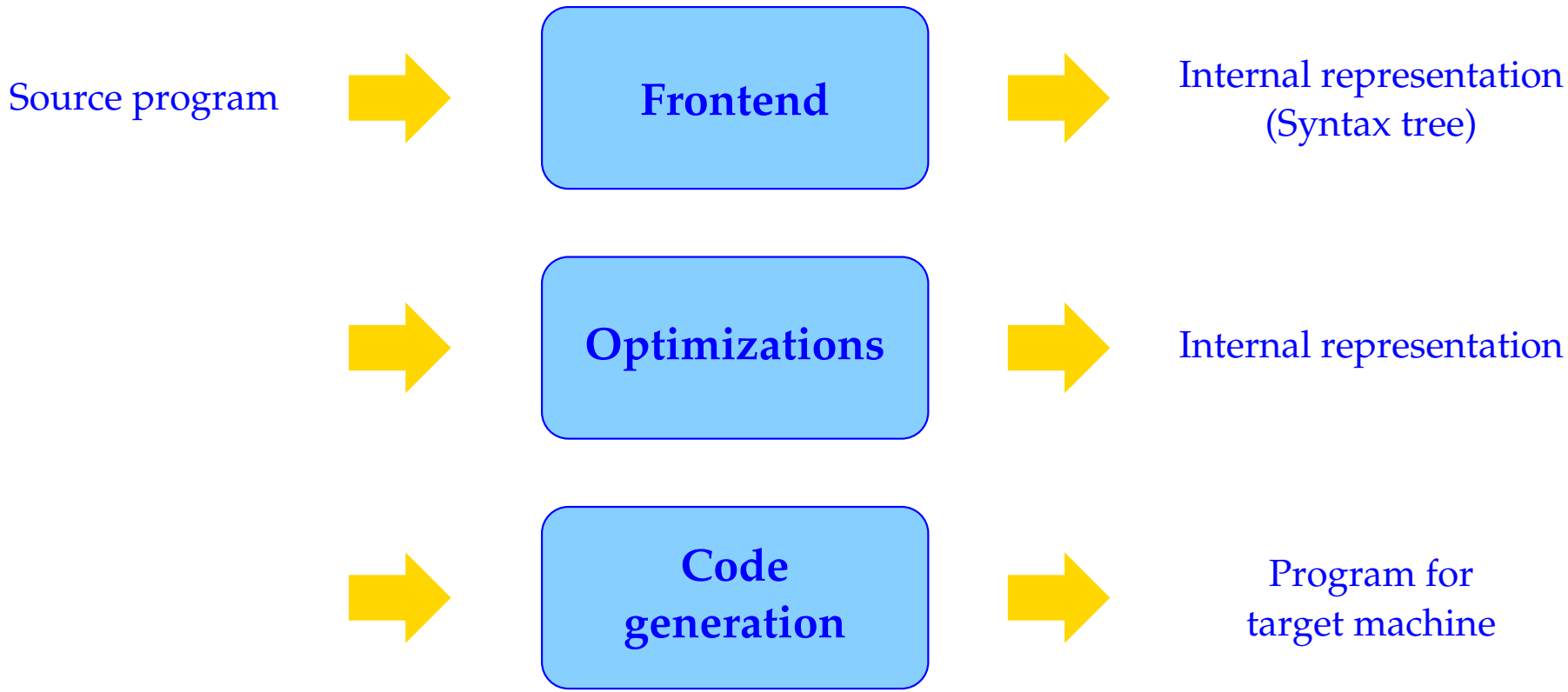
Reinhard Wilhelm + Helmut Seidl

# The Translation of C

*Saarbrücken + München*

## Structure of a compiler:

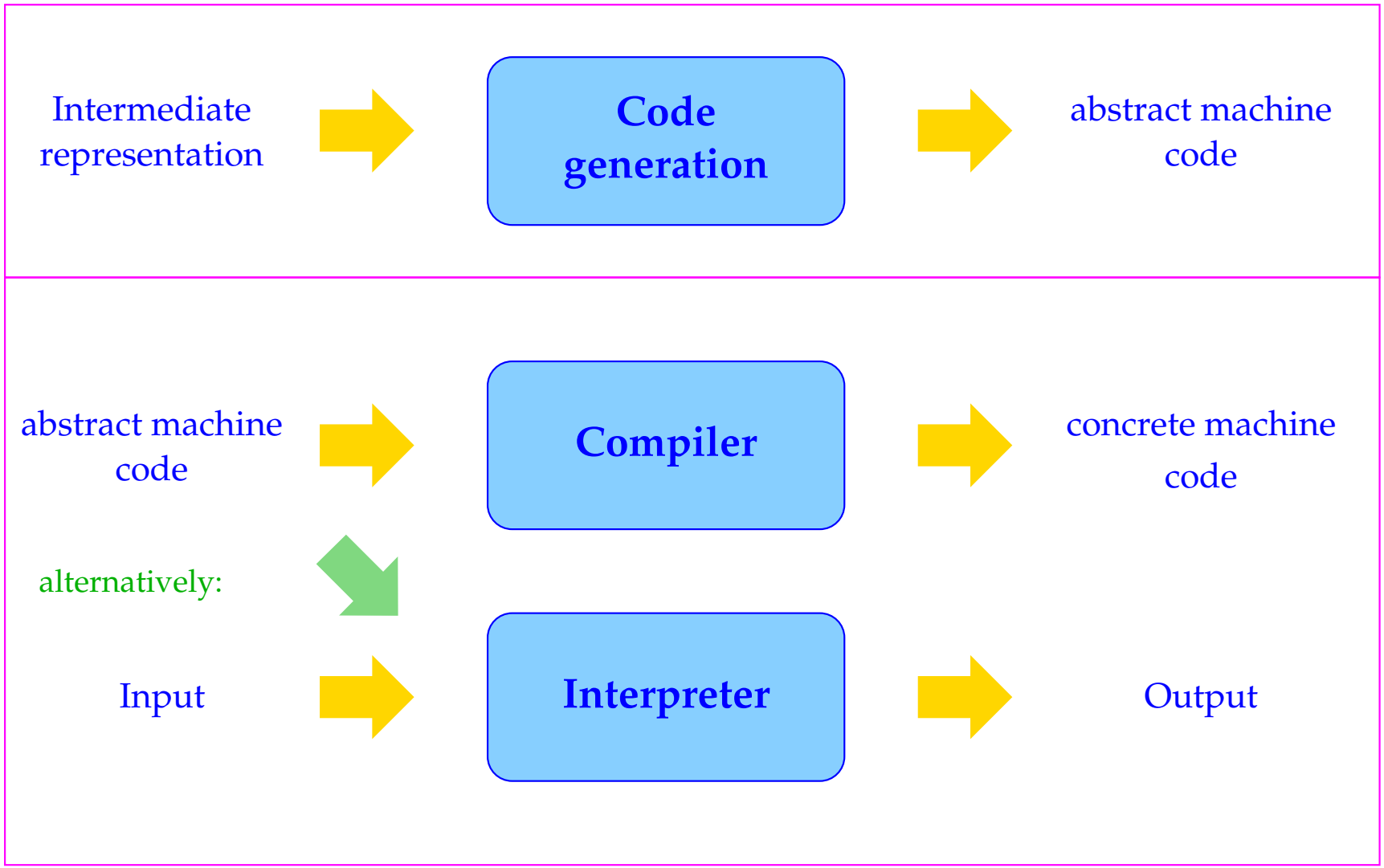


## Subtasks in code generation:

Goal is a good exploitation of the hardware resources:

1. Instruction Selection: Selection of efficient, semantically equivalent instruction sequences;
2. Register Allocation: Best use of the available processor registers
3. Instruction Scheduling: Reordering of the instruction stream to exploit intra-processor parallelism

For several reasons, e.g. modularization of code generation and portability, code generation may be split into two phases:


Intermediate representation
Code generation
abstract machine code
abstract machine code
Compiler
concrete machine code
alternatively:
Input
Interpreter
Output

## Abstract machine

- idealized architecture,
- simple code generation,
- easily implemented on real hardware.

## Advantages:

- Porting the compiler to a new target architecture is simpler,
- Modularization makes the compiler easier to modify,
- Translation of program constructs is separated from the exploitation of architectural features.

