



Compilation et Analyse de Programmes (CAP)

Exercise book

Contents

1 Architecture (TARGET18)	3
2 Grammars and attributes	4
3 Semantics	5
4 Static Semantics/Typing	6
5 Code Generation	8
5.1 Rules for three address code generation	8
5.2 Code generation for Mu	8
5.3 Language expansions	8
6 Intermediate Representations	11
6.1 Instruction Scheduling	11
6.2 SSA form	11
7 Liveness, Dataflow analysis, Register Allocation	12
7.1 Liveness analysis	12
7.2 Available expressions	13
7.3 Register Allocation	15
8 Additional exercises	16
9 Abstract Interpretation	18
10 Procedures: semantics, code generation	20
11 Abstract Machines	21

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):

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

   ;; ADD INSTRUCTIONS HERE

6

11

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

halt:
21  jump halt                    ; stop

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

msg:
31  .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:

$$\begin{array}{l} e ::= c \quad \text{constant} \\ | x \quad \text{variable} \\ | e + e \quad \text{addition} \\ | e \times e \quad \text{multiplication} \\ | \dots \end{array}$$

and the mini-while language:

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

EXERCISE #1 ► Semantics of arithmetic expressions

Show the two following properties:

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

$$\mathcal{A}[a[a'/x]]\sigma = \mathcal{A}[a]\sigma[x \mapsto \mathcal{A}[a']\sigma]$$

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:

$$\begin{array}{l}
 e ::= c \quad \text{constant} \\
 | x \quad \text{variable} \\
 | e + e \quad \text{addition} \\
 | e \times e \quad \text{multiplication} \\
 | \dots
 \end{array}$$

and the mini-while language:

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

We expand the language with variable declarations:

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

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

Declarations

$$\frac{\text{var } x : t \rightarrow_d [x \mapsto t]}{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) = \emptyset} D_1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2$$

Expressions Like in the course, for instance:

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

Commands Like in the course, for instance:

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

And a program

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

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 $e_1? e_2: e_3$: 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	<pre>dr <-newTemp() code.add(InstructionLETI(dr, c)) return dr</pre>
x	<pre>#get the place associated to x. regval<-getTemp(x) return regval</pre>
e_1+e_2	<pre>t1 <- GenCodeExpr(e_1) t2 <- GenCodeExpr(e_2) dr <- newTemp() code.add(InstructionADD(dr, t1, t2)) return dr</pre>
e_1-e_2	<pre>t1 <- GenCodeExpr(e_1) t2 <- GenCodeExpr(e_2) dr <- newTemp() code.add(InstructionSUB(dr, t1, t2)) return dr</pre>
true	<pre>dr <-newTemp() code.add(InstructionLETI(dr, 1)) return dr</pre>
$e_1 < e_2$	<pre>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</pre>

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

x = e	<pre> dr <- GenCodeExpr(e) #a code to compute e has been generated find loc the location for var x code.add(instructionLET(loc,dr)) </pre>
S1; S2	<pre> #concat codes GenCodeSmt(S1) GenCodeSmt(S2) </pre>
if b then S1 else S2	<pre> 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) </pre>
while b do S done	<pre> 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. </pre>

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 referred 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 assignment is *killed* by the block. A variable whose value is used in this bloc (before any assignment) is *generated*.

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

$$LV_{entry}(\ell) = (LV_{exit}(\ell) \setminus kill_{LV}(\ell)) \cup gen_{LV}(\ell)$$

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

EXERCISE #1 ► Live variables

Generate the CFG for the following program:

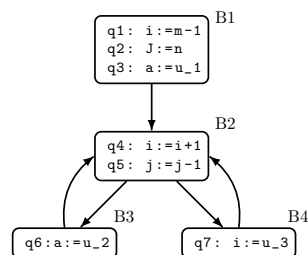
```
while d>0 do {
  a:=b+c;
  d:=d-b;
  e:=a+f;
  if e>0 then {
    f:=a-d;
    b:=d+f;
  }
  else{
    e:=a-c;
  }
  b:=a+c;
}
```

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:

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

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

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

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.

