Dereferencing of Pointers:

The application of the operator $\ast$ to the expression $e$ returns the contents of the storage cell, whose address is the R-value of $e$:

$$\text{code}_L (\ast e) \rho = \text{code}_R e \rho$$

Example: Given the declarations

```c
struct t { int a[7]; struct t *b; };
int i, j;
struct t *pt;
```

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

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

$$\text{code}_L (e \rightarrow a) \rho = \text{code}_R e \rho$$

loadc ($\rho a$)

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

\[
\text{code}_L ((pt \to b) \to a)[i + 1] \rho = \text{code}_R ((pt \to b) \to a) \rho = \text{code}_R ((pt \to b) \to a) \rho
\]

\[
= \text{code}_R (i + 1) \rho \quad \text{loada} 1
\]

\[
= \text{loadc} 1 \quad \text{loadc} 1
\]

\[
= \text{mul} \quad \text{add}
\]

\[
= \text{add} \quad \text{loadc} 1 \quad \text{mul} \quad \text{add}
\]
For arrays, their R-value equals their L-value. Therefore:

$$\text{code}_R ((pt \rightarrow b) \rightarrow a) \rho = \text{code}_R (pt \rightarrow b) \rho = \text{loada } 3$$
$$\text{loadc } 0$$
$$\text{add}$$
$$\text{loadc } 7$$
$$\text{add}$$
$$\text{load}$$
$$\text{loadc } 0$$
$$\text{add}$$

In total, we obtain the instruction sequence:

<table>
<thead>
<tr>
<th>loada 3</th>
<th>load</th>
<th>loada 1</th>
<th>loadc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadc 7</td>
<td>loadc 0</td>
<td>loadc 1</td>
<td>mul</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
</tbody>
</table>
## 7 Conclusion

We tabulate the cases of the translation of expressions:

<table>
<thead>
<tr>
<th>Code Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{code}_L (e_1[e_2]) \rho = \text{code}_R e_1 \rho )</td>
<td>if ( e_1 ) has type ( t^* ) or ( t[^] )</td>
</tr>
<tr>
<td>( \text{code}_R e_2 \rho )</td>
<td></td>
</tr>
<tr>
<td>( \text{loading} \</td>
<td>t</td>
</tr>
<tr>
<td>( \text{mul} )</td>
<td></td>
</tr>
<tr>
<td>( \text{add} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{code}_L (e.a) \rho = \text{code}_L e \rho )</td>
<td></td>
</tr>
<tr>
<td>( \text{loading} \ (\rho \ a) )</td>
<td></td>
</tr>
<tr>
<td>( \text{add} )</td>
<td></td>
</tr>
</tbody>
</table>
\[
\text{code}_L \ (*e) \ \rho \quad = \quad \text{code}_R \ e \ \rho \\
\text{code}_L \ x \ \rho \quad = \quad \text{loadc} \ (\rho \ x) \\
\text{code}_R \ (&e) \ \rho \quad = \quad \text{code}_L \ e \ \rho \\
\text{code}_R \ e \ \rho \quad = \quad \text{code}_L \ e \ \rho \quad \text{if } e \text{ is an array} \\
\text{code}_R \ (e_1 \ \Box \ e_2) \ \rho \quad = \quad \text{code}_R \ e_1 \ \rho \\
\text{code}_R \ e_2 \ \rho \\
\text{op} \quad \text{op instruction for operator ‘} \Box \text{'}
\]
\[
\text{code}_R \ q \ \rho \quad = \quad \text{loadc} \ q \quad q \ \text{constant}
\]

\[
\text{code}_R \ (e_1 = e_2) \ \rho \quad = \quad \text{code}_R \ e_2 \ \rho \\
\quad \quad \quad \text{code}_L \ e_1 \ \rho \\
\quad \quad \quad \text{store}
\]

\[
\text{code}_R \ e \ \rho \quad = \quad \text{code}_L \ e \ \rho \\
\quad \quad \quad \text{load} \quad \quad \quad \text{otherwise}
\]
Example: \[\text{int } a[10], (*b)[10]; \quad \text{with } \rho = \{a \mapsto 7, b \mapsto 17\}.\]

For the statement: \[\ast a = 5;\] we obtain:

\[
\begin{align*}
\text{code}_L (\ast a) \rho &= \text{code}_R a \rho = \text{code}_L a \rho = \text{loadc } 7 \\
\text{code } (\ast a = 5; ) \rho &= \text{loadc } 5 \\
&\quad \text{loadc } 7 \\
&\quad \text{store} \\
&\quad \text{pop}
\end{align*}
\]

