

Programming-Language Semantics and Compiler Design (Sémantique des Langages de Programmation et Compilation)

Tutorials (Travaux Dirigés)

Univ. Grenoble Alpes — UFR IM²AG

Master of Science in Informatics at Grenoble (MoSIG)
Master 1 informatique (M1 info)

Academic Year 2017 - 2018

CONTENTS

1	Introduction	5
1.1	Lexicon, Syntax, and Semantics	5
1.2	Grammars and Abstract syntax	5
1.3	A Simple Compiler	7
1.3.1	Introduction	7
1.3.2	Lexical Analysis	8
1.3.3	Syntactic Analysis	8
1.3.4	Type checking	9
1.3.5	Code Generation	9
1.3.6	Global Optimizations Independent from the Machine	10
2	Semantic Analysis	11
3	Natural Operational Semantics of Language While	17
3.1	Properties of Arithmetical and Boolean Expressions	17
3.1.1	Free Variables	17
3.1.2	Substitution	17
3.2	Natural Operational Semantics of (Pure) Language While	18
3.3	Natural Operational Semantics of Extensions of Language While	19
4	Natural Operational Semantics of Languages Block and Proc	21
4.1	Natural Operational Semantics of Language Block	21
4.2	Natural Operational Semantics of Language Proc	21
5	Structural Operational Semantics	25
6	Axiomatic semantics - Hoare Logic	27
7	Provably-Correct Implementation	31
7.1	The Abstract Machine AM	31
7.2	Properties of AM	31
7.3	Code generation	32
7.4	Correctness of code generation	33
8	Intermediate-Code Optimization using Data-flow Analysis	35
8.1	Defined and Used variables	35
8.1.1	Defining assigned and used variables in statements	35
8.1.2	Adding an operator for parallel execution	35
8.2	Preliminaries	36
8.2.1	Lattice, recursive equations, fix-points	36
8.2.2	Control-flow Graph (CFG)	37
8.3	Available Expressions	37
8.4	Live Variables	38

CONTENTS

8.5 Constant Propagation	39
9 Code Generation (of Assembly Code)	41

INTRODUCTION, REMINDERS, AND COMPILER ARCHITECTURE

1.1 Lexicon, Syntax, and Semantics

Exercise 1 — Compilation errors

Give examples of code snippets (in any programming language) that produce the following errors.

- | | |
|-----------------------|-------------------------------|
| 1. A lexical error. | 3. A static-semantics error. |
| 2. A syntactic error. | 4. A dynamic-semantics error. |

1.2 Grammars and Abstract syntax

Exercise 2 — From languages to grammars

We consider vocabulary $V = \{a, b, c\}$. For each of the following languages, propose a grammar.

- | | |
|--|---|
| 1. $L_1 = a^* \cdot b \cdot c^*$ | 3. $L_3 = \{a^n \cdot b \cdot c^m \mid 0 < n < m\}$ |
| 2. $L_2 = \{a^n \cdot b \cdot c^n \mid n \geq 0\}$ | 4. $L_4 = \{w \mid w \text{ is a palindrome}\}$ |

Exercise 3 — Automata and grammars for regular languages

We consider regular language $L_0 = a^* \cdot b^*$. Consider grammar G_0 defined by the following production rules:

$$S \rightarrow A \cdot B; \quad A \rightarrow a \cdot A \mid \epsilon; \quad B \rightarrow b \cdot B \mid \epsilon$$

where S is the axiom describing language L_0 . The finite-state automaton $A_0 = (Q, V, \delta, q_0, F)$ recognizes this language:

- | | | |
|----------|-------|---|
| Q | $=$ | $\{q_0, q_1\}$ is the set of states |
| V | $=$ | $\{a, b\}$ is the input alphabet |
| q_0 | \in | Q is the initial state |
| F | $=$ | $\{q_1\}$ is the set of accepting states |
| δ | $=$ | $\{(q_0, a, q_0), (q_0, \epsilon, q_1), (q_1, b, q_1)\}$ is the transition relation |

1. Give a derivation of grammar G_0 for string $aaaabb$.

2. Give the sequence of transitions of automaton A_0 allowing to recognize the string $aabb$.
3. Give the sequence of configurations of automaton A_0 allowing to recognize the string $aabb$.

Exercise 4 — Context-free languages and grammars

We consider the context-free language $L_1 = \{a^n \cdot b^n \mid n \geq 0\}$. Consider grammar G_1 defined by the following production rules: $S \rightarrow a S b$ and $S \rightarrow \epsilon$, where S is the axiom describing language L_1 . The push-down automaton $A_1 = (Q, V, \Gamma, \Delta, Z, q_0, F)$ recognizes this language.

- $Q = \{q_0, q_1, q_2, q_3\}$ is the set of states
- $V = \{a, b\}$ is the input alphabet
- $\Gamma = \{Z, A\}$ is the stack alphabet
- $Z \in \Gamma$ is the initial stack symbol
- $q_0 \in Q$ is the initial state
- $F = \{q_3\}$ is the set of accepting states
- $\Delta = \{ (q_0, a, Z) \rightarrow (q_1, Z A),$
 $(q_1, a, u) \rightarrow (q_1, \omega, u A),$
 $(q_1, b, u A) \rightarrow (q_2, u),$
 $(q_2, b, u A) \rightarrow (q_2, u),$
 $(q_2, \epsilon, Z) \rightarrow (q_3, \epsilon) \}$
- Δ is the set of transitions, with $\omega \in V^*$ and $u \in \Gamma^*$

1. Give the derivation of grammar G_1 for string $aabb$.
2. Give the sequence of configurations of automaton A_1 allowing to recognize string $aabb$.

Exercise 5 — Grammars for arithmetic expressions

We want to define a grammar purposed to describe *arithmetic expressions* built from binary operators for subtraction ($-$) and multiplication ($*$) and for which operands are integers (e).

1. We first consider grammar G_0 with the following production rules:
 $Z \rightarrow E \quad E \rightarrow E - E \quad E \rightarrow e$
 Using sequence “10 - 2 - 3”, show that this grammar is ambiguous.
2. To eliminate the ambiguity of G_0 , we consider grammar G_1 , defined as follows:
 $Z \rightarrow E \quad E \rightarrow E - T \quad E \rightarrow T \quad T \rightarrow e$
 Draw the derivation tree corresponding to the example from the previous question. We say here that subtraction is “left associative”.
3. How G_1 should be modified to introduce a multiplication operator such that:
 - multiplication is left associative;
 - multiplication has precedence over subtraction.
 Justify your answer by constructing syntactic trees for the following expressions: “10 - 2 * 3”, “10 * 2 - 3”.
4. We want to allow to put a sub-expression between parenthesis so as to write e.g., “(10 - 2) * 3” or “10 * (2 - 3)”. Modify the previous grammar. Build the syntactic tree for the expression given as examples.
5. We add the unary minus operator (noted $-$), that has precedence over $-$ and $*$. Modify the previous grammar accordingly. Build the syntactic tree for “10 * -2 - -3”.

Exercise 6 — Grammars and ambiguity

We consider the following grammar that describes statements from a programming language:

$$\begin{aligned} Z &\longrightarrow I \\ I &\longrightarrow \text{if } e \text{ then } I \\ I &\longrightarrow \text{if } e \text{ then } I \text{ else } I \\ I &\longrightarrow a \end{aligned}$$

1. Show that this grammar is ambiguous: find a sentence from the language that leads to two different derivation trees.
2. Propose a solution to make this grammar non-ambiguous (by for instance modifying the described language).
3. Is it possible to make this grammar non-ambiguous without modifying the described language?

Exercise 7 — From derivation trees to machine code

Let G be a grammar describing assignments where terminals c and i designate an integer constant and an identifier, respectively.

$$\begin{aligned} Z &\rightarrow A ; & E &\rightarrow E + T \mid T & F &\rightarrow c \mid i \mid (E) \\ A &\rightarrow i := E & T &\rightarrow T * F \mid F \end{aligned}$$

We consider the two following assignments where x and y are identifiers:

- (1) $x := 5 + x * 2 + y;$ (2) $y := (5 + x) * 2 + y;$
1. Build the derivations trees corresponding to assignments (1) and (2).
 2. We consider a processor with registers noted R_i and with the following instruction set:
 - LD R_i, op loads the value of op in register R_i , op denotes a variable or a constant,
 - ST R_i, x gives the value contained in register R_i to variable x ,
 - ADD $R_i, op1, op2$ puts the sum $op1 + op2$ in register R_i ,
 - MULT $R_i, op1, op2$ puts the product $op1 * op2$ in register R_i ,
where $op1$ denotes a register and $op2$ denotes a register or a constant.

