Script **generated by TTT**

Title: Petter: Compilerbau (13.07.2015)

Date: Mon Jul 13 14:16:53 CEST 2015

Duration: 65:05 min

Pages: 35

The switch-Statement

Idea:

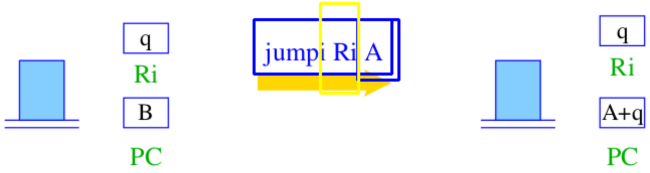
- Suppose choosing from multiple options in *constant time* if possible
- use a *jump table* that, at the ith position, holds a jump to the ith alternative
- in order to realize this idea, we need an *indirect jump* instruction

---

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;
}
```

Consecutive Alternatives

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

switch (e) {
  case 0: s_0; **break**;
  $\vdots$
  case $k-1$: s_{k-1}; **break**;
  default: s_k; **break**;
}

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

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

$$\begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_k \\ C: & \end{array}$$

Consecutive Alternatives

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

switch (e) {
  case 0: s_0; **break**;
  $\vdots$
  case $k-1$: s_{k-1}; **break**;
  default: s_k; **break**;
}

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

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

$$\begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_k \\ C: & \end{array}$$

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

Translation of the check^i Macro

The macro $\mathit{check}^i\ l\ u\ B$ 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~~ $B + u = B + k$

$$\begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_k \\ C: & \end{array}$$

Translation of the check^i Macro

The macro $\mathit{check}^i\ l\ u\ B$ 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:

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

$$\begin{array}{rl} B: & \text{jump}\ A_0 \\ \vdots & \vdots \\ & \text{jump}\ A_k \\ C: & \end{array}$$

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:

```
check^i l u B  =     loadc R_{i+1} 0
                     geq R_{i+2} R_i R_{i+1}        B :  jump A_0
                     jumpz R_{i+2} E
                     sub R_i R_i R_{i+1}              ⋮       ⋮
                     loadc R_{i+1} u
                     geq R_{i+2} R_i R_{i+1}              jump A_k
                     jumpz R_{i+2} D                C :
                 E : loadc R_i (u − l)
                 D : jumpi R_i B
```

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 not omit $check$ completely

---

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

---

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:

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

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

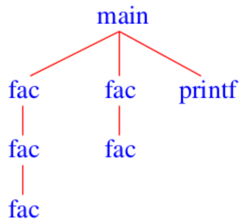
---

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

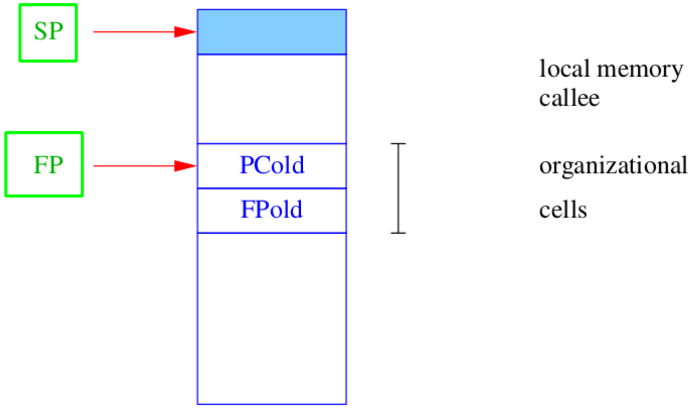
The formal parameters and the local variables of the various instances of a function must be kept separate

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



Split of Obligations

void f (~~void~~ int);

void f (a, b) int a, int b,

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`

f(42, " "); ✓

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

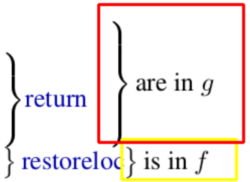
} call

} ... } is in g

} are in f

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

} restoreloc } is in f

} are in g

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:

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:

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

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

+1

✓

✓

Translation of Function Calls

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

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

New instructions:

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

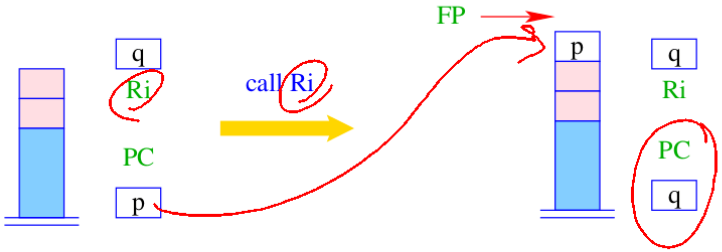
Rescuing EP and FP

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~~1] = 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:

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

Result of a Function

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

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

alternative without result value:

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

Result of a Function

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

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

alternative without result value:

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

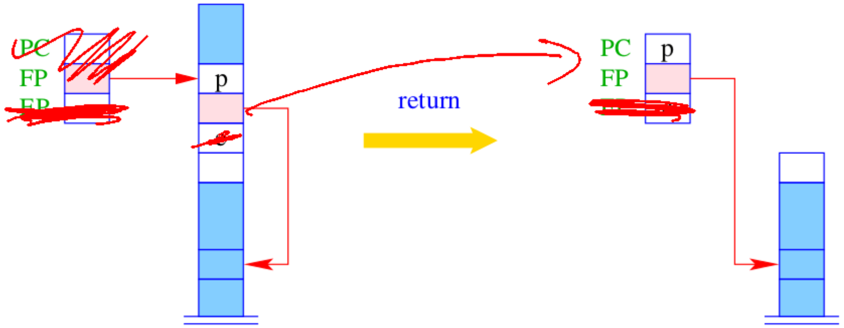
global registers are otherwise not used inside a function body:

- 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, 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:

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

Assumptions:

Translation of Functions

The translation of a function is thus defined as follows:

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

Assumptions:

- the function has n parameters

Translation of Functions

The translation of a function is thus defined as follows:

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

Assumptions:

- the function has n parameters
- the local variables are stored in registers $R_1, \dots R_l$
- the parameters of the function are in $R_{-1}, \dots R_{-n}$
- ρ' is obtained by extending ρ with the bindings in $decls$ and the function parameters $args$

Translation of Whole Programs

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

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

Translation of 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:	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	

End of presentation. Click to exit.