



Compilation et Analyse de Programmes (CAP)

Exercise book

Contents

1 Architecture (LEIA)	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	9
7 Liveness, Dataflow analysis, Register Allocation	10
7.1 Liveness analysis	10
7.2 Available expressions	11
7.3 Register Allocation	13
8 Procedures: semantics, code generation	14
9 Abstract Interpretation	16
10 Hoare	18
11 Abstract Machines	19

Chapter 1

Architecture (LEIA)

EXERCISE #1 ► **TD: print 'Z'**

Write a program in LEIA assembly that writes the character 'Z' 10 times in the output terminal (use the write instruction).

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

Write a program LEIA 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  print 'And the result is'      ;print message
   print r1                      ;print the result
   jump 0                        ; stop

11 beg:
   .word 2
   .word 3
   .word 4
   end:
16 .word 5
```

Chapter 2

Gammars 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 : integer
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 LEIA 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 (LEIA 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.