Programming Language Semantics and Compiler Design

(Sémantique des Langages de Programmation et Compilation)

Generation of Assembly Code

Frédéric Lang & Laurent Mounier
firstname.lastname@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, Inria,
Laboratoire d'Informatique de Grenoble & Verimag

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM^{2}AG
www.univ-grenoble-alpes.fr — im2ag.univ-grenoble-alpes.fr

Outline - Generation of Assembly Code

Outline - Generation of Assembly Code

Introduction

Machine "M"

Code Generation for Language **While**

Code Generation for Language **Block**

Code Generation for Language **Proc**

Main issues for code generation

- input: (well-typed) source pgm AST (or intermediate code)
- output: machine level code (assembly, relocatable, or absolute code)

Expected properties for the output

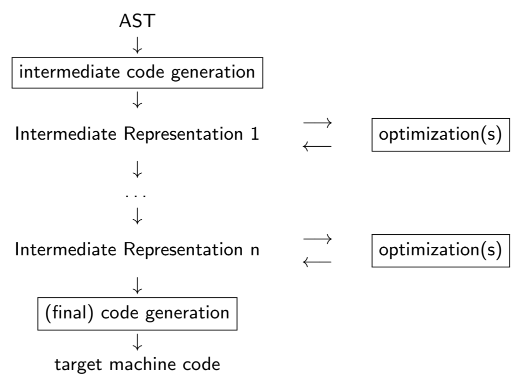
- compliance with the target machine
  instruction set, architecture, memory access, OS, ...
- correctness of the generated code
  semantically equivalent to the source pgm
- optimality w.r.t. non-functional criteria
  execution time, memory size, energy comsumption, ...
- and security w.r.t. external (cyber-)attacks
  hardened code, no information leakage, checks for vulnerability detection, ...

Main issues for code generation (ctd)

Tasks of the Code Generator

- **Instruction selection**: choosing appropriate target-machine instructions to implement the (IR) statements.
  Complexity depends on:
  - how abstract is the IR,
  - "expressiveness of instruction set" (e.g., support of some types),
  - expected quality of the output code according to some criteria (speed and size).
- **Registers allocation and assignment**: deciding what variables to keep in which registers at every location (when the target machine uses registers).
- **Instruction ordering**: deciding the scheduling order for the execution of instructions.
  - It affects the efficiency of the code and the required registers.
  - It is generally not possible to obtain an optimal (NP-complete) ⇒ heuristics

A pragmatic approach



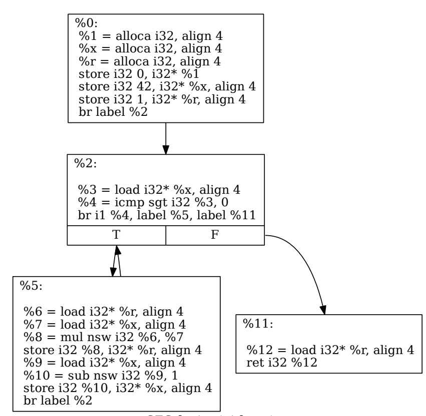
Intermediate Representations

- 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.

Remark Other intermediate representations are used in the optimization phases. □

Example 1: LLVM IR (register machine)

```
int main() {
int x, r;
x=42 ; r=1 ;
while (x>0) {
   r = r*x;
   x = x-1;
} ;
return r ;
}
```



CFG for 'main' function

Example 2: Java ByteCode (stack machine)

```
public static int main() {
int x, r;
x=42 ; r=1 ;
while (x>0) {
   r = r*x;
   x = x-1;
} ;
return r ;
}
```

```
public static int main(java.lang.String[]);
    Code:
       0: bipush        42
       2: istore_1
       3: iconst_1
       4: istore_2
       5: iload_1
       6: ifle          20
       9: iload_2
      10: iload_1
      11: imul
      12: istore_2
      13: iload_1
      14: iconst_1
      15: isub
      16: istore_1
      17: goto          5
      20: iload_2
      21: ireturn
```


Outline - Generation of Assembly Code

Machine "M"

Machine with Registers

- Unlimited register set, $\{R_0, R_1, R_2, \dots\}$.
- Special registers:
  - program counter `PC`,
  - stack pointer `SP`,
  - frame pointer `FP`,
  - register `R0` (contains always 0).

  (the exact purpose of these registers will become clear later)

Instructions, addresses, and integers take 4 bytes in memory.

Addressing

- Address of variable `x` is `E - offx` where:
  - `E` = address of the environment where `x` is defined
  - `offx` = offset of `x` within this environment (staticaly computed, stored in the symbol table)
- Addressing modes:
  `Ri`, `val` (immediate), `Ri +/- Rj`, `Ri +/- offset`

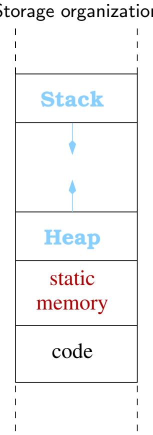
Instruction Set

- Usual arithmetic instructions `OPER`: `ADD`, `SUB`, `AND`, etc.
- Usual (conditional) branch instructions `BRANCH`: `BA`, `BEQ` (=), `BGT` (>), BLT (<), `BGE` (≥), `BLE` (≤), `BNE` (≠).
- Usual calling instructions to labels or register: `CALL`.

instruction	informal semantics
`OPER Ri, Rj, Rk`	Ri ← Rj oper Rk
`OPER Ri, Rj, val`	Ri ← Rj oper val
`CMP Ri, Rj`	Ri - Rj (set cond flags)
`LD Ri, [adr]`	Ri ← Mem[adr]
`ST Ri, [adr]`	Mem[adr] ← Ri
`BRANCH label`	if cond then PC ← `label` else PC ← PC + 4

Run-Time Environment

Storage organization



- Static data:
  - computed at compile-time
  - allocated at load-time (once for all)
  - ex: global variables, constant strings, etc.
- Dynamic data:
  - in the stack
    - allocated with procedure activation
    - life-span: procedure execution
    - ex: local data of a proc. (parameters and local vars)
  - in the heap:
    - managed using `malloc` and `free`
    - life-span: from alloc to free (possibly after pgm termination)
    - ex: dynamic arrays

Outline - Generation of Assembly Code

Language **While**

Reminder

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

Remark Terms are well-typed.
$\rightarrow$ distinction between boolean and arithmetic expr. □

Language **While**

Informal code generation

Informal code generation

Give the "Machine M" code for the following statements:

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

Functions for code generation

Notation

- Code*: instruction sequences for machine "M"
- ||: concatenation operator for code and sequences of code

GCStm : Stm → Code*
GCStm(s) computes the code C corresponding to statement s.

GCAExp : Exp → Code* × Reg
GCAExp(e) returns a pair (C, i) where C is the code allowing to

1. computes the value of e,
2. stores it in Ri.

GCBExp : BExp× $\mathcal{L}$abel× $\mathcal{L}$abel → Code*
GCBExp(b, ltrue, lfalse) produces the code C that computes the value of b and branches to label ltrue when this value is "true" and to lfalse otherwise.

Auxiliary functions

AllocRegister	:	→ Reg allocates a new register Ri
newLabel	:	→ Labels produces a new label
GetOffset	:	Var → $\mathbb{Z}$ returns the offset corresponding to the specified name which depends on the position at which the variable is declared (shall be defined precisely for blocks and procedures)

Function GCStm

Assignments, sequential and iterative compositions

GCStm (x := e)	=	Let in	(C,i) = GCAExp(e), k=GetOffset(x) C || ST Ri, [FP + k]
GCStm (s_1 ; s_2)	=	Let in	C_1 = GCStm(s_1), C_2 = GCStm(s_2) C_1 || C_2
GCStm (while e do s od)	=	Let in	lb=newLabel(), ltrue=newLabel(), lfalse=newLabel() lb:|| GCBExp(e,ltrue,lfalse)|| ltrue:|| GCStm(s)|| BA lb|| lfalse:

Function GCStm (ctd)

Conditional statement

GCStm (if e then s_1 else s_2)	=	Let in	lnext=newLabel(), ltrue=newLabel(), lfalse=newLabel() GCBExp(e,ltrue,lfalse)|| ltrue: GCStm(s_1)|| BA lnext || lfalse:|| GCStm(s_2)|| lnext:

Function GCAexp

Arithmetic expressions

GCAExp(x)	=	Let	i=AllocRegister()
			k=GetOffset(x)
		in	((LD Ri, [FP + k]),i)
GCAExp(n)	=	Let	i=AllocRegister()
		in	((ADD R_i, R_0, 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)

Function GCBexp

Boolean expressions

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 and 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)

Similar principle for $e_1 > e_2$, $e_1 \geq e_2$, etc.

Exercises/Examples

Code generation

Assume that the offsets of variables `x`, `y`, and `z` are -4, -8, and -12, respectively.
Give the "Machine M" code for the following statements:

1. if $x > 0$ then $z := x$ else $z := y$ fi
2. `x := 10;` while $x > 10$ do $x := x - 1$ od

Adding new statements to **While**

Extend the code generation function

- to consider statements of the form repeat S until b,
- to consider Boolean expressions of the form `b1 xor b2`,
- to consider arithmetical expressions of the form `b ? e1 : e2`.

Outline - Generation of Assembly Code

Introduction

Machine "M"

Code Generation for Language **While**

Code Generation for Language **Block**

Code Generation for Language **Proc**

Blocks

Syntax

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

Remark Variables are not initialized and assumed to be of type **Int**. □

Problems raised for code generation

→ to preserve scoping rules:

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

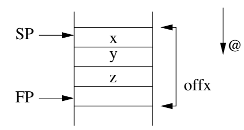
Possible locations to store local variables

→ registers vs memory

Storing local variables in memory - Example 1

Access to local variables within a block

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



- A memory environment is associated to each declaration in D_V.
- Register `FP` contains the address of the current environment.
- (Static) offsets are associated to each local variables.

Definition 1 (Offset of a local variable)

The offset of a local variable is $-4 \times i$, where i is the position of the variable in the sequence of local declarations.

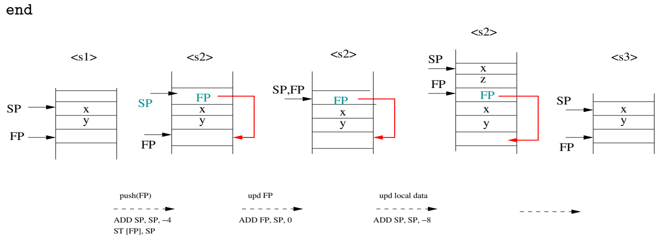
Example 1 (Offset of a local variable)

For `var x ; var y ; var z ;`: `GetOffset(z)` = -4, `GetOffset(y)` = -8, `GetOffset(x)` = -12.

Storing local variables in memory - Example 2

Access to local variables in case of nested blocks

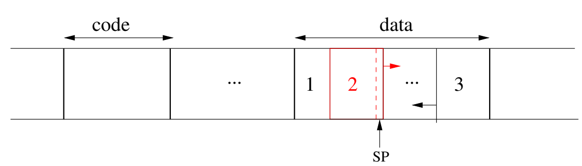
```
begin
  var x ; var y ; <s1>
  begin
    var x ; var z ; <s2>
  end ;
  <s3>
end
```



- entering/leaving a block → allocate/de-allocate a mem. env.
- nested block env. have to be linked together: "Ariadne link"

⇒ a stack of memory environments . . . (∼ 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 auxiliary 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)
  ⇒ adr(x) = FP + GetOffset(x)
- if x is a non local variable
  ⇒ it is defined in a "nesting" memory env. E
  ⇒ adr(x) = adr(E) + GetOffset(x)
  adr(E) can be accessed through the "Ariadne link" . . .

Access to non-local variables

The number n of indirections to perform on the "Ariadne 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 statically computed...

Remark The number of indirections can be statically computed because the programming language has a semantics with static bindings. □

Access to non-local variables: example

Example 2 (Access to non-local variables and number of indirections)

```
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 z
- 1 indirection to access y
- 2 indirections to access 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 $r = p$:

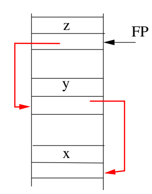
  FP + GetOffset(x)

- 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

An alternative for block variables

For code generation, all local variables of nested blocks can be flattened, i.e., shifted up to the outermost block environment

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

$\Rightarrow$ all variable adresses are relative to env. `E1`:

- no need to push/pop `FP` at each block entry/exit (no prologue/epilogue)
- no need to generate code for computing variable adresses at runtime

Demo: code generation for nested blocks

Outline - Generation of Assembly Code

Outline - Generation of Assembly Code

Syntax of Language **Proc**

Reminder

Procedure declarations:

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

Statements:

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

FP_L: list of formal parameters; EP_L: list of effective parameters

- only one single block, no nested procedures ($\equiv$ C language)
- We assume (first) value-passing for integer parameters.

Example of program in **Proc**

```
begin
var z ;  // global variable

proc p1 (x, y) is
      var t ;  // local variable
      z := 42 ;
      t := x + y + z ;
    end

proc p2 (x) is
       var z ; // local variable (hides the global one)
       call p1(x, 12) ; z := z+x ;
    end

call p2(42) ;  // main block body
end
```


Outline - Generation of Assembly Code

Main issues for code generation with procedures

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 instruction of `Q`

During the call:
- access to local/global variables
- access to parameter values

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

Outline - Generation of Assembly Code

Information exchanged between callers and callees?

- parameter values
- return address
- address of the caller memory environment (dynamic 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 addresses should be computable by the caller and the callee)

⇒
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 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

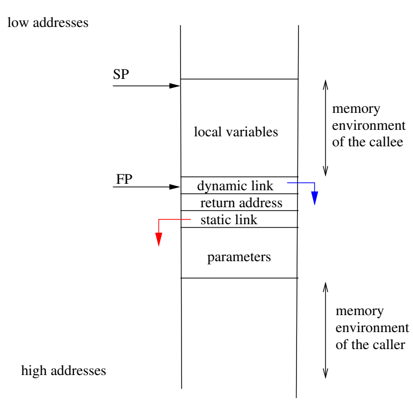
When it ends, the callee:

- 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 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-4
Dynamic link	←FP
Return address	←FP+4
Param$_n$	←FP+8
...	
Param$_1$	←FP+8+4*n

Definition 2 (Offset of a variable or a parameter)

- For local variable $\texttt{var}_i$, as before, $\texttt{GetOffset}(\texttt{var}_i)$ is $-4 \times i$.
- For parameter $\texttt{param}_i$, $\texttt{GetOffset}(\texttt{param}_i)$ is $8 + 4 \times (n - i)$.

Instructions CALL and RET

- Assigning a code location (i.e., a label) to a register.
- Usual calling instructions to labels or register: `CALL`.
- Procedure return.

instruction	informal semantics
`SET R label`	R ← @(label)
`CALL label`	branch to the procedure labelled with `label` `PUSH(PC)` ∥ PC← `label`
`CALL R`	branch to the address contained in register `R` `PUSH(PC)` ∥ PC← `R`
`RET`	end of procedure: `PC` ← `MEM[SP]` // `SP` ← `SP + 4`

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
                                     p = newlabel(),
                                     C=GCStm(s)
                                  in p: Prologue(0) ||
                                     C ||
                                     Epilogue
```

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

Remark GCProc is applied to each procedure declaration. □

Code generation for a procedure declaration (ctd)

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      ! erase previous backup of FP
RET                 ! return to caller
```

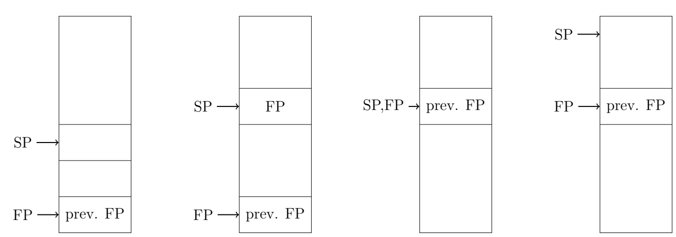
RET:

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


Illustration of the prologue

Prologue (size):

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



Outline - Generation of Assembly Code

Code generation for a procedure call

Four steps:

1. evaluate and push each effective parameter (**with passing-by-value mode** for now)
2. push the return address and branch to the callee
3. de-allocate the parameter zone

GCStm (**call** p (ep))	=	Let	(C, size) = GCParam(ep)
		in	
			C ∥
			CALL p ∥
			ADD SP, SP, size

CALL p:

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

Code generation for the evaluation of parameters

GCParam : $EP_L \to \mathrm{Code}^* \times \mathbb{N}$

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 ∥ Push (R_i) ∥ C, 4 + size)

Access to local/global variables?

```
begin
var z ;  // global variable

proc p (x, y) is
      var t ;  // local variable
      z := t+42 ;
end

z := 8 ;
call p(z, 12)
end
```

local variables

static offset w.r.t. the frame pointer FP

@t = FP-4 (within p environment)

global variables

- stored in a memory zone those address @glob is fixed at load time
- static offset w.r.t @glob

@z = @glob - 4 (within the global variable memory area)

Outline - Generation of Assembly Code

Introducing parameters

Let `f` be a function which effective parameter is at address `FP-8`.

Passing by **value**	Passing by **address**

`LD R2, [FP-8]`
`push(R2)`
`CALL f`
`ADD SP, SP, +4`
	`ADD R2, FP, -8`
	`push(R2)`
	`CALL f`
	`ADD SP, SP, +4`

Passing by **result**	Passing by **value-result**

`ADD SP, SP, -4`
`CALL f`
`LD R2, [SP]`
`ST R2, [FP-8]`
`ADD SP, SP, +4`
	`LD R2, [FP-8]`
	`push(R2)`
	`CALL f`
	`LD R2, [SP]`
	`ST R2, [FP-8]`
	`ADD SP, SP, +4`

In practice: ABI (Application Binary Interface)

↪ to "standardize" how processor resources should be used
⇒ required to ensure compatibilities at binary level

- sizes, layouts, and alignments of basic data types
- **calling conventions**
  argument & return value passing, saved registers, etc.
- system calls to the operating system
- the binary format of object files, program libraries, etc.

	Cleans Stack	Arguments	Arg Ordering
cdecl	Caller	On the Stack	Right-to-left
fastcall	Callee	ECX,EDX, then stack	Left-to-Right
stdcall	Callee	On the Stack	Left-to-Right
VC++ thiscall	Callee	EDX (this), then stack	Right-to-left
GCC thiscall	Caller	On the Stack (this pointer first)	Right-to-left

Figure: some calling conventions

Outline - Generation of Assembly Code

Example

```
begin
var z ;

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

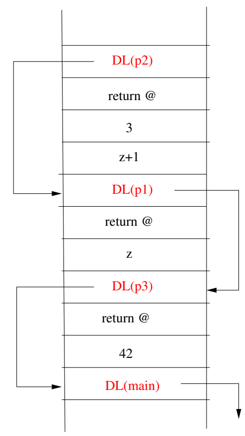
proc p2(x, y) is z := x + y end ;

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

call p3(42)
end
```

- Give the execution stack when p2 is executed.
- Give the code for the block body and for procedures p1, p2 and p3.

Execution stack during the execution of p2



Global variable z is located on a dedicated memory area of base address `@glob`

The block body

Global variable z is at address `@glob -4`

```
var z ;
...
call p3(42) ;
```

Assembly code for the block body:

```
prologue(0)  ! no local variables
! preparing the call p3(42)
! pushing 42
ADD R1, R0, #42
push (R1)
CALL p3
! clean the stack
ADD SP, SP, +4
epilogue()
```


Code for procedure p1

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

Assembly code:

```
prologue(0)  ! no local variables
! z := 0
ST R0, [@glob-4] ! z:=0
! call p2(z+1,3)
!push z+1
LD R1, [@glob_4] ! R1:=z
ADD R2, R2, 1
push(R2) ! 1st param
! push 3
ADD R1, R0, 3
push(R1) ! 2nd param
CALL p2
ADD SP, SP, 8 ! clean the stack
epilogue()
```

First example

Code for procedure p2

```
proc p2(x, y) is z := x + y ;
```

Assembly code:

```
prologue(0)  ! no local variables
LD R1, [FP+12] ! R1:=x
LD R2, [FP+8]  ! R2:=y
ADD R1, R1, R2 ! R1:=x+y
!assignment
ST R1, [@glob-4] ! z:=R1
epilogue()
```


First example

Code for procedure p3

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

Assembly code:

```
prologue(4) ! one local variable (z)
CALL p1
ADD SP, SP, +4 ! clean the stack
!z+x
LD R1, [FP-4]  ! R1:=z
LD R2, [FP+8]  ! R2:=x
ADD R2, R1, R2 ! R2:=z+x
ST R2, [FP-4]  ! z:=R2
epilogue()
```


Exercises

Consider the following extensions to language **Proc**:

- recursive procedures
- functions
- other parameter modes (by reference, by result)
- procedures as variables and parameters
- nested procedures?

↪ Have a look to code generated by real compilers (https://godbolt.org/)

Summary - Generation of Assembly Code

(Machine) Code generation from **While**, **Block**, and **Proc**

- Expected properties of the generated code: compliance, correctness, optimality.
- Machine M (with registers) and its instruction set.
- Formal code-generation functions.
- Nested blocks and (non nested) procedures.
- A well-defined protocol between the caller and the callee.