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 \( B \).

At the end of each alternative is an unconditional jump out of the switch-statement.

Simplification:

We only regard switch-statements of the following form:

\[
s \equiv \text{switch} \ (e) \{
  \text{case 0: } s_{00}, \text{break;}
  \text{case 1: } s_{01}, \text{break;}
  \vdots
  \text{case } k-1: s_{0k-1}, \text{break;}
  \text{default: } s_{0k}
}\]

\( s \) is then translated into the instruction sequence:

\[
\begin{align*}
\text{check 0 k B} & \quad \text{dup} \quad \text{dup} \quad \text{jumpi B} \\
\text{loadc 0} & \quad \text{loadc k} \quad A: \quad \text{pop} \\
\text{eq} & \quad \text{le} \quad \text{loadc k} \\
\text{jumpz A} & \quad \text{jumpz A} \quad \text{jumpi B}
\end{align*}
\]

- 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 \( \text{dup} \).
- The R-value of \( e \) is replaced by \( k \) before the indexed jump is executed if it is less than 0 or greater than \( k \).
code $s \rho = \text{code}_{\varepsilon \rho}$
check $0 \leq k \leq B$

$C_6$: code $s_{0_{\varepsilon}} \rho$
  jump D

$\ldots$

$C_\varepsilon$: code $s_{s_{\varepsilon}} \rho$
  jump D

- The Macro check $0 \leq k \leq B$ checks, whether the R-value of $\varepsilon$ 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 $0 \leq k \leq B$

\begin{align*}
\text{check} & \quad \text{dup} \quad \text{dup} \quad \text{jumpi} \quad B \\
& \quad \text{loadc} \quad \text{loadc} \quad \text{A} \quad \text{pop} \\
& \quad \text{geq} \quad \text{le} \quad \text{loadc} \quad \text{k} \\
& \quad \text{jumpz} \quad \text{A} \quad \text{jumpz} \quad \text{A} \quad \text{jumpi} \quad B \\
\end{align*}

- The R-value of $\varepsilon$ 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 $\varepsilon$ is replaced by $k$ before the indexed jump is executed if it is less than 0 or greater than $k$. 

\begin{align*}
\text{check} & \quad \text{dup} \quad \text{dup} \quad \text{jumpi} \quad B \\
& \quad \text{loadc} \quad \text{loadc} \quad \text{A} \quad \text{pop} \\
& \quad \text{geq} \quad \text{le} \quad \text{loadc} \quad \text{k} \\
& \quad \text{jumpz} \quad \text{A} \quad \text{jumpz} \quad \text{A} \quad \text{jumpi} \quad B \\
\end{align*}

- The R-value of $\varepsilon$ 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 $\varepsilon$ 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 \textit{definitely} in the interval $[0, k]$, the macro check is not needed.

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 \textit{definitely} in the interval $[0, k]$, the macro check is not needed.

The Macro check $0 \leq k \leq B$ 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.
code \( s \rho \) = code \( e \rho \) check 0 \( k \) B

\[\begin{align*}
C_0: & \quad \text{jump } D \\
C_1: & \quad \text{jump } \ldots \\
C_2: & \quad \text{jump } D
\end{align*}\]

- The Macro check \( 0 \leq k \) 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.

5 Storage Allocation for Variables

Goal:

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

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 \( d = t_1 x_1; \ldots; t_i x_i \) (\( t_i \) basic type) the address environment \( \rho \) such that

\[\rho x_i = i, \quad i = 1, \ldots, k\]

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 \( \alpha \) instead of 0, we have to decrease the R-value of \( \epsilon \) by \( \alpha \) before using it as an index.
- If all potential values of \( \epsilon \) are definitely in the interval \([0, k]\), the macro check is not needed.

5.1 Arrays

Example: \( \text{int [11] } a; \)

The array \( a \) consists of 11 components and therefore needs 11 cells.

\[\begin{align*}
& a[10] \\
& \vdots \\
& a[0]
\end{align*}\]
5.1 Arrays

Example:  \texttt{int [11] a;}

The array \texttt{a} consists of 11 components and therefore needs 11 cells.
\( \rho a \) is the address of the component \( a[0] \).

\begin{align*}
\texttt{a[10]} \\
\vdots \\
\texttt{a[0]}
\end{align*}

We need a function \texttt{sizeof} (notation: \texttt{\textbackslash{}cdot \textbackslash{}cdot}), computing the space requirement of a type:

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

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 \texttt{\textbackslash{}cdot \textbackslash{}cdot} can be computed at compile time, also \( \rho \) can be computed at compile time.

