9 Functions

The definition of a function consists of

- a name, by which it can be called,
- a specification of the formal parameters;
- maybe a result type;
- a statement part, the body.

For C holds:

$$\text{code}_{\text{R}}\ f\ \rho \quad = \quad \text{loadc}\ \_f \quad = \quad \text{starting address of the code for } f$$

$\Longrightarrow$ Function names must also be managed in the address environment!

Example:

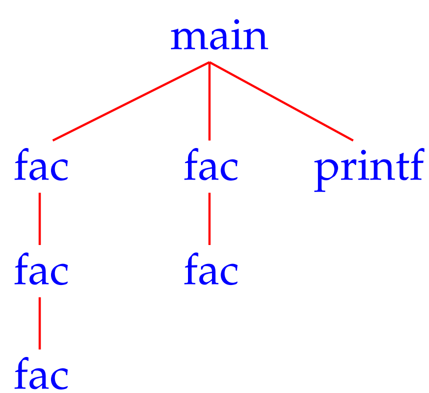
```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf ("%d", n);
}
```

At any time during the execution, several instances of one function may exist, i.e., may have started, but not finished execution.

An instance is created by a call to the function.

The recursion tree in the example:



We conclude:

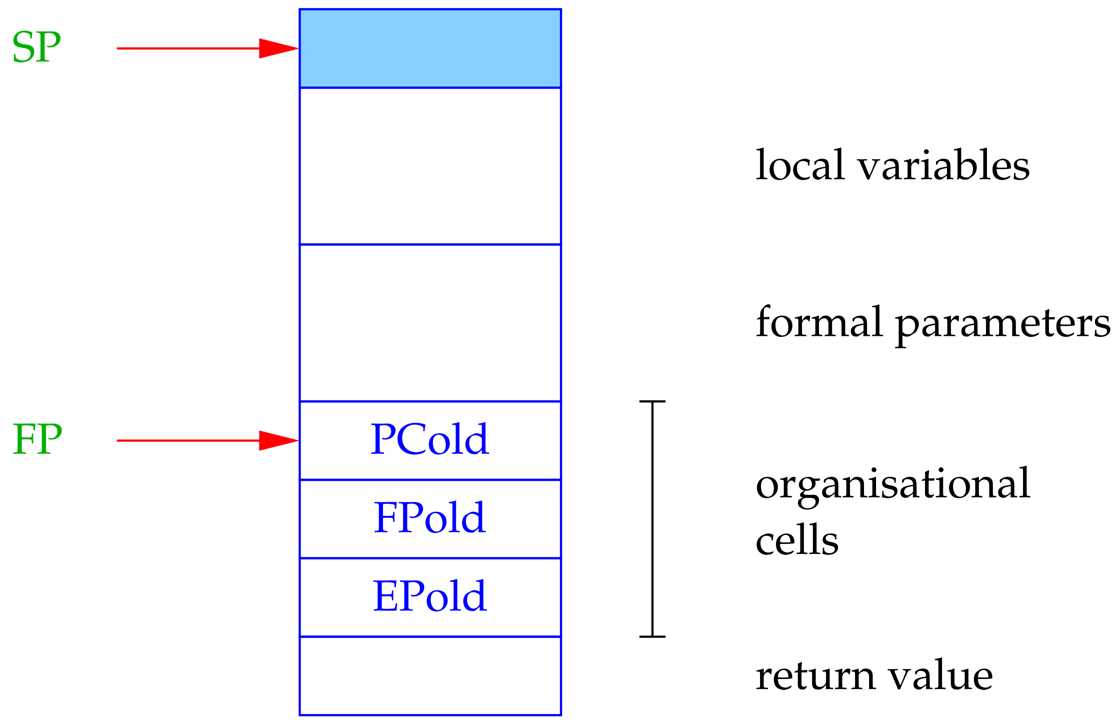
The formal parameters and local variables of the different instances of the same function must be kept separate.

