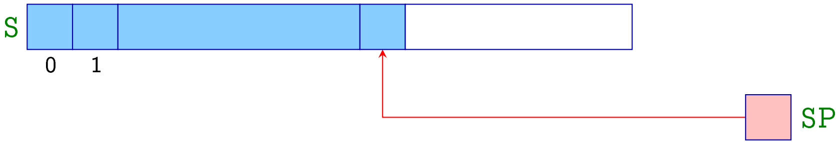
CMa — simple C Abstract Machine

CMa architecture

- An abstract machine has set of *instructions* which can be executed in an abstract hardware.
- The abstract hardware may be seen as a collection of certain data structures used by instructions
- . . . and controlled by the *run-time system*.

CMa architecture

Stack:

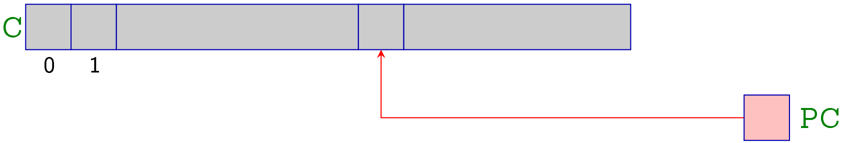


S	=	Stack — memory area for data where insertion and deletion of items uses LIFO principle.
SP	=	Stack-Pointer — register containing an address of the topmost item.

Simplification: all non-structural values are of the same size and fit into a single cell of the stack.

CMa architecture

Code:



C = Code-store — memory area for a program code; each cell contains a single AM instruction.

PC = Program Counter — register containing an address of the instruction to be executed *next*.

Initially, PC contains the address 0; ie. C[0] contains the first instruction of the program.

CMa architecture

Execution of the program:

- Machine loads an instruction at C[PC] to the register IR (Instruction-Register), then increments the program counter PC, and finally executes the instruction:

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

- Execution of an istruction (eg. jump) may change the contents of the program counter PC.
- The main loop of the machine is stopped by the instruction `halt`, which returns the control back to the environment.
- We will introduce the rest of the instructions step by step as necessary.

Simple expressions and assignement

Problem: evaluate an expression like $(6 + 2) * 4 - 1$;
i.e. generate a sequence of instructions which

- finds the value of the expression, and
- pushes it to top of the stack.

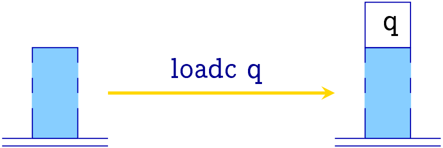
Idea:

- first evaluate subexpressions,
- save these values to top of the stack, and
- execute an instruction corresponding to the operator.

Simple expressions and assignement

General principles:

- instructions assume arguments to be in topmost cells of the stack,
- an execution of the instruction consumes its arguments,
- the result is saved in top of the stack.

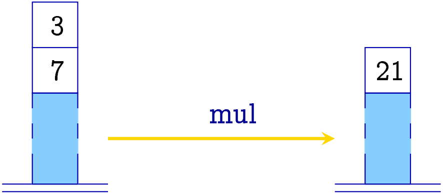


```
SP++;
S[SP] = q;
```

Instruction `loadc q` doesn't have arguments and pushes the constant `q` to top of the stack.

NB! In pictures, the contents of SP is represented implicitly by the height of the stack.

Simple expressions and assignement



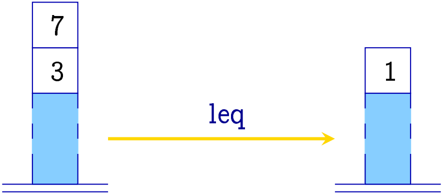
```
SP--;
S[SP] = S[SP] * S[SP+1];
```

The instruction `mul` assumes two arguments in the stack, consumes them, and pushes their product to top of the stack

Instructions corresponding to other arithmetic and logic operators `add`, `sub`, `div`, `mod`, `and`, `or`, `xor`, `eq`, `neq`, `le`, `leq`, `ge` and `geq` work analoguously.

Simple expressions and assignement

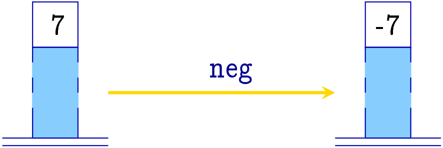
Example: operator `leq`



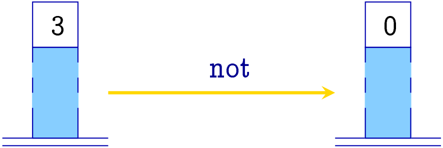
NB! The integer 0 represents the boolean "false"; all other integers represent "true".

Simple expressions and assignement

Unary operators `neg` and `not` consume one argument and produce a single result value:



```
S[SP] = -S[SP];
```



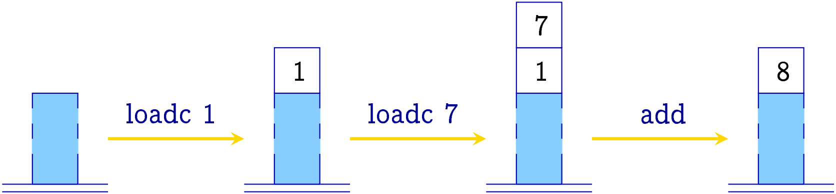
```
if (S[SP] ≠ 0)
    S[SP] = 0;
else
    S[SP] = 1;
```

Simple expressions and assignement

Example: code for the expression $1 + 7$:

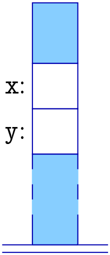
```
loadc 1
loadc 7
add
```

Execution of the code results:



Simple expressions and assignement

- Variables correspond to cells of the stack S:



- Code generation is specified in terms of functions code, code_L and code_R.
- Parameters: a *syntactic construction* to be compiled and an *address environment* (ie. a function mapping variables to their relative addresses in the stack).

