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*. 
CMAa architecture

Stack:

\[ S = \text{Stack} - \text{memory area for data where insertion and deletion of items uses LIFO principle.} \]

\[ SP = \text{Stack-Pointer} - \text{register containing an address of the topmost item.} \]

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 instruction (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.
Problem: evaluate an expression like $(6 + 2) \times 4 - 1$; 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 assignment

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.

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.

```
loadc q
SP++;  
S[SP] = q;
```
Simple expressions and assignment

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 assignment

Example: operator `leq`

```
7
3
```

```
1
```

`leq`  

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

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

\[
\text{neg} \quad 7 \rightarrow -7
\]

\[
\text{not} \quad 3 \rightarrow 0
\]

\[
\text{if } (\text{S[SP]} \neq 0) \quad \text{S[SP]} = 0;
\]

\[
\text{else} \quad \text{S[SP]} = 1;
\]

\[
\text{S[SP]} = -\text{S[SP]};
\]
Simple expressions and assignment

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

```
loadc 1
loadc 7
add
```

Execution of the code results:
Simple expressions and assignment

- Variables correspond to cells of the stack $S$:

```
 x:
 y:
```

- Code generation is specified in terms of functions $\text{code}$, $\text{code}_L$ and $\text{code}_R$.

- 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 assignment

- Variables are used in two different ways.
- For instance, in the assignment $x = y + 1$ 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 whether we need its $L$-value or $R$-value.
  \[
  \begin{align*}
  \text{L-value of a variable} & \quad = \quad \text{its address} \\
  \text{R-value of a variable} & \quad = \quad \text{its ”real” value}
  \end{align*}
  \]
- Function $\text{code}_L\ e\ \rho$ emits a code computing a L-value of the expression $e$ in the environment $\rho$.
- Function $\text{code}_R\ e\ \rho$ does the same for the R-value.
- NB! Not every expression has a L-value (eg.: $x + 1$).
Simple expressions and assignment

- Compilation of binary operators:
  \[ \text{code}_R (e_1 + e_2) \rho = \text{code}_R e_1 \rho,\]
  \[ \text{code}_R e_2 \rho,\]
  \[ \text{add} \]
  - Similarly for other binary operators.

- Compilation of unary operators:
  \[ \text{code}_R (-e) \rho = \text{code}_R e \rho,\]
  \[ \text{neg} \]
  - Similarly for other unary operators.

- Compilation of primitive constant values:
  \[ \text{code}_R q \rho = \text{loadc} q \]
Simple expressions and assignment

- Compilation of variables:

  \[
  \text{code}_L \ x \ \rho \quad = \quad \text{loadc} \ (\rho \ x) \\
  \text{code}_R \ x \ \rho \quad = \quad \text{code}_L \ x \ \rho \\
  \quad \text{load}
  \]

- Compilation of assignment expressions:

  \[
  \text{code}_R \ (x = e) \ \rho \quad = \quad \text{code}_R \ e \ \rho \\
  \quad \text{code}_L \ x \ \rho \\
  \quad \text{store}
  \]
Simple expressions and assignment

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 assignment

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:

\[
\begin{align*}
S[S[SP]] &= S[SP-1] \\
SP &--
\end{align*}
\]

**NB!** Differs from the analogous P-machine instruction in the Wilhelm/Maurer book.
Simple expressions and assignment

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

then $\text{code}_R e \rho$ emits the code:

\[
\begin{align*}
\text{loadc} & \ 7 & \text{sub} \\
\text{load} & \ & \text{loadc} \ 4 \\
\text{loadc} & \ 1 & \text{store}
\end{align*}
\]

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

\[
\begin{align*}
\text{loada} \ q & = \text{loadc} \ q \\
\text{load} & \\
\text{storea} \ q & = \text{loadc} \ q \\
\text{store} &
\end{align*}
\]
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.