Write the sequence of instructions corresponding to the traduction of (1) and (2).

3. We want to generate code for the machine from derivation trees. In the sequence proposed in the previous question, to which node should the instructions of the machine be associated with? Decorate the derivation trees of (1) and (2) with the sequences obtained: each instruction should decorate a node from the tree, certain nodes should not be decorated.
4. Inspiring from the result of question 3, draw simplified versions of trees (1) and (2). A simplified tree is an abstract tree containing only decorated nodes.
5. Propose an abstract syntax for G (in the form of a grammar). Propose several ways to write this grammar.

Exercise 8 — Statement for - concrete/abstract grammar

The following code snippets give examples of “for” statements in ADA and C :

```
for i in 1..N loop           for (i=0 ; i<N ; i++) {
  -- statements              /* statements */
end loop ;                  }
```

1. Propose a concrete syntax (even a simplified one) for statement “for” for each language.
2. Propose a corresponding abstract syntax.

1.3 A Simple Compiler

1.3.1 Introduction

The logical steps of a compiler usually consists of:

- | | |
|-----------------------|---------------------------------|
| 1. Lexical Analysis | 4. Intermediate Code Generation |
| 2. Syntactic Analysis | 5. Code Optimization |
| 3. Semantic Analysis | 6. (Target) Code Generation. |

A language, its grammar

```

Program      ::= Block
Block        ::= begin Declaration_list ; Statement_list end
Declaration_list ::= Declaration_list ; Declaration | Declaration
Declaration  ::= Idf := Expression : Type
Type         ::= integer
Statement_list ::= Statement_List ; Statement | Statement
Statement    ::= Idf := Expression | skip
              | if Expression then Statement else Statement endif
              | if Expression then Statement endif |
              while Expression do Statement done
Expression   ::= Disjunction
Disjunction  ::= Conjunction | Disjunction or Conjunction
Conjunction  ::= Comparison | Conjunction and Comparison
Comparison   ::= Relation | Relation = Relation
Relation     ::= Sum | Sum < Sum
Sum          ::= Term | Sum + Term | Sum - Term
Term         ::= Factor | Factor * Term
Factor       ::= Not Factor | Denotation | Idf | '('Expression')'
```

Abstract Grammar

```

S  ::= x := a | skip | S; S | if b then S else S | while b do S
a  ::= n | a + a | a - a | a * a | x
b  ::= true | false | a = a | a ≤ a | ¬b | b ∧ b
```

1.3.2 Lexical Analysis

A scanner (lexical analyzer) has as input a character string and outputs a pair lexical class, element of the class.

Exercise 9 — Lexical analysis

Determine the lexical classes and give them a specification as an automaton or a regular expression.

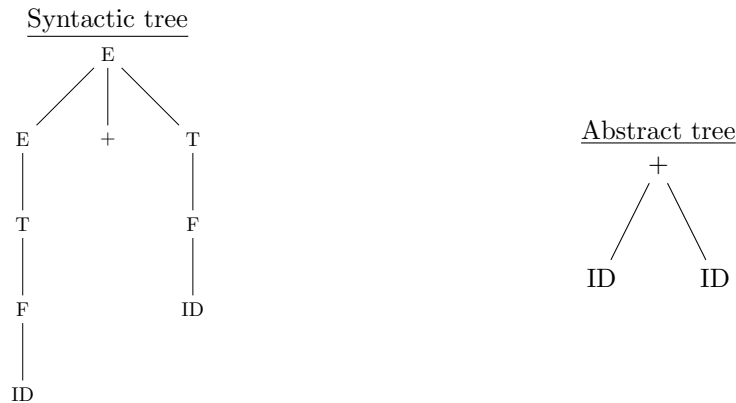
1.3.3 Syntactic Analysis

A parser (syntactic analyzer) has as input a flow of lexical classes and outputs an abstract tree described by the abstract grammar. The syntactic tree is obtained by rule derivation of the abstract grammar. The abstract tree is obtained from the derivation tree: it should contain operations as nodes and IDs as leaves. When reducing, the derivation tree to a syntax tree, terminals are “lifted” in the tree. You will use the following grammar for expressions:

```

E  → E + T | E - T | T
T  → T * F | F
F  → ID | NUM
```

For instance, for the expression $a + b$, we get:



Exercise 10 — Derivation trees and AST

Build the derivation trees and the abstract syntax trees corresponding to the expressions below.

1. $a + b - c$
2. $a - b + c$
3. $a + b * c$

1.3.4 Type checking

Type checking is performed on the abstract tree.

Exercise 11 — Rules for type checking

1. Give informally the rules for type checking. Rules should be given on the abstract grammar.

1.3.5 Code Generation

As input to code generation, we have an abstract tree. As output, we have a sequence of statements. Considering the abstract syntax of Section 1.3.1, we define the following generation functions:

- A function that takes as input a statement S , and that produces a sequence of machine statements.
- A function that takes an arithmetical expression, and that produces a sequence of machine statements and the register that contains the result.
- A function that takes as input a Boolean expression and that produces a sequence of machine statements and the register that contains the result.

The M Machine

In the following, we describe a machine with registers. The arithmetical and logical operations update the condition codes. The arithmetical and logical operations, noted OPER, are ADD, SUB, AND, OR.

General registers are noted Ri. The PC register designates the program counter, the register FP designates the basis of local environment, and the register SP the head of the stack. Statements and addresses are coded on 4 bytes, the stack pointer designates the last occupied slot. We push on the stack by decrementing by 4 the stack pointer et we pop on the stack by incrementing by 4 the register SP. Integers are encoded on 4 bytes.

To each variable of a program is associated an address of the form $FP - \text{shift}$, where **shift** is computed and inserted in the symbol table.

In this exercise, we are interested in elementary operations, which syntax is given below. We will see branching operations and procedure calls later in lecture course on code generation.

OPER Ri, Rj, Rk Ri is the destination register

OPER Ri, Rk, val Ri is the destination register

LD Ri, [addr]

ST Ri, [addr]

Addresses are described as follows:

addr ::= Ri + Rj | Ri + val | Ri | val

Exercise 12 — Generating 3-address code

Give the sequence of code that corresponds to the following expressions or sequences of statements. We suppose having the following symbol table.

identifier	shifting
x	4
y	8
z	12
u	16
v	20
w	24

1. $x-y+z$
2. $x := 3$
3. $x := 3 ; y := x+2 ;$
4. $x := 3 ; y := 2 ; z := x + y$
5. $x := 3 ; y := 2 ; z := x + y ; u := x + y + z ; v := z + y ; w := y + v ;$

Local optimizations

Exercise 13 — Optimizing code

Minimize the number of registers used in the previous code sequences. For this purpose, we can be interested in:

- the minimal number of register to evaluate an expression;
- the analysis of active variables to minimize the LD and ST operations that can be avoided;
- a bound of the total number of registers and the backup policy.

The backup policy is in place when the number of registers is not sufficient in order to perform the computation. In that case, the value of a register can be “backed up” in the local environment (after the variables of the local environment). That is for the symbol table mentioned below, the first free slot is located at [FP-28].

1.3.6 Global Optimizations Independent from the Machine

Exercise 14 — Global optimizations

Let us consider the following program written in an intermediate representation:

(1) $a := 1$	(4) $d := a+b$	(7) $f := a+b$
(2) $c := a+b$	(5) $c := c+2$	(8) $c := c-1$
(3) if $c>0$ goto 7	(6) goto 9	(9) $g := d+c$

1. Build the control-flow graph.
2. Suppress redundant computations.
3. Suppress the assignments that can be avoided, while supposing that:
 - no variable is used outside the code snippet,
 - g is used outside the code snippet.
4. Perform constant propagation.

SEMANTIC ANALYSIS

Exercise 15 — Program correctly typed or not?

Consider the environment $\Gamma = [x_1 \mapsto \text{Int}, x_2 \mapsto \text{Int}, x_3 \mapsto \text{Bool}]$. Indicate whether the following programs are correctly typed or not.

1. Program 1:

```
x1 := 3;
while ¬x3 do
  x1 := x2 + 1;
  x3 := x3 and true
od
```

2. Program 2:

```
x1 := 3 * x1 + 1;
if x2 and ¬x3 then
  x1 := x2 + 1
else
  x1 := x2;
fi
```

Exercise 16 — Sequential or Collateral evaluation in declarations

Consider the sequence of variable declarations $D_V = \text{var } x_1 := 3; \text{var } x_2 := 2 * x_1 + 1; \text{var } x_3 := \text{true}$ and the initial environment $\Gamma_V = []$.

1. Compute the resulting environment by updating Γ_V with D_V using *sequential* evaluation.
2. Compute the resulting environment by updating Γ_V with D_V using *collateral* evaluation.

