Example: \( \text{int } a[10]; \ b; \) with \( \rho = \{ a \mapsto 7, b \mapsto 17 \} \).

For the statement: \( \text{\texttt{\_\_}a = 5;} \) we obtain:

\[
\begin{align*}
\text{\texttt{\_\_} code} \ (a) \ \rho & = \text{\texttt{\_\_} code} \ a \ \rho = \text{\texttt{\_\_} code} \ a \ \rho = \text{\texttt{\_\_} load} \ 7 \\
\text{\texttt{\_\_} code} \ (a = 5) \ \rho & = \text{\texttt{\_\_} load} \ 5 \\
& \text{\texttt{\_\_} load} \ 7 \\
& \text{\texttt{\_\_} store} \\
& \text{\texttt{\_\_} pop}
\end{align*}
\]

As an exercise translate:

\( s_1 = b = (\&a) + 2; \) and \( s_2 = * (b + 3) = 5; \)
**Dererencing of Pointers:**

The application of the operator \(\*=\) to the expression \(e\) returns the contents of the storage cell, whose address is the R-value of \(e\):

\[
\text{code}_L ((\ast e)\rho) = \text{code}_L e \rho
\]

**Example:**

Given the declarations:

\[
\text{struct } t \{ \text{int } a[7]; \text{struct } t \star b; \};
\]

\[
\text{int } i, j;
\]

\[
\text{struct } t \star pt;
\]

and the expression \(((\ast pt) \rightarrow a)[i + 1]\)

Because of \(e \rightarrow a \equiv (\ast e).a\) holds:

\[
\text{code}_L (e \rightarrow a) \rho = \text{code}_L e \rho
\]

**For arrays, their R-value equals their L-value. Therefore:**

\[
\text{code}_L ((\ast pt) \rightarrow a) \rho = \text{code}_L (\ast pt) \rho
\]

In total, we obtain the instruction sequence:

\[
\text{load } 3 \quad \text{load} \quad \text{load } 1 \quad \text{load} \quad \text{add} \quad \text{add} \quad \text{add}
\]
Be \( \rho = \{ i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7 \} \). Then:

\[
\begin{align*}
\text{code}_1 ((pt \to b) \to a)[i+1] \rho &= \text{code}_2 ((pt \to b) \to a) \rho \\
&= \text{code}_2 ((pt \to b) \to a) \rho \\
&= \text{loada} 1 \\
&= \text{loadc} 1 \\
&= \text{add} \\
&= \text{loada} 1 \\
&= \text{mul} \\
&= \text{add}
\end{align*}
\]

For arrays, their R-value equals their L-value. Therefore:

\[
\begin{align*}
\text{code}_1 ((pt \to b) \to a) \rho &= \text{code}_2 (pt \to b) \rho \\
&= \text{loada} 3 \\
&= \text{load} 0 \\
&= \text{load} 7 \\
&= \text{add} \\
&= \text{load} \\
&= \text{loadc} 0 \\
&= \text{add}
\end{align*}
\]

In total, we obtain the instruction sequence:

\[
\begin{align*}
\text{loada} 3 \\
\text{load} 0 \\
\text{load} 7 \\
\text{loadc} 0 \\
\text{mul} \\
\text{add} \\
\text{add} \\
\text{add}
\end{align*}
\]

7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\begin{align*}
\text{code}_1 (e_1[e_2]) \rho &= \text{code}_2 e_1 \rho \\
&= \text{code}_2 e_2 \rho \\
&= \text{loadc} [i] \\
&= \text{mul} \\
&= \text{add} \\
&\quad \text{if } e_1 \text{ has type } t* \text{ or } t[
\end{align*}
\]

\[
\begin{align*}
\text{code}_1 (e) \rho &= \text{code}_2 e \rho \\
&= \text{loadc} (\rho e) \\
&= \text{loada} (\rho e) \\
&= \text{add}
\end{align*}
\]

\[
\begin{align*}
\text{code}_1 (\ast e) \rho &= \text{code}_2 e \rho \\
&= \text{loadc} (\rho x) \\
\end{align*}
\]

\[
\begin{align*}
\text{code}_1 (\& e) \rho &= \text{code}_2 e \rho \\
&= \text{loadc} e \rho \\
&= \text{code}_2 e \rho \\
&= \text{op} \\
&\quad \text{op instruction for operator } \&
\end{align*}
\]

\[
\begin{align*}
\text{code}_1 (e_{\#}) \rho &= \text{code}_2 e \rho \\
&= \text{code}_2 e \rho \\
&= \text{code}_2 e \rho \\
&= \text{loadc} (\rho e) \\
&= \text{code}_2 (\rho e) \\
&= \text{code}_2 (\rho e) \\
&= \text{add}
\end{align*}
\]
\[
\begin{align*}
\text{code}_\mathbb{1}(\ast e) \rho &= \text{code}_\mathbb{2} e \rho \\
\text{code}_\mathbb{1} x \rho &= \text{loadc} (\rho x) \\
\text{code}_\mathbb{2} (\& e) \rho &= \text{code}_\mathbb{1} e \rho \\
\text{code}_\mathbb{2} e \rho &= \text{code}_\mathbb{1} e \rho & \text{if } e \text{ is an array} \\
\text{code}_\mathbb{2} (e_1 \sqcup e_2) \rho &= \text{code}_\mathbb{2} e_1 \rho \\
&\quad \text{code}_\mathbb{2} e_2 \rho \\
&\quad \text{op} & \text{op instruction for operator ‘\(\sqcup\)’}
\end{align*}
\]

Example: \(\text{int } a[10], +b; \) with \(\rho = \{ a \mapsto 7, b \mapsto 17 \}\).

