
Imperative Programming Languages

We start our presentation with the translation of an imperative programming language. As an example, we consider a subset of the programming language C. Our target language for translation is the instruction set of a suitable virtual machine, which has been specifically designed for this purpose. This virtual machine is called *C-Machine* or CMA for short.

2.1 Language Concepts and Their Compilation

Imperative programming languages offer, among other things, the following concepts, which have to be mapped onto concepts, and control sequences of virtual or real computers:

Variables. Variables are containers for data objects whose contents (value) can be changed during the execution of the program. Values may change through the execution of *statements* such as assignments. Multiple variables can be grouped into aggregates such as arrays and records (structs). The current values of the variables at any given time constitute parts of the *state* of the program execution at this particular time. Variables in programs are identified by individual *names*. As constants, functions, etc., are also identified by names, these names are called *variable identifiers*, *constant identifiers*, etc., if these kinds of names are to be differentiated. Variable identifiers must be mapped to memory locations, which during program execution will contain their values. If the programming language provides functions with local variables, new *instances* of the local variable identifier are created at function calls. This means that also new memory locations must be allocated to these. When the function terminates, the locations for the local instances can again be released. This kind of memory management can conveniently be implemented by means of a run-time stack.

Expressions. Expressions are terms consisting of constants, names, and operators which can be *evaluated* to values. In general, their values depend on the current program state, since each evaluation of an expression *e* uses the current values of the variables occurring in *e*.

Explicit Specification of Control Flow. Most imperative programming languages provide a jump statement, **goto**, which can be translated directly into the unconditional jump instruction of the target machine. Higher-level control constructs such as conditional statements (*if*) or iterative statements (*while*, *do-while*, *for*) are compiled by means of *conditional jumps*. A conditional jump typically follows an instruction sequence for the evaluation of a condition. Case distinctions (*switch-case*) can be efficiently realized through *indexed jumps*. Thereby the jump target address, given in the instruction, is modified according to a previously calculated value.

Functions. Functions and procedures serve as *functional abstraction*, which creates a new statement from a possibly complex statement or sequence of statements. A *call* to this newly defined statement at a program location executes the sequences of statements specified by the definition of the function. After its completion, execution returns with the value computed by the function — given there is any. If the function has *formal parameters*, the function can be called with different *actual parameters*. For the implementation of functions, the instruction set of the machine should provide a jump instruction that memorizes its origin so that control can return to the location of the call. The body of the function must be supplied with the actual parameters at each evaluation (call). These parameters together with the instances of local variables, are conveniently maintained by a stack-style memory management, which is often supported by dedicated machine instructions.

2.2 The Architecture of the C-Machine

Each virtual machine provides a set of *instructions*, which are executed on a virtual hardware. This virtual hardware is mostly *emulated* in software. The execution state is thereby saved in data structures, which are accessed by the instructions and managed by the *run-time system*.

For the sake of clarity, we introduce the architecture and the instructions step by step, as and when required for translating concepts from the source language. We start by introducing the basic memory layout, some registers, and the *main execution cycle* of the C-Machine.

The C-Machine has a *main memory* S of length $\max S + 1$. At its lower end, that is, from address 0 onwards, lies a *stack* that can grow and shrink. The register SP (Stack Pointer) always points to the topmost location in the stack. For all instructions of the virtual machine, we adopt the convention that their operands are expected on top of the stack and that their results (if any) are also delivered there. As a simplification, we assume that values of the scalar types fit into one memory location of main memory. As scalar types, we only consider **int** and pointer types, that is, addresses.

The *program store* of the C-Machine contains the program to be executed. It has length $\max C + 1$. At each clock cycle, one instruction of the C-Machine is fetched from some location of the program store for execution. The *Program Counter*, the register PC , always contains the address of the next instruction to be executed. This is loaded into an *Instruction Register*, IR , and subsequently executed. Before execution, the content of the program counter PC is incremented by 1, which in a sequential

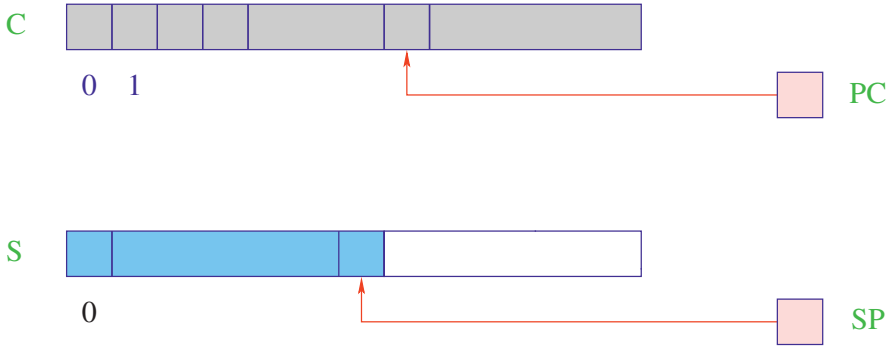


Fig. 2.1. The memory layout of the C-Machine and the registers *PC* and *SP*

execution causes the *PC* to point to the next instruction to be executed. If the current instruction is a jump, it overwrites the contents of the program counter *PC* with its target address.

Thus, the main execution cycle of the C-Machine is given by:

```
while (true) {
     $IR \leftarrow C[PC]; PC++;$ 
    execute (IR);
}
```

At the start of program execution, the register *PC* contains the value 0. Program execution, thus, starts with the execution of the instruction in $C[0]$. The C-Machine stops by executing the instruction **halt**. The execution of this instruction exits the execution cycle and returns control back to the operating system. The return value of the program is given by the contents of some dedicated memory location. Since access to the memory location $S[0]$ is forbidden, we assume here that this is the location with the address 1.

2.3 Simple Expressions and Assignments

In this section, we introduce the translation of arithmetic and logic expressions. Each expression must be translated into an instruction sequence whose execution results in the (current) value of the expression. Consider, for example, the expression $(1 + 7) \cdot (2 + 5)$. What does an instruction sequence that evaluates this expression and pushes its result onto the stack look like?

If the expression consists of just one constant, e.g., 7, this task is easy. Only one instruction is needed that pushes a given value onto the stack. For this purpose, we introduce the instruction **loadc** *q* for any constant *q* (Fig. 2.2).

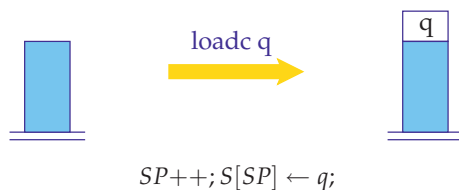


Fig. 2.2. The instruction **loadc** q

The instruction **loadc** q does not require further arguments. It pushes the value of the constant q onto the stack.

Consider an expression which consists of an operator applied to operands. In order to evaluate such an expression, the operands, here $(1 + 7)$ and $(2 + 5)$, are recursively evaluated first, and their values returned on top of the stack. The application of the operator, here \cdot , is then evaluated by consuming these intermediate results from the stack and instead pushing the value of the whole expression onto the stack.

We will frequently encounter translation schemes that proceed by recursion on the structure of a program fragment (here: expressions). This recursion is supported by our design decision that arithmetic, logic, and comparison instructions expect their operands on top of the stack and then replace them with their respective results. In Fig. 2.3, the behavior of these instructions is exemplified by the instruction **mul**.

The instruction **mul** expects two arguments on top of the stack, multiplies them,

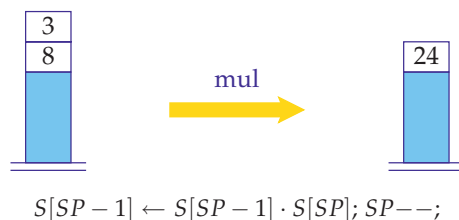


Fig. 2.3. The instruction **mul**

thereby consuming them, and then pushes the result onto the stack. The remaining binary arithmetic and logical instructions **add**, **sub**, **div**, **mod**, **and**, **or**, as well as the comparisons **eq**, **neq**, **le**, **leq**, **gr**, and **geq**, work analogously.

Comparison operators also expect two operands at the top of the stack, compare them, and push the result of the comparison onto the stack. The result is supposed to be a logical value, thus representing *true* or *false*. In the case of C, logical values are represented as integer values: 0 denotes *false*, all other values *true*. Figure 2.4 shows how the instruction **leq** compares two integer values.

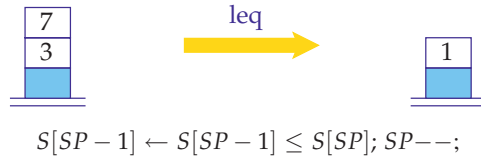


Fig. 2.4. The instruction **leq**

Unary instructions, such as **neg** and **not** consume only one operand. As they also return one value as result, they thus replace the value on top of the stack. As an example we show the instruction **neg** that flips the sign of a number (Fig. 2.5).

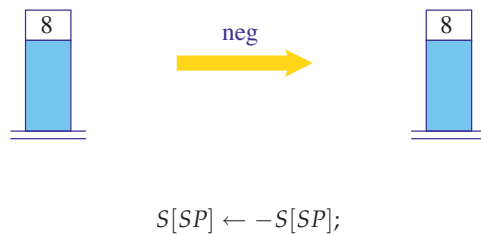


Fig. 2.5. The instruction **neg**

For the expression $1 + 7$, the following instruction sequence is generated:

loadc 1; loadc 7; add

Figure 2.6 shows how this instruction sequence is executed at run-time.



Fig. 2.6. Executing the instruction sequence for $1 + 7$

The instruction sequences to be generated for a statement or an expression are specified by **code**-functions. These functions receive program fragments as argument. They decompose their argument recursively, assemble instruction sequences for each of the components and combine these to an instruction sequence for the whole program fragment. Here, we are not concerned with analyzing the syntax of C programs, that is, with identifying their syntactical structures. Also, we assume that

the input program is *type-correct* and that the *type* of each variable, each function and each subexpression is available.

Program variables are allocated in the main memory S where their values are *stored* in memory locations. The generated code uses the addresses of these locations to load the current values of the variables or to save new values (see Fig. 2.7).

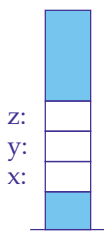


Fig. 2.7. The implementation of variables

Therefore, the **code**-functions require a function ρ which assigns to each variable x its address in main memory. The function ρ is called *address environment*. As we will see later, the address of a variable is in fact a *relative address*, that is, a constant difference between two absolute addresses in S , namely the address of the location of this variable and the initial address of a memory area, which we will later call a *frame* or *stack frame*. Such a frame will contain space for the instances of all variables, parameters, etc., of a function. For the moment, we may assume that $\rho(x)$ is the address of x relative to the beginning of S .

In imperative languages, variables can be used in two ways. Consider for example the assignment $x \leftarrow y + 1$. For variable y , it is the value that is required to determine the value of the expression $y + 1$. The value of variable x , on the other hand, is not important. For variable x , the address of the memory location is required that holds the value of x . The newly computed value needs to be stored in this memory location. We conclude that, when translating assignments, a variable identifier that occurs on the left side of the assignment has to be compiled differently from a variable identifier that occurs on the right side. From the variable identifier on the left, the address of its memory location is required. This value is called *left value* (*L-value*) of the identifier. From the variable identifier that occurs on the right, its value is required, more precisely, the contents of the memory cell associated with the identifier. This value is called the *right value* (*R-value*) of the identifier. Our code functions therefore may be subscripted with **L** or **R**. The function **code_L** generates code for computing the *L-value* while the function **code_R** generates code for computing the *R-value*. The function **code** (without subscript) translates statements, statement sequences, function definitions or whole programs. We may note already here that, while every expression has an *R-value*, not every expression has an *L-value*. A simple example is the expression $y + 1$. The value of this expression only temporarily occurs on top of the stack and therefore may not be accessed via a fixed address.

