Topic:

Code Synthesis

Generating Code: Overview

We inductively generate instructions from the AST:

- there is a rule stating how to generate code for each non-terminal of the grammar
- the code is merely another attribute in the syntax tree
- code generation makes use of the already computed attributes

In order to specify the code generation, we require

- a semantics of the language we are compiling (here: C standard)
- a semantics of the machine instructions

$\rightsquigarrow$ we commence by specifying machine instruction semantics

Chapter 1:
The Register C-Machine

The Register C-Machine (R-CMa)

We generate Code for the Register C-Machine.
The Register C-Machine is a virtual machine (VM).

- there exists no processor that can execute its instructions
- . . . but we can build an interpreter for it
- we provide a visualization environment for the R-CMa
- the R-CMa has no **`double`**, **`float`**, **`char`**, **`short`** or **`long`** types
- the R-CMa has no instructions to communicate with the operating system
- the R-CMa has an unlimited supply of registers

The R-CMa is more realistic than it may seem:

- the mentioned restrictions can easily be lifted
- the *Dalvik VM/ART* or the *LLVM* are similar to the R-CMa
- an interpreter of R-CMa can run on any platform

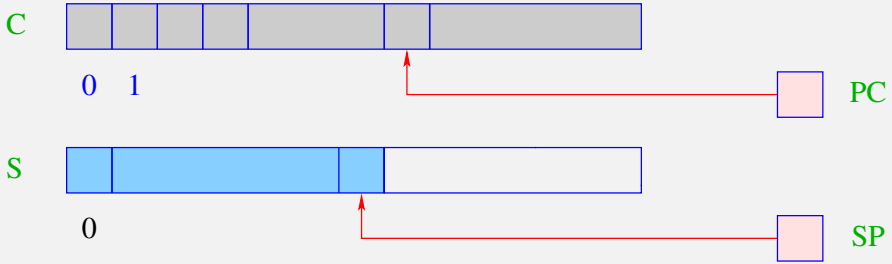
Virtual Machines

A virtual machine has the following ingredients:

- any virtual machine provides a set of instructions
- instructions are executed on virtual hardware
- the virtual hardware is a collection of data structures that is accessed and modified by the VM instructions
- ... and also by other components of the run-time system, namely functions that go beyond the instruction semantics
- the interpreter is part of the run-time system

Components of a Virtual Machine

Consider Java as an example:



A virtual machine such as the Dalvik VM has the following structure:

- S: the data store – a memory region in which cells can be stored in LIFO order $\rightsquigarrow$ stack.
- SP: ($\widehat{=}$ stack pointer) pointer to the last used cell in S
- beyond S follows the memory containing the heap
- C is the memory storing *code*
  - each cell of C holds exactly one virtual instruction
  - C can only be *read*
- PC ($\widehat{=}$ program counter) address of the instruction that is to be executed next
- PC contains 0 initially

Executing a Program

- the machine loads an instruction from C[PC] into the instruction register IR in order to execute it
- before evaluating the instruction, the PC is incremented by one

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

- node: the PC must be incremented before the execution, since an instruction may modify the PC
- the loop is exited by evaluating a halt instruction that returns directly to the operating system

Chapter 2:
Generating Code for the Register C-Machine

Simple Expressions and Assignments in R-CMa

Task: evaluate the expression $(1 + 7) * 3$
that is, generate an instruction sequence that

- computes the value of the expression and
- keeps its value accessible in a reproducable way

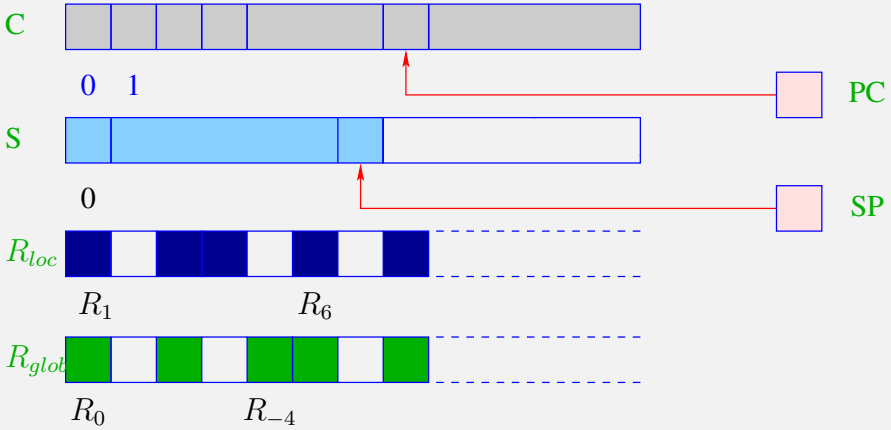
Idea:

- first compute the value of the sub-expressions
- store the intermediate result in a temporary register
- apply the operator
- loop

Principles of the R-CMa

The R-CMa is composed of a stack, heap and a code segment, just like the JVM; it additionally has register sets:

- *local* registers are $R_1, R_2, \ldots R_i, \ldots$
- *global* register are $R_0, R_{-1}, \ldots R_j, \ldots$



The Register Sets of the R-CMa

The two register sets have the following purpose:

1. the *local* registers R_i
   - save temporary results
   - store the contents of local variables of a function
   - can efficiently be stored and restored from the stack
2. the *global* registers R_i
   - save the parameters of a function
   - store the result of a function

Note:
for now, we only use registers to store temporary computations

Idea for the translation: use a register counter i:

- registers R_j with $j < i$ are *in use*
- registers R_j with $j \geq i$ are *available*

Translation of Simple Expressions

Using variables stored in registers; loading constants:

instruction	semantics	intuition
loadc R_i c	$R_i = c$	load constant
move R_i R_j	$R_i = R_j$	copy R_j to R_i

We define the following translation schema (with $\rho\ x = a$):

$$
\begin{aligned}
\mathrm{code}_{\mathrm{R}}^{i}\ c\ \rho &= \text{loadc } R_i\ c \\
\mathrm{code}_{\mathrm{R}}^{i}\ x\ \rho &= \text{move } R_i\ R_a \\
\mathrm{code}_{\mathrm{R}}^{i}\ x = e\ \rho &= \mathrm{code}_{\mathrm{R}}^{i}\ e\ \rho \\
&\quad\ \text{move } R_a\ R_i
\end{aligned}
$$

Translation of Expressions

Let op $= \{add,\ sub,\ div,\ mul,\ mod,\ le,\ gr,\ eq,\ leq,\ geq,\ and,\ or\}$. The R-CMa provides an instruction for each operator op.

$$\text{op}\ \ R_i\ R_j\ R_k$$

where R_i is the target register, R_j the first and R_k the second argument.

Correspondingly, we generate code as follows:

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

Example: Translate $3 \star 4$ with $i = 4$:

$$\begin{aligned} \text{code}^4_{\text{R}}\ \ 3 \star 4\ \ \rho \quad = \quad & \text{code}^4_{\text{R}}\ \ 3\ \ \rho \\ & \text{code}^5_{\text{R}}\ \ 4\ \ \rho \\ \text{code}^4_{\text{R}}\ \ 3 \star 4\ \ \rho \quad = \quad & \text{loadc}\ R_4\ 3 \\ & \text{loadc}\ R_5\ 4 \\ & \text{mul}\ R_4\ R_4\ R_5 \end{aligned}$$

Managing Temporary Registers

Observe that temporary registers are re-used: translate `3*4+3*4` with $t = 4$:

$$\begin{array}{lcl} \mathrm{code}_{\mathrm{R}}^{4}\ \texttt{3*4+3*4}\ \rho & = & \mathrm{code}_{\mathrm{R}}^{4}\ \texttt{3*4}\ \rho \\ & & \mathrm{code}_{\mathrm{R}}^{5}\ \texttt{3*4}\ \rho \\ & & \mathrm{add}\ R_4\ R_4\ R_5 \end{array}$$

where

$$\begin{array}{lcl} \mathrm{code}_{\mathrm{R}}^{i}\ \texttt{3*4}\ \rho & = & \mathrm{loadc}\ R_i\ 3 \\ & & \mathrm{loadc}\ R_{i+1}\ 4 \\ & & \mathrm{mul}\ R_i\ R_i\ R_{i+1} \end{array}$$

we obtain

$$\begin{array}{lcl} \mathrm{code}_{\mathrm{R}}^{4}\ \texttt{3*4+3*4}\ \rho & = & \mathrm{loadc}\ R_4\ 3 \\ & & \mathrm{loadc}\ R_5\ 4 \\ & & \mathrm{mul}\ R_4\ R_4\ R_5 \\ & & \mathrm{loadc}\ R_5\ 3 \\ & & \mathrm{loadc}\ R_6\ 4 \\ & & \mathrm{mul}\ R_5\ R_5\ R_6 \\ & & \mathrm{add}\ R_4\ R_4\ R_5 \end{array}$$