Idea:

Allocate a special storage area for each instance of a function.

In sequential programming languages these storage areas can be managed on a stack. They are therefore called Stack Frames.

9.1 Storage Organization for Functions



FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used to address the formal parameters and the local variables.

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

- the FP
- the continuation address after the call and
- the actual EP.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

- Generate code for the body!
- Generate code for calls!

9.2 Computing the Address Environment

We have to distinguish two different kinds of variables:

1. globals, which are defined externally to the functions;
2. locals/automatic (including formal parameters), which are defined internally to the functions.

$$\Longrightarrow$$

The address environment ρ associates pairs $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ with their names.

Note:

- There exist more refined notions of visibility of (the defining occurrences of) variables, namely nested blocks.
- The translation of different program parts in general uses different address environments!

Example (1):

$\boxed{0}$

```
int i;
struct list {
    int info;
    struct list * next;
} * l;
```

$\boxed{1}$

```
int ith (struct list * x, int i) {
    if (i ≤ 1) return x →info;
    else return ith (x →next, i − 1);
}
```

$\boxed{2}$

```
main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l,i));
}
```

address environment at $\boxed{0}$

$$
\begin{array}{lccc}
\rho_0 : & i & \mapsto & (G, 1) \\
 & l & \mapsto & (G, 2) \\
 & \text{ith} & \mapsto & (G, \_\text{ith}) \\
 & \text{main} & \mapsto & (G, \_\text{main}) \\
 & & \ldots &
\end{array}
$$

Example (2):

$\boxed{0}$

```
int i;
struct list {
    int info;
    struct list * next;
} * l;
```

$\boxed{1}$

```
int ith (struct list * x, int i) {
    if (i ≤ 1) return x →info;
    else return ith (x →next, i − 1);
}
```

$\boxed{2}$

```
main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l,i));
}
```

$\boxed{1}$ inside of ith:

ρ_1 :	i	$\mapsto$	$(L, 2)$
	x	$\mapsto$	$(L, 1)$
	l	$\mapsto$	$(G, 2)$
	ith	$\mapsto$	$(G,$ _ith$)$
	main	$\mapsto$	$(G,$ _main$)$
		$\dots$	

Example (3):

$\boxed{0}$

```
int i;
struct list {
    int info;
    struct list * next;
} * l;
```

$\boxed{1}$

```
int ith (struct list * x, int i) {
    if (i ≤ 1) return x →info;
    else return ith (x →next, i − 1);
}
```

$\boxed{2}$

```
main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l,i));
}
```

$\boxed{2}$ inside of main:

$$\rho_2 : \begin{array}{rcl} i & \mapsto & (G, 1) \\ l & \mapsto & (G, 2) \\ k & \mapsto & (L, 1) \\ \text{ith} & \mapsto & (G, \_\text{ith}) \\ \text{main} & \mapsto & (G, \_\text{main}) \\ & \ldots & \end{array}$$

9.3 Calling/Entering and Leaving Functions

Be f the actual function, the Caller, and let f call the function g, the Callee.

The code for a function call has to be distributed among the Caller and the Callee:

The distribution depends on who has which information.

Actions upon calling/entering g:

1. Saving FP, EP	mark	available in f
2. Computing the actual parameters		
3. Determining the start address of g		
4. Setting the new FP	call	
5. Saving PC and jump to the beginning of g		
6. Setting the new EP	enter	available in g
7. Allocating the local variables	alloc	

Actions upon leaving g:

1. Restoring the registers FP, EP, SP	return
2. Returning to the code of f, i.e. restoring the PC	

Altogether we generate for a call:

$$
\begin{aligned}
\mathrm{code}_{\mathrm{R}}\; g(e_1, \ldots, e_n)\; \rho \quad = \quad & \mathrm{mark} \\
& \mathrm{code}_{\mathrm{R}}\; e_1\; \rho \\
& \ldots \\
& \mathrm{code}_{\mathrm{R}}\; e_m\; \rho \\
& \mathrm{code}_{\mathrm{R}}\; g\; \rho \\
& \mathrm{call\ n}
\end{aligned}
$$

where n = space for the actual parameters

Note:

- Expressions occurring as actual parameters will be evaluated to their R-value $\Longrightarrow$ Call-by-Value-parameter passing.
- Function g can also be an expression, whose R-value is the start address of the function to be called ...

