Compiler Construction

Reinhard Wilhelm

Winter Term 07/08
This course is about the Implementation of Programming Languages.

There exist several alternative methods:

- **Interpretation**, which is natural (semantics)
- **Compilation**, which is efficient
Principle of Interpretation

Positive  No precomputation on the program text
           ⇒ short startup-time, fast reaction for small programs

Negative  Program parts are repeatedly analyzed during execution,
           less efficient access to program variables
           ⇒ slower execution speed
Principle of Compilation

Two Phases (at two different times):

- Translation of the source program into a machine program (at compile time)
- Execution of the machine program on input data (at run time)
The Essence of Compilation\textsuperscript{1}

Exploitation of Static Information to speed up execution.

Static vs. Dynamic Information

\begin{itemize}
\item \textbf{Static Information}
  \begin{itemize}
  \item available at compile time
  \item can be determined solely on the basis of the program text
  \item primary can be read off the program text
  \item derived can be computed from primary or derived information
  \end{itemize}
\item \textbf{Dynamic Information}
  \begin{itemize}
  \item only available at run time
  \end{itemize}
\end{itemize}

\textsuperscript{1}A. Ershov
Examples

static

primary
1. (the set of) declared variables
2. type of defining occurrences (statically typed languages)
3. size of arrays (C, Pascal), dimension of arrays

derived
1. space for declared variables
2. addresses for declared variables
3. defining occurrences for applied occurrences
4. type of applied occurrences.

dynamic

1. value of expressions, value of index expressions
2. value of pointers
3. defining occ. for appl. occ, types of appl. occ. (LISP)
4. invoked method (OO)
Trade-off: Compilation Time vs. Execution Time

Preprocessing of the source program provides for

- efficient access to the values of program variables at run time
- global program transformations to increase execution speed.

Disadvantage: Compilation takes time

Advantage: Program execution is sped up
⇒ compilation pays off in long running or often run programs
Structure of a Compiler

Source program → Frontend → Optimizations → Code generation → Internal representation (Syntax tree) → Internal representation → Program for target machine
Treated in this course:

1. Virtual machine CMa for C,
2. Code generation for C and the CMa,
3. Frontend (lexical, syntactic, and semantic analysis),
4. Analysis and optimization,

1. and 2. treated in the textbook:

3. - 5. treated in textbook:
Wilhelm/Maurer: Übersetzerbau, Springer 1997
Wilhelm/Maurer: Compiler Design, Addison-Wesley 1995
Subtasks in Code Generation for Real Machines

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:
Structure of a Compiler

- Intermediate representation → Code generation → abstract machine code
- abstract machine code → Compiler → concrete machine code
- Alternatively: Input → Interpreter → Output

- Code generation
- abstract machine code
- Compiler
- concrete machine code
- Input
- Interpreter
- Output
Virtual Machines

Virtual Machine (also called Abstract Machines)

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

Advantages

1. Porting the compiler to a new target architecture is simpler.
2. Modularization makes the compiler easier to modify.
3. Translation of program constructs is separated from the exploitation of architectural features.
4. Virtual machine can be used as a security device (cf. JVM).
Virtual (Abstract) Machines

Virtual machines for some programming languages:

- Algol 60 → Algol Object Code
- Pascal → P-machine
- SmallTalk → Bytecode
- Prolog → WAM ("Warren Abstract Machine")
- SML, Haskell → STGM
- Java → JVM
The Translation of C
The Architecture of Virtual Machines

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

The CMa is a virtual machine for C.
For the CMa we need:
The Data Store

\( S \) (data) store onto which new cells are allocated in a LIFO discipline
\[ \Rightarrow \text{Stack}. \]

\( SP \) (Stack Pointer) register which contains the address (index) of the topmost allocated cell
\[ \text{Simplifying assumption: All types of scalar data fit into one cell of } S. \]
Stack Machines vs. Register Machines

The CMa is a Stack Machine, unlike machines one finds in nature. These are Register Machines.

**Stack Machines**: Operations are performed (on the values in) the topmost cells of the stack,

**Register Machines**: Operations are performed (on the values in) registers.
The Code/Instruction Store

C Code store which contains the program. Each cell of field C can store exactly one virtual instruction.

PC (Program Counter) register which contains the address (index) of the instruction to be executed next. Initially, PC contains the address 0. ⇒ 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 an Instruction Register IR and executes it.
- PC is incremented by 1 before the execution of the instruction.