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

\[
\text{code } (s \ s s) \rho = \text{code } s \rho \\
\text{code } ss \rho
\]

\[
\text{code } \varepsilon \rho = \quad \text{// empty sequence}
\]

- Instruction \texttt{pop} removes the topmost stack cell:

```
    7
    
```

```
pop
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.
Instruction **jump A** performs an unconditional jump to the address A; the stack doesn’t change:

\[
\text{PC} \quad \rightarrow \quad \text{PC} = A;
\]
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 $s \equiv \text{if } (e) \ s_1$:
- generate a code for the condition $e$ and statement $s_1$;
- insert the conditional jump instruction in between.

\[
\text{code (if } (e) \ s_1) \ \rho \quad = \quad \begin{array}{c}
\text{code}_R \ e \ \rho \\
\text{jumpz} \\
\text{jumpz} \ A \\
\text{code } s_1 \ \rho \\
A: \ldots \\
\end{array}
\]
Conditional statements and loops

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

$$\text{code} \ (\text{if } (e) \ s_1 \ \text{else } s_2) \ \rho = \begin{cases} 
\text{code}_R \ e \ \rho \\
\text{jumpz } A \\
\text{code for } s_1 \\
\text{jump } B \\
A: \text{code } s_2 \ \rho \\
B: \ldots \\
\text{code for } s_2 \\
\ldots 
\end{cases}$$
Conditional statements and loops

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

$$s \equiv \begin{cases} x = x - y; & \text{if } (x > y) \\ y = y - x; & \text{else} \end{cases}$$

then code $s \rho$ emits a code:

$$\begin{array}{llll}
\text{loada 4} & \text{loada 4} & \text{A: loada 7} \\
\text{loada 7} & \text{loada 7} & \text{loada 4} \\
\text{ge} & \text{sub} & \text{sub} \\
\text{jumpz A} & \text{storea 4} & \text{storea 7} \\
\text{pop} & \text{pop} & \\
\text{jump B} & & \text{B: ...} \\
\end{array}$$
Conditional statements and loops

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

\[
\text{code } (\text{while } (e) \ s_1) \ \rho = \\
A: \ \text{code}_R \ e \ \rho \\
\quad \text{jumpz } B \\
\quad \text{code } s_1 \ \rho \\
\quad \text{jump } A \\
B: \ldots
\]

\[
\begin{align*}
\text{code}_R \text{ for } e \\
\text{jumpz} \\
\text{code for } s_1 \\
\text{jump} \\
\ldots
\end{align*}
\]
Conditional statements and loops

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

$$s \equiv \text{while} (a > 0) \{ \begin{array}{l}
c = c + 1; \quad (ii) \\
a = a - b; \quad (iii)
\end{array} \}$$

then code $s \rho$ emits a code:

A: loada 7 loada 9 loada 7 jump A
loadc 0 loadc 1 loada 8 B: ...
ge add sub
jumpz B storea 9 storea 7
pop pop (i) (ii) (iii)
Conditional statements and loops

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

  \[
  \text{code } (\text{for } (e_1; e_2; e_3) s_1) \rho = \text{code}_R e_1 \rho \\
  \text{pop } A: \text{code}_R e_2 \rho \\
  \text{jumpz } B \\
  \text{code } s_1 \rho \\
  \text{code}_R e_3 \rho \\
  \text{pop } A \\
  \text{jump } A \\
  B: \ldots
  \]
Conditional statements and loops

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