Figure 2.8 contains the code functions `codeR` and `codeL` for some expressions. The *L*-value of a variable *x* is provided directly by the address environment. To

```

codeR (e1 + e2) ρ = codeR e1 ρ
                      codeR e2 ρ
                      add
                      // analogous for the other binary operators

codeR (-e) ρ         = codeR e ρ
                      neg
                      // analogous for other unary operators

codeR q ρ            = loadc q

codeL x ρ             = loadc ρ(x)

codeR x ρ            = codeL x ρ
                      load

```

Fig. 2.8. The definitions of `codeR` and `codeL` for some expressions

determine the *R*-value of *x*, however, an instruction *s* required which allows us to replace the address of a memory location on top of the stack with its contents. For this purpose, we introduce the instruction **load** (Fig. 2.9).

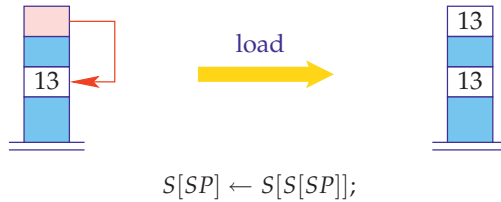
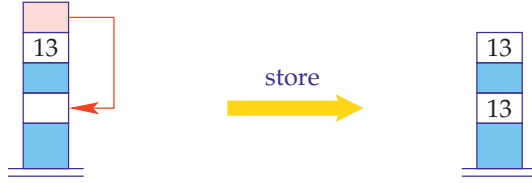


Fig. 2.9. The instruction **load**

In the programming language C, assignments such as $x \leftarrow y + 1$ are *expressions*. The value of this expression is the value of the right side of the assignment. The *R*-value on the left side of the assignment changes, by means of a *side-effect* when evaluating the assignment expression. For the translation of an assignment, an instruction **store** (Fig. 2.10) is required. The instruction **store** expects two arguments on top of the stack: a value *w* and, above it, an address *a*. Then the value *w* is stored into memory at address *a* while the value *w* is returned on top of the stack. In an address environment $\rho = \{x \mapsto 4, y \mapsto 7\}$ the following sequence of instructions computes the *R*-value of $x \leftarrow y + 1$:

loadc 7; load; loadc 1; add; loadc 4; store



$$S[S[SP]] \leftarrow S[SP - 1]; SP--;$$

Fig. 2.10. The instruction **store**

First the value of the right side is computed. Then follows an instruction sequence for computing the L -value of the left side, in our case **loadc** 4. The assignment itself is eventually carried out by the instruction **store**. In general, an assignment is translated as follows:

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

Example 2.3.1 Consider a program with three *int* variables a, b, c . The address environment ρ maps a, b, c onto the addresses 5, 6, and 7, respectively. The translation of the assignment $a \leftarrow (b + (b \cdot c))$ proceeds as follows:

$$\begin{aligned} &\text{code}_R(a \leftarrow (b + (b \cdot c))) \rho \\ &= \text{code}_R(b + (b \cdot c)) \rho; \text{code}_L a \rho; \text{store} \\ &= \text{code}_R b \rho; \text{code}_R(b \cdot c) \rho; \text{add}; \text{code}_L a \rho; \text{store} \\ &= \text{loadc } 6; \text{load}; \text{code}_R(b \cdot c) \rho; \text{add}; \text{code}_L a \rho; \text{store} \\ &= \text{loadc } 6; \text{load}; \text{code}_R b \rho; \text{code}_R c \rho; \text{mul}; \text{add}; \text{code}_L a \rho; \text{store} \\ &= \text{loadc } 6; \text{load}; \text{loadc } 6; \text{load}; \text{code}_R c \rho; \text{mul}; \text{add}; \text{code}_L a \rho; \text{store} \\ &= \text{loadc } 6; \text{load}; \text{loadc } 6; \text{load}; \text{loadc } 7; \text{load}; \text{mul}; \text{add}; \text{loadc } 5; \text{store} \end{aligned}$$

□

In our examples, certain patterns reappear again and again and always lead to similar instruction sequences. The translation often generates instruction sequences that load the value of a constant address (that is, an address known at compile-time) and then uses this address either to load the contents of this memory location or to store a value there. As an optimization, special instructions could be introduced for these tasks:

$$\begin{aligned} \text{loada } q &= \text{loadc } q \\ &\quad \text{load} \\ \text{storea } q &= \text{loadc } q \\ &\quad \text{store} \end{aligned}$$

The new instructions may increase the efficiency of the generated code: on the one hand, the generated code becomes shorter; on the other hand, an implementation of, for example, the instruction **loada** 7 may proceed more efficiently than first creating the constant 7 on the stack, and then overwriting it in the next step with the contents of the memory location with address 7.

2.4 Statements and Statement Sequences

In C, if e is an expression, then $e;$ is a *statement*. Statements do not return values. The value of the Stack Pointer SP must, therefore, be the same before and after the execution of the instruction sequence generated for a statement. Thus, the statement $e;$ is translated as follows:

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

pop

where the instruction **pop** removes the top element of the stack (Fig. 2.11).

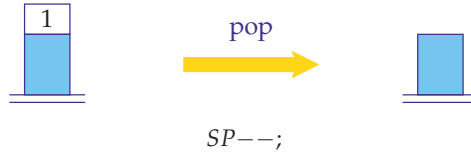


Fig. 2.11. The instruction **pop**

If we are able to generate code for one individual statement, it is easy to also generate code for sequences of statements. For that, the code sequences for the individual statements in the sequence are concatenated:

$$\begin{aligned} \text{code}(s \text{ ss}) \rho &= \text{code } s \rho \\ &\quad \text{code ss } \rho \\ &\quad // \quad s \text{ is a statement, ss is a sequence of statements} \\ \text{code } \varepsilon \rho &= // \quad \text{an empty sequence of statements} \end{aligned}$$

2.5 Conditional and Iterative Statements

Let us now turn to conditional and iterative statements, usually called loops. In the following, we present schemes for the translation of one-sided and two-sided *if* statements:

if (e) s
if (e) s_1 **else** s_2

as well as for *while* and *for* loops:

while (e) s
for ($e_1; e_2; e_3$) s

where e, e_i are expressions and s, s_i are single statements or statement sequences that are enclosed in a block.

For deviating from a linear sequence of execution, suitable *jump* instructions are required. *Unconditional* jumps redirect program execution to a fixed given location. *Conditional* jumps perform the jump only if a certain condition is satisfied. In our case, this condition is satisfied if the top element of the stack equals 0 (Fig. 2.12). Instead of absolute instruction addresses as in our definition, *relative* addresses could

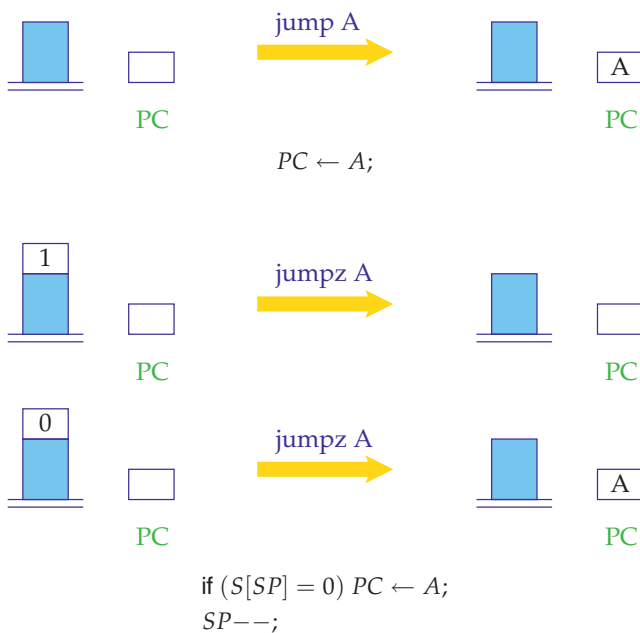


Fig. 2.12. The jump instructions **jump** and **jumpz A**

alternatively be used as jump targets. In this case, the jump targets are given relative

to the code address of the jump instruction. This can be advantageous as smaller addresses would suffice to describe jump targets. Relative addresses also make it easier to *relocate* the code, that is to place it at any location in the code memory.

In order to specify the translation schemes, we find it convenient to introduce *symbolic labels* for instructions that are used as targets in jump instructions. Such a label stands for the address of the instruction that carries the label. In a second pass following code generation, the symbolic labels can be replaced by absolute instruction addresses.

We start with a one-sided conditional statement s of the form **if** (e) s' . Code generation for s places the code for evaluating e and s' consecutively into the program memory and additionally inserts jump instructions such that a correct control flow is guaranteed (Fig. 2.13). In case of one-sided conditionals, a conditional jump must be inserted following the code for evaluating the condition e . This jump branches to the instruction immediately *after* the code for statement s : If at run-time, the condi-

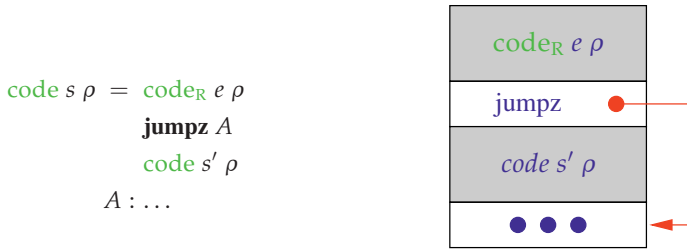


Fig. 2.13. Code generation for one-sided conditional statements

tion e is evaluated to 0, program execution immediately continues after the code for statement s . Otherwise, if the condition e evaluates to a value different from 0, the instruction sequence for the statement s' is executed.

We use the same strategy for the code generation for a two-sided conditional statement s of the form **if** (e) s_1 **else** s_2 : the code sequences for e , s_1 and s_2 are consecutively placed in the program memory. In between suitable jumps are inserted to guarantee a correct control flow (Fig. 2.14). A conditional jump again follows the code for the condition e . The jump target is the beginning of the *else* block s_2 . An *unconditional* jump is inserted immediately after the code for s_1 . Its jump target is the first instruction after the code for statement s . This prevents the code for statement s_2 from being executed following the code for statement s_1 .

Example 2.5.1 Assume that $\rho = \{x \mapsto 4, y \mapsto 7\}$ and consider the conditional statement s of the form:

```

if ( $x > y$ )
     $x \leftarrow x - y$ ;
else  $y \leftarrow y - x$ ;

```

$\text{code } s \rho =$ $\text{code}_R e \rho$
 $\text{jumpz } A$
 $\text{code } s_1 \rho$
 $\text{jump } B$
 $A : \text{code } s_2 \rho$
 $B : \dots$

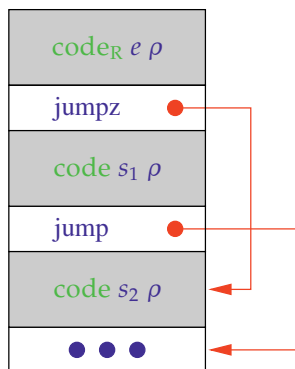


Fig. 2.14. Code generation for two-sided conditional statements

Then $\text{code } s \rho$ results in:

loada 4	loada 4	$A :$ loada 7
loada 7	loada 7	loada 4
gr	sub	sub
jumpz A	storea 4	storea 7
	pop	pop
	jump B	$B : \dots$

□

We now consider a *while* loop s of the form **while** $(e) s'$. The instruction sequence generated for s is shown in Fig. 2.15. A conditional jump to the first instruction after

$\text{code } s \rho = A : \text{code}_R e \rho$
 $\text{jumpz } B$
 $\text{code } s' \rho$
 $\text{jump } A$
 $B : \dots$

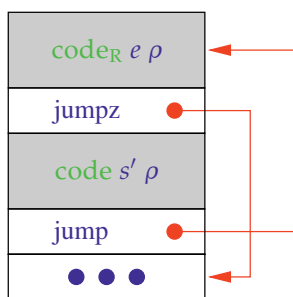


Fig. 2.15. Translation of a *while* loop

the loop is inserted immediately after the code for evaluating the condition e . At the end of the code for the loop body s' an unconditional jump back to the beginning of the code for the condition e is inserted.

