Script generated by TTT

Title: Simon: Compilerbau (15.07.2013)

Date: Mon Jul 15 14:15:55 CEST 2013

Duration: 81:57 min

Pages: 74

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 — are in f
4. backup of FP, EP — mark — are in f
5. set the new FP — call — are in f
6. back up of PC und jump to the beginning of g — call — are in f
7. setup new EP — enter — are in g
8. allocate space for local variables — alloc — are in g

actions taken on leaving g:

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

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:

code^i_L $e(e_1 . e_n)$ = code^i_L e ρ

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

Translation of Function Calls

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

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

$f(x, g(2), y)$

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

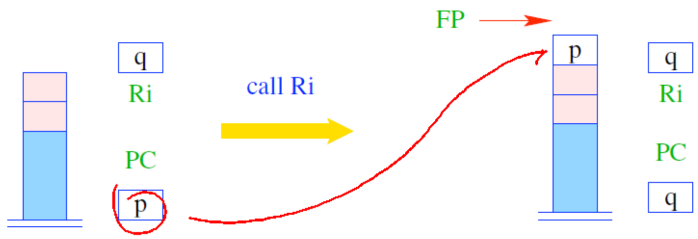
Rescuing EP and FP

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.



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

push PC

Result of a Function

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

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

Result of a Function

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

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

alternative without result value:

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

Result of a Function

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

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

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

Result of a Function

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

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

alternative without result value:

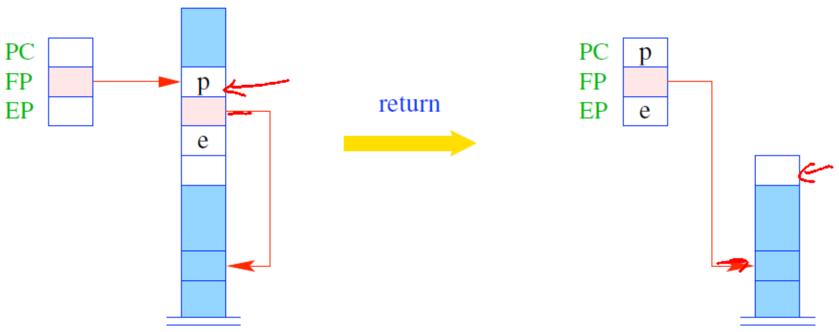
$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

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

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:

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

Assumptions:

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

Translation of Functions

The translation of a function is thus defined as follows:

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

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 $args$

Translation of Functions

The translation of a function is thus defined as follows:

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

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

Are the move instructions always necessary?