Abstract machines for some programming languages:

| | | | |
|---|---|---|---|
| Algol 60 | $\longrightarrow$ | Algol Object Code | |
| Pascal | $\longrightarrow$ | P-machine | |
| SmallTalk | $\longrightarrow$ | Bytecode | |
| Prolog | $\longrightarrow$ | WAM | ("Warren Abstract Machine") |
| SML, Haskell | $\longrightarrow$ | STGM | |
| Java | $\longrightarrow$ | JVM | |

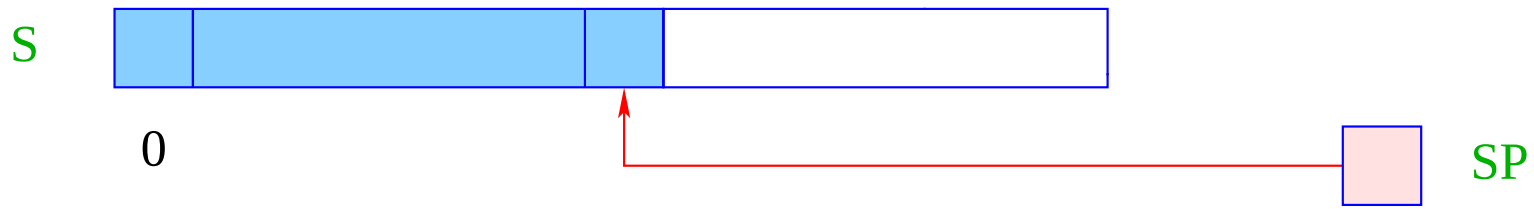
# The Translation of C

# 0 The Architecture of the CMa

- Each abstract machine provides a set of instructions
- Instructions are executed on the abstract hardware
- This abstract hardware can be viewed as a set of arrays and registers, which the instructions access
- ... and which are managed by the run-time system

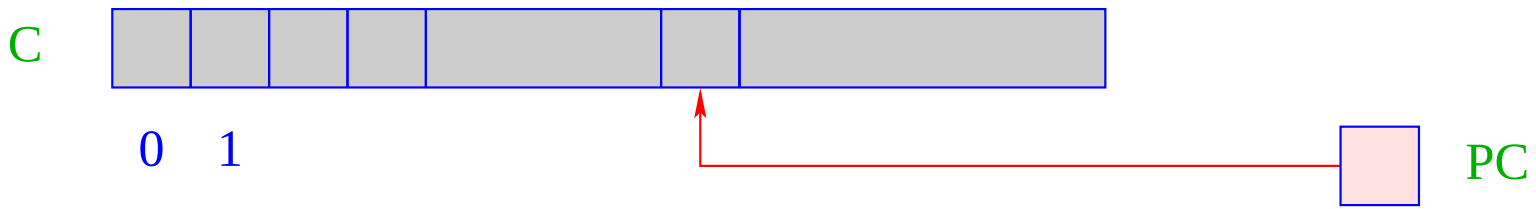
For the CMa we need:

## The Data Store:




- S is the (data) store, onto which new cells are allocated in a LIFO discipline
  $\Longrightarrow$ Stack.
- SP ($\widehat{=}$ Stack Pointer) is a register, which contains the address (index) of the topmost allocated cell,
  Simplification: All types of scalar data fit into one cell of S.

## The Code/Instruction Store:




- C is the Code store, which contains the program.

  Each cell of field C can store exactly one abstract instruction.

- PC ($\widehat{=}$ Program Counter) is a register, which contains the address (index) of the instruction to be executed next.

- Initially, PC contains the address 0.

  $\Longrightarrow$ C[0] contains the instruction to be executed first.

## Execution of Programs: (the main cycle of the machine)

- The machine loads the instruction in C[PC] into a Instruction-Register IR and executes it
- PC is incremented by 1 before the execution of the instruction

```
while (true) {
        IR = C[PC]; PC++;
        execute (IR);
}
```

- The execution of the instruction may overwrite the PC (jumps).
- The Main Cycle of the machine will be halted by executing the instruction halt , which returns control to the environment, e.g. the operating system
- More instructions will be introduced by demand

# 1 Simple expressions and assignments

Problem: evaluate the expression $(1 + 7) * 3$ !

More precisely: generate an instruction sequence, which

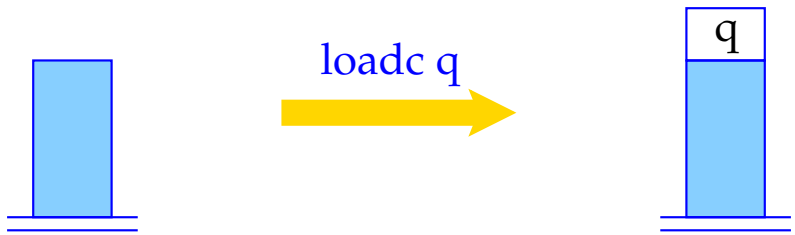
- determines the value of the expression and
- pushes it on top of the stack...

Idea:

- first compute the values of the subexpressions,
- save these values on top of the stack,
- then apply the operator, which leaves the result on top of the stack.

## The general principle:

- instructions expect their (implicit) operands on top of the stack,
- execution of an instruction consumes its operands,
- results, if any, are stored on top of the stack.

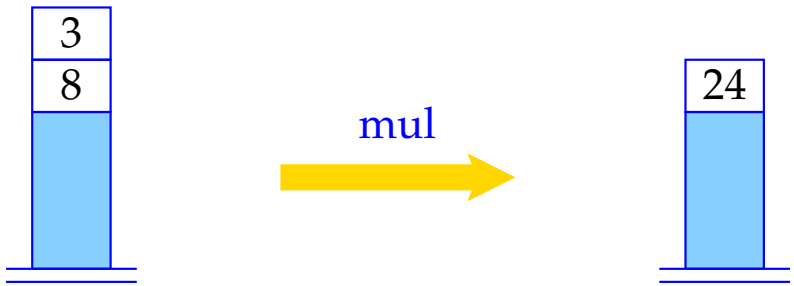



$SP \leftarrow SP + 1;$

$S[SP] \leftarrow q;$

Instruction loadc q needs no operand on top of the stack, pushes the constant q onto the stack.

Note: the content of register SP is only implicitly represented, namely through the height of the stack.



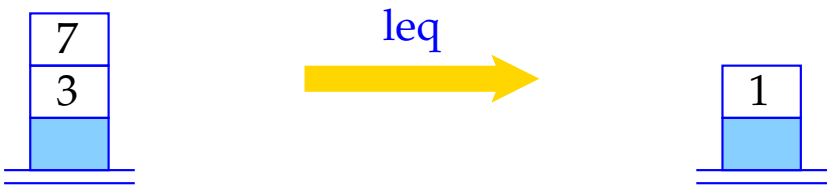

$SP \leftarrow SP - 1;$

$S[SP] \leftarrow S[SP] * S[SP+1];$

mul expects two operands on top of the stack, consumes both, and pushes their product onto the stack.

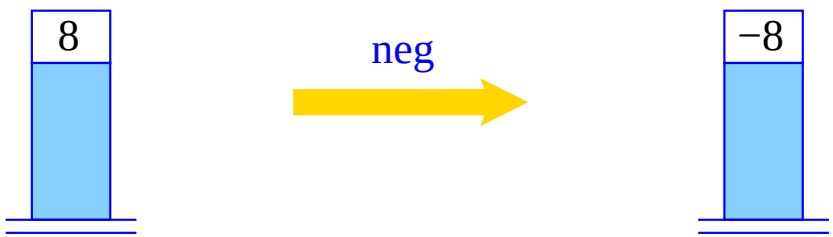
... the other binary arithmetic and logical instructions, add, sub, div, mod, and, or and xor, work analogously, as do the comparison instructions eq, neq, le, leq, gr and geq.

Example: The operator leq




Remark: 0 represents *false*, all other integers *true*.

Unary operators neg and not consume one operand and produce one result.

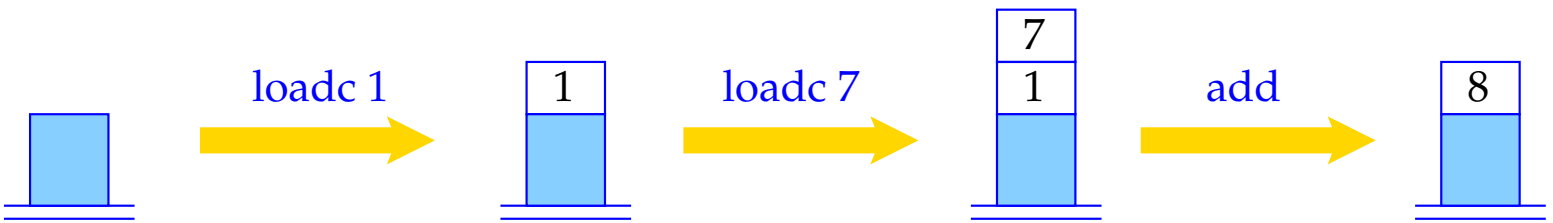



S[SP] ← – S[SP];

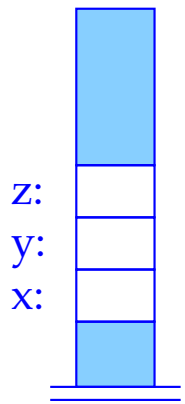
Example: Code for $1 + 7$:

loadc 1 loadc 7 add

Execution of this code sequence:



Variables are associated with cells in S:




Code generation will be described by some Translation Functions, code, $\text{code}_L$, and $\text{code}_R$.

Arguments: A program construct and a function $\rho$. $\rho$ delivers for each variable $x$ the relative address of $x$. $\rho$ is called Address Environment.

Variables can be used in two different ways:

## Example: $x = y + 1$

We are interested in the value of $y$, but in the address of $x$.

The syntactic position determines, whether the L-value or the R-value of a variable is required.