Semantics of Operators

The operators have the following semantics:

add R_i R_j R_k	$R_i = R_j + R_k$
sub R_i R_j R_k	$R_i = R_j - R_k$
div R_i R_j R_k	$R_i = R_j / R_k$
mul R_i R_j R_k	$R_i = R_j * R_k$
mod R_i R_j R_k	$R_i = signum(R_k) \cdot k$ with $\lvert R_j \rvert = n \cdot \lvert R_k \rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k \rvert$
le R_i R_j R_k	$R_i =$ if $R_j < R_k$ then 1 else 0
gr R_i R_j R_k	$R_i =$ if $R_j > R_k$ then 1 else 0
eq R_i R_j R_k	$R_i =$ if $R_j = R_k$ then 1 else 0
leq R_i R_j R_k	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq R_i R_j R_k	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and R_i R_j R_k	$R_i = R_j \,\&\, R_k$ // bit-wise and
or R_i R_j R_k	$R_i = R_j \mid R_k$ // bit-wise or

Note: all registers and memory cells contain operands in $\mathbb{Z}$

Translation of Unary Operators

Unary operators op $= \{neg,\ not\}$ take only two registers:

$$\text{code}_{\text{R}}^{i}\ \text{op}\ e\ \rho \quad = \quad \begin{array}{l}\text{code}_{\text{R}}^{i}\ e\ \rho \\ \text{op}\ R_i\ R_i\end{array}$$

Note: We use the same register.

Example: Translate -4 into R_5:

$$\begin{array}{lcl}\text{code}_{\text{R}}^{5}\ -4\ \rho & = & \text{code}_{\text{R}}^{5}\ \ 4\ \ \rho \\ \text{code}_{\text{R}}^{5}\ -4\ \rho & = & \text{loadc}\ R_5\ 4 \\ & & \text{neg}\ R_5\ R_5\end{array}$$

The operators have the following semantics:

$$\begin{array}{ll}\text{not}\ R_i\ R_j & R_i \leftarrow \text{if}\ R_j = 0\ \text{then}\ 1\ \text{else}\ 0 \\ \text{neg}\ R_i\ R_j & R_i \leftarrow -R_j\end{array}$$

Applying Translation Schema for Expressions

Suppose the following function is given:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Let $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ be the address environment.
- Let R_4 be the first free register, that is, $i = 4$.

$$
\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}_\text{R}^4\ \texttt{y+z*3}\ \rho \\
&\quad \text{move}\ R_1\ R_4 \\
\text{code}_\text{R}^4\ \texttt{y+z*3}\ \rho &= \text{move}\ R_4\ R_2 \\
&\quad \text{code}_\text{R}^5\ \texttt{z*3}\ \rho \\
&\quad \text{add}\ R_4\ R_4\ R_5 \\
\text{code}_\text{R}^5\ \texttt{z*3}\ \rho &= \text{move}\ R_5\ R_3 \\
&\quad \text{code}_\text{R}^6\ \texttt{3}\ \rho \\
&\quad \text{mul}\ R_5\ R_5\ R_6 \\
\text{code}_\text{R}^6\ \texttt{3}\ \rho &= \text{loadc}\ R_6\ 3
\end{aligned}
$$

$\rightsquigarrow$ the assignment `x=y+z*3` is translated as

$\text{move}\ R_4\ R_2; \text{move}\ R_5\ R_3; \text{loadc}\ R_6\ 3; \text{mul}\ R_5\ R_5\ R_6; \text{add}\ R_4\ R_4\ R_5; \text{move}\ R_1\ R_4$

Chapter 3:
Statements and Control Structures

About Statements and Expressions

General idea for translation:

$\text{code}^i\ s\ \rho$: generate code for statement s

$\text{code}^i_{\text{R}}\ e\ \rho$: generate code for expression e into R_i

Throughout: $i, i+1, \ldots$ are free (unused) registers

For an *expression* $x = e$ with $\rho\ x = a$ we defined:

$$\begin{aligned} \text{code}^i_{\text{R}}\ x = e\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e\ \rho \\ &\quad\ \ \text{move}\ R_a\ R_i \end{aligned}$$

However, $x = e;$ is also an *expression statement*:

- Define:

$$\text{code}^i\ e_1 = e_2;\ \rho \quad = \quad \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho$$

The temporary register R_i is ignored here. More general:

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

- Observation: the assignment to e_1 is a side effect of the evaluating the expression $e_1 = e_2$.

Translation of Statement Sequences

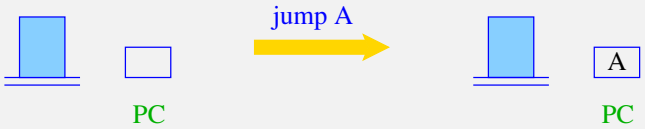
The code for a sequence of statements is the concatenation of the instructions for each statement in that sequence:

$$
\begin{array}{lcll}
\text{code}^i\ (s\ ss)\ \rho & = & \text{code}^i\ s\ \rho & \\
 & & \text{code}^i\ ss\ \rho & \\
\text{code}^i\ \varepsilon\ \rho & = & // & \textit{empty sequence of instructions}
\end{array}
$$

Note here: s is a statement, ss is a sequence of statements

Jumps

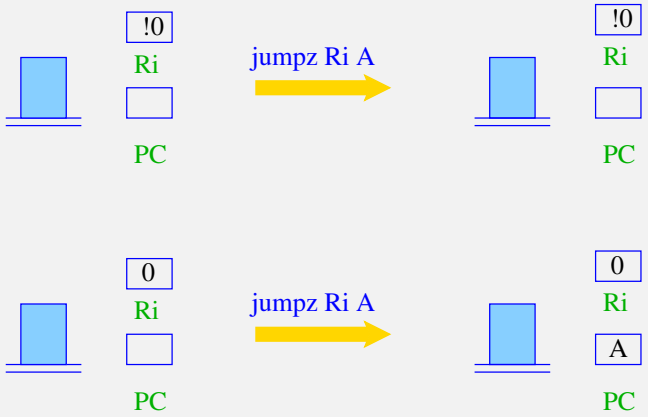
In order to diverge from the linear sequence of execution, we need *jumps*:



PC = A;

Conditional Jumps

A conditional jump branches depending on the value in R_i:



if (R_i == 0) PC = A;

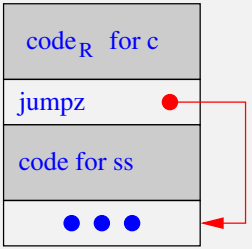
Simple Conditional

We first consider $s \equiv \texttt{if}\ \ (\ c\ )\ \ \ ss$.
...and present a translation without basic blocks.

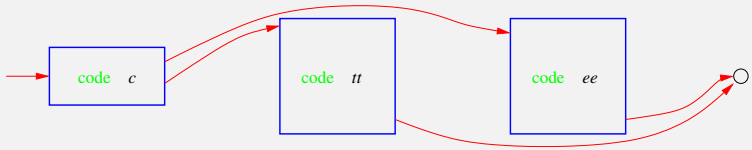
Idea:

- emit the code of c and ss in sequence
- insert a jump instruction in-between, so that correct control flow is ensured

$$
\begin{aligned}
\text{code}^i\ \ s\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ c\ \rho \\
&\qquad\ \ \text{jumpz}\ R_i\ A \\
&\qquad\ \ \text{code}^i\ ss\ \rho \\
&\ \ \text{A}: \quad \ldots
\end{aligned}
$$



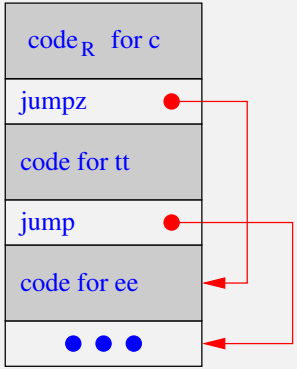
General Conditional



Translation of **if** (c) tt **else** ee.

$$\text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho\ =$$

$$\begin{array}{ll} & \text{code}^i_{\mathrm{R}}\ c\ \rho \\ & \text{jumpz}\ R_i\ A \\ & \text{code}^i\ tt\ \rho \\ & \text{jump}\ B \\ A: & \text{code}^i\ ee\ \rho \\ B: & \end{array}$$



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)

move R_i R_4
move R_{i+1} R_7
gr R_i R_i R_{i+1}
jumpz R_i A

(ii)

move R_i R_4
move R_{i+1} R_7
sub R_i R_i R_{i+1}
move R_4 R_i
jump B

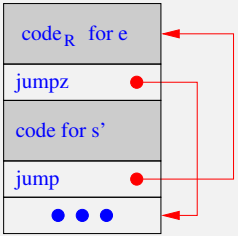
(iii)

A : move R_i R_7
move R_{i+1} R_4
sub R_i R_i R_{i+1}
move R_7 R_i
B :

Iterating Statements

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

$$
\begin{aligned}
\text{code}^i\ \texttt{while}(e)\ s\ \rho \quad = A: \quad & \text{code}^i_{\text{R}}\ e\ \rho \\
& \text{jumpz}\ R_i\ B \\
& \text{code}^i\ s\ \rho \\
& \text{jump}\ A \\
B: \quad &
\end{aligned}
$$



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 $e_1;\ \mathbf{while}\ (e_2)\ \{s'\ e_3;\}$ – as long as s' does not contain a **continue** statement.

Thus, we translate:

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

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

PC = A + R_i;

Consecutive Alternatives

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

```
switch (e) {
  case 0: s_0; break;
  ⋮
  case k − 1: s_{k−1}; break;
  default: s_k; break;
}
```

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

$$
\begin{array}{lll}
\text{code}^i\ s\ \rho = & \text{code}^i_{\text{R}}\ e\ \rho & \\
& \mathit{check}^i\ 0\ k\ B & B: \quad \text{jump}\ A_0 \\
A_0: & \text{code}^i\ s_0\ \rho & \vdots \qquad \vdots \\
& \text{jump}\ C & \qquad\ \ \text{jump}\ A_k \\
\vdots & \vdots & C: \\
A_k: & \text{code}^i\ s_k\ \rho & \\
& \text{jump}\ C &
\end{array}
$$

$\mathit{check}^i\ l\ u\ B$ 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 A_k

we define:

$$
\begin{array}{lll}
check^i\ l\ u\ B & = & \text{loadc } R_{i+1}\ l \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ E \\
 & & \text{sub } R_i\ R_i\ R_{i+1} \\
 & & \text{loadc } R_{i+1}\ k \\
 & & \text{geq } R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz } R_{i+2}\ D \\
E: & & \text{loadc } R_i\ k \\
D: & & \text{jumpi } R_i\ B
\end{array}
\qquad
\begin{array}{ll}
B: & \text{jump } A_0 \\
\vdots & \vdots \\
 & \text{jump } A_k \\
C: &
\end{array}
$$

Note: a jump jumpi R_i B with $R_i = u$ winds up at $B + u$, the default case

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 $check$

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
- 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

Chapter 4:

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:

$$\mathrm{code}_{\mathrm{R}}^{i}\ f\ \rho \quad = \quad \mathrm{loadc}\ \ R_i\ \_f \qquad \text{with } \_f \text{ starting address of } f$$

Observe:

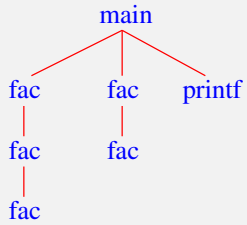
- function names must have an address assigned to them
- since the size of functions is unknown before they are translated, the addresses of forward-declared functions must be inserted later

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 instances 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

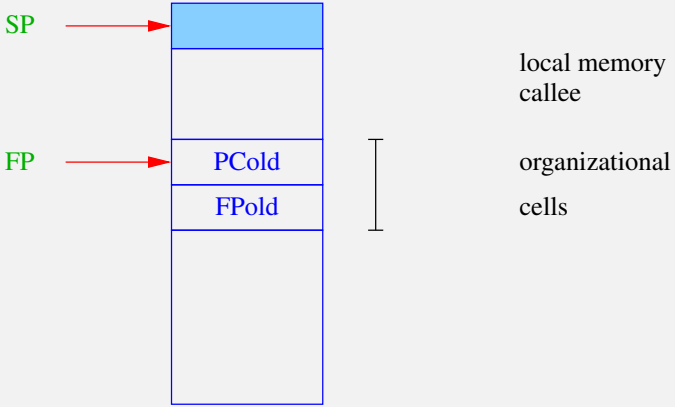
The formal parameters and the local variables of the various instances of a function must be kept separate

Idea for implementing functions:

- set up a region of memory each time it is called
- in sequential programs this memory region can be allocated 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 $\widehat{=}$ frame pointer: points to the last organizational cell
- used to recover the previously active stack frame

Split of Obligations

Definition

Let f be the current function that calls a function g.

- f is dubbed *caller*
- g is dubbed *callee*

The code for managing function calls has to be split between caller and callee.
This split cannot be done arbitrarily since some information is only known in that caller or only in the callee.

Observation:

The space requirement for parameters is only know by the caller:
Example: `printf`

Principle of Function Call and Return

actions taken on entering g:

1. compute the start address of g
2. compute actual parameters in globals
3. backup of caller-save registers
4. backup of FP
5. set the new FP
6. back up of PC and jump to the beginning of g
7. copy actual params to locals

saveloc, mark, call are in f; ... is in g

actions taken on leaving g:

1. compute the result into R_0
2. restore FP, SP
3. return to the call site in f, that is, restore PC
4. restore the caller-save registers

return are in g; restoreloc is 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 live in *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 R_i, and g must restore it before returning

Translation of Function Calls

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

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

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 the FP

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


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

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:

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

alternative without result value:

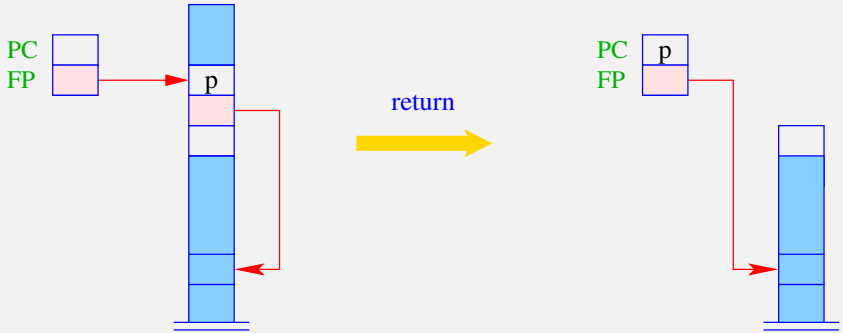
$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

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 and FP.




```
PC = S[FP];
SP = FP-2;
FP = S[SP+1];
```

Translation of Functions

The translation of a function is thus defined as follows:

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

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.

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

Assumptions:

- $\rho = \emptyset$ assuming that we have no global variables
- ρ_{f_i} contain the addresses of the functions up to f_i
- $\rho_1 \oplus \rho_2 = \lambda x\,.\begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$

Translation of the `fac`-function

Consider:

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

```
_fac:   move R1 R-1      save param.
i = 2   move R2 R1       if (x<=0)
        loadc R3 0
        leq R2 R2 R3
        jumpz R2 _A      to else
        loadc R2 1       return 1
        move R0 R2
        return
        jump _B          code is dead
```

```
_A:     move R2 R1       x*fac(x-1)
i = 3   loadc R3 _fac
i = 4   move R4 R1       x-1
i = 5   loadc R5 1
i = 6   sub R4 R4 R5
i = 5   move R-1 R4      fac(x-1)
i = 3   saveloc R1 R2
        mark
        call R3
        restoreloc R1 R2
        move R3 R0
i = 4   mul R2 R2 R3
i = 3   move R0 R2       return x*...
        return
_B:     return
```