Translation of Whole Programs

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

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

Translation of Whole Programs

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

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

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);
}
```

_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: $i = 3$	move R_2 R_1	`x*fac(x-1)` R_3
$i = 3$	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	

Topic:

Variables in Memory

Register versus Memory

so far:

- all variables are stored in registers
- all function parameters and the return value are stored in registers

limitations:

- a real machine has only a finite number of registers
- in C it is possible to take the address of a variable
- arrays cannot be translated due to indexing

Register versus Memory

so far:

- all variables are stored in registers
- all function parameters and the return value are stored in registers

limitations:

- a real machine has only a finite number of registers
- in C it is possible to take the address of a variable
- arrays cannot be translated due to indexing

idea: store variables on the stack

Variables in Memory: L-Value and R-Value

Variables can be used in two different ways.

example: `a[x] = y + 1`

for `y` we need to know the value of the memory cell, for `a[x]` we are interested in the address

r-value of x	=	content of x
l-value of x	=	address of x

compute r- and l-value in register R_i:

$\text{code}_R^i\ e\ \rho$	generates code to compute the r-value of e, given the environment ρ
$\text{code}_L^i\ e\ \rho$	analogously for the l-value

note:
Not every expression has an l-value (e.g.: $x + 1$).

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

accordingly, we define $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ as follows:

- $\rho\, x = \langle G, a\rangle$: variable x is stored at absolute address a
- $\rho\, x = \langle L, a\rangle$: variable x is stored at address $FP + a$
- $\rho\, x = \langle R, a\rangle$: variable x is stored in register R_a

Address Environment

A variable by stored in four different ways:

1. Global: a variable is global
2. Local: a variable is stored on the stack frame
3. Register: a variable is stored in a local register R_i or a global register R_i

accordingly, we define $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ as follows:

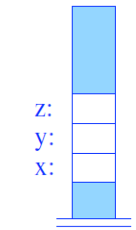
- $\rho\, x = \langle G, a \rangle$: variable x is stored at absolute address a
- $\rho\, x = \langle L, a \rangle$: variable x is stored at address $FP + a$
- $\rho\, x = \langle R, a \rangle$: variable x is stored in register R_a

Observe: a variable x can only have one entry in ρ

However:

- ρ may be change with the program point

Necessity of Storing Variables in Memory

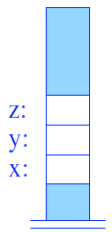


Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$
- separate compilation becomes difficult, since code of function depends on n

Furthermore:

Necessity of Storing Variables in Memory



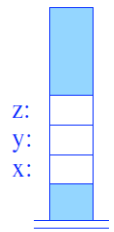
Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

- a variable x (`int` or `struct`) whose address has been taken must be stored in memory, i.e. $\rho\, x = \langle L, o \rangle$ or $\rho\, x = \langle G, o \rangle$

Necessity of Storing Variables in Memory



Global variables:

- could be assigned throughout to registers $R_1 \dots R_n$
- separate compilation becomes difficult, since code of function depends on n
- simple solution: store global variables in memory

Furthermore:

- a variable x (`int` or `struct`) whose address has been taken must be stored in memory, i.e. $\rho\, x = \langle L, o \rangle$ or $\rho\, x = \langle G, o \rangle$
- an access to an array is always done through a pointer, hence, it must be stored in memory
- optimization: store individual elements of a `struct` in register while no pointer accesses may reach the structure

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j \rangle$ then:

$$\begin{aligned} \mathrm{code}^i_\mathrm{R}\ x = e_2\ \rho \quad &= \quad \mathrm{code}^i_\mathrm{R}\ e_2\ \rho \\ &\quad\quad \mathrm{move}\ R_j\ R_i \end{aligned}$$

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j \rangle$ then:

$$\begin{aligned} \mathrm{code}^i_\mathrm{R}\ x = e_2\ \rho \quad &= \quad \mathrm{code}^i_\mathrm{R}\ e_2\ \rho \\ &\quad\quad \mathrm{move}\ R_j\ R_i \end{aligned}$$

but: undefined result if $\rho\ x = \langle L, a \rangle$ or $\rho\ x = \langle G, a \rangle$.

Translation of Statements

Statements such as `x=2*y` have so far been translated by:

- computing the r-value of `2*y` in register R_i,
- copying the content of R_i into the register $\rho(x)$

formally: let $\rho(x) = \langle R, j \rangle$ then:

$$\begin{aligned} \mathrm{code}^i_\mathrm{R}\ x = e_2\ \rho \quad &= \quad \mathrm{code}^i_\mathrm{R}\ e_2\ \rho \\ &\quad\quad \mathrm{move}\ R_j\ R_i \end{aligned}$$

$a[x] = y+1$

but: undefined result if $\rho\ x = \langle L, a \rangle$ or $\rho\ x = \langle G, a \rangle$.

idea:

- compute the r-value of e_2 in register R_i,
- compute the l-value of e_1 in register R_{i+1} and
- write e_2 to address e_1 using a store instruction

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\mathrm{code}^i\ e\ \rho = \mathrm{code}^i_\mathrm{R}\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a \rangle$)?

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\text{code}^i\ e\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a\rangle$)?

- Thus, for the case $e_1 = x$ and $\rho\ x = \langle R, j\rangle$ does *not* hold:

$$\begin{aligned} \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho \quad = \quad & \text{code}^i_{\text{R}}\ e_2\ \rho \\ & \text{code}^{i+1}_{\text{L}}\ e_1\ \rho \\ & \text{store}\ R_{i+1}\ R_i \end{aligned}$$

Translation of L-Values

new instruction: store R_i R_j with semantics $S[R_i] = R_j$

definition for assignments:

$$\text{code}^i\ e\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

So how do we translate $x = e$ (with $\rho\ x = \langle G, a\rangle$)?

- Thus, for the case $e_1 = x$ and $\rho\ x = \langle R, j\rangle$ does *not* hold:

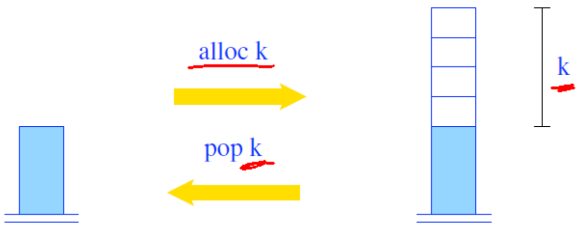
$$\begin{aligned} \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho \quad = \quad & \text{code}^i_{\text{R}}\ e_2\ \rho \\ & \text{code}^{i+1}_{\text{L}}\ e_1\ \rho \\ & \text{store}\ R_{i+1}\ R_i \end{aligned}$$

- The l-value of a variable is computed as follows:

$$\text{code}^i_{\text{L}}\ x\ \rho \quad = \quad \text{loadc}\ R_i\ a$$

Allocating Memory for Local Variables

Given: a function with k local **int** variables that need to be stored in memory.



alloc k	SP = SP + k;
pop k	SP = SP - k;

The instruction alloc k reserves space for k variables on the stack, pop k frees this space again.

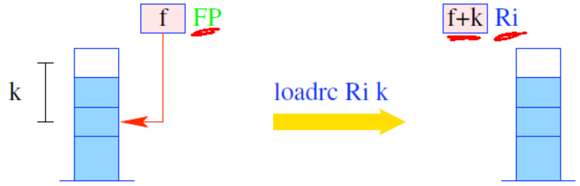
Access to Local Variables

Accesses to local variables are relative to FP. We therefore modify code_{L} to cater for variables in memory.

For $\rho\ x = \langle L, a\rangle$ we define

$$\text{code}^i_{\text{L}}\ x\ \rho = \text{loadrc}\ R_i\ a \text{ if } \rho\ x = \langle L, a\rangle$$

Instruction loadrc R_i k computes the sum of FP and k.



$$R_i = FP + k$$

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}^i_{\text{L}}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\,x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\,x = \langle L, a\rangle \end{cases}$$

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}^i_{\text{L}}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\,x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\,x = \langle L, a\rangle \end{cases}$$

Note: for $\rho\,x = \langle R, j\rangle$ the function code^i_{L} is not defined!

General Computation of the L-Value of a Variable

Computing the address of a variable in R_i is done as follows:

$$\text{code}^i_{\text{L}}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\,x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\,x = \langle L, a\rangle \end{cases}$$

Note: for $\rho\,x = \langle R, j\rangle$ the function code^i_{L} is not defined!

Observations:

- intuitively: a register has no address
- during the compilation the l-value of a register may never be computed
- this requires a case distinction for assignments

Macro-Command for Accessing Local Variables

Define: the command $\text{load}\ R_i\ R_j$ sets R_i to the value at address R_j.

Thus: $\text{loadrc}\ R_i\ a;\ \text{load}\ R_j\ R_i$: sets R_j to x where $\rho\,x = \langle L, a\rangle$.

In general: Load variable x into register R_i:

$$\text{code}^i_{\text{R}}\ x\ \rho = \begin{cases} \text{loada}\ R_i\ a & \text{if } \rho\,x = \langle G, a\rangle \\ \text{loadr}\ R_i\ a & \text{if } \rho\,x = \langle L, a\rangle \\ \text{move}\ R_i\ R_j & \text{if } \rho\,x = \langle R, i\rangle \end{cases}$$

Macro-Command for Accessing Local Variables

Define: the command load $R_i\ R_j$ sets R_i to the value at address R_j.

Thus: loadrc $R_i\ a$; load $R_j\ R_i$: sets R_j to x where $\rho\, x = \langle L, a\rangle$.

In general: Load variable x into register R_i:

$$\mathrm{code}_R^i\ x\ \rho = \begin{cases} \mathrm{loada}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \mathrm{loadr}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \\ \mathrm{move}\ R_i\ R_j & \text{if } \rho\, x = \langle R, i\rangle \end{cases}$$

Analogously: for write operations we define:

$$\begin{aligned} \mathrm{storer}\ a\ R_j &\equiv \mathrm{loadrc}\ R_i\ a \\ &\ \mathrm{store}\ R_i\ R_j \\ \mathrm{storea}\ a\ R_j &\equiv \mathrm{loadc}\ R_i\ a \\ &\ \mathrm{store}\ R_i\ R_j \end{aligned}$$

i.e. storea $a\ R_j$ is a *macro*. Define special case (where $\rho\, x = \langle G, a\rangle$):

$$\begin{aligned} \mathrm{code}_R^i\ x = e_2\ \rho \quad &= \quad \mathrm{code}_R^i\ e_2\ \rho \\ & \quad \mathrm{code}_L^{i+1}\ x\ \rho \\ & \quad \mathrm{store}\ R_{i+1}\ R_i \end{aligned}$$

Macro-Command for Accessing Local Variables

Define: the command load $R_i\ R_j$ sets R_i to the value at address R_j.

Thus: loadrc $R_i\ a$; load $R_j\ R_i$: sets R_j to x where $\rho\, x = \langle L, a\rangle$.

In general: Load variable x into register R_i:

$$\mathrm{code}_R^i\ x\ \rho = \begin{cases} \mathrm{loada}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \mathrm{loadr}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \\ \mathrm{move}\ R_i\ R_j & \text{if } \rho\, x = \langle R, i\rangle \end{cases}$$

Analogously: for write operations we define:

$$\begin{aligned} \mathrm{storer}\ a\ R_j &\equiv \mathrm{loadrc}\ R_i\ a \\ &\ \mathrm{store}\ R_i\ R_j \\ \mathrm{storea}\ a\ R_j &\equiv \mathrm{loadc}\ R_i\ a \\ &\ \mathrm{store}\ R_i\ R_j \end{aligned}$$

i.e. storea $a\ R_j$ is a *macro*. Define special case (where $\rho\, x = \langle G, a\rangle$):

$$\begin{aligned} \mathrm{code}_R^i\ x = e_2\ \rho \quad &= \quad \mathrm{code}_R^i\ e_2\ \rho \\ & \quad \mathrm{storea}\ a\ R_i \end{aligned}$$

Data Transfer Instructions of the R-CMa

read- and write accesses of the R-CMa are as follows:

instruction	semantics	intuition
load $R_i\ R_j$	$R_i \leftarrow S[R_j]$	load value from address
loada $R_i\ c$	$R_i \leftarrow S[c]$	load global variable
loadr $R_i\ c$	$R_i \leftarrow S[FP + c]$	load local variable
store $R_i\ R_j$	$S[R_i] \leftarrow R_j$	store value at address
storea $c\ R_i$	$S[c] \leftarrow R_i$	write global variable
storer $c\ R_i$	$S[FP + c] \leftarrow R_i$	write local variable

instructions for computing addresses:

instruction	semantics	intuition
loadc $R_i\ c$	$R_i \leftarrow c$	load constant
loadrc $R_i\ c$	$R_i \leftarrow FP + c$	load constant relative to FP

instructions for general data transfer:

instruction	semantics	intuition
move $R_i\ R_j$	$R_i \leftarrow R_j$	transfer value between registers
move $R_i\ k\ R_j$	$[S[SP+i] \leftarrow S[R_j+i]]_{i=0}^{k-1}$ $R_i \leftarrow SP; SP \leftarrow SP + k$	copy k values onto the stack

Determining the Address-Environment

variables in the symbol table are tagged in one of three ways:

G 1. global variables, defined outside of functions (or as `static`);
L 2. local (automatic) variables, defined inside functions, accessible by pointers;
R 3. register (automatic) variables, defined inside functions.

Example:

```
int x, y;
void f(int v, int w) {
  int a;
  if (a>0) {
    int b;
    g(&b);
  } else {
    int c;
  }
}
```

v	$\rho(v)$
x	$\langle G, 0\rangle$
y	$\langle G, 1\rangle$
v	$\langle R, -1\rangle$
w	$\langle R, -2\rangle$
a	$\langle R, 1\rangle$
b	$\langle L, 0\rangle$
c	$\langle R, 2\rangle$

Determining the Address-Environment

variables in the symbol table are tagged in one of three ways:

1. global variables, defined outside of functions (or as `static`);
2. local (automatic) variables, defined inside functions, accessible by pointers;
3. register (automatic) variables, defined inside functions.

Example:

```
int x, y;
void f(int v, int w) {
  int a;
  if (a>0) {
    int b;
    g(&b);
  } else {
    int c;
  }
}
```

v	$\rho(v)$
x	⟨ G , 0 ⟩
y	⟨ G , 1 ⟩
v	⟨ R , -1 ⟩
w	⟨ R , -2 ⟩
a	⟨ R , 1 ⟩
b	⟨ L , 0 ⟩
c	⟨ R , 2 ⟩

Function Arguments on the Stack

- C allows for so-called *variadic functions*
- an unknown number of parameters: $R_{-1}, R_{-2}, \ldots$
- problem: callee cannot index into global registers

example:

```
int printf(const char * format, ...);
char *s =
  "Hello␣%s!\nIt's␣%i␣to␣%i!\n";

int main(void) {
  printf(s ,"World", 5, 12);
  return 0;
}
```

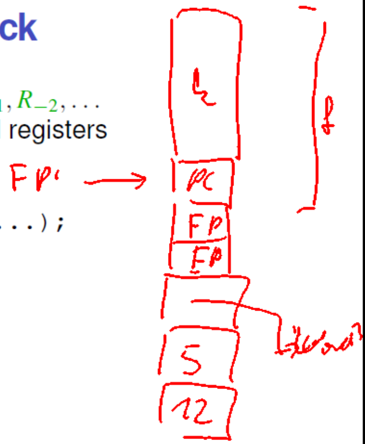

Function Arguments on the Stack

- C allows for so-called *variadic functions*
- an unknown number of parameters: $R_{-1}, R_{-2}, \ldots$
- problem: callee cannot index into global registers

example:

```
int printf(const char * format, ...);
char *s =
  "Hello␣%s!\nIt's␣%i␣to␣%i!\n";