L-value of $x$ = address of $x$

R-value of $x$ = content of $x$

| | |
|---|---|
| $\text{code}_R\ e\ \rho$ | produces code to compute the R-value of $e$ in the address environment $\rho$ |
| $\text{code}_L\ e\ \rho$ | analogously for the L-value |

## Note:

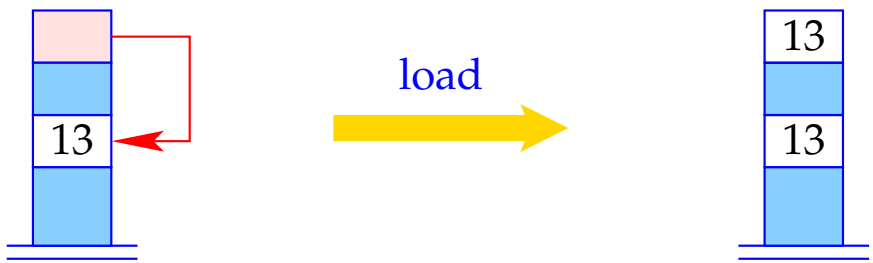
Not every expression has an L-value (Ex.: $x + 1$).

We define:

$$
\begin{array}{lcl}
\text{code}_\text{R}\ (e_1 + e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{add} \\
 & & \quad \text{... analogously for the other binary operators} \\
\text{code}_\text{R}\ (-e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\
 & & \text{neg} \\
 & & \quad \text{... analogously for the other unary operators} \\
\text{code}_\text{R}\ q\ \rho & = & \text{loadc q} \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & & \ldots
\end{array}
$$

$$\text{code}_R\ x\ \rho \quad = \quad \text{code}_L\ x\ \rho$$
$$\text{load}$$

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

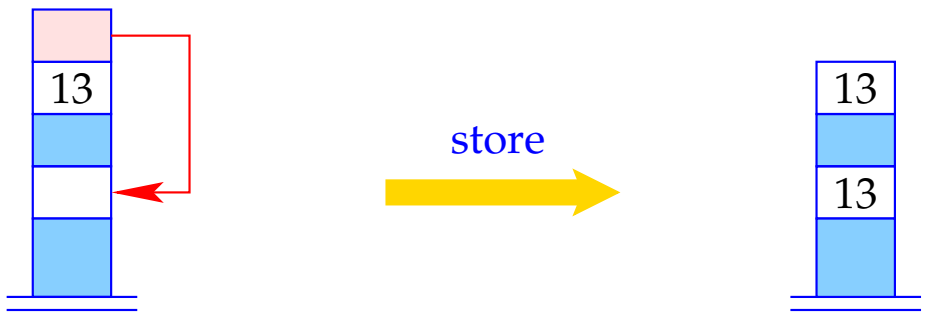



S[SP] ← S[S[SP]];

$$\begin{aligned} \text{code}_R\ (x = e)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \text{code}_L\ x\ \rho \\ & \text{store} \end{aligned}$$

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

Note: this is different from the corresponding store–instruction of the P–machine in Wilhelm/Maurer!




S[S[SP]] ← S[SP-1];

SP ← SP – 1;

## Example:

Code for $e \equiv x = y - 1$ with $\rho = \{x \mapsto 4, y \mapsto 7\}$.

$\text{code}_R\ e\ \rho$ produces:

```
loadc 7
load
```

```
load 1
sub
```

```
loadc 4
store
```

## Improvements:

Introduction of special instructions for frequently used instruction sequences, e.g.,

```
loada q     =   loadc q
                load

storea q    =   loadc q
                store
```

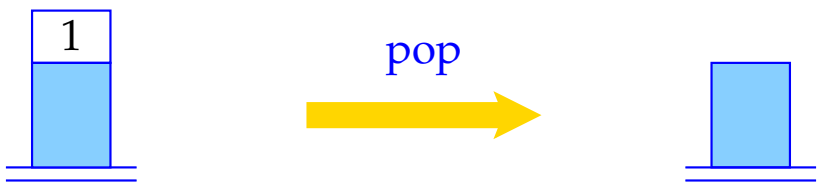
# 2 Statements and Statement Sequences

Is $e$ an expression, then $e;$ is a statement.

Statements do not deliver a value. The contents of the SP before and after the execution of the generated code must therefore be the same.

$$\text{code } e;\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

The instruction pop eliminates the top element of the stack.



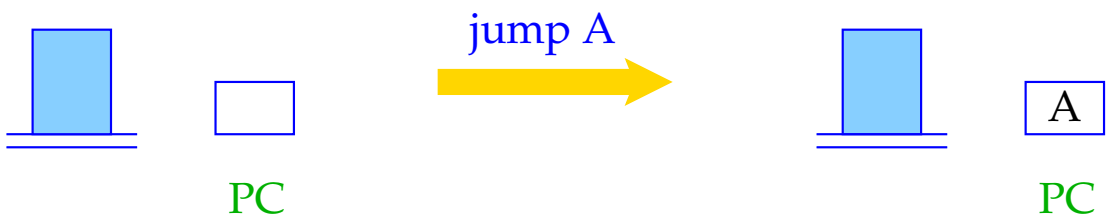

SP $\leftarrow$ SP – 1;

The code for a statement sequence is the concatenation of the code for the statements of the sequence:

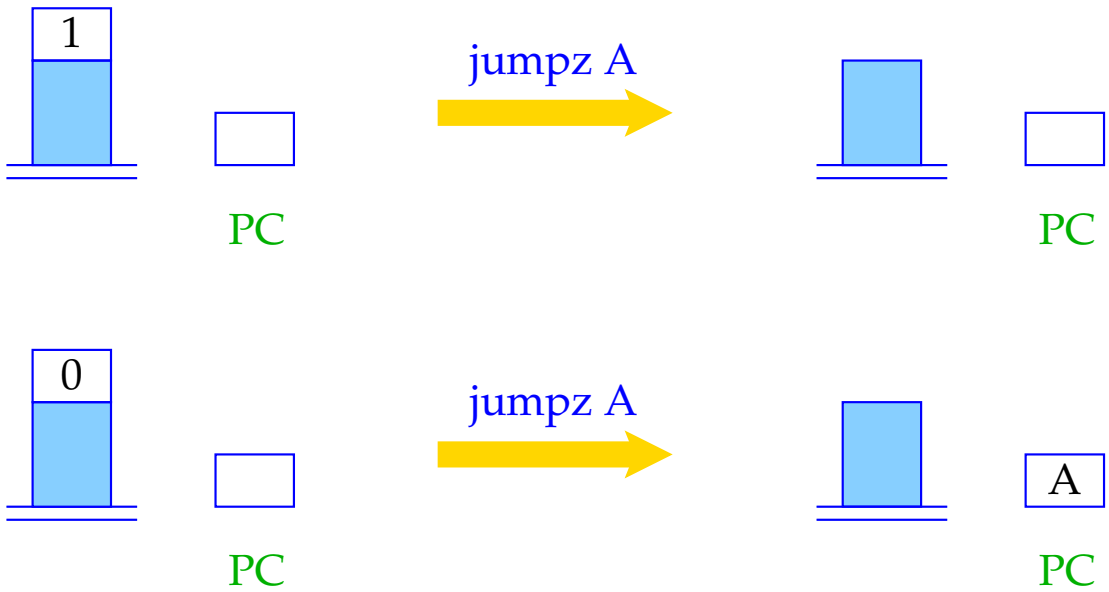
$$
\begin{array}{lcll}
\text{code}\ (s\ ss)\ \rho & = & \text{code}\ s\ \rho & \\
 & & \text{code}\ ss\ \rho & \\
\text{code}\ \varepsilon\ \rho & = & // & \textit{empty sequence of instructions}
\end{array}
$$

# 3 Conditional and Iterative Statements

We need jumps to deviate from the serial execution of consecutive statements:




PC ← A;




if (S[SP] == 0) PC ← A;

SP ← SP – 1;

For ease of comprehension, we use symbolic jump targets. They will later be replaced by absolute addresses.

Instead of absolute code addresses, one could generate relative addresses, i.e., relative to the actual PC.

## Advantages:

- smaller addresses suffice most of the time;
- the code becomes relocatable, i.e., can be moved around in memory.

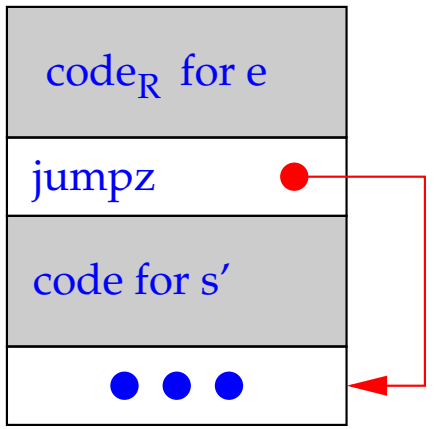
## 3.1 One-sided Conditional Statement

Let us first regard $s \equiv \textbf{if}\ (e)\ s'$.

Idea:

- Put code for the evaluation of $e$ and $s'$ consecutively in the code store,
- Insert a conditional jump (jump on zero) in between.

