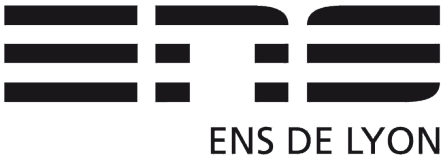


Compilation et Analyse de Programmes (CAP)

Exercise book

Contents

Chapter 1
Architecture (TARGET18)

EXERCISE #1 ▶ **TD : print 'Z'**
Write a program in TARGET18 assembly that writes the character 'Z' 10 times in the output terminal (use the print instruction).

EXERCISE #2 ▶ **Add one to array elements**

Write a program TARGET18 that adds all the elements of an array in memory (between "beg" and "end" addresses):

```
; ;;  add array elements – Student File

; ;;    ADD INSTRUCTIONS HERE




        lea r0 msg
        print string r0
        print signed r1                 ;print the result

halt:
jump halt                               ; stop

beg:
        .const 4 #0010                  ; 2
        .const 4 #0000          ; 0
        .const 4 #0001          ; 1
end:
        .const 4 #1111          ;-1

msg:
        .string "And the result is "
```

Chapter 2
Grammars and attributes

EXERCISE #1 ▶ **Declarations of variables**
Write a grammar that accepts declarations of variables like:

```
int x=1;
float y,z;
int t;
float u,v=0;
```

and rejects:

```
int x, int y;
```

Then write an attribution that prints individual declarations (of the first case) like:

```
int x=1; float y; float z; int t; float u; float v=0;
```

EXERCISE #2 ▶ **XML Files**

We give the following grammar:

```
L ->E L
|
E -> A L B
|    ident
A -> < ident >
B -> </ ident >
```

1. Give the derivation tree for the chain <html><head>toto</head>titi</foo>.
2. Attribute this grammar to verify that opening and closing tags refer to the same identifiers.

Chapter 3
Semantics

Abstract syntax

Recall the abstract syntax of the course for expressions:

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e+e & \textit{addition} \\ & | & e\times e & \textit{multiplication} \\ & | & \ldots & \end{array}$$

and the mini-while language:

$$\begin{array}{rcll} S(Smt) & ::= & x := expr & \text{assign} \\ & | & skip & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \\ & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

EXERCISE #1 ▶ **Semantics of arithmetic expressions**
Show the two following properties:

1. Let $a \in$ **Aexp** a given arithmetic expression. Let σ, σ' be two states. Show that if $(\forall x \in X, \sigma(x) = \sigma'(x))$, then $\mathscr{A}[a]\sigma = \mathscr{A}[a]\sigma'$.
2. Let $a' \in$ **Aexp**, show that:
$$\mathscr{A}[a[a'/x]]\sigma = \mathscr{A}[a]\sigma[x \mapsto \mathscr{A}[a']\sigma]$$

EXERCISE #2 ▶ **Repeat**
We want to add the command repeat S until b to the mini-while language seen in the course.

1. Give semantics rules to define repeat S until b without using `while`.
2. Show that the constructions:
   (a) repeat S until b and
   (b) S; `if` b `then` skip else (repeat S until b).

   are semantically equivalent.
3. Give a program transformation to transform any program with the repeat S until b construction into another one without this construction. You can use the `while` construction, of course.

Chapter 4
Static Semantics/Typing

Abstract syntax

Recall the abstract syntax of the course for expressions:

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e+e & \textit{addition} \\ & | & e\times e & \textit{multiplication} \\ & | & \ldots & \end{array}$$

and the mini-while language:

$$\begin{array}{rcll} S(Smt) & ::= & x := expr & \text{assign} \\ & | & skip & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \\ & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

We expand the language with variable declarations:

$$D(decl) \quad ::= \quad \texttt{var } x : t \quad \text{type declaration}$$

We recall that environnements associate a type to variables (Γ). Here, the environnement is constructed by the following rules:

Declarations

$$\frac{}{\texttt{var } x : t \rightarrow_d [x \mapsto t]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Expressions Like in the course, for instance:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

Commands Like in the course, for instance:

$$\frac{\Gamma \vdash b : \texttt{boolean} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while } b \texttt{ do } S \texttt{ done} : \texttt{void}}$$

And a program

$$\frac{D \rightarrow \Gamma \quad \Gamma \vdash C : \texttt{void}}{\Gamma \vdash DC : \texttt{void}}$$

EXERCISE #1 ► **Well typed**
Type the program:

```
var x1 : integer ; var x2 : integer ; var x3 : boolean
x1 := 3 ;
while (not x3) do
x1 := x2 + 1 ;
x3 := x3 and true
done
```

EXERCISE #2 ► **Expand expressions**

Complete the abstract syntax and the static semantics (typing) of expressions with the new construction `e1? e2: e3` : if e_1 is true then the expression has value e_2 else e_3.

EXERCISE #3 ► **Expand the statements**

Complete the abstract syntax and the static semantics (typing) of statements with an extended for:

```
for i in e1 .. e2 S
```

2 cases:

- The instruction declares the i variable (like in Ada)
- i should be declared before.

Chapter 5
Code Generation

5.1 Rules for three address code generation

The code we generate will have an unbounded number of temporaries (`tmp0`, `tmp1`, ...) but actual TARGET18 instructions (`ADD`, `AND`, `JUMP` ...) or the conditional instruction.

The code generation functions have the following signatures (pseudo-code is given in a companion file):

`GenCodeExpr` : AExp $\rightarrow$ Code* $\times$ $\mathbb{N}$
`GenCodeSmt` : Inst $\rightarrow$ Code*

where $Code*$ is a sequence of 3-address instructions (TARGET18 with temporaries). As a side effect, the code generation for statements might update a map $Var \rightarrow \mathbb{N}$ (program variable to a temporary where to find its current value).

Auxiliary functions:

`newTemp()` : $\rightarrow \mathbb{N}$
`newLabel()` : $\rightarrow \mathbb{N}$

5.2 Code generation for Mu

EXERCISE #1 ▶ **By hand!**
Using the code generation rules, generate the three-address code for the following (mini-while) program:

```
var x1,x2: int;
x1 = 3;
x2 = 7 + x1;
while (x2<x1) do
   x2 = x2-1;
```

5.3 Language expansions

EXERCISE #2 ▶ **Rules for boolean expressions**
Write a code generation rule for the `xor` boolean operator.

EXERCISE #3 ▶ **Rules for statements**
Write a code generation rule for the `repeat S until e` statement.

c	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, c))`
	`return dr`

x	`#get the place associated to x.`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, 1))`
	`return dr`

$e_1 < e_2$	`dr <- newTemp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- newLabel()`
	`code.add(InstructionLETI(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, 'sge' , t2)`
	`code.add(InstructionLETI(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

Figure 5.1: 3@ Code generation for numerical or Boolean expressions (t1 and t2 are already defined)

x = e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`find loc the location for var x`
	`code.add(instructionLET(loc,dr))`

S1; S2	`#concat codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` *b* `then` *S*1 `else` *S*2	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "eq", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` *b* `do` *S* `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "eq", 0))`
	`GenCodeSmt(S) #execute S`
	`code.add(InstructionJUMP(ltest)) #and jump to the test`
	`code.addLabel(lendwhile) #else it is done.`

Figure 5.2: 3@ Code generation for Statements

Chapter 6
Intermediate Representations

6.1 Instruction Scheduling

EXERCISE #1 ▶ **Sethi-Ullman**
Consider the expression $E = ((a+b)*(a-b))+1$ where a and b are stored in **stack slots**. a and b will be refered as $[a]$ and $[b]$ in the load instruction.

1. What is the minimum amount of registers required to evaluate E?
2. Give the corresponding TARGET18 code (with temporaries, not physical registers).
3. Draw the liveness intervals for your code. Show an execution point with a maximum number of temporaries alive. Conclusion?

6.2 SSA form

EXERCISE #2 ▶ **Convert!**
Consider the following program:

```
a = 3;
b = 2;
while (a < 15) {
  c = 0 ;
  if (b < 6) {
    a = a * 2 ;
    b = b + 1 ;
  }
  else {
    a = a + 1 ;
    b = b * 2 ;
  }
  c = c + a;
}
return a + (b + c);
```

1. Construct the CFG.
2. Translate into SSA form.

Chapter 7
Liveness, Dataflow analysis, Register Allocation

7.1 Liveness analysis

Let us recall the notations here: A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this bloc (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

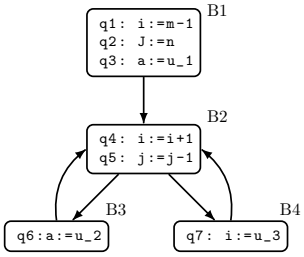
EXERCISE #1 ▶ **Live variables**
Generate the CFG for the following program:

```
while d>0 do {
    a:=b+c;
    d:=d-b;
    e:=a+f;
    if e>0 then {
        f:=a-d;
        b:=d+f;
    }
    else{
        e:=a-c;
        }
    b:=a+c;
}
```

On this CFG:

- Write and solve the equations defining the sets Gen, $Kill$, In and Out for the liveness analysis, pay attention to initialisations.
- Suppress the dead code.

EXERCISE #2 ▶ **Live Variables**
After code generation, we obtain the following graph:



On this graph, perform liveness analysis and suppress the dead code.

7.2 Available expressions

We recall:

$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} \end{cases}$$

$$AE_{exit}(\ell) = \left(AE_{entry}(\ell) \setminus kill_{AE}(\ell)\right) \cup gen_{AE}(\ell)$$

EXERCISE #3 ▶ **Common (sub) expressions**
On the following 3-address code:

```
(1)   a=b+c
(2)   b=a+c
      d=c+e
      si d>7 aller a (8)
      t=a+c
      v=c+e
      aller a (10)
(8)   t=b+c
      v=a+c
(10)  b=a+c
      a=b+c
```

1. Construct the CFG.
2. For each block, compute the sets `gen` and `kill` sets for common subexpressions.
3. Compute the set of all available expressions at the entry and exit of each block.
4. Optimise the code.

EXERCISE #4 ▶ **With a loop**
Same questions with:

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

7.3 Register Allocation

EXERCISE #5 ► **Code production and register allocation**

Consider the expression $E = ((n * (n+1)) + (2 * n)$. We assume that we have:

- A multiplication instruction `mul` t1,t2,t3 that computes t1 := t2*t3.
- A "immediate load" instruction `ldi` t1 4.
- The variable n is stored in the stack slot referred as $[n]$ in the load instruction.

1. Generate a 3 address-code with temporaries and `ldr` instruction to load n. Do it as blindly as possible (no temporary recycling).
2. (Without applying liveness analysis) Draw the liveness intervals. How many registers are sufficient to compute this expression?
3. Draw the interference graph (nodes are variables, edges are liveness conflicts).
4. Color this graph with three colors using the algorithm seen in the course.
5. Give a register allocation with $K = 2$ registers using the *iterative* register allocation algorithm seen in course.