# A WAM Implementation for the Meta-logic Programming Language 'Log

University of Houston

Department of Computer Science

Iliano Cervesato

July 24th, 1992

# Overview

- Meta-programming in ’Log
- The WAM
- A WAM implementation of ’Log

# Meta-programs

... are *programs that treat other programs as data.*

**They are used in**:

- Theorem Proving
- AI (Knowledge Based Systems, ...)
- Integrated Programming Environments
- ...

**Meta-programming in Logic Programming**:

- 1982: Bowen & Kowalski's paper
- 1985: *MetaProlog* (Bowen et al.)
- 1986: λ*Prolog* (Miller et al.)
- 1989: *Reflective Prolog* (Costantini, Lanzarone)
- 1990: *Gödel* (Hill, Lloyd)
- 1991: ... 'Log

# ’Log

... a logic meta-programming language

**New features**:

- names
- structural representations
- constraints

**Objectives**:

- treat any syntactic entity of the language at the meta-level
- efficient implementation in time and space;
- sound and complete logical semantics
- offer the users simple, powerful and easy-to-use meta-programming facilities

# Names

are *atomic* ground terms associated with *each* syntactic entities of 'Log

**Examples**:

- characters: **ch(**a**)** **ch(**Z**)**
- symbols: **sy(**foo**)** **sy(**Alpha**)**
- terms: **tr(**f(a,g(X),X)**)** **tr(**Alpha**)**
- clauses: **cl(**append([A|L1],L2,[A|L3]) :-
  append(L1,L2,L3)**)**
- programs: **pg(**
  append([],L,L).
  append([A|L1],L2,[A|L3]) :-
  append(L1,L2,L3).
  **)**

NB:

- *sy(foo)* and *sy(Alpha)* **do not** unify
- *X* and *tr(X)* **do** unify

# Structural representations

represent an object as the list of the names of its components

**Examples**:

- symbols: **[**ch(f),ch(o),ch(o)**]**
  **[**ch(A),ch(l),ch(p),ch(h),ch(a)**]**
- terms: **[**sy(f),sy(a),[sy(g),sy(X)],sy(X)**]**
  sy(Alpha)
- clauses: **[**tr(append([A|L1],L2,[A|L3])),
  tr(append(L1,L2,L3))**]**
- programs: **[**cl(append([],L,L)),
  cl(append([A|L1],L2,[A|L3]) :-
  append(L1,L2,L3))**]**

NB:

- *[sy(f),X]* is not a structural representation: only *some* of its ground instances are
- *[sy(f),X]* is an *incomplete structural representation*

# Constraints

'Log provides 4 operators to relate names and structural representations

**Examples**:

- symbols: sy(foo) **<=s=>** [ch(f),ch(o),ch(o)]
- terms: tr(f(a,g(X),X)) **<=t=>** [sy(f),sy(a),[sy(g),sy(X)],sy(X)]
- clauses: cl(...) **<=c=>** ...
- programs: pg(...) **<=p=>** ...

$\chi$ <=*x*=> $\upsilon$ expresses *constraints* for the variable appearing in $\chi$ and $\upsilon$

**?-** tr(f(g(a),b,C)) <=t=> [sy(f),**A**,sy(b),sy(C)].
**A** $\rightarrow$ [sy(g),sy(a)].

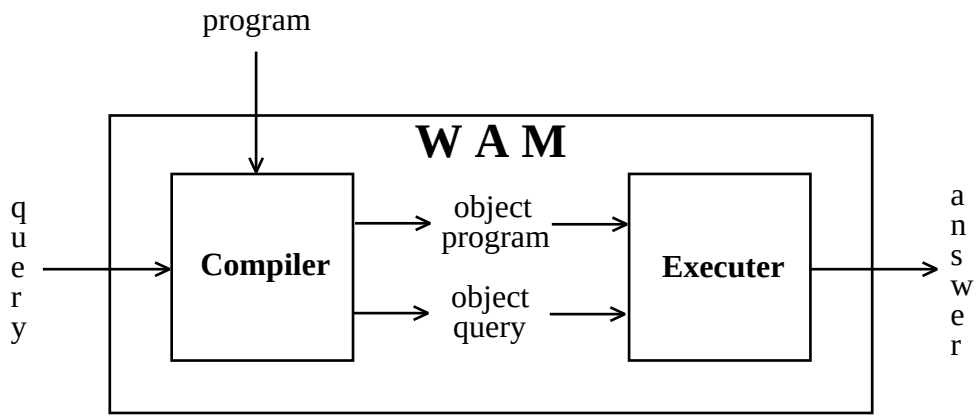
**?- N** <=t=> [sy(f),[sy(g),sy(a)],sy(b),sy(C)].
**N** $\rightarrow$ tr(f(g(a),b,C)).