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

- Instruction execution 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.
Simple Expressions and Assignments

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

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

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.
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 \texttt{leq}

\[
\begin{array}{c}
7 \\
3 \\
\end{array}
\quad \texttt{leq} \quad \begin{array}{c}
1 \\
\end{array}
\]

Remark: 0 represents \textit{false}, all other integers \textit{true}.

Unary operators \texttt{neg} and \texttt{not} consume one operand and produce one result.

\[
\begin{array}{c}
8 \\
\end{array}
\quad \texttt{neg} \quad \begin{array}{c}
-8 \\
\end{array}
\]

\[S[SP] \leftarrow - S[SP];\]
Example: Code for $1 + 7$

Execution of this code sequence:
Variables

Variables are associated with cells in \( S \): 

\[
\begin{array}{c}
\text{z:} \\
\text{y:} \\
\text{x:}
\end{array}
\]

Code generation will be described by some Translation Functions, \( \text{code} \), \( \text{code}_L \), 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.

Note: The functions \( \text{code} \), \( \text{code}_L \), and \( \text{code}_R \) and the address environment \( \rho \) are part of the compiler!
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.

\[
\begin{align*}
\text{L-value of } x & = \text{ address of } x \\
\text{R-value of } x & = \text{ content of } x
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \text{code}_R ) e ( \rho )</th>
<th>produces code to compute the R-value of e in the address environment ( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{code}_L ) e ( \rho )</td>
<td>analogously for the L-value</td>
</tr>
</tbody>
</table>
Note:

- Not every expression has an L-value (Ex.: \( x + 1 \) where \( x \) is not of pointer type) \( \rightarrow \) not assignable.
- Some objects have L-values, but cannot be assigned to (Ex. arrays).
- \texttt{code}_L computes L-values, even for non-assignable arguments.
- The compiler will discover that it should not generate code that uses L-values for assignments in these cases.
We define:

\[
\text{code}_R (e_1 + e_2) \rho = \text{code}_R e_1 \rho \\
\text{code}_R e_2 \rho \\
\text{add}
\]

... analogously for the other binary operators

\[
\text{code}_R (-e) \rho = \text{code}_R e \rho \\
\text{neg}
\]

... analogously for the other unary operators

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

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

\[
S[SP] \leftarrow S[S[SP]]; 
\]
store

\[
\text{code}_R \ (x = e) \ \rho \ = \ \text{code}_R \ e \ \rho \\
\text{code}_L \ x \ \rho \\
\text{store}
\]

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

\[
\text{S[S[SP]]} \gets \text{S[SP-1]}; \\
\text{SP} \gets \text{SP - 1};
\]
Example:

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

\(\text{code}_R\ e\ \rho\) produces:

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

Improvements:

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

\[
\begin{align*}
\text{loada q} &= \text{loadc q} \\
\text{storea q} &= \text{loadc q}
\end{align*}
\]
Statements and Statement Sequences

If \( e \) is 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.

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

The instruction \( \text{pop} \) eliminates the top element of the stack.

\[
\text{SP } \leftarrow \text{SP } - 1;
\]
Statements and Statement Sequences

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

\[
\text{code } (s \; ss) \; \rho \; = \; \text{code } s \; \rho \\
\text{code } ss \; \rho \\
\text{code } \varepsilon \; \rho \; = \; \text{//empty sequence of instructions}
\]
Conditional and Iterative Statements

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

```
PC ← A;
```
Conditional and Iterative Statements

if (S[SP] == 0) PC ← A;
SP ← SP − 1;
Conditional and Iterative Statements

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.
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 (\textit{jump on zero}) in between.

\[
\text{code } s \; \rho \quad = \quad \text{code}_R \; e \; \rho \\
\text{jumpz } A \\
\text{code } s' \; \rho \\
A : \ldots
\]
Two-sided Conditional Statement

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

\[
\text{code } s \ \rho = \ \text{code}_R \ e \ \rho \\
\text{jumpz } A \\
\text{code } s_1 \ \rho \\
\text{jump } B \\
A : \ \text{code } s_2 \ \rho \\
B : \ldots
\]
Example:

Let \( \rho = \{ x \mapsto 4, y \mapsto 7 \} \) and

\[
\begin{align*}
    s & \equiv \text{if}(x > y) & (i) \\
    & x = x - y; & (ii) \\
    & \text{else } y = y - x; & (iii)
\end{align*}
\]

code \( s \rho \) produces:

\[
\begin{array}{l l l l}
    \text{loada 4} & \text{loada 4} & A: \text{loada 7} \\
    \text{loada 7} & \text{loada 7} & \text{loada 4} \\
    \text{gr} & \text{sub} & \text{sub} \\
    \text{jumpz A} & \text{storea 4} & \text{storea 7} \\
    & \text{pop} & \text{pop} \\
    & \text{jump B} & \text{jump B} \\
    (i) & (ii) & (iii)
\end{array}
\]
while-Loops

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

```
code s \rho \ = 
  A : \ \text{code}_R \ e \ \rho
  \text{jumpz} \ B
  \text{code} \ s' \ \rho
  \text{jump} \ A
B : \ \ldots
```
Example

Let \( \rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\} \) and \( s \) the statement:

\[
\text{while} \ (a > 0) \ \{ \ c = c + 1; \ a = a - b; \ \}
\]

\( \text{code } s \ \rho \) produces the sequence:

A: 
- \text{loada 7}
- \text{loadc 0}
- \text{gr}
- \text{jumpz B}

B: 
- \text{loada 7}
- \text{loadc 1}
- \text{add}
- \text{storea 9}
- \text{pop}
- \text{jump A}

- \text{loada 8}
- \text{sub}
- \text{storea 7}
- \text{pop}

. . .
for-Loops

The for-loop \( s \equiv \text{for} \ (e_1; e_2; e_3) \ s' \) is equivalent to the statement sequence \( e_1; \ \text{while} \ (e_2) \ \{ s' \ e_3; \} \) – provided that \( s' \) contains no continue-statement.

We therefore translate:

\[
\begin{align*}
\text{code } s \ \rho &= \ \text{code}_{\text{R}} \ e_1 \\
\text{pop} \\
A : \ &\text{code}_{\text{R}} \ e_2 \ \rho \\
\text{jumpz } B \\
\text{code } s' \ \rho \\
\text{code}_{\text{R}} \ e_3 \ \rho \\
\text{pop} \\
\text{jump } A \\
B : \ \ldots
\end{align*}
\]
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.

\[
\text{PC} \leftarrow \text{B} + \text{S}[\text{SP}];
\text{SP} \leftarrow \text{SP} - 1;
\]
We only regard `switch`-statements of the following form:

\[
    s \equiv \text{switch} (e) \{
        \text{case } 0: \quad ss_0 \text{ break;} \\
        \text{case } 1: \quad ss_1 \text{ break;} \\
        \vdots \\
        \text{case } k - 1: \quad ss_{k-1} \text{ break;} \\
        \text{default: } \quad ss_k
    \}
\]

`s` is then translated into the instruction sequence:
code $s \rho = code_R e \rho$
check $0 \ k \ B$

C$_0$: code $ss_0 \rho$
jump D

... jump C$_k$

C$_k$: code $ss_k \rho$
jump D

B: jump C$_0$

- The Macro $\text{check } 0 \ k \ B$ 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.

- Each alternative unconditionally jumps out of the $\text{switch}$-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 \texttt{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.
dup

\[ S[SP + 1] \leftarrow S[SP]; \]
\[ SP \leftarrow SP + 1; \]
Remarks

- 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.
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

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

A set of consecutive memory cells, of static size.
Access through integer indices starting at 0.
Example: \texttt{int a[11];}
The array \texttt{a} consists of 11 components and therefore needs 11 cells.
\( \rho a \) is the address of the component \texttt{a[0]}.
We need a function `sizeof` (notation: `| · |`), 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 \( d \equiv t_1 \times_1; \ldots; t_k \times_k; \)

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

Since `| · |` can be computed at compile time, also `\rho` can be computed at compile time.
Extend $\text{code}_L$ and $\text{code}_R$ to expressions with array components. Let $t[c] a$ be the definition of an array $a$. To determine the start address of a component $a[i]$, we compute $\rho a + |t| \times (R$-value of $i$).

