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 consumption, . . .
**A pragmatic approach**

```
AST ↓
intermediate code generation ↓
Intermediate Representation 1 ↓
. . . ↓
Intermediate Representation n ↓
(final) code generation ↓
target machine code
```

- Optimization(s)

Yassine Lakhnech, Sémantique Start C3 C4 – p.5/50
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Informal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPER Ri, Rj, Rk</td>
<td>$R_i \leftarrow R_j \text{ oper } R_k$</td>
</tr>
<tr>
<td>OPER Ri, Rk, val</td>
<td>$R_i \leftarrow R_j \text{ oper } \text{val}$</td>
</tr>
<tr>
<td>CMP Ri, Rj</td>
<td>$R_i - R_j \text{ (set cond flags)}$</td>
</tr>
<tr>
<td>LD Ri, [adr]</td>
<td>$R_i \leftarrow \text{Mem}[adr]$</td>
</tr>
<tr>
<td>ST Ri, [adr]</td>
<td>$\text{Mem}[adr] \leftarrow R_i$</td>
</tr>
<tr>
<td>BRANCH label</td>
<td>if cond then PC $\leftarrow$ label</td>
</tr>
<tr>
<td></td>
<td>else PC $\leftarrow$ PC + 4</td>
</tr>
<tr>
<td>CALL label</td>
<td>branch to the procedure labelled with label</td>
</tr>
<tr>
<td>RET</td>
<td>end of procedure</td>
</tr>
</tbody>
</table>
**The while language**

\[ p ::= d ; c \]
\[ d ::= \text{var } x \mid d ; d \]
\[ s ::= x := a \mid s ; s \mid \text{if } b \text{ then } s \text{ else } s \mid \text{while } b \; s \]
\[ a ::= n \mid x \mid a + a \mid a * a \mid \ldots \]
\[ b ::= a = a \mid b \text{ and } b \mid \text{not } b \mid \ldots \]

**Rk:** terms are well-typed
→ distinction between boolean and arithmetic expr.

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

1. \[ y := x + 42 \ast (3 + y) \]
2. \[ \text{if (not } x = 1) \text{ then } x := x + 1 \]
   [else \( x := x - 1 ; y := x \)]
Functions for Code Generation

\[ GCStm : \text{Stm} \rightarrow \text{Code}^* \]

\[ GCStm(s) \text{ computes the code } C \text{ corresponding to statement } s. \]

\[ GCAExp : \text{Exp} \rightarrow \text{Code}^* \times \text{Reg} \]

\[ GCAExp(e) \text{ returns a pair } (C, i) \text{ where } C \text{ is the code allowing to 1. compute the value of } e, 2. \text{ store it in } R_i. \]

\[ GCBExp : \text{BExp} \times \text{Label} \times \text{Label} \rightarrow \text{Code}^* \]

\[ GCBExp(b, ltrue, lfalse) \text{ produces code } C \text{ allowing to compute the value of } b \text{ and branch to label } ltrue \text{ when this value is “true” and to } lfalse \text{ otherwise.} \]
Auxilliary functions

AllocRegister : → Reg
allocate a new register $R_i$

newLabel : → Labels
produce a new label

GetOffset : Var → $\mathbb{N}$
returns the offset corresponding to the specified name

$\| \$ denotes concatenation for Code sequences.
<table>
<thead>
<tr>
<th>$\text{GCStm}\ (x := e)$</th>
<th>$\text{Let } (C,i) = \text{GCAExp}(e), \ k = \text{GetOffset}(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{in } C</td>
</tr>
<tr>
<td>$\text{GCStm}\ (c_1 ; c_2)$</td>
<td>$\text{Let } C_1 = \text{GCStm}(c_1), \ C_2 = \text{GCStm}(c_2)$</td>
</tr>
<tr>
<td></td>
<td>$\text{in } C_1</td>
</tr>
</tbody>
</table>
\textbf{GCStm (2)}

\begin{verbatim}
GCStm(while e c) = Let lb=newLabel(),
ltrue=newLabel(),
lfalse=newLabel()
in lb:\[GCBExp(e,ltrue,lfalse)\]
ltrue:\[GCStm(c)\]
BA lb\]\[lfalse:
\end{verbatim}
GCStm (3)

\[
\text{GCStm}(\text{if } e \text{ then } c_1 \text{ else } c_2) = \text{Let}\begin{array}{c}
\text{nnext=}
\text{newLabel}()
, \\
\text{ltrue=}
\text{newLabel}()
, \\
\text{lfalse=}
\text{newLabel}()
\end{array}
\text{in}\begin{array}{c}
\text{GCBExp}(e,ltrue,lfalse)\parallel \\
ltrue:\begin{array}{c}
\text{GCStm}(c_1)\parallel \\
\text{BA~nnext}~\parallel \\
lfalse:\begin{array}{c}
\text{GCStm}(c_2)\parallel \\
nnext:
\end{array}
\end{array}
\end{array}
\]
<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
</table>
| $GCA_{Exp}(x)$ | Let $i = \text{AllocRegister}()$
| | $k = \text{GetOffset}(x)$
| | in $((\text{LD } R_i, [FP-k]), i)$ |
| $GCA_{Exp}(n)$ | Let $i = \text{AllocRegister}()$
| | in $((\text{ADD } R_i, R_0, n), i)$ |
| $GCA_{Exp}(e_1 + e_2)$ | Let $(C_1, i_1) = GCA_{Exp}(e_1)$,
| | $(C_2, i_2) = GCA_{Exp}(e_2)$,
| | $k = \text{AllocRegister}()$
| | in $((C_1 \parallel C_2 \parallel \text{ADD } R_k, R_{i_1}, R_{i_2}), k)$ |
GCBexp