**?- N** <=t=> [sy(f),**A**,sy(b),sy(C)], **A**=[sy(g),sy(a)].
**A** $\rightarrow$ [sy(g),sy(a)],
**N** $\rightarrow$ tr(f(g(a),b,C)).

**?- N** <=t=> [sy(f),**A**,**B**,sy(C)], **A**=[sy(g),sy(a)].
**A** $\rightarrow$ [sy(g),sy(a)],
**N** <=t=> [sy(f),[sy(g),sy(a)],**B**,sy(C)]. $\Leftarrow$

# Warren Abstract Machine

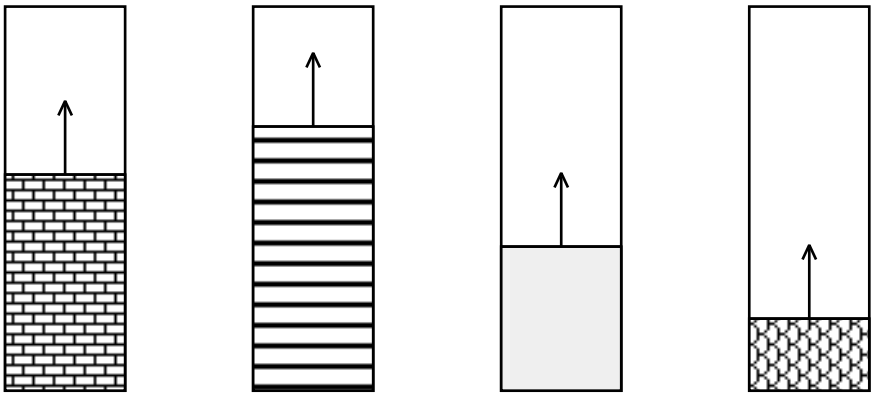
The WAM is an architecture for *compiling* and *executing* Prolog code.




- the *object code* is an assembly language
- the *executer* operates like a traditional line-code assembler
- some common instructions are:

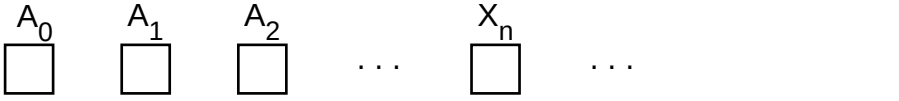
```
get_variable     X5,     A1
unify_constant   foo
set_value        X4,     A0
put_list         X3
allocate         4
call             append, 0
try_me_else      app_2,  3
```

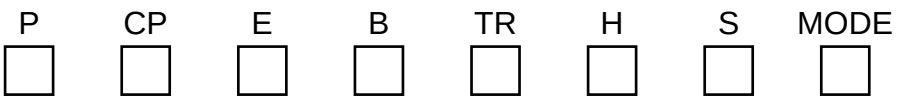
# Run-time support





**Registers**:



# Compilation of a clause

$h$ :- $b_1$,...,$b_n$.

h: allocate N
<***get*** code for h>
<***put*** code for $b_1$>
call $b_1$
...
<***put*** code for $b_n$>
call $b_n$
deallocate

**Example**:

append([A|L1],L2,[A|L3]) :- append(L1,L2,L3).

```
append: allocate        0
        get_list        A0
        unify_variable  X3
        unify_variable  X4
        get_list        A2
        unify_value     X3
        unify_variable  X5
        put_value       X4, A0
        put_value       X5, A2
        call            append
        deallocate
```

get code

put code

# WAM for ’Log

**Novel aspects**:

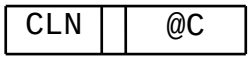
- heap representation of names
- constraint handling