In consequence:

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

... or more general:

$$\text{code}_L \ e_1[e_2] \ \rho \ = \ \text{code}_R \ e_1 \ \rho$$
$$\text{code}_R \ e_2 \ \rho$$
$$\text{loadc} \ |t|$$
$$\text{mul}$$
$$\text{add}$$
Remarks

- 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 $e$: \[ \text{code}_R \ e \ \rho = \text{code}_L \ e \ \rho \]
- In C, the following are equivalent (as L-values):
  \[ a[2] = * (a + 2) \]

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.
Structures (Records)

A set of named components of possibly different types. Access through the component names (selectors).
Simplifying assumption:
Names of structure components are not used elsewhere. Alternatively, one could manage a separate environment $\rho_{st}$ for each structure type $st$.
Let $\textbf{struct} \{ \textbf{int} \ a; \textbf{int} \ b; \} \ x; \quad \text{be part of a declaration list.}$

- $x$ 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 \( t \equiv \text{struct} \{ t_1, c_1; \ldots; t_k, c_k; \} \). We have

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

\[
\rho \cdot c_1 = 0 \quad \text{and} \quad \rho \cdot c_i = \rho \cdot c_{i-1} + |t_{i-1}| \quad \text{for} \quad i > 1
\]

We thus obtain:

\[
\text{code}_L (e.c) \rho = \text{code}_L e \rho
\]

\[
\text{loadc} (\rho c)
\]

\[
\text{add}
\]
Example

Let \( \text{struct} \{ \text{int} \ a; \text{int} \ b; \} \ x; \) such that \( \rho = \{ x \mapsto 13, a \mapsto 0, b \mapsto 1 \}. \)

This yields:

\[
\text{code}_L \ (x.b) \ \rho = \begin{align*}
\text{loadc} & \ 13 \\
\text{loadc} & \ 1 \\
\text{add} & \\
\end{align*}
\]
Pointer and Dynamic Storage Management

Pointers allow the access to anonymous, dynamically allocated objects, whose lifetime is not subject to the LIFO-principle. ⇒ Need another potentially unbounded storage area $H$ – the Heap.

NP $\triangleq$ New Pointer; points to the lowest occupied heap cell.

EP $\triangleq$ 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’s distance to SP 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 are two ways to set a pointer:

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

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

   new

2. The application of the address operator `&` to a variable returns a pointer to this variable, i.e. its address (\(\equiv\text{L-value}\)). Therefore:

   \[
   \text{code}_R (\&e) \rho = \text{code}_L e \rho
   \]
new

if (NP - S[SP] ≤ EP)
    S[SP] ← NULL;
else {
    NP ← NP - S[SP];
    S[SP] ← NP;
}
new cont’d

- NULL is a special pointer constant, identified with the integer constant 0.
- The NULL-pointer is returned in the case of a collision of stack and heap.
Dereferencing of Pointers

Let $e$ be a pointer-valued expression. The application of the operator $* \ e$ to the expression $e$ returns:

- as R-value the contents and
- as L-value the address of the storage cell whose address is the R-value of $e$,

$$\text{code}_L \ (\ast e) \ \rho = \text{code}_R \ e \ \rho$$

Dereferencing of a pointer component $a$ of a structure pointed to by $e$

$$e \rightarrow a \equiv (\ast e).a$$
Example

Given the definition

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

and the expression 

```c
((pt → b) → a)[i + 1]
```

Because of

```c
e → a ≡ (*e).a
```

holds:

```
\text{code}_L \ (e \rightarrow a) \ \rho \ = \ \text{code}_R \ e \ \rho \\
\text{loadc} \ (\rho \ a) \\
\text{add}
```
Example cont’d
Let \( \rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\} \). Then:

\[
\text{code}_L \left( (pt \rightarrow b) \rightarrow a \right)[i + 1] \rho = \text{code}_R \left( (pt \rightarrow b) \rightarrow a \right) \rho = \text{code}_R \left( (pt \rightarrow b) \rightarrow a \right) \rho
\]

loada 1
loadc 1
mul
add
add
\[ \text{code}_R \ ( (pt \rightarrow b) \rightarrow a ) \ \rho \ = \ \text{code}_R \ ( pt \rightarrow b ) \ \rho \ = \ \text{loada} \ 3 \ \\
\text{loadc} \ 0 \ \\
\text{add} \ \\
\text{loada} \ 1 \ \\
\text{loadc} \ 1 \ \\
\text{mul} \ \\
\text{loadc} \ 0 \ \\
\text{add} \] 