Exercise 17 — Program correctly typed or not?

Indicate whether the following programs are correctly typed or not (in the empty environment) using the two modes of variable declarations (sequential and collateral).

1. begin

```
  var x := 5 ;
  var y := x + 5 ;
  x := y+4
end
```

2. begin

```
  var x := 5 ;
  begin
    var x := x < 3 ;
    x := not x
  end ;
  x := 4
```

3. begin

```
  var x := 5 ;
  var y := y + x ;
  x := y
end
```

Exercise 18 — Program correctly typed or not?

We consider the following program.

begin

```
  var x := 2 ;
  var y := true ;
  var z = x + 2 ;
```

```

begin
  var u := x + 6 ;
  var v := not y ;
  x := u (* observation point 1 *)
end
y := false (* observation point 2 *)
end

```

1. Give the value of the environment at the two observation points, using the two modes of variable declarations (sequential and collateral).

Exercise 19 — Adding a typing rule for a new construct

We are interested in the construct/expression “ $a_1 ? a_2 : a_3$ ” which is available in C or Java. The informal semantics of this construct is as follows: if a_1 is true then the value of this expression is a_2 else the value is a_3 .

1. Complete the abstract syntax of expressions to support this construct.
2. Give typing rules for this construct.

Exercise 20 — Introducing floats and type conversion

We want to add the type `Float` to the `While` language.

1. Complete the abstract syntax and the type system to support the type `Float` where no conversion is allowed between `Int` and `Float`.
2. Complete the type system to allow *implicit* conversion from `Int` to `Float`.
3. Complete the abstract syntax and the type system to allow the *explicit* conversion from `Int` to `Float` through an appropriate type conversion operator.

Exercise 21 — Typing rules for the for and repeat constructs

We add two new statements to the `While` language (introduced in the lecture session):

- a “repeat” statement: `repeat S until b`
- a “for” statement: `for x from e_1 to e_2 do S`

1. Give the typing rule(s) associated to the “repeat” statement.
2. Give the typing rule(s) associated to the “for” statement. You will distinguish between two cases:
 - the “for” statement *declares* the variable x (like in Ada or Java), the scope of this new variable is S ;
 - the “for” statement *does not declare* the variable x (like in C), and therefore x has to exist in the current environment.

Exercise 22 — Other forms of variable declarations

Modify the type system seen in the course when variable declarations can take the following additional forms.

1. `var x : t`
2. `var x := e : t`

Exercise 23 — Type-checking a program

We consider the type system seen in the course and the following program.

```

begin
  var x := 3
  proc p is x := x + 1

```

```

proc q is call p
begin
  proc p is x := x + 5
  call q
  call p
end
call p
end

```

1. Determine whether this program is correctly type in the case of *static* binding for variables and procedures.
2. Determine whether this program is correctly type in the case of *dynamic* binding for variables and procedures.

Exercise 24 — Mutually-recursive procedures

We consider the programs below.

<pre> begin proc p is call p ; call p ; end </pre>	<pre> begin proc p1 is call p2 ; proc p2 is call p1 ; call p1 ; end </pre>
--	--

1. Show that, with the type system defined so far for the Proc language, the programs are *incorrect*.
2. Modify this type system to take into account such (*mutually*) *recursive* procedures. Verify that these programs are now correct with the new type system.
Clue. Each sequence of procedure declaration should be analyzed twice: a first time to build its associated local environment, and a second time to check its correctness with respect to this local environment.

Exercise 25 — Correctly-initialized variables

A variable is said to be *correctly initialized* if it is never *used* before being assigned with an expression containing only correctly initialized variables. Let us consider for instance the following program:

`x := 0 ; y := 2 + x ; z := y + t ; u := 1 ; u := w ; v := v+1 ;`

In this program:

- `x` and `y` are correctly initialized ;
- `z` is not correctly initialized (because `t` is not correctly initialized); `u` is not correctly initialized (because `w` is not correctly initialized); and `v` is not correctly initialized (because `v` is not correctly initialized).

Some compilers, such as `javac`, reject programs that contain non-correctly initialized variables. We want to define in this exercise a type system which formalizes this check. To do so, we consider the following judgments:

- an environment is simply a set V of correctly initialized variables;
- $V \vdash e$ means that “in the environment V , expression e is correct (it does not contain non correctly initialized variables)”;
- $V \vdash S \mid V'$ means that “in the environment V , statement S is correct and produces the new environment V' ”.

1. Give the corresponding type system for the While language (without blocks nor procedures).
2. Apply the type system to the following code snippet, using $\Gamma = \emptyset$:
 - a) $x := 1; \text{if } x = 0 \text{ then } y := x + 1 \text{ else } y := x - 1 \text{ fi}$
 - b) $x := 1; \text{if } x = 0 \text{ then } x := x + 1 \text{ else } y := x - 1 \text{ fi},$
 - c) $x := 1; \text{while } x \leq 10 \text{ do } y := x + y ; x := x + 1.$
3. Show (on an example) that, similarly to `javac`, your type system may reject programs that would be correct at run-time.

Exercise 26 — Procedures with one parameters

We consider the following modified abstract syntax where procedures can have one parameter:

$$\begin{aligned} Dp &::= \text{proc } p(y : t) \text{ is } S ; Dp \mid \dots \\ S &::= \dots \mid \text{call } x(e) \end{aligned}$$

1. Modify the type system to handle procedures with one parameter.
2. Use the extended type system to prove that the following program is correctly typed.

```
begin
  var x := 3
  proc p (u : int) is x := u + 1
  begin
    var x := true
    proc p (u : bool) is not u
    call p (x)
  end
  call p (x)
end
```

Exercise 27 — Considering functions

We extend language `Proc` to handle procedures that return value, aka functions. This entails that functions can be called within expressions.

1. Extend the abstract grammar of `Proc`.
2. Extend the type system of `Proc` accordingly.

Exercise 28 — Adding parameters to procedures in the type system

We aim at extending the While language to add *parameters* to procedures. We shall proceed in several steps.

1. Consider only `in` parameters.
2. Consider only `out` parameters.
3. Consider both `in` and `out` parameters.
4. Take into account the extra rule (inspired from the Ada language), stating that:
 - `out` parameters cannot appear in right-hand side of an assignment;
 - `in` parameters cannot appear in left-hand side of an assignment.
5. Show that, in this last case, your type system may *reject* correct programs because of this rule. How could you solve this problem?

Exercise 29 — Sub-typing and dynamic types

We extend language `While` by introducing the notion of *sub-typing* through the following syntax for blocks, where t is a **type identifier** and `extends` means “is a sub-type of” (like in Java):

$$\begin{aligned} S &::= \dots \mid \text{begin } D_T ; D_V ; S \text{ end} \\ D_T &::= \text{type } t \text{ extends } B_T ; D_T \mid \varepsilon \\ B_T &::= \text{Top} \mid \text{Int} \mid \text{Bool} \mid t \end{aligned}$$

We aim to define a type system for this language which reflects the usual notion of sub-typing, namely:

- The sub-typing relation is a partial order \sqsubseteq whose greatest element is **Top**. It can be formalized by a *type hierarchy* (X, \sqsubseteq) , where X is a set of declared types (including the predefined types **Top**, **Int** and **Bool**).
- A value of type t_2 can be assigned to a variable of type t_1 whenever $t_2 \sqsubseteq t_1$. The converse is false.

1. Propose a type system which takes these rules into account. Judgments could be of the form:
 - $(X, \sqsubseteq), \Gamma \vdash S$, meaning that “in the environment Γ and with the type hierarchy (X, \sqsubseteq) , the statement S is well-typed” ;
 - $(X, \sqsubseteq), \Gamma \vdash e : t$, meaning that “in the environment Γ and with the type hierarchy (X, \sqsubseteq) , the expression e is well-typed and of type t ” ;
 - $(X, \sqsubseteq) \vdash D_T \mid (X', \sqsubseteq')$, meaning that “type declaration D_T is correct within the type hierarchy (X, \sqsubseteq) and produces the type hierarchy (X', \sqsubseteq') ” ;
 - $(X, \sqsubseteq), \Gamma \vdash D_V \mid \Gamma_l$, meaning that “in the environment Γ and with the type hierarchy (X, \sqsubseteq) , the variable declaration D_V is correct and produces the environment Γ_l ”.
2. Show that the following program is rejected by your type system:

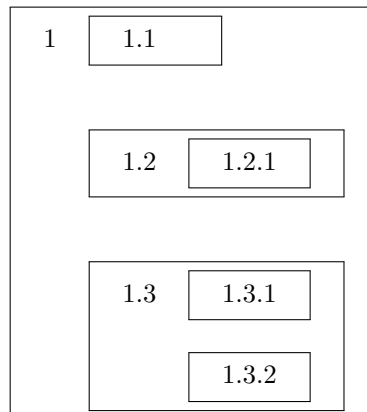
```
begin
  type t extends Int ;
  var x1 : Int ;
  var x2 : t ;
  var x3 : Int ;
  x1 := x2 ;
  x3 := x1 ;
  x2 := x3
end
```

3. Although rejected by your type system, the previous program is perfectly safe (it does not violate the informal sub-typing rules). However, its correctness can only be ensured at run-time, by introducing a notion of *dynamic type* to each identifier. This dynamic type corresponds to the actual type value held by this identifier at each program step (contrarily to the *static type*, the one *declared* for this variable).