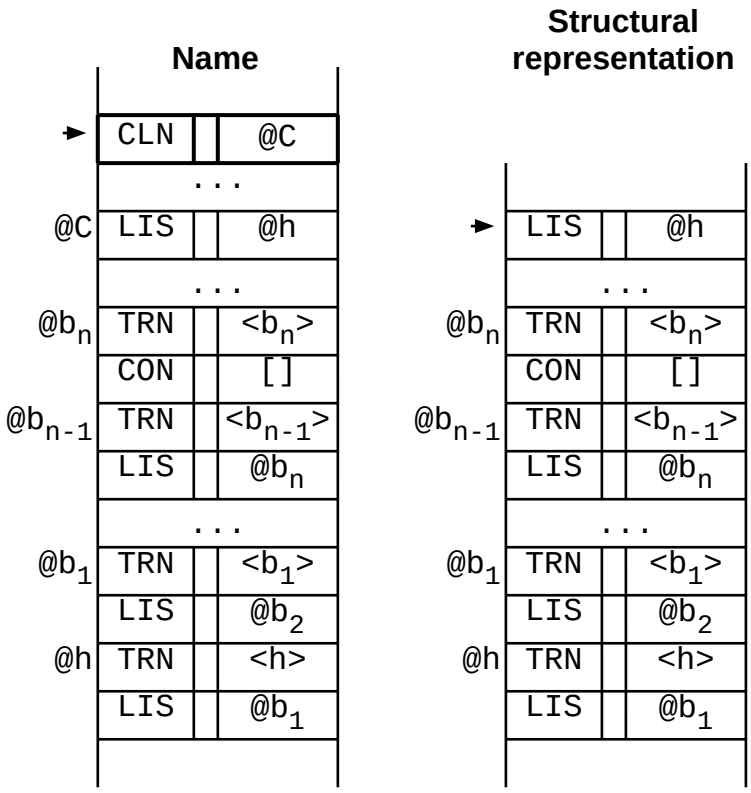
# Name representation

Names are represented as *marked structural representations*.

Clause

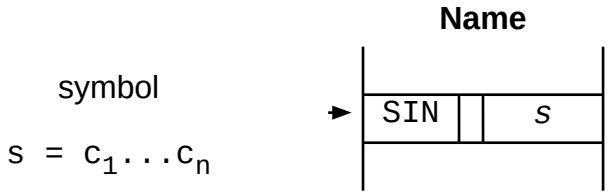
C = h:-$b_1$,..,$b_n$.



# Name representation (Cont'd)

Characters and *usually* symbols are represented *atomically*

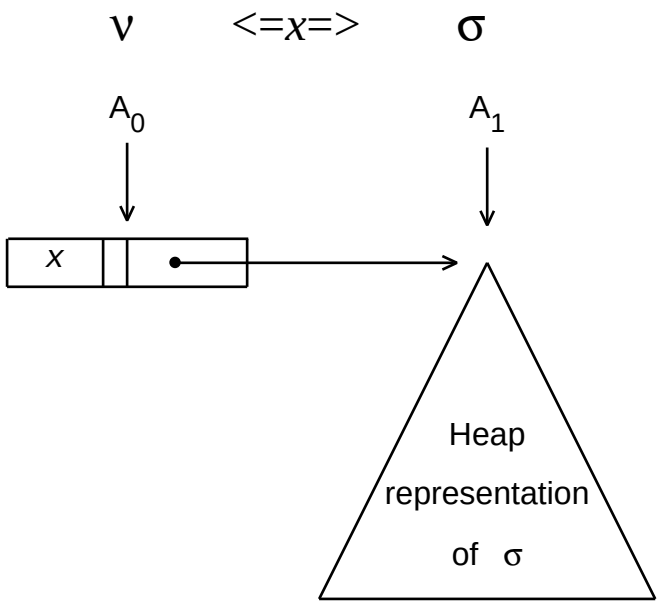



The atomic representation of symbol names complicates the unification algorithm

# Constraints

Compilation of $\nu$ <=***x***=> $\sigma$:

- **put** the structural representation $\sigma$ in $A_1$
- *create* an *x*-name cell in $A_0$
- **get** the name $\nu$ from $A_0$
- process the constraint



