Example for if-statement

Let $\rho = \{ x \mapsto 4, y \mapsto 7 \}$ and let $s$ be the statement

```c
if (x>y) {
  x = x - y;
} else {
  y = y - x;
}
```

Then $\text{code}^t s \rho$ yields:

(i) $\text{code}^t_s \rho$

(ii) $\text{code}^t_A \rho$

(iii) $\text{code}^t_R \rho$

Iterating Statements

We only consider the loop $s \equiv \text{while} (e) s'$. For this statement we define:

$$\text{code}^t \text{while} (e) s \rho = A : \text{code}^t \text{while} (e) s' \rho, B : \text{code}^t \text{for} (s') \rho, A : \text{jmpz} R_1 B, \text{jump} A$$

- $\text{code}^t \text{for} (s') \rho$
- $\text{jump}$
**Example: Translation of Loops**

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

```c
while (a>0) {
    c = c + 1; /* (ii) */
    a = a - b; /* (iii) */
}
```

Then \( \text{code}^l s \rho \) evaluates to:

\[ A: \text{move } R_i R_j \]
\[ \text{load } R_i R_{i+1} \]
\[ \text{gr } R_i R_i R_{i+1} \]
\[ \text{jumpe } R_i B \]

\[ B: \]

**for-Loops**

The for-loop \( s' \equiv \text{for}(e_1; e_2; e_3) \) \( s' \) is equivalent to the statement sequence \( e_1 \) while \( (e_2) \{ s' \} e_3 \) as long as \( s' \) does not contain a continue statement. Thus, we translate:

\[
\text{code}^l \text{for}(e_1; e_2; e_3) s \rho = \text{code}^l e_1 \rho \]
\[
A: \quad \text{code}^l e_2 \rho \]
\[
\text{jumpz } R_i B \]
\[
\text{code}^l s \rho \]
\[
\text{code}^l e_3 \rho \]
\[
\text{jump } A \]

\[ B: \]

**The switch-Statement**

**Idea:**

- Suppose choosing from multiple options in constant time if possible
- Use a jump table that, at the \( i \)th position, holds a jump to the \( i \)th alternative
- In order to realize this idea, we need an indirect jump instruction

\[ \text{switch } (x) \{ \ldots \} \]
\[ x \in [0, b] \]

\[ \text{case } 0 \{ \]
\[ \quad \text{jump } B + x \]
\[ \}

\[ B: \]
The switch-Statement

Idea:
- Suppose choosing from multiple options in constant time if possible
- use a jump table that, at the $i$th position, holds a jump to the $i$th alternative
- in order to realize this idea, we need an indirect jump instruction

Consecutive Alternatives

Let $\text{switch } s$ be given with $k$ consecutive case alternatives:

```java
switch (e) {
    case $c_0$: $s_0$; break;
    :
    case $c_k-1$: $s_{k-1}$; break;
    default: $s$; break;
}
```

that is, $c_i + 1 = c_{i+1}$ for $i = [0, k - 1]$.

Define $\text{code'} s\rho$ as follows:

- $\text{code'} s\rho = \text{code'} s\rho$
- $\text{check}^i c_0 c_{k-1} B$
- $A_0: \text{code'} s_0\rho$
- $\text{jump } D$
- $: :$
- $A_{k-1}: \text{code'} s_{k-1}\rho$
- $\text{jump } D$

Consecutive Alternatives

Let $\text{switch } s$ be given with $k$ consecutive case alternatives:

```java
switch (e) {
    case $c_0$: $s_0$; break;
    :
    case $c_k-1$: $s_{k-1}$; break;
    default: $s$; break;
}
```

that is, $c_i + 1 = c_{i+1}$ for $i = [0, k - 1]$.

Define $\text{code'} s\rho$ as follows:

- $\text{code'} s\rho = \text{code'} s\rho$
- $\text{check}^i c_0 c_{k-1} B$
- $B: \text{jump } A_0$
- $A_0: \text{code'} s_0\rho$
- $: :$
- $\text{jump } D$
- $: :$
- $A_{k-1}: \text{code'} s_{k-1}\rho$
- $\text{jump } D$

$\text{check}^i l u B$ checks if $l \leq R_i < u$ holds and jumps accordingly.
Translation of the \texttt{check} Macro

The macro \texttt{check} \( l u B \) checks if \( l \leq R_i < u \). Let \( k = u - l \).
- if \( l \leq R_i < u \) it jumps to \( B + R_i - l \)
- if \( R_i < l \) or \( R_i \geq u \) it jumps to \( C \)

\[
B : \quad \text{jump } A_0 \\
\vdots \quad \vdots \\
\quad \text{jump } A_{k-1} \\
C : \quad \text{...}
\]

\[
R_i \in [l, u - 1] \\
R_i \geq k
\]

\[
E : \quad \text{load } R_i k \\
D : \quad \text{jump } R_i B
\]

