Example:

The following well-known function computes the factorial of a natural number:

```ocaml
letrec fac = fn x → if x ≤ 1 then 1
                      else x · fac (x - 1)

in fac 7
```

As usual, we only use the minimal amount of parentheses.

There are two Semantics:

**CBV**: Arguments are evaluated before they are passed to the function (as in SML);

**CBN**: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

---

12 Architecture of the MaMa:

We know already the following components:

- **C** = Code-store – contains the MaMa-program; each cell contains one instruction;
- **PC** = Program Counter – points to the instruction to be executed next;
S = Runtime-Stack – each cell can hold a basic value or an address;
SP = Stack-Pointer – points to the topmost occupied cell;
as in the CMs implicitly represented;
FP = Frame-Pointer – points to the actual stack frame.

... it can be thought of as an abstract data type, being capable of holding data
objects of the following form:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td>173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Basic Value

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Closure

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Function

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

We also need a heap H:

The instruction new (tag, args) creates a corresponding object (B, C, F, V) in H
and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- codev e — (generates code that) computes the Value of e, stores it in the
  heap and returns a reference to it on top of the stack (the normal case);
- codeg e — computes the value of e, and returns it on the top of the stack
  (only for Basic types);
- codec e — does not evaluate e, but stores a Closure of e in the heap and
  returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:
13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

\[
\begin{align*}
\text{code}_3 \ b \ p \ s d &= \text{load} \ b \\
\text{code}_3 (\oplus_1 \ e) \ p \ s d &= \text{code}_3 e \ p \ s d \\
& \quad \text{op}_1 \\
\text{code}_3 (e_1 \ \ominus_2 e_2) \ p \ s d &= \text{code}_3 e_1 \ p \ s d \\
& \quad \text{code}_3 e_2 \ p \ (s d + 1) \\
& \quad \text{op}_2
\end{align*}
\]

Note:

- \( \rho \) denotes the actual address environment, in which the expression is translated.
- The extra argument \( s d \), the stack difference, simulates the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.
- The instructions \( \text{op}_1 \) and \( \text{op}_2 \) implement the operators \( \oplus_1 \) and \( \ominus_2 \), in the same way as the operators \( \text{neg} \) and \( \text{add} \) implement negation resp. addition in the CMAs.
- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

\[
\begin{align*}
\text{code}_3 \ e \ p \ s d &= \text{code}_3 e \ p \ s d \\
& \quad \text{get basic}
\end{align*}
\]

\[
\text{code}_3 (\text{if } e_0 \ \text{then } e_1 \ \text{else } e_2) \ p \ s d =
\begin{align*}
\text{jump} \ A \\
\text{code}_3 e_0 \ p \ s d \\
\text{jump} \ B \\
A: \text{code}_3 e_2 \ p \ s d \\
B: \ ...
\end{align*}
\]
For code\(\gamma\) and simple expressions, we define analogously:

\[
\begin{align*}
\text{code}_{\gamma} b \rho \sigma d &= \text{loadc by mkbasic} \\
\text{code}_{\gamma} (\mathbb{2} e) \rho \sigma d &= \text{code}_{\gamma} e \rho \sigma d \\
\text{op}_{\gamma}: \text{mkbasic} \\
\text{code}_{\gamma} (e_1 \sqcap e_2) \rho \sigma d &= \text{code}_{\gamma} e_1 \rho \sigma d \\
&\quad \text{code}_{\gamma} e_2 \rho (\sigma d + 1) \\
\text{op}_{\gamma}: \text{mkbasic} \\
\text{code}_{\gamma} (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \sigma d &= \text{code}_{\gamma} e_0 \rho \sigma d \\
&\quad \text{jump } A \\
&\quad \text{code}_{\gamma} e_1 \rho \sigma d \\
&\quad \text{jump } B \\
&\quad A: \text{ code}_{\gamma} e_2 \rho \sigma d \\
&\quad B: \ldots
\end{align*}
\]

Also:

\[
\begin{align*}
\text{mkbasic} &\quad \text{new } (B, \text{or}) \\
S[SP] &= \text{new } (B, S[SP])
\end{align*}
\]
14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function $f$:

```plaintext
let $c = 5$

\[ f = \text{fn } a \Rightarrow \text{let } b = a \times a \]
\[ \text{in } b = c \]
```

The function $f$ uses the global variable $c$ and the local variables $a$ (as formal parameter) and $b$ (introduced by the inner let).

The binding of a global variable is determined, when the function is constructed (static scoping!), and later only looked up.

---

Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the -component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In constrast, local variables should be administered on the stack ...
Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the -component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In contrast, local variables should be administered on the stack ...

\[ \rho : \text{Vars} \rightarrow \{L, G\} \]

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

Let \( e' \left[ e_0 \ldots e_{m-1} \right] \) be the application of a function \( e' \) to arguments \( e_0, \ldots, e_{m-1} \).

Warning:

The arity of \( e' \) does not need to be \( m \) :-)

- \( f \) may therefore receive less than \( n \) arguments (under supply);
- \( f \) may also receive more than \( n \) arguments, if \( f \) is a functional type (over supply).