Example 2.5.2 Assume that $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ is an address environment and s the statement:

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

Then **code** $s \rho$ produces the sequence:

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

□

Code generation may get more complicated if the body of the loop contains *break* or *continue* statements. A **break** can be interpreted as an unconditional jump to the first instruction following the loop. A **continue**, instead, only jumps to the end of the loop body, which means to the beginning of the code for the condition. Thus for correct code generation, the code function **code** additionally has to maintain the current jump targets for **break** or **continue**. For details we refer to Exercise 6.

A *for* loop s of the form **for** $(e_1; e_2; e_3) s'$ is equivalent to the following sequence of statements:

$e_1; \textbf{while } (e_2) \{s' e_3;\}$

provided that s' does not contain a *continue* statement. In this case, we translate:

code $s \rho = \text{code}_R e_1$
 pop
A : **code**_R $e_2 \rho$
 jumpz B
 code $s' \rho$
 code_R $e_3 \rho$
 pop
 jump A
B : ...

In the presence of *continue* statements in the body s' of the loop, we proceed with the translation in a similar way. We only make sure that every **continue** in s' is interpreted as an unconditional jump to the end of s' , that is, to the first instruction of the code for e_3 (if it exists).

Let us now turn to *switch* statements as provided by the programming language C. This statement is meant to efficiently support indexed jumps depending on the value of a selector expression. For simplicity, we assume that all occurring cases are from a range 0 to $k - 1$ for some constant k . All other values of the selection expression are supposed to jump to the *default* alternative. To simplify matters further, we assume that the cases are arranged in ascending order and each case is terminated with a **break**. Our *switch* statement s , thus, is of the form:

```

switch (e) {
    case 0:  ss0 break;
    case 1:  ss1 break;
    ⋮
    case k - 1:  ssk-1 break;
    default:  ssk
}

```

where ss_i are sequences of statements. *switch* statements can be translated by using *indexed jumps*. An indexed jump is a jump in which the jump target is computed at run-time. For that, we introduce the instruction **jumpi** B , which jumps to the sum of B and the value on top of the stack (Fig. 2.16). Then, the instruction sequence for

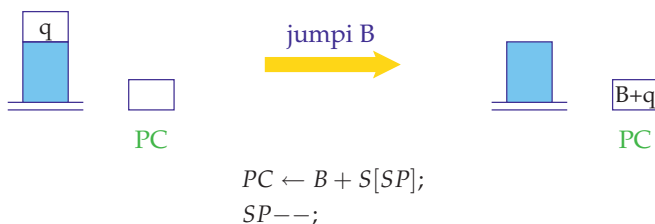


Fig. 2.16. The indexed jump **jumpi** B

the *switch* statement s is given by:

$\text{code } s \rho = \text{code}_R e \rho$ $\text{check } 0 \ k \ B$	$C_0:$ $\text{code } ss_0 \rho$ $\text{jump } D$ \dots $C_k:$ $\text{code } ss_k \rho$ $\text{jump } D$	$B:$ $\text{jump } C_0$ \dots $\text{jump } C_k$ $D:$ \dots
---	---	--

The *macro* $\text{check } 0 \ k \ B$ produces code for checking if the R -value of e lies in the interval $[0, k]$ and for performing an indexed jump. The table of possible jump targets is located at address B . The jump table contains direct jumps to the first

instruction of each alternative. A jump to the first instruction immediately following the *switch* statement is inserted at the end of each alternative. The macro check could be implemented as follows:

check 0 k B	=	dup	dup	jumpi B
		loadc 0	loadc k	A : pop
		geq	leq	loadc k
		jumpz A	jumpz A	jumpi B

The integer value on top of the stack, which is used for the case selection, is needed twice for comparing it with the lower and upper bound of the interval $[0, k - 1]$, respectively. Only if the selection value is from the interval $[0, k - 1]$ can it be used for indexing the jump table. The selection value must be *duplicated* before every comparison since every comparison in our virtual machine consumes the value. For this, we use the instruction **dup** (Fig. 2.17). The implementation of the macro is quite

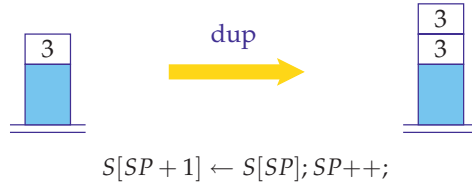


Fig. 2.17. The instruction **dup**

straightforward. For values i in the interval $[0, k - 1]$, the target of the indexed jump is an unconditional jump to the start address of the i -th alternative. In contrast, if the R -value of e is smaller than 0 or larger than k , it is replaced with k before using it for the indexed jump. For this value k the target of the indexed jump is the unconditional jump to the *default* alternative.

In our translation scheme, the jump table is placed at the very end of the instruction sequence for the *switch* statement. As an alternative, it could have been placed directly after the macro **check**. This choice could have saved some unconditional jumps. Code generation may then, however, need further traversals over the statement in order to collect the start addresses of the different cases.

The translation scheme for the simplified *switch* statement can be generalized. If the smallest occurring value is u (instead of 0), the R -value of e is first decremented by u before using it as an index. A strictly ascending order of the selector values is not necessary. Some alternatives may not be terminated by **break**. Also gaps in the interval of possible cases can be tolerated. Then the missing entries of the jump table are filled with unconditional jumps to the beginning of the *default* alternative. Problems only arise if the interval of possible selector values is very large while at the same time few values are actually used. In Exercise 8 approaches can be developed to overcome such difficulties.

2.6 Memory Allocation for Variables of Basic Types

In this section we introduce some important concepts of compiler design, namely the concepts of *compile-time* and *run-time*, *static*, and *dynamic*. At *compile-time*, a given C program is compiled into a CMA program. At *run-time* this compiled C program is executed with input data. *Static* information about a C program is all information about this program that is known at compile-time or that can be computed or derived from other known information. *Dynamic* information is all information that only becomes available when executing the CMA program with the input data.

We have already seen examples of static and dynamic information about C programs. Static information includes, for instance, the target addresses of conditional or unconditional jumps, since they are, after all, computed from the source program with the aid of the `code` function. Obviously, this also holds true for all of the generated CMA program. Thus, the CMA program as a whole is static. In general, the values of variables and, thus, also the values of expressions containing variables, are dynamic. These values may depend on input values of the program, which become available only at run-time. Since the values of the conditions in statements are dynamic, the control flow after evaluating a condition is also dynamic.

Consider a list of variable declarations of a C program of the form:

$$t_1\ x_1; \dots; t_k\ x_k;$$

For the moment, let us assume that all types are basic. According to our assumption about the size of memory locations and the availability of memory, the values for each variable x_i of a basic type **int** or **float**, **char**, as well as enumeration types and pointer variables, can be stored in a single memory location. This means that we do not attempt, as is done in real compilers, to pack several small values into one word. We obtain thus a simple scheme for storage allocation. We assign consecutive addresses to variables in the order of their appearance in the list of declarations of the program. The addresses start at the beginning of the stack. For the moment we consider only programs without blocks and functions. The first assigned address is 1. For reasons that will become clear when we consider functions and blocks, the assigned addresses are called *relative addresses*. The absolute address 1, thus, is interpreted as the address 1 relative to the base address 0.

Let ρ denote a function that maps variables to their respective relative addresses. Our strategy of storage allocation then means that

$$\rho(x_i) = i \text{ for } 1 \leq i \leq k.$$

The relative addresses assigned in this way are static, since they result (in a simple way) from preprocessing the source program, namely, from the positions of the variables within the list of declarations.

These addresses are, of course, located in the memory area that has been reserved for the stack of the C-Machine. When we deal with functions and procedures, it will turn out that we are actually talking about several stacks, nested into each other. One is *large* and contains the data sections of all functions that have been called and not

yet returned and, thus, grows or shrinks when a function is entered or exited, respectively. The others are *small* and *accommodate*, for each not yet terminated function, the intermediate values that are used during the evaluation of expressions. The assignment of memory locations, that is, addresses, to variables defines the structure of the data sections.

2.7 Memory Allocation for Arrays and Structures

The C programming language only provides *static arrays*. Consider the following declaration of an array a :

```
int  $a$  [11];
```

How many memory locations will be occupied by the array a ? Obviously, a consists of the eleven elements:

$$a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10]$$

According to our assumptions, each element occupies one memory location. Thus, the array requires eleven memory locations for its elements. We place these elements consecutively into the stack and record in the address environment for a the start address, that is, the address of the element $a[0]$.

In general, the elements need not always be of basic type, but can themselves be composite. Therefore, we need an auxiliary function for constructing the memory allocation for composite variables. For each type t , this function determines the number of necessary memory locations:

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

This auxiliary function is provided in the C programming language through the library function `sizeof`.

Also in the presence of complex types such as array, we still place the declared variables consecutively in memory. For a sequence d of declarations of the form $t_1 x_1; \dots; t_k x_k$; we define the address environment ρ by:

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

We emphasize that this address environment can be computed at compile-time directly from the declaration d . Accordingly for storing value, say 42, into the element $a[0]$ of the array a , we could use the following sequence of instructions:

```
loadc 42; loadc  $\rho(a)$ ; store; pop
```

where the address $\rho(a)$ is statically known.

Code generation becomes more interesting for the assignment $a[i] \leftarrow 42$; where i is an *int* variable. The variable i gets its value only at run-time. Therefore, instructions must be generated that first compute the current value of i . Then, the corresponding element of the array is selected by adding the required offset to the start address $\rho(a)$:

loadc 42; loadc $\rho(a)$; loadc $\rho(i)$; load; add; store; pop

More generally, let a be an expression that represents an array of elements of type t . For determining the start address of the element $a[e]$, first the start address of the array a is determined. By computing the R -value of e , the index of the selected element is obtained. This index, scaled with the required space for each single element, is added to the start address of the array, in order to obtain the address of the location for the expression $a[e]$. Altogether, the value

$$(L\text{-value of } a) + |t| * (R\text{-value of } e)$$

is computed. In order to generate the corresponding code, we extend the code function `codeL` to indexed array expressions by:

$$\begin{aligned} \text{code}_L a[e] \rho &= \text{code}_L a \rho \\ &\quad \text{code}_R e \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{add} \end{aligned}$$

If the L -value of an indexed array expression $a[e]$ is known, the related R -value can be obtained by loading the contents of the addressed memory location. However, this only works if the elements of the indexed array fit exactly into one memory location. We will soon discuss how composite R -values can be dealt with.

First, we look at the problem of memory allocation and addressing of *aggregates* or *structures*. Let the aggregate variable x be declared by:

struct t {int a ; int b ; } x ;

Then we assign the address of the first available memory location to the variable x , as before. The components of x *relative* obtain addresses relative to the start of the structure, such that $a \mapsto 0$, $b \mapsto 1$. These relative addresses depend only on the type t . We, thus, collect them in the function `offsets`, which assigns the suitable relative addresses to pairs (t, c) of aggregate types and their components.

