

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 IM²AG
www.univ-grenoble-alpes.fr → im2ag.univ-grenoble-alpes.fr

Outline - Generation of Assembly Code

Introduction

Machine "M"

Code Generation for Language **While**

Code Generation for Language **Block**

Code Generation for Language **Proc**

Outline - Generation of Assembly Code

Introduction

Machine "M"

Code Generation for Language **While**

Code Generation for Language **Block**

Code Generation for Language **Proc**

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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 consumption, ...
- ▶ and **security** w.r.t. external (cyber-)attacks
hardened code, no information leakage, checks for vulnerability detection, ...

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

1 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

2 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly 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.

Remark Other intermediate representations are used in the optimization phases.

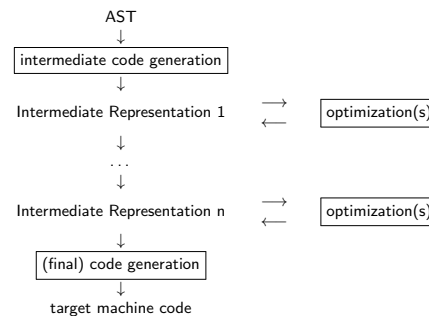
Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

4 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

A pragmatic approach



Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

3 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

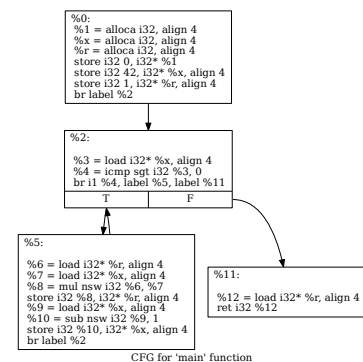
Generation of Assembly Code

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 ;
}

```



Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

5 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

Example 2: Java ByteCode (stack machine)

```
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
```

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

6 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

7 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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 (statically computed, stored in the symbol table)
- ▶ Addressing modes:
Ri, val (immediate), Ri +/- Rj, Ri +/- offset

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

7 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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	$Ri \leftarrow Rj \text{ oper } Rk$
OPER Ri, Rj, val	$Ri \leftarrow Rj \text{ oper val}$
CMP Ri, Rj	$Ri - Rj$ (set cond flags)
LD Ri, [adr]	$Ri \leftarrow \text{Mem}[\text{adr}]$
ST Ri, [adr]	$\text{Mem}[\text{adr}] \leftarrow Ri$
BRANCH label	if cond then $PC \leftarrow \text{label}$ else $PC \leftarrow PC + 4$

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

8 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

9 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

10 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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.
→ distinction between boolean and arithmetic expr. □

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

10 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

Language While

Informal code generation

Informal code generation

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

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

11 | 58

Functions for code generation

Notation

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

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

GCBExp : **BExp** $\times$ **Label** $\times$ **Label** $\rightarrow$ **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 : $\rightarrow$ **Reg**
allocates a new register Ri

newLabel : $\rightarrow$ **Labels**
produces a new label

GetOffset : **Var** $\rightarrow$ $\mathbb{Z}$
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	(C,i) = GCAExp(e), k=GetOffset(x)
		in	C ST Ri, [FP + k]
GCStm (s ₁ ; s ₂)	=	Let	C ₁ = GCStm(s ₁), C ₂ = GCStm(s ₂)
		in	C ₁ C ₂
GCStm (while e do s od)	=	Let	lb=newLabel(), ltrue=newLabel(), lfalse=newLabel()
		in	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	lnext=newLabel(), ltrue=newLabel(), lfalse=newLabel()
		in	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 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)

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), C ₁ C ₂
		in	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 e₁ > e₂, e₁ ≥ e₂, 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

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Blocks

Generation of Assembly Code

Syntax

$$S ::= \dots \mid \text{begin } D_V ; S \text{ end}$$

$$D_V ::= \text{var } x \mid D_V ; D_V$$

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**

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 22 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Storing local variables in memory - Example 1

Access to local variables within a block

Generation of Assembly Code

```
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 ;`: $\text{GetOffset}(z) = -4$, $\text{GetOffset}(y) = -8$, $\text{GetOffset}(x) = -12$.

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 23 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Storing local variables in memory - Example 2

Access to local variables in case of nested blocks

Generation of Assembly Code

```
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**)

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 24 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Structure of the memory

Generation of Assembly Code

- 1: global variables
- 2: **execution stack**, SP = last occupied address
- 3: heap (for dynamic allocation)

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 25 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Code generation for variable declarations

Generation of Assembly Code

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

SizeDecl(d) computes the size of declarations d

SizeDecl (<code>var x</code>)	=	4 (x of type Int)
SizeDecl (<code>d₁ ; d₂</code>)	=	Let $v_1 = \text{SizeDecl}(d_1)$, $v_2 = \text{SizeDecl}(d_2)$ in $v_1 + v_2$

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 26 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Code generation for blocks

Generation of Assembly Code

```
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 ||
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 27 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

With the help of some auxiliary functions ...

Generation of Assembly Code

prologue(size)	epilogue	push register (Ri)
ADD SP, SP, -4	ADD SP, FP, 0	ADD SP, SP, -4
ST FP, [SP]	LD FP, [SP]	ST Ri, [SP]
ADD FP, SP, 0	ADD SP, SP, +4	
ADD SP, SP, -size		

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 28 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Access to variables from a block?

Generation of Assembly Code

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

What is the memory address of x ?

- ▶ if x is a **local** variable (w.r.t the current block)
⇒ $\text{adr}(x) = \text{FP} + \text{GetOffset}(x)$
- ▶ if x is a **non local** variable
⇒ it is defined in a "nesting" memory env. E
⇒ $\text{adr}(x) = \text{adr}(E) + \text{GetOffset}(x)$
 $\text{adr}(E)$ can be accessed through the "Ariadne link" ...

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 29 | 58

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:

- ▶ $r \leq p$
- ▶ $n = p - r$

⇒ 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)

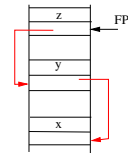
```
begin
  var x ; /* env. E1, nesting level = 1 */
  begin
    var y ; /* env. E2, nesting level = 2 */
    begin
      var z ; /* env. E3, nesting level = 3 */
      x := y + z /* s, nesting level = 3 */
    end
  end
end
```

From statement s :

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

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

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 $r < p$:

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

Outline - Generation of Assembly Code

Introduction

Machine “M”

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

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
```

⇒ all variable addresses 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 addresses at runtime

Demo: code generation for nested blocks

Outline - Generation of Assembly Code

Introduction

Machine “M”

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

Syntax of Language Proc

Reminder

Procedure declarations:

```
Pgm ::= ... | begin DV ; DP ; S end
DP ::= proc p (FPL) is DV S end DP | ε
FPL ::= x, FPL | ε
```

Statements:

```
S ::= ... | call p(EPL)
EPL ::= AExp, EPL | ε
```

FPL : 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

Introduction

Machine "M"

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

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

Introduction

Machine "M"

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

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

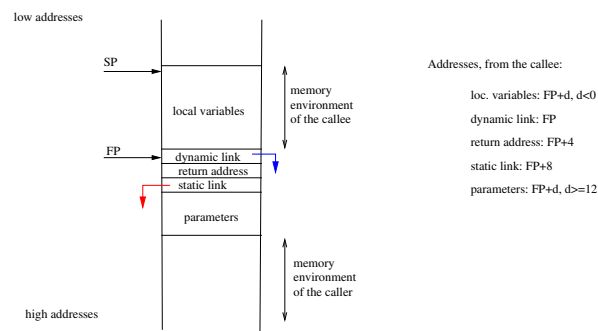
When it ends, the callee:

- ▶ 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



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 var_i, as before, $\text{GetOffset}(\text{var}_i)$ is $-4 \times i$.
- ▶ For parameter param_i, $\text{GetOffset}(\text{param}_i)$ 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	$R \leftarrow \text{@(label)}$
CALL label	branch to the procedure labelled with label $\text{PUSH(PC)} \parallel \text{PC} \leftarrow \text{label}$
CALL R	branch to the address contained in register R $\text{PUSH(PC)} \parallel \text{PC} \leftarrow R$
RET	end of procedure: $\text{PC} \leftarrow \text{MEM[SP]} \parallel \text{SP} \leftarrow \text{SP} + 4$

Code generation for a procedure declaration

GCProc : $D_P \rightarrow \text{Code}^*$

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
```

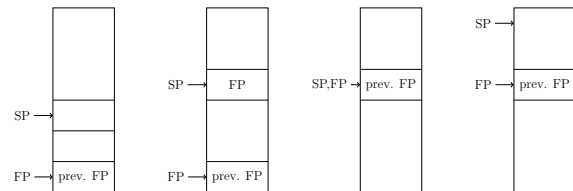
RET:

```
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

Introduction

Machine "M"

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

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 : $E_P \rightarrow \text{Code}^* \times \mathbb{N}$

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 whose 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

Introduction

Machine "M"

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

49 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

50 | 58

Outline - Generation of Assembly Code

Introduction

Machine "M"

Code Generation for Language While

Code Generation for Language Block

Code Generation for Language Proc

Syntax of Proc

Issues for Code Generation

A Protocol between the Caller and Callee

Code Generation

Other Modes for Passing Parameters

Example / Exercises

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.

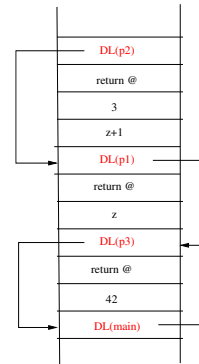
Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

51 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

Execution stack during the execution of p2



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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

52 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

The block body

Global variable z is at address $\text{@glob} - 4$

```
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()
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

53 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Generation of Assembly Code

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()
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

54 | 58

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC Generation of Assembly Code First example Code for procedure p2 <pre> proc p2(x, y) is z := x + y ; end 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() </pre> Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 55 58	Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC Generation of Assembly Code First example Code for procedure p3 <pre> proc p3 (x) is begin var z ; call p1() ; z := z+x ; end 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() </pre> Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 56 58
Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC Generation of Assembly Code 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/) Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 57 58	Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC Generation of Assembly Code 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. Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 58 58