Possible stack organisations:

- Addressing of the arguments can be done relative to FP
- The local variables of \( e' \) cannot be addressed relative to FP.
- If \( e' \) is an \( n \)-ary function with \( n < m \), i.e., we have an over-supplied function application, the remaining \( m - n \) arguments will have to be shifted.
Possible stack organisations:

+ Addressing of the arguments can be done relative to FP
  - The local variables of \( \epsilon' \) cannot be addressed relative to FP.
  - If \( \epsilon' \) is an \( n \)-ary function with \( n < m \), i.e., we have an over-supplied function application, the remaining \( m - n \) arguments will have to be shifted.

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

Let \( \epsilon \equiv \epsilon_0 \ldots \epsilon_{m-1} \) be the application of a function \( \epsilon' \) to arguments \( \epsilon_0, \ldots, \epsilon_{m-1} \).

Warning:

The arity of \( \epsilon' \) does not need to be \( m \) :-)

- \( f \) may therefore receive less than \( n \) arguments (under supply);
- \( f \) may also receive more than \( n \) arguments, if \( t \) is a functional type (over supply).
Alternative:

+ The further arguments $a_0, \ldots, a_{n-1}$ and the local variables can be allocated above the arguments.

Way out:

- We address both, arguments and local variables, relative to the stack pointer SP. 
  "!!!"

- However, the stack pointer changes during program execution...

- Addressing of arguments and local variables relative to FP is no more possible. (Remember: $n$ is unknown when the function definition is translated.)
• The difference between the current value of SP and its value \( sp_0 \) at the entry of the function body is called the stack distance, \( sd \).

• Fortunately, this stack distance can be determined at compile time for each program point, by simulating the movement of the SP.

• The formal parameters \( x_0, x_1, x_2, \ldots \) successively receive the non-positive relative addresses \( 0, -1, -2, \ldots \), i.e., \( \rho x_i = (L, -i) \).

• The absolute address of the \( i \)-th formal parameter consequently is 
  \[ sp_0 - i = (SP - sd) - i \]

• The local let-variables \( y_1, y_2, y_3, \ldots \) will be successively pushed onto the stack:

With CBN, we generate for the access to a variable:

\[
\text{code} \ x \ \rho \ sd = \ \text{getvar} \ x \ \rho \ sd
\]

\[
\text{eval}
\]

The instruction \text{eval} checks, whether the value has already been computed or whether its evaluation has to yet to be done \( \text{------} \) will be treated later \( \text{-----} \).

With CBV, we can just delete \text{eval} from the above code schema.

The (compile-time) macro \text{getvar} is defined by:

\[
\text{getvar} \ x \ \rho \ sd = \ \text{let} \ (t, i) = \rho x \ \text{in}
\]

\[
\begin{align*}
\text{case} \ t \ \text{of} \\
L & \Rightarrow \text{pushloc} \ (sd - i) \\
G & \Rightarrow \text{pushglob} \ i
\end{align*}
\]

end
With CBN, we generate for the access to a variable:

\[
\text{code} \ x \ p \ sd = \ \text{getvar} \ x \ p \ sd\ \text{eval}
\]

The instruction \( \text{eval} \) checks whether the value has already been computed or whether its evaluation has to yet to be done \( \ldots \) will be treated later \( \ldots \)

With CBV, we can just delete \( \text{eval} \) from the above code schema.

The (compile-time) macro \( \text{getvar} \) is defined by:

\[
\text{getvar} \ x \ p \ sd = \ \text{let} \ (t, i) = p \ x \ \text{in} \ \\
\text{case} \ t \ \text{of} \ \\
L \Rightarrow \ \text{pushloc} \ (sd - i) \ \\
G \Rightarrow \ \text{pushglob} \ i \\
\text{end}
\]

Correctness argument:

Let \( sp \) and \( sd \) be the values of the stack pointer resp. stack distance before the execution of the instruction. The value of the local variable with address \( i \) is loaded from \( S[a] \) with

\[
a = sp - (sd - i) = (sp - sd) + i = sp_0 + i
\]

... exactly as it should be \( \therefore \)
The access to global variables is much simpler:

\[
\begin{align*}
\text{SP} &= \text{SP} + 1; \\
S[\text{SP}] &= \text{GP} \cdot v[i];
\end{align*}
\]

Example:

\[
\begin{align*}
\text{code} e \rho 1 &= \text{getvar } e \rho 1 = 1 \text{ pushloc } 0 \\
&\quad \text{eval} 2 \text{ eval} \\
&\quad \text{getbasic} 2 \text{ getbasic} \\
&\quad \text{getvar } e \rho 2 = 2 \text{ pushglob } 0 \\
&\quad \text{eval} 3 \text{ eval} \\
&\quad \text{getbasic} 3 \text{ getbasic} \\
&\quad \text{add} 3 \text{ add} \\
&\quad \text{mbkbasic} 2 \text{ mbkbasic}
\end{align*}
\]