More generally, let t be an aggregate type of the form **struct** $\{t_1 \ c_1; \dots t_k \ c_k\}$. We define:

$$\begin{aligned} \text{offsets}(t, c_1) &= 0 \quad \text{and} \\ \text{offsets}(t, c_i) &= \text{offsets}(t, c_{i-1}) + |t_{i-1}| \quad \text{for } i > 1 \end{aligned}$$

The size of the aggregate type t is the sum of the sizes of its components:

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

The addressing of aggregate components is analogous to the addressing of elements in an array and consists of the following steps:

1. Load the start address of the aggregate;
2. Increment the address by the relative address of the member.

This produces the final address as a usable L -value. In general, let e be an expression of the aggregate type t , which has a member c . Then, the following code is generated for the computation of the L -value:

```
codeL (e.c) ρ    =   codeL e ρ
                      loadc m
                      add
```

where $m = \text{offsets}(t, c)$. How is the R -value of a member obtained? For components whose type is composed, it is obviously not enough to load the contents of the addressed memory location. Instead, a whole block must be loaded onto the top of the stack. To deal with this situation, we generalize the instruction **load** to instructions **load** m for any non-negative value m . These instructions place at the top of the stack the contents of m consecutive memory locations, starting from the address currently on top of the stack (Fig. 2.18). In particular, the instruction **load** 1 is equivalent

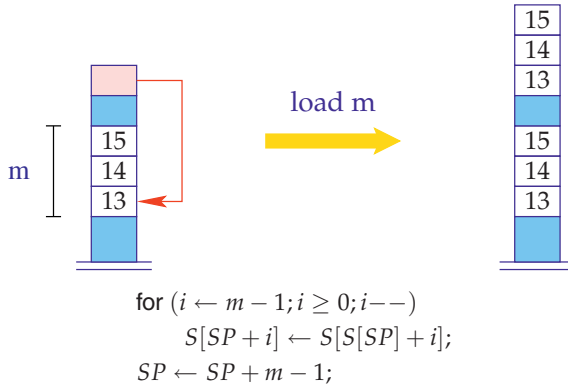


Fig. 2.18. The instruction **load** m

to our previous instruction **load**. Thus, for computing the R -value of an expression e of an aggregate type of size m , we generate code by:

```
codeR (e) ρ    =   codeL e ρ
                      load m
```

On purpose, we have restricted the applicability of this code scheme to expressions of aggregate types. Arrays are also composite types. For historical reasons, the R -value of an array a in C is *not* the sequence of R -values of its elements, but the *start address* of a . The reason is that, according to the C philosophy, the R -value of the array a is considered as a *pointer* to the memory area where the elements of a are located. The R -value of the array a is, thus, of the type $t *$, where t is the type of the elements of a . If e is an expression that represents an array, it follows:

$$\text{code}_R e \rho \quad = \quad \text{code}_L e \rho$$

We have already introduced how to load composed structures. But we need also the ability to store structures. For this, we generalize the instruction **store** to instructions **store m** for any non-negative value m (Fig. 2.19). The general form of the assignment

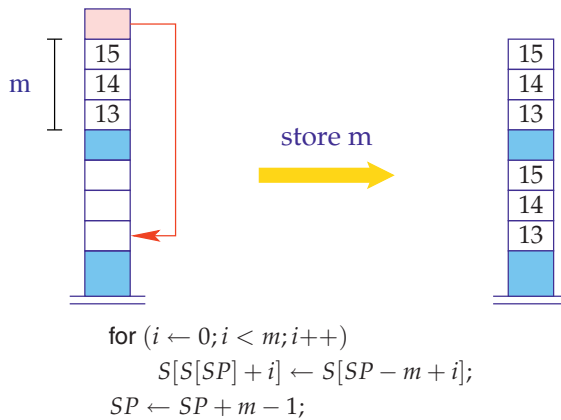


Fig. 2.19. The instruction **store m**

of a value of aggregate type t is, thus:

$$\begin{aligned} \text{code}_R (e_1 \leftarrow e_2) \rho &= \text{code}_R e_2 \rho \\ &\quad \text{code}_L e_1 \rho \\ &\quad \text{store } |t| \end{aligned}$$

Of course, we allow ourselves the abbreviations:

$$\begin{aligned} \text{loada } q \ m &= \text{loadc } q \\ &\quad \text{load } m \\ \text{storea } q \ m &= \text{loadc } q \\ &\quad \text{store } m \end{aligned}$$

As we can no longer assume that an expression always has an R -value of size 1, we must ensure that with a statement e ; all m locations that belong to the value e are removed from the stack. Instead of a single instruction **pop**, we could simply insert m such instructions – or we could allow ourselves a new instruction **pop** m as an optimization. The implementation of this instruction is left to the reader.

2.8 Pointers and Dynamic Memory Allocation

In imperative programming languages, pointers and dynamic memory allocation for anonymous objects are closely related. So far, we have only considered memory allocation for variables that are introduced by declarations. In the declaration, a name is introduced for the object, and a static (relative) address is assigned to this name.

If a programming language supports pointers, these can be used to access objects without names. Linked data structures can be implemented by means of pointers. Their size may vary dynamically, and individual objects are not allocated by a declaration, but via a dynamic call to a memory allocator. The semantics of C are not very precise with respect to the *life-ranges* of dynamically allocated objects. The implementation of a programming language can deallocate the memory occupied by an object already before the end of its life-range without breaking the semantics, as long as it is guaranteed that program execution may no longer access the object. The task of freeing memory that is occupied by unreachable objects is called the *garbage collection*.

As briefly outlined in Sect. 2.1, data areas for the local variables of functions (and their organizational needs) is allocated in the lower end of the data region S when functions are entered and deallocated when functions are exited. This stack-style allocation and deallocation of memory does not match the life-ranges of dynamically allocated objects. Therefore, dynamically allocated objects are placed in a storage area called *heap*, which is located at the upper end of S . The heap grows toward the stack whenever an object is dynamically allocated (Fig. 2.20). A new register of the

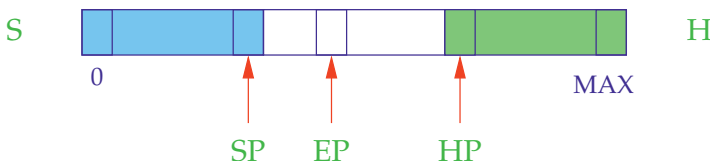


Fig. 2.20. The heap of the C-Machine

C-machine, the *Heap Pointer*, HP , points to the lowest occupied memory location in the heap. New objects on the heap are created by means of the instruction **new** (Fig. 2.21). The instruction **new** interprets the value on top of the stack as the size of the object to be created and replaces it with the start address of the memory block

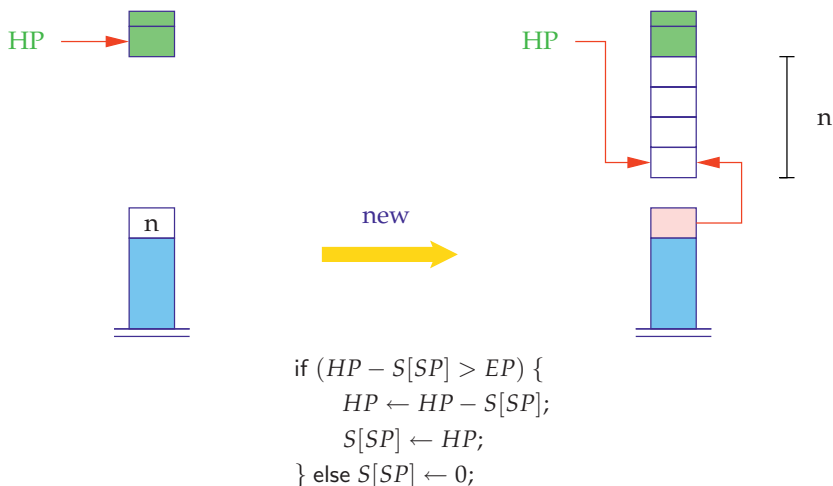


Fig. 2.21. The instruction **new**

of the newly allocated object. Beforehand, we check whether there is still enough free storage space available for the object to be created. If there is not enough space available, the instruction **new** returns the value 0 instead.

For checking whether a collision of stack and heap has occurred, it suffices to compare the registers *HP* and *SP*. In principle, such a comparison must be inserted at each change of the stack height. In order to avoid these comparisons, our C-Machine additionally provides the register *EP*, the Extreme Pointer (Fig. 2.20). This register is supposed to point to the highest stack location that the register *SP* may point to during the evaluation of the current function call. As shown in Exercise 11, the maximum number of stack locations necessary for the evaluation of each expression can be precomputed at compile-time. Therefore, the new value for *EP* can be computed from the *SP* when entering a function. A *stack overflow* is, thus, already detected when entering or when leaving a function call.

Computing with pointer values means to be able

- to *create* pointers, that is, to return references to specific objects in memory; as well as
- to *dereference* pointers, that is, to access memory locations via pointers.

Technically, a C pointer is nothing but a memory address. In C, there are two ways to produce pointers: with a call of the library function **malloc** or through the use of the address operator **&**. A call **malloc**(*e*) for an expression *e* computes the *R*-value *m* of *e* and returns the start address of a new memory region of size *m*. By means of the instruction **new**, we translate:

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

new

Note that a call to the **malloc** function never fails. If there is not enough space available for the new object, the call still returns an address value, namely the value 0. A thorough programmer will, therefore, always check the return value of a call to **malloc** against 0, detect and handle this error situation correctly.

The address operator **&** when applied to the expression e , returns a pointer to the storage object of e , that is, to the object of the type of e that is located at the start address of e . Thus, the R -value of the expression $\&e$ equals the L -value of the expression e :

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

Assume that e is an expression that evaluates to the pointer value p . This pointer value is the address of a memory object o . The memory object o can be accessed by *dereferencing* the pointer p , that is, by applying the prefix operator $*$. Since the R -value of e represents the L -value of $*e$, we define:

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

An important feature of the programming language C is that it supports *pointer arithmetic*. That means that an *int* value a can be added to or subtracted from a pointer p . Pointer arithmetic is meant to support traversing sequences of similar memory objects by means of a pointer. If the pointer p points to a value of type t , then the expression $p + a$ represents a pointer to the a -th next memory object. This means for a sum $e_1 + e_2$ where e_1 is of type $t *$ and e_2 is of type **int**, the R -value of e_2 must first be scaled with $|t|$, before it can be added to the R -value of e_1 . An analogous implicit scaling takes place at subtractions. Thus, we define in this case:

$$\begin{aligned} \text{code}_R (e_1 + e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{add} \\ \text{code}_R (e_1 - e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{sub} \end{aligned}$$

As an application, we obtain a translation scheme for the expression $e_1[e_2]$ for an expression e_1 of type $t *$ and an *int* expression e_2 . Since the indexed pointer expression is an abbreviation for the expression $*(e_1 + e_2)$, we obtain:

$$\begin{aligned}
\text{code}_L e_1[e_2] \rho &= \text{code}_L (* (e_1 + e_2)) \rho \\
&= \text{code}_R (e_1 + e_2) \rho \\
&= \text{code}_R e_1 \rho \\
&\quad \text{code}_R e_2 \rho \\
&\quad \text{loadc } |t| \\
&\quad \text{mul} \\
&\quad \text{add}
\end{aligned}$$

It is worth noting that this scheme is consistent with our scheme for array indexing. If the expression e_1 defines an array, the L -value of e_1 would have been used instead of the R -value as in the case of references. Since for arrays, the L -value and the R -value agree, our compilation scheme can be used both for indexed accesses for references and for arrays.

At the end of this section, we consider a slightly larger example where several translation schemes are applied.

Example 2.8.1 For a declaration:

```

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

```

the expression $e \equiv ((pt \rightarrow b) \rightarrow a)[i + 1]$ is to be translated. Here, the operator \rightarrow is an abbreviation for a dereference followed by a selection. This means:

$$e \rightarrow c \equiv (*e).c$$

If e has type $t *$ for a structure t with a member c , we therefore obtain for $m = \text{offsets}(t, c)$:

$$\begin{aligned}
\text{code}_L (e \rightarrow c) \rho &= \text{code}_L ((*e).c) \rho \\
&= \text{code}_L (*e) \rho \\
&\quad \text{loadc } m \\
&\quad \text{add} \\
&= \text{code}_R e \rho \\
&\quad \text{loadc } m \\
&\quad \text{add}
\end{aligned}$$

In our example, we have $\text{offsets}(t, a) = 0$, and $\text{offsets}(t, b) = 7$. Let us assume that we are given an address environment

$$\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3\}$$

Then, the resulting code for the expression e is:

$$\begin{array}{llll}
\text{code}_L e \rho & = & \text{code}_L ((pt \rightarrow b) \rightarrow a) \rho & = & \text{code}_L ((pt \rightarrow b) \rightarrow a) \rho \\
& & \text{code}_R (i + 1) \rho & & \text{loada } 1 \\
& & \text{loadc } 1 & & \text{loadc } 1 \\
& & \text{mul} & & \text{add} \\
& & \text{add} & & \text{loadc } 1 \\
& & & & \text{mul} \\
& & & & \text{add}
\end{array}$$

where:

$$\begin{array}{llll}
\text{code}_L ((pt \rightarrow b) \rightarrow a) \rho & = & \text{code}_R (pt \rightarrow b) \rho & = & \text{loada } 3 \\
& & \text{loadc } 0 & & \text{loadc } 7 \\
& & \text{add} & & \text{add} \\
& & & & \text{load} \\
& & & & \text{loadc } 0 \\
& & & & \text{add}
\end{array}$$

Altogether we obtain the sequence:

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

This sequence could not as easily be derived by hand without systematically applying the translation schemes. We also observe that our schemes still leave room for various optimizations. For instance, additions of 0 could have been saved as well as multiplications by 1. These inefficiencies could have been avoided directly during code generation, by introducing appropriate compilation schemes for special cases. To keep the schemes clear, we did not follow this option. Instead, we rely on a post-pass optimizer to carry out these local code improvements separately. \square

The translation schemes for pointer expressions are now complete. What remains open is the question of how to implement the explicit *release* of memory blocks. Releasing the memory block pointed to by a pointer is problematic, because other pointers still might point into this memory block. Such pointers would after the release be called *dangling*.

