Script generated by TTT

Title: Simon: Compilerbau (30.06.2014)

Date: Mon Jun 30 14:15:39 CEST 2014

Duration: 90:40 min

Pages: 58

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compiler Construction I

Dr. Michael Petter, Dr. Axel Simon

SoSe 2014

Example for if-statement

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

```
if (x>y) {          /* (i)   */
   x = x - y;       /* (ii)  */
} else {
   y = y - x;       /* (iii) */
}
```

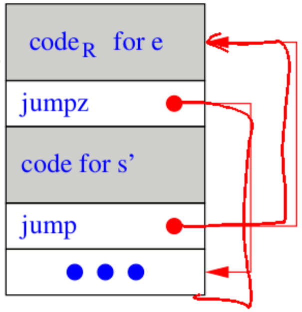
Then $\text{code}^i\ s\ \rho$ yields:

(i)	(ii)	(iii)	
move R_i R_4	move R_i R_4	A :	move R_i R_7
move R_{i+1} R_7	move R_{i+1} R_7		move R_{i+1} R_4
gr R_i R_i R_{i+1}	sub R_i R_i R_{i+1}		sub R_i R_i R_{i+1}
jumpz R_i A	move R_4 R_i		move R_7 R_i
	jump B	B :	

Iterating Statements

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

code^i `while(e) s` ρ $= A$: $\text{code}^i_R\ e\ \rho$
jumpz R_i B
$\text{code}^i\ s\ \rho$
jump A
B :



Example: Translation of Loops

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

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

Then $\text{code}^i\ s\ \rho$ evaluates to:

A: move R_i R_7
loadc R_{i+1} 0
gr R_i R_i R_{i+1}
jumpz R_i B

move R_i R_9
loadc R_{i+1} 1
add R_i R_i R_{i+1}
move R_9 R_i
⋮
B:

Example: Translation of Loops

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

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

Then $\text{code}^i\ s\ \rho$ evaluates to:

	(i)	(ii)	(iii)
$A:$	move R_i R_7	move R_i R_9	move R_i R_7
	loadc R_{i+1} 0	loadc R_{i+1} 1	move R_{i+1} R_8
	gr R_i R_i R_{i+1}	add R_i R_i R_{i+1}	sub R_i R_i R_{i+1}
	jumpz R_i B	move R_9 R_i	move R_7 R_i
			jump A
		$B:$	

for-Loops

The **for**-loop $s \equiv$ **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:

$$
\begin{array}{rcl}
\text{code}^i\ \texttt{for}(e_1; e_2; e_3)\ s\ \rho & = & \text{code}^i_R\ e_1\ \rho \\
 & A: & \text{code}^i_R\ e_2\ \rho \\
 & & \text{jumpz}\ R_i\ B \\
 & & \text{code}^i\ s\ \rho \\
 & & \text{code}^i_R\ e_3\ \rho \\
 & & \text{jump}\ A \\
 & B: &
\end{array}
$$

The switch-Statement

Idea:

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

switch (x) { $x \in [0, k]$
case 0:
⋮
k:

jump B + x

The switch-Statement

Idea:

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

PC = A + R_i;

Consecutive Alternatives

Let **switch** s be given with k consecutive **case** alternatives:

```
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 $i = [0, k-1]$.

Consecutive Alternatives

Let **switch** s be given with k consecutive **case** alternatives:

```
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 $\text{code}^i\ s\ \rho$ as follows:

$$\begin{array}{rcl} \text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho \\ & & \text{check}^i\ c_0\ c_{k-1}\ B \\ & A_0: & \text{code}^i\ s_0\ \rho \\ & & \text{jump}\ D \\ & \vdots & \vdots \\ & A_{k-1}: & \text{code}^i\ s_{k-1}\ \rho \\ & & \text{jump}\ D \end{array} \qquad \begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_{k-1} \\ C: & \text{code}^i\ s_d\ \rho \\ D: & \end{array}$$

