Simplification:

We only regard **switch**-statements of the following form:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
& & \quad \textbf{case}\ 0\text{:} \quad ss_0\ \textbf{break};\\
& & \quad \textbf{case}\ 1\text{:} \quad ss_1\ \textbf{break};\\
& & \qquad \vdots \\
& & \quad \textbf{case}\ k-1\text{:} \quad ss_{k-1}\ \textbf{break};\\
& & \quad \textbf{default:} \quad ss_k \\
& & \}
\end{array}
$$

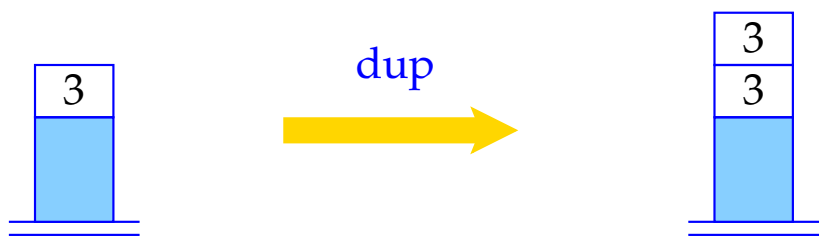
s is then translated into the instruction sequence:

code s ρ	=	code_R e ρ	C_0:	code ss_0 ρ	B:	jump C_0
		check 0 k B		jump D		...
				...		jump C_k
			C_k:	code ss_k ρ	D:	...
				jump D		

- The Macro check 0 k B checks, whether the R-value of e is in the interval $[0, k]$, and executes an indexed jump into the table B
- The jump table contains direct jumps to the respective alternatives.
- At the end of each alternative is an unconditional jump out of the **switch**-statement.

```
check 0 k B   =   dup        dup                 jumpi B
                  loadc 0    loadc k       A:    pop
                  geq        le                  loadc k
                  jumpz A    jumpz A             jumpi B
```

- The R-value of e is still needed for indexing after the comparison. It is therefore copied before the comparison.
- This is done by the instruction dup.
- The R-value of e is replaced by k before the indexed jump is executed if it is less than 0 or greater than k.



S[SP+1] = S[SP];

SP++;

Note:

- The jump table could be placed directly after the code for the Macro check. This would save a few unconditional jumps. However, it may require to search the **switch**-statement twice.
- If the table starts with u instead of 0, we have to decrease the R-value of e by u before using it as an index.
- If all potential values of e are definitely in the interval $[0, k]$, the macro check is not needed.

5 Storage Allocation for Variables

Goal:

Associate statically, i.e. at compile time, with each variable x a fixed (relative) address $\rho\, x$

Assumptions:

- variables of basic types, e.g. **int**, ... occupy one storage cell.
- variables are allocated in the store in the order, in which they are declared, starting at address 1.

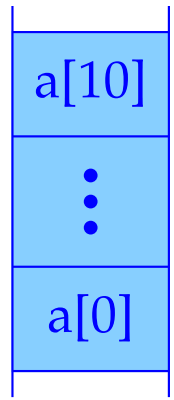
Consequently, we obtain for the declaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$ (t_i basic type) the address environment ρ such that

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Arrays

Example: **int** [11] a;

The array a consists of 11 components and therefore needs 11 cells.
$\rho\, a$ is the address of the component $a[0]$.



We need a function sizeof (notation: $|\cdot|$), computing the space requirement of a type:

$$|t| = \begin{cases} 1 & \text{if } t \text{ basic} \\ k \cdot |t'| & \text{if } t \equiv t'[k] \end{cases}$$

Accordingly, we obtain for the declaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{aligned} \rho\, x_1 &= 1 \\ \rho\, x_i &= \rho\, x_{i-1} + |t_{i-1}| \qquad \text{for } i > 1 \end{aligned}$$

Since $|\cdot|$ can be computed at compile time, also ρ can be computed at compile time.

Task:

Extend code_L and code_R to expressions with accesses to array components.

Be $t[c]\ a;$ the declaration of an array a.

To determine the start address of a component $a[i]$, we compute $\rho\, a + |t| * (\textit{R-value of } i)$.

In consequence:

$$
\begin{array}{lcl}
\text{code}_L\ a[e]\ \rho & = & \text{loadc}\ (\rho\, a) \\
 & & \text{code}_R\ e\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

... or more general:

$$
\begin{aligned}
\text{code}_\text{L}\ e_1[e_2]\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\
& \text{code}_\text{R}\ e_2\ \rho \\
& \text{loadc}\ |t| \\
& \text{mul} \\
& \text{add}
\end{aligned}
$$

Remark:

- In C, an array is a pointer. A declared array a is a pointer-constant, whose R-value is the start address of the array.
- Formally, we define for an array e: $\text{code}_\text{R}\ e\ \rho = \text{code}_\text{L}\ e\ \rho$
- In C, the following are equivalent (as L-values):

$$2[a] \qquad a[2] \qquad a + 2$$

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.

5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere.
Alternatively, one could manage a separate environment ρ_{st} for each structure type st.

Be **struct** { **int** a; **int** b; } x; part of a declaration list.

- x has as relative address the address of the first cell allocated for the structure.
- The components have addresses relative to the start address of the structure. In the example, these are $a \mapsto 0$, $b \mapsto 1$.