Translation of the \texttt{check} Macro

The macro \texttt{check} \( l u B \) checks if \( l \leq R_i < u \). Let \( k = u - l \).
- if \( l \leq R_i < u \) it jumps to \( B + R_i - l \)
- if \( R_i < l \) or \( R_i \geq u \) it jumps to \( C \)

we define:

\[
\text{check} \ l \ u \ B = \quad \text{load } R_{i+1} l \\
\quad \text{geq } R_{i+2} R_i R_{i+1} \\
\quad \text{jumpz } R_{i+2} E \\
\quad \text{sub } R_i R_i R_{i+1} \\
\quad \text{load } R_{i+1} k \\
\quad \text{geq } R_{i+2} R_i R_{i+1} \\
\quad \text{jumpz } R_{i+2} D \\
\quad \vdots \\
\quad \vdots \\
\quad \text{jump } A_0 \\
B : \quad \text{jump } A_0 \\
\vdots \\
\quad \text{jump } A_{k-1} \\
C : \quad \text{...}
\]

Note: a jump \texttt{jump } R_i B \text{ with } R_i = k \text{ winds up at } C.

Improvements for Jump Tables

This translation is only suitable for \texttt{certain switch}-statement.
- In case the table starts with 0 instead of \texttt{L}, we don’t need to subtract it from \texttt{L} before we use it as index
- if the value of \texttt{L} is guaranteed to be in the interval \([l, u]\), we can omit \texttt{check}
- can we implement the \texttt{switch}-statement using an \texttt{L}-attributed system without symbolic labels?
Improvements for Jump Tables

This translation is only suitable for certain switch-statement.
- In case the table starts with 0 instead of 1 we don’t need to subtract it from \( e \) before we use it as index
- if the value of \( e \) is guaranteed to be in the interval \([1, n]\), we can omit check
- can we implement the switch-statement using an L-attributed system without symbolic labels?
  - difficult since \( B \) is unknown when check' is translated
  - \( \sim \) use symbolic labels or basic blocks

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an if-ladder, that is, a sequence of if-statements
- for \( n \) cases, an if-cascade (tree of conditionals) can be generated \( \sim O(\log n) \) tests

General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an if-ladder, that is, a sequence of if-statements
- for \( n \) cases, an if-cascade (tree of conditionals) can be generated \( \sim O(\log n) \) tests
- if the sequence of numbers has small gaps (\( \leq 3 \)), a jump table may be smaller and faster
General translation of switch-Statements

In general, the values of the various cases may be far apart:
- generate an if-ladder, that is, a sequence of if-statements
- for $n$ cases, an if-cascade (tree of conditionals) can be generated $\sim O(\log n)$ tests
- if the sequence of numbers has small gaps ($\leq 3$), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases

Translation into Basic Blocks

**Problem:** How do we connect the different basic blocks?

**Idea:**
- translation of a function: create an empty block and store a pointer to it in the node of the function declaration
Translation into Basic Blocks

**Problem:** How do we connect the different basic blocks?

**Idea:**
- Translation of a function: create an empty block and store a pointer to it in the node of the function declaration.
- Pass this block down to the translation of statements.
- Each new statement is appended to this basic block.

---

Translation into Basic Blocks

**Problem:** How do we connect the different basic blocks?

**Idea:**
- Translation of a function: create an empty block and store a pointer to it in the node of the function declaration.
- Pass this block down to the translation of statements.
- Each new statement is appended to this basic block.
- A two-way if-statement creates three new blocks:
  1. One for the then-branch, connected with the current block by a jump-edge.
  2. One for the else-branch, connected with the current block by a jump-edge.
  3. One for the following statements, connect to the then- and else-branch by a jump edge.
- Similar for other constructs.

---

Ingredients of a Function

The definition of a function consists of:
- A **name** with which it can be called;
- A specification of its **formal parameters**;
- Possibly a **result type**;
- A sequence of **statements**.

In C we have:

\[ \text{code} \quad f \quad \rho = \quad \text{load} \quad f \quad \text{with} \quad \_f \quad \text{starting address of} \quad f \]

**Observe:**
- Function names must have an address assigned to them.
- Since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later.
Memory Management in Functions

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

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

At run-time several instance may be active, that is, the function has been called but has not yet returned.

The recursion tree in the example:

```
        main
       /   \
       /     \n      /       \
     /         \
fac       fac
     /   \       /   \ 
    /     \     /     \ 
   fac     fac fac
       /     /   / \
      /   fac fac
```

Memory Management in Function Variables

The formal parameters and the local variables of the various (instances) of a function must be kept separate

Idea for implementing functions:

- set up a region of memory each time it is called
Memory Management in Function Variables

The formal parameters and the local variables of the various (instances) of a function must be kept separate.