int main(void) {
  printf(s ,"World", 5, 12);
  return 0;
}
```



idea:

- push *variadic* parameters from *right to left* onto the stack
- the first parameter lies right below PC, FP, EP
- for a prototype $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k,\ \ldots)$ we set:

$$x_1 \mapsto \langle R, -1\rangle \qquad x_k \mapsto \langle R, -k\rangle$$

$$x_{k+1} \text{ at } \langle L, -3\rangle \qquad x_{k+i} \text{ at } \langle L, -3 - |\tau_{k+1}| - \ldots - |\tau_{k+i-1}|\rangle$$

Chapter 2:

Arrays and Pointers

Arrays

Example: `int[11] a;`

f() {

int x1;

int x2;

- the array a contains 11 elements and therefore requires 11 cells.
- ρa is the address of `a[0]`.

a[10]
⋮
a[0]

k

x_2 -

x_1

FP → PC

Define the function $|\cdot|$ to compute the required space of a type:

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

For a sequence of declarations $d \equiv t_1\; x_1; \;\ldots\; t_k\; x_k;$ we have:

$$\begin{array}{lll} \rho\, x_1 & = & 1 \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| \qquad \text{für } i > 1 \end{array}$$

$|\cdot|$ can be computed at compile type and, hence, ρ too.
Note: $|\cdot|$ is required to translate the `sizeof` operator in C

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.
In order to compute the address of `a[i]`, we need to compute $\rho\, a + |t| * $ *(R-Wert von i)*. Thus:

$\text{code}_\text{L}^i\; e_1$

$$\begin{array}{lll} \text{code}_\text{L}^i\; e_2[\cancel{e_1}]\; \rho & = & \text{code}_\text{R}^i\; e_1\; \rho \\ & & \text{code}_\text{R}^{i+1}\; e_2\; \rho \\ & & \text{loadc}\; R_{i+2}\; |t| \\ & & \text{mul}\; R_{i+1}\; R_{i+1}\; R_{i+2} \\ & & \text{add}\; R_i\; R_i\; R_{i+1} \end{array}$$

$e_1[e_2]$

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.
In order to compute the address of `a[i]`, we need to compute $\rho\, a + |t| * $ *(R-Wert von i)*. Thus:

$$\begin{array}{lll} \text{code}_\text{L}^i\; e_2[e_1]\; \rho & = & \text{code}_\text{R}^i\; e_1\; \rho \\ & & \text{code}_\text{R}^{i+1}\; e_2\; \rho \\ & & \text{loadc}\; R_{i+2}\; |t| \\ & & \text{mul}\; R_{i+1}\; R_{i+1}\; R_{i+2} \\ & & \text{add}\; R_i\; R_i\; R_{i+1} \end{array}$$

Note:

- An array in C is simply a *pointer*. The declared array `a` is a *pointer constant*, whose r-value is address of the first field of `a`.
- Formally, we compute the r-value of a field `e` as $\text{code}_\text{R}^i\; e\; \rho = \text{code}_\text{L}^i\; e\; \rho$
- in C the following are equivalent (as l-value, not as types):

`2[a]` `a[2]` `a+2`

C `structs` (Records)

Note:
The same field name may occur in different `structs`
Here: The component environment ρ_{st} relates to the currently translated structure st.
Let `struct { int a; int b; } x;` be part of a declaration list.

- `x` is a variable of the size of (at least) the sum of the sizes of its fields
- we populate ρ_{st} with addresses of fields that are *relative* to the beginning of `x`, here $a \mapsto 0$, $b \mapsto 1$.

C `structs` (Records)

Note:
The same field name may occur in different `structs`
Here: The component environment ρ_{st} relates to the currently translated structure st.
Let `struct { int a; int b; } x;` be part of a declaration list.

- `x` is a variable of the size of (at least) the sum of the sizes of its fields
- we populate ρ_{st} with addresses of fields that are *relative* to the beginning of `x`, here $a \mapsto 0$, $b \mapsto 1$.

In general, let $t \equiv$ `struct` $\{ \; t_1 \; v_1; \dots ; \; t_k \; v_k \; \}$, then

$$|t| := \sum_{i=1}^{k} |t_i| \qquad \rho_{st} \; v_1 := 0 \qquad \rho_{st} \; v_i := \rho_{st} \; v_{i-1} + |t_{i-1}| \quad \text{für } i > 1$$

We obtain:

$$\begin{aligned} \text{code}^i_{\text{L}} \; (e.c) \; \rho \quad = \quad & \text{code}^i_{\text{L}} \; e \; \rho \\ & \text{loadc} \; R_{i+1} \; (\rho_{st} \; c) \\ & \text{add} \; R_i \; R_i \; R_{i+1} \end{aligned}$$

Pointer in C

Computing with pointers means

1. to create pointers, that is, to obtain the address of a variable;
2. to dereference pointers, that is, to access the pointed-to memory

Creating pointers:

- through the use of the address-of operator: `&` yields a pointer to a variable, that is, its (≙l-value). Thus define:

$$\text{code}^i_{\text{R}} \; \&e \; \rho \quad = \quad \text{code}^i_{\text{L}} \; e \; \rho$$

Example:
Let `struct { int a; int b; } x;` with $\rho = \{x \mapsto 13\}$ and $\rho_{st} = \{a \mapsto 0, b \mapsto 1\}$.
Then

$$\begin{aligned} \text{code}^i_{\text{L}} \; (x.b) \; \rho \quad = \quad & \text{loadc} \; R_{i+1} \; 13 \\ & \text{loadc} \; R_i \; 1 \\ & \text{add} \; R_i \; R_i \; R_{i+1} \end{aligned}$$

Dereferencing Pointers

Applying the $*$ operator to an expression e yields the content of the cell whose l-value is stored in e:

$$\begin{aligned} \text{code}^i_{\text{R}} \; {*e} \; \rho \quad = \quad & \text{code}^i_{\text{L}} \; e \; \rho \\ & \text{load} \; R_i \; R_i \end{aligned}$$

Example: Consider

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

and the expression $e \equiv$ `((pt -> b) -> a)[i+1]`

Since `e->a` $\equiv$ `(*e).a` we get:

$$\begin{aligned} \text{code}^i_{\text{L}} \; (e \to a) \; \rho \quad = \quad & \text{code}^i_{\text{L}} \; e \; \rho \\ & \text{loadc} \; R_{i+1} \; (\rho \; a) \\ & \text{add} \; R_i \; R_i \; R_{i+1} \end{aligned}$$

Translation of Array Accesses

Extend code_L and code_R with indexed array accesses.

Let `t[c] a;` be the declaration of an array a.
In order to compute the address of `a[i]`, we need to compute $\rho\, a + |t| * (\textit{R-Wert von } i)$. Thus:

sizeof(a)/sizeof(a[0])

$$