Simple expressions and assignement

- Variables are used in two different ways.
- For instance, in the assignement $x = y + 1$ we are interested of the *value* of the variable y, but of the *address* of the variable x.
- The syntactic placement of the variable determines wether we need its *L-value* or *R-value*.

$$\begin{array}{lcl} \text{L-value of a variable} & = & \text{its address} \\ \text{R-value of a variable} & = & \text{its "real" value} \end{array}$$

- Function $\mathrm{code}_L\ e\ \rho$ emits a code computing a L-value of the expression e in the environment ρ.
- Function $\mathrm{code}_R\ e\ \rho$ does the same for the R-value.
- NB! Not every expression has a L-value (eg.: $x + 1$).

Simple expressions and assignement

- Compilation of binary operators:

$$\begin{array}{lcl} \mathtt{code}_R\ (e_1 + e_2)\ \rho & = & \mathtt{code}_R\ e_1\ \rho \\ & & \mathtt{code}_R\ e_2\ \rho \\ & & \mathtt{add} \end{array}$$

  - Similarly for other binary operators.
- Compilation of unary operators:

$$\begin{array}{lcl} \mathtt{code}_R\ (-e)\ \rho & = & \mathtt{code}_R\ e\ \rho \\ & & \mathtt{neg} \end{array}$$

  - Similarly for other unary operators.
- Compilation of primitive constant values:

$$\mathtt{code}_R\ q\ \rho \quad = \quad \mathtt{loadc\ q}$$

Simple expressions and assignement

- Compilation of variables:

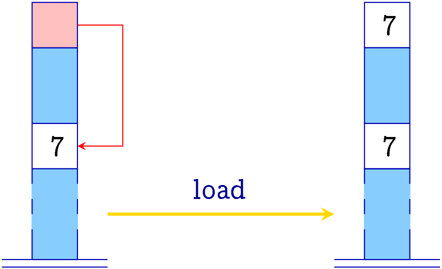
$$\begin{array}{lcl} \mathtt{code}_L\ x\ \rho & = & \mathtt{loadc}\ (\rho\ x) \\ \mathtt{code}_R\ x\ \rho & = & \mathtt{code}_L\ x\ \rho \\ & & \mathtt{load} \end{array}$$

- Compilation of assignement expressions:

$$\begin{array}{lcl} \mathtt{code}_R\ (x = e)\ \rho & = & \mathtt{code}_R\ e\ \rho \\ & & \mathtt{code}_L\ x\ \rho \\ & & \mathtt{store} \end{array}$$

Simple expressions and assignement

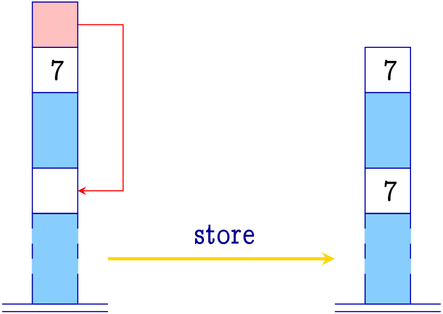
Instruction `load` copies the contents of the stack cell pointed by the argument to top of the stack:



```
S[SP] = S[S[SP]];
```

Simple expressions and assignement

Instruction `store` saves the contents of the second cell to the stack cell pointed by the topmost cell, but leaves the second cell to top of the stack:



```
S[S[SP]] = S[SP-1];
SP--;
```

NB! Differs from the analogous P-machine instruction in the Wilhelm/Maurer book.

Simple expressions and assignement

Example: let $e \equiv (x = y - 1)$ and $\rho = \{x \mapsto 4, y \mapsto 7\}$,

then $\texttt{code}_R$ e ρ emits the code:

```
loadc 7      sub
load         loadc 4
loadc 1      store
```

Optimization: introduce special instructions for frequently occuring combinations of instructions, e.g.:

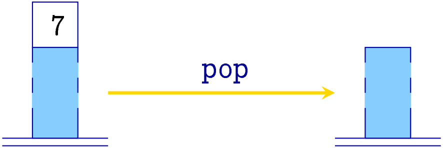
```
loada q   =   loadc q
              load
storea q  =   loadc q
              store
```

Statements and their sequences

- If e is an expression, then $e;$ is a statement.
- A statement doesn't have any arguments, nor have a value.
- Hence, the contents of the register SP must remain unchanged after the execution of the code corresponding to the statement.

$$\begin{array}{lcl}
\text{code}\ (e;)\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{pop} \\
\text{code}\ (s\ \ ss)\ \rho & = & \text{code}\ s\ \rho \\
 & & \text{code}\ ss\ \rho \\
\text{code}\ \varepsilon\ \rho & = & \qquad\qquad // \text{ empty sequence}
\end{array}$$

- Instruction pop removes the topmost stack cell:



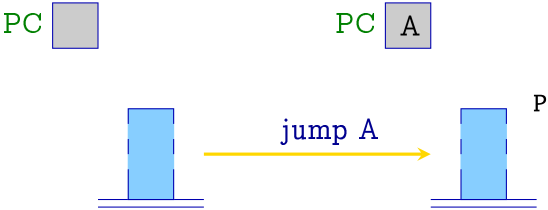
`SP--;`

Conditional statements and loops

- For simplicity, we use symbolic labels as targets of jumps, which later are replaced by absolute addresses.
- Instead of absolute addresses we could use relative addresses; i.e. relative w.r.t. the actual value of PC.
- Advantages of the last approach are:
  - in general, relative addresses are *smaller*;
  - the code is *relocatable*.