Even if we assume that the programmer always knows what she is doing, the heap could become *fragmented* after several releases, as shown in Fig. 2.22. The free memory regions can be scattered quite unevenly over the heap. There are various techniques for reusing these regions during program execution. In any case, further data structures are required by the run-time system for this purpose. Their administration increases the cost of calls to the functions **malloc** or **free**.

In our minimal compiler, we follow a different strategy: we ignore releases of memory. This implementation strategy is obviously correct. If it is not necessarily

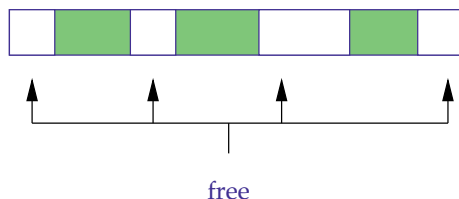


Fig. 2.22. The heap after the release of several memory blocks

optimal w.r.t. consumption of memory space, it is at least simple. Accordingly, we translate:

$$\text{code}(\text{free}(e);) \rho = \text{code}_R e \rho \text{ pop}$$

2.9 Functions

As a preparation for the translation of functions, we briefly recall the related concepts, terms, and problems. The *declaration* of a function consists of:

- a name by which it can be called,
- the specification of the type of the return value,
- the specification of the formal parameters, which, together with the type for the return value, form the input and output interface,
- a function *body*, which consists of a sequence of local declarations and statements.

If the function does not return a value, that is, if it is a *procedure*, the type of the return value is specified as **void**.

A function is *called*, or *invoked*, when it is applied to actual parameter values within the statement part of another function. A called function may itself call other functions, including itself. When a called function f has executed all its statements, it *returns*, and the caller, that is the function that invoked f , continues execution immediately after the call.

All function calls that occur during the execution of a program can be organized into an ordered tree, the *call tree* of the program execution. Each node in the call tree is labeled with a function name. The root of the call tree is labeled with the name of the function *main*, whose call starts program execution. If a node is labeled with a function name f , the sequence of its direct successors consists of the functions successively invoked when executing the body of f . The label f may occur multiple times in the call tree; to be precise, it occurs as often as f is called during this execution of the program. We call each occurrence of f an *incarnation* of f . An incarnation is characterized by the path from the root of the tree to the node. We call this path the *incarnation path* of an incarnation of f .

Consider the state of program execution when a certain incarnation of f is active. All predecessors of this incarnation, that is, all nodes on the incarnation path, have already been called, but have not yet returned. We say that all these incarnations are at this moment *live*.

Example 2.9.1 Consider the C program:

```

int n;

int fac(int n) {
    if (n ≤ 0) return 1;
    else return n * fac(n - 1);
}

int main() {
    int r;
    n ← 2;
    r ← fac(n) + fac(n - 1);
    return r;
}

```

Figure 2.23 shows the call tree for this program. The arrow points to the second

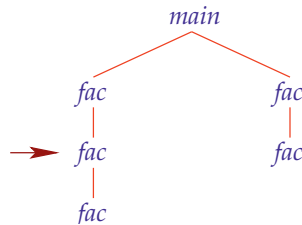


Fig. 2.23. The call tree for the example program

recursive call of the function fac . At this moment the main function $main$ and two recursive calls of fac are live. □

Programs may have different program executions since they depend on input. Accordingly, there can be more than one call tree for a given program p . Since the number of different executions of p may be infinite, there can also be infinitely many different call trees for p .

A name can occur multiple times in a program. A *defining occurrence* of a name is an occurrence in a declaration or in a formal parameter list where it is defined. All other occurrences are called *applied occurrences*.

Consider the names occurring in functions. The names that are introduced as formal parameters or via local declarations are called *local*. When a function is called, new *incarnations* are created for all local names. Space is allocated for each variable according to the types specified in the declaration and (if necessary) filled with an initial value. In particular, the formal parameters receive the values of the actual parameters. The *life-range* of the created variable incarnation is equal to the life-range of the incarnation of the function. Therefore, the space occupied by locals can be released when returning from the function.¹ This behavior can be realized with

¹ These should be distinguished from *static* variables which are declared local to the given function f , but global to any incarnation of f .

a stack-style memory management. Accordingly, the stack frame that was allocated for formal parameters, for locally declared variables, and for intermediate values (see Fig. 2.24) when entering the function is released when returning from the function.

Dealing with applied occurrences of names that are non-local is not as straightforward. These names are *global* with respect to the given function or block. The *visibility* and/or *scope rules* of the programming language determine how to find, for a given applied occurrence of a name, the corresponding defining occurrence. The dual, but equivalent, perspective is to identify for a given defining occurrence of a name x , all program locations where applied occurrences of x refer to the given defining occurrence.

From languages like ALGOL we know the following visibility rule: a defining occurrence of a name is visible in the *program unit* directly containing the declaration or specification minus all contained program units that introduce a new definition of this name. Here, *program unit* stands for a function or a *block*. Blocks are considered in Exercise 12.

Based on the given visibility rule, we reconsider the problem of establishing a relationship between defining and applied occurrences. When searching for the defining occurrence of a given applied occurrence of a name x , we start in the declaration part of the immediate program unit wherein the applied occurrence of x occurs. If no defining occurrence of x is found there, we continue with the enclosing program unit, and so forth. If no defining occurrence can be found in all enclosing program units, including the whole program, then a programming error is encountered.

The dual approach is to start from a defining occurrence of x and then search the corresponding program unit for all occurring applied occurrences of x . This scan is blocked for program units that introduce new definitions of x .

Example 2.9.2 Consider the program from Example 2.9.1. The variable n is defined outside all functions and therefore is global with respect to function *main*. As the formal parameter of function *fac* is also named n , this global variable is not visible inside the function body. The applied occurrences of n within the body of *fac* therefore refer to the formal parameter of *fac*. □

The given visibility rule corresponds to *static* scoping. Static scoping means that applied occurrences of names that are global to a program unit refer to the defining occurrences occurring in *textually* enclosing program units. The corresponding binding of names is static as it depends only on the program itself and not on the (dynamic) execution of the program. Every use of the global name x at run-time refers to the same incarnation of the statically bound defining occurrence of x .

In contrast, *dynamic* scoping means that an access to a global name is bound to the last created incarnation of this name – regardless of the function in which the name is defined. Static scoping is standard for all ALGOL-like languages and also for modern functional languages, such as HASKELL and OCAML, while older dialects of LISP use dynamic scoping.

Example 2.9.3 Consider the following program:

```

int  $x \leftarrow 1$ ;
void  $q()$  {
     $\text{printf}(\text{"\%d", } x);$ 
}

int  $\text{main}()$  {
    int  $x \leftarrow 2$ ;
     $q()$ ;
}

```

With static scoping, the applied occurrence of the variable x in function q refers to the global variable x . Therefore, the value 1 is printed. In contrast, the dynamically last defining occurrence of the variable x before the call to function q is the one in the function main . With dynamic scoping, the value of the applied occurrence of x that is printed in q is 2. \square

In contrast to PASCAL, the programming language ANSI-C does not allow nested function definitions. This design decision greatly simplifies the treatment of visibility. For ANSI-C, it suffices to distinguish between two kinds of variables: *global* variables, which are declared outside of function definitions, and *local* or (in C jargon) *automatic* variables, which are defined local to particular functions.

2.9.1 Memory Organization of the C-Machine

In the following, we describe the organization of the memory area for maintaining the sequence of live incarnations of functions. This part of S is the *run-time stack* of the CMA. The run-time stack consists of a sequence of *stack frames*, one for each live incarnation of a function, in the same order in which these appear in the incarnation path. In order to efficiently support returns from functions, the stack frames are linked together. Each stack frame for the incarnation of a function f contains a reference to the stack frame of the incarnation of the function g that invoked f . This incarnation of g is called the *dynamic predecessor* of f .

The stack frame corresponding to an incarnation of f is allocated when f is called. The organization of a stack frame of the C-Machine is shown in Fig. 2.24. We introduce a further register, the *Frame Pointer*, FP , which points to a well-defined place in the current stack frame. In the stack frame, further space is allocated for storing registers whose contents must be saved when entering a function and restored when returning from the call. In the C-Machine these are the registers PC , FP , and EP . The saved value of PC is the *return address* from where program execution is supposed to resume when the function call is completed. The saved value of FP is the link to the stack frame of the calling function, that is the *dynamic predecessor* of the current function call. Finally, the value of EP must be saved because it is only valid for the current call.

The locations where these three register are saved are called the *organizational cells*, because they assist in organizing the start and end of functions correctly and efficiently. The organizational cells form a section within the stack frame of the current function incarnation. In this stack frame, we allocate the formal parameters and local variables of the current function. This allows us to access these variables relative to the Frame Pointer FP with *fixed relative addresses*. If the Frame Pointer points to

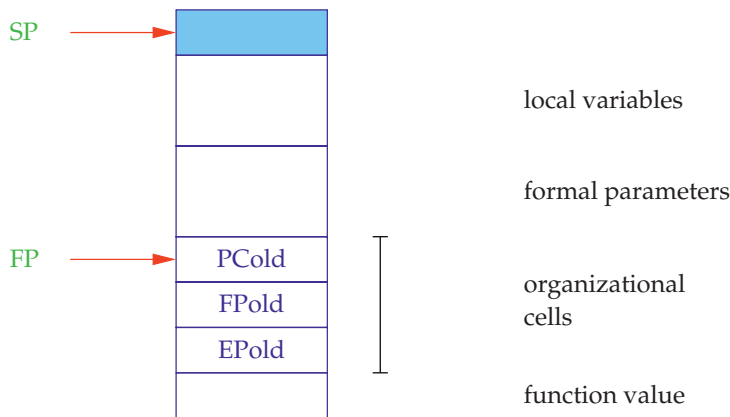


Fig. 2.24. A stack frame of the C Machine

the top organizational cell (that is, the saved *PC*), positive relative addresses, starting with 1, can be assigned to the local variables.

Above the data section, we allocate the *local stack*, that is, the stack that we introduced in Sect. 2.2 for the evaluation of expressions. The maximal height of this local stack can also be statically determined (see Exercise 11).

Conceptually, two more things must be allocated in the stack frame, the actual parameters and the return value of the function. With the actual parameters we have a problem: the programming language C allows the definition of functions with *variable-length* parameter lists, such as the function *printf*, whose first parameter is mandatory, while the number of further actual parameters only becomes evident from the call. Within such a function, code generation may only assume that the mandatory parameters are available. In order to assign fixed relative addresses also for these parameters, we use a trick: the actual parameters are placed on the stack *below* the organizational cells and in *reverse order*! In this way, the first parameter is placed on top of the second and so on. Relative addresses therefore are now *negative numbers* starting from -3 . If the first parameter, for example, has size m , then it receives the relative address $-(m - 2)$.

