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:

\[
\text{code}_R \ f \ \rho \ \ = \ \ \text{loadc}_f \ \ = \ \ \text{starting address of the code for } f
\]

\[\longrightarrow\] Function names must also be managed in the address environment!
Example:

```c
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** $\overset{=}{{\text{Frame Pointer}}}$; 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.

The address environment $\rho$ 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):

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

1 int ith (struct list * x, int i) {
    if (i <= 1) return x -> info;
    else return ith (x -> next, i - 1);
}
```

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

<table>
<thead>
<tr>
<th>address</th>
<th>environment at</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>(G, 1)</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>(G, 2)</td>
<td></td>
</tr>
<tr>
<td>ith</td>
<td>(G, _ith)</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>(G, _main)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example (2):

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

int ith (struct list * x, int i) {
    if (i <= 1) return x -> info;
    else return ith (x -> next, i - 1);
}
```

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

1. inside of ith:
   - $i \mapsto (L, 2)$
   - $x \mapsto (L, 1)$
   - $l \mapsto (G, 2)$
   - $ith \mapsto (G, _ith)$
   - $main \mapsto (G, _main)$
   ...

2. main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l, i));
}
Example (3):

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

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

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

| inside of main: |
|-----------------|-----------------|-----------------|
| ρ₂:             |
| i → (G, 1)      |
| l → (G, 2)      |
| k → (L, 1)      |
| ith → (G, _ith) |
| main → (G, _main) |
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** \( \texttt{g} \): 

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

Actions upon **leaving** \( \texttt{g} \): 

1. Restoring the registers \( \texttt{FP, EP, SP} \) \( \text{return} \) 
2. Returning to the code of \( f \), i.e. restoring the \( \texttt{PC} \) 

78
Altogether we generate for a call:

$$\text{code}_R \ g(e_1, \ldots, e_n) \ \rho = \ \text{mark}$$
$$\text{code}_R \ e_1 \ \rho$$
$$\ldots$$
$$\text{code}_R \ e_m \ \rho$$
$$\text{code}_R \ g \ \rho$$
$$\text{call} \ n$$

where \( n \) = space for the actual parameters

**Note:**

- Expressions occurring as actual parameters will be evaluated to their \( R \)-value \( \implies \) 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 \texttt{int (*)()} \texttt{g;}, the two calls

\[(\ast g)() \quad \text{und} \quad g()\]

are equivalent  

\textbf{Normalization:} Dereferencing of a function pointer is ignored.

• Structures are copied when they are passed as parameters.

In consequence:

\[
\begin{align*}
\text{code}_R \ f \ \rho & = \ \text{loadc} \ (\rho f) & f \ \text{a function name} \\
\text{code}_R \ (\ast e) \ \rho & = \ \text{code}_R \ e \ \rho & e \ \text{a function pointer} \\
\text{code}_R \ e \ \rho & = \ \text{code}_L \ e \ \rho \\
& \quad \text{move} \ k & e \ \text{a structure of size} \ k
\end{align*}
\]
for (i = k-1; i ≥ 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:

```plaintext
code  t f (specs) {V_defs ss}   ρ  =

    _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 \( |t| \leq 1 \)
- \( q = maxS + k \) 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 \texttt{enter q} sets $EP$ to its new value. Program execution is terminated if not enough space is available.

\begin{align*}
EP &= SP + q; \\
\text{if } (EP \geq NP) &\text{ Error ("Stack Overflow"));}
\end{align*}
The instruction `alloc k` reserves stack space for the local variables.

```
SP = SP + k;
```
The instruction \texttt{return} pops the actual stack frame, i.e., it restores the registers \texttt{PC}, \texttt{EP}, \texttt{SP}, and \texttt{FP} and leaves the return value on top of the stack.

\[\text{PC} = S[\text{FP}]; \text{EP} = S[\text{FP}-2];\]
\[\text{if } (\text{EP} \geq \text{NP}) \text{ Error ("Stack Overflow");}\]
\[\text{SP} = \text{FP}-3; \text{FP} = S[\text{SP}+2];\]
Local variables and formal parameters are addressed relative to the current FP. We therefore modify codeL for the case of variable names.

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

$$\text{code}_L \ x \ \rho = \begin{cases} 
\text{loadc} \ j & \text{tag} = G \\
\text{loadrc} \ j & \text{tag} = L 
\end{cases}$$
The instruction `loadrc j` computes the sum of `FP` and `j`.

```
FP f

loadrc j

FP f
f+j
```

```
SP++;  
S[SP] = FP+j;
```
As an optimization one introduces the instructions \texttt{loadr} \texttt{j} and \texttt{storer} \texttt{j}.
This is analogous to \texttt{loada} \texttt{j} and \texttt{storea} \texttt{j}.

\begin{align*}
\texttt{loadr} \texttt{j} & = \texttt{loadrc} \texttt{j} \\
& \text{load}
\end{align*}

\begin{align*}
\texttt{storer} \texttt{j} & = \texttt{loadrc} \texttt{j} \\
& \text{store}
\end{align*}

The code for \texttt{return} \texttt{e;} corresponds to an assignment to a variable with relative address $-3$.

\begin{align*}
\texttt{code return} \texttt{e;} \; \rho & = \; \texttt{code} \texttt{R} \; \texttt{e} \; \rho \\
& \texttt{storer} \; -3 \\
& \texttt{return}
\end{align*}
Example: For the function

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

we generate:

\[
\_\text{fac}: \quad \text{enter } q \\
\quad \text{alloc } 0 \\
\quad \text{loadr } 1 \\
\quad \text{loadc } 0 \\
\quad \text{leq} \\
\quad \text{jumpz } A \\
\]

\[
A: \quad \text{loadr } 1 \\
\quad \text{mark} \\
\quad \text{storer } -3 \\
\quad \text{return} \\
\quad \text{loadr } 1 \\
\quad \text{loadc } 1 \\
\quad \text{sub} \\
\quad \text{mark} \\
\quad \text{storer } -3 \\
\quad \text{return} \\
\]

\[
B: \quad \text{return} \\
\quad \text{loadc } \_\text{fac} \\
\quad \text{call } 1
\]

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:

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

Be \( p \equiv V_{\text{defs}} \ F_{\text{def}1} \ldots F_{\text{def}_n} \), a program, where \( F_{\text{def}_i} \) defines a function \( f_i \), of which one is named \( \text{main} \).

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

- Code for the function definitions \( F_{\text{def}_i} \);
- Code for allocating the global variables;
- Code for the call of \( \text{main}() \);
- the instruction \( \text{halt} \).
We thus define:

\[
\begin{align*}
\text{code } p \emptyset & \quad = \quad \text{enter } (k + 6) \\
& \quad \quad \quad \text{alloc } (k + 1) \\
& \quad \quad \quad \text{mark} \\
& \quad \quad \quad \text{loadc } \text{_main} \\
& \quad \quad \quad \text{call } 0 \\
& \quad \quad \quad \text{pop} \\
& \quad \quad \quad \text{halt} \\
\text{\_f}_1 & \quad : \quad \text{code } F_{\text{def}_1} \rho \\
\vdots \\
\text{\_f}_n & \quad : \quad \text{code } F_{\text{def}_n} \rho
\end{align*}
\]

where

- $\emptyset \equiv$ empty address environment;
- $\rho \equiv$ global address environment;
- $k \equiv$ space for global variables
- $\text{\_main} \in \{\text{\_f}_1, \ldots, \text{\_f}_n\}$