Chapter 8

Additional exercises

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 referred 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?

EXERCISE #2 ► Constant propagation - Bonus

In this exercise we will study the following program:

```
z:=3
x:=1
while (x>0) {
  if (x=1) then
    y:=7;
  else
    y:=z+4;
  x:=3
  print y
}
```

1. Draw the CFG where blocks are statements.
2. Which of variables x, y, z are constants? What modification can be done on the code if we know this information?
3. How is a "i'm a constant" information generated? killed? Is it a forward or backward dataflow propagation?
4. How to merge two incoming informations $x \mapsto 1$ and $x \mapsto 2$ in a test?
5. Give an algorithm to compute all the constants available at the beginning of each block.

EXERCISE #3 ► Adapted from exam 2018

We consider the following Mu program:

```
var x,y,z,t:int;
x=12; y=3+x; z=4+y; t=x-y+z;
```

For readability reasons, $temp_i$ is renamed into t_i . The 3 address code generation process of Lab 4/5 produces the following code, where $(t, z, y, x) \mapsto (t1, t2, t2, t3)$:

```
.let t4 12
copy t3 t4
3 .let t5 3
  add t6 t5 t3
  copy t2 t6
  .let t7 4
  add t8 t7 t2
8 copy t1 t8
  sub t9 t3 t2
```



```
add t10 t9 t1
copy t0 t10
```

1. Fill the array with the result of the liveness analysis. Each star in a line will mean “the temporary is alive at the entry of this line”:

code	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
.let t4 12										
copy t3 t4										
.let t5 3										
add t6 t5 t3										
copy t2 t6										
.let t7 4										
add t8 t7 t2										
copy t1 t8										
sub t9 t3 t2										
add t10 t9 t1										
copy t0 t10										

2. Draw the interference graph.

In the rest of the exercise we will use the following notations: • Color 1 is red, and is associated to register R3; • Color 2 is blue, and is associated to register R4; • Color 3 is green, and is associated to register R5;

3. Color the graph with the heuristic of the course and 2 colors. Do not forget to draw the color stack.
4. What are the variable(s) to spill ? Allocate this/these variable in memory without live range splitting (second algorithm of the course) and generate code for the associated instructions.

Chapter 9

Abstract Interpretation

EXERCISE #1 ► **Parity**

Properly define the *parity* abstract domain. Illustrate on examples how it works on Mu (including modulo operators and a new constructor `random`).

EXERCISE #2 ► **Intervals**

Properly define the interval abstract domain. Illustrate the behavior on the simple example:

```
x:=4;
while (x<100) x:=2+x;
```

EXERCISE #3 ► **Intervals for arrays**

On a well-chosen example, illustrate how the interval domain can be used to detect illegal accesses in arrays.

EXERCISE #4 ► **Variations on intervals and planes**

In a big program, we find:

```
i=0;
while (true) {
  /* pilot the plane */
  /* without touching variable i */

  i++;
  if (i >= 20) {
    i=0;
  }
}
```

Somewhere in the loop there are some accesses `t[i]` in a circular buffer implemented as an array `t`, and the correct indices are `0..19`. We thus have to print warnings if we cannot prove that they indeed are in this interval.

- Compute the successive iterations for intervals on `i`, with the standard widening. what interval do you find?
- apply one more loop iteration from the invariant you obtained. It is satisfying ?

Now consider the following variant:

```
i=0;
while (true) {
  /* pilot the plane */
  /* without touching variable i */

  i++;
  if (i == 20) {
    i=0;
  }
}
```

Same questions. Give a cheap solution to the problem.

EXERCISE #5 ▶ A lack of relationship

```

/* x is in [-3, 6] */
y = x;

/* bla bla */

z = 1+x*y;
y = sqrt(z);

```

Compute intervals (forwards) for y , z ? Will there be a warning for taking the square root of a possibly negative number? Is there a problem? Can you imagine a solution?

EXERCISE #6 ▶ [Exam 2016]

source P. Cousot

A multiplication $m \times n$ can be checked by summing the digits of integer m modulo 9, summing the digits of n modulo 9, and checking that their product modulo 9 is equal to the sum of the digits of the result r modulo 9. For example,

$$\begin{array}{r}
 1234 \rightarrow 10 \bmod 9 = 1 \\
 \times 5678 \rightarrow 26 \bmod 9 = 8 \\
 \hline
 = 7006652 \rightarrow 26 \bmod 9 = 8
 \end{array}$$

is a success, but

$$\begin{array}{r}
 1234 \rightarrow 10 \bmod 9 = 1 \\
 \times 5678 \rightarrow 26 \bmod 9 = 8 \\
 \hline
 = 7006651 \rightarrow 27 \bmod 9 = 7 \neq 1 \times 8 \bmod 9
 \end{array}$$

fails.

- Is a success a proof? Is a failure a proof?
- Show that this is an abstraction. Which property of modular arithmetic do you use?
- Generalise this idea and construct an abstract domain \mathcal{A} of congruences modulo 9.
- Give an example of a program and an assertion that will be proved non reachable with your new abstract domain.
- Extend to a congruence domain.

Chapter 10

Procedures: semantics, code generation

Abstract syntax:

$$\begin{aligned} S &\in \mathbf{Stm} \\ S &::= x := a \mid \text{skip} \mid S_1; S_2 \mid \\ &\quad \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \\ &\quad \text{while } b \text{ do } S \mid \\ &\quad \text{begin } D_V D_P; S \text{ end} \mid \\ &\quad \text{call } p \\ D_V &::= \text{var } x := a; D_V \mid \varepsilon \\ D_P &::= \text{proc } p \text{ is } S; D_P \mid \varepsilon \end{aligned}$$

EXERCISE #1 ► **Operational semantics**

For the following miniwhile with procedures program:

```
(0) begin
(1)   var x:=0
(2)   proc p is x:=x*2
(3)   proc q is call p
(4)   begin
(5)     var x:=5
(6)     proc p is x:=x+1
(7)     call q
(8)     y:=x
(9)   end
(10)end
```

Apply the semantic rules of the course with:

- A Dynamic binding for variables and procedures.
- Dynamic binding for variables, and static for procedures.

EXERCISE #2 ► **Procedures with parameters**

Add a new abstract syntax for the mini-while with procedures with two parameters. Give a formal “call by value” semantics (environment, update, call). Give an example.

EXERCISE #3 ► **Code generation**

Generate the code for the following program:

```
begin
  var x,z:int
  procedure p is
  begin
    var y:int
    y := 5;
    z := x+y;
  end
  x:=3;
  call p;
  log(z);
end
```

Chapter 11

Abstract Machines

EXERCISE #1 ► Let in

The compilation of `let y = 40 in y+2` gives the following (abstract) code: **Cst 40;Let "y";Cst 2; Access "y"; Add; EndLet**. Using the execution rules of the course, execute it step by step:

Code	Env.	Stack
Cst 40;Let "y";Cst 2; Access "y"; Add; EndLet	\emptyset	\emptyset
Let "y"; ... EndLet		
Cst 2; ... EndLet		
Access "y"; Add; EndLet		
Add; EndLet		
EndLet		
(end)		

EXERCISE #2 ► Functions

The compilation of `(fun x-> x+1) 42` gives the following abstract code to be executed.

Cst 41;Closure("x",Acces x;Cst 1; Add; Ret);App

Code	Env.	Stack
Cst 41;Closure("x",Acces x;Cst 1; Add; Ret);App	\emptyset	\emptyset
Closure("x",Acces x;Cst 1; Add; Ret); App	\emptyset	Vint(41)