Idea for implementing functions:
- set up a region of memory each time it is called
- in sequential programs this memory region can be allocate on the stack
- thus, each instance of a function has its own region on the stack
- these regions are called stack frames

Organization of a Stack Frame

- stack representation: grows upwards
- SP points to the last used stack cell

SP

PCold
FPold
EPold

local memory callee
organizational cells
local memory callee

FP ≡ frame pointer: points to the last organizational cell
use to recover the previously active stack frame

Organization of a Stack Frame

- stack representation: grows upwards
- SP points to the last used stack cell

SP

PCold
FPold
EPold

local memory callee
organizational cells
local memory callee

FP ≡ frame pointer: points to the last organizational cell
use to recover the previously active stack frame

Principle of Function Call and Return

actions taken on entering g:

1. compute the start address of g
2. compute actual parameters
3. backup of caller-save registers
4. backup of FP, EP
5. set the new FP
6. back up of PC and jump to the beginning of g
7. setup new EP
8. allocate space for local variables
   
   { saveloc, mark } are in f
   { call } are in g
   { enter, alloc } are in g

actions taken on leaving g:

1. compute the result
2. restore FP, EP, SP
3. return to the call site in f, that is, restore PC
4. restore the caller-save registers
5. clean up stack

   { return } are in g
   { restoreloc, pop k } are in f
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:

- automatic variables live in *local* registers $R_i$
- intermediate results also live in *local* registers $R_i$
- parameters *global* registers $R_i$ (with $i \leq 0$)
- global variables:

- the $i$th argument of a function is passed in register $R_i$
Managing Registers during Function Calls

The two register sets (global and local) are used as follows:
- automatic variables live in local registers \( R_i \)
- intermediate results also live in local registers \( R_i \)
- parameters global registers \( R_i \) (with \( i \leq 0 \))
- global variables: let's suppose there are none

Convention:
- the \( i \)th argument of a function is passed in register \( R_i \)
- the result of a function is stored in \( R_0 \)
- local registers are saved before calling a function

Definition

Let \( f \) be a function that calls \( g \). A register \( R_i \) is called
- caller-saved if \( f \) backs up \( R_i \), and \( g \) may overwrite it
- callee-saved if \( f \) does not back up \( g \) must restore it before it returns

Translation of Function Calls

A function call \( g(e_1, \ldots, e_n) \) is translated as follows:

\[
\text{code}^R_0 g(e_1, \ldots, e_n) \rho = \begin{cases}
\text{code}^R_0 g \rho & \text{if } f \text{ does not back up } g \\
\text{code}^R_{i+1} e_1 \rho & \text{if } f \text{ backs up } R_i \\
\vdots & \\
\text{code}^R_{i+n} e_n \rho & \\
\text{move } R_{i-1} R_{i+1} \\
\vdots & \\
\text{move } R_{i-n} R_{i+n} \\
\text{saveloc } R_1 R_{i-1} \\
\text{mark} \\
\text{call } R_i \\
\text{saveloc } R_1 R_{i-1} \\
\text{move } R_i R_0 
\end{cases}
\]

New instructions:
- \text{saveloc } R_i R_j \text{ pushes the registers } R_i, R_{i+1}, \ldots, R_j \text{ onto the stack}
- \text{mark} \text{ backs up the organizational cells}
- \text{call } R_i \text{ calls the function at the address in } R_i
- \text{restoreloc } R_i R_j \text{ pops } R_i, R_{i-1}, \ldots, R_j \text{ off the stack}
Rescuing EP and FP

The instruction mark allocates stack space for the return value and the organizational cells and backs up FP and EP.

\[ S[SP+1] = EP; \]
\[ S[SP+2] = FP; \]
\[ SP = SP + 2; \]

Calling a Function

The instruction call rescues the value of PC+1 onto the stack and sets FP and PC.

Result of a Function

The global register set is also used to communicate the result value of a function:

\[ \text{code}^f \text{ return } e \rho = \text{code}_k e \rho \]
\[ \text{move } R_0, R_i \]
\[ \text{return} \]

Result of a Function

The global register set is also used to communicate the result value of a function:

\[ \text{code}^f \text{ return } e \rho = \text{code}_k e \rho \]
\[ \text{move } R_0, R_i \]
\[ \text{return} \]

alternative without result value:

\[ \text{code}^f \text{ return } \rho = \text{return} \]
Return from a Function

The instruction `return` relinquishes control of the current stack frame, that is, it restores PC, EP and FP.

\[
\begin{array}{c}
\text{PC} \quad \text{FP} \\
\text{p} \quad \text{e} \\
\text{EP} \\
\end{array}
\]