\begin{aligned}
\text{code}_L^i\ e_2[e_1]\ \rho \quad = \quad & \text{code}_{\cancel{R}}^{iL}\ e_1\ \rho \\
& \text{code}_R^{i+1}\ e_2\ \rho \\
& \text{loadc}\ R_{i+2}\ |t| \\
& \text{mul}\ R_{i+1}\ R_{i+1}\ R_{i+2} \\
& \text{add}\ R_i\ R_i\ R_{i+1}
\end{aligned}
$$

$\&a[0] \equiv a$

Note:

- An array in C is simply a *pointer*. The declared array `a` is a *pointer constant*, whose r-value is address of the first field of `a`.
- Formally, we compute the r-value of a field `e` as $\cancel{\text{code}_R^i\ e\ \rho = \text{code}_L^i}\ e\ \rho$
- in C the following are equivalent (as l-value, not as types):
  `2[a]` `a[2]` `a+2`

C `structs` (Records)

Note:
The same field name may occur in different `structs`
Here: The component environment ρ_{st} relates to the currently translated structure st.
Let `struct { int a; int b; } x;` ~~be part~~ of a declaration list.

- `x` is a variable of the size of (at least) the sum of the sizes of its fields
- we populate ρ_{st} with addresses of fields that are *relative* to the beginning of `x`, here $a \mapsto 0, b \mapsto 1$.

In general, let $t \equiv$ `struct` $\{\ t_1\ v_1; \ldots ;\ t_k\ v_k\ \}$, then

$$|t| := \sum_{i=1}^{k} |t_i| \qquad \rho_{st}\ v_1 := 0 \qquad \rho_{st}\ v_i := \rho_{st}\ v_{i-1} + |t_{i-1}| \quad \text{für } i > 1$$

We obtain:

$$
\begin{aligned}
\text{code}_L^i\ (e.c)\ \rho \quad = \quad & \text{code}_L^i\ e\ \rho \\
& \text{loadc}\ R_{i+1}\ (\rho_{st}\ c) \\
& \text{add}\ R_i\ R_i\ R_{i+1}
\end{aligned}
$$

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

$\rightsquigarrow$ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

$\rightsquigarrow$ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

$$
\begin{aligned}
\text{code}_R^i\ e\ \rho \quad = \quad & \text{code}_L^{i+1}\ e\ \rho \\
& \text{move}\ R_i\ k\ R_{i+1} \qquad e \text{ a structure of size } k
\end{aligned}
$$

Passing Compound Parameters

Consider the following declarations:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

640 k

$\leadsto$ How do we pass parameters that are not basis types?

- idea: *caller* passes a pointer to the structure
- problem: *callee* could modify the structure
- solution: *caller* passes a pointer to a copy

$$\text{code}_{\text{R}}^{i}\ e\ \rho \quad = \quad \begin{array}{l}\text{code}_{\text{L}}^{i+1}\ e\ \rho \\ \text{move}\ R_i\ k\ R_{i+1}\end{array} \qquad e \text{ a structure of size } k$$

new instruction: move

Invariant of Heap and Stack

- the stack and the heap may not overlap

Possible Implementations of `free`

1. Leave the problem of dangling pointers to the programmer. Use a data structure to manage allocated and free memory. $\leadsto$ `malloc` becomes expensive
2. Do nothing:

$$\text{code}^{i}\ \textbf{free}(e)\ \rho \quad = \quad \text{code}_{\text{R}}^{i}\ e\ \rho$$

$\leadsto$ simple and efficient, but not for reactive programs

3. Use an automatic, possibly "conservative" garbage collection, that occasionally runs to reclaim memory that certainly is not in use anymore. Make this re-claimed memory available again to `malloc`.

Translation of Programs

Before the execution of a program, the runtime sets:

$\text{SP} = -1 \qquad \text{FP} = \text{EP} = 0 \qquad \text{PC} = 0 \qquad \text{NP} = \text{MAX}$

Let $p \equiv V\_defs \quad F\_def_1 \ldots F\_def_n$ be a program where F_def_i defines a function f_i of which one is called `main`.

The code for the program p is comprised of:

- code for each function definition F_def_i;
- code to initialize global variables
- code that calls `main()`
- an instruction halt.

Instructions for Starting a Program

A program $P = F_1; \ldots F_n$ has to have one `main` function.

$$
\begin{array}{rcl}
\text{code}^1\ P\ \rho & = & \text{enter}\ (k+3) \\
 & & \text{alloc}\ k \\
 & & \text{loadc}\ R_1\ \texttt{\_main} \\
 & & \text{saveloc}\ R_1\ R_0 \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{restoreloc}\ R_1\ R_0 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^i\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^i\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Instructions for Starting a Program

A program $P = F_1; \ldots F_n$ has to have one `main` function.

$$
\begin{array}{rcl}
\text{code}^1\ P\ \rho & = & \text{enter}\ (k+3) \\
 & & \text{alloc}\ k \\
 & & \text{loadc}\ R_1\ \texttt{\_main} \\
 & & \text{saveloc}\ R_1\ R_0 \\
 & & \text{mark} \\
 & & \text{call}\ R_1 \\
 & & \text{restoreloc}\ R_1\ R_0 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^i\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^i\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

assumptions:

- k are the number of stack location set aside for global variables
- saveloc R_1 R_0 has no effect (i.e. it backs up no register)
- ρ contains the address of all functions and global variable

Translation of Functions

The translation of a function is modified as follows:

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

Randbedinungen:

- enter ensures that enough stack space is available (q: number of required stack cells)

Translation of Functions

The translation of a function is modified as follows:

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

Randbedinungen:

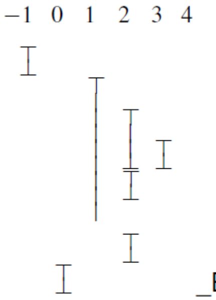
- enter ensures that enough stack space is available (q: number of required stack cells)
- alloc reserves space on the stack for local variables ($k < q$)

Register Coloring for the `fac`-Function

Note: def-use liveness

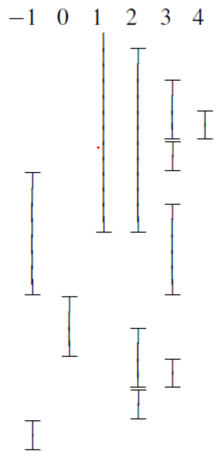
```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