Rewrite the (natural) operational semantics of the **While** language to take into account this notion of *dynamic type* and perform the type-checking at run-time. You can extend the configurations with a (dynamic) environment ρ which associates its dynamic type to each identifier.

Exercise 30 — Nested blocks and global environment

To define the type system of **Block** (possibly with nested blocks, but without procedures), we propose a notion of *global* environment in which each identifier is *uniquely* defined. More precisely, we assume a hierarchical numbering of blocks:



An environment now associates a type to a **pair** in $\mathbf{Vars} \times \mathbb{N}^*$, and a statement is type-checked **within** a given block.

1. Define the corresponding judgments and type system¹.

Exercise 31 — Static vs Dynamic Type system

We consider the following While program (with a command 'write'):

```
begin  var x := 2;
      var y := 1;
      proc p is x := x + y end;
      begin var y := true;
            call p;
            write x;
          end;
      end;
```

1. According to the static semantics for variables and procedures, what does this program write?
2. Is this program well-typed in the static semantics type system?
3. According to the dynamic semantics for variables and procedures, what happens with this program?
4. Is this program well-typed in the dynamic semantics type system? We deduce that even if a program is well-typed in the static type system, it does not matter when we want to execute it with a dynamic semantics!
5. Propose a modification of this program so that it is well-typed in the dynamic semantics type system, and it displays a Boolean.
6. (optional) If you master the static-dynamic semantics, you can try to exhibit a program which is well-typed in the dynamic type system but not in the static-dynamic type system. You can use a second procedure **q**.

¹ \mathbb{N}^* denotes the set of finite words over \mathbb{N} .

NATURAL OPERATIONAL SEMANTICS OF LANGUAGE WHILE

In this exercise series, we consider language While and its natural operational semantics where configurations belong to $\text{Stm} \times \text{State} \cup \text{State}$, i.e., are of the form (S, σ) or σ .

3.1 Properties of Arithmetical and Boolean Expressions

3.1.1 Free Variables

The set of free variables of an arithmetical or a Boolean expression is defined to be the set of variables occurring in it.

Exercise 32 — Locating free variables

Indicate the free variables in the following arithmetical expressions.

1. $x + 1$
2. $3 * x + y$

Exercise 33 — Free variables of arithmetical expressions

1. Define, in a compositional manner, a function $FV : \text{Aexp} \rightarrow 2^{\text{Vars}}$ that computes the *free variables* for an arithmetic expression.
2. Prove that only the free variables of an arithmetical expression may influence its value.

Exercise 34 — Free variables of Boolean expressions

1. Define, in a compositional manner, a function $FV : \text{Bexp} \rightarrow 2^{\text{Vars}}$ that computes the *free variables* of Boolean expressions.
2. Prove that only the free variables of a Boolean expression may influence its value.

3.1.2 Substitution

In the lecture, we define the notion of substitution for a state.

Exercise 35 — Computing some substitutions

Indicate the values of variables in the following states, where $\sigma = [x \mapsto 1, y \mapsto 2, z \mapsto 3]$:

1. $\sigma[x \mapsto y + z]$
2. $\sigma[x \mapsto 2 * x + y, y \mapsto x]$

Substituting a_0 to y in an arithmetical expressions a consists in replacing every occurrence of y by a_0 , and is noted $a[y \mapsto a_0]$.

Exercise 36 — Defining Substitution for arithmetical expressions

We consider the arithmetical expressions defined in the course lecture.

1. Define formally substitution for arithmetical expressions.
2. Prove that the defined substitution is correct.

Exercise 37 — Defining substitution for Boolean expressions

Define substitution for Boolean expressions where variables are replaced by arithmetical expressions.

3.2 Natural Operational Semantics of (Pure) Language While

Exercise 38 — Computing some (simple) derivation trees

Consider the state σ_0 which maps all variables but x and y to 0, maps x to 5, and y to 7.

1. Give a derivation tree for the following statements on σ_0 :
 - a) $(x := y; x := z); y := z,$
 - b) $(z := x; x := y); y := z.$
2. Give a derivation tree for the following statements:
 - a) **if** $x + y \geq 3$ **then** $y := x$ **else** $x := y$ **fi**,
 - b) **if** $y \geq x$ **then** $y := x$ **else** $x := y$ **fi**.

Exercise 39 — Computing some derivation trees

Consider the empty state $[]$ ¹. Give a derivation tree for the following statement:

$y := 1; x := 2; \text{while } x \geq 1 \text{ do } (y := y * x; x := x - 1) \text{ od.}$

Exercise 40 — Computing some (simple) derivation trees

Consider state σ_0 where x has value 17 and y has value 5.

1. Construct a derivation tree for the following statement when executed on σ_0 :

$z := 0; \text{while } y \leq x \text{ do } (z := z + 1; x := x - y) \text{ od}$

Exercise 41 — Execution terminates or loops?

For each of the following statements (where x designates a variable of type \mathbb{Z}), argue whether:

- its execution loops in every state, or
- its execution stops in every state, or
- there are states from which the execution terminates, and some from which it does not.

1. **while** $1 \leq x$ **do** $(y := y * x; x := x - 1)$ **od**
2. **while** **true** **do** **skip** **od**
3. **while** $\neg(x = 1)$ **do** $(y := y * x; x := x - 1)$ **od**

Exercise 42 — Execution terminates or loops? - proofs

Prove your answers for the previous exercise.

Exercise 43 — Operational semantics for arithmetical and Boolean expressions

¹Recall that a state is a partial function for variable names Vars to \mathbb{Z} . The empty state is thus the partial function undefined for every variable name.

The purpose of this exercise is to define an alternative definition to the semantics of arithmetical and Boolean expressions. The semantics given in the lecture course is a declarative semantics defined inductive on the syntax of arithmetical expressions. In this exercise, we want to define an operational semantics.

1. Define an operational semantics for the set of arithmetical expressions **AExp**. The semantics should have two kinds of configurations:
 - (a, σ) denoting that a has to be evaluated in state σ , and
 - z denoting the final value (an element of \mathbb{Z}).

That is the set of configurations of the underlying transition system is

$$\text{Configurations} = (\text{Aexp} \times \text{State}) \cup \mathbb{Z}.$$

2. Prove that the semantics of a defined in this way is equivalent to the one defined in the lecture course (i.e., equivalent to the semantics defined by the inductive function \mathcal{A}).
3. Define an operational semantics for the set of Boolean expressions **Bexp**.
4. Prove that the meaning of a defined in this way is equivalent to the one defined in the lecture course (i.e., equivalent to the meaning defined by the inductive function \mathcal{A}).

Exercise 44 — A “subset” of language While

We consider the language defined by the following BNF:

$$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

1. What can we say about termination of programs written in this language?
2. Prove your statement.

Exercise 45 — Determinism of the natural operational semantics of language While

Prove that the natural operational semantics of language While is deterministic.

Exercise 46 — Associativity of sequential composition

1. Prove that, for all statements S_1, S_2, S_3 , the following statements are semantically equivalent:
 - $S_1; (S_2; S_3)$, and
 - $(S_1; S_2); S_3$.
2. Prove that, in general, $S_1; S_2$ is not semantically equivalent to $S_2; S_1$.

3.3 Natural Operational Semantics of Extensions of Language While

Exercise 47 — Extending language While with construct `repeat ... until ...`

We want to add the following statement to language While:

`repeat S until b`

The informal semantics of this construct is that statement S should be executed until the Boolean condition b becomes true.

1. Provide the semantics rules in order to define `repeat S until b` without using the `while b do ... od` construction.
2. Prove that the following statements are semantically equivalent:

- **repeat** S **until** b , and
 - S ; **if** b **then** **skip** **else** (**repeat** S **until** b) **fi**.
3. We want to prove that statement **repeat** S **until** b does not add expressiveness to language **While**. To do so, give a function which transforms every program with the statement **repeat** S **until** b into a program in language **While**. Is the given transformation computable? Compare the size of a program and its image resulting of the transformation.