Conditional statements and loops

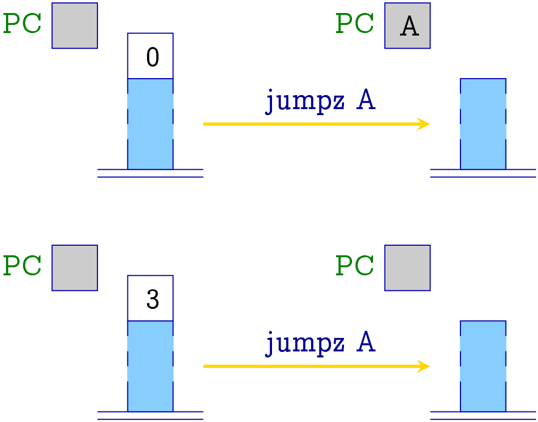
Instruction `jump A` performs an unconditional jump to the address A; the stack doesn't change:



```
PC = A;
```

Conditional statements and loops

Instruction `jumpz A` performs a conditional jump; it jumps to the address `A` only if the topmost stack cell contains 0:



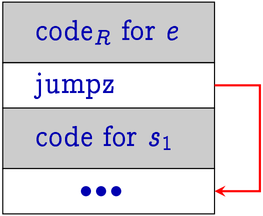
```
if (S[SP] = 0)
    PC = A;
SP--;
```

Conditional statements and loops

Compilation of if-statements $s \equiv \textbf{if}\ (e)\ s_1$:

- generate a code for the condition e and statement s_1;
- insert the conditional jump instruction in between.

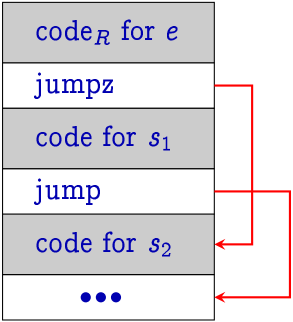
$$
\begin{array}{rl}
\mathsf{code}\ (\textbf{if}\ (e)\ s_1)\ \rho & = \\
 & \mathsf{code}_R\ e\ \rho \\
 & \mathtt{jumpz}\ \mathtt{A} \\
 & \mathsf{code}\ s_1\ \rho \\
\mathtt{A}: & \ldots
\end{array}
$$



Conditional statements and loops

- Compilation of if-else-statements $s \equiv \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2$:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathtt{code}\ (\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2)\ \rho \quad =} \\
 & \mathtt{code}_R\ e\ \rho \\
 & \mathtt{jumpz}\ \ \mathtt{A} \\
 & \mathtt{code}\ s_1\ \rho \\
 & \mathtt{jump}\ \ \mathtt{B} \\
\mathsf{A}: & \mathtt{code}\ s_2\ \rho \\
\mathsf{B}: & \ldots
\end{array}
$$



Conditional statements and loops

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

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

then code s ρ emits a code:

```
loada 4        loada 4         A: loada 7
loada 7        loada 7            loada 4
ge             sub                sub
jumpz A        storea 4           storea 7
               pop                pop
               jump B          B: ...
```

(i) (ii) (iii)

Conditional statements and loops

- Compilation of while-loops $s \equiv \textbf{while}\ (e)\ s_1$:

$\texttt{code}\ (\textbf{while}\ (e)\ s_1)\ \rho \quad =$

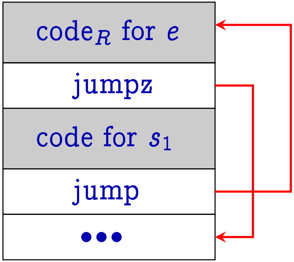
$\texttt{A:}\ \texttt{code}_R\ e\ \rho$

$\quad \texttt{jumpz B}$

$\quad \texttt{code}\ s_1\ \rho$

$\quad \texttt{jump A}$

$\texttt{B:}\ \ldots$



$\texttt{code}_R$ for e

jumpz

code for s_1

jump

•••

Conditional statements and loops

Example: let $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ and

$$\begin{array}{llll} s & \equiv & \textbf{while}\ (a > 0)\ \{ & (i) \\ & & \qquad c = c + 1; & (ii) \\ & & \qquad a = a - b; & (iii) \\ & & \} & \end{array}$$

then code $s\ \rho$ emits a code:

```
A: loada 7
   loadc 0
   ge
   jumpz B
```