In total, we obtain the instruction sequence:
Conclusion

We tabulate the cases of the translation of expressions:

\[
\begin{align*}
\text{code}_L (e_1[e_2]) \; \rho & = \text{code}_R e_1 \; \rho \\
& \quad \text{code}_R e_2 \; \rho \\
& \quad \text{loadc} \; |t| \\
& \quad \text{mul} \\
& \quad \text{add} \quad \text{if } e_1 \text{ has type pointer to } t \text{ or } t[] \\
\text{code}_L (e.a) \; \rho & = \text{code}_L e \; \rho \\
& \quad \text{loadc} \; (\rho \; a) \\
& \quad \text{add}
\end{align*}
\]
\[ \text{code}_L \ (*e) \ \rho \quad = \quad \text{code}_R \ e \ \rho \]
\[ \text{code}_L \ x \ \rho \quad = \quad \text{loadc} \ (\rho \ x) \]
\[ \text{code}_R \ (&e) \ \rho \quad = \quad \text{code}_L \ e \ \rho \]
\[ \text{code}_R \ e \ \rho \quad = \quad \text{code}_L \ e \ \rho \quad \quad \text{if e is an array} \]
\[ \text{code}_R \ (e_1 + e_2) \ \rho \quad = \quad \text{code}_R \ e_1 \ \rho \]
\[ \text{code}_R \ e_2 \ \rho \quad \text{loadc} \ |t| \]
\[ \text{mul} \quad \text{scaling} \]
\[ \text{add} \]
\[
\text{code}_R \ (e_1 \quad \square \quad e_2) \ \rho \ = \ \text{code}_R \ e_1 \ \rho \\
\text{code}_R \ e_2 \ \rho \\
op \ \rho \quad \text{op instruction for} \ \quad \square
\]

\[
\text{code}_R \ q \ \rho \ = \ \text{loadc} \ q \\
q \ \text{constant}
\]

\[
\text{code}_R \ (e_1 = e_2) \ \rho \ = \ \text{code}_R \ e_2 \ \rho \\
\text{code}_L \ e_1 \ \rho \\
\text{store}
\]

\[
\text{code}_R \ e \ \rho \ = \ \text{code}_L \ e \ \rho \\
\text{load} \\
otherwise
\]
**Example**

```plaintext
int a[10], *b; with ρ = \{ a \mapsto 7, b \mapsto 17 \}.
Consider the statement: s₁ ≡ *a = 5;
We then have:

\[
\begin{align*}
\text{code}_L (*a) \rho & = \text{code}_R a \rho = \text{code}_L a \rho = \text{loadc } 7 \\
\text{code } s₁ \rho & = \text{loadc } 5 \\
& \text{loadc } 7 \\
& \text{store} \\
& \text{pop}
\end{align*}
\]

Exercise: Translate:

s₂ ≡ b = (&a) + 2; and s₃ ≡ *(b + 3) = 5;
```
\[ s_2 \equiv b = (\& a) + 2; \quad \text{and} \quad s_3 \equiv *(b + 3) = 5; \]

\[
\text{code } (s_2 \ s_3) \rho = \begin{array}{l}
\text{loadc 7} \\
\text{loadc 2} \\
\text{loadc 1} \\
\text{mul} \\
\text{add} \\
\text{loadc 17} \\
\text{store} \\
\text{pop} \\
\end{array} \quad \begin{array}{l}
\text{loadc 5} \\
\text{loadc 17} \\
\text{load} \\
\text{loadc 3} \\
\text{loadc 1} \\
\text{mul} \\
\text{add} \\
\text{store} \\
\text{pop} \\
\end{array}
\]

\text{end of } s_2 \quad \text{end of } s_3
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
```
Freeing Occupied Storage - Potential Solutions

- Trust the programmer. Manage freed storage in a particular data structure (free list)
  ⇒ malloc or free may become expensive.
- Do nothing, i.e.:
  \[
  \text{code free}(e); \quad \rho \quad = \quad \text{code}_R \ e \ \rho \\
  \text{pop}
  \]
  ⇒ simple and (in general) efficient.
- Use an automatic, potentially “conservative” Garbage Collection, which occasionally collects certainly inaccessible heap space.
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:

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

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

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

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

At 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 set of instances of an execution forms a recursion tree. The recursion tree in the example:

```
        main
        /
       /  
      fac  fac  printf
     /  /  
    fac fac
   /  
  fac
```
Conclusion

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

**Idea:**
Allocate a dedicated 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**.
FP $\equiv$ 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.

Simplifying assumption: The return value fits into one storage cell.

Translation tasks for functions:

- Generation of code for the body.
- Generation of code for calls.
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.

⇒

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)

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

int ith (struct list * x, int i) {
    if (i <= 1) return x -> info;
    else return ith (x -> next, i - 1);
}
```

```
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, _ith) 
    main \mapsto (G, _main) 
    ... 
```
Example (2)

0 \hspace{1cm} \textbf{int} i;
\hspace{1cm} \textbf{struct} list \{ 
\hspace{2cm} \textbf{int} \textit{info};
\hspace{2cm} \textbf{struct} list * next;
\hspace{1cm} \} * l;

1 \hspace{1cm} \textbf{int} \textit{ith} (\textbf{struct} list * \textit{x}, \textbf{int} \textit{i}) \{ 
\hspace{2cm} \textbf{if} (\textit{i} \leq 1) \textbf{return} \textit{x} \rightarrow \textit{info};
\hspace{2cm} \textbf{else} \textbf{return} \textit{ith} (\textit{x} \rightarrow \textit{next}, \textit{i} - 1);
\}

2 \hspace{1cm} \textbf{main} () \{ 
\hspace{2cm} \textbf{int} \textit{k};
\hspace{2cm} \textbf{scanf} ("%d", &\textit{i});
\hspace{2cm} \textbf{scanlist} (&\textit{l});
\hspace{2cm} \textbf{printf} ("\n\n\t%d\n", \textit{ith} (\textit{l}, \textit{i}));
\}

1 \hspace{1cm} \textbf{inside of \textit{ith}}: 
\rho_1 : \begin{align*}
\textit{i} & \mapsto (L, 2) \\
\textit{x} & \mapsto (L, 1) \\
\textit{l} & \mapsto (G, 2) \\
\textit{ith} & \mapsto (G, \_\text{ith}) \\
\textbf{main} & \mapsto (G, \_\text{main}) \\
\end{align*} 
\ldots
Example (3)

1 \[ \text{int } i; \]
   \[ \text{struct list } \{ \]
   \[ \text{  int } \text{info}; \]
   \[ \text{  struct list } * \text{next}; \]
   \[ \} * l; \]

2 \[ \text{main () } \{ \]
   \[ \text{  int } k; \]
   \[ \text{  scanf } ("%d", &i); \]
   \[ \text{  scanlist } (&l); \]
   \[ \text{  printf } ("\n\t%d\nn", \text{ith } (l,i)); \]
\[ \} \]

2 \[ \text{inside of main:} \]
\[ \begin{align*}
\rho_2 : & \quad i \mapsto (G, 1) \\
& \quad l \mapsto (G, 2) \\
& \quad k \mapsto (L, 1) \\
& \quad \text{ith} \mapsto (G, \_\text{ith}) \\
& \quad \text{main} \mapsto (G, \_\text{main}) \\
& \ldots
\end{align*} \]

0 \[ \begin{align*}
\text{int } i; \\
\text{struct list } \{ \\
\text{  int } \text{info}; \\
\text{  struct list } * \text{next}; \\
\} * l;
\end{align*} \]
Let $f$ be 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  
2. Computing the actual parameters  
3. Determining the start address of \( g \)  
4. Setting the new FP  
5. Saving PC and  
   jump to the beginning of \( g \)  
6. Setting the new EP  
7. Allocating the local variables  

Actions upon leaving \( g \):  
1. Restoring the registers FP, EP, SP  
   Returning to the code of \( f \),  
2. i.e. restoring the PC
We generate for a call:

\[ \text{code}_R \ g(e_1, \ldots, e_k) \ \rho = \text{mark} \]
\[ \text{code}_R \ e_1 \ \rho \]
\[ \ldots \]
\[ \text{code}_R \ e_k \ \rho \]
\[ \text{code}_R \ g \ \rho \]
\[ \text{call} \ n \]

where \( n \) is the size of the space for the actual parameters.

Note:

- Expressions occurring as actual parameters will be evaluated to their **R-value** \( \Rightarrow \) 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 \( \text{int } (*)() \ g; \), the two calls

\[
(*g)() \quad \text{and} \quad g()
\]

are equivalent!

Normalization: Dereferencing of a function pointer is ignored.

Structures are copied when they are passed as parameters.