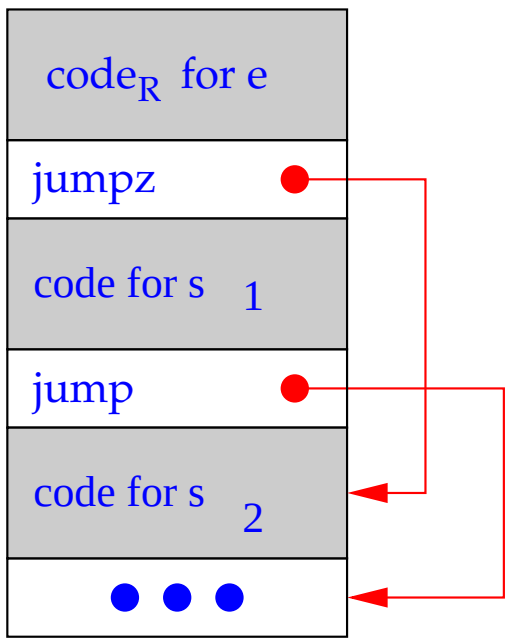
$$
\begin{array}{lcl}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s'\ \rho \\
 & \text{A}: & \ldots
\end{array}
$$



## 3.2 Two-sided Conditional Statement

Let us now regard $s \equiv \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2$. The same strategy yields:

$$
\begin{array}{rcl}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code } s_1\ \rho \\
 & & \text{jump B} \\
\text{A}: & & \text{code } s_2\ \rho \\
\text{B}: & & \ldots
\end{array}
$$



Example: Be $\rho = \{x \mapsto 4, y \mapsto 7\}$ and

$$
\begin{array}{lll}
s \equiv & \textbf{if}\,(x > y) & (i) \\
& \quad x = x - y; & (ii) \\
& \textbf{else}\; y = y - x; & (iii)
\end{array}
$$

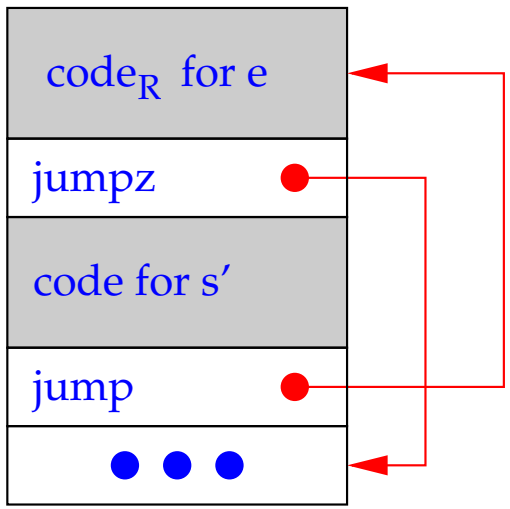
code $s$ $\rho$ produces:

| | | | | |
|---|---|---|---|---|
| loada 4 | | loada 4 | A: | loada 7 |
| loada 7 | | loada 7 | | loada 4 |
| gr | | sub | | sub |
| jumpz A | | storea 4 | | storea 7 |
| | | pop | | pop |
| | | jump B | B: | ... |
| $(i)$ | | $(ii)$ | | $(iii)$ |

## 3.3 while-Loops

Let us regard the loop $s \equiv \textbf{while}\ (e)\ s'$. We generate:

$$
\begin{array}{lll}
\text{code}\ s\ \rho & = & \\
 & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$



Example: Be $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and $s$ the statement:

$$\textbf{while}\ (a > 0)\ \{c = c + 1;\ a = a - b;\}$$

code $s\ \rho$ produces the sequence:

```
A:  loada 7     loada 9     loada 7     B:  ...
    loadc 0     loadc 1     loada 8
    gr          add         sub
    jumpz B     storea 9    storea 7
                pop         pop
                            jump A
```

## 3.4 for-Loops

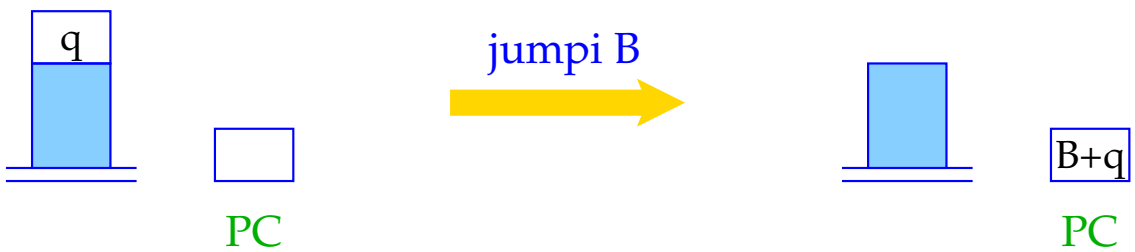
The **for**-loop $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ is equivalent to the statement sequence $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – provided that $s'$ contains no **continue**-statement.
We therefore translate:

$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_\text{R}\ e_1 \\
 & & \text{pop} \\
 & \text{A}: & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{code}_\text{R}\ e_3\ \rho \\
 & & \text{pop} \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$

## 3.5 The switch-Statement

Idea:

- Multi-target branching in constant time!
- Use a jump table, which contains at its $i$-th position the jump to the beginning of the $i$-th alternative.
- Realized by indexed jumps.




PC ← B + S[SP];

SP ← SP – 1;

## Simplification:

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

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

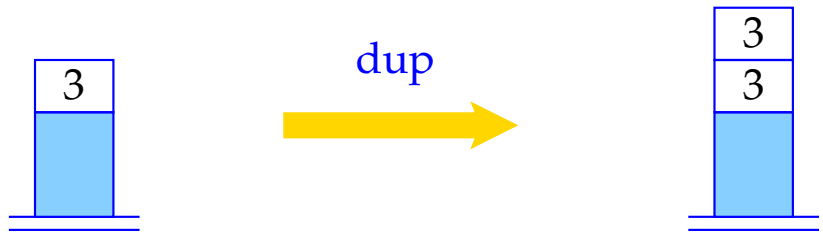
$s$ is then translated into the instruction sequence:

$$
\begin{array}{lcl@{\qquad}l@{\qquad}l@{\qquad}l}
\text{code } s\ \rho & = & \text{code}_R\ e\ \rho & C_0: & \text{code } ss_0\ \rho & B: \quad \text{jump } C_0 \\
 & & \text{check } 0\ k\ B & & \text{jump D} & \quad \ldots \\
 & & & & \ldots & \quad \text{jump } C_k \\
 & & & C_k: & \text{code } ss_k\ \rho & D: \quad \ldots \\
 & & & & \text{jump D} &
\end{array}
$$

- 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 ← SP + 1;

## 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.

# 4 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 defined, starting at address 1.

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

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

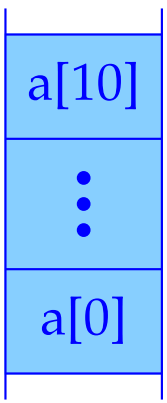
## 4.1 Arrays

A set of consecutive memory cells, of static size.

Access through integer indeces starting at 0.

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

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 definition $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

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

Since $|\cdot|$ can be computed at compile time, also $\rho$ 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 definition 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:

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

... or more general:

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

## Remark:

- In C, an array is a pointer. A defined 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.

## 4.2 Structures (Records)

A set of named components of possibly different types.

Access through the component names (selectors).

### Simplification:

Names of structure components are not used elsewhere.
Alternatively, one could manage a separate environment $\rho_{st}$ 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 $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

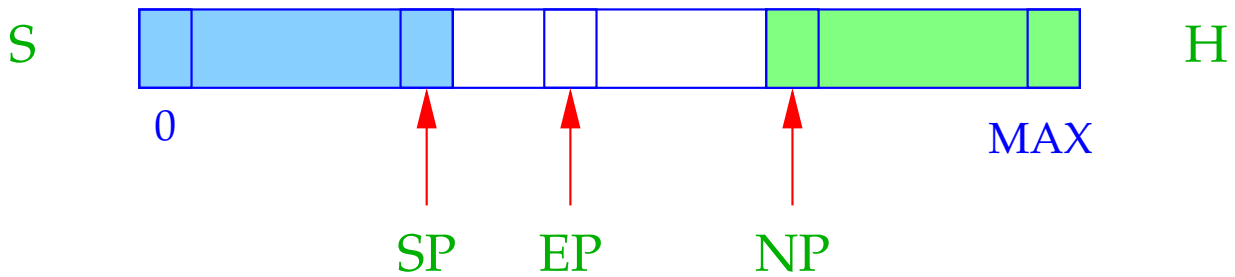
This yields:

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

# 5 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 current function).

## Idea:

- Stack and Heap grow towards 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.
- EP can be determined statically.
- 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, i.e. access the value in a storage cell pointed to by a pointer.

