Accordingly, we translate a function definition:

```
code f(specs){V_defs ss} \rho =
  \_f: enter q // initialize EP
  alloc k // allocate the local variables
  code ss \rho_f
  return // return from call
```

where

- \( q = max + k \) with
- \( max \) = maximal length of the local stack
- \( k \) = size of the local variables
- \( \rho_f \) = address environment for \( f \)
  // takes \textit{specs}, \textit{V_defs} and \( \rho \) into account

The instruction \textbf{enter q} sets the EP to the new value. If not enough space is available, program execution terminates.

The instruction \textbf{call} saves the return address and sets FP and PC onto the new values.
Accordingly, we obtain for a call to a function with at least one parameter and one return value:

\[
\text{code}_k \{ g(e_1, \ldots, e_m) \rho \} = \text{code}_k e_1 \rho \\
... \\
\text{code}_k e_m \rho \\
\text{mark} \\
\text{call} \\
\text{slide} (m - 1)
\]

where \( m \) is the size of the actual parameters.

The instruction \return{} pops the current stack frame. This means it restores the registers \text{PC}, \text{EP} and \text{FP} and returns the return value on top of the stack.

9.4 Access to Variables, Formal Parameters and Returning of Values

Accesses to local variables or formal parameters are relative to the current \text{FP}. Accordingly, we modify \text{code}_e \{ \} for names of variables.

For \( \rho x = (\text{tag}, j) \) we define

\[
\text{code}_k x \rho = \begin{cases} 
\text{load}_c j & \text{tag} = G \\
\text{loadrc} j & \text{tag} = L
\end{cases}
\]
The instruction `loadrc j` computes the sum of FP and `j`.

```
FP f
load rc j
FP f j+
SP++; S[SP] = FP+j.
```

As an optimization, we introduce analogously to `loada j` and `storea j` the new instructions `load r j` and `store r j`:

```
load r j = loadrc j
load
store r j = loadrc j;
store
```

The code for `return c` corresponds to an assignment to a variable with relative address −3.

```
code return c; r = code re c
     store -3
     return
```

**Example** For function

```c
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac (x − 1);
}
```

we generate:

```
- fac: enter q
    enter alloc 0
    storer -3
    loadr -3
    return
    load 1
    sub
    return
    mark
    load -4c
    call
    slide 0
```

where \( \rho_{le} : x \mapsto (L, -3) \) and \( q = 5 \).

\[ 0 + 1 + 1 + 3 = 5 \]
10 Translation of Whole Programs

Before program execution, we have:

\[
SP = -1 \quad FP = EP = -1 \quad PC = 0 \quad NP = \text{MAX}
\]

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:

\[
\begin{align*}
\text{code } p \emptyset &= \text{enter } (k + 4) \\
&\quad \text{alloc } (k + 1) \\
&\quad \text{mark} \\
&\quad \text{load: } \_\text{main} \\
&\quad \text{call} \\
&\quad \text{slide } k \\
&\quad \text{halt} \\
\_f_1\colon &\text{ code } F_{\text{def}_1} \rho \\
\vdots \\
\_f_n\colon &\text{ code } F_{\text{def}_n} \rho
\end{align*}
\]

where \( \emptyset \equiv \text{empty address environment}; \rho \equiv \text{global address environment}; k \equiv \text{size of the global variables} \)
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:

$$ e ::= b ~|~ x ~|~ (\begin{array}{c} e_1 \circ e_2 \end{array}) ~|~ (e_1 \circ e_2) $$

- (if $e_1$ then $e_2$ else $e_3$)
- (let $e_1$ in $e_2$)
- (let rec $x_1 = e_1$ and ... and $x_n = e_n$ 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 int as basic type.
Example:

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

```plaintext
let rec fac x = if x ≤ 1 then 1
  else x · fac (x - 1)
```

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).

---

A program is an expression \( e \) of the form:

\[
\begin{align*}
  e &::= b \mid \mathit{x} \mid (e_1 \mathbin{\&} e_2) \mid (e_1 \mathbin{\|} e_2) \\
  & \mid (\mathit{if} \ e_1 \ \mathit{then} \ e_2 \ \mathit{else} \ e_3) \\
  & \mid (\mathit{let} \ \mathit{x}_1 = e_1 \ \mathbin{\text{in}} \ e_2) \\
  & \mid (\mathit{let} \ \mathit{rec} \ \mathit{x}_1 = e_1 \ \mathbin{\text{and}} \ldots \mathbin{\text{and}} \ \mathit{x}_n = e_n \ \mathbin{\text{in}} \ e_0)
\end{align*}
\]

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 \( \mathit{int} \) as basic type.

---

Example:

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

```plaintext
let rec fac = fun x → if x ≤ 1 then 1
  else x · fac (x - 1)
```

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;

```
C = 0 1               PC

C = Code-store – contains the MaMa-program;
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:

v

B \[ -173 \]

Basic Value

C

cp gp

Closure

F

cp ap gp

Function

V n

\( v[0] \) \ldots \( v[n-1] \)

Vector

We also need a heap \( H \):

- Tag
- Code Pointer
- Value
- Heap Pointer