(i)

```
loada 9
loadc 1
add
storea 9
pop
```

(ii)

```
loada 7
loada 8
sub
storea 7
pop
```

(iii)

```
   jump A
B: ...
```

Conditional statements and loops

- A for-loop $s \equiv \textbf{for}\ (e_1;\ e_2;\ e_3)\ s_1$ is equivalent with the while-loop $e_1;\ \textbf{while}\ (e_2)\ \{s_1\ e_3;\}$ (assuming, that s_1 doesn't contain any continue-statements)

$$
\begin{array}{rcrl}
\mathsf{code}\ (\textbf{for}\ (e_1;\ e_2;\ e_3)\ s_1)\ \rho & = & & \mathsf{code}_R\ e_1\ \rho \\
 & & & \texttt{pop} \\
 & & \mathsf{A}: & \mathsf{code}_R\ e_2\ \rho \\
 & & & \texttt{jumpz B} \\
 & & & \mathsf{code}\ s_1\ \rho \\
 & & & \mathsf{code}_R\ e_3\ \rho \\
 & & & \texttt{pop} \\
 & & & \texttt{jump A} \\
 & & \mathsf{B}: & \ldots
\end{array}
$$

Conditional statements and loops

- In general, switch-statements should be translated into nested if-statements:

$$
\begin{array}{lll}
\textbf{switch } (e)\ \{ & & x = e; \\
\quad \textbf{case } c_0: \quad ss_0 \ \textbf{break}; & & \textbf{if } (x == c_0)\ ss_0 \\
\quad \textbf{case } c_1: \quad ss_1 \ \textbf{break}; & & \textbf{else if } (x == c_1)\ ss_1 \\
\qquad \ldots & \Longrightarrow & \qquad \ldots \\
\quad \textbf{case } c_{k-1}: ss_{k-1} \ \textbf{break}; & & \textbf{else if } (x == c_{k-1})\ ss_{k-1} \\
\quad \textbf{default}: \quad ss_k & & \textbf{else } ss_k \\
\} & &
\end{array}
$$

- By sorting the labels and using binary seach, it's possible to decrease the number of comparisions to the logarithm of the number of labels.

Conditional statements and loops

- In specific cases it's possible to have a constant time branching.
- Consider a switch-statement in the form:

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

Conditional statements and loops

$$
\begin{array}{lclrlrl}
\mathsf{code}\ s\ \rho & = & \mathsf{code}_R\ e\ \rho & \mathsf{C}_0: & \mathsf{code}\ ss_0\ \rho & \mathsf{B}: & \mathtt{jump}\ \mathtt{C}_0 \\
 & & \mathsf{check}\ 0\ k\ \mathsf{B} & & \mathtt{jump}\ \mathtt{D} & & \ldots \\
 & & & & \ldots & & \mathtt{jump}\ \mathtt{C}_k \\
 & & & \mathsf{C}_k: & \mathsf{code}\ ss_k\ \rho & \mathsf{D}: & \ldots \\
 & & & & \mathtt{jump}\ \mathtt{D} & &
\end{array}
$$

- Macro check $0\ k$ B tests whether the R-value of the condition is in between $[0, k]$, and then performs an indexed jump.
- An i-th element of the "jump tabel" B contains a unconditional jump instruction to the beginning of the code corresponding to the i-th branch.
- Each branch ends with the unconditional jump.

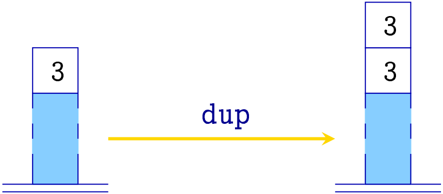
Conditional statements and loops

```
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 the condition is used both for comparision and indexing, hence it must be duplicated before comparisions.
- If R-value is not in between $[0, k]$, it will be replaced by the constant k before the jump.

Conditional statements and loops

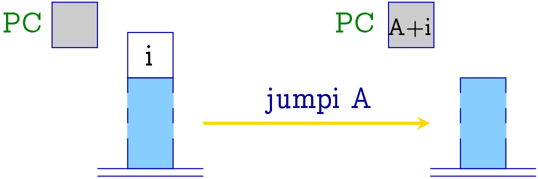
Instruction dup duplicates the topmost cell of the stack:



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

Conditional statements and loops

Instruction `jumpi A` performs an indexed jump:



```
PC = A + S[SP];
SP--;
```

Conditional statements and loops

- Jump table B may be placed just after the macro check; it allows to save some unconditional jumps.
- If the range of values starts with u (and is not 0), then u must be subtracted from the R-value of e before indexing.
- If all potential values of e are in range $[0, k]$, then the macro check is not needed.

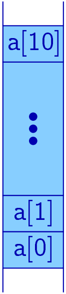
Arrays, records and static memory management

- Goal: *statically* (i.e. compile-time) to bind with each variable x a fixed (relative) address $\rho\ x$.
- We assume that variables of primitive types (e.g. **int**, ...) fit into a single memory cell.
- Bind variables to addresses starting from 1 using their declaration order.
- Hence, in the case of declarations $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$ (where t_i is primitive type) we get an address environment ρ s.t.

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

Arrays, records and static memory management

- Array is a sequence of memory cells.
- Uses integer indices for an access of its individual elements.
- Example: declaration $\mathbf{int}[11]a$; defines an array with 11 elements.



Arrays, records and static memory management

- Define a function sizeof (notation $|\cdot|$) which finds the required memory amount to represent a value of a given type:

$$|t| \quad = \quad \begin{cases} 1 & \text{if } t \text{ is a primitive type} \\ k \cdot |t'| & \text{if } t \equiv t'[k] \end{cases}$$

- Hence, in the case of declarations $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{array}{lcl} \rho\ x_1 & = & 1 \\ \rho\ x_i & = & \rho\ x_{i-1} + |t_{i-1}| \quad i > 1 \end{array}$$

- Since $|\cdot|$ can be computed compile-time, it is also possible to compute the address environment ρ in compile-time.

Arrays, records and static memory management

- Let $t\ a[c]$; be an array declaration.
- Then, the address of its i-th element is $\rho\ a + |t| \times (\text{rval of } i)$

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

- In general, an array can be given by an expression which must be evaluated before indexing.
- In C, an array is a *pointer-constant* which R-value is the start address of the array.

Arrays, records and static memory management

$$\begin{array}{lcll}
\mathrm{code}_L\ (e_1[e_2])\ \rho & = & \mathrm{code}_R\ e_1\ \rho & \\
 & & \mathrm{code}_R\ e_2\ \rho & \\
 & & \texttt{loadc}\ |t| & \\
 & & \texttt{mul} & \\
 & & \texttt{add} & \\
\mathrm{code}_R\ e\ \rho & = & \mathrm{code}_L\ e\ \rho & e \text{ is an array}
\end{array}$$

- NB! In C, the following are equivalent (as L-values):

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

- Normalization: array variables and expressions which evaluate to an array are before indexing brackets; index expressions are inside brackets.

Arrays, records and static memory management

- Record is set of fields; each field may be of different type.
- Fields are accessed by names (selectors).
- For simplicity, we assume that field names are unique.
  - Alternative: for each record type st have a separate environment ρ_{st}.
- Let $\textbf{struct}\ \{\ \textbf{int}\ a;\ \ \textbf{int}\ b;\ \}\ x;$ be a declaration:
  - the address of the record x is the address of its first cell;
  - field addresses are relative to the address of the record; i.e. in the example above $a \mapsto 0, b \mapsto 1$.

Arrays, records and static memory management

- Let $t \equiv \textbf{struct}\ \{\ t_1\ c_1;\ \ldots\ t_k\ c_k;\ \ \}$, then

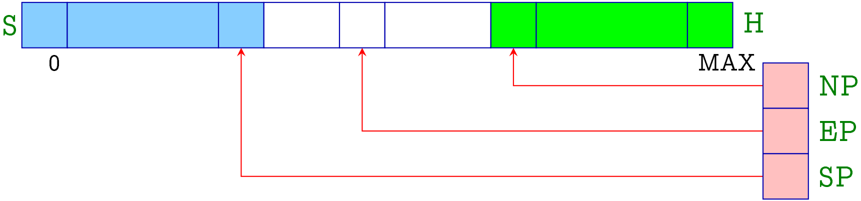
$$\begin{array}{rcll} |t| & = & \sum_{i=1}^{k} |t_i| & \\ \rho\ c_1 & = & 0 & \\ \rho\ c_i & = & \rho\ c_{i-1} + |t_{i-1}| & \qquad i > 1 \end{array}$$

- Thus, an address of the field $x.c_i$ is $\rho\ x + \rho\ c_i$

$$\begin{array}{rcl} \mathsf{code}_L\ (e.c)\ \rho & = & \mathsf{code}_L\ e\ \rho \\ & & \mathtt{loadc}\ (\rho\ c) \\ & & \mathtt{add} \end{array}$$

References and dynamic memory management

Heap:



H = Heap — memory area for dynamically allocated data.

NP = New-Pointer — register containig the address of the lowermost used cell in the heap.

EP = Extreme-Pointer — register containig the address of the topmost cell to where SP may point during execution of the given function.

References and dynamic memory management

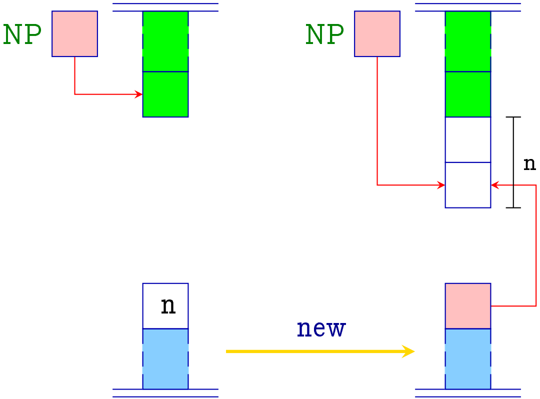
- Stack and heap grow towards each other and must not overlap (stack overflow).
- Both, incrementing SP or decrementing NP, may result to the overflow.
- Register EP helps to avoid an overflow in the case of stack operations.
- The value of EP can be determined statically.
- But when allocating memory from the heap, one must check for the overflow.

References and dynamic memory management

- *Pointers* allow access to ananymous, dynamically created, objects which life-time doesn't follow LIFO principle.
- Pointer values are returned by the following operations:
  - a call to the function $\mathbf{malloc}(e)$ allocates a memory area of size e and returns a beginning address of the area.
  - an application of the address operator & to a variable returns an address of the variable (ie. its L-value).

$$\begin{array}{lcl} \mathrm{code}_R\ (\mathbf{malloc}(e))\ \rho & = & \mathrm{code}_R\ e\ \rho \\ & & \mathtt{new} \\ \mathrm{code}_R\ (\&e)\ \rho & = & \mathrm{code}_L\ e\ \rho \end{array}$$

References and dynamic memory management



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

- NULL is a special reference constant; equivalent to the integer 0.
- In the case of overflow returns NULL-pointer.

References and dynamic memory management

- Referenced values can be accessed by the following ways:
  - an application of the dereferencing operator $*$ to expression e returns the content of a memory cell which address is a R-value of e;
  - a record field selection through a pointer $e{\rightarrow}c$ is equivalent to the expression $(*e).c$.

$$\begin{array}{lcl} \mathtt{code}_L\ (*e)\ \rho & = & \mathtt{code}_R\ e\ \rho \\ \mathtt{code}_L\ (e{\rightarrow}c)\ \rho & = & \mathtt{code}_R\ e\ \rho \\ & & \mathtt{loadc}\ (\rho\ c) \\ & & \mathtt{add} \end{array}$$

References and dynamic memory management

Example: let be given the following declarations:

$$\begin{array}{l} \textbf{struct}\ t\ \{\ \textbf{int}\ a[7];\ \textbf{struct}\ t\ \ *b;\ \};\\ \textbf{int}\ i,\ j;\\ \textbf{struct}\ t\ \ *pt; \end{array}$$

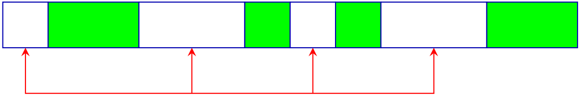
Then $\rho = \{\ a \mapsto 0,\ b \mapsto 7,\ i \mapsto 1,\ j \mapsto 2,\ pt \mapsto 3\ \}$.

For the expression $((pt{\rightarrow}b){\rightarrow}a)[i+1]$ the following code is emitted:

```
loada 3     load        loada 1     loadc 1
loadc 7     loadc 0     loadc 1     mul
add         add         add         add
```

References and dynamic memory management

- Memory is freed by calling the C-function **free**(e).
- The given memory area is marked as a free and is put to the special free list, from where **malloc** can reuse it if necessary.
- Problems:
  - after freeing, there might be still some accessible references pointing to the memory area (dangling references);
  - over the time, memory might get *fragmented*;



free memory fragments

  - keeping track of the free list might be relatively costly.

References and dynamic memory management

- Alternative: in the case of functsion **free** do nothing.

$$\begin{array}{lcl} \mathsf{code}\ (\mathbf{free}(e);)\ \rho & = & \mathsf{code}_R\ e\ \rho \\ & & \mathtt{pop} \end{array}$$

- If memory is full, deallocate the unaccessible memory automatically using garbage collection.
  - \+ Allocation and "deallocation" is simple and very efficient.
  - \+ No "dangling references".
  - \+ Several garbage collection algorithms defragment the used memory.
  - – Garbage collection may take time, hence there might be noticable pauses during the execution of the program.

Functions

- A function definition consists of four parts:
  - a *name* of the function, which is used when function is called;
  - a specification of *formal parameters*;
  - a *return type* of the function;
  - a *body* of the function.
- In C the following holds:

$$\text{code}_R\; f\; \rho \qquad = \qquad \_f \qquad = \qquad \text{starting address of } f \text{ code}$$

- Hence, the address environment must also keep track of function names!

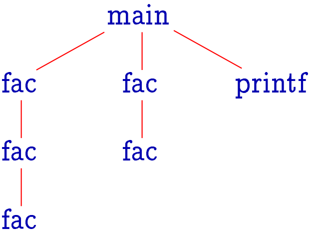
Functions

- 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);
}
```

- The same function may have several simultaneously active instances.



main

fac fac printf

fac fac

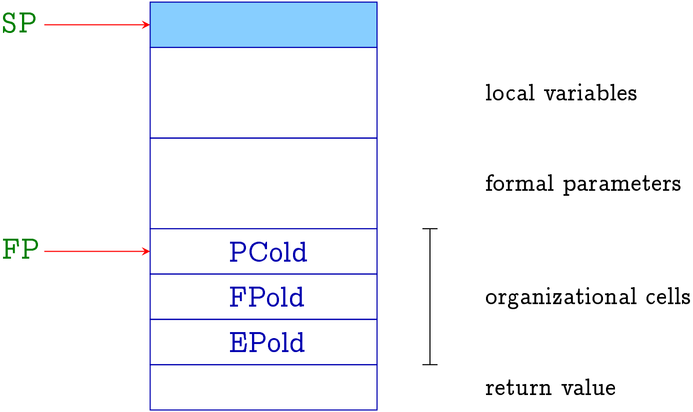
fac

Functions

- Formal parameters and local variables of each instance of the function must be kept separately.
- For this we allocate in stack a special memory region called Stack Frame).
- FP (Frame Pointer) is a register which points to the last *organizational cell* of the active frame, and which is used for addressing of formal parameters and local variables.

Functions

Structure of a frame:



Functions

- After function returns, the caller must be able to continue its execution in its own frame.
- Hence, when calling a function the following must be saved:
  - frame address FP of the caller;
  - code address from where to continue after the return (ie. program counter PC);
  - the maximal possible stack address of the caller EP.
- Simplification: we assume that return values fit into a single cell.

Functions

- We need to distinguish two kinds of variables:
  - *global* variables which are defined outside of functions;
  - *local* (or automatic) variables (incl. formal parameters) which are defined inside of fnctions.
- The address environment ρ binds variable names with pairs

$$(tag, a) \in \{G, L\} \times \mathbb{N}$$

- NB! Many languages restrict the scope of a variable inside block.
- Different parts of a program generally use different address environments.

