The code for \( \text{return } c; \) corresponds to an assignment to a variable with relative address \(-3\).

\[
\begin{align*}
\text{code } & \text{return } c; \rho = \text{code } \varepsilon \rho \\
& \quad \text{storer } -3 \\
& \quad \text{return}
\end{align*}
\]

**Example** For function

\[
\begin{align*}
\text{int } \text{fac} \ (\text{int } x) \{ \\
& \quad \text{if } (x \leq 0) \text{ return } 1; \\
& \quad \quad \text{else return } x \times \text{fac} \ (x - 1); \\
& \}
\end{align*}
\]

we generate:

\[
\begin{align*}
\text{A: } & \text{enter } q \\
& \text{alocr } 0 \\
& \text{loadr } -3 \\
& \text{return} \\
& \text{load } 1 \\
& \text{sub} \\
& \text{mark} \\
& \text{loadr } _{\text{fac}} \\
& \text{call} \\
& \text{slide } 0
\end{align*}
\]

\[
\begin{align*}
\text{B: } & \text{mul} \\
& \text{storer } -3 \\
& \text{return} \\
& \text{loadr } -3 \\
& \text{return} \\
& \text{load } 1 \\
& \text{mark} \\
& \text{load } 0 \\
& \text{jump } \text{B} \\
& \text{sub} \\
& \text{mark} \\
& \text{loadr } _{\text{fac}} \\
& \text{call} \\
& \text{slide } 0
\end{align*}
\]

where \( \rho_{\text{fac}} \colon x \mapsto (L, -3) \) and \( q = 1 + 5 = 6 \).
10 Translation of Whole Programs

Before program execution, we have:

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

Let \( p \equiv V_{\text{defs}} \cdot F_{\text{def}_{1}} \cdot \ldots \cdot F_{\text{def}_{n}} \cdot \) 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 \( F_{\text{def}} \);
- code for the allocation of global variables;
- code for the call \( \text{von} \ \text{main}() \);
- the instruction \( \text{halt} \).

Then we define:

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

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

\[
\text{code } p \varnothing = \begin{array}{l}
\text{enter } (k+5) \\
\text{alloc } (k+1) \\
\text{mark} \\
\text{loadc\_main} \\
\text{call} \\
\text{slide } (k) \\
\text{halt} \\
_{f_1}: \text{code } F\_\text{def}_1 \rho \\
\vdots \\
_{f_n}: \text{code } F\_\text{def}_n \rho 
\end{array}
\]

where $\varnothing \equiv \text{empty address environment;}$
$\rho \equiv \text{global address environment;}$
$k \equiv \text{size of the global variables}$

---

10 Translation of Whole Programs

Before program execution, we have:

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

Let \( p \equiv V\_\text{def} \ F\_\text{def}_1 \ldots F\_\text{def}_n \) 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 \( \text{von \ main()} \);
- the instruction \( \text{halt} \).
The Translation of Functional Programming Languages

11. The Language PFP

We only regard a mini-language PFP (Pure Functions).

Example:

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

```
let rec fac = fun x -> if x = 0 then 1 else x * fac(x - 1)
```

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

CPS: Arguments are passed uncalculated; they are only evaluated when their value is needed (as in Haskell).
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;

---

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).

---

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 CMAs implicitly represented;

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

- **Tag**
- **Code Pointer**
- **Value**
- **Heap Pointer**

**S**

- **SP** = Runtime Stack – each cell can hold a basic value or an address;
- **FP** = Stack Pointer – points to the topmost occupied cell;
  - as in the CMs explicitly represented;

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

```
     v
    ---
   |   |
   B -173 |
   ---
     cp  gp
     ---
     C
     ---
   |   |
   cp ap gp
   ---
     ---
     F
     ---
   v[0] ....... v[n-1]
   ---
     ---
     V
```
The instruction \textit{new} (\texttt{arg, args}) creates a corresponding object (B, C, E, V) in H and returns a reference to it.

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

- \texttt{code}_1 \ e \quad \text{(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)};
- \texttt{code}_2 \ e \quad \text{computes the value of \( e \), and returns it on the top of the stack (only for Basic types)};
- \texttt{code}_3 \ e \quad \text{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.

\section{Simple expressions}

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

\[
\begin{align*}
\texttt{code}_3 \ b \ & \rho \ s d = \text{load} b \\
\texttt{code}_3 \ (\circ_1 \ e) \ & \rho \ s d = \texttt{code}_3 \ e \ \rho \ s d \quad \text{if} \ \circ_1 \\
\texttt{code}_3 \ (e_1 \ \circ_2 \ e_2) \ & \rho \ s d = \texttt{code}_3 \ e_1 \ \rho \ s d \\
& \texttt{code}_3 \ e_2 \ \rho \ (sd + 1) \\
& \circ_2 \\
\end{align*}
\]

The instruction \textit{new} (\texttt{arg, args}) creates a corresponding object (B, C, E, V) in H and returns a reference to it.

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

- \texttt{code}_1 \ e \quad \text{(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)};
- \texttt{code}_2 \ e \quad \text{computes the value of \( e \), and returns it on the top of the stack (only for Basic types)};
- \texttt{code}_3 \ e \quad \text{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.
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 $\oplus_1$ and $\oplus_2$, in the same way as the the operators neg and 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:

\[
\text{code}_e \; \rho \; sd = \text{code}_v \; \rho \; sd \\
\begin{array}{l}
\text{getbasic}
\end{array}
\]

13 Simple expressions

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

\[
\begin{array}{l}
\text{code}_b \; \rho \; sd = \text{loadc} \; b \\
\text{code}_v \; (\oplus_1 \; e) \; \rho \; sd = \text{code}_e \; \rho \; sd \\
\text{code}_v \; (e_1 \; \oplus_2 \; e_2) \; \rho \; sd = \text{code}_e \; e_1 \; \rho \; sd \\
\text{code}_v \; e_2 \; \rho \; (sd + 1) = \text{code}_e \; e_2 \; \rho \; (sd + 1) \\
\end{array}
\]

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 $\oplus_1$ and $\oplus_2$, in the same way as the the operators neg and 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:

\[
\text{code}_e \; \rho \; sd = \text{code}_v \; \rho \; sd \\
\begin{array}{l}
\text{getbasic}
\end{array}
\]

\[\text{if } (H[SP] = B_.) \]
\[\text{else} \]
\[S[SP] = H[SP] \triangleright; \]
\[\text{getbasic} \]
\[\text{IT} \]
14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function $f$:

\[
\begin{align*}
\text{let } & \quad c = 5 \\
\text{in let } & \quad f = \text{fun } a \\
& \quad \text{let } b = a + c \\
\text{in } & \quad f c
\end{align*}
\]

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.