If the function returns a value, we should reserve a canonical place for it where it can be accessed with a fixed address relative to *FP*. We could introduce a separate section of the stack for the return value. After the return from the function call, though, the space for the actual parameters is no longer needed. Therefore, we choose to reuse this section for storing the return value.

2.9.2 Dealing with Local Variables

In Sects. 2.6 and 2.7, we described how to assign memory locations, that is, addresses, to names defined in declaration lists. There, we only considered programs with one list of declarations and one list of statements. In the absence of function definitions, it was not necessary to distinguish between global and local variables. In fact, we used *absolute addressing* for accessing variables, that is, variables were accessed relative to the start address of the memory area S .

An address environment assigns (relative) addresses to names. For a real C program, a name may also identify a function. The address assigned to a function f should be the (absolute) start address of the *code* for f in the program store C .

With formal parameters and local variables, on the other hand, we wish to access their incarnations in the current function call. The addressing in this case is relative to the Frame Pointer FP . In order to distinguish these different modes of addressing, we extend the address environment ρ in such a way that ρ maintains, for each defined occurrence of a name, not only a relative address, but also the information of whether the name is global or local. The address environment ρ , thus, has now the functionality:

$$\rho : Names \rightarrow \{G, L\} \times \mathbb{Z}$$

where the tags G and L identify global or local scope, respectively. To allow access relative to the FP for local variables or formal parameters, it is enough to generalize the code function code_L for names. For $\rho(x) = (tag, j)$ we now define:

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

The new instructions **loadrc** j push the value $FP + j$ on top of the stack (Fig. 2.25). As an optimization we again introduce special instructions for frequently occurring

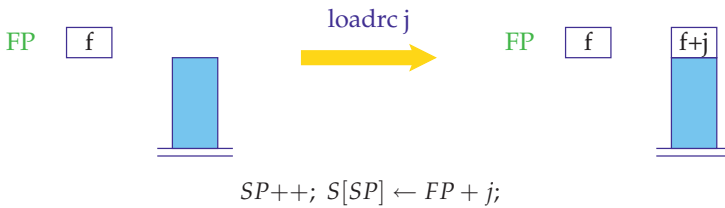


Fig. 2.25. The instruction **loadrc** j

instruction sequences:

$$\begin{aligned} \text{loadr } j \ m &= \text{loadrc } j \\ &\quad \text{load } m \\ \text{storer } j \ m &= \text{loadrc } j \\ &\quad \text{store } m \end{aligned}$$

where we also write **loadr** j and **storer** j for **loadr** j 1 and **storer** j 1, respectively.

With this change, we can apply the translation schemes, which we developed step by step in the last sections, also to the bodies of functions. We only must additionally supply the address environment ρ_f at the entry point of a function f . In particular, we must ensure that the address environment provides the correct bindings for the visible names whenever it is used.

For processing a global variable declaration, we define a function `elab_global`. This function takes a pair (ρ, n) of an address environment ρ and a first available relative address n , together with a declaration $d \equiv t\ x$, and produces an extended address environment together with the next available relative address. We define:

$$\text{elab_global } (\rho, n) (d) = (\rho \oplus \{x \mapsto (G, n)\}, n + |t|)$$

The expression $\rho \oplus \{x \mapsto a\}$ denotes the (partial) function obtained from ρ by adding for the argument x the value a . If ρ is already defined for x , then the old value in ρ for x is overwritten with the new value a .

Analogously, we define functions `elab_formal` and `elab_local` for processing declarations of formal parameters and local variables, respectively:

$$\begin{aligned} \text{elab_formal } (\rho, z) (t\ x) &= (\rho \oplus \{x \mapsto (L, z - |t|)\}, z - |t|) \\ \text{elab_local } (\rho, n) (t\ x) &= (\rho \oplus \{x \mapsto (L, n)\}, n + |t|) \end{aligned}$$

The function `elab_local` is analogous to the function `elab_global` for processing declarations of global variables – the one difference is that now, instead of the tag G , the tag L is assigned. In contrast, we must be careful when defining the function `elab_formal`. Every further parameter must receive a *smaller* address. Instead of the first available relative address, we assign to the next parameter the lowest location z occupied so far by the stack frame, minus the size of the type of the variable.

By using these functions repeatedly, we can process lists of global variables, formal parameters, and local variables, respectively:

$$\begin{aligned} \text{elab_globals}(\rho, n) () &= (\rho, n) \\ \text{elab_globals}(\rho, n) (t\ x; ll) &= \text{elab_globals} (\text{elab_global } (\rho, n) (t\ x)) (ll) \\ \text{elab_formals}(\rho, z) () &= (\rho, z) \\ \text{elab_formals}(\rho, z) (t\ x, dd) &= \text{elab_formals} (\text{elab_formal } (\rho, z) (t\ x)) (dd) \\ \text{elab_locals}(\rho, n) () &= (\rho, n) \\ \text{elab_locals}(\rho, n) (t\ x; ll) &= \text{elab_locals} (\text{elab_local } (\rho, n) (t\ x)) (ll) \end{aligned}$$

Assume that we are given a function f without return value with lists dd and ll of declarations of formal parameters and local variables, respectively. From an address environment ρ for global names we obtain, thus, as address environment ρ_f for the function f :

$$\begin{aligned}
\rho_f = & \text{ let } \rho = \rho \oplus \{f \mapsto (G, _f)\} \\
& \text{ in let } (\rho, _) = \text{elab_formals } (\rho, -2) \text{ dd} \\
& \text{ in let } (\rho, _) = \text{elab_locals } (\rho, 1) \text{ ll} \\
& \text{ in } \rho
\end{aligned}$$

where $_f$ identifies the start address of the code for f .

If the function f has a return value, we also remember in the address environment ρ_f for f the relative address in the stack frame where the return value is stored. For this we introduce a local auxiliary variable ret . Let t be the type of the return value and m be the space requirement of the mandatory formal parameters.

- We can place the return type at the lower boundary of the section for the formals, that is, from relative address $-(m+2)$ onwards if $|t| \leq m$. For this, we expand our definition of ρ_f with the binding $\text{ret} \mapsto (L, -(m+2))$.
- A larger section is required for the return value than for the mandatory parameters if $|t| > m$. In this case, the return value is allocated from address $-(|t|+2)$ onwards. Thus, we add to ρ_f the binding $\text{ret} \mapsto (L, -(|t|+2))$.

Example 2.9.4 Consider again the C program of Example 2.9.1. As global names, we have n for a global variable together with fac and $main$ for functions. Then

$$\rho_0 = \{n \mapsto (G, 1)\}$$

is the address environment of global names known when processing the declarations of the function fac . From this address environment, we obtain the address environment ρ_{fac} within fac , by first adding the binding $fac \mapsto (G, _fac)$ to ρ_0 . Then, we record the binding $n \mapsto (L, 3)$ for the formal parameter n , which replaces the binding for the corresponding global variable. Finally, we add the relative address of the return value, which is -3 . Thus, we obtain:

$$\rho_{fac} = \{fac \mapsto (G, _fac), n \mapsto (L, 3), \text{ret} \mapsto (L, -3)\}$$

The name of the function fac is known already before the definition of the function $main$. Thus,

$$\rho_1 = \{n \mapsto (G, 1), fac \mapsto (G, _fac)\}$$

With respect to this environment, the address environment ρ_{main} inside the function $main$ is given by:

$$\begin{aligned}
\rho_{main} = & \{fac \mapsto (G, _fac), main \mapsto (G, _main), \\
& n \mapsto (G, 1), r \mapsto (L, 1), \text{ret} \mapsto (L, -3)\}
\end{aligned}$$

Even though the function $main$ does not have formal parameters, it does have a return value of size 1. As in the function fac , this value receives the relative address -3 . In contrast to the function fac , the global variable n is not hidden in function $main$ by a formal parameter with the same name. Instead, ρ_{main} contains a binding for the additional local variable r as well as a binding for the global function name $main$.

□

2.9.3 Function Call and Return

Let us discuss the two crucial problems with the implementation of C functions, the call and, thus, starting the execution of a function and the return, that is, exiting after having processed a function's body.

First we examine the call of a function. Let f be the currently active function. Its stack frame is at the top of the stack. Now assume that function f calls a function g . An instruction sequence must be generated for the call of g that processes the call of g and leaves the return value on top of the stack. Note that this instruction sequence computes an R -value (as long as the function does not return **void**). In particular, this return value has no (sensible) L -value. This means that according to our translation schemes the return value cannot be directly accessed by a selector. One solution to this problem is to apply a program transformation before code generation that places the return values of all problematic function calls in auxiliary local variables. As an example, the assignment $x \leftarrow f(y + 2).a$; is transformed into the block:

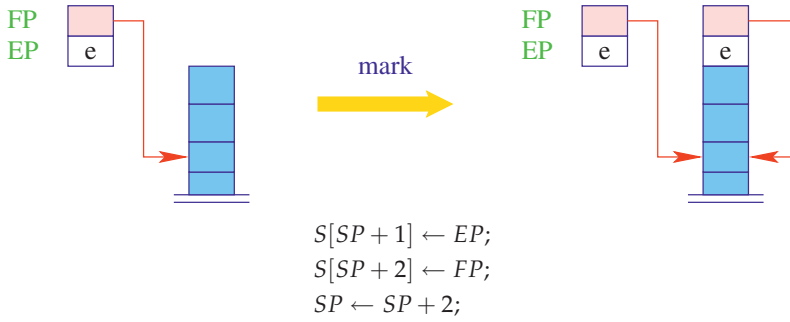
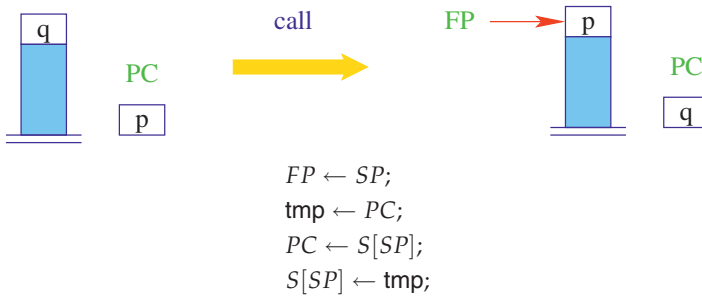
$$\{ \textbf{struct } t \text{ tmp}; \text{ tmp} \leftarrow f(y + 2); x \leftarrow \text{tmp}.a; \}$$

for a new variable tmp , where t is the return value of function f .

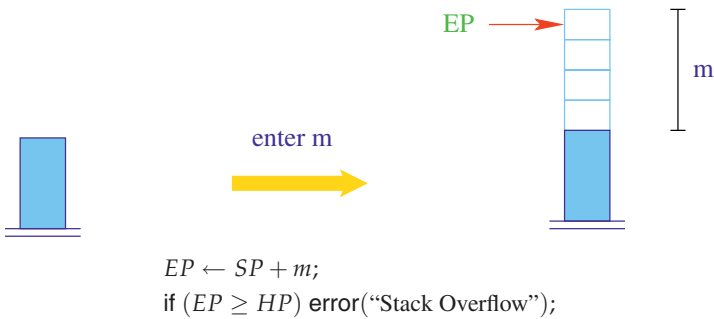
The following sequence of actions must be executed for starting the execution of the function g :

1. The values of the actual parameters must be computed and pushed onto the stack.
2. The old values of registers EP and FP must be pushed onto the stack.
3. The start address of function g must be computed.
4. The return address must be computed and recorded in the corresponding organizational cell.
5. Register FP must be made to point to the top of the new stack frame.
6. Control must proceed to the start address of g .
7. Register EP for the new call must be set to the current value of g .
8. Space must be reserved on the stack for the local variables of g .