In consequence:

\[
\begin{align*}
\text{code}_R f \rho & = \rho f & f \text{ a function name} \\
\text{code}_R (*e) \rho & = \text{code}_R e \rho & e \text{ a function pointer} \\
\text{code}_R e \rho & = \text{code}_L e \rho & \text{move } k \text{ e a structure of size } k
\end{align*}
\]
for (i = k-1; i ≥ 0; i--) 
    S[SP+i] ← S[S[SP]+i];
SP ← SP+k-1;
Mark

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

\[ S[SP+2] \leftarrow EP; \]
\[ S[SP+3] \leftarrow FP; \]
\[ SP \leftarrow SP + 4; \]
call

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:

```plaintext
code t g (specs) { V_defss } \rho =
```

where

- `t` = return type of `g` with `|t| \leq 1`
- `q` = `maxS + k` (wobei)
- `maxS` = maximal depth of the local stack
- `k` = space for the local variables
- `\rho_g` = address environment for `g`

`takes care of specs, V_defss and \rho`
The instruction \texttt{enter q} sets \texttt{EP} to its new value. Program execution is terminated if not enough space is available.

\[
\text{EP} \leftarrow \text{SP} + q;
\]  
\[
\text{if } (\text{EP} \geq \text{NP})
\]  
\[
\text{Error ("Stack Overflow")};
\]
The instruction `alloc k` reserves stack space for the local variables.

\[ SP \leftarrow SP + q; \]
return

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 ← S[FP]; EP ← S[FP-2];
if (EP ≥ NP) Error ("Stack Overflow");
SP ← FP-3; FP ← S[SP+2];
```
The addressing of local variables and formal parameters is relative to the actual FP. We therefore modify $\text{code}_L$ for the case of variable names. For $\rho \ x = (\text{tag}, j)$ we define

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

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

\[ \text{FP} \quad f \quad \text{loadrc} \quad j \quad \text{FP} \quad f \quad f+j \]

\[ \text{SP} \leftarrow \text{SP} + 1; \]
\[ \text{S[SP]} \leftarrow \text{FP} + j; \]
loadr

As an optimization one introduces the instructions \( \text{loadr} \ j \) and \( \text{storer} \ j \). This is analogous to \( \text{loada} \ j \) and \( \text{storea} \ j \).

\[
\text{loadr} \ j \quad = \quad \text{loadrc} \ j \\
\text{load}
\]

\[
\text{storer} \ j \quad = \quad \text{loadrc} \ j \\
\text{store}
\]

The code for \( \text{return} \ e; \) corresponds to an assignment to a variable with relative address \(-3\).

\[
\text{code} \ \text{return} \ e; \ \rho \quad = \quad \text{code}_R \ e \ \rho \\
\text{storer} \ -3 \\
\text{return}
\]
Example

For the function

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

we generate:

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

where \( ρ_{fac} : x \mapsto (L, 1) \) and \( q = 1 + 4 + 2 = 7 \).
Translation of Whole Programs

The state before program execution starts:

\[
SP \leftarrow -1 \quad FP \leftarrow EP \leftarrow 0 \quad PC \leftarrow 0 \quad NP \leftarrow MAX
\]

Let \( p \equiv V_{\text{defs}} F_{\text{def}_1} \ldots F_{\text{def}_n} \) be a program where \( F_{\text{def}_i} \) defines a function \( f_i \), of which one is named \( \text{main} \).

The code for the program \( p \) consists of:

- Code for the function definitions \( F_{\text{def}_i} \);
- Code for allocating the global variables;
- Code for the call of \( \text{main}() \);
- the instruction \( \text{halt} \).
We thus define for  \( p \equiv V_{defs} \ F_{def_1} \ldots F_{def_n} \):

\[
\text{code } p \emptyset \quad = \quad \begin{align*}
&\text{enter } (k + 5) & \text{set EP} \\
&\text{alloc } k & \text{allocate global variables} \\
&\text{mark} & \text{create stack frame} \\
&\text{loadc } \_\text{main} \\
&\text{call } 0 & \text{call main} \\
&\text{halt} \\
_{f_1}: & \text{code } F_{def_1} \rho \\
&\vdots & \quad \\
_{f_n}: & \text{code } F_{def_n} \rho 
\end{align*}
\]

where  \( \emptyset \equiv \) empty address environment;  \\
\( \rho \equiv \) global address environment;  \\
\( k \equiv \) space for global variables