- Function names are regarded as constant pointers to functions, similarly to declared arrays. The R-value of such a pointer is the start address of the function.

- For a variable **int** $(*)()\ g;$, the two calls

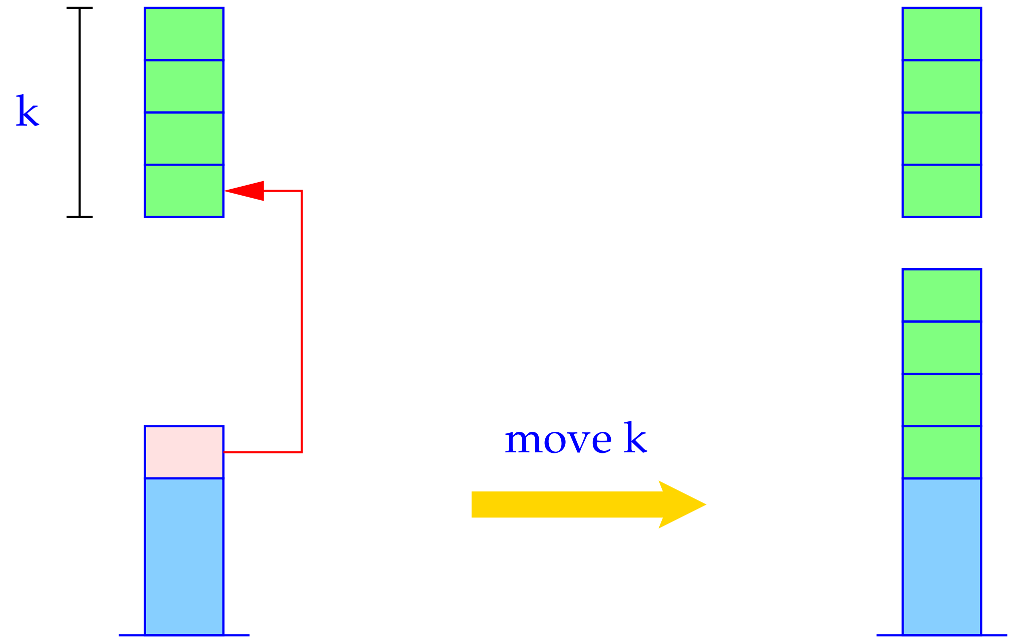
$$(*g)() \qquad \text{und} \qquad g()$$

  are equivalent :-)
  Normalization: Dereferencing of a function pointer is ignored.

- Structures are copied when they are passed as parameters.

In consequence:

$$\begin{array}{lcll}
\text{code}_R\ f\ \rho & = & \text{loadc}\ (\rho\ f) & f \text{ a function name} \\
\text{code}_R\ (*e)\ \rho & = & \text{code}_R\ e\ \rho & e \text{ a function pointer} \\
\text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & \\
 & & \text{move k} & e \text{ a structure of size } k
\end{array}$$