  ```
  switch (e) {
    x = e;
    case c_0 : ss_0 break; if (x == c_0) ss_0
    case c_1 : ss_1 break; else if (x == c_1) ss_1
    ...                                                \Rightarrow ...
    case c_{k-1} : ss_{k-1} break; else if (x == c_{k-1}) ss_{k-1}
    default : ss_k                                else ss_k
  }
  ```

- By sorting the labels and using binary search, it's possible to decrease the number of comparisons 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:

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

\[
\text{code } s \rho = \text{code}_R e \rho \\
\text{check } 0 \ k \ B \\
C_0: \text{code } ss_0 \ \rho \\
\quad \text{jump } D \\
\ldots \\
C_k: \text{code } ss_k \ \rho \\
\quad \text{jump } D \\
B: \text{jump } C_0 \\
\ldots \\
\text{jump } C_k \\
D: \ldots
\]

- 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 table" \(B\) contains a unconditional jump instruction to the beginning of the code corresponding to the \(i\)-th branch.
- Each branch ends with the unconditional jump.
The R-value of the condition is used both for comparison and indexing, hence it must be duplicated before comparisons.

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:

\[
\text{PC} \rightarrow \text{PC} = A + S[\text{SP}]; \quad \text{SP}--;
\]
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 $\rho$ s.t.

$$\rho \ x_i = i, \quad i = 1, \ldots, k$$
Arrays, records and static memory management

- Array is a sequence of memory cells.
- Uses integer indices for an access of its individual elements.
Arrays, records and static memory management

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

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

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

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

- Since `|·|` can be computed compile-time, it is also possible to compute the address environment \( \rho \) in compile-time.
Let $t \ a[c]$; be an array declaration.

Then, the address of its $i$-th element is $\rho \ a + |t| \times (\text{rval of } i)$

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

In general, an array can be given by an expression which must be evaluated before indexing.

In C, an array is a \textit{pointer-constant} which R-value is the start address of the array.
Arrays, records and static memory management

\[
\text{code}_L (e_1[e_2]) \rho = \text{code}_R e_1 \rho \\
\text{code}_R e_2 \rho \\
\text{loadc} |t| \\
\text{mul} \\
\text{add} \\
\text{code}_R e \rho = \text{code}_L e \rho \quad e \text{ is an array}
\]

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

\[
a[2] \quad 2[a] \quad a + 2
\]

**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 $\rho_{st}$.
- Let $\text{struct } \{ \text{int } a; \text{ int } b; \} \ x; \text{ be a declaration:}$
  - 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 $t \equiv \text{struct} \{ t_1 \, c_1; \ldots \, t_k \, c_k; \}$, then

  $|t| = \sum_{i=1}^{k} |t_i|$
  $\rho \, c_1 = 0$
  $\rho \, c_i = \rho \, c_{i-1} + |t_{i-1}| \quad i > 1$

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

  $\text{code}_L \, (e.c) \, \rho = \text{code}_L \, e \, \rho$
  $\text{loadc} \, (\rho \, c)$
  $\text{add}$
References and dynamic memory management

Heap:

\[ H = \text{Heap} \] — memory area for dynamically allocated data.

\[ NP = \text{New-Pointer} \] — register containing the address of the lowermost used cell in the heap.

\[ EP = \text{Extreme-Pointer} \] — register containing 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 anonymous, dynamically created, objects which life-time doesn’t follow LIFO principle.

- Pointer values are returned by the following operations:
  - a call to the function `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).

\[
\begin{align*}
\text{code}_R \ (\text{malloc}(e)) \ \rho & \ = \ \text{code}_R \ e \ \rho \\
\text{code}_R \ (&e) \ \rho & \ = \ \text{code}_L \ e \ \rho
\end{align*}
\]
References and dynamic memory management

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

```c
if (NP - S[SP] <= EP) { S[SP] = NULL; } else { NP = NP - S[SP]; S[SP] = NP; }
```
References and dynamic memory management

- Referenced values can be accessed by the following ways:
  - an application of the dereferencing operator \( \ast \) 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 \((\ast e).c\).

\[
\begin{align*}
\text{code}_L (\ast e) \rho &= \text{code}_R e \rho \\
\text{code}_L (e \rightarrow c) \rho &= \text{code}_R e \rho \\
&\quad \text{loadc} (\rho c) \\
&\quad \text{add}
\end{align*}
\]
References and dynamic memory management

Example: let be given the following declarations:

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

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;
- keeping track of the free list might be relatively costly.
References and dynamic memory management

- Alternative: in the case of function `free` do nothing.
  
  ```
  code (free(e);) \rho = code_R e \rho 
  pop
  ```

- 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:

\[
\text{code}_R f \rho = _f = \text{starting address of } f \text{ code}
\]

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

Example:

```c
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.
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:

- **SP**
- **FP**
  - PCold
  - FPold
  - EPold

- local variables
- formal parameters
- organizational cells
- return value
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 functions.

- The address environment $\rho$ 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

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));
}

global env.

\[\begin{align*}
\rho_0 & \quad i \mapsto (G, 1) \\
        & \quad l \mapsto (G, 2) \\
        & \quad ith \mapsto (G, _ith) \\
        & \quad main \mapsto (G, _main) \\
\end{align*}\]
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));
  }

env. for function ith

\[ \begin{align*}
\rho_1 & \quad x \mapsto (L, 1) \\
& \quad i \mapsto (L, 2) \\
& \quad l \mapsto (G, 2) \\
& \quad ith \mapsto (G, _ith) \\
& \quad main \mapsto (G, _main) \\
\end{align*} \]
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));
}

env. for function main
ρ2
  k → (L, 1)
  i → (G, 1)
  l → (G, 2)
  ith → (G, _ith)
  main → (G, _main)
**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;
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;
6. setting a new EP;
7. allocating space for local variables.