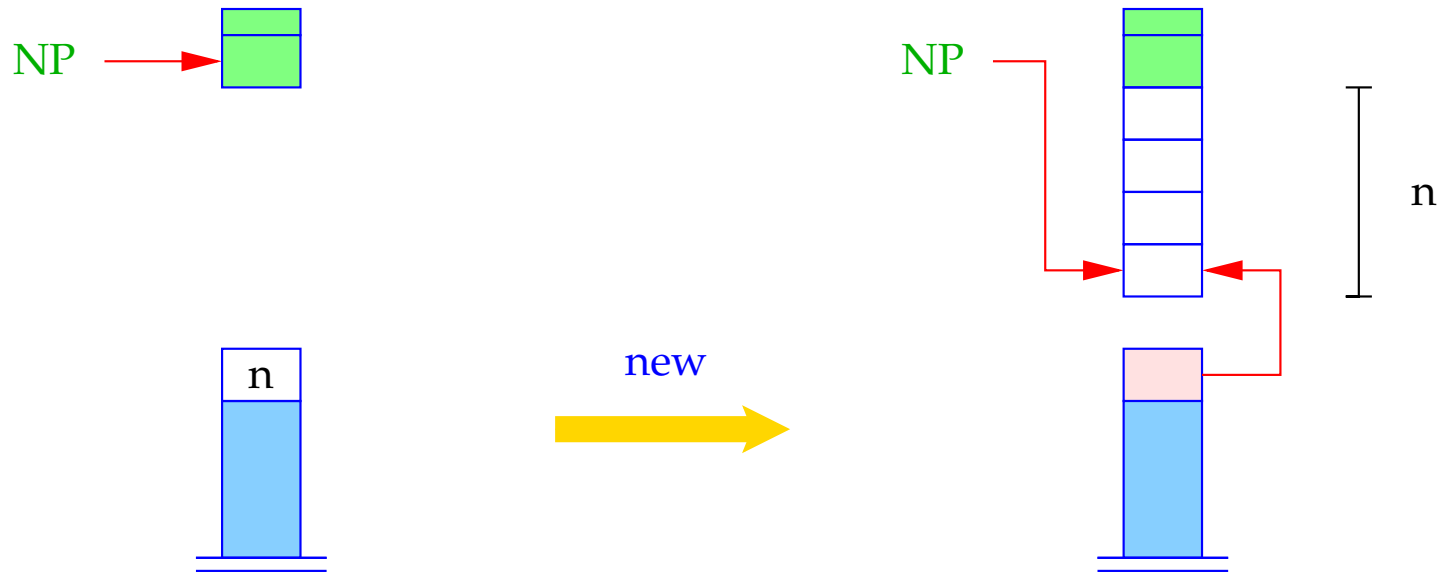
There a two ways to set a pointer:

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

$$\begin{aligned} \text{code}_\text{R}\ \textbf{malloc}\,(e)\ \rho \quad &= \quad \text{code}_\text{R}\ e\ \rho \\ & \qquad\ \ \text{new} \end{aligned}$$

(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.

## 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 $e$:

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

## Example:

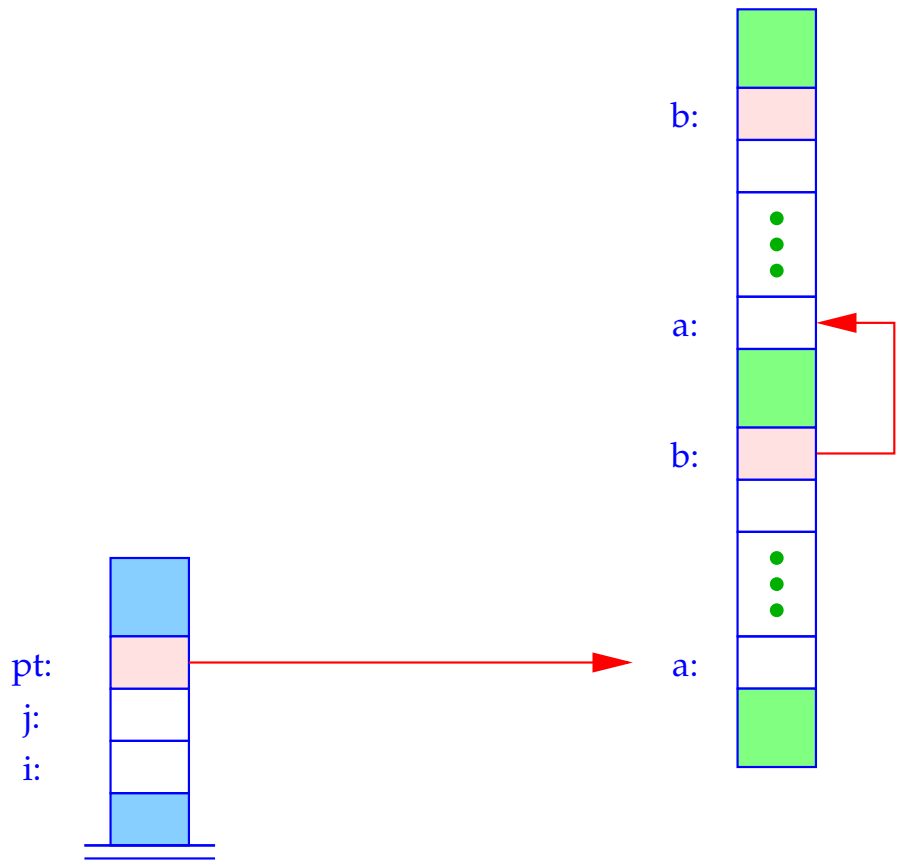
Given the definition

**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 $e \rightarrow a \equiv (*e).a$ holds:

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


b:
a:
b:
a:
pt:
j:
i:

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

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

$=$ $\text{code}_R\ ((pt \to b) \to a)\ \rho$

$\text{code}_R\ (i+1)\ \rho$

loadc 1

mul

add

$=$ $\text{code}_R\ ((pt \to b) \to a)\ \rho$

loada 1

loadc 1

add

loadc 1

mul

add

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

$$\begin{array}{lcll} \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho & = & \text{code}_\text{R}\ (pt \to b)\ \rho & = \quad \text{loada 3} \\ & & \text{loadc 0} & \quad\ \text{loadc 7} \\ & & \text{add} & \quad\ \text{add} \\ & & & \quad\ \text{load} \\ & & & \quad\ \text{loadc 0} \\ & & & \quad\ \text{add} \end{array}$$

In total, we obtain the instruction sequence:

| | | | |
|---|---|---|---|
| loada 3 | load | loada 1 | loadc 1 |
| loadc 7 | loadc 0 | loadc 1 | mul |
| add | add | add | add |

# 6 Conclusion

We tabulate the cases of the translation of expressions:

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (e_1[e_2])\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{loadc}\ |t| & \\
 & & \text{mul} & \\
 & & \text{add} & \text{if } e_1 \text{ has type } t \text{ or } t[] \\
 & & & \\
\text{code}_\text{L}\ (e.a)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{loadc}\ (\rho\, a) & \\
 & & \text{add} &
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) & \\
\text{code}_\text{R}\ (\&e)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \text{if } e \text{ is an array} \\
\text{code}_\text{R}\ (e_1 \,\square\, e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{op} & \text{op instruction for operator `}\square\text{'}
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_R\ q\ \rho & = & \text{loadc q} & q \text{ constant} \\
\\
\text{code}_R\ (e_1 = e_2)\ \rho & = & \text{code}_R\ e_2\ \rho & \\
 & & \text{code}_L\ e_1\ \rho & \\
 & & \text{store} & \\
\\
\text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & \\
 & & \text{load} & \text{otherwise}
\end{array}
$$

Example: **int** $a[10],\ *b;$ with $\rho = \{a \mapsto 7, b \mapsto 17\}$.

Consider the statement: $s_1 \equiv *a = 5;$

We then have:

$$\text{code}_L\ (*a)\ \rho = \text{code}_R\ a\ \rho = \text{code}_L\ a\ \rho = \text{loadc } 7$$

$$\text{code}\ s_1\ \rho = \text{loadc } 5$$
$$\text{loadc } 7$$
$$\text{store}$$
$$\text{pop}$$

As an excercise translate:

$$s_2 \equiv b = (\&a) + 2; \quad \text{and} \quad s_3 \equiv *(b + 3) = 5;$$

$$s_2 \equiv b = (\&a) + 2; \qquad \text{and} \qquad s_3 \equiv *(b+3) = 5;$$

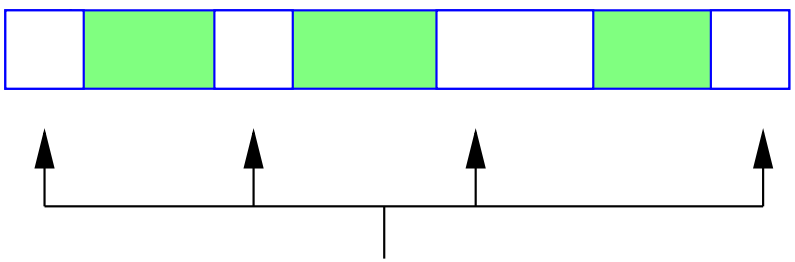
$\text{code}\ (s_2\ s_3)\ \rho$ =

```
loadc 7                 loadc 5
loadc 2                 loadc 17
loadc 1     // scaling  load
mul                     loadc 3
add                     loadc 1     // scaling
loadc 17                mul
store                   add
pop         // end of s2 store
                        pop         // end of s3
```

# 7 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) $\Longrightarrow$ **malloc** or **free** may become expensive.
- Do nothing, i.e.:

$$\text{code } \textbf{free}\,(e);\ \rho \quad = \quad \begin{array}{l}\text{code}_{\text{R}}\ e\ \rho \\ \text{pop}\end{array}$$

  $\Longrightarrow$ simple and (in general) efficient.
- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.