Exercise 48 — **Extending language While with construct for ... from ... to ... do ...**

We want to add another iterative construct to language **While**. Consider the statement

for x **from** a_1 **to** a_2 **do** S .

where the first expression a_1 is the initial value that x is assigned to, the second expression is the limit that x should be assigned to. Moreover, the “step” of the loop is fixed.

The purpose of this exercise is to extend the semantics of language **While** by providing appropriate rules for this construct (and without using the **while** ... **do** ... **od** construct). You may need to assume that you have an “inverse” to \mathcal{N} , so that there is a numeral for each number that may arise during computation. There are several alternatives depending on what is allowed for S . Examples of criteria are:

- Evaluation of a_1 and a_2 are done once (at the beginning) or each time the loop body is executed.
 - Evaluation of a_1 and a_2 are done each time the loop body is executed.
1. Provide semantics rules for an alternative where the first criterion holds.
 2. Provide semantics rules for for an alternative where the second criterion holds.
 3. Consider the state σ which maps x to 5. Evaluate the following statement in σ :

$y := 1$; **for** z **from** 1 **to** x **do** $y := y + x$; $x := x - 1$.

NATURAL OPERATIONAL SEMANTICS OF LANGUAGES BLOCK AND PROC

In this exercise series, we consider language Block and Proc and its natural operational semantics.

4.1 Natural Operational Semantics of Language Block

Exercise 49 — Computing the semantics of a program in Block

Compute the semantics of the following program on $\sigma_0 = [x \mapsto 0]$:

```
begin  vary := 1;  
      (x := 1;  
       begin var x := 2 ; y := x + 1 end  
       x := y + x)  
end
```

4.2 Natural Operational Semantics of Language Proc

Exercise 50 — Completing the semantics of Proc with static links

Complete the semantics of Proc with blocks and procedures with static links for variables and procedures.

Exercise 51 — Computing the semantics of a program

Let us consider the following code snippet seen in the lecture course:

```
begin  var x := 0;  
      proc p is x := x * 2 end;  
      proc q is call p end;  
      begin  
        var x := 5;  
        proc p is x := x + 1 end;  
        call q; y := x;  
      end;  
end
```

Assume that variables x and y have been previously assigned, before entering this code, to x_0 and y_0 respectively. Moreover, assume that there is no procedure declared before entering this code snippet (i.e., the procedure environment is empty). Compute the semantics of this program according to the following three variants:

1. Dynamic links for procedures and variables.
2. Static links for procedures and dynamic link for variables.
3. Static links for procedures and variables.

Exercise 52 — Computing the semantics of a program

Let us consider the following program seen in the lecture course:

```
begin  var x := 2;
      proc p is x := 0 end;
      proc q is begin var x := 1; proc p is call p end; call p; end; end
      call q;
end
```

Compute the semantics of this program according to the following variants.

1. With static links for procedures, dynamic links for variables, and the recursive call rule,
2. With static links for procedures, dynamic links for variables and the non-recursive call rule.

Exercise 53 — Procedures as variables

We add the following statement to language While with blocks and procedures:

$\text{Stm} ::= p := S.$

Give a semantics to this language. Your semantics should be conservative wrt. the semantics of language While.

1. Consider the semantics with dynamic links for variables and procedures.
2. Consider the semantics with static links for variables and procedures.

Exercise 54 — Procedures with parameters

We modify the syntax of procedures to allow parameters:

$$\begin{aligned} S &\in \text{Stm} \\ S &::= x := a \mid \text{skip} \mid S_1; S_2 \\ &\quad \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } b \text{ do } S \text{ od} \\ &\quad \mid \text{begin } D_V \ D_P ; S \text{ end} \\ &\quad \mid \text{call } p(a_1, a_2) \\ D_V &::= \text{var } x := a; D_V \mid \epsilon \\ D_P &::= \text{proc } p(x_1, x_2) \text{ is } S \text{ end; } D_P \mid \epsilon \end{aligned}$$

We are interested in the semantics with *static link for procedures and dynamic link for variables*.

1. Modify the semantics of procedure declaration and call in order to obtain a semantics with call-by-value.
2. Same question with call-by-reference.
3. Same question with call by result.
4. Same question with call by value-result.

Exercise 55 — Completing a program

We are interested in the semantics with static links for variables and procedures. We extend language Proc with a write command: *write x* prints out the value of $\sigma \circ \rho(x)$, but variable environment and storage function are left unchanged.

We consider the following program:

```
begin
  var x := 2;
  begin
    var y := 7;
    begin
      var x := 5;
      var y := 0;
      call p;
    end
  end
end
```

1. Place the instructions `proc p is x := x * y end;` and `write x` so that the value 14 appears on screen.
2. Justify your answer by computing the output using the rules.

STRUCTURAL OPERATIONAL SEMANTICS

In this series we start from the structural operational semantics of language **While** as seen in the course.

Exercise 56 — Computing some derivation sequences

Give a derivation sequence and the associated derivation trees for the following statements on the following states:

1. $(z := x; x := y); y := z$ on $[x \mapsto 2, y \mapsto 5, z \mapsto 7]$,
2. $(y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1) \text{ od})$ on $[x \mapsto 5, y \mapsto 7]$,
3. $(z := 0; \text{while } y \leq x \text{ do } (z := z + 1; x := x - y) \text{ od})$ on $[x \mapsto 17, y \mapsto 5]$.

Exercise 57 — Some properties of the structural semantics

Prove the following claims:

1. If $(S_1, \sigma) \Rightarrow^k \sigma'$ then $(S_1; S_2, \sigma) \Rightarrow^k (S_2; \sigma')$.
That is the execution of S_1 is not influenced by the statement following it.
2. If $(S_1; S_2, \sigma) \Rightarrow^k \sigma''$ then there exists σ' and k_1 such that: $(S_1, \sigma) \Rightarrow^{k_1} \sigma'$ and $(S_2, \sigma') \Rightarrow^{k-k_1} \sigma''$.
That is, if it takes k steps to execute $S_1; S_2$ in σ , then there exists an integer k_1 s.t. it takes k_1 steps to execute S_1 in σ to yield a final configuration σ' . Then executing S_2 in σ' takes $k - k_1$ steps.

Exercise 58 — A Structural Semantics for Blocks

Provide a structural operational semantics for the language **Block**.

Exercise 59 — Determinism of the structural operational semantics

1. Show that the structural operational semantics of **While** is deterministic.
2. Deduce that there is exactly one derivation sequence starting in a configuration (S, σ) .
3. Argue that a statement S of **While** cannot both terminate and loop on a state σ and hence it can not both be always terminating and always looping.

Exercise 60 — Equivalence between the natural and the structural semantics

Prove the following claims.

1. If $(S, \sigma) \rightarrow \sigma'$ then $(S, \sigma) \Rightarrow^* \sigma'$.
It means that if the execution of a statement terminates in a given state in the natural semantics, then it will terminate in the same state in the structural semantics.
2. If $(S, \sigma) \Rightarrow^k \sigma'$ then $(S, \sigma) \rightarrow \sigma'$.
It means that if the execution of a statement terminates in a given state in the operational semantics, then it will terminate in the same state in the operational semantics.
3. Deduce that the natural semantics is equivalent to the structural semantics.

Exercise 61 — **Equivalence between some statements**

1. Show that the two following statements are semantically equivalent (You may use the fact that the semantics is deterministic.):
 - $S; \text{skip}$, and
 - S .
2. Same question for
 - $\text{repeat } S \text{ until } b$, and
 - $S; \text{while } \neg b \text{ do } S \text{ od}$.

Exercise 62 — **Associativity of sequential composition in the structural semantics**

1. Prove that, for all statements S_1, S_2, S_3 , the following statements are semantically equivalent:
 - $S_1; (S_2; S_3)$
 - $(S_1; S_2); S_3$

You may use any result of the previous exercises.

2. Prove that, in general, $S_1; S_2$ is not semantically equivalent to $S_2; S_1$.

AXIOMATIC SEMANTICS - HOARE LOGIC

Exercise 63

Prove that the following Hoare triples are valid.

1. $\{x = 0\} x := x + 1; x := x + 1 \{x = 2\}$
2. $\{x > 0\} y := 1 \{x = x * y\}$
3. $\{x > a\} x := x + 1 \{x > a + 1\}$
4. $\{x = 2 * a\} a := a + 2 \{x = 4 * a - 4\}$
5. $\{x > a\} x := x + 1; x := x + x \{x > 2a + 2\}$
6. $\{x \geq 0\} \text{ if } x \geq 0 \text{ then } y := 8 \text{ else } y := 9 \text{ fi } \{x = 8\}$

Exercise 64

Prove that the following Hoare triples are valid.