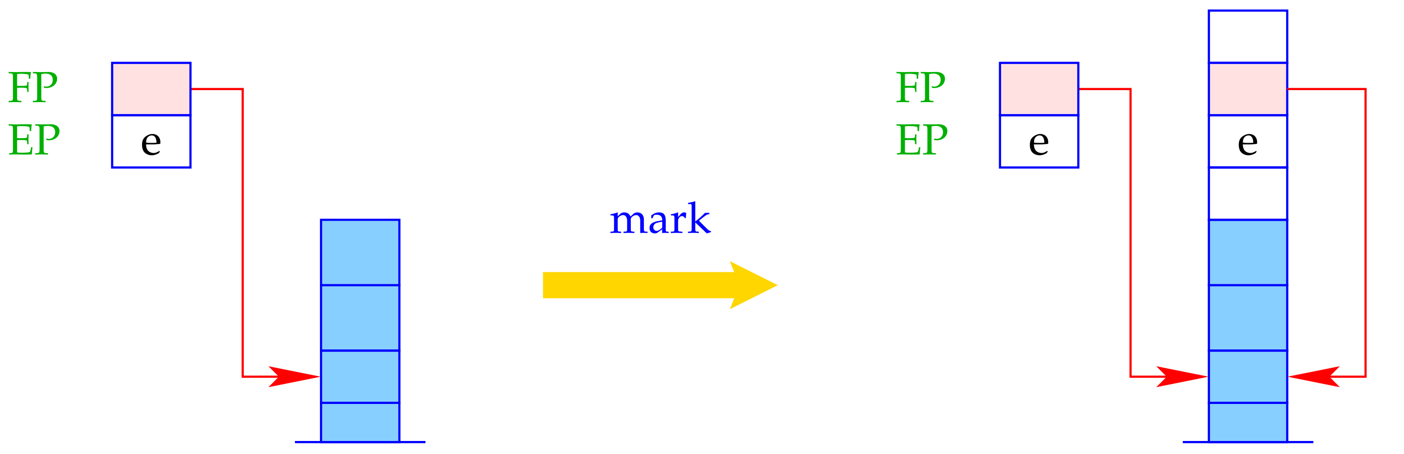


for (i = k-1; i$\geq$0; i--)

S[SP+i] = S[S[SP]+i];

SP = SP+k–1;

The instruction mark allocates space for the return value and for the organizational cells and saves the FP and EP.

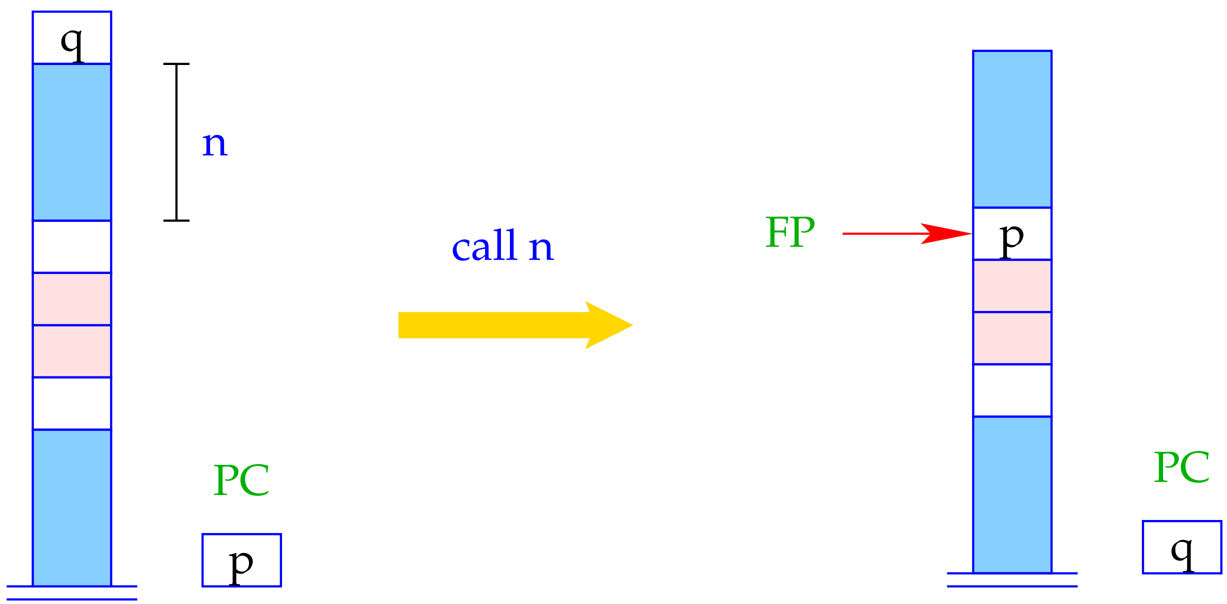


S[SP+2] = EP;

S[SP+3] = FP;

SP = SP + 4;

The instruction call n saves the continuation address and assigns FP, SP, and PC their new values.