# 8 Functions

The definition of a function consists of

- a name, by which it can be called,
- a specification of the formal parameters;
- maybe a result type;
- a statement part, the body.

For C holds:

$$\text{code}_R\ f\ \rho \quad = \quad \_f \quad = \quad \text{starting address of the code for } f$$

$\Longrightarrow$ Function names must also be managed in the address environment!

## Example:

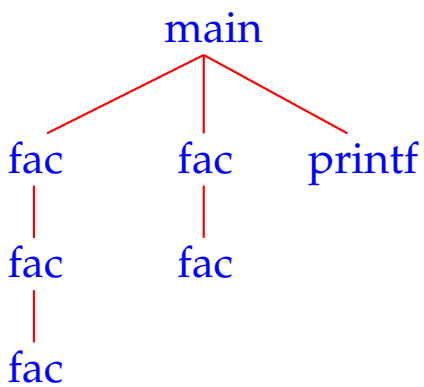
```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x * fac(x − 1);
}
```

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

At any time during the execution, several instances of one function may exist, i.e., may have started, but not finished execution.

An instance is created by a call to the function.

The recursion tree in the example:



## We conclude:

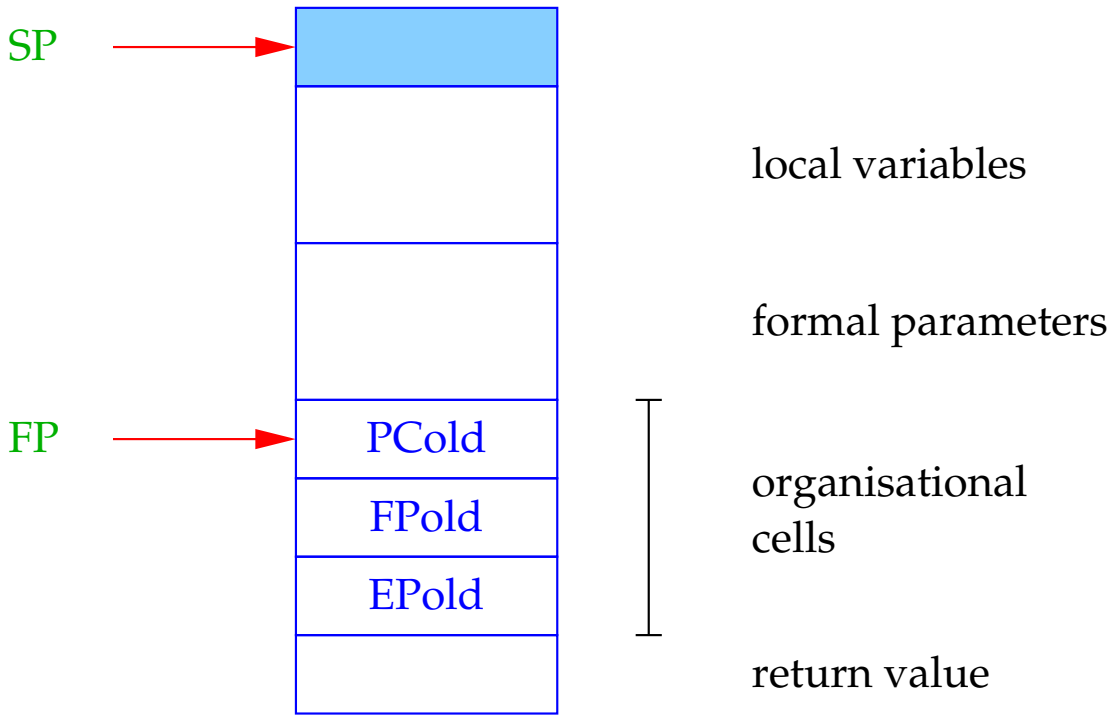
The formal parameters and local variables of the different instances of the same function must be kept separate.

## Idea:

Allocate a special storage area for each instance of a function.

In sequential programming languages these storage areas can be managed on a stack. They are therefore called Stack Frames.

## 8.1 Storage Organisation for Functions




FP $\widehat{=}$ Frame Pointer; points to the last organizational cell and is used to address the formal parameters and the local variables.

The caller must be able to continue execution in its frame after the return from a function. Therefore, at a function call the following values have to be saved into organizational cells:

- the FP
- the continuation address after the call and
- the actual EP.

Simplification: The return value fits into one storage cell.

Translation tasks for functions:

- Generate code for the body!
- Generate code for calls!

## 8.2 Computing the Address Environment

We have to distinguish two different kinds of variables:

1. globals, which are defined externally to the functions;
2. locals/automatic (including formal parameters), which are defined internally to the functions.

$$\Longrightarrow$$

The address environment $\rho$ associates pairs $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ with their names.

Note:

- There exist more refined notions of visibility of (the defining occurrences of) variables, namely nested blocks.
- The translation of different program parts in general uses different address environments!

# Example (1):

```
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\t%d\n", ith (l,i));
    }
```

| address | environment | | at 0 |
|---|---|---|---|
| $\rho_0$ : | $i$ | $\mapsto$ | $(G, 1)$ |
| | $l$ | $\mapsto$ | $(G, 2)$ |
| | ith | $\mapsto$ | $(G, \_\text{ith})$ |
| | main | $\mapsto$ | $(G, \_\text{main})$ |
| | | $\mapsto$ | ... |

# Example (2):

```
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\t%d\n", ith (l,i));
    }
```

---

1 inside of ith:

$\rho_1$ :

| | | |
|---|---|---|
| $i$ | $\mapsto$ | $(L, 2)$ |
| $x$ | $\mapsto$ | $(L, 1)$ |
| $l$ | $\mapsto$ | $(G, 2)$ |
| ith | $\mapsto$ | $(G, \_\text{ith})$ |
| main | $\mapsto$ | $(G, \_\text{main})$ |
| | $\mapsto$ | $\ldots$ |

# Example (3):

```
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\t%d\n", ith (l,i));
    }
```

---

2 inside of main:

$$
\begin{array}{lrcl}
\rho_2 : & i & \mapsto & (G, 1) \\
 & l & \mapsto & (G, 2) \\
 & k & \mapsto & (L, 1) \\
 & \text{ith} & \mapsto & (G, \text{\_ith}) \\
 & \text{main} & \mapsto & (G, \text{\_main}) \\
 & & \mapsto & \ldots
\end{array}
$$

## 8.3 Calling/Entering and Leaving Functions

Be $f$ the actual function, the Caller, and let $f$ call the function $g$, the Callee.

The code for a function call has to be distributed among the Caller and the Callee:

The distribution depends on who has which information.

Actions upon calling/entering $g$:

1. Saving FP, EP — mark
2. Computing the actual parameters
3. Determining the start address of $g$
4. Setting the new FP — call
5. Saving PC and jump to the beginning of $g$ — call
6. Setting the new EP — enter
7. Allocating the local variables — alloc

(mark, call, enter, alloc: available in $f$)

(enter, alloc: available in $g$)

Actions upon leaving $g$:

1. Restoring the registers FP, EP, SP
2. Returning to the code of f, i.e. restoring the PC

(return)

Altogether we generate for a call:

$$
\begin{aligned}
\text{code}_\text{R}\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \text{mark} \\
& \text{code}_\text{R}\ e_1\ \rho \\
& \ldots \\
& \text{code}_\text{R}\ e_n\ \rho \\
& \text{code}_\text{R}\ g\ \rho \\
& \text{call n}
\end{aligned}
$$

## Note:

- Expressions occurring as actual parameters will be evaluated to their R-value $\Longrightarrow$ Call-by-Value-parameter passing.
- Function $g$ can also be an expression, whose R-value is the start address of the function to be called ...

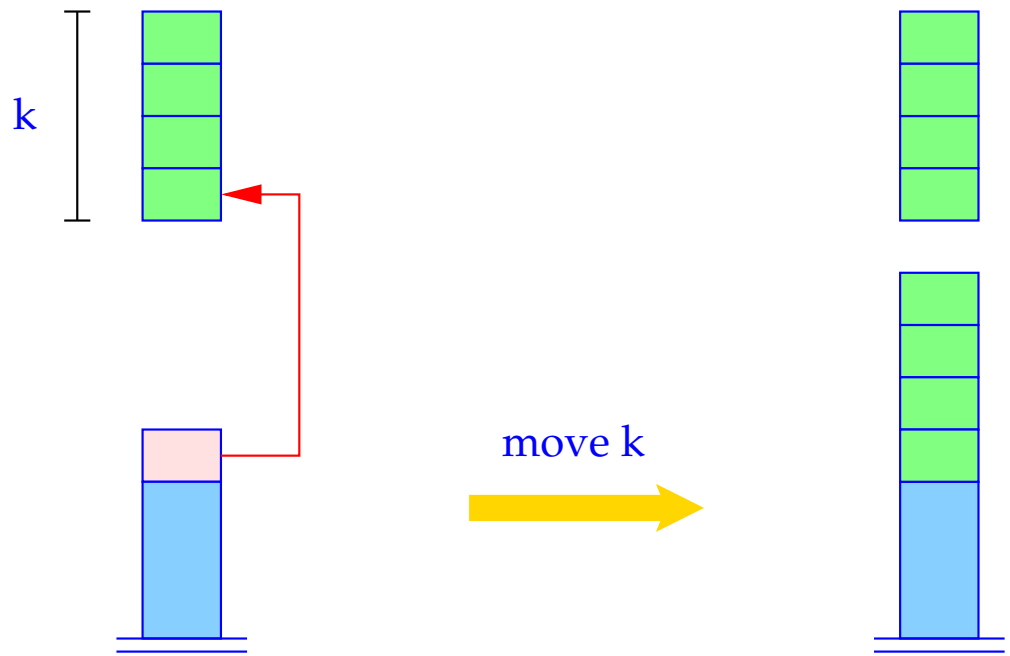
- Function names are regarded as constant pointers to functions, similarly to defined arrays. The R-value of such a pointer is the start address of the function code.
- Note! For a variable **int** $(*)()\ g;$ , the two calls

  $$(*g)() \qquad \text{and} \qquad g()$$

  are equivalent!
  Normalization: Dereferencing of a function pointer are ignored.
- Structures are copied when they are passed as parameters.