Consecutive Alternatives

Let **switch** s be given with k consecutive **case** alternatives:

```
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 $\text{code}^i\ s\ \rho$ as follows:

$$\begin{array}{rcl} \text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho \\ & & \text{check}^i\ c_0\ c_{k-1}\ B \\ & A_0: & \text{code}^i\ s_0\ \rho \\ & & \text{jump}\ D \\ & \vdots & \vdots \\ & A_{k-1}: & \text{code}^i\ s_{k-1}\ \rho \\ & & \text{jump}\ D \end{array} \qquad \begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_{k-1} \\ C: & \end{array}$$

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

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ 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

$$\begin{array}{rl} B: & \text{jump } A_0 \\ \vdots & \quad \vdots \\ & \text{jump } A_{k-1} \\ C: & \end{array}$$

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ 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:

$$\begin{array}{rcrl} check^i\ l\ u\ B & = & & \text{loadc } R_{i+1}\ l \\ & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\ & & & \text{jumpz } R_{i+2}\ E \\ & & & \text{sub } R_i\ R_i\ R_{i+1} \\ & & & \text{loadc } R_{i+1}\ k \\ & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\ & & & \text{jumpz } R_{i+2}\ D \\ & & E: & \text{loadc } R_i\ k \\ & & D: & \text{jumpi } R_i\ B \end{array}$$

$$\begin{array}{rl} B: & \text{jump } A_0 \\ \vdots & \quad \vdots \\ & \text{jump } A_{k-1} \\ C: & \end{array}$$

Translation of the $check^i$ Macro

The macro $check^i\ l\ u\ B$ 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:

$$\begin{array}{rcrl} check^i\ l\ u\ B & = & & \text{loadc } R_{i+1}\ l \\ & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\ & & & \text{jumpz } R_{i+2}\ E \\ & & & \text{sub } R_i\ R_i\ R_{i+1} \\ & & & \text{loadc } R_{i+1}\ k \\ & & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\ & & & \text{jumpz } R_{i+2}\ D \\ & & E: & \text{loadc } R_i\ k \\ & & D: & \text{jumpi } R_i\ B \end{array}$$

$$\begin{array}{rl} B: & \text{jump } A_0 \\ \vdots & \quad \vdots \\ & \text{jump } A_{k-1} \\ C: & \end{array}$$

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

Improvements for Jump Tables

This translation is only suitable for *certain* **switch**-statement.

- In case the table starts with 0 instead of u 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 $[l, u]$, we can omit check
- can we implement the **switch**-statement using an 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 u 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 $[l, u]$, we can omit check
- can we implement the **switch**-statement using an L-attributed system without symbolic labels?
  - difficult since B is unknown when $check^i$ is translated
  - $\rightsquigarrow$ use symbolic labels or basic blocks

General translation of switch-Statements

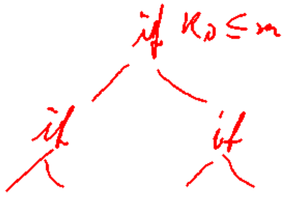
In general, the values of the various cases may be far apart:

- generate an **if**-ladder, that is, a sequence of **if**-statements

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 $\rightsquigarrow O(\log n)$ 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 $\rightsquigarrow O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 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 $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 3), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases

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 $\leadsto O(\log n)$ tests
- if the sequence of numbers has small gaps (≤ 3), a jump table may be smaller and faster
- one could generate several jump tables, one for each sets of consecutive cases
- an **`if`** cascade can be re-arranged by using information from *profiling*, so that paths executed more frequently require fewer tests

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

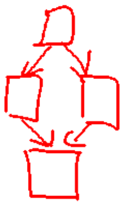
Problem: How do we connect the different basic blocks?
Idea:

- 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

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 jumpz-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

Chapter 5:

Functions

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:

$$\text{code}_R^i\ f\ \rho \quad = \quad \text{loadc}\ \_f \qquad \text{with } \_f \text{ starting address of } f$$

Observe:

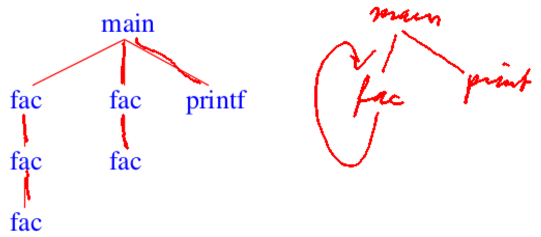
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

Memory Management in Functions

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

```
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:



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:

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

Memory Management in Function Variables

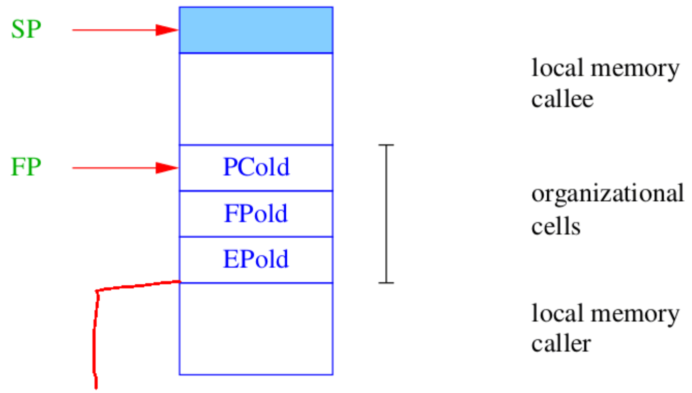
The formal parameters and the local variables of the various (instances) of a function must be kept separate

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

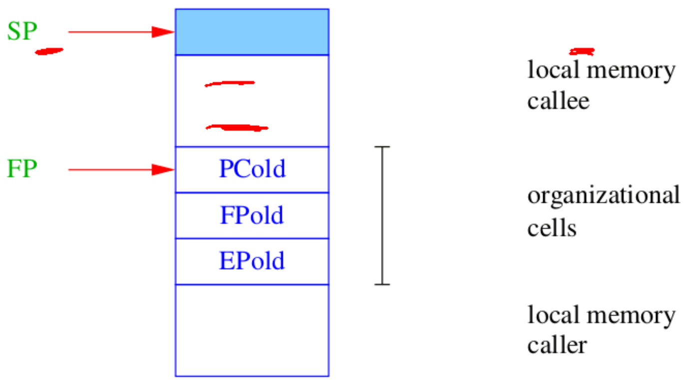
- stack representation: grows upwards
- SP points to the last used stack cell



- 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



- FP ≙ frame pointer: points to the last organizational cell
- use to recover the previously active stack frame
- EP has to do with the heap, will come to that later

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 } saveloc
4. backup of FP, EP } mark
5. set the new FP
6. back up of PC and jump to the beginning of g } call
7. setup new EP } enter
8. allocate space for local variables } alloc

(steps 1–6 are in f; steps 7–8 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 } return
4. restore the caller-save registers } restoreloc
5. clean up stack } pop k

(steps 1–3 are in g; steps 4–5 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 $i > 0$
- intermediate results also live in *local* registers R_i
- parameters *global* registers R_i (with $i \leq 0$)
- global variables:

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:

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

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

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 R_i does not back up g must restore it before it returns

Translation of Function Calls

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

$$
\begin{array}{rcl}
\mathrm{code}_R^i\ \mathrm{g}(e_1,\ldots e_n)\ \rho & = & \mathrm{code}_R^i\ \mathrm{g}\ \rho \\
 & & \mathrm{code}_R^{i+1}\ e_1\ \rho \\
 & & \vdots \\
 & & \mathrm{code}_R^{i+n}\ e_n\ \rho \\
 & & \mathrm{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \mathrm{move}\ R_{-n}\ R_{i+n} \\
 & & \mathrm{saveloc}\ R_1\ R_{i-1} \\
 & & \mathrm{mark} \\
 & & \mathrm{call}\ R_i \\
 & & \mathrm{restoreloc}\ R_1\ R_{i-1} \\
 & & \mathrm{move}\ R_i\ R_0
\end{array}
$$

Translation of Function Calls

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

$$
\begin{array}{rcl}
\mathrm{code}_R^i\ \mathrm{g}(e_1,\ldots e_n)\ \rho & = & \mathrm{code}_R^i\ \mathrm{g}\ \rho \\
 & & \mathrm{code}_R^{i+1}\ e_1\ \rho \\
 & & \vdots \\
 & & \mathrm{code}_R^{i+n}\ e_n\ \rho \\
 & & \mathrm{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \mathrm{move}\ R_{-n}\ R_{i+n} \\
 & & \mathrm{saveloc}\ R_1\ R_{i-1} \\
 & & \mathrm{mark} \\
 & & \mathrm{call}\ R_i \\
 & & \mathrm{restoreloc}\ R_1\ R_{i-1} \\
 & & \mathrm{move}\ R_i\ R_0
\end{array}
$$

New instructions:

- saveloc R_i R_j pushes the registers $R_i, R_{i+1} \ldots R_j$ onto the stack
- mark backs up the organizational cells
- call R_i calls the function at the address in R_i
- restoreloc R_i R_j pops $R_j, R_{j-1}, \ldots R_i$ off the stack

Translation of Function Calls

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

$$
\begin{array}{rcl}
\mathrm{code}_R^i\ \mathrm{g}(e_1,\ldots e_n)\ \rho & = & \mathrm{code}_R^i\ \mathrm{g}\ \rho \\
 & & \mathrm{code}_R^{i+1}\ e_1\ \rho \\
 & & \vdots \\
 & & \mathrm{code}_R^{i+n}\ e_n\ \rho \\
 & & \mathrm{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \mathrm{move}\ R_{-n}\ R_{i+n} \\
 & & \mathrm{saveloc}\ R_1\ R_{i-1} \\
 & & \mathrm{mark} \\
 & & \mathrm{call}\ R_i \\
 & & \mathrm{restoreloc}\ R_1\ R_{i-1} \\
 & & \mathrm{move}\ R_i\ R_0
\end{array}
$$

$$
\begin{array}{l}
\stackrel{?}{=}\ \mathrm{code}_R^i\ e_1\ \rho \\
\mathrm{move}\ R_{-1}\ R_i \\
\vdots \\
\mathrm{code}_R^i\ e_n\ \rho \\
\mathrm{move}\ R_{-n}\ R_i \\
\mathrm{code}_R^i\ g\ \rho \\
\mathrm{saveloc}\ R_1\ R_{i-1} \\
\mathrm{mark} \\
\mathrm{call}\ R_i \\
\mathrm{restoreloc}\ R_1\ R_{i-1} \\
\mathrm{move}\ R_i\ R_0
\end{array}
$$

New instructions:

- saveloc R_i R_j pushes the registers $R_i, R_{i+1} \ldots R_j$ onto the stack
- mark backs up the organizational cells
- call R_i calls the function at the address in R_i
- restoreloc R_i R_j pops $R_j, R_{j-1}, \ldots R_i$ off the stack

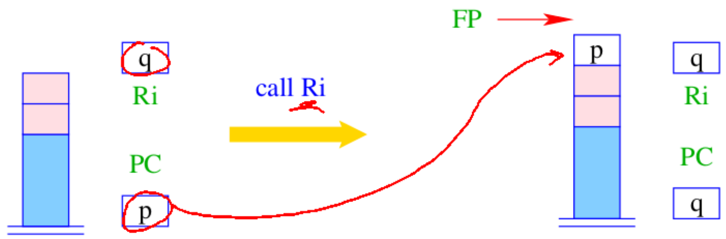
Rescuing EP and FP

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.




```
SP = SP+1;
S[SP] = PC;
FP = SP;
PC = Ri;
```


Result of a Function

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

$$\text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad \text{code}^i_R\ e\ \rho$$
$$\text{move}\ R_0\ R_i$$
$$\text{return}$$

Result of a Function

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

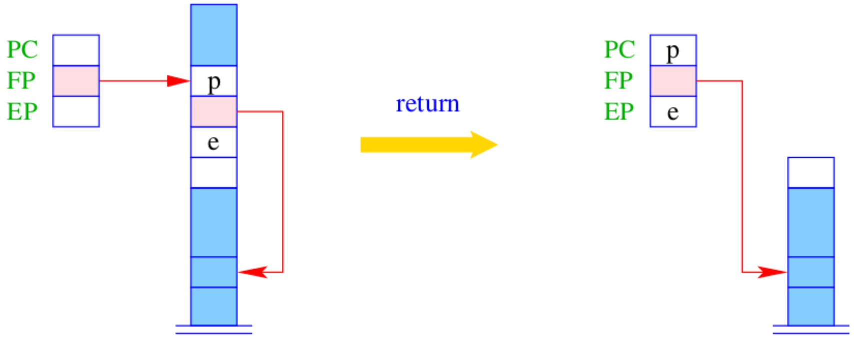
$$\text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad \text{code}^i_R\ e\ \rho$$
$$\text{move}\ R_0\ R_i$$
$$\text{return}$$

alternative without result value:

$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

Return from a Function

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



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:

$$\begin{array}{lll} \text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter } q \\ & & \text{move } R_{l+1}\ R_{-1} \\ & & \vdots \\ & & \text{move } R_{l+n}\ R_{-n} \\ & & \text{code}^{l+n+1}\ ss\ \rho' \\ & & \text{return} \end{array}$$

Assumptions:

Translation of Functions

The translation of a function is thus defined as follows:

$$\begin{array}{lll} \text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter } q \\ & & \text{move } R_{l+1}\ R_{-1} \\ & & \vdots \\ & & \text{move } R_{l+n}\ R_{-n} \\ & & \text{code}^{l+n+1}\ ss\ \rho' \\ & & \text{return} \end{array}$$

Assumptions:

- the function has n parameters

Translation of Functions

The translation of a function is thus defined as follows:

$$\begin{array}{lll} \text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho & = & \text{enter } q \\ & & \text{move } R_{l+1}\ R_{-1} \\ & & \vdots \\ & & \text{move } R_{l+n}\ R_{-n} \\ & & \text{code}^{l+n+1}\ ss\ \rho' \\ & & \text{return} \end{array}$$

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:

$$\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho \quad = \quad \begin{array}{l} \text{enter } q \\ \text{move } R_{l+1}\ R_{-1} \\ \quad \vdots \\ \text{move } R_{l+n}\ R_{-n} \\ \text{code}^{l+n+1}\ ss\ \rho' \\ \text{return} \end{array}$$

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 $R_{-1}, \ldots R_{-n}$
- ρ' is obtained by extending ρ with the bindings in $decls$ and the function parameters $args$
- return is not always necessary

Translation of Functions

The translation of a function is thus defined as follows:

$$\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho \quad = \quad \begin{array}{l} \text{enter } q \\ \text{move } R_{l+1}\ R_{-1} \\ \quad \vdots \\ \text{move } R_{l+n}\ R_{-n} \\ \text{code}^{l+n+1}\ ss\ \rho' \\ \text{return} \end{array}$$

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 $R_{-1}, \ldots R_{-n}$
- ρ' is obtained by extending ρ with the bindings in $decls$ and the function parameters $args$

Result of a Function

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

$$\text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad \begin{array}{l} \text{code}^i_{\text{R}}\ e\ \rho \\ \text{move } R_0\ R_i \\ \text{return} \end{array}$$

alternative without result value:

$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

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:

$$\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho \quad = \quad \begin{array}{l} \text{enter } q \\ \text{move } R_{l+1}\ R_{-1} \\ \quad \vdots \\ \text{move } R_{l+n}\ R_{-n} \\ \text{code}^{l+n+1}\ ss\ \rho' \\ \text{return} \end{array}$$

Assumptions:

- the function has n parameters
- the local variables are stored in registers $R_1, \ldots R_l$

Translation of Whole Programs

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

$$
\begin{array}{rrl}
\text{code}^1\ P\ \rho & = & \text{loadc}\ R_1\ \_\text{main} \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{halt} \\
 & \_f_1: & \text{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
 & \_f_n: & \text{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Translation of Whole Programs

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

$$
\begin{array}{rrl}
\text{code}^1\ P\ \rho & = & \text{loadc}\ R_1\ \_\text{main} \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{halt} \\
 & \_f_1: & \text{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
 & \_f_n: & \text{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Assumptions:

- $\rho = \emptyset$ assuming that we have no global variables
- ρ_{f_i} 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 `fac`-function

Consider:

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

$\rho\{x \mapsto 1\}$

_fac:	enter 5	3 mark+call
	move R_1 R_{-1}	save param.
$i=2$	move R_2 R_1	`if (x<=0)`
	loadc R_3 0	
	leq R_2 R_2 R_3	
	jumpz R_2 _A	to else
	loadc R_2 1	`return 1`
	move R_0 R_2	
	return	
	jump _B	code is dead

_A:	move R_2 R_1	`x*fac(x-1)`
$i=3$	move R_3 R_1	`x-1`
$i=4$	loadc R_4 1	
	sub R_3 R_3 R_4	
$i=3$	move R_{-1} R_3	`fac(x-1)`
	loadc R_3 _fac	
	saveloc R_1 R_2	
	mark	
	call R_3	
	restoreloc R_1 R_2	
	move R_3 R_0	
	mul R_2 R_2 R_3	
	move R_0 R_2	`return x*...`
	return	
_B:	return	