\_fac: \quad \text{enter } q \quad \text{loadc } 1 \quad \text{A: loadr } -3 \quad \text{mul} \\
\quad \text{alloc } 0 \quad \text{storer } -3 \quad \text{loadr } -3 \quad \text{storer } -3 \\
\quad \text{loadr } -3 \quad \text{return} \quad \text{loadr } -3 \quad \text{return} \\
\quad \text{loadc } 0 \quad \text{return} \quad \text{loadc } 1 \quad \text{return} \\
\quad \text{leq} \quad \text{sub} \quad \text{mark} \\
\quad \text{jumpz A} \quad \text{jump B} \quad \text{loadc _fac} \\
\quad \text{call} \quad \text{slide } 0 \quad \text{call} \\

\text{where } \rho_{\text{fac}} : x \mapsto (L, -3) \quad \text{and} \quad q = 5.
10 Translation of Whole Programs

Before program execution, we have:

\[
\begin{align*}
SP &= -1 \\
FP &= EP = -1 \\
PC &= 0 \\
NP &= \text{MAX}
\end{align*}
\]

Let \( p \equiv V_{\text{defs}} \; F_{\text{def}}_1 \; \ldots \; F_{\text{def}}_n \), denote a program where \( F_{\text{def}}_i \) is the definition of a function \( f_i \) of which one is called \( \text{main} \).

The code for the program \( p \) consists of:

- code for the function definitions \( F_{\text{def}}_i \);
- code for the allocation of global variables;
- code for the call of \( \text{int main()} \);
- the instruction \( \text{halt} \) which returns control to the operating system together with the value at address 0.
Then we define:

\[
\text{code } p \emptyset = \text{enter } (k + 4) \\
\text{alloc } (k + 1) \\
\text{mark} \\
\text{loadc } \_\text{main} \\
\text{call} \\
\text{slide } k \\
\text{halt} \\
\_f_1: \text{code } F\_\text{def}_1 \rho \\
\_f_n: \text{code } F\_\text{def}_n \rho
\]

where \( \emptyset \triangleq \text{empty address environment;} \)
\( \rho \triangleq \text{global address environment;} \)
\( k \triangleq \text{size of the global variables} \)
The Translation of Functional Programming Languages
11 The language PuF

We only regard a mini-language PuF (“Pure Functions”). We do not treat, as yet:

- Side effects;
- Data structures.
A program is an expression $e$ of the form:

$$e ::= b \mid x \mid (\Box_1 e) \mid (e_1 \Box_2 e_2)$$
$$\mid (\text{if } e_0 \text{ then } e_1 \text{ else } e_2)$$
$$\mid (e' e_0 \ldots e_{k-1})$$
$$\mid (\text{fun } x_0 \ldots x_{k-1} \to e)$$
$$\mid (\text{let } x_1 = e_1 \text{ in } e_0)$$
$$\mid (\text{let rec } x_1 = e_1 \text{ and } \ldots \text{ and } x_n = e_n \text{ in } e_0)$$

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a let-expression, i.e. an expression with locally defined variables, or
- a let-rec-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow $\text{int}$ as basic type.
Example:

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

\[
\text{let rec } \text{fac } \ = \ \text{fun } x \rightarrow \text{if } x \leq 1 \text{ then } 1 \\
\text{else } x \cdot \text{fac } (x - 1)
\]

\text{in} \ \text{fac} \ 7

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

There are two Semantics:

\textbf{CBV}: Arguments are evaluated \textit{before} they are passed to the function (as in SML);

\textbf{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:

\[ C \]

- **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 CMa implicitly represented;
FP = Frame-Pointer – points to the actual stack frame.
We also need a heap $H$: 

![Diagram of heap structure]
... it can be thought of as an **abstract data type**, being capable of holding data objects of the following form:

\[
\begin{align*}
\text{v} & \\
\text{B} & \quad -173 \\
\text{cp} & \quad \text{gp} \\
\text{C} & \\
\text{cp} & \quad \text{ap} & \quad \text{gp} \\
\text{F} & \\
\text{v}[0] & \quad \cdots & \quad \text{v}[n-1] \\
\text{V} & \quad n & \\
\end{align*}
\]
The instruction \texttt{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 $e$:

- \texttt{code}_V\ e — (generates code that) computes the \texttt{Value} of $e$, stores it in the heap and returns a reference to it on top of the stack (the normal case);

- \texttt{code}_B\ e — computes the value of $e$, and returns it on the top of the stack (only for Basic types);

- \texttt{code}_C\ 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:

\[
\begin{align*}
\text{code}_B b \rho \text{ sd} &= \text{loadc } b \\
\text{code}_B (\square_1 e) \rho \text{ sd} &= \text{code}_B e \rho \text{ sd} \\
&\quad \text{op}_1 \\
\text{code}_B (e_1 \square_2 e_2) \rho \text{ sd} &= \text{code}_B e_1 \rho \text{ sd} \\
&\quad \text{code}_B e_2 \rho \text{ (sd + 1) } \\
&\quad \text{op}_2
\end{align*}
\]
\[ \text{code}_B (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \text{ sd} = \text{code}_B e_0 \rho \text{ sd} \]

jumpz A
\[ \text{code}_B e_1 \rho \text{ sd} \]

jump B
\[ \text{code}_B e_2 \rho \text{ sd} \]

A: \[ \text{...} \]

B: \[ \text{...} \]
Note:

- $\rho$ denotes the actual **address environment**, in which the expression is translated.

- The extra argument $sd$, the **stack difference**, *simulates* the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.

- The instructions $op_1$ and $op_2$ implement the operators $\Box_1$ and $\Box_2$, in the same way as the the operators $\text{neg}$ and $\text{add}$ implement negation resp. addition in the CMa.

- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

  $$\text{code}_B \ e \ \rho \ sd \ = \ \text{code}_V \ e \ \rho \ sd$$

  $$\text{getbasic}$$
if (H[S[SP]] != (B, _))
    Error "not basic!";
else
    S[SP] = H[S[SP]].v;
For $\text{code}_V$ and simple expressions, we define analogously:

$$
\text{code}_V b \, \rho \, \text{sd} = \text{loadc b; mkbasic}
$$

$$
\text{code}_V (\, \square_1 \, e \,) \, \rho \, \text{sd} = \text{code}_B e \, \rho \, \text{sd}
$$

$$
\text{code}_V (e_1 \, \square_2 \, e_2) \, \rho \, \text{sd} = \text{code}_B e_1 \, \rho \, \text{sd}
$$

$$
\text{code}_B e_2 \, \rho \, (\text{sd} + 1)
$$

$$
\text{code}_V (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \, \rho \, \text{sd} = \text{code}_B e_0 \, \rho \, \text{sd}
$$

$$
\text{jumpz A}
$$

$$
\text{code}_V e_1 \, \rho \, \text{sd}
$$

$$
\text{jump A}
$$

$$
A: \text{code}_V e_2 \, \rho \, \text{sd}
$$

$$
B: \ldots
$$
S[SP] = new (B,S[SP]);
14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function \( f \):

\[
\text{let } c = 5 \\
\text{in let } f = \text{fun } a \rightarrow \text{let } b = a \times a \text{ in } b + c \\
\text{in } f \ c
\]

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.