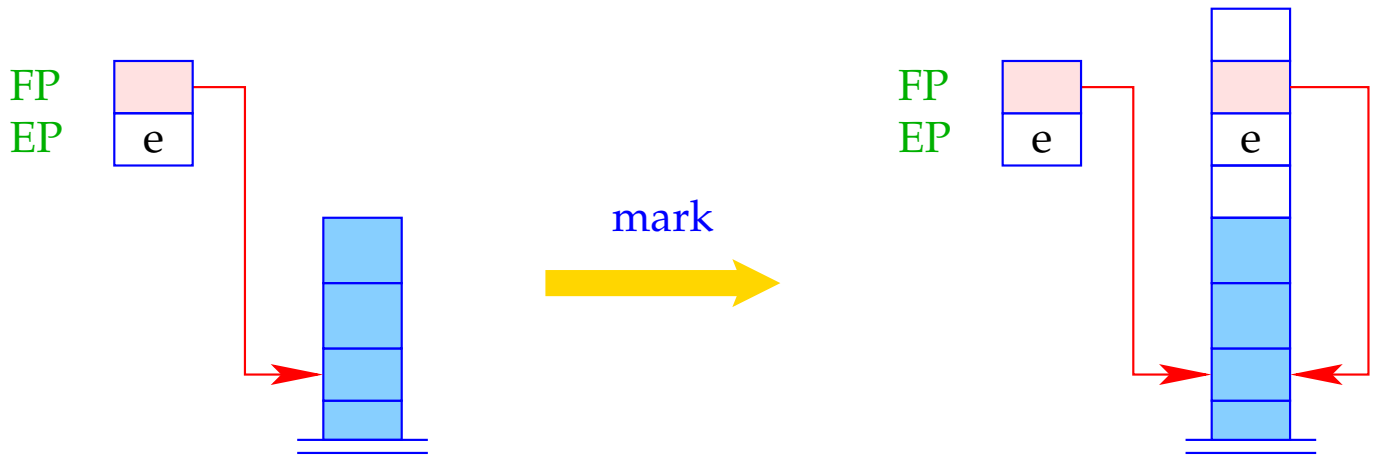
In consequence:

| | | | |
|---|---|---|---|
| $\text{code}_\text{R}\ f\ \rho$ | $=$ | $\rho\ f$ | $f$ a function name |
| $\text{code}_\text{R}\ (*e)\ \rho$ | $=$ | $\text{code}_\text{R}\ e\ \rho$ | $e$ a function pointer |
| $\text{code}_\text{R}\ e\ \rho$ | $=$ | $\text{code}_\text{L}\ e\ \rho$ | |
| | | move k | $e$ a structure of size $k$ |




```
for (i = k-1; i≥0; i--)
        S[SP+i] ← S[S[SP]+i];
SP ← SP+k–1;
```

The instruction mark allocates space for the return value and for the organizational cells and saves the FP and EP.

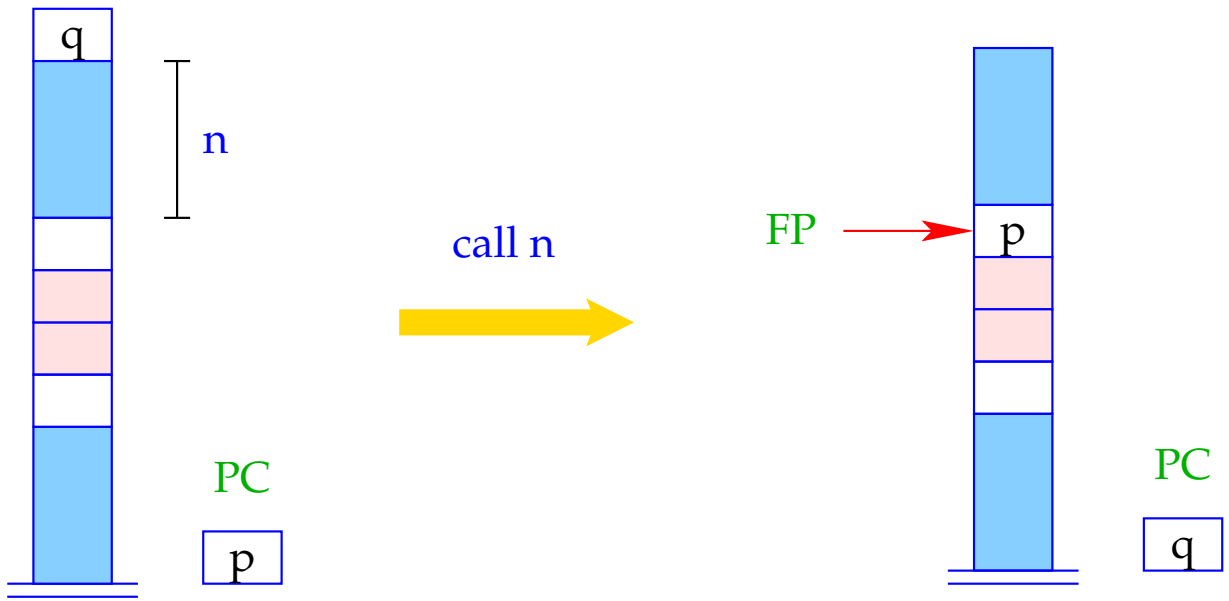



S[SP+2] ← EP;

S[SP+3] ← FP;

SP ← SP + 4;

The instruction call n saves the continuation address and assigns FP, SP, and PC their new values.




FP ← SP - n - 1;

S[FP] ← PC;

PC ← S[SP];

SP ← SP – 1;

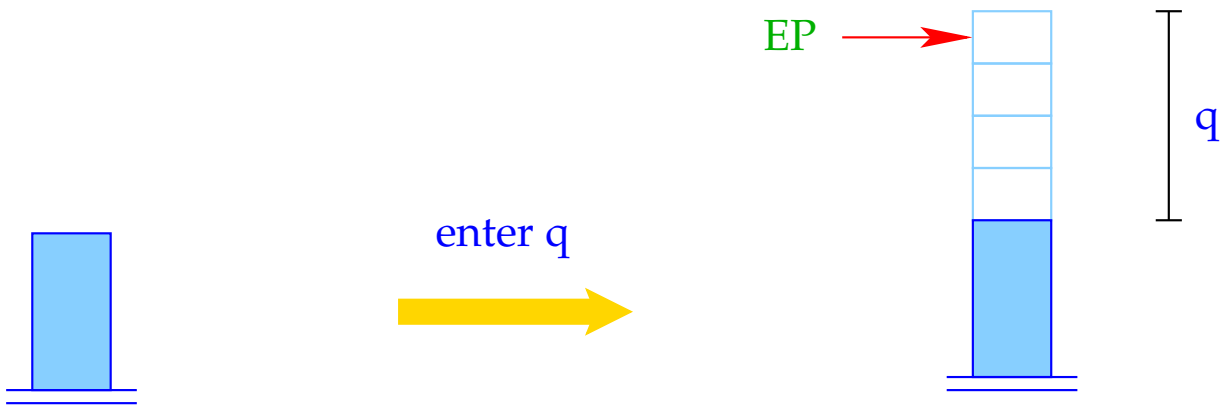
Correspondingly, we translate a function definition:

$$\begin{array}{lll} \text{code} & t\ \mathrm{f}\ (specs)\{V\_defs \quad ss\} & \rho \quad = \\ & \quad \mathrm{\_f:} \quad \text{enter q} \quad // \quad \textit{Setting the EP} & \\ & \qquad\quad \text{alloc k} \quad // \quad \textit{Allocating the local variables} & \\ & \qquad\quad \text{code } ss\ \rho_{\mathrm{f}} & \\ & \qquad\quad \text{return} \quad // \quad \textit{leaving the function} & \end{array}$$

where

| | | | |
|---|---|---|---|
| $t$ | = | return type of $f$ with $\lvert t\rvert \leq 1$ | |
| q | = | $maxS + \mathrm{k}$ | wobei |
| $maxS$ | = | maximal depth of the local stack | |
| k | = | space for the local variables | |
| $\rho_{\mathrm{f}}$ | = | address environment for $f$ | |
| | | // *takes care of specs, V_defs* and $\rho$ | |

The instruction enter q sets EP to its new value. Program execution is terminated if not enough space is available.