Return

\[
\begin{array}{c}
\text{PC} \\
\text{p} \\
\text{EP} \\
\end{array}
\]

PC = S[FP]; EP = S[FP-2];
SP = FP-3; FP = S[SP+2];

Translation of Functions

The translation of a function is thus defined as follows:

\[
\begin{array}{c}
code^1_{t_r} \cdot (\text{args}) \{ \text{decls} \} \cdot ss \cdot \rho =
\end{array}
\]

enter \ q
move \ R_{t+1} \ R_{-1}
move \ R_{t+n} \ R_{-n}

\[
\begin{array}{c}
code^{t_{r+n+1}}_{ss} \rho'
\end{array}
\]

return

Assumptions:
- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots, R_l \)
- the parameters of the function are in \( R_{-1}, \ldots, R_{-n} \)
Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 \left< t, \xi(\text{args}) \{ \text{decls} \ \text{ss} \} \ \rho \right> = \begin{align*}
\text{enter} & \quad q \\
\text{move} & \quad R_{i+1} \ R_{-i} \\
\vdots & \\
\text{move} & \quad R_{i+n} \ R_{-n} \\
\text{code}^1 & \quad \text{ss} \ \rho' \\
\text{return} & 
\end{align*}
\]

Assumptions:

- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
- the parameters of the function are in \( R_{-1}, \ldots R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)
- \text{return} is not always necessary

Result of a Function

The global register set is also used to communicate the result value of a function:

\[
\text{code}^1 \text{return} \ e \ \rho = \begin{align*}
\text{code}^1 & \quad e \ \rho \\
\text{move} & \quad R_0 \ R_i \\
\text{return} & 
\end{align*}
\]

Alternative without result value:

\[
\text{code}^1 \text{return} \ \rho = \text{return}
\]

global registers are otherwise not used inside a function body:

- advantage: at any point in the body another function can be called without backing up global registers
- disadvantage: on entering a function, all global registers must be saved

Translation of Functions

The translation of a function is thus defined as follows:

\[
\text{code}^1 \left< t, \xi(\text{args}) \{ \text{decls} \ \text{ss} \} \ \rho \right> = \begin{align*}
\text{enter} & \quad q \\
\text{move} & \quad R_{i+1} \ R_{-i} \\
\vdots & \\
\text{move} & \quad R_{i+n} \ R_{-n} \\
\text{code}^1 & \quad \text{ss} \ \rho' \\
\text{return} & 
\end{align*}
\]

Assumptions:

- the function has \( n \) parameters
- the local variables are stored in registers \( R_1, \ldots R_l \)
- the parameters of the function are in \( R_{-1}, \ldots R_{-n} \)
- \( \rho' \) is obtained by extending \( \rho \) with the bindings in \( \text{decls} \) and the function parameters \( \text{args} \)
Translation of Whole Programs

A program $P = F_1; \ldots; F_n$ must have a single main function.

\[
\text{code}^1 P \rho = \begin{array}{l}
\text{loadc } R_1 \text{ _main} \\
\text{mark} \\
\text{call } R_1 \\
\text{halt} \\
\_f_1 : \text{code}^1 F_1 \rho \otimes \rho_f \\
\_f_2 : \text{code}^1 F_2 \rho \otimes \rho_f \\
\ldots \\
\_f_n : \text{code}^1 F_n \rho \otimes \rho_f
\end{array}
\]

Assumptions:
- $\rho = \emptyset$ assuming that we have no global variables
- $\rho_f$ contain the addresses the local variables
- $\rho_1 \oplus \rho_2 = \lambda x . \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$

Translation of the \texttt{fac}-function

Consider:

\[
\text{int fac(int x) \{} \\
\text{if } (x <= 0) \text{ then} \\
\quad i = 3 \\
\quad \text{move } R_2 R_1 \\
\quad \text{x*fac(x-1)} \\
\quad \text{else} \\
\quad \text{return x*fac(x-1);} \\
\text{to else} \\
\text{return 1;} \\
\text{end}
\]

\[
\_\text{fac:} \begin{array}{l}
\text{move } R_1 R_{-1} \\
\text{move } R_2 R_3 \\
\text{leq } R_2 R_2 R_3 \\
\text{jumpz } R_2 A \\
\text{move } R_3 R_1 \\
\text{move } R_2 R_2 \\
\text{jump } B
\end{array}
\]

\[
\begin{array}{l}
\_A: \text{move } R_3 R_1 \\
\text{move } R_3 R_1 \\
\text{move } R_2 R_1 \\
\text{move } R_2 R_1 \\
\text{move } R_2 R_1 \\
\text{return x*...}
\end{array}
\]

\[
\begin{array}{l}
\_B: \text{return} \\
\text{code is dead}
\end{array}
\]