied,
; additionally VALREG must be written to the screen.

(definstr has-succeded()
  (do ((var *user-variabels* (rest var)))
      ((null var))
    (format *standard-output*
          "~% ~a = ~a " (car (first var)) (show-term (cdr (first var)))))
  (format *standard-output* "~% VALREG = ~a" (show-term (reg VALREG)))
  (if (y-or-n-p "~%More solutions? ")
      (fail)
      (throw 'halt nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                              call-is                                           ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call-is should be used for queries of the form : ?- <passive> is <active>
; === (call-is <passive> <active>)

(defun call-is (isterm anruf)
  (init)
  (setq  *user-variabels* nil)
  (let* ((name      (first anruf))
        (arg-list  (rest anruf))
        (len       (length arg-list))
        (carg-list (mapcar #'construct-term arg-list))
        (isterm    (construct-term isterm))
        (istermt   (word-tag isterm))
        (istermv   (word-value isterm))
         (getcmd    nil)
        (name-len
         (intern (format nil "~a/~a" name len))))
    (dotimes (i len)
      (set-argument-reg i (nth  i carg-list)))
   (apply (get 'allocate 'instruction) '(1) )
   (setmem (perm-variable 0) isterm)
    (let
```

```
   ((code `((try proc 1)
            (trust fail 1)
            proc
            (call ,name-len 1)
             (valreg_read_value_perm 0)
            (has-succeded)
            fail
            (has-failed)
             (deallocate))
    ))
   (setf (get 'top-level 'procedure) code)
   (set-reg P (make-word :tag 'code
                         :value (cons 'top-level code)))))
(setq  *user-variabels* (reverse *user-variabels*))
(stepp))
```

Appendix D
Listing of the added instructions in the BBT

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_constant ( c )                               ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the constant c into register VALREG

(defun wpm-valreg_write_constant ( c )
  (wpm-ds-create-element const c wpm-ds*valreg 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_nil ()                                       ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the special constant nil into register VALREG.
;  NIL represents an empty list

(defun wpm-valreg_write_nil ()
  (wpm-ds-create-element const nil wpm-ds*valreg 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_variable                                     ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  creates an unbound variable on the global stack
;  puts a reference to it into the register VALREG

(defun wpm-valreg_write_variable ()
  (wpm-ds-create-element ref wpm-ds*gpos wpm-ds*valreg 0)
  (wpm-ds-create-element unbound wpm-ds*gpos wpm-ds*memory wpm-ds*gpos)
  (wpm-ds-set-var wpm-ds*gpos (+ wpm-ds*gpos 2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_x_value ( an )                               ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  puts the value of register An into register VALREG

(defun wpm-valreg_write_x_value ( an )
  (wpm-ds-copy-element wpm-ds*reg (* 2 an) wpm-ds*valreg 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_y_value (n)                                  ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; puts the variable Yn  into VALREG

(defun wpm-valreg_write_y_value (n)
 (wpm-ds-copy-element wpm-ds*memory (+ wpm-ds*currenv 3 (* 2 n)) wpm-ds*valreg 0)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
;                    wpm-valreg_write_y_unsafe_value (n)                           ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  the instruction dereferences Yn and puts the result in VALREG.
;  If Yn dereferences to a variable in the current environment, that variable
;  is bound to a new global variable. The binding is trailed if necessary.

(defun wpm-valreg_write_y_unsafe_value (n)
  (let* ((temp (wpm-ds-deref wpm-ds*memory (+ 3 (* 2 n) wpm-ds*currenv)))
       (temp.adr (cadr temp)))
    (case (wpm-ds-get-tag wpm-ds*memory temp.adr)
    ((1 2 3 4 5 ) (wpm-ds-copy-element wpm-ds*memory temp.adr wpm-ds*valreg 0 ))
```

```
      (t (cond ((> temp.adr wpm-ds*currenv)
              (wpm-ds-create-element (wpm-ds-get-tag wpm-ds*memory temp.adr)
                                     wpm-ds*gpos wpm-ds*memory wpm-ds*gpos)

              (wpm-ds-create-element ref wpm-ds*gpos wpm-ds*valreg 0)
              (wpm-ds-create-element ref wpm-ds*gpos wpm-ds*memory temp.adr)
              (wpm-ds-set-var wpm-ds*gpos (+ 2 wpm-ds*gpos)))
              (t
              (wpm-ds-create-element ref temp.adr wpm-ds*valreg 0)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                  wpm-valreg_write_list ()                                       ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  creates a list at the current top of the global stack
;  a reference to the beginning of the list is put into register VALREG
;  execution proceeds in write mode

(defun wpm-valreg_write_list ()
  (wpm-ds-create-element list wpm-ds*gpos wpm-ds*valreg 0)
  (wpm-ds-set-var wpm-ds*mode 'write))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   wpm-valreg_write_structure (f)                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  pushes the functor f for the structure onto the global stack.
;  Remark: functor f is (name arity)
;  Puts the corresponding structure pointer into register VALREG.
;  Execution proceeds in write mode

(defun wpm-valreg_write_structure (f)
  (wpm-ds-create-element structure wpm-ds*gpos wpm-ds*valreg 0)
  (wpm-ds-create-structure-name f)
  (wpm-ds-set-var wpm-ds*gpos (1+ wpm-ds*gpos))
  (wpm-ds-set-var wpm-ds*mode 'write))


;-------------------------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   wpm-valreg_read_x_variable (n)                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Represents a left hand side of an is-primitive that is an unbound X-variable.
;  The instruction simply gets