\[
\begin{align*}
\text{GCBExp}(e_1 = e_2, ltrue, lfalse) &= \text{Let} \quad (C_1, i_1) = \text{GCAExp}(e_1), \\
&\quad (C_2, i_2) = \text{GCAExp}(e_2), \\
&\quad \text{in} \quad C_1 \parallel C_2 \parallel \\
&\quad \text{CMP } Ri_1, Ri_2 \\
&\quad \text{BEQ } ltrue \\
&\quad \text{BA } lfalse \\
\text{GCBExp}(e_1 \text{ et } e_2, ltrue, lfalse) &= \text{Let} \quad l = \text{newLabel}() \\
&\quad \text{in} \quad \text{GCBExp}(e_1, l, lfalse) \parallel \\
&\quad \quad l: \parallel \\
&\quad \quad \text{GCBExp}(e_2, ltrue, lfalse) \\
\text{GCBExp}(\text{NOT } e, ltrue, lfalse) &= \text{GCBExp}(e, lfalse, ltrue)
\end{align*}
\]
Exercises

• code obtained for
  • \( y := x + 42 \times (3 + y) \)
  • if (not \( x = 1 \)) then \( x := x + 1 \)
    else \( x := x - 1 \); \( y := x \);

• add new statements (e.g, \texttt{repeat})
• add new operators (e.g, \texttt{b ? e1 : e2})
Extension 1: blocks
Blocks

Syntax

\[ S ::= \cdots | \text{begin } D_V ; S \text{ end} \]
\[ D_V ::= \text{var } x | D_V ; D_V \]

Rk: variables are uninitialized and assumed to be of type \text{Int}

Problems raised for code generation
→ to preserve scoping rules:
• local variables should be \textit{visible} inside the block
• their \textit{lifetime} should be limited to block execution

Possible locations to store local variables
→ registers vs \textit{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
end
```

- entering/leaving a block → allocate/de-allocate a mem. env.
- nested block env. have to be linked together: “Ariane 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**

\[ \text{SizeDecl: } D_V \rightarrow \mathbb{N} \]

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

<table>
<thead>
<tr>
<th>SizeDecl(var x)</th>
<th>=</th>
<th>4 (x of type Int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SizeDecl(d₁ ; d₂)</td>
<td>=</td>
<td>Let v₁ = SizeDecl(d₁), v₂ = SizeDecl(d₂) in v₁ + v₂</td>
</tr>
</tbody>
</table>
Code Generation for blocks

\[
\text{GCStm}(\text{begin } d \ ; \ s \ ; \ \text{end}) = \text{Let } \text{size} = \text{SizeDecl}(d), \\
C = \text{GCStm}(s) \\
in \text{ADD, SP, SP, -4 } || \\
\text{ST FP, [SP] } || \\
\text{ADD FP, SP, 0 } || \\
\text{ADD SP, SP, size } || \\
C || \\
\text{ADD SP, FP, 0 } || \\
\text{LD FP, [SP] } || \\
\text{ADD SP, SP, 4 } ||
\]
With the help of some auxiliary functions ...
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 “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 statically 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)

$\text{adr}(x)$ is obtained by executing the following code:

- **if** $r = p$:
  
  $\text{FP} + \text{GetOffset}(x)$

- **if** $r < p$:
  
  $\text{LD Ri, [FP]}
  
  \text{LD Ri, [Ri]}$ \text{ (}$p - r - 1$ \text{) times}
  
  $\text{Ri} + \text{GetOffset}(x)$
**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

Code generated for statement $s$