Actions on leaving the callee:

1. restoring registers FP, EP and SP;
2. returning to f-s code; ie. restoring PC.
Functions

\[
\text{code}_R \ (g(e_1, \ldots, e_n)) \ \rho = \begin{align*}
\text{mark} \\
\text{code}_R \ e_1 \ \rho \\
\vdots \\
\text{code}_R \ e_n \ \rho \\
\text{code}_R \ g \ \rho \\
\text{call} \ n
\end{align*}
\]

- 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 `int (*g)();`, the calls `g()` and `(*g)()` are equivalent.
- If arguments are structs, they are copied.

\[
\begin{align*}
\text{code}_R \ f \ \rho & = \ \text{loadc} \ (\rho \ f) & \text{if } f \text{ is a function name} \\
\text{code}_R \ (*e) \ \rho & = \ \text{code}_R \ e \ \rho & \text{if } e \text{ is a function pointer} \\
\text{code}_R \ e \ \rho & = \ \text{code}_L \ e \ \rho & \text{if } e \text{ is a struct of size } k \\
& \quad \text{move } k
\end{align*}
\]
Instruction move $k$ copies $k$ cells to top of the stack:

```
for (i=k-1; i\geq 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`:

\[
\begin{align*}
S[SP+2] &= EP; \\
S[SP+3] &= FP; \\
SP &= SP + 4;
\end{align*}
\]
Functions

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

\[
\text{FP} = \text{SP} - n - 1;
\text{S}[\text{FP}] = \text{PC};
\text{PC} = \text{S}[\text{SP}];
\text{SP}--;
\]
Functions

code \( (t f (\text{args})\{\text{vars ss}\}) \rho \) = \_f: \text{enter q}

 allocate \_k

code (ss) \rho_f

return

where \( q = maxS + k \)

\( maxS \) = maximum depth of the local stack

\( k \) = space for local variables

\( \rho_f \) = \( f \)-s aaddress environment
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:

\[
SP = SP + k;
\]
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**:

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

Instruction **loadrc** \( j \) calculates the sum of **FP** and \( j \):

\[
\text{FP} \ f \quad \quad \quad \quad \quad \quad \text{FP} \ f
\]

\[
\text{loadrc} \ j \quad \quad \quad \quad \quad \quad \text{SP}++; \\
S[SP] = \text{FP} + j;
\]
Functions

Analoguously to instructions \texttt{loada} \( j \) and \texttt{storea} \( j \) we introduce instructions \texttt{loadr} \( j \) and \texttt{storer} \( j \):

\[
\begin{align*}
\texttt{loadr} \ j & \quad = \quad \texttt{loadrc} \ j \\
\texttt{store} \ j & \quad = \quad \texttt{loadrc} \ j \\
\end{align*}
\]

\begin{align*}
\texttt{return}\text{-statement} & \quad \text{corresponds to the assignment to a variable} \\
\text{with the relative address} \ -3: \\
\texttt{code} \ (\texttt{return} \ e;) \ \rho & \quad = \quad \texttt{code}_R \ e \ \rho \\
\texttt{storer} \ -3 & \quad \text{and} \\
\texttt{return} & \\
\end{align*}
Functions

Example:

```c
int fac(int x) {
    if (x <= 0) return 1;
    else return x * fac(x - 1);
}
```

Then $\rho_{\text{fac}} = \{x \mapsto (L, 1)\}$ 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    sub
leq    jumpz A
jump B
```

B: return

```
loadc _fac
call 1
```
Compilation of the complete program

An initial state of the abstract machine:

\[ \text{SP} = -1 \quad \text{FP} = \text{EP} = 0 \quad \text{PC} = 0 \quad \text{NP} = \text{MAX} \]

Let \( p \equiv \text{vars } f\text{def}_1 \ldots f\text{def}_n \), where \( f\text{def}_i \) 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 \( f\text{def}_i \);
- allocation of memory for global variables;
- code of a call to the function \text{main}(\);
- instruction \text{halt}.
Compilation of the complete program

code $p \emptyset = \text{enter} (k + 6) \quad \text{pop}

\text{alloc} (k + 1) \quad \text{halt}

\text{mark} \quad _{f_1} : \text{code} \ fdef_1 \ \rho

\text{loadc} \ \text{main} \quad \ldots

\text{call} \ 0 \quad _{f_n} : \text{code} \ fdef_n \ \rho

\text{where } \emptyset = \text{empty address environment}
\rho = \text{global address environment}
k = \text{space for global variables}