_fac:
enter 5
move R_1 R_{-1}
move R_2 R_1
loadc R_3 0
leq R_2 R_2 R_3
jumpz R_2 $\_A$
loadc R_2 1
move R_0 R_2
return
jump $\_B$



_A:
move R_2 R_1
move R_3 R_1
loadc R_4 1
sub R_3 R_3 R_4
move R_{-1} R_3
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

_B:
return

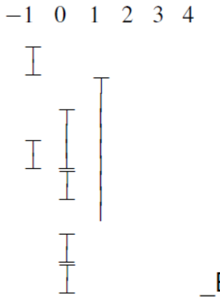


Register Coloring for the `fac`-Function

Note: def-use liveness coloring

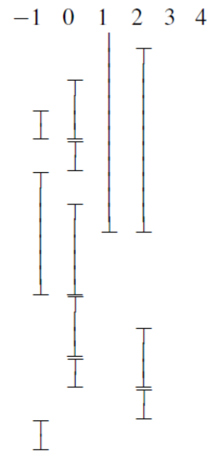

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```


_fac:
enter 5
move R_1 R_{-1}
move R_0 R_1
loadc R_{-1} 0
leq R_0 R_0 R_{-1}
jumpz R_2 $\_A$
loadc R_2 1
move R_0 R_2
return
jump $\_B$



_A:
move R_2 R_1
move R_0 R_1
loadc R_{-1} 1
sub R_0 R_0 R_{-1}
move R_{-1} R_0
loadc R_3 $\_fac$
saveloc R_1 R_2
mark
call R_3
restoreloc R_1 R_2
move R_0 R_0
mul R_2 R_2 R_0
move R_0 R_2
return

_B:
return



Outlook

register allocation has several other uses:

- remove unnecessary move instructions