Example:

\[
\begin{align*}
\text{bigger}(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Y) \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Z), \text{is_bigger}(Z, Y) \\
\text{? is_bigger(}\text{elephant, dog})
\end{align*}
\]

A More Realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [\ ], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \\
? & \text{app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

Remark:

\[
\begin{align*}
[\ ] & \text{ the atom empty list} \\
[H|Z] & \text{ binary constructor application} \\
[a, b, Z] & \text{ shortcut for: } [a][b][Z][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][]
A program $p$ is constructed as follows:

$$ t ::= a | X | f(t_1, \ldots, t_n) $$

$$ g ::= p(t_1, \ldots, t_k) | X = t $$

$$ c ::= p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_k $$

$$ p ::= c_1; \ldots; c_m; g $$

- A term $t$ is either an atom, a variable, an anonymous variable or a constructor application.
- A goal $g$ is either a literal, i.e., a predicate call, or a unification.
- A clause $c$ consists of a head $p(X_1, \ldots, X_n)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program $p$ consists of a sequence of clauses together with a single goal as a query.

### Procedural View of Proll programs:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>procedure call</td>
</tr>
<tr>
<td>predicate</td>
<td>procedure</td>
</tr>
<tr>
<td>clause</td>
<td>definition</td>
</tr>
<tr>
<td>term</td>
<td>value</td>
</tr>
<tr>
<td>unification</td>
<td>basic computation step</td>
</tr>
<tr>
<td>binding of variables</td>
<td>side effect</td>
</tr>
</tbody>
</table>

**Note:**
- Predicate calls ...
- ... do not have a return value.
- ... affect the caller through side effects only.
- ... may fail. Then the next definition is tried.

### Architecture of the WiM:

#### The Code Store:

- **C** = Code store – contains WiM program;
  every cell contains one instruction;
- **PC** = Program Counter – points to the next instruction to execute;
The Runtime Stack:

- **S** = Runtime Stack – every cell may contain a value or an address;
- **SP** = Stack Pointer – points to the topmost occupied cell;
- **FP** = Frame Pointer – points to the current stack frame.
  Frames are created for predicate calls, contain cells for each variable of the current clause.

The Heap:

- **H** = Heap for dynamically constructed terms;
- **HP** = Heap-Pointer – points to the first free cell;

- The heap is maintained like a stack as well.
- A `new-instruction` allocates an object in H.
- Objects are tagged with their types (as in the MaMa) ...

- **A** = atom
- **R** = variable
- **S** = unbound variable
- **structure** = (n+1) cells
28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment $\rho$ returns, for each clause variable $X$ its address (relative to $FP$) on the stack. Then $\text{code}_A t \rho$ should ... 

- construct a presentation of $t$ in the heap; and
- return a reference to it on top of the stack.

Idea:

- Construct the tree during a post-order traversal of $t$
- with one instruction for each new node!

Example: $t = f(g(X, Y), a, Z)$. Assume that $X$ is initialized, i.e., $[SFP + \rho X]$ contains already a reference, $Y$ and $Z$ are not yet initialized.

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $X$).

Note: Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_A a \rho &= \text{putatom} a \\
\text{code}_A f(t_1, \ldots, t_n) \rho &= \text{code}_A t_1 \rho \\
\text{code}_A X \rho &= \text{putvar} (\rho X) \\
\text{code}_A X \rho &= \text{putref} (\rho X) \\
\text{code}_A \_ \rho &= \text{putanon} \\
\end{align*}
\]

For $f(g(X, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ this results in the sequence:

- putref 1
- putvar 2
- putvar 3
- putstruct g/2
- putstruct f/3

Note: Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_A a \rho &= \text{putatom} a \\
\text{code}_A f(t_1, \ldots, t_n) \rho &= \text{code}_A t_1 \rho \\
\text{code}_A X \rho &= \text{putvar} (\rho X) \\
\text{code}_A X \rho &= \text{putref} (\rho X) \\
\text{code}_A \_ \rho &= \text{putanon} \\
\end{align*}
\]
\[ \rho(X) \leftarrow \pi(\bar{X}, Y) \]

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g., \( \bar{X} \)).

**Note:** Arguments are always initialized!

Then we define:

\[
\begin{align*}
\text{code}_{A} a \rho &= \text{putatom} a \\
\text{code}_{A} f(t_{1}, \ldots, t_{n}) \rho &= \text{code}_{A} t_{1} \rho \\
\text{code}_{A} \bar{X} \rho &= \text{putvar} (\rho X) \\
\text{code}_{A} \bar{X} \rho &= \text{putref} (\rho X) \\
\text{code}_{A} \bar{\rho} \rho &= \text{putanon} \\
& \quad \text{putstruct } t/n
\end{align*}
\]

For \( f(g(\bar{X}, Y), a, Z) \) and \( \rho = \{ X \rightarrow 1, Y \rightarrow 2, Z \rightarrow 3 \} \) this results in the sequence:

- putref 1
- putvar 2
- putvar 3
- putstruct g/2
- putstruct t/3

The instruction \( \text{putvar } i \) introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:

\[ \text{SP} = \text{SP} + 1; \]
\[ S[\text{SP}] = \text{new } (R, \text{HP}); \]
\[ S[\text{FP} + 1] = S[\text{SP}]; \]

