We define:

\[
\begin{align*}
\text{code}_{E} (e_1 + e_2) \rho &= \text{code}_{E} e_1 \rho \\
&\quad \text{code}_{E} e_2 \rho \\
&\quad \text{add} \\
&\quad \text{... analogously for the other binary operators} \\
\text{code}_{E} (-e) \rho &= \text{code}_{E} e \rho \\
&\quad \text{neg} \\
&\quad \text{... analogously for the other unary operators} \\
\text{code}_{E} q \rho &= \text{loade} q \\
\text{code}_{E} x \rho &= \text{loade} (\rho \times) \\
&\quad \text{...}
\end{align*}
\]
code \( x \rho \) = code\(_l\) \( x \rho \)

load

The instruction load loads the contents of the cell, whose address is on top of the stack.

\[ S[SP] = S[S[SP]] \]

store writes the contents of the second topmost stack cell into the cell, whose address is on top of the stack, and leaves the written value on top of the stack.

Note: this differs from the code generated by gcc ??

\[ S[S[SP]] = S[S[SP-1]] \]

SP \( \sim \)

Simplification:

We only regard switch-statements of the following form:

\[
s \equiv \text{switch}(c) \left\{ \right. \\
\text{case 0: } s_{s_0} \text{ break;}
\text{case 1: } s_{s_1} \text{ break;}
\vdots
\text{case } k - 1: s_{s_{k-1}} \text{ break;}
\text{default: } s_{s_k}
\left\} \right.
\]

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

\[
\text{code } s \rho = \text{code}_l \rho
\]

\[
\text{check 0 } k \text{ B}
\]

\[
\text{jump D}
\]

\[
\text{B: jump C}_0 ... \\
\text{C}_k: \text{code } s_{s_k} \rho \text{ jump D}
\]

- The Macro check 0 \( k \) B checks whether the R-value of \( r \) 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:
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 $d \equiv x_1, \ldots, x_k; \ldots$ (basic type) the address environment $\rho$ such that

\[
\rho(x_i) = i, \quad i = 1, \ldots, k
\]
We need a function \( \text{sizeof} \) (notation \( |t| \)), computing the space requirement of a type:

\[
|t| = \begin{cases} 
1 & \text{if } t \text{ basic} \\
|t_1| + |t_2| & \text{if } t = t_1 \times t_2 
\end{cases}
\]

Accordingly, we obtain for the declaration \( d \equiv t_1 \times \ldots \times t_k \):

\[
\rho x_i = \begin{cases} 
1 & \text{if } i = 1 \\
\rho x_{i-1} + |t_{i-1}| & \text{for } i > 1 
\end{cases}
\]

Since \( |\cdot| \) can be computed at compile time, also \( \rho \) can be computed at compile time.

**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 \( c \):
  \[ \text{code}_c \ e \ \rho = \text{code}_e \ e \ \rho \]
- In C, the following are equivalent (as L-values):
  \[ 2[a] \]  
  \[ 2[a] = 2 \]

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

---

**Task:**

Extend \( \text{code}_a \) and \( \text{code}_l \) 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 \( \rho \ a \ |t| + (R\text{-value of } t) \).

In consequence:

\[
\text{code}_a \ e[c] \ \rho = \begin{cases} 
\text{load} \ (\rho \cdot a) & \text{if } e \ \rho = e \\
\text{load} \ t & \text{if } e \ \rho = c \\
\text{mul} & \\
\text{add} & 
\end{cases}
\]

... or more general:

---

**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:**

\[
\text{typedef } a \text{ struct } \{ \text{int} \ a; \text{int} \ b; \text{int} \ c; \} \text{ part of a declaration list.}
\]

- \( a \) 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 \( a \mapsto 0, b \mapsto 1 \).
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 \( \rho_s \) for each structure type \( st \).

Be

\[
\text{struct \{ \text{int } a; \text{ int } b; \} \ x; \quad \text{part of a declaration list.}
\]

- \( x \) 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 \( a \mapsto 0, b \mapsto 1 \).

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.

\( \rightarrow \) We need another potentially unbounded storage area \( H \) – the Heap.