As an exercise translate:

\[s_1 \equiv b = (\&a) + 2; \quad \text{and} \quad s_2 \equiv (b + 3)[0] = 5;\]
\text{code } (s_1 s_2) \rho = \text{loadc } 7 \quad \text{loadc } 5 \\
\text{loadc } 2 \quad \text{loadc } 17 \\
\text{loadc } 10 \quad // \quad \text{size of } \textbf{int}[10] \\
\text{mul} \quad // \quad \text{scaling} \\
\text{add} \\
\text{loadc } 17 \\
\text{store} \\
\text{pop} \quad // \quad \text{end of } s_1 \\
\text{store} \\
\text{pop} \quad // \quad \text{end of } s_2
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):

```
frei
```
Potential Solutions:

• Trust the programmer. Manage freed storage in a particular data structure (free list) $\implies$ `malloc` or `free` my become expensive.

• Do nothing, i.e.:

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

$\implies$ 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;
- a possible result type;
- a block of statements.

In C, we have:

\[
\text{code}_R f \rho = \text{load c }_f = \text{start address of the code for } f
\]

\[\implies\] Function names must be maintained within the address environment!
Example

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

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

At every point of execution, several instances (calls) of the same function may be active, i.e., have been started, but not yet completed.

The recursion tree of the example:
We conclude:

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

Idea

Allocate a dedicated memory block for each call of a function.

In sequential programming languages, these memory blocks may be maintained on a stack. Therefore, they are also called stack frames.
9.1 Memory Organization for Functions

**FP** $\equiv$ **Frame Pointer**; points to the last **organizational cell** and is used for addressing the formal parameters and local variables.
Caveat

- The local variables receive relative addresses $+1, +2, \ldots$
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP :-) 
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for `printf`. 
- The memory block of parameters is recycled for storing the return value of the function :-) 

Simplification The return value fits into a single memory cell.
Caveat

- The local variables receive relative addresses $+1, +2, \ldots$
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP :-)
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for *printf*.
- The memory block of parameters is recycled for storing the return value of the function :-))

Simplification: The return value fits into a single cell.

Tasks of a Translator for Functions:

- Generate code for the body of the function!
- Generate code for calls!
9.2 Determining Address Environments

We distinguish two kinds of variables:

1. *global*/extern that are defined outside of functions;
2. *local*/intern/automatic (including formal parameters) which are defined inside functions.

The address environment $\rho$ maps names onto pairs $(tag, a) \in \{G, L\} \times \mathbb{Z}$.  

**Caveat**

- In general, there are further refined grades of visibility of variables.
- Different parts of a program may be translated relative to different address environments!
Example

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);
}

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

Address Environments Occurring in the Program:

Outside of the Function Definitions:

\[ \rho_0 : \begin{align*} i & \mapsto (G, 1) \\ l & \mapsto (G, 2) \\ \text{ith} & \mapsto (G, \_\text{ith}) \\ \text{main} & \mapsto (G, \_\text{main}) \end{align*} \]

Inside of \text{ith}:

\[ \rho_1 : \begin{align*} i & \mapsto (L, -4) \\ x & \mapsto (L, -3) \\ l & \mapsto (G, 2) \\ \text{ith} & \mapsto (G, \_\text{ith}) \\ \text{main} & \mapsto (G, \_\text{main}) \end{align*} \]
Caveat

- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells :-) 
- For a prototype $\tau f(\tau x_1, \ldots, \tau x_k)$ we define:

$$
x_1 \mapsto (L, -2 - |\tau_1|) \quad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)
$$
Caveat

• The actual parameters are evaluated from right to left !!
• The first parameter resides directly below the organizational cells  :-)
• For a prototype $\tau f(\tau_1 x_1, \ldots, \tau_k x_k)$ we define:

$$
x_1 \mapsto (L, -2 - |\tau_1|) \quad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)
$$

2. Inside of main:

$$
\rho_2 : \quad i \mapsto (G, 1) \\
\quad l \mapsto (G, 2) \\
\quad k \mapsto (L, 1) \\
\quad \text{ith} \mapsto (G, _{ith}) \\
\quad \text{main} \mapsto (G, _{main}) \\
\quad \ldots
$$
9.3 Calling/Entering and Exiting/Leaving Functions

Assume that $f$ is the current function, i.e., the caller, and $f$ calls the function $g$, i.e., the callee.

The code for the call must be distributed between the caller and the callee. The distribution can only be such that the code depending on information of the caller must be generated for the caller and likewise for the callee.

Caveat

The space requirements of the actual parameters is only known to the caller ...