```
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]ST R4, [R5 + offx]     ! x = y + z
```
Extension 2: Procedures
Syntax

Procedure declarations:

\[
D_P ::= \text{proc } p \left(FP_L\right) \text{ is } S \; D_P \; \epsilon
\]
\[
FP_L ::= \text{x, } FP_L \; \epsilon
\]

Statements:

\[
S ::= \cdots \mid \text{begin } D_V \; D_P \; S \text{ end} \mid \text{call } p(EP_L)
\]
\[
EP_L ::= \text{AExp, } EP_L \; \epsilon
\]

\(FP_L\): 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 instruction 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 ⇒ when p is executed:

• access to the memory env. of main =
  definition environment of the callee, static link

• access 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)

⇒

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

Addresses, from the caller:
- memory environment of the caller

Variables and links in the stack:
- Parameters: FP+d, d>=12
- Local variables: FP+d, d<0
- Return address: FP+4
- Dynamic link: FP
- Static link: FP+8
- Stack pointer (SP)
- Frame pointer (FP)
Memory environment of the callee

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc. var(n)</td>
<td>← SP, FP - 4(n)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Loc. var(1)</td>
<td>← FP</td>
</tr>
<tr>
<td>Dynamic link</td>
<td>← FP</td>
</tr>
<tr>
<td>Return address</td>
<td>← FP+4</td>
</tr>
<tr>
<td>Static link</td>
<td>← FP+8</td>
</tr>
<tr>
<td>Param(n)</td>
<td>← FP+12</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Param(1)</td>
<td>← FP+8+4(n)</td>
</tr>
</tbody>
</table>
Code generation for a procedure declaration

\[ \text{GCProc} : D_P \rightarrow \text{Code}^* \]

\( \text{GCStm}(dp) \) computes the code  \( C \) corresponding to procedure declaration  \( dp \).

\[
\begin{align*}
\text{GCProc} \left( \text{proc } p \ (F P_L) \ \text{is } s \ \text{end} \right) &= \text{Let} \\
C &= \text{GCStm}(s) \\
\quad \text{in} \\
\quad \text{Prologue}(0) \ || \\
\quad C \ || \\
\quad \text{Epilogue}
\end{align*}
\]

\[
\begin{align*}
\text{GCProc} \left( \text{proc } p \ (F P_L) \ \text{is begin } dv \ ; \ dp \ ; \ s \ \text{end} \right) &= \text{Let} \\
size &= \text{SizeDecl}(dv), \\
C &= \text{GCStm}(s) \\
\quad \text{in} \\
\quad \text{Prologue}(size) \ || \\
\quad C \ || \\
\quad \text{Epilogue}
\end{align*}
\]

\textbf{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

\[
\begin{align*}
\text{GCStm} \ (\text{call p (ep)}) &= \text{Let} \quad (C, \text{size}) = \text{GCParm}(ep) \ \\
&\quad \text{in} \ \\
&\quad C \ || \ \\
&\quad \text{Push (StaticLink(p)) ||} \ \\
&\quad \text{CALL p ||} \ \\
&\quad \text{ADD SP, SP, size+4}
\end{align*}
\]

\[
\text{CALL p:}
\]

\[
\begin{align*}
\text{ADD R1, PC, +4} \ // \text{Push (R1)} \ // \text{BA p}
\end{align*}
\]
**Parameters evaluation**

\[ \text{GCP} \text{Param} : EP_L \rightarrow \text{Code}^* \times \mathbb{N} \]

\[ \text{GCStm(ep)} = (c, n) \quad \text{where } c \quad \text{is the code to evaluate and "push" each effective parameter of ep and } n \quad \text{is the size of pushed data.} \]

| \text{GCP} \text{Param}(\varepsilon) | = | (\varepsilon, 0) |
| --- | --- |
| \text{GCP} \text{Param}(a ; ep) | = | \text{Let} \ (Ca, i) = \text{GCA} \text{exp}(a), \ (C, \text{size}) = \text{GCP} \text{Param}(ep) \ |
|  | in \ (Ca \parallel \text{Push}(R_i) \parallel C, 4 + \text{size}) |
**Static link and non local variable access ?**

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

  proc Main is
  proc P1 (...)
  ...
  proc Pn (...)
  begin
  var x ...
  end
  \( \rightarrow x \) is named \( Main.P_1 \cdots .P_n.x \)

- This notation induces a partial order:

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

- 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 $x^\star = P$ (hence $x^\star \leq P$).

- If $g$ and $x$ are declared in $f$, then $x$ can be accessed from $g$ since $x^\star = f$ 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 $x^\star = Main$, $f_2 = Main.f_1.f_2$ ($x^\star \leq f_2$)

- 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 $x^\star = Main.p_1$ 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>( x = \text{variable or parameter} )</th>
<th>( x = \text{procedure} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^\bullet = P )</td>
<td>( \text{adr}(x) = \text{FP} + d_x )</td>
<td>( \text{SL}(x) = \text{FP} )</td>
</tr>
</tbody>
</table>
| \( x^\bullet < P \)  
\( x = M.P_1 \cdots P_k \)  
\( P = M.P_1 \cdots P_k \cdots P_n \) | \( \text{n-k-1 indirections} \)  
\( \text{LD } R, [FP+8] \)  
\( \text{LD } R, [R+8] \times (n - k - 1) \)  
\( \text{adr}(x) = R + d_x \) | \( \text{n-k-1 indirections} \)  
\( \text{LD } R, [FP+8] \)  
\( \text{LD } R, [R+8] \times (n - k - 1) \)  
\( \text{SL}(x) = R \) |
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

\[ p := q2 \]
\[ \ldots \]
\[ \text{call } p \]

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?