1. $\{y \geq 0\} S \{z = y!\}$, where S is:

```

x := y ;
z := 1 ;
while 1 < x do
  z := z * x ;
  x := x-1
od

```
2. $\{x \geq 0\} S \{x = 2a + b\}$, where S is:

```

a := x div 2 ;
if even(x) then
  b := 0
else
  b := 1
fi

```
3. $\{a \geq 0 \wedge b > 0\} S \{a = b * q + r \wedge r \geq 0 \wedge r < b\}$, where S is:

```

q := 0 ;
r := a ;
while b <= r do
  q := q + 1 ;
  r := r-b ;
od

```
4. $\{b \geq 0\} S \{p = b^2\}$, where S is:

```

c := b ;
p := 0 ;
while c > 0 do
  p := p + b ;
  c := c-1 ;
od

```
5. $\{n \geq 1\} S \{p = m * n\}$, where S is:

```

p := 0 ;
c := 1 ;
while c <= n do
  p := p + m ;
  c := c+1
od

```
6. $\{n \geq 0\} S \{x = \text{Fib}(n)\}$, where S is:

```

x := 0 ;
y := 1 ;
c := n ;
while c > 0 do
  h := y ;
  y := x + y ;
  c := c - 1
od

```

Exercise 65

Prove that the following Hoare triples are valid.

1. $\{\text{True}\} S \{y1 * y1 \leq x \wedge (y1 + 1) * (y1 + 1) > x\}$, 2. $\{\text{True}\} S \{p = |x - y|\}$, where S is:
where the loop invariant is $y1 * y2 \leq x$ and

S is:

<pre> y1 := 0 ; y2 := 1 ; y3 := 1 ; while y3 <= x do y1 := y1+1 ; y2 := y2+2 ; y3 := y1+y2 ; y := x + y ; c := c - 1 od </pre>	<pre> a := x ; b := y ; if a > b then p := a-b else p := b-a fi </pre>
---	---

Exercise 66

Let S be the following program:

```

u:=0;
while x>1 do
  if even(x) then
    x := x / 2;
    y := y*2
  else
    x:=x-1;
    u:=u+y
  fi
od;
y:=y+u

```

1. Use the Hoare logic to show that $\vdash \{x = x_0 \wedge y = y_0 \wedge x_0 > 0\} S \{y = x_0 * y_0\}$

Exercise 67

We consider the pre-condition and the post-condition in the incomplete Hoare triple below:

$$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\} \dots \{z = n * m\}$$

1. Give a program S which satisfies the specification and which uses only addition and subtraction:
2. Demonstrate using Hoare logic that your program satisfies the specification.

Exercise 68

We consider the pre-condition and the post-condition in the incomplete Hoare triple below:

$$\{x = n \wedge y = m \wedge m \geq 0 \wedge n \geq 0\} S \{z = n^m\}$$

1. Give a program that satisfies the specification and which uses only addition and subtraction.
2. Demonstrate using Hoare logic that your program satisfies the specification.

Exercise 69

We consider the statement

for i from 1 to n do S

where n is the denotation of a natural number in \mathbb{N} and S is a statement in which i is not modified.

1. Give a semantics rule and an Hoare rule for this statement.

Exercise 70

We add the statement:

repeat S **until** b

to the While language.

1. Give an inference rule for **repeat** S **until** b .
2. Demonstrate the correctness of your rule.
3. Is the rule complete? (prove your answer)

Exercise 71

1. Demonstrate that the predicate transformer wlp is:
 - a) Monotone wrt. the implication.
 - b) $\text{wlp}(S, P \wedge Q) \iff (\text{wlp}(S, P) \wedge \text{wlp}(S, Q))$.
2. Does the following hold ?
 - a) $\text{wlp}(S, P \vee Q) \iff (\text{wlp}(S, P) \vee \text{wlp}(S, Q))$.
 - b) $\text{wlp}(S, \neg P) \iff \neg \text{wlp}(S, P)$

Exercise 72

We consider the following predicate transformer:

$$\text{post}(S, P) = \{\sigma' \mid \exists \sigma \cdot \sigma \models P \wedge (S, \sigma) \rightarrow \sigma'\}$$

1. Prove that: $\{P\} S \{Q\}$ iff $\text{post}(S, P) \Rightarrow Q$.
2. Prove that: $\vdash \{P\} S \{\text{post}(S, P)\}$.
3. Deduce that the Hoare logic is complete.

PROVABLY-CORRECT IMPLEMENTATION

7.1 The Abstract Machine AM

The purpose of these exercises is to get familiar with the instructions and the semantics of AM.

Exercise 73 — Computing some executions of AM

1. Compute the execution of

`push-1 · fetch(x) · add · store(x)`

on an initial memory $m = [x \mapsto 3]$.

2. Compute the execution of `loop(True, noop)` in any memory m .
3. Compute the execution of `fetch(x) · fetch(y) · le · branch(push-1 · add, noop)` in an initial memory $m = [x \mapsto 2, y \mapsto 1]$.

Exercise 74 — Abstracting machine code

`push-0 · store(z) · fetch(x) · store(r)
loop(fetch(r) · fetch(y) · le,
fetch(y) · fetch(r) · sub · store(r) ·
push-1 · fetch(z) · add · store(z)
)`

What is the function computed by this code?

7.2 Properties of AM

The purpose of these exercises is to exhibit some properties of the abstract machine. These properties are similar to the properties we exhibited for the Structural Operational Semantics and shall be used later to prove the correctness of code generation.

Exercise 75

Prove that code and stack contents can be extended.

Formally: $\forall c_1, c_2, c \in \mathbf{Code}, \forall s_1, s_2, s \in \mathbf{Stack}, \forall m_1, m_2 \in \mathbf{State} :$

$$(c_1, s_1, m_1) \triangleright^k (c_2, s_2, m_2) \text{ implies } (c_1 \cdot c, s_1 \cdot s, m_1) \triangleright^k (c_2 \cdot c, s_2 \cdot s, m_2)$$

Exercise 76 — Code can be composed and decomposed

Prove that the code can be composed and decomposed.

Formally: $\forall c_1, c_2 \in \mathbf{Code}, \forall s_1, s_2 \in \mathbf{Stack}, \forall m_1, m_2 \in \mathbf{State} :$

$$(c_1 \cdot c_2, s, m) \triangleright^k (\epsilon, s_2, m_2) \text{ implies } \exists k' \in \mathbb{N}, (\epsilon, s', m') \in \mathbf{Config} : (c_1, s, m) \triangleright^{k'} (\epsilon, s', m') \wedge (c_2, s', m') \triangleright^{k-k'} (\epsilon, s_2, m_2)$$

Exercise 77 — Relation \triangleright is deterministic

Prove that the relation \triangleright is deterministic.

Formally: $\forall c, c_1, c_2 \in \mathbf{Code}, \forall s_1, s_2 \in \mathbf{Stack}, \forall m_1, m_2 \in \mathbf{State} :$

$$(c, s, m) \triangleright (c_1, s_1, m_1) \wedge (c, s, m) \triangleright (c_2, s_2, m_2) \text{ implies } (c_1, s_1, m_1) = (c_2, s_2, m_2)$$

7.3 Code generation

The purpose of these exercises is to define completely code generation for all possible statements of the While language. Similarly to what was done before when giving an operational semantics to the While language, we proceed in several steps: defining code generation for arithmetical expressions, for Boolean expressions and statements.

Exercise 78 — Code generation for arithmetical expressions

We are interested in the code generation function for arithmetical expressions.

1. Give the signature of this function.
2. Give the complete definition of the function.

Exercise 79 — Generating code for some arithmetical expressions

Generate the code for the following arithmetical expressions by applying the corresponding generation function.

1. $x + 1$
2. $2 * x$

Exercise 80 — \mathcal{CA} and associativity

In the lecture and exercises, we have seen that the semantic function for arithmetical expressions is associative, i.e., $\mathcal{A}[(a_1 + (a_2 + a_3))] = \mathcal{A}[(a_1 + a_2) + a_3]$, for any $a_1, a_2, a_3 \in \mathbf{Aexp}$.

1. Show that it is not the case that $\mathcal{CA}[(a_1 + (a_2 + a_3))] = \mathcal{CA}[(a_1 + a_2) + a_3]$.
2. Show that $\mathcal{CA}[(a_1 + (a_2 + a_3))]$ and $\mathcal{CA}[(a_1 + a_2) + a_3]$ behave in the same manner.

Exercise 81 — Code generation for Boolean expressions

We are interested in the code generation function for Boolean expressions.