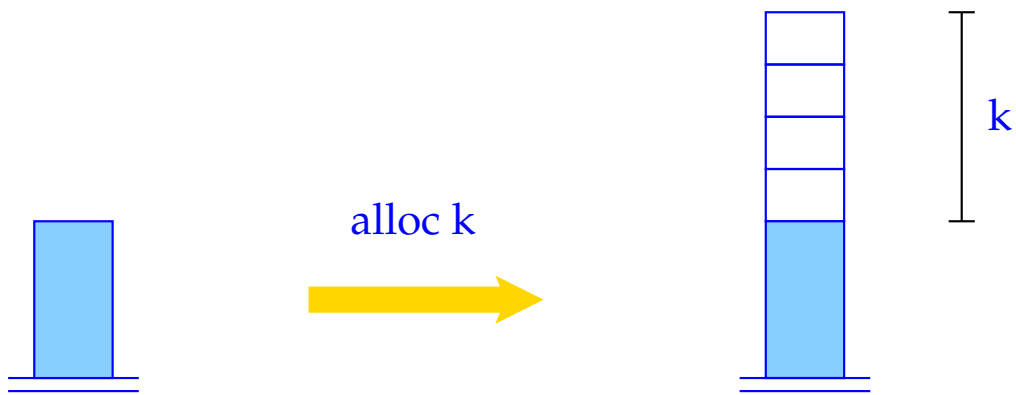



EP $\leftarrow$ SP + q;

if (EP $\geq$ NP)

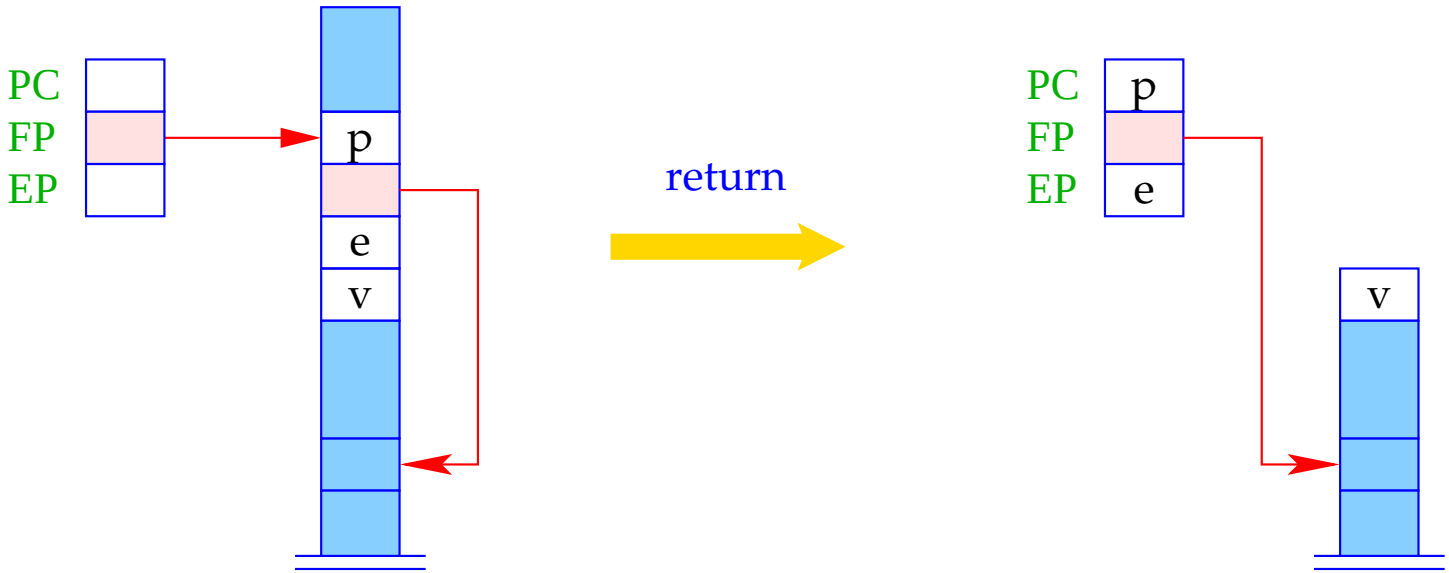
Error ("Stack Overflow");

The instruction alloc k reserves stak space for the local variables.




SP ← SP + q;

The instruction return pops the actual stack frame, i.e., it restores the registers PC, EP, SP, and FP and leaves the return value on top of the stack.




PC $\leftarrow$ S[FP]; EP $\leftarrow$ S[FP-2];

if (EP $\geq$ NP) Error ("Stack Overflow");

SP $\leftarrow$ FP-3; FP $\leftarrow$ S[SP+2];

## 8.4 Access to Variables and Formal Parameters, and Return of Values

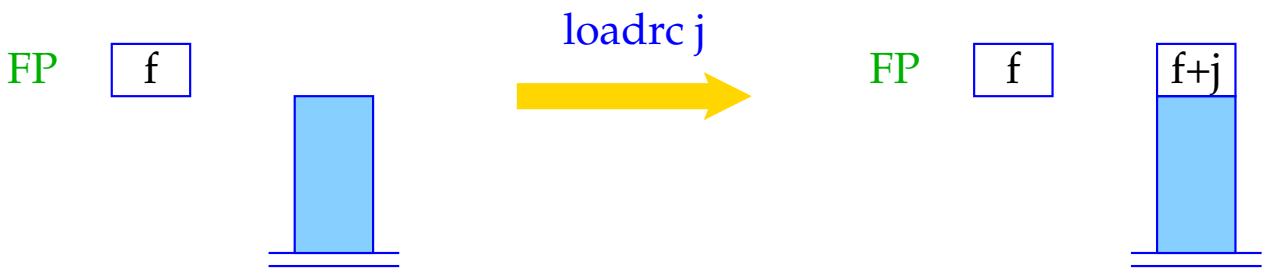
The addressing of local variables and formal parameters is relative to the actual FP.

We therefore modify $\text{code}_\text{L}$ for the case of variable names.

For $\rho\ x = (tag, j)$ we define

$$\text{code}_\text{L}\ x\ \rho = \begin{cases} \text{loadc j} & tag = G \\ \text{loadrc j} & tag = L \end{cases}$$

The instruction loadrc j computes the sum of FP and j.




SP ← SP + 1;

S[SP] ← FP+j;

As an optimization one introduces the instructions loadr j and storer j .
This is analogous to loada j and storea j.

$$\begin{array}{lcl} \text{loadr j} & = & \text{loadrc j} \\ & & \text{load} \end{array}$$

$$\begin{array}{lcl} \text{storer j} & = & \text{loadrc j} \\ & & \text{store} \end{array}$$

The code for **return** *e;* corresponds to an assigment to a variable with relative address $-3$.

$$\begin{array}{lcl} \text{code } \textbf{return}\, e;\ \rho & = & \text{code}_{\text{R}}\ e\ \rho \\ & & \text{storer -3} \\ & & \text{return} \end{array}$$

Example: For the function

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

we generate:

```
_fac:  enter q        loadc 1       A:  loadr 1          mul
       alloc 0        storer -3         mark           storer -3
       loadr 1        return            loadr 1        return
       loadc 0        jump B            loadc 1   B:   return
       leq                              sub
       jumpz A                          loadc _fac
                                        call 1
```

where $\rho_{\text{fac}} : x \mapsto (L, 1)$ and $q = 1 + 4 + 2 = 7$.

# 9 Translation of Whole Programs

The state before program execution starts:

$$\text{SP} \leftarrow -1 \qquad \text{FP} \leftarrow \text{EP} \leftarrow 0 \qquad \text{PC} \leftarrow 0 \qquad \text{NP} \leftarrow \text{MAX}$$

Be $p \equiv V\_defs \quad F\_def_1 \ \ldots \ F\_def_n$, a program, where $F\_def_i$ defines a function $f_i$, of which one is named main.

The code for the program $p$ consists of:

- Code for the function definitions $F\_def_i$;
- Code for allocating the global variables;
- Code for the call of main();
- the instruction halt.

We thus define for $p \equiv V\_defs \quad F\_def_1 \ \ldots \ F\_def_n$:

| | | | |
|---|---|---|---|
| code $p\ \emptyset$ | = | enter (k + 5) | set EP |
| | | alloc k | allocate global variables |
| | | mark | create stack frame |
| | | loadc _main | |
| | | call 0 | call main |
| | | halt | |
| | $\_f_1$: | code $F\_def_1\ \rho$ | |
| | | ⋮ | |
| | $\_f_n$: | code $F\_def_n\ \rho$ | |

where $\emptyset \quad \widehat{=}$ empty address environment;

$\rho \quad \widehat{=}$ global address environment;

k $\widehat{=}$ space for global variables