Simplification:

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

\[
  s \equiv \text{switch} \ (e) \ 
  \begin{cases} 
    \text{case } 0: & ss_0 \ \text{break}; \\
    \text{case } 1: & ss_1 \ \text{break}; \\
    \vdots \\
    \text{case } k-1: & ss_{k-1} \ \text{break}; \\
    \text{default: } & ss_k 
  \end{cases}
\]

\(s\) is then translated into the instruction sequence:
\[
\text{code } s \rho = \text{code}_R e \rho \\
\text{check } 0 k B \\
C_0: \text{code } ss_0 \rho \\
B: \text{jump } C_0 \\
\text{jump } D \\
\ldots \\
C_k: \text{code } ss_k \rho \\
D: \ldots \\
\text{jump } D
\]

- The Macro \text{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 \textit{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 $d \equiv t_1 x_1; \ldots t_k x_k; \ (t_i \text{ basic type})$ the address environment $\rho$ such that

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

Example: \texttt{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: `| · |`), 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 \( d \equiv t_1 x_1; \ldots t_k x_k; \)

\[
\rho x_1 = 1 \\
\rho x_i = \rho x_{i-1} + |t_{i-1}| \quad \text{for } i > 1
\]

Since `| · |` can be computed at compile time, also \( \rho \) can be computed at compile time.
Task:

Extend $\text{code}_L$ and $\text{code}_R$ 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 $\rho a + |t| \cdot (R\text{-value of } i)$.

In consequence:

\[
\text{code}_L \ a[e] \ \rho \quad = \quad \text{loadc} \ (\rho \ a) \\
\text{code}_R \ e \ \rho \\
\text{loadc} \ |t| \\
\text{mul} \\
\text{add}
\]

… or more general:
\[
\text{code}_L \ e_1[e_2] \ \rho \quad = \quad \text{code}_R \ e_1 \ \rho \\
\text{code}_R \ e_2 \ \rho \\
\text{loadc} |t| \\
\text{mul} \\
\text{add}
\]

**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 \( e \): \( \text{code}_R \ e \ \rho = \text{code}_L \ e \ \rho \)
- In C, the following are equivalent (as L-values):
  \[
  2[a] \quad a[2] \quad a + 2
  \]

**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 $\rho_{st}$ for each structure type $st$.

Be \textbf{struct} \{ \texttt{int} a; \texttt{int} b; \} \texttt{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 \( t \equiv \text{struct} \{ t_1 \ c_1; \ldots t_k \ c_k; \} \). We have

\[
|t| = \sum_{i=1}^{k} |t_i|
\]

\[
\rho c_1 = 0 \quad \text{and} \quad \rho c_i = \rho c_{i-1} + |t_{i-1}| \quad \text{for } i > 1
\]

We thus obtain:

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

\[
\text{loadc} (\rho \ c) \quad \text{add}
\]
Example:

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

This yields:

\[
\text{code}_{L} \ (x.b) \ \rho \quad = \quad \text{loadc} \ 13 \\
\quad \quad \quad \quad \text{loadc} \ 1 \\
\quad \quad \quad \quad \text{add}
\]
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.

\[ \Rightarrow \quad \text{We need another potentially unbounded storage area } H \text{ – the Heap.} \]

\[
\begin{array}{ccccccc}
S & | & | & | & | & | & H \\
\hline
0 & | & | & | & | & | \\
SP & | & | & | & | & NP \\
| & | & | & | & | & MAX
\end{array}
\]

\[ \begin{align*}
NP & \equiv \text{New Pointer}; \text{ points to the lowest occupied heap cell.} \\
EP & \equiv \text{Extreme Pointer}; \text{ points to the uppermost cell, to which } SP \text{ can point} \\
& \quad \text{(during execution of the actual function).}
\end{align*} \]
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 are 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:

   \[
   \text{code}_R \ malloc(e) \ \rho = \text{code}_R \ e \ \rho \quad \text{new}
   \]

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

   \[
   \text{code}_R \ (&e) \ \rho = \text{code}_L \ e \ \rho
   \]
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.