FP = SP - n - 1;

S[FP] = PC;

PC = S[SP];

SP--;

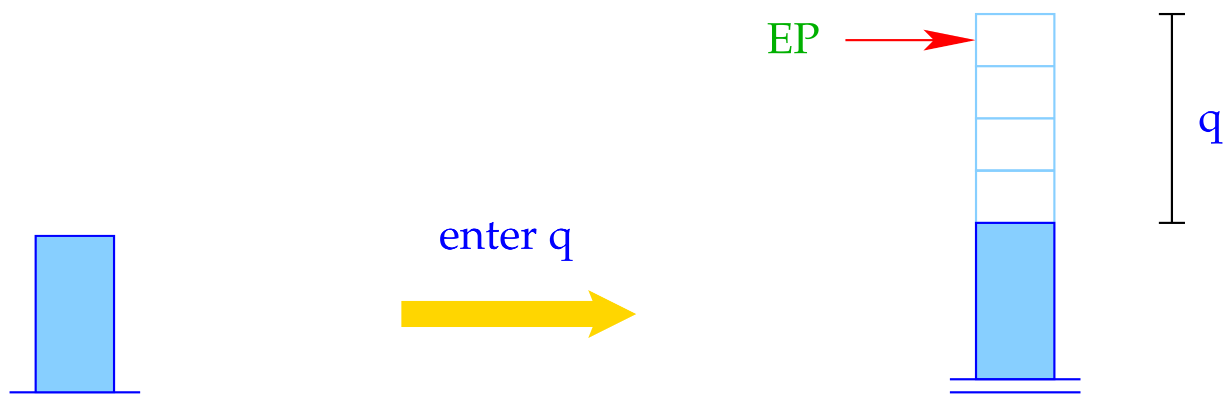
Correspondingly, we translate a function definition:

$$\text{code} \quad t\ \mathrm{f}\ (specs)\{V\_defs \quad ss\} \quad \rho \quad =$$

```
_f:   enter q        //  Setting the EP
      alloc k        //  Allocating the local variables
      code ss ρ_f
      return         //  leaving the function
```

where	t	$=$	return type of f with $\lvert t \rvert \leq 1$
	q	$=$	$maxS + \mathrm{k}$ where
	$maxS$	$=$	maximal depth of the local stack
	k	$=$	space for the local variables
	ρ_{f}	$=$	address environment for f
			// *takes care of specs, V_defs* and ρ

The instruction enter q sets EP to its new value. Program execution is terminated if not enough space is available.

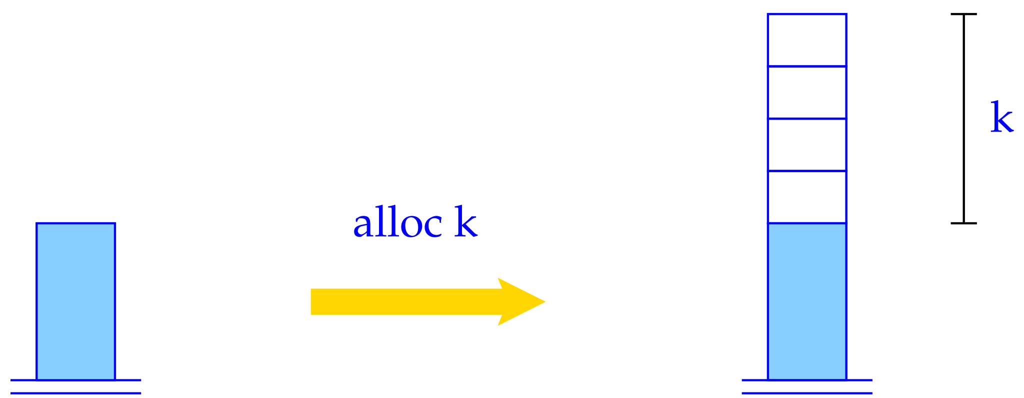


EP = SP + q;

if (EP $\geq$ NP)