;  the value of register VALREG and stores it in Xn.

(defun wpm-valreg_read_x_variable (an)
  (wpm-ds-copy-element wpm-ds*reg (* 2 an) wpm-ds*valreg 0))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                   wpm-valreg_read_y_variable (n)                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Represents a left hand side of an is-primitive that is an unbound Y-variable.
;  The instruction simply gets
;  the value of register VALREG and stores it in yn

(defun wpm-valreg_read_y_variable (n)
  (wpm-ds-copy-element wpm-ds*valreg 0 wpm-ds*memory
                                     (+ wpm-ds*currenv 3 (* 2 n))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    wpm-valreg_read_x_value                                     ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is an bound variable.
;  The instruction gets
;  register VALREG and unifies that with the contents of variable an.

(defun wpm-valreg_read_x_value (an)
  (wpm-unify  wpm-ds*valreg 0 wpm-ds*reg (* 2 an)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    wpm-valreg_read_y_value                                     ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is an bound variable.
;  The instruction gets
;  register VALREG and unifies that with the contents of variable yn.

(defun wpm-valreg_read_y_value (n)
  (wpm-unify  wpm-ds*valreg 0 wpm-ds*memory (+ wpm-ds*currenv 3 (* 2 n))))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    wpm-valreg_read_constant (c)                                ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is a constant.
;  The instruction gets VALREG and dereferences it. If the result is a
;  reference to a variable, that variable is bound to the constant c and the
;  binding is trailed if neccesary. Otherwise the result is compared
;  with the constant c; if the two values are not idendical backtracking occurs

(defun wpm-valreg_read_constant (c)
  (let* ((temp (wpm-ds-deref wpm-ds*valreg 0))
         (temp.stack (car temp))
         (temp.adr (cadr temp))
         (temp.tag (wpm-ds-get-tag temp.stack temp.adr)))
    (case temp.tag
      (0 (wpm-trail temp.adr)
        (wpm-ds-create-element const c temp.stack temp.adr))
      (1 (cond ((not (equal (wpm-ds-get-addr temp.stack temp.adr) c))(wpm-fail))))
      (t (wpm-fail)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                ;
;                    wpm-valreg_read_nil()                                       ;
;                                                                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  represents a left hand side of an is-primitive that is the constant nil.

(defun wpm-valreg_read_nil ()
  (let* ((temp (wpm-ds-deref wpm-ds*valreg 0))
         (temp.stack (car temp))
         (temp.adr (cadr temp))
         (temp.tag (wpm-ds-get-tag temp.stack temp.adr)))
    (case temp.tag
      (0 (wpm-trail temp.adr)
        (wpm-ds-create-element const nil temp.stack temp.adr))
      (1 (cond ((not (equal (wpm-ds-get-addr temp.stack temp.adr) nil))(wpm-
fail))))
      (t (wpm-fail)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                    wpm-valreg_read_structure (f)                                ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Marks the beginning of a structure occurring in a lhs of an is-primitive.
;  The instruction gets the value of register VALREG and dereferences it.
;  If the result is a reference to a variable that variable is bound to a new
;  structure pointer, pointing at the top of the global stack and the binding
;  is trailed if necessary. Functor f is pushed onto the global stack, and
;  execution proceeds in write mode. Otherwise if the result is a structure
;  and its functor is identical to f. The register nextarg is set to point to
;  the arguments of the structure and execution proceeds in read mode.

(defun wpm-valreg_read_structure (f)
  (let* ((temp (wpm-ds-deref wpm-ds*valreg 0))
         (temp.stack (car temp))
         (temp.adr (cadr temp))
         (temp.tag (wpm-ds-get-tag temp.stack temp.adr)))
    (case temp.tag
      (0 (wpm-trail temp.adr)
        (wpm-ds-create-element structure wpm-ds*gpos temp.stack temp.adr)
         (wpm-ds-create-structure-name f)
        (wpm-ds-set-var wpm-ds*gpos (+ wpm-ds*gpos 1))
        (wpm-ds-set-var wpm-ds*mode 'write))
      (2 (let ((struc.adr (wpm-ds-get-addr temp.stack temp.adr)))
           (cond ((equal (wpm-ds-get-tag wpm-ds*memory struc.adr) f)
                  (wpm-ds-set-var wpm-ds*nextarg (+ struc.adr 1))
                  (wpm-ds-set-var wpm-ds*mode 'read))
                 (t (wpm-fail)))))
      (t (wpm-fail)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                    wpm-valreg_read_list                                         ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  this instruction is a special case of the wpm-read_valreg_structure.
;  the structure in question is in this case a list.

(defun wpm-valreg_read_list ()
  (let* ((temp (wpm-ds-deref wpm-ds*valreg 0))
         (temp.stack (car temp))
         (temp.adr (cadr temp))
         (temp.tag (wpm-ds-get-tag temp.stack temp.adr)))
    (case temp.tag
      (0 (wpm-trail temp.adr)
        (wpm-ds-create-element list wpm-ds*gpos temp.stack temp.adr)
        (wpm-ds-set-var wpm-ds*mode 'write))
      (3 (wpm-ds-set-var wpm-ds*nextarg (wpm-ds-get-addr temp.stack temp.adr))
        (wpm-ds-set-var wpm-ds*mode 'read))
      (t (wpm-fail)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                 ;
;                    wpm-proceed_true                                             ;
;                                                                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; wpm-proceed_true sets the VALREG-register to T (LISP constant TRUE)

(defun wpm-proceed_true ()
  "the program counter pc is reset to the continuation pointer pc"
  (wpm-ds-create-element const T wpm-ds*valreg 0)
  (wpm-ds-set-var wpm-ds*pc wpm-ds*continst))
```