Then the sequence of instructions for the body of function g can be executed. When designing translation schemes, we must distribute the listed actions among the caller f and the callee g . Such a distribution must take the respective knowledge into account. It is, for example, only the caller f that can provide the values of the actual parameters, while only function g knows the space requirement for its local variables. In our list, the borderline between caller code and callee code lies between points (6) and (7). For saving the registers EP and FP in point (2), we introduce the instruction **mark** (Fig. 2.26). This instruction pushes the contents of both registers consecutively onto the stack. For setting the new FP , saving the return address, and jumping to the code of the callee in points (6), (4) and (5), we introduce the instruction **call** (Fig. 2.27). The instruction **call** is the last instruction of the call sequence of g in the caller f . When executing this instruction, the PC register corresponds exactly to the return address! This means that the actions (4) and (5) are jointly implemented by swapping the contents of the topmost stack cell with the contents of register PS .

Fig. 2.26. The instruction **mark**Fig. 2.27. The instruction **call**

Only actions (7) and (8) remain. We set the new *EP* relative to the current *SP* with the help of an instruction **enter** *m* where *m* is the total number of stack locations required inside the called function (Fig. 2.28). This instruction checks whether there

Fig. 2.28. The instruction **enter** *m*

is enough space on the stack for executing the current call. If this is not the case, program execution is aborted with an error message.

Finally, allocating m memory locations for the local variables is implemented by incrementing the register SP with m . This increment is realized by the instruction **alloc** m (Fig. 2.29). In particular, **alloc** 0 has no effect.

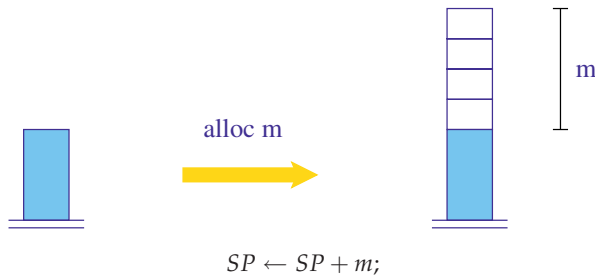


Fig. 2.29. The instruction **alloc** m

The instruction *alloc* m can also be used for allocating stack space for the return value of the function, in case that the space for the actual parameters is not sufficient. We now provide a translation scheme for a function call e of the form $g(e_1, \dots, e_n)$. Let t be the return type of g , m the space requirement of the formal parameters, and $m' \geq m$ the space requirement of the actual parameters. Then we define:

```

codeR  $e \rho =$  alloc  $q$ 
               codeR  $e_n \rho$ 
               ...
               codeR  $e_1 \rho$ 
               mark
               codeR  $g \rho$ 
               slide  $q' |t|$     where
                $q = \max(|t| - m', 0)$ 
                $q' = \text{if } (|t| \leq m) \ m' - m$ 
                    $\text{else } \max(m' - |t|, 0)$ 

```

First, enough space is allocated to accommodate the return value below the organizational cells. If the return value is not located at the lower boundary of the stack frame, it must be moved down after the call. This is realized by the instruction **slide** q m (Fig. 2.30).

According to our translation scheme, the first argument of the second *slide* instruction is 0, whenever the function has no optional parameters. One can show that it is never necessary to simultaneously move results after and allocate additional stack space before the call.

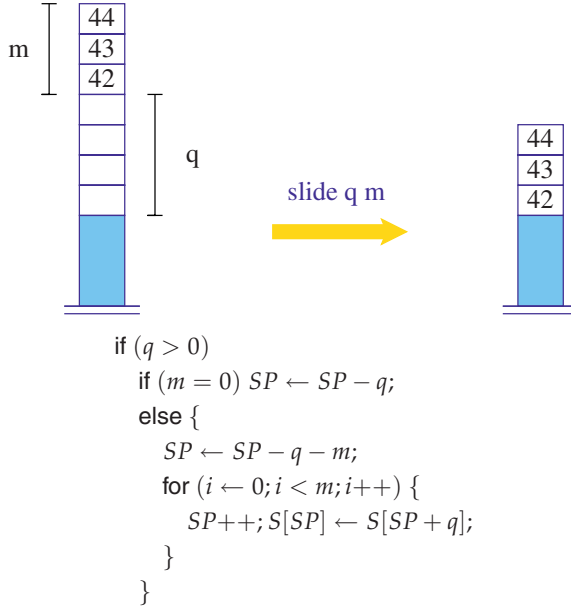


Fig. 2.30. The instruction **slide** $q\ m$

Our scheme generates code for each actual parameter e_i which computes the R -value of e_i in the address environment ρ . This is consistent with passing parameters to the function *by value*. Other imperative programming languages, such as PASCAL or C++, also support parameter passing *by reference*. If the formal parameter x is a reference parameter, the L -value of the actual parameter is passed to x , instead of the R -value. Each access to the formal parameter x in the body of the function requires an *indirection* through the relative address of x . For a reference parameter x , we therefore modify the computation of the L -value of x to:

$$\text{code}_L x \rho = \text{loadr } \rho(x)$$

For the expression g which evaluates to the function to be called, code is generated for computing the R -value. This scheme allows us to call functions through pointers. In C, a function name is considered as a *reference* whose R -value equals its L -value:

$$\text{code}_R f \rho = \text{code}_L f \rho = \text{loadc } \rho(f)$$

Example 2.9.5 Consider the recursive function call $\text{fac}(n - 1)$ in the program of Example 2.9.1. Then, our translation scheme generates the instruction sequence:

alloc 0; loadr -3; loadc 1; sub; mark; loadc $_fac$; call; slide 0 1;

□

After having thoroughly examined the actions taken when entering a function, we now turn to the actions that are executed when exiting a function. These are:

1. (possibly) saving the return value,
2. recovering the registers *EP* and *FP*,
3. cleaning up the stack and jumping back to the code of the caller.

These actions can be executed entirely by the called function. Since we manage the start address of the return value – if it exists – in the address environment, saving the return value can be treated as an assignment. The other two tasks can be merged into the instruction **return** *q* (Fig. 2.31). Here, the constant *q* equals the number of

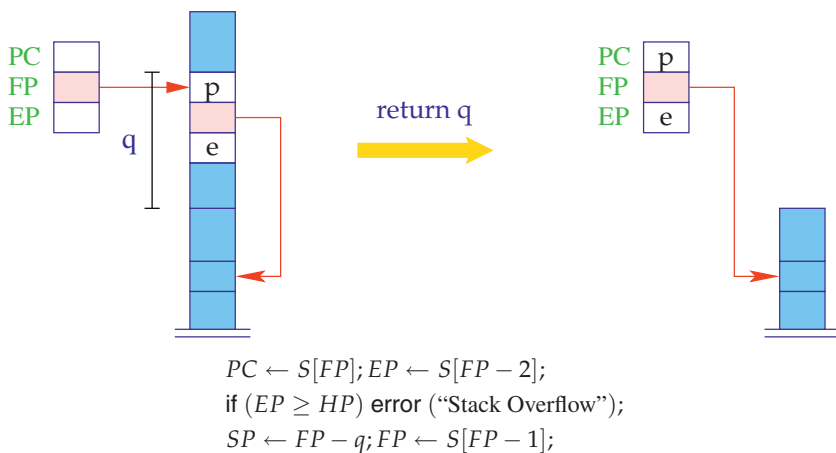


Fig. 2.31. The instruction **return** *q*

cells above the return value that are supposed to be removed. A test for a collision between stack and heap must be performed before restoring the *EP* since the heap pointer might have been decreased during the function call. Program execution is aborted with an error message if the value of the *EP*, which is to be restored, is not smaller than the current value of *HP*.

Accordingly, the C statements **return**; and **return** *e*; for an expression *e* of type *t* are translated as follows:

$\text{code}(\text{return};) \rho = \text{return } (m + 3)$
 $\text{code}(\text{return } e;) \rho = \text{code}_R e \rho$
 $\text{storer } \rho(\text{ret}) |t|$
 $\text{return } q \quad \text{where}$
 $q = 3 + \max\{m - |t|, 0\}$

where m is the space requirement of the function for all its formal parameters. Overall, the definition d of a function f of the form:

$$t \ f \ (params) \ \{ locals \ ss \}$$

with return type t , declaration of formal parameters $params$, declaration of local variables $locals$, and statements sequence ss , is translated as follows:

```
code  $d \ \rho =$    $\_f :$   enter  $k$ 
                  alloc  $l$ 
                  code  $ss \ \rho_f$ 
                  return  $q$ 
```

where k is the maximal space requirement for a call to function f , l is the number of memory locations for the local variables, ρ_f is the address environment for f , which is obtained from ρ along with $params$, $locals$ as well as the size of the return type t , and $q = \max\{m - |t|, 0\}$, if m is the space requirement for the mandatory formal parameters.

Example 2.9.6 The definition of the function fac of Example 2.9.1 is translated into:

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

Here, we have omitted the instructions **alloc** 0 and **slide** 0 1. Also note that the jump with address B is not reachable and, thus, can safely be removed. \square

2.10 Translation of Programs

It remains to specify how a complete C program p should be translated. For that, we assume that the value of the Heap Pointer, HP , before program execution is the largest memory address $+1$ and all other registers have the value 0. In particular, this means that the execution of the CMA program for p starts with the instruction in $C[0]$.

The program p is a sequence of declarations of global variables and functions, one of which defines the function **int** $main()$. Note that, for simplicity, we do not consider command line parameters for the function $main$. The code for program p , thus, consists of:

- code for the allocation of global variables;
- code for the function definitions;
- code for the call of function *main*;
- the instruction **halt**, to terminate program execution.

For simplicity, we assume that the variable declarations in p appear before the function declarations. Program p , thus, has the form:

$$dd\ df_1 \dots df_n$$

where dd is a sequence of variable declarations and df_1, \dots, df_n are respectively the declarations of functions f_1, \dots, f_n with $f_n \equiv \text{main}$. Assume that \emptyset identifies the empty address environment. Then

$$(\rho_0, k) = \text{elab_globals}(1, \emptyset)\ dd$$

is the pair consisting of the address environment for the global variables of p and the first free relative address. If $_f_i$ is the start address of function f_i , then the address environment before execution of the i -th function definition is given by:

$$\rho_i = \rho_{i-1} \oplus \{f_i \mapsto _f_i\} \quad (i = 1, \dots, n)$$

We obtain the following translation scheme for p :

code p	=	enter $(k + 3)$ alloc k mark loadc $_f_n$ call slide $(k - 1)\ 1$ halt
$_f_1$:		code $df_1\ \rho_1$
		\vdots
$_f_n$:		code $df_n\ \rho_n$

Before the call to the main function f_n , a total of $(k - 1)$ memory cells for globals are allocated on top of the (forbidden) memory location with address 0. One more cell is reserved for the return value of f_n . Since SP has the value 0 before program execution, SP must be incremented by k . When executing the instruction **call**, a total of 3 further memory cells have been allocated on the stack. Accordingly, the instruction **enter** at program start must set the register EP to $SP + k + 3 = k + 3$. The return value of the initial call to *main* is placed in the memory location with address k . The instruction **halt**, which is supposed to return control to the operating system, does not know about globals and therefore expects the return value in a specific memory location, say with address 1. Therefore, the return value after the call is slid downwards by $k - 1$ positions.

Warnings

The translation of imperative languages is now complete. For didactical reasons, we have made several simplifications. Some important real-world complications have been ignored. One such complication is incurred by bounded word sizes.

Let us focus on the treatment of constants. We have liberally allowed arbitrary *int* constants as parts of CMA instructions, sometimes even two constant operands for one instruction. A machine similar to our C-Machine, but for PASCAL – the P-Machine – has differentiated between cases in which the constant operands fit within the admissible space for the instruction and those where this is not the case. Large constants were placed in a dedicated constant table. Such a table is also mandatory for arbitrary length constants such as strings. The instructions may contain references to this table. The same operation may be implemented by several instructions that differ in that they expect their operands in the instruction itself or in the constant table. In the latter case, relative addresses are stored in the instruction itself. The compiler, or a downstream assembler in the case of the Zürich P4 compiler, generated different instructions depending on the sizes of the constants and stored the constants in the instructions or the constant table.