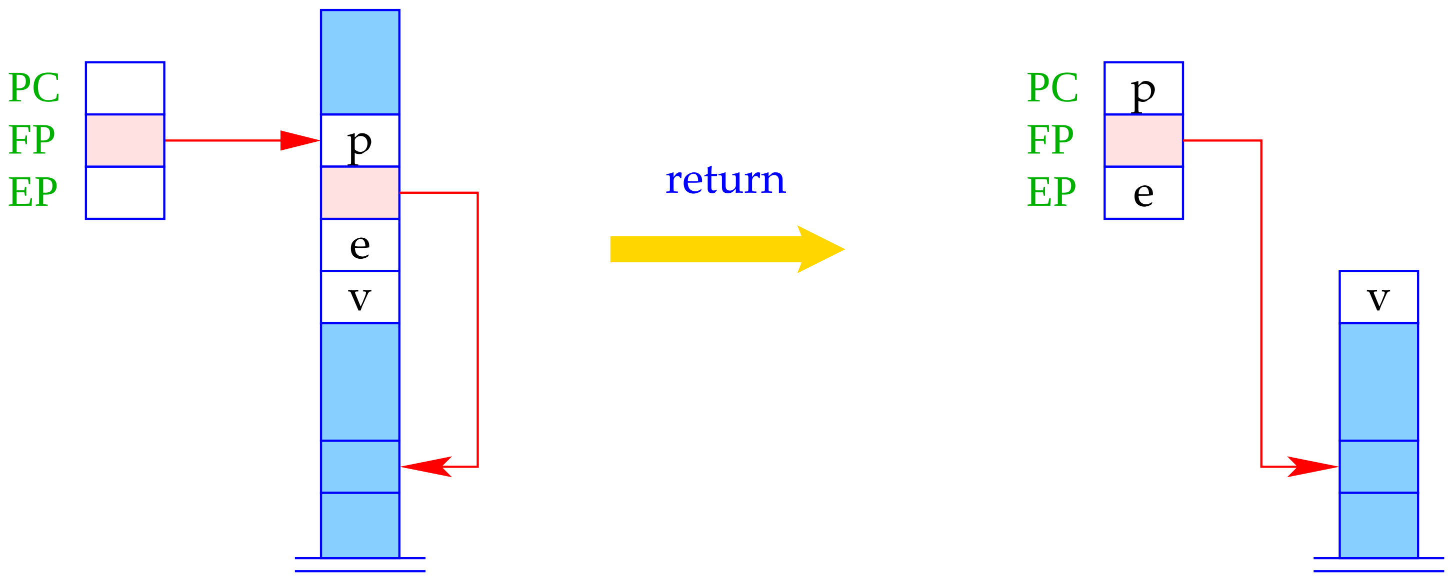
Error (“Stack Overflow”);

The instruction alloc k reserves stack space for the local variables.



SP = SP + k;

The instruction return pops the actual stack frame, i.e., it restores the registers PC, EP, SP, and FP and leaves the return value on top of the stack.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error ("Stack Overflow");

SP = FP-3; FP = S[SP+2];