Task:

Extend \texttt{code\_1} and \texttt{code\_2} 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

\( \rho a + |t| \) (R-value of \( t \)).

In consequence:

\begin{align*}
\text{code\_1, a[i]} \quad \rho & = \quad \text{loadc} (\rho a) \\
\text{code\_2} \quad e \quad \rho & = \quad \text{loadc} [t]\text{mul} \\
\text{add} & \quad \text{or more general:}
\end{align*}
Task:

Extend \texttt{code}\_1 and \texttt{code}\_2 to expressions with accesses to array components.

Be \( t[i] \) \( a \) the declaration of an array \( a \).

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

In consequence:

\[
\text{code}_2 e[a] \rho = \text{loadc} (\rho a) \\
\text{code}_2 e \rho \\
\text{loadc} |t| \\
mul \\
add
\]

... or more general:

\[
\text{code}_2 e_1[e_2] \rho = \text{code}_2 e_1 \rho \\
\text{code}_2 e_2 \rho \\
\text{loadc} |t| \\
mul \\
add
\]

Remark:

- In C, an array is a \texttt{pointer}. A declared array \( a \) is a \texttt{pointer-constant}, whose R-value is the start address of the array.
- Formally, we define for an array \( e \): \( \text{code}_1 e \rho = \text{code}_1 e \rho \)
- 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.

\[
\text{code}_1 e_1[e_2][\rho] = \text{code}_1 e_2[\rho]
\]

5.2 Structures

In \texttt{Modula} and \texttt{Pascal}, structures are called Records.

Simplification:

Names of structure components are not used elsewhere. Alternatively, one could manage a separate environment \( \rho_\text{st} \) for each structure type \( \text{st} \).

Be \( \text{struct} \{ \text{int} a; \text{int} b; \} x; \) 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 \( a \leftrightarrow 0, b \leftrightarrow 1 \).
Let $t = \{ t_1, c_1; \ldots; t_k, c_k \}$. We have

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

$\rho c_1 = 0$ and

$\rho c_i = \rho c_{i-1} + |t_{i-1}| \text{ for } i > 1$

We thus obtain:

\[
\begin{align*}
\text{code}_{\ell}(x.c) & \quad \rho \quad = \quad \text{code}_{\ell} \; \rho \\
& \quad \text{loadc}(\rho \circ c) \\
& \quad \text{add}
\end{align*}
\]

Example:

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

\[
\begin{align*}
\text{code}_{\ell}(x.b) \; \rho & \quad = \quad \text{loadc } 13 \\
& \quad \text{loadc } 1 \\
& \quad \text{add}
\end{align*}
\]

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.

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

\[
\begin{array}{c}
\text{SP} \\
0 \\
\text{EP} \\
\text{NP} \\
\text{MAX}
\end{array}
\]

NP  $\Downarrow$ New Pointer; points to the lowest occupied heap cell.
EP  $\Downarrow$ 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 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:
   
   ```
   code_R malloc(e) ρ = code_R e ρ
   ```

2. The application of the address operator `&` to a variable returns a pointer to this variable, i.e. its address (≡ L-value). Therefore:
   
   ```
   code_R (&e) ρ = code_L e ρ
   ```

---

**Derereferencing 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`:

```
code_L (*e) ρ = code_R e ρ
```

**Example:**

Given the declarations

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

and the expression `((pt->b) -> a)[i + 1]`

Because of  

```c
 e → a ≡ (*e).a 
```

holds:

```c
 code_L (e → a) ρ = code_R e ρ  
loadc (ρa)  
add
```

---

**Dereferencing 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`:

```
code_L (*e) ρ = code_R e ρ
```

**Example:**

Given the declarations

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

and the expression `((pt → b) → a)[i + 1]`

Because of  

```c
 e → a ≡ (*e).a 
```

holds:

```c
 code_L (e → a) ρ = code_R e ρ  
loadc (ρa)  
add
```