1. Give the signature of this function.
2. Give the complete definition of the function.

Exercise 82 — Generating code for some Boolean expressions

Generate the code for the following Boolean expressions by applying the corresponding generation functions.

1. $x \leq 2$
2. $2 * x = 5 * y$

Exercise 83 — Code generation for statements

We are interested in the code generation function for statements.

1. Give the signature of this function.
2. Give the complete definition of the function.

Exercise 84 — Generating code for a program

Give the target code obtained when translating the *factorial program*.

```
y := 1; while ¬(x = 1) do y := y * x; x := x - 1 od
```

Exercise 85 — Generating code for a statement

Use the code generation functions to generate code for the following statement:

```
z := 0; while y ≤ x do z := z + 1; x := x - y od
```

Exercise 86 — Generating code for the repeat until construct

Extend the code generation function of statement to generate code for the **repeat until** construct. The definition of the code generation function should remain compositional and the instruction set of the machine should remain the same.

7.4 Correctness of code generation

The purpose of these exercises is to show that the code generation for the abstract machine is correct. It is performed in several steps.

Reminder from the course: in the following, for the sake of simpler notations, we assume that the memory component of AM and the state in the operational semantics are merged, i.e., we do not make any distinction between m and σ .

Exercise 87 — Correctness for Arithmetical expressions

Prove that the code generation function as defined in Exercise 78 is correct. Formally:

$$\forall a \in \text{Aexp} : (\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$$

Exercise 88 — Correctness for Boolean expressions

Prove that the code generation function as defined in Exercise 81 is correct. Formally:

$$\forall b \in \text{Bexp} : (\mathcal{CB}[b], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{B}[b]\sigma, \sigma)$$

Exercise 89 — Correctness for Statements

Prove that the code generation function as defined in Exercise 83 is correct. The proof amounts to proving the two following claims.

1. $\forall S \in \text{Stm}, \forall \sigma, \sigma' \in \text{State} : (S, \sigma) \rightarrow \sigma' \text{ implies } (\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, \epsilon, \sigma')$
2. $\forall S \in \text{Stm}, \forall \sigma, \sigma' \in \text{State}, \forall k \in \mathbb{N} : (\mathcal{CS}[S], \epsilon, \sigma) \triangleright^k (\epsilon, e, \sigma') \text{ implies } (S, \sigma) \rightarrow \sigma' \text{ and } e = \epsilon$

INTERMEDIATE-CODE OPTIMIZATION USING DATA-FLOW ANALYSIS

8.1 Defined and Used variables

We recall the syntax of the While language:

$$\begin{aligned} S &::= x := a \mid \text{skip} \mid S; S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } S \\ a &::= n \mid x \mid a + a \\ b &::= \text{true} \mid \text{false} \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \end{aligned}$$

8.1.1 Defining assigned and used variables in statements

We want to define the functions **Def** and **Use** which associate, to each syntactic construct, the set of assigned and used variables respectively. Assigned variables are the variables appearing in the left-hand side of an assignment. Used variables are the variables appearing on the right-hand side of assignments or in expressions.

Exercise 90 — A formal definition of **Def** and **Use**

We want to define formally functions **Def** and **Use**.

1. What is the signature of these functions?
2. Consider conditional statements built using the `if ... then S_1 else S_2 fi` construct. What are the possible situations (in terms of definition and use) for a variable depending on S_1 and S_2 ?
3. Based on the observation made in the previous question, one needs two definitions of the **Def** and **Use** functions. Give formal definitions for these functions.

Exercise 91 — Applying **Def** and **Use**

1. Apply functions **Def** and **Use** to: `if $a < b$ then $c := d - y$ else $y := e - x$ fi`.

8.1.2 Adding an operator \parallel for parallel execution

We extend the syntax of statements in the following way:

$$S ::= \dots \mid S \parallel S.$$

Exercise 92 — Parallelism in Natural Operational Semantics

We consider operator \parallel for parallelizing statements.

1. Recall the natural operational semantics of this operator.

In the following, we consider the While language extended with the \parallel operator and its natural operational semantics.

Exercise 93 — Applying the parallelism operator

Applying natural operational semantics rules extended with the rules for \parallel , compute the obtained state(s) by executing the following commands on the initial state $\sigma_0 = [x \mapsto 1, y \mapsto 1]$.

1. $x := x + 1 \parallel y := y + 1$
2. $x := y + 1 \parallel y := x + 2$

Exercise 94 — A sufficient condition so that parallelism does not influence computation

Give a sufficient condition such that the two previously defined rules yield the same result. That is, when we evaluate $S_1 \parallel S_2$ in a state σ , then the obtained state σ_2 is independent of the order of evaluation of S_1 and S_2 .

Hint: One can use the functions **Def** and **Use**, previously defined.

Exercise 95

Propose a unique semantic rule that evaluates $S_1 \parallel S_2$ on the state σ . This rule should propose an evaluation of S_1 and S_2 on state σ supposing that the previous condition holds.

Exercise 96

Apply the previously defined semantic rules to the statement of Exercise 93.

8.2 Preliminaries

8.2.1 Lattice, recursive equations, fix-points

Exercise 97

We consider set $E = \{a, b, c\}$.

1. Draw the lattice $(2^E, \subseteq)$, where 2^E denotes the powerset of E .
2. For each of the following subsets of E , indicate whether it contains a maximum or not, what is the set of its upper-bounds, what is its least upper bound (when it exists).
 1. $\{\{a\}, \{b\}\}$
 2. $\{\{a, b\}, \{b, c\}, \{a, c\}\}$
 3. $\{\{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$
 4. 2^E

Exercise 98

For a set X endowed with an order relation \leq , we say that function f is monotonic if $\forall x, y : x \leq y \implies f(x) \leq f(y)$. We consider the following functions:

- “complement” function: $F_1(x) = 2^{\{a, b, c\}} \setminus x$
- “union with $\{b, c\}$ ” function: $F_2(x) = x \cup \{b, c\}$
- “intersection with $\{b, c\}$ ” function: $F_3(x) = x \cap \{b, c\}$
- function $F_4 = F_3 \circ F_2$.

1. Indicate which functions are monotonic on $(2^{\{a, b, c\}}, \subseteq)$.
2. We are now interested in recursive equations of the form $X = F(X)$ defined on $(2^{\{a, b, c\}}, \subseteq)$. Give (without proof) the set of solutions of these equations for F_1 , F_2 , F_3 and F_4 .
3. Compute the least solution (i.e., the least fix-point) of equation $X = F_2(X)$ by successively computing $F_2(\dots F_2(F_2(\emptyset)))$.

4. Compute the greatest solution (i.e., the greatest fix-point) of equation $X = F_3(X)$ by successively computing $F_3(\dots F_3(F_3(\{a, b, c\})))$.
5. Are the computations possible for equation $X = F_4(X)$? And for equation $X = F_1(X)$?

8.2.2 Control-flow Graph (CFG)

Exercise 99

1. Draw the CFG of the following program.

```
x := 3 ;
while (x < 10) {
  y := x+1 ;
  if (y<5) {
    z := 2*x ;
    y := y-1 ;
  }
  else {
    z := y + 1
  }
  x := x+1 ;
}
y := y + z ;
```

Exercise 100

We consider the following 3-address code sequence:

```
1. a := 1
2. b := 2
3. e := a+b
4. d := c-a
5. if a+b>0 goto 11
6. d := b*d
7. goto 8
8. d := a+b
9. e := e+1
10. goto 3
11. b := a+b
12. e := c-a
13. if c > 3 goto 3
14. c := a+b
15. b := a-d
end
```

1. Split this sequence into basic blocks, and draw the resulting control flow graph.

8.3 Available Expressions

Exercise 101

We consider the program in Exercise 100 and its CFG.

1. Give the set of data-flow equations for computing *available expressions*.
2. Solve these equations.
3. Suppress *redundant computations*.