Actions when *entering* \( g \):

1. Evaluating the actual parameters
2. Saving of \( FP, EP \)
3. Determining the start address of \( g \)
4. Setting of the new \( FP \)
5. Saving \( PC \) and
   
   Jump to the beginning of \( g \)
6. Setting of new \( EP \)
7. Allocating of local variables

\[ \{ \text{mark} \} \]
\[ \{ \text{call} \} \]
\[ \{ \text{enter} \} \]
\[ \{ \text{alloc} \} \]

\[ \{ \text{are part of } f \} \]
\[ \{ \text{are part of } g \} \]
Actions when terminating the call:

1. Storing of the return value
2. Restoring of the registers FP, EP, SP
3. Jumping back into the code of $f$, i.e.,
   Restauration of the PC
4. Popping the stack
Accordingly, we obtain for a call to a function with at least one parameter and one return value:

\[
\text{code}_R g(e_1, \ldots, e_n) \; \rho \; = \; \text{code}_R e_n \; \rho \\
\ldots \\
\text{code}_R e_1 \; \rho \\
\text{mark} \\
\text{code}_R g \; \rho \\
\text{call} \\
\text{slide} (m - 1)
\]

where $m$ is the size of the actual parameters.
Remark

• Of every expression which is passed as a parameter, we determine the R-value $\Rightarrow$ call-by-value passing of parameters.

• The function $g$ may as well be denoted by an expression, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...
• Similar to declared arrays, function names are interpreted as constant pointes onto function code. Thus, the R-value of this pointer is the start address of the function.

• **Caveat!** For a variable \texttt{int (*)() g;} the two calls

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

are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant :-)

• During passing of parameters, these are copied.

Consequently,

\[
\begin{align*}
\text{code}_R f \rho & = \text{loadc (}\rho f\text{)} & f \text{ name of a function} \\
\text{code}_R (*e) \rho & = \text{code}_R e \rho & e \text{ function pointer} \\
\text{code}_R e \rho & = \text{code}_L e \rho \\
& \quad \text{move} k & e \text{ a structure of size } k
\end{align*}
\]

where
for (i = k-1; i ≥ 0; i--)

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

\[ SP = SP+k-1; \]
The instruction \textit{mark} saves the registers \texttt{FP} and \texttt{EP} onto the stack.

\begin{align*}
\text{S[SP+1]} &= \text{EP}; \\
\text{S[SP+2]} &= \text{FP}; \\
\text{SP} &= \text{SP} + 2;
\end{align*}
The instruction `call` saves the return address and sets `FP` and `PC` onto the new values.

```plaintext
tmp = S[SP];
S[SP] = PC;
FP = SP;
PC = tmp;
```
The instruction \textit{slide} copies the return values into the correct memory cell:

$$
\text{tmp} = S[SP];
SP = SP - m;
S[SP] = \text{tmp};
$$
Accordingly, we translate a function definition:

\[
\text{code } t \ f \ (\text{specs})\{V_{\text{defs}} \ ss\} \ \rho \ = \\
\_f: \text{ enter } q \quad // \text{ initialize } EP \\
\text{ alloc } k \quad // \text{ allocate the local variables} \\
\text{ code } ss \ \rho_f \\
\text{ return } \quad // \text{ return from call}
\]

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 specs, \( V_{\text{defs}} \) and \( \rho \) into account
The instruction enter q sets the EP to the new value. If not enough space is available, program execution terminates.

\[ EP = SP + q; \]

if (EP ≥ NP)

Error (“Stack Overflow”);
The instruction `alloc k` allocates memory for locals on the stack.

\[ SP = SP + k; \]
The instruction \texttt{return} pops the current stack frame. This means it restores the registers \texttt{PC}, \texttt{EP} and \texttt{FP} and returns the return value on top of the stack.

\[
\begin{align*}
\text{PC} &= S[\text{FP}]; \\
\text{EP} &= S[\text{FP}-2]; \\
\text{if} \ (\text{EP} \geq \text{NP}) \ &\text{Error ("Stack Overflow")}; \\
\text{SP} &= \text{FP}-3; \\
\text{FP} &= S[\text{SP}+2];
\end{align*}
\]
9.4 Access to Variables, Formal Parameters and Returning of Values

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

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.

```
SP++;
S[SP] = FP+j;
```
As an optimization, we introduce analogously to $\text{load}a_j$ and $\text{store}a_j$ the new instructions $\text{load}r_j$ and $\text{store}r_j$:

$$\text{load}r_j \quad = \quad \text{load}rc_j$$

load

$$\text{store}r_j \quad = \quad \text{load}rc_j;$$

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

\begin{verbatim}
code return e; \rho = codeR e \rho
  storer -3
  return
\end{verbatim}

Example For function

\begin{verbatim}
int fac (int x) {
  if (x \leq 0) return 1;
  else return x * fac (x - 1);
}
\end{verbatim}

we generate: