Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier

{lakhnech,mounier}@imag.fr

http://www-verimag.imag.fr/~lakhnech

http://www-verimag.imag.fr/~mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)

Grenoble Universités

(Université Joseph Fourier, Grenoble INP)

Code Generation

Overview

1. Introduction
2. The “M” Machine
3. Code generation for basic `while`
4. Extension 1: blocks and procedures
5. Extension 2: some OO features

Main issues for code generation

- input : (well-typed) source pgm AST
- output : machine level code

Expected properties for the output:

- compliance with the target machine
  instruction set, architecture, memory access, OS, . . .
- correctness of the generated code
  semantically equivalent to the source pgm
- optimality w.r.t. non-functional criteria
  execution time, memory size, energy comsumption, . . .

A pragmatic approach

AST

↓

intermediate code generation

↓

Intermediate Representation 1 ⇄ optimization(s)

↓

...

↓

Intermediate Representation n ⇄ optimization(s)

↓

(final) code generation

↓

target machine code

Intermediate Representations

- Abstractions of a real target machine
  - generic code level instruction set
  - simple addressing modes
  - simple memory hierarchy

- Examples
  - a “stack machine”
  - a “register machine”
  - etc.

Rk: other intermediate representations are used in the optimization phases . . .

The "M" Machine

- Machine with (unlimited) registers Ri
  special registers: program counter `PC`, frame pointer `FP`, stack pointer `SP`, register `R0` (contains always 0)
- Instructions, addresses, and integers take 4 bytes in memory
- Address of variable `x` is `E - offx` where:
  - `E` = address of the environment definition of `x`
  - `offx` = offset of `x` within this environment (staticaly computed, stored in the symbol table)
- Addressing modes:
  `Ri`, `val` (immediate), `Ri +/- Rj`, `Ri +/- offset`
- usual arithmetic instructions `OPER`: `ADD`, `SUB`, `AND`, etc.
- usual (conditional) branch instructions `BRANCH`: `BA`, `BEQ`, `BGT`, etc.

Instruction Set

instruction	informal semantics
`OPER Ri, Rj, Rk`	Ri $\leftarrow$ Rj oper Rk
`OPER Ri, Rk, val`	Ri $\leftarrow$ Rj oper val
`CMP Ri, Rj`	Ri-Rj (set cond flags)
`LD Ri, [adr]`	Ri $\leftarrow$ Mem[adr]
`ST Ri, [adr]`	Mem[adr] $\leftarrow$ Ri
`BRANCH label`	if cond then PC $\leftarrow$ `label`
	else PC $\leftarrow$ PC + 4
`CALL label`	branch to the procedure
	labelled with `label`
`RET`	end of procedure

The `while` *language*

p	::=	d ; c
d	::=	var x | d ; d
s	::=	x := a | s ; s | if b then s else s | while b s
a	::=	n | x | a + a | a * a | ...
b	::=	a = a | b and b | not b | ...

Rk: terms are well-typed
$\longrightarrow$ distinction between boolean and arithmetic expr.

Exo: Give the “M Machine” code for the following terms:

1. `y := x+42 * (3+y)`
2. `if (not x=1) then x := x+1`
   `else x := x-1 ; y := x ;`

Functions for Code Generation

`GCStm` : Stm $\rightarrow$ Code*

`GCStm(s)` *computes the code* `C` *corresponding to statement* `s`.

`GCAExp` : Exp $\rightarrow$ Code* $\times$ `Reg`

`GCAExp(e)` *returns a pair (*`C`*,* `i`*) where* `C` *is the code allowing to 1. compute the value of* `e`*, 2. store it in* `Ri`*.*

`GCBExp` : BExp$\times$ $\mathcal{L}$abel$\times$ $\mathcal{L}$abel $\rightarrow$ Code*

`GCBExp(b, ltrue, lfalse)` *produces code* `C` *allowing to compute the value of* `b` *and branch to label* `ltrue` *when this value is "true" and to* `lfalse` *otherwise.*

Auxilliary functions

`AllocRegister`	:	$\rightarrow$ `Reg` allocate a new register `Ri`
`newLabel`	:	$\rightarrow$ `Labels` produce a new `label`
`GetOffset`	:	Var $\rightarrow$ $\mathbb{N}$ returns the offset corresponding to the specified name

$\|$ denotes concatenation for Code sequences.

GCStm

`GCStm(x := e)`	=	`Let`	(C,i)=`GCAExp`(e),
			k=`GetOffset`(x)
		`in`	C$\|$ `ST` Ri, [FP-k]
`GCStm(`c_1 `;` c_2`)`	=	`Let`	C_1 = `GCStm`(c_1),
			C_2 = `GCStm`(c_2)
		`in`	$C_1 \| C_2$

GCStm (2)

```
GCStm(while e c)  =  Let  lb=newLabel(),
                          ltrue=newLabel(),
                          lfalse=newLabel()
                     in   lb:||
                          GCBExp(e,ltrue,lfalse)||
                          ltrue:||
                          GCStm(c)||
                          BA lb||
                          lfalse:
```

GCStm (3)

```
GCStm(if e then c1 else c2)  =  Let  lnext=newLabel(),
                                     ltrue=newLabel(),
                                     lfalse=newLabel()
                                in   GCBExp(e,ltrue,lfalse
                                     ltrue:
                                     GCStm(c1)||
                                     BA lnext ||
                                     lfalse:||
                                     GCStm(c2)||
                                     lnext:
```

GCAexp

GCAExp(x)	=	Let	i=AllocRegister()
			k=GetOffset(x)
		in	((LD Ri,[FP-k]),i)
GCAExp(n)	=	Let	i=AllocRegister()
		in	((ADD Ri,R0,n),i)
GCAExp(e_1 + e_2)	=	Let	(C_1,i_1)=GCAExp(e_1),
			(C_2,i_2)=GCAExp(e_2),
			k=AllocRegister()
		in	(($C_1 \| C_2 \|$ ADD Rk, Ri_1,Ri_2),k)

GCBexp

GCBExp (e_1 = e_2,ltrue,lfalse)	=	Let	(C_1,i_1)=GCAExp(e_1),
			(C_2,i_2)=GCAExp(e_2),
		in	$C_1 \| C_2 \|$
			CMP Ri_1, Ri_2
			BEQ ltrue
			BA lfalse
GCBExp (e_1 et e_2,ltrue,lfalse)	=	Let	l=newLabel()
		in	GCBExp(e_1,l,lfalse)$\|$
			l:$\|$
			GCBExp(e_2,ltrue,lfalse)
GCBExp(NOT e,ltrue,lfalse)	=		GCBExp(e,lfalse,ltrue)

Exercises

- code obtained for
  - `y := x+42 * (3+y)`
  - `if (not x=1) then x := x+1`
    `else x := x-1 ; y := x ;`
- add new statements (e.g, `repeat`)
- add new operators (e.g, `b ? e1 : e2`)

Extension 1: blocks

Blocks

Syntax

$$
\begin{array}{rcl}
S & ::= & \cdots \mid \textbf{begin}\ D_V\ ;\ S\ \textbf{end} \\
D_V & ::= & \textbf{var}\ x \mid D_V\ ;\ D_V
\end{array}
$$

Rk: variables are unitialized and assumed to be of type **Int**

Problems raised for code generation

$\rightarrow$ to preserve scoping rules:

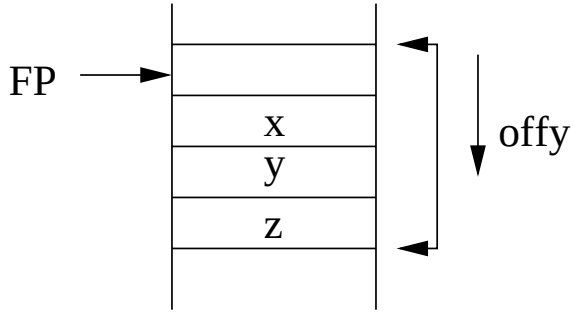
- local variables should be *visible* inside the block
- their *lifetime* should be limited to block execution

Possible locations to store local variables

$\rightarrow$ registers vs memory

Storing local variables in memory - Example 1

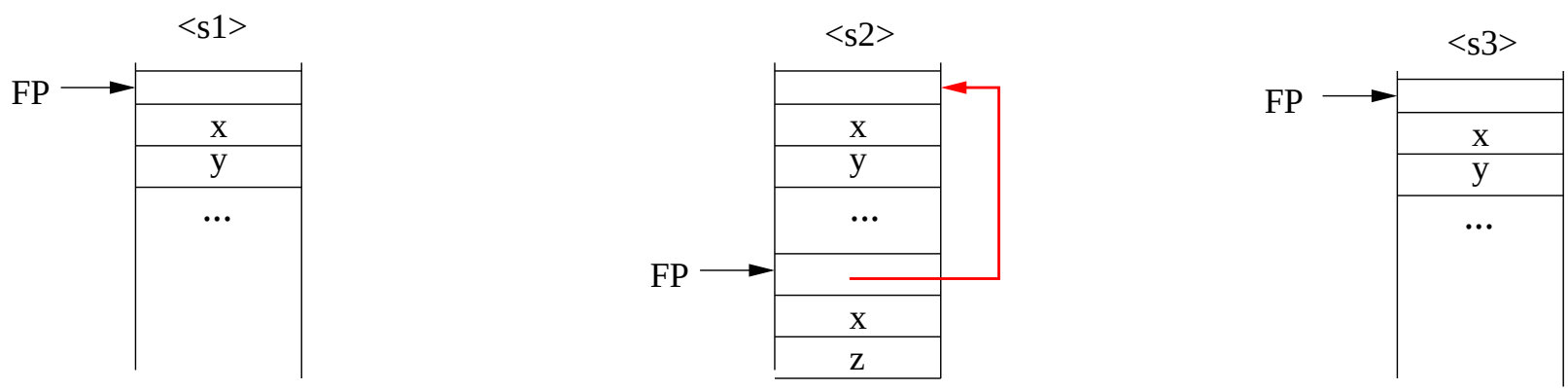
```
begin
   var x ; var y ; var z ;
   ...
end
```



- a *memory environment* is associated to each declaration Dv
- register FP contains the address of the current environment
- (static) offsets are associated to each local variables

Storing local variables in memory - Example 2

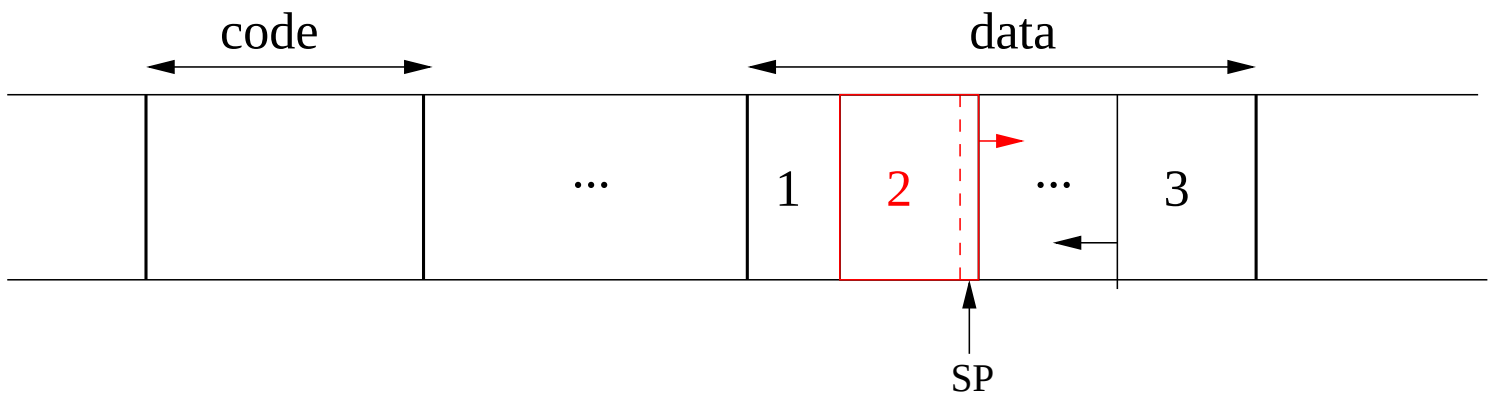
```
begin
   var x ; var y ; <s1>
   begin
      var x ; var z ; <s2>
   end ;
   <s3>
end
```



- entering/leaving a block $\rightarrow$ allocate/de-allocate a mem. env.
- nested block env. have to be linked together: "Ariane link"

$\Rightarrow$ a stack of memory environments . . . ($\sim$ operational semantics)

Structure of the memory



1: global variables

2: execution stack, SP = last occupied address

3: heap (for dynamic allocation)

Code generation for variable declarations

`SizeDecl` : $D_V \rightarrow \mathbb{N}$

`SizeDecl(d)` *computes the size of declarations* `d`

`SizeDecl(var x)`	=	`4` (x of type `Int`)	
`SizeDecl(d`$_1$ `; d`$_2$`)`	=	`Let`	v_1 = `SizeDecl(d`$_1$`)`,
			v_2 = `SizeDecl(d`$_2$`)`
		`in`	$v_1 + v_2$

Code Generation for blocks

```
GCStm(begin d ; s ; end)  =  Let   size =SizeDecl(d),
                                   C=GCStm(s)
                             in    ADD, SP, SP, -4 ||
                                   ST FP, [SP] ||
                                   ADD FP, SP, 0 ||
                                   ADD SP, SP, size ||
                                   C ||
                                   ADD SP, FP, 0 ||
                                   LD FP, [SP] ||
                                   ADD SP, SP, 4 ||
```

With the help of some auxilliary functions . . .

prologue(size)	epilogue	push register(Ri)

ADD SP, SP, -4
ST FP, [SP]
ADD FP, SP, 0
ADD SP, SP, size
	ADD SP, FP, 0
	LD FP, [SP]
	ADD SP, SP, +4
		ADD SP, SP, -4
		ST Ri, [SP]

GCStm(begin d ; s ; end)	=	Let	size =SizeDecl(d),
			C=GCStm(s)
		in	Prologue(size) ‖
			C ‖
			Epilogue

Access to variables from a block ?

```
...
begin
   var ...
   x := ...
end
```

What is the memory address of x ?

- if x is a local variable (w.r.t the current block)
  $\Rightarrow$ adr(x) = FP + GetOffset(x)
- if x is a non local variable
  $\Rightarrow$ it is defined in a “nesting” memory env. E
  $\Rightarrow$ adr(x) = adr(E) + GetOffset(x)
  adr(E) can be accessed through the “Ariane link” . . .

Access to non local *variables*

The number n of indirections to perform on the “Ariane link” depends on the “distance” between:

- the nesting level of the current block : p
- the nesting level of the target environment : r

More precisely:

- $r \leq p$
- $n = p - r$

$\Rightarrow n$ can be staticaly computed . . .

Example

```
begin
   var x ; /* env. E1, nesting level = 1 */
   begin
     var y ; /* env. E2, nesting level = 2 */
     begin
        var z ; /* env. E3, nesting level = 3 */
        x := y + z /* s, nesting level = 3 */
     end
   end
end
```

From statement s:

- no indirection to access to z
- 1 indirection to access to y
- 2 indirections to access to x

Code generation for variable access

1. the nesting level r of each identifier x is computed during type-checking;
2. it is associated to each occurrence of x in the AST
   (via the symbol table)
3. function GCStm keeps track of the current nesting level p
   (incremented/decremented at each block entry/exit)

adr(x) is obtained by executing the following code:

- if $r = p$:

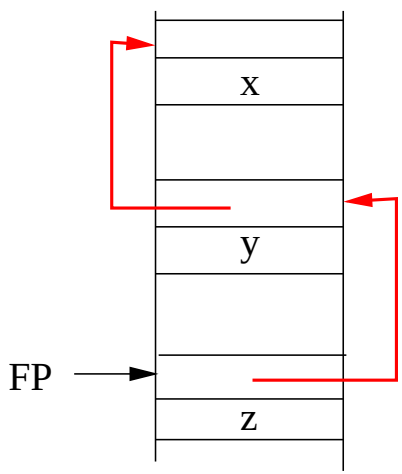
  FP + GetOffset(x)

- if $r < p$:

  `LD Ri, [FP]`
  `LD Ri, [Ri]}` $(p - r - 1)$ times
  `Ri` + GetOffset(x)

Example (ctn'd)

```
begin
   var x ; /* env. E1, nesting level = 1 */
   begin
     var y ; /* env. E2, nesting level = 2 */
     begin
        var z ; /* env. E3, nesting level = 3 */
        x := y + z /* s, nesting level = 3 */
     end
   end
end
```




```
LD  R1, [FP]        ! R1 = adr(E2)
LD R2,  [R1 + offy]     ! R2 = y
LD R3,  [FP + offz]     ! R3 = z
ADD R4, R2, R3      ! R4 = y+z
LD  R5, [FP]
LD  R5, [R5]        ! R5 = adr(E1)
ST R4,  [R5 + offx]     ! x = y + z
```

Code generated for statement s

Extension 2: Procedures

Syntax

Procedure declarations:

$$\begin{array}{rcl} D_P & ::= & \textbf{proc}\ p\ (FP_L)\ \ \textbf{is}\ S\ ;\ D_P \mid \epsilon \\ FP_L & ::= & \textbf{x},\ FP_L \mid \epsilon \end{array}$$

Statements:

$$\begin{array}{rcl} S & ::= & \cdots \mid \textbf{begin}\ D_V\ ; D_P\ ;\ S\ \textbf{end} \mid \textbf{call}\ p(EP_L) \\ EP_L & ::= & AExp,\ EP_L \mid \epsilon \end{array}$$

FP_L: formal parameters list ; EP_L: effective parameters list

Rk: we assume here value-passing of integer parameters . . .

Example

```
var z ;

proc p1 () is
    begin
      proc p2(x, y) is z := x + y ;
      z := 0 ;
      call p2(z+1, 3) ;
    end

proc p3 (x) is
    begin
       var z ;
       call p1() ; z := z+x ;
    end

call p3(42) ;
```

Main issues for code generation

Procedure `P` is calling procedure `Q` . . .

Before the call:

- set up the memory environment of `Q`
- evaluate and “transmit” the effective parameters
- switch to the memory environment of `Q`
- branch to first intruction of `Q`

During the call:

- access to local/non local procedures and variables
- access to parameter values

After the call:

- switch back to the memory environment of `P`
- resume execution to the `P` instruction following the call

Access to non-local variables

```
proc main is
begin              /* definition env. of p */
   var x ;
   proc p() is x:=3 ;
   proc q() is
     begin
         var x ;
         proc r() is call p() ;
         call r() ;
     end ;
   call q() ;
end
```

Static binding $\Rightarrow$ when p is executed:

- acces to the memory env. of `main` =
  definition environment of the callee, static link
- acces to the memory env. of `r`
  memory environment of the caller, dynamic link

Information exchanged between callers *and* callees *?*

- parameter values
- return address
- address of the caller memory environment (dynamic link)
- address of the callee environment definition (static link)

This information should be stored in a memory zone:

- dynamically allocated
  (exact number of procedure calls cannot be foreseen at compile time)
- accessible from both parties
  (those address could be computed by the caller and the callee)

$\Rightarrow$

inside the execution stack, at well defined offsets w.r.t FP

A possible "protocol" between the two parties

Before the call, the caller:

- evaluates the effective parameters
- pushes their values
- pushes the static link of the callee
- pushes the return address, and branch to the callee's 1st instruction

when it begins, the callee:

- pushes FP (dynamic link)
- assigns SP to FP (memory env. address)
- allocates its local variables on the stack

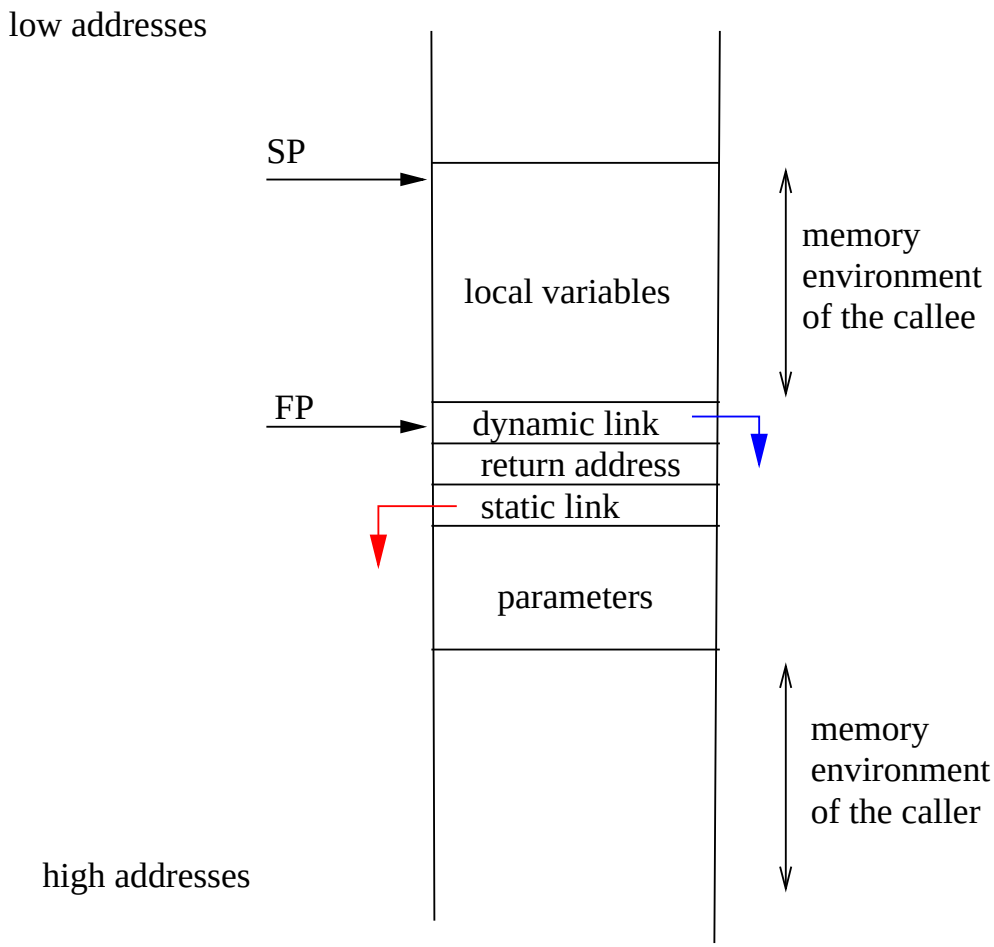
when it ends, the callee:

- de-allocates its local variables
- restores FP to caller's memory env. (dynamic link)
- branch to the return address, and pops it from the stack

After the call, the caller

- de-allocates the static link and parameters

Organization of the execution stack



Addresses, from the callee:

loc. variables: FP+d, d<0

dynamic link: FP

return address: FP+4

static link: FP+8

parameters: FP+d, d>=12

Memory environment of the callee

...	0
Loc. var_n	←SP, FP- 4*n
...	
Loc. var_1	←FP
Dynamic link	←FP
Return address	←FP+4
Static link	←FP+8
Param_n	←FP+12
...	
Param_1	←FP+8+4*n

Code generation for a procedure declaration

`GCProc` : $D_P \rightarrow$ Code*

`GCStm(dp)` *computes the code* `C` *corresponding to procedure declaration* `dp`.

```
GCProc (proc p (FP_L) is s end )  =  Let
                                        C=GCStm(s)
                                     in Prologue(0) ||
                                        C ||
                                        Epilogue
```

```
GCProc (proc p (FP_L) is begin dv ; dp ; s end )  =  Let  size =SizeDecl(dv),
                                                         C=GCStm(s)
                                                     in  Prologue(size) ||
                                                         C ||
                                                         Epilogue
```

Rk: this function is applied to each procedure declaration

Prologue & Epilogue

Prologue (size):

```
push (FP)              ! dynamic link
ADD FP, SP, 0          ! FP := SP
ADD SP, SP, -size       ! loc. variables allocation
```

Epilogue:

```
ADD SP, FP, 0          ! SP := FP, loc. var. de-allocation
LD FP, [SP]            ! restore FP
ADD SP, SP, +4
RET                    ! return to caller
```

RET:

```
LD PC, [SP]  //  ADD SP, SP, +4
```

Code Generation for a procedure call

Four steps:

1. evaluate and push each effective parameter
2. push the static link of the callee
3. push the return address and branch to the callee
4. de-allocate the parameter zone

```
GCStm (call p (ep))   =   Let   (C, size) = GCParam(ep)
                          in
                                C ||
                                Push (StaticLink(p)) ||
                                CALL p ||
                                ADD SP, SP, size+4
```

CALL p:

ADD R1, PC, +4 // Push (R1) // BA p

Parameters evaluation

`GCParam` : $EP_L \rightarrow$ Code* $\times \mathbb{N}$

`GCStm(ep)=(c,n)` *where* `c` *is the code to evaluate and "push" each effective parameter of* `ep` *and* `n` *is the size of pushed data.*

`GCParam` (ε)	=	(ε, 0)
`GCParam(a ; ep)`	=	`Let`
		(Ca, i) = `GCAexp` (a),
		(C, size) = `GCParam` (ep)
		`in`
		(Ca $\parallel$ Push (R_i) $\parallel$ C, 4 + size)

Static link and non local variable access ?

- A global (unique) name is given to each identifier:

```
proc Main is
   proc P1 (...) is
     ...
         proc Pn (...) is
          begin
             var x ...
          end
```

$\rightarrow$ x is named $Main.P_1.\cdots.P_n.x$

- This notation induces a partial order:

$$(Main.P_1 \cdots .P_n \leq Main.P'_1 \cdots .P'_{n'}) \Leftrightarrow (n \leq n' \text{ and } \forall k \leq n. P_k = P'_k)$$

- For an identifier $x = Main.P_1 \cdots .P_n.x$,
  $x^\bullet = Main.P_1 \cdots .P_n$ is the definition environment of x

- For any identifier x (variable or procedure), procedure P can access x iff $x^\bullet \leq P$.

Examples

- A variable x declared in P can be accessed from P since $x^\bullet = P$ (hence $x^\bullet \leq P$).
- If g and x are declared in f, then x can be accessed from g since $x^\bullet = f$ and $f \leq g$.
- If x and f_1 are declared in $Main$, f_2 is declared in f_1, then x can be accessed from f_2 since $x^\bullet = Main$, $f_2 = Main.f_1.f_2$ ($x^\bullet \leq f_2$)
- If p_1 and p_2 are both declared in $Main$, x is declared in p_1, then x cannot be accessed from p_2, since $x^\bullet = Main.p_1$ and $Main.p_1 \not\leq Main.p_2$

Code Generation for accessing (non-) local identifiers

d_x: offset of x (variables or parameters) in its definition environment ($x^\bullet$)
P: current procedure

Condition	x = variable or parameter	x = procedure
$x^\bullet = P$	adr(x) = FP+d$_x$	SL(x) = FP
$x^\bullet < P$ $x = M.P_1 \cdots P_k$ $P = M.P_1 \cdots P_k \cdots P_n$	n-k-1 indirections `LD R,[FP+8]` `LD R,[R+8]`$\} \times (n-k-1)$ adr(x) = R+d$_x$	n-k-1 indirections `LD R,[FP+8]` `LD R,[R+8]`$\} \times (n-k-1)$ SL(x)=R

Back to the 1st example

```
var z ;

proc p1 () is
    begin
      proc p2(x, y) is z := x + y ;
      z := 0 ;
      call p2(z+1, 3) ;
    end

proc p3 (x) is
    begin
       var z ;
       call p1() ; z := z+x ;
    end

call p3(42) ;
```

Exercice:

- give the execution stack when p2 is executed
- give the code for procedures p1 and p2

Exercice

Consider the following extensions

- functions
- other parameter modes (by reference, by result)
- dynamic binding for variables and procedures ?

Procedures used as variables or parameters

```
var z1 ;
var p proc (int) ; /* p is a procedure variable */
proc p1 (x : int) is z1 := x ;
proc p2 (q : proc (int)) is call q(2) ;

proc q1 is
     begin
       var z1 ;
       proc q2 (y int) is z1 := x ;
       p := q2 ;
       call p ;
     end

p := p1 ;
call p ;
call p2 (p1) ;
```

Q: what code to produce for `p := ...` ? for `call p2(p1)` ? for `call p` ?

Information associated to a procedure at code level

```
p := q2
...
call p
```

To translate a procedure call, we need:

- the address of its 1st instruction
- the address of its environment definition

$\Rightarrow$ Variable p should store both information
$\Rightarrow$ At code level, a procedure type is a pair
(address of code, address of memory environment)

Exercice: code produced for the previous example ?