The instruction \( \text{putanon} \) introduces a new unbound variable but does not store a reference to it in the stack frame:

\[ \text{SP} = \text{SP} + 1; \]
\[ S[\text{SP}] = \text{new } (R, \text{HP}); \]
The instruction `putref i` pushes the value of the variable onto the stack:

```plaintext
SP = SP + 1;
S[SP] = deref S[FP + i];
```

The auxiliary function `deref` contracts chains of references:

```plaintext
ref deref (ref v) {
    if (H[v]==(r,v) && v!=u) return deref (v);
    else return s;
}
```

The instruction `putstruct f/n` builds a constructor application in the heap:

```plaintext
v = new (S, f, n);
SP = SP - n + 1;
for (i=1; i<=n; ++i)
    H[v+i] = S[SP+i-1];
S[SP] = v;
```
29 The Translation of Literals (Goals)

Idea:
- Literals are treated as procedure calls.
- We first allocate a stack frame.
- Then we construct the actual parameters (in the heap).
- ... and store references to these into the stack frame.
- Finally, we jump to the code for the procedure/predicate.

$\text{code}_C \ p(t_1, \ldots, t_k) \ \rho = \begin{cases} 
\text{mark } B \\
\text{code}_A \ t_1 \ \rho \\
\vdots \\
\text{code}_A \ t_k \ \rho \\
\text{call } p/k
\end{cases}$ // allocates the stack frame

B: ... // calls the procedure p/k

Example: $p(a, X, g(X, Y))$ with $\rho = \{X \mapsto 1, Y \mapsto 2\}$
We obtain:

- mark B
- putref 1
- putatom a
- putvar 1
- call p/3
- putvar 2
- B: ...
- putstruct g/2

Stack Frame of the WiM:

- SP
- FP
- posCont.
- FPold
- local stack
- local variables
- 6 org. cells
Remarks:

- The positive continuation address records where to continue after successful treatment of the goal.
- Additional organizational cells are needed for the implementation of backtracking will be discussed at the translation of predicates.

Stack Frame of the WiM:

![Diagram of stack frame with SP, FP, positive continuation, and local variables.]

6 org. cells

local stack

local variables

The instruction \textit{mark B} allocates a new stack frame:

\[ SP = SP + 6; \]
\[ S[SP] = B; S[SP-1] = FP; \]
30 Unification

Convention:

- By \( \mathcal{X} \), we denote an occurrence of \( X \); it will be translated differently depending on whether the variable is initialized or not.
- We introduce the macro \( \text{put} \mathcal{X} \rho : \)
  
  \[
  \begin{align*}
  \text{put} X \rho &= \text{putvar} (\rho X) \\
  \text{put} \_ \rho &= \text{putanon} \\
  \text{put} \mathcal{X} \rho &= \text{putref} (\rho X)
  \end{align*}
  \]

\( \chi = a \)
Let us translate the unification \( X = t \).

Idea 1:
- Push a reference to (the binding of) \( X \) onto the stack;
- Construct the term \( t \) in the heap;
- Invent a new instruction implementing the unification \( \Rightarrow \).
Example:

Consider the equation:

$$U = f(g(X, Y), a, Z)$$

Then we obtain for an address environment

$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

```
putref 4  putref 1  putatom a  unify
putvar 2  putvar 3
putstruct g/2  putstruct f/3
```

The instruction `unify` calls the run-time function `unify()` for the
topmost two references:

```
unify ([SP-1], [SP]);
SP = SP-2;
```

```
bool unify (ref u, ref v) {
  if (u == v) return true;
  if (H[u] == (R, _)) {
    if (H[v] == (R, _)) {
      if (u==v) {
        H[u] = (R, v); trail (u); return true;
      } else {
        H[v] = (R, u); trail (v); return true;
      }
    } elseif (check (u,v)) {
      H[u] = (R, v); trail (u); return true;
    } else {
      backtrack(); return false;
    }
  }
  ...
```
if (H[v] == (n, _)) {
    if (check (v,u)) {
        H[v] = (n,u); trail (v); return true;
    } else {
        backtrack(); return false;
    }
} else {
    if (H[u] == (A,a) && H[v] == (A,a))
        return true;
    if (H[u] == (n, f/n) && H[v] == (n, f/n)) {
        for (int i=1; i<=n; i++)
            if (unify (deref (H[u+i]), deref (H[v+i]))) return false;
        backtrack(); return true;
    } else {
        backtrack(); return false;
    }
}
- The run-time function `trail()` records the creation of a new binding.
- The run-time function `backtrack()` initiates backtracking.
- The auxiliary function `check()` performs the occur-check: it tests whether a variable (the first argument) occurs inside a term (the second argument).
- Often, this check is skipped, i.e.,

```c
bool check (ref u, ref v) {
    if (u == v) return false;
    if (H[v] == ($, 1/u)) {
        for (int i=1; i<=n; i++)
            if (!check(b, deref (H[v+i])))
                return false;
    }
    return true;
}
```
Discussion:

- The translation of an equation $\bar{x} = t$ is very simple $\Rightarrow$
- Often the constructed cells immediately become garbage $\Rightarrow$

Idea 2:

- Push a reference to the run-time binding of the left-hand side onto the stack.
- Avoid to construct sub-terms of $t$ whenever possible!
- Translate each node of $t$ into an instruction which performs the unification with this node!!