Functions

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(”%d”, ith(l, i));
}
```

Functions

$\boxed{0}$

```
int i;
struct list {
   int info;
   struct list *next;
} *l;
```

$\boxed{1}$

```
int ith (struct list *x, int i) {
   if (i ≤ 1) return x→info;
   else return ith(x→next, i−1);
}
```

$\boxed{2}$

```
main () {
   int k;
   scanf("%d", &i);
   scanlist(&l);
   printf("%d", ith(l, i));
}
```

$\boxed{0}$ global env.

ρ_0

$$\begin{array}{rcl} i & \mapsto & (G, 1) \\ l & \mapsto & (G, 2) \\ \text{ith} & \mapsto & (G, \_ith) \\ \text{main} & \mapsto & (G, \_main) \end{array}$$

Functions

$\boxed{0}$

```
int i;
struct list {
   int info;
   struct list *next;
} *l;
```

$\boxed{1}$

```
int ith (struct list *x, int i) {
   if (i ≤ 1) return x→info;
   else return ith(x→next, i−1);
}
```

$\boxed{2}$

```
main () {
   int k;
   scanf("%d", &i);
   scanlist(&l);
   printf("%d", ith(l, i));
}
```

$\boxed{1}$ env. for function ith

$$\rho_1 \quad \begin{aligned} x &\mapsto (L, 1)\\ i &\mapsto (L, 2)\\ l &\mapsto (G, 2)\\ \mathrm{ith} &\mapsto (G, \_ith)\\ \mathrm{main} &\mapsto (G, \_main) \end{aligned}$$

Functions

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

---

2 env. for function main

ρ_2

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

Functions

- Let f be a function which calls another g.
- Function f is the caller and function g the callee.
- The code emitted for a function call is divided between the caller and the callee.
- The exact division depends from who has what information.

Functions

- Actions during the function call and entering to the callee:
  1. saving registers FP and EP; } `mark`
  2. computing actual arguments of the function;
  3. determining the start address $\_g$ of the callee;
  4. setting a new FP;
  5. saving PC and jumping to $\_g$; } `call`
  6. setting a new EP; } `enter`
  7. allocating space for local variables. } `alloc`
- Actions on leaving the callee:
  1. restoring registers FP, EP and SP;
  2. returning to f-s code; ie. restoring PC. } `return`

Functions

$$\begin{array}{rcl} \mathtt{code}_R\ (g(e_1,\ldots,e_n))\ \rho & = & \mathtt{mark} \\ & & \mathtt{code}_R\ e_1\ \rho \\ & & \ldots \\ & & \mathtt{code}_R\ e_n\ \rho \\ & & \mathtt{code}_R\ g\ \rho \\ & & \mathtt{call\ n} \end{array}$$

- Expressions standing for actual parameters are evaluated for their R-value
  - call-by-value parameter passing.
- Function g might be an expression which R-value is callee's starting address.

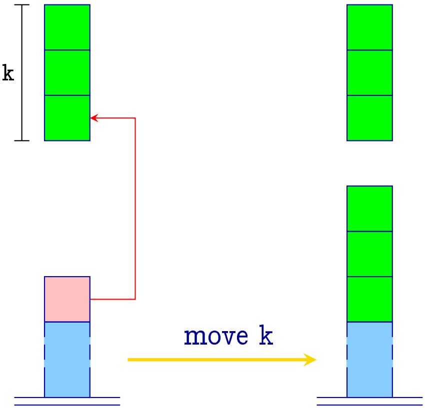
Functions

- Function name is a *pointer constant* which R-value is the starting address of the function code.
- Dereferencing a function pointer returns the same pointer.
  - Example: in the case of the declaration $\mathbf{int}\ (*)()g;$, the calls $g()$ and $(*g)()$ are equivalent.
- If arguments are structs, they are copied.

$$
\begin{array}{lcll}
\mathrm{code}_R\ f\ \rho & = & \mathtt{loadc}\ (\rho\ f) & f \text{ is a function name} \\
\mathrm{code}_R\ (*e)\ \rho & = & \mathrm{code}_R\ e\ \rho & e \text{ is a function pointer} \\
\mathrm{code}_R\ e\ \rho & = & \mathrm{code}_L\ e\ \rho & e \text{ is a struct of size } k \\
 & & \mathtt{move}\ \ \mathtt{k} &
\end{array}
$$

Functions

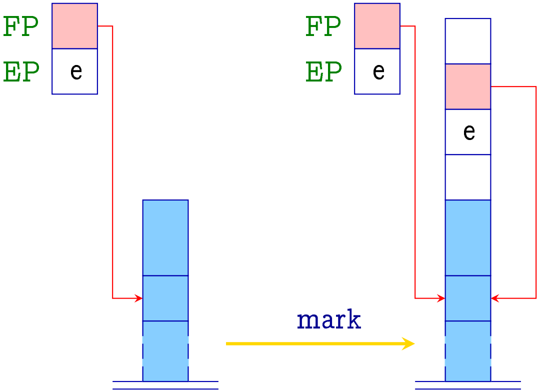
Instruction `move k` copies *k* cells to top of the stack:



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

Functions

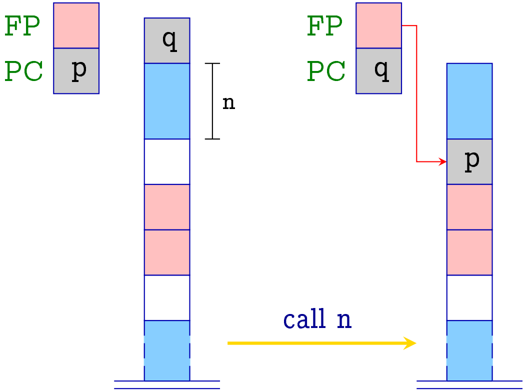
Instruction mark allocates space for organizational cells and for the return value, and saves registers FP and EP:



```
S[SP+2] = EP;
S[SP+3] = FP;
SP = SP + 4;
```

Functions

Instruction `call n` saves the continuation address and assigns new values to FP, SP and PC:



```
FP = SP - n - 1;
S[FP] = PC;
PC = S[SP];
SP--;
```

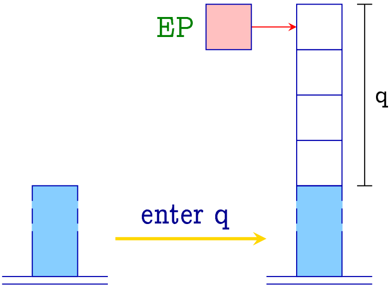
Functions

$$\begin{array}{lcll} \mathsf{code}\ (t\ f\ (args)\{vars\ ss\})\ \rho & = & \_f\!: & \mathtt{enter\ q} \\ & & & \mathtt{alloc\ k} \\ & & & \mathsf{code}\ (ss)\ \rho_f \\ & & & \mathtt{return} \end{array}$$

$$\begin{array}{llcl} \text{where} & q & = & maxS + k \\ & maxS & = & \text{maximum depth of the local stack} \\ & k & = & \text{space for local variables} \\ & \rho_f & = & f\text{-s aaddress environment} \end{array}$$

Functions

Instruction `enter q` sets the register EP:

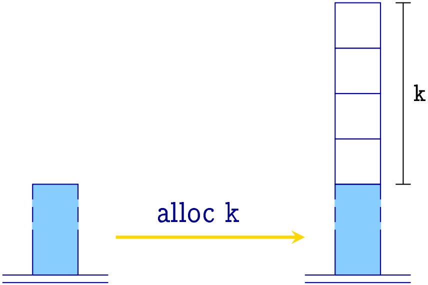


```
EP = SP + q;
if (EP≥NP)
    Error ("Stack Overflow");
```

NB! If there is not enough space, the execution is interrupted.

Functions

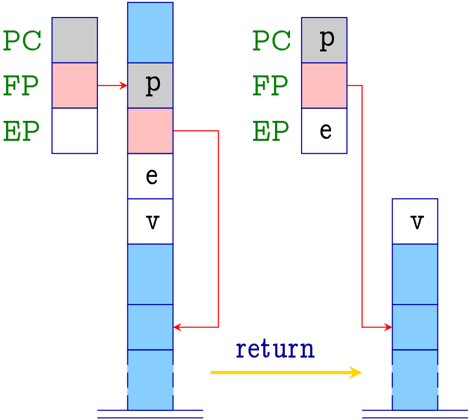
Instruction `alloc k` allocates space in stack for local variables:



`SP = SP + k;`

Functions

Instruction `return` restores registers PC, FP and EP, and leaves the return value in top of the stack:



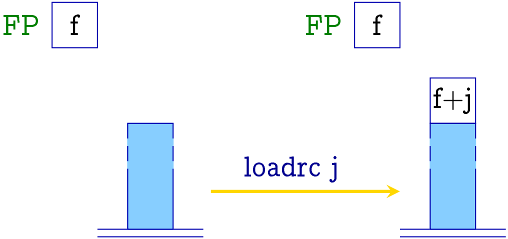
```
PC = S[FP];
EP = S[FP-2];
if (EP≥NP)
    Error ("Stack Overflow");
SP = FP - 3;
FP = S[SP+2];
```

Functions

The access to local variables and formal parameters is relative with respect to the register FP:

$$\text{code}_L\ x\ \rho \quad = \quad \begin{cases} \texttt{loadc j} & \text{if } \rho\ x = (G, j) \\ \texttt{loadrc j} & \text{if } \rho\ x = (L, j) \end{cases}$$

Instruction `loadrc j` calculates the sum of FP and `j`:



```
SP++;
S[SP] = FP + j;
```

Functions

Analoguously to instrctions `loada j` and `storea j` we introduce instructions `loadr j` and `storer j`:

```
loadr j   =  loadrc j
             load
storer j  =  loadrc j
             store
```

return-statement corresponds to the assignment to a variable with the relative address -3:

$$\text{code}\ (\mathbf{return}\ e;)\ \rho \quad = \quad \text{code}_R\ e\ \rho \quad \texttt{storer -3} \quad \texttt{return}$$

Functions

Example:

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

Then $\rho_{fac} = \{x \mapsto (L, 1)\}$ and the code to be emitted is:

```
_fac: enter 7     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
```

Compilation of the complete program

An initial state of the abstract machine:

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

Let $p \equiv vars\ fdef_1\ \ldots\ fdef_n$, where $fdef_i$ is a definition of function f_i and one of the functions has a name main.

The emitted code consists of following parts:

- code corresponding to function definitions $fdef_i$;
- allocation of memory for global variables;
- code of a call to the function main();
- instruction `halt`.

Compilation of the complete program

$$\begin{array}{lllrl}
\text{code}\ p\ \emptyset & = & \texttt{enter}\ (k+6) & & \texttt{pop} \\
 & & \texttt{alloc}\ (k+1) & & \texttt{halt} \\
 & & \texttt{mark} & \_f_1 : & \text{code}\ \mathit{fdef}_1\ \rho \\
 & & \texttt{loadc}\ \_\texttt{main} & & \ldots \\
 & & \texttt{call}\ \texttt{0} & \_f_n : & \text{code}\ \mathit{fdef}_n\ \rho
\end{array}$$

where $\emptyset$ = empty address environment

ρ = global address environment

k = space for global variables