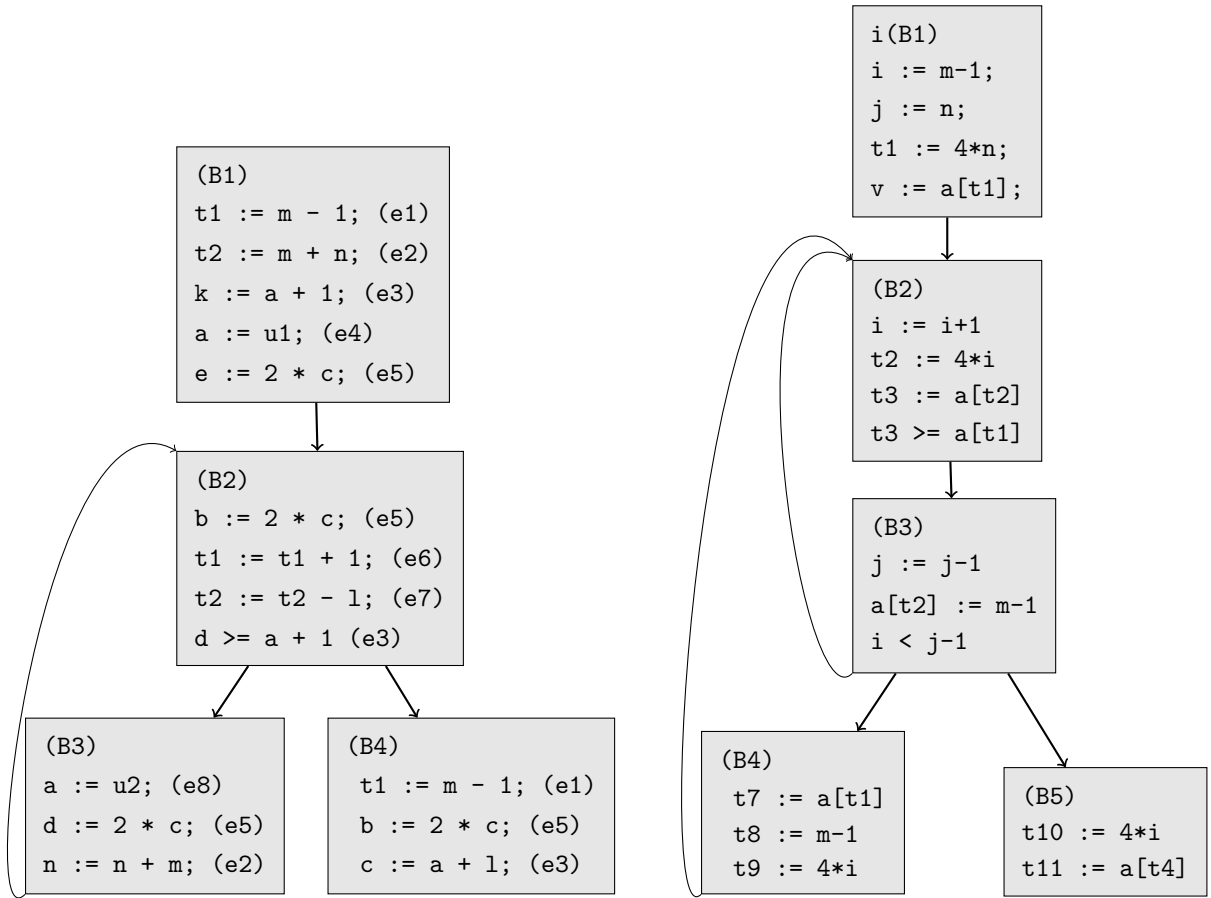


Figure 8.1: CFGs for Exercise 102

Exercise 102

For each of the CFGs in Figure 8.1:

1. Give the set of data-flow equations for computing *available expressions*.
2. Solve these equations.
3. Suppress *redundant computations*.

Exercise 103

Suppress redundant computations in the following program:

```

a := 5
c := 1
L1: if c>a goto L2
    c := c+c
    goto L1
L2: a:= c-a
    c:=0
  
```

8.4 Live Variables

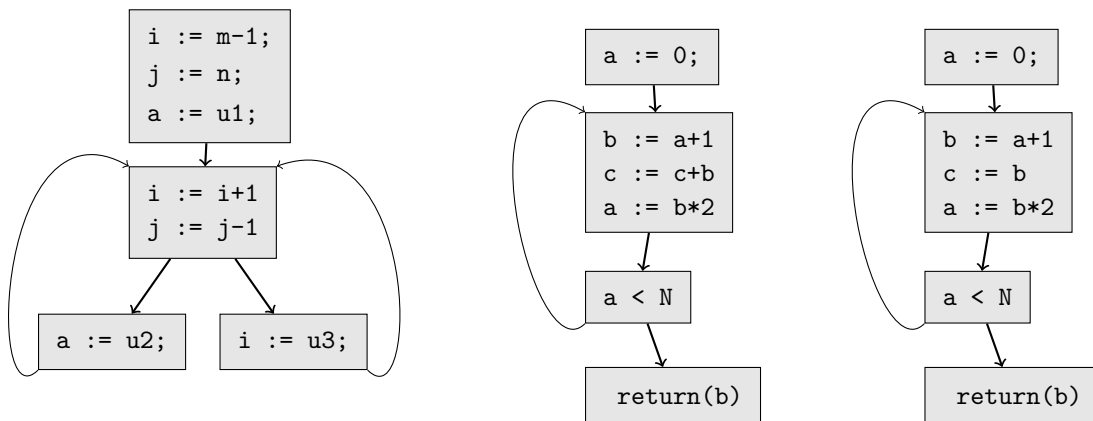


Figure 8.2: CFGs for Exercise 105

Exercise 104

We consider the following program:

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

1. Write the 3-address code sequence corresponding to this program.
2. Split this sequence into basic blocks, and draw the resulting control flow graph.
3. Give the set of data-flow equations for computing *live variables*.
4. Solve these equations.
5. Suppress *useless assignments*.

Exercise 105

For each of the CFGs depicted in Fig. 8.2 (obtained after some intermediate code generation):

1. Give the set of data-flow equations for computing *live variables*.
2. Solve these equations.
3. Suppress *useless assignments*.

8.5 Constant Propagation

Exercise 106

We consider the CFGs in Figure 8.3. Modify these CFGs by performing constant propagation.

Exercise 107

In the CFGs in Figure 8.4, propagate constants:

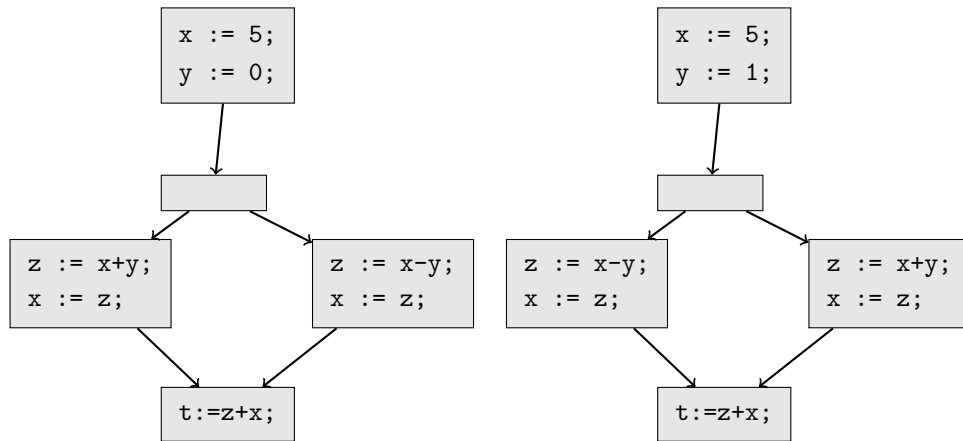


Figure 8.3: CFGs for Exercise 106

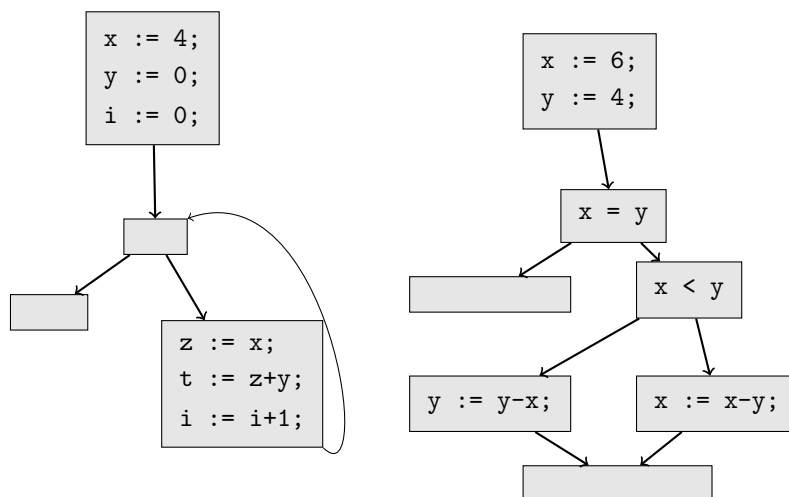


Figure 8.4: CFGs for Exercise 107

CODE GENERATION (OF ASSEMBLY CODE)

Exercise 108

We consider the following program.

```
main(c) {  
    int x, y, z;  
    x = 5;  
    y = 1;  
    z = x + y;  
}
```

1. Determine the symbol table (integers are coded on 4 bytes).
2. Generate assembly code.

Exercise 109

Let us consider the following program with its symbol table.