9.4 Access to Variables and Formal Parameters, and Return of Values

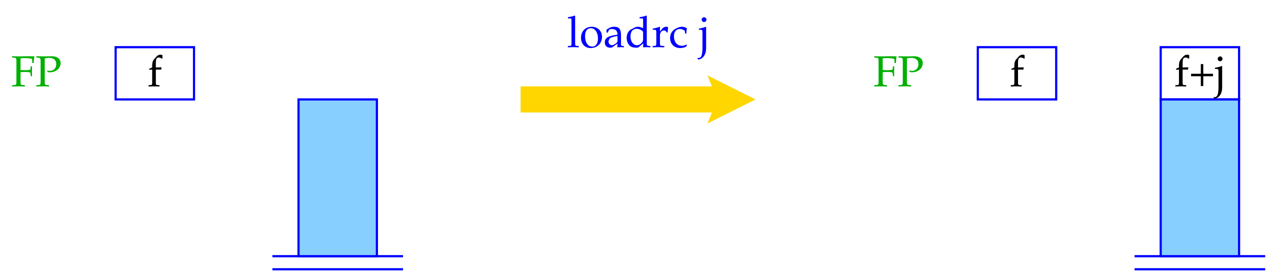
Local variables and formal parameters are addressed relative to the current FP.

We therefore modify code_L for the case of variable names.

For $\quad \rho\, x = (tag, j) \quad$ we define

$$\text{code}_\text{L}\; x\; \rho = \begin{cases} \text{loadc j} & tag = G \\ \text{loadrc j} & tag = L \end{cases}$$

The instruction loadrc j computes the sum of FP and j.



SP++;

S[SP] = FP+j;

As an optimization one introduces the instructions loadr j and storer j . This is analogous to loada j and storea j.

loadr j = loadrc j
load

storer j = loadrc j
store

The code for **return** e; corresponds to an assignment to a variable with relative address -3.

code **return** $e;\ \rho$ = code$_R$ $e\ \rho$
storer -3
return

Example: For the function

$$
\begin{array}{l}
\textbf{int}\ \text{fac}\ (\textbf{int}\ x)\ \{ \\
\quad \textbf{if}\ (x \leq 0)\ \textbf{return}\ 1; \\
\quad \textbf{else return}\ x * \text{fac}\ (x - 1); \\
\}
\end{array}
$$

we generate:

_fac:	enter q	loadc 1	A:	loadr 1		mul
	alloc 0	storer -3		mark		storer -3
	loadr 1	return		loadr 1		return
	loadc 0	jump B		loadc 1	B:	return
	leq			sub		
	jumpz A			loadc _fac		
				call 1		

where $\rho_{\text{fac}} : x \mapsto (L, 1)$ and $q = 1 + 4 + 2 = 7.$

10 Translation of Whole Programs

The state before program execution starts:

$$\mathrm{SP} = -1 \qquad \mathrm{FP} = \mathrm{EP} = 0 \qquad \mathrm{PC} = 0 \qquad \mathrm{NP} = \mathrm{MAX}$$

Be $p \equiv V\_defs \quad F\_def_1 \ldots F\_def_n$, a program, where F_def_i defines a function f_i, of which one is named main.

The code for the program p consists of:

- Code for the function definitions F_def_i;
- Code for allocating the global variables;
- Code for the call of main();
- the instruction halt.

We thus define:

$$
\begin{array}{rcrl}
\text{code } p\ \emptyset & = & & \text{enter } (\mathrm{k}+6) \\
 & & & \text{alloc } (\mathrm{k}+1) \\
 & & & \text{mark} \\
 & & & \text{loadc \_main} \\
 & & & \text{call } 0 \\
 & & & \text{pop} \\
 & & & \text{halt} \\
 & & \_\mathrm{f}_1: & \text{code } F\_def_1\ \rho \\
 & & & \vdots \\
 & & \_\mathrm{f}_n: & \text{code } F\_def_n\ \rho
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & \emptyset & \widehat{=} & \text{empty address environment;} \\
 & \rho & \widehat{=} & \text{global address environment;} \\
 & \mathrm{k} & \widehat{=} & \text{space for global variables} \\
 & \text{\_main} & \in & \{\_\mathrm{f}_1, \ldots, \_\mathrm{f}_n\}
\end{array}
$$