NP \( \equiv \) New Pointer; points to the lowest occupied heap cell.
EP \( \equiv \) 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:

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

2. The application of the address operator \& to a variable returns a **pointer** to this variable, i.e. its address (\( \in \) L-value). Therefore:

   \[ \text{code} \; \&e \; \rho = \text{code} \; e \; \rho \]

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:

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

Example:

Given the declarations

\[
\begin{align*}
\text{struct } t \{ & \text{ int } a[7]; \text{ struct } t \; *p; \}; \\
\text{ int } i, j;
\end{align*}
\]

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

Because of \( e \rightarrow a = (\star e)_a \) holds:

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

\[
\begin{align*}
\text{load} \; (\rho a) \quad \text{add} \quad -
\end{align*}
\]

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

\[
\text{code}_\mathcal{L} ((pt \to b) \to a) \rho = \text{code}_\mathcal{L} (pt \to b) \rho = \text{loada} 3 \text{ loada} 0 \text{ add} \\
\text{loada} 7 \text{ add} \\
\text{add}
\]

In total, we obtain the instruction sequence:

\[
\begin{align*}
\text{loada} 3 & \quad \text{load} & \quad \text{loada} 1 & \quad \text{loada} 1 \\
\text{loada} 7 & \quad \text{loada} 0 & \quad \text{loada} 1 & \quad \text{mul} \\
\text{add} & \quad \text{add} & \quad \text{add} & \quad \text{add}
\end{align*}
\]

7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\begin{align*}
\text{code}_\mathcal{L} (e_1 [e_2]) \rho &= \text{code}_\mathcal{L} e_1 \rho \\
\text{code}_\mathcal{L} e_2 \rho &\text{ loadc } [t] \\
&\text{ mul} \quad \text{add} \quad \text{if } e_1 \text{ has type } t\times \text{ or } t[] \\
\text{code}_\mathcal{L} (e \cdot a) \rho &= \text{code}_\mathcal{L} e \rho \\
&\text{ loadc } (\rho a) \\
&\text{ add}
\end{align*}
\]
7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\text{code}_\varepsilon (e_1 e_2) \rho = \text{code}_\varepsilon e_1 \rho \\
\text{code}_\varepsilon e_2 \rho \\
\text{loade} |!| \text{ mul} \\
\text{add} \quad \text{if } e_1 \text{ has type } t\times \text{ or } t[] \\
\text{code}_\varepsilon (e, a) \rho = \text{code}_\varepsilon e \rho \\
\text{loade} (a a) \\
\text{add} \\
\varepsilon_1
\]

\[
\text{code}_\varepsilon (*e) \rho = \text{code}_\varepsilon e \rho \\
\text{code}_\varepsilon x \rho = \text{loade} (\rho x) \\
\text{code}_\varepsilon (e \& e) \rho = \text{code}_\varepsilon e \rho \\
\text{code}_\varepsilon e \rho = \text{code}_\varepsilon e \rho \\
\text{if } e \text{ is an array} \\
\text{code}_\varepsilon (e_1 \triangledown e_2) \rho = \text{code}_\varepsilon e_1 \rho \\
\text{code}_\varepsilon e_2 \rho \\
\text{op} \quad \text{op instruction for operator } \triangledown \\
\varepsilon_2
\]

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

For the statement: \[\text{\&a } = 5;\] we obtain:

\[
\text{code}_\varepsilon \text{\&a } = 5; \rho = \text{code}_\varepsilon \text{\&a } = \text{code}_\varepsilon \text{\&a } = \text{loade} 7 \\
\text{code}_\varepsilon (\text{\&a } = 5) \rho = \text{loade} 7 \\
\text{\&a } = 5; \rho = \text{loade} 7 \\
\text{loade} 7 \\
\text{pop} \\
\varepsilon_3
\]

As an exercise translate:

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

\[\text{fragments} \]

\[\text{free}\]

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

\[\Rightarrow \text{Function names must be maintained within the address environment!}\]
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:

```
     main
      /|
     / |    printf
    /   |
   /     |
  fac   fac
   |
  fac
```

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:

```c
code_r f \rho = \text{load } e_{-f} = \text{start address of the code for } f
```

Function names must be maintained within the address environment!
9.1 Memory Organization for Functions

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

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 $(tag, a) \in \{G, L\} \times Z$.

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;
} = i;
2 main () {
    int i;
    scanf("%d", &i);
    scanlist(&i);
1 int ith (struct list * x, int i) {
    if (i <= 1) return x -> info;
    else return ith (x -> next, i - 1);
}
```