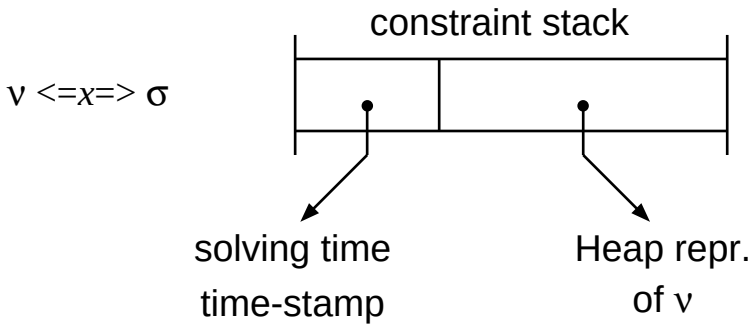
# Handling constraints

**Problems**:

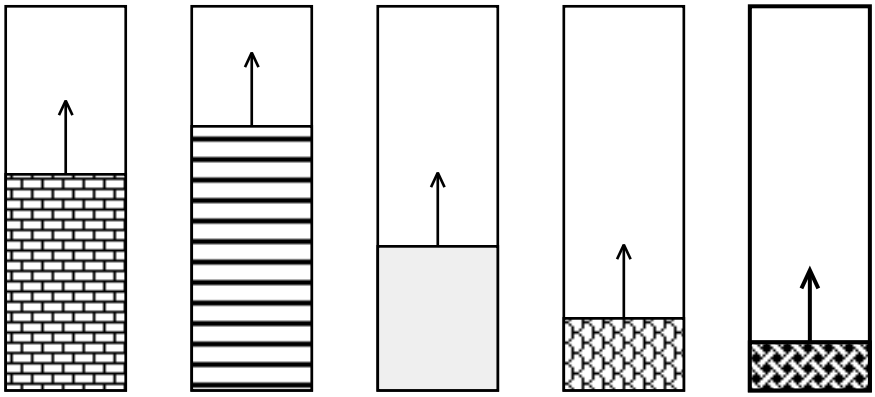
- constraints resolution is usually *delayed*
- *active constraints* must be kept track of
- unification can *solve* or *invalidate* an active constraint
- active constraints must be *checked* periodically for validity and satisfaction
- backtracking must *undo* constraint passivation

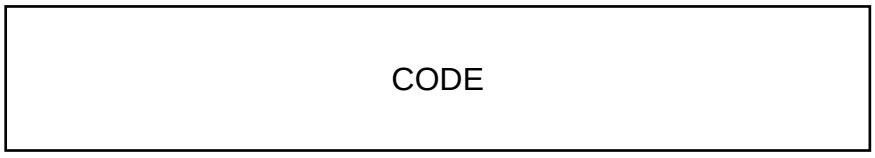
**Solutions**:

- keep track of *all* the constraints in a LIFO queue
- *mark* the solved constraints
- check an active constraints *when changes occur*
- use *chronological marking* for backtracking

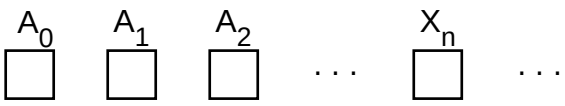


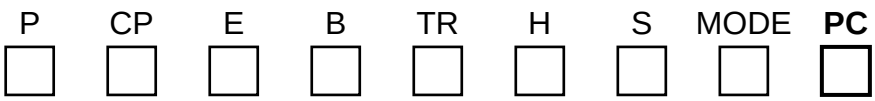
# Run-time support for constraint




**Registers**:



# Conclusions

- The WAM-based compilation and execution of ’Log have been analysed
- The WAM design has been modified to accomodate the new features of ’Log
- A programming environment for ’Log has been produced. It contains:
  - — a compiler
  - — an executer
  - — a WAM-based debugger
  - — miscellaneous tools
- The effiency of the system is satisfactory

# Further works

In the resulting system:

- use the system for testing the features of ’Log
- improve the efficiency
- hard-wire higher level meta-programming facilities

With ’Log:

- test several syntaxes
- add reflection
- hide <=*x*=>
- develop higher-level languages on top of it