For the statement: \(+a = 5; \) we obtain:

\[
\begin{align*}
\text{code}_\mathbb{1}(+a) \rho &= \text{code}_\mathbb{2} a \rho = \text{code}_\mathbb{1} a \rho = \text{loadc } 7 \\
\text{code } (+a = 5) \rho &= \text{loadc } 5 \\
&\quad \text{loadc } 7 \\
&\quad \text{store} \\
&\quad \text{pop}
\end{align*}
\]

As an exercise translate:

\(s_1 = b = (\& a) + 2; \) and \(s_2 = s(b + 3) = 5; \)
\[
\int (x^2) [10],
\]

Example: \( \int a[10], b \) with \( \rho = \{a \mapsto 7, b \mapsto 17 \} \).

For the statement: \( +a = 5 \), we obtain:

\[
\begin{align*}
\text{code}_L(a) \rho & = \text{code}_L(b) \rho = \text{code}_L(a) \rho = \text{load} 7 \\
\text{code}(a = 5) \rho & = \text{load} 5 \\
& \quad \text{load} 7 \\
& \quad \text{store} \\
& \quad \text{pop}
\end{align*}
\]

As an exercise translate:

\( s_1 = b = \text{size}(a + 2) \) and \( s_2 = x(b + 3) = 5 \)
Example: \( \text{int } a[10] \); \( +b; \) with \( \rho = \{ a \mapsto 7, b \mapsto 17 \} \).

For the statement: \( +a = 5; \) we obtain:

\[
\begin{align*}
\text{code}_{e} (\ast a) \rho &= \text{code}_{d} x \rho = \text{code}_{d} y \rho = \text{load } 7 \\
\text{code}_{c} (\ast a = 5; ) \rho &= \text{load } 5 \\
&\quad \text{load } 7 \\
&\quad \text{store} \\
&\quad \text{pop}
\end{align*}
\]

As an exercise translate:

\( s_{1} = b = (\&a) + 2; \) and \( s_{2} = x(b + 3) = 5; \)

\[64\]

\[65\]
8 Freeing Occupied Storage

Problems:
- The freed storage area is still referenced by other pointers (dangling references).
- After several deallocations, the storage could look like this (fragmentation):

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Potential Solutions:
- Trust the programmer. Manage freed storage in a particular data structure (free list) \( \longrightarrow \) malloc or free may become expensive.
- Do nothing, i.e.:

\[
\text{code free (e); } \rho = \text{code e } \rho \text{ pop}
\]

\( \longrightarrow \) simple and (in general) efficient.
- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.
Potential Solutions:

- Trust the programmer. Manage freed storage in a particular data structure (free list) malloc or free my become expensive.
- Do nothing, i.e.:
  \[
  \text{code free } \rho; \quad \text{code } \rho \quad \text{pop}
  \]
  simple and (in general) efficient.
- Use an automatic, potentially “conservative” Garbage-Collection, which occasionally collects certainly inaccessible heap space.

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:

\[
\text{code } \rho \quad \text{load } \rho \quad \text{starting address of the code for } f
\]

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);
}
```

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:

```
main
  fac
    fac
      fac
      fac
```

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

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!

FP = Frame Pointer; points to the last organizational cell and is used to address the formal parameters and the local variables.
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 ρ associates pairs \((tag, a) \in \{G, L\} \times N_0\) 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 * x, int j;

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

Example (2):

```c
0 int i;
struct list * x, int j;

1 int ith(struct list * x, int j) {
    if (j <= 1) return x->info;
    else return ith(x->next, i - 1);
}
```
Example (3):

```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);
   }
```

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
2. Computing the actual parameters
3. Determining the start address of $g$
4. Setting the new FP
5. Saving PC and jump to the beginning of $g$
6. Setting the new EP
7. Allocating the local variables

Actions upon **leaving** $g$:
1. Restoring the registers FP, EP, SP
2. Returning to the code of $f$, i.e. restoring the PC

Altogether we generate for a call:

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

where $n = \text{space for the actual parameters}$

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...
Actions upon calling/entering \textit{g}:

1. Saving FP, EP
2. Computing the actual parameters
3. Determining the start address of \textit{g}
4. Setting the new FP
5. Saving PC and jump to the beginning of \textit{g}
6. Setting the new EP
7. Allocating the local variables

Actions upon leaving \textit{g}:

1. Restoring the registers FP, EP, SP
2. Returning to the code of \textit{i}, i.e. restoring the PC

\begin{align*}
\text{code}_{\text{g}}(e_1, \ldots, e_n) \ & \equiv \ \text{mark} \\
\text{code}_{\text{e}_1} \ & \equiv \ \text{call} \\
\vdots \\
\text{code}_{\text{e}_n} \ & \equiv \ \text{call} \\
\text{code}_{\text{g}} \ & \equiv \ \text{call} \\
\text{call} \ & \equiv \ n
\end{align*}

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

\textbf{Note:}

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