Open Questions and Further References

In this chapter we have *specified* the translation of a C-like language into a language of a suitable virtual machine. By now, the interested reader might want to know how to *implement* the given translation schemes.

- The translation schemes are defined by recursion on the syntactic structure of programs. They assume that the syntactic structure is known. Given a suitable hierarchical representation of programs, the schemes can be combined to the definition of recursive code functions. One might wonder whether the same recursive functions could also recognize the structure of programs beforehand. To a certain extent this is possible, and we will discuss efficient methods for recognizing the structure of programs in the subsequent volume.
- For the compilation of assignments and expressions, the types of variables and expressions were used. We did not specify how the type of an expression is determined. That is part of the static analysis, which will also be discussed in the next volume.
- In the examples, we realized that the translation schemes did not always lead to the optimal, that is, the most efficient CMA instruction sequence. Techniques for improving translation schemes that lead to better, maybe even optimal, instruction sequences, will be addressed in Volume 4 on code generation. Here, non-local information about dynamic, that is run-time, properties is needed. Techniques for such static program analysis are discussed in depth in Volume 3.
- The use of virtual machines simplifies code generation. New problems arise if target programs for real machines are to be generated in a way that exploits the

potentials offered by their architectures. One problem is to optimally make use of the register file of the target machine. Such problems will also be addressed in Volume 4.

2.11 Exercises

1. *Code generation for expressions.*

Generate CMA code for the following expressions:

$$\begin{aligned} a &\leftarrow 2 \cdot (c + (b - 3)) \\ b &\leftarrow b \cdot (a + 3) \end{aligned}$$

Assume the following address environment:

$$\rho = \{a \mapsto 5, b \mapsto 6, c \mapsto 7\}$$

Execute the generated code by displaying the stack contents after each instruction! Assume that the variables are initialized with the values $a = 22$, $b = 33$ and $c = 44$.

2. *Code generation for loops.*

Generate CMA code for the two loops:

```

while ( $x > y$ ) {
    if ( $2 \cdot y > x$ )  $y \leftarrow y + x$ ;
    else  $x \leftarrow x - y$ ;
}

for ( $x \leftarrow 0; x < 42; x \leftarrow x + z$ )
    if ( $\neg(x = y)$ )  $z \leftarrow z + 1$ ;

```

Use the following address environment:

$$\rho = \{x \mapsto 2, y \mapsto 3, z \mapsto 5\}$$

3. *Code generation for statement sequences.*

Consider the following sequence of statements:

```

 $z \leftarrow 1;$ 
while ( $n > 0$ ) {
     $j \leftarrow 1;$ 
     $y \leftarrow x;$ 
    while ( $2 \cdot j \leq n$ ) {
         $y \leftarrow y \cdot y;$ 
         $j \leftarrow j \cdot 2;$ 
    }
     $z \leftarrow y \cdot z;$ 
     $n \leftarrow n - j;$ 
}

```

- What does this statement sequence compute?
- Translate the statement sequence into CMA code! Use the following address environment:

$$\rho = \{n \mapsto 1, j \mapsto 2, x \mapsto 3, y \mapsto 4, z \mapsto 5\}$$

4. *Reverse engineering.*

Consider the following CMA code:

loadc 0		pop	storea 1
loadc 1		jump B	pop
loadc 13	A :	loada 3	loadc 2
loada 3		loada 2	loada 2
loadc 1		geq	mul
le		jumpz B	storea 2
jumpz A		loada 1	pop
loadc -1		loadc 1	jump A
storea 1		add	B : halt

- Execute this CMA code. Assume that prior to program execution the value of SP is 0.
 - What does the given code compute when the memory location with address 3 is given as input?
5. *Short circuit evaluation.*

Let b , e_1 , and e_2 be arbitrary expressions:

- A conditional expression in C has the form $b ? e_1 : e_2$. Its value is the value of e_1 if $b \neq 0$ and the value of e_2 if $b = 0$. Design a translation scheme for `codeR` ($b ? e_1 : e_2$) ρ .
- *Short circuit evaluation* for a Boolean expression means that the second term of a conjunction (disjunction) is not evaluated if the evaluation of the first term is already 0 (or a non-zero value in the case of a disjunction).

Give translation schemes for $\text{code}_R(e_1 \wedge e_2) \rho$ (and $\text{code}_R(e_1 \vee e_2) \rho$) using *short circuit evaluation*!

6. *Breaks.*

Modify the scheme for the translation of loops so that it can deal with *break* statements, which cause the immediate exit from the loop! For this, extend the translation function with a further argument l , which specifies the jump target to which control proceeds in the case of a **break**.

7. *Continues.*

A **continue** statement causes a jump to the end of the body of the enclosing loop. How must the translation schemes be modified in order to correctly translate **continue** statements at arbitrary locations?

Hint: Extend the translation function with yet another argument!

8. *Jump tables I.*

Extend the translation scheme for the *switch* statement to allow negative values as cases and also (singular and small) gaps within the range of case values. Dream up heuristics to deal with large and irregular gaps.

9. *Jump tables II.*

We have translated the *switch* statement by means of a relative jump into the jump table from which a further direct jump leads to the start address of the selected alternative. The first jump can be saved if the jump table does not contain jump instructions but jump targets. Then the instruction **jumpi** must be replaced by an instruction **jumpi'**, which jumps straight to the address stored in the table. There are two variants:

- a) The jump table is located *after* the code for individual cases. Then the **jumpi'** receives the start address of the table as argument;
- b) The jump table is located directly behind the instruction **jumpi'**, that is, *before* the code for the individual cases. Then, the start address must be provided as an argument to the instruction.

Define adequate instructions **jumpi'** for both cases and give translation schemes for each!

10. *Code generation for pointers.*

Consider the following definitions:

```

int * m, n;
struct list {
    int info;
    struct list * next;
} * l, * tmp;
m ← malloc(sizeof(int));
*m ← 4;
l ← null;
for (n ← 5; n ≤ *m; n++) {
    tmp ← l;
    l ← malloc(sizeof(struct list));
    l → next ← tmp;
    l → info ← n;
}

```

- Compute an address environment for the variables.
- Generate code for the program.
- Execute the generated code.

11. *Extreme Pointer.*

In order to set the Extreme Pointer *EP*, the compiler requires the maximal size of the local stack. This value can be computed at compile-time.

Define a function t that computes for the expression e a precise upper bound $t(e)$ for the number of stack locations that are maximally needed for the evaluation of e .

- Compute $t(e)$ for the two extreme cases:
 $a_1 + (a_2 + (\dots + (a_{n-1} + a_n) \dots))$ and $(\dots ((a_1 + a_2) + a_3) + \dots) + a_n$
(a_i constants or basic type variables).
- Compute $t(e)$ for the expressions given in Exercise 1 and Example 2.8.1.
- Extend the definition of the function t to C statements, in particular, *if*, *for*, and *while* statements.

12. *Blocks.*

Extend the code generation function for statement sequences to blocks. For this, allow that variable declarations may occur in arbitrary positions within a block. Declared variables are supposed to be visible from the point where they are introduced to the end of the current block.

As an example, consider the following program:

```

int x;
x ← 1;
{
    int x;
    x ← 2;
    write(x);
}
write(x);

```

Here, variable x first receives the value of 1. Another variable with the name x is introduced in the inner block, which receives the value 2. When the inner block is finished, this second variable x disappears and the first variable x with value of 1 becomes visible again.

Hint: Together with the address environment, maintain the first relative address behind the locals.

13. *Initialization of variables.*

Modify the function `code` so that variables can be initialized. For example, it should be possible to compile the following definition:

```
int a ← 0;
```

14. *Post- and preincrement.* Give transation schemes for:

- ++ and -- (prefix and postfix).
- +← and −←.

Take care that the operators are not only defined for variables, but can also be applied to arbitrary expressions.

15. *Code generation for functions.*

Consider the following definitions:

```

int n;
struct tree {
    int info;
    struct tree * left, * right;
} * t;
struct tree * mktree (int d, int * n) {
    struct tree * t;
    if (d ≤ 0) return null;
    else {
        t ← malloc(sizeof(struct tree));
        t → left ← mktree(d − 1, n);
        t → info ← *n; *n ← *n + 1;
        t → right ← mktree(d − 1, n);
        return t;
    }
}

```

Compute the address environment for the function *mktree* and generate code for the function as well as for the assignment:

$$t \leftarrow \text{mktree}(5, \&n);$$

16. *Reference parameters.*

C++ offers, in addition to parameter passing *by value*, also parameter passing *by reference*. Consider the following code fragment:

```

int a;
void f(int &x) { x ← 7; }
int main() {
    f(a); return 0;
}

```

if the formal parameter *x* is used in the body of the function *f*, then the target variable is the one whose *L*-value has been passed to *x* as actual parameter. After the execution of *f*(*a*), the variable *a* should contain the value 7.

Translate the example program and check that it behaves as expected.

17. *Variable argument lists.*

In this exercise, we consider functions with variable argument lists. The parameter lists of such functions first enumerate the mandatory parameters, followed by "..." for the optional ones. Given that these all have type *t*, the *R*-value of the next optional parameter should be accessed by means of the call *next*(*t*). Consider for example the function:

```

int sum(int n, ...) {      // Sum of n numbers
    int result  $\leftarrow$  0;
    while (n > 0) {
        result  $\leftarrow$  result + next(int);
        n  $\leftarrow$  n - 1;
    }
    return result;
}

```

Possible calls are for example *sum* (5, *a*, *b*, *c*, *d*, *e*) and *sum* (3, 1, 2, 3).

- Verify whether our translation for such function calls is adequate.
- Invent an implementation for *next*(*t*).

18. *Code generation for programs.*

Translate the following program:

```

int result;
int fib (int n) {
    int result;
    if (n < 0) return -1;
    switch (n) {
        case 0 : return 0; break;
        case 1 : return 1; break;
        default : return fib (n - 1) + fib (n - 2);
    }
}

int main() {
    int n;
    n  $\leftarrow$  5;
    result  $\leftarrow$  fib (n);
    return 0;
}

```

19. *CMA interpreters.*

Implement an interpreter for the C-Machine in the programming language of your choice.

- Choose a suitable data type for instructions. Take into account that some instructions have arguments, but some have not.
- Implement the datastructures C and S.
- Test your interpreter with the factorial function and the main function:

```

int main() { return fac(9); }

```


2.12 List of CMA Registers

EP , Extreme Pointer	p. 28
FP , Frame Pointer	p. 35
HP , Heap Pointer	p. 27
PC , Program Counter	p. 8
SP , Stack Pointer	p. 8

2.13 List of Code Functions of the CMA

code	p. 12
code_L	p. 12
code_R	p. 12

2.14 List of CMA Instructions

add	p. 10	jumpz	p. 16	neg	p. 11
and	p. 10	jumpi	p. 20	neq	p. 10
alloc	p. 42	leq	p. 10	new	p. 28
call	p. 40	le	p. 10	or	p. 10
div	p. 10	load	p. 13	pop	p. 15
dup	p. 21	loada	p. 14	return	p. 44
enter	p. 41	loadc	p. 9	slide	p. 42
eq	p. 10	loadr	p. 38	store	p. 13
geq	p. 10	loadrc	p. 37	storea	p. 14
gr	p. 10	malloc	p. 28	storer	p. 38
halt	p. 46	mark	p. 40	sub	p. 10
jump	p. 16	mul	p. 10		

2.15 References

Language-oriented virtual machines have been around for quite a while. They have been introduced for simplifying compilation and for porting to different machines. In [RR64] a virtual machine for ALGOL60, the *Algol Object Code* (AOC), was described.

The model for current virtual machines for imperative languages is the P-Machine which was used for the widely distributed P4 Pascal compiler from Zürich. It is described in [Amm81] and in [PD82], where sources for the P4 compiler, assembler, and P-Machine interpreter can be found.



<http://www.springer.com/978-3-642-14908-5>

Compiler Design

Virtual Machines

Wilhelm, R.; Seidl, H.

2010, XIII, 187 p., Hardcover

ISBN: 978-3-642-14908-5