Let $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. We have

$$
\begin{aligned}
|t| &= \sum_{i=1}^{k} |t_i| \\
\rho\, c_1 &= 0 \quad \text{and} \\
\rho\, c_i &= \rho\, c_{i-1} + |t_{i-1}| \quad \text{for } i > 1
\end{aligned}
$$

We thus obtain:

$$
\begin{aligned}
\text{code}_{\text{L}}\ (e.c)\ \rho \quad = \quad & \text{code}_{\text{L}}\ e\ \rho \\
& \text{loadc}\ (\rho\, c) \\
& \text{add}
\end{aligned}
$$

Example:

Be **struct** { **int** a; **int** b; } x; such that $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

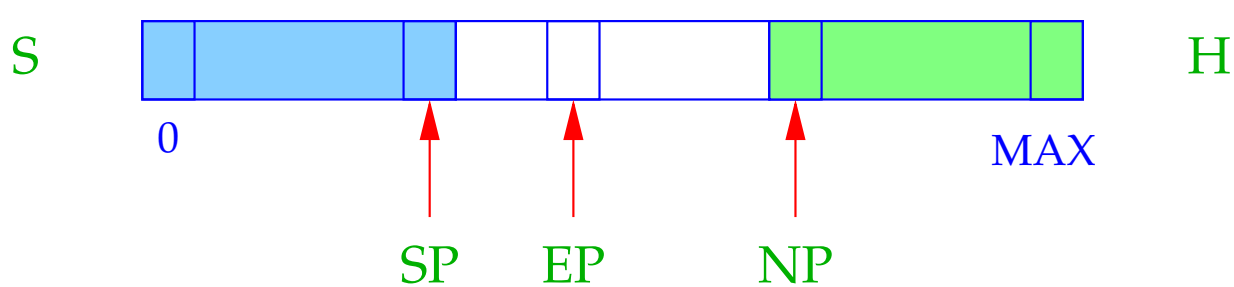
This yields:

$$\text{code}_{\text{L}}\ (x.b)\ \rho \quad = \quad \begin{array}{l} \text{loadc } 13 \\ \text{loadc } 1 \\ \text{add} \end{array}$$

6 Pointer and Dynamic Storage Management

Pointer allow the access to anonymous, dynamically generated objects, whose life time is not subject to the LIFO-principle.

$\Longrightarrow$ We need another potentially unbounded storage area H – the Heap.



NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

Idea:

- Stack and Heap grow toward each other in S, but must not collide. (Stack Overflow).
- A collision may be caused by an increment of SP or a decrement of NP.
- EP saves us the check for collision at the stack operations.
- The checks at heap allocations are still necessary.

What can we do with pointers (pointer values)?

- set a pointer to a storage cell,
- dereference a pointer, access the value in a storage cell pointed to by a pointer.

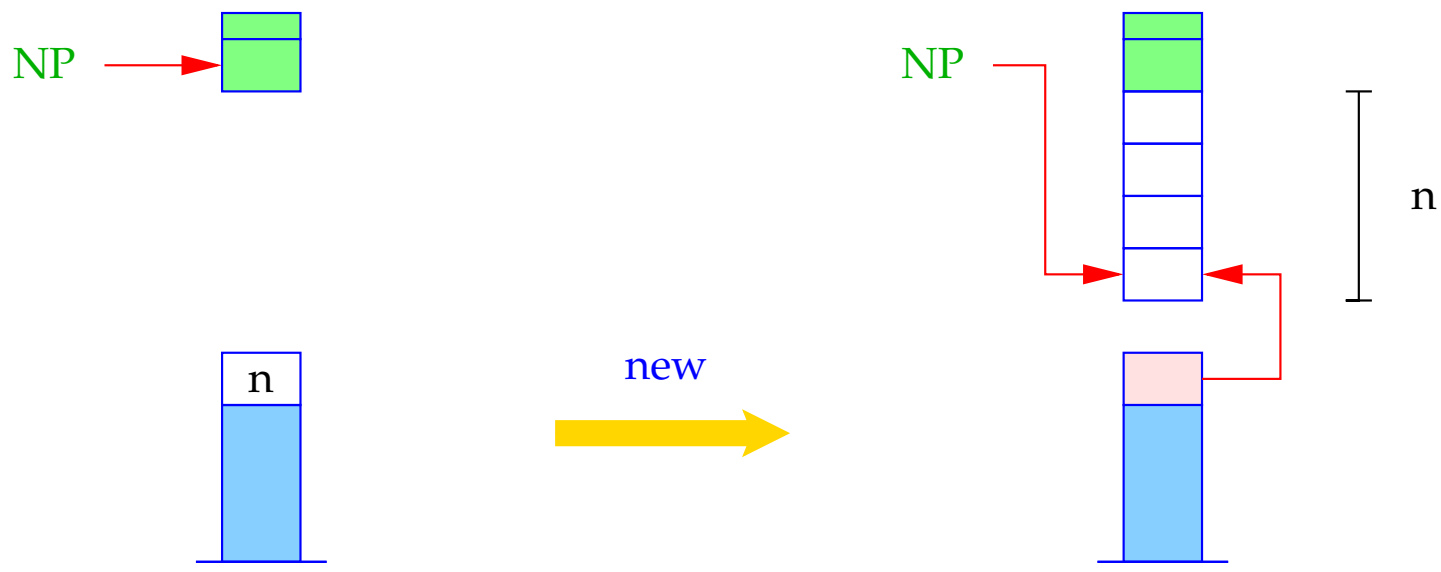
There a two ways to set a pointer:

(1) A call **malloc** (e) reserves a heap area of the size of the value of e and returns a pointer to this area:

$$\text{code}_\text{R}\ \textbf{malloc}\,(e)\ \rho \quad = \quad \text{code}_\text{R}\ e\ \rho$$
$$\text{new}$$

(2) The application of the address operator & to a variable returns a pointer to this variable, i.e. its address ($\widehat{=}$ L-value). Therefore:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$




```
if (NP - S[SP] ≤ EP)
        S[SP] = NULL;
else {
        NP = NP - S[SP];
        S[SP] = NP;
}
```

- NULL is a special pointer constant, identified with the integer constant 0.
- In the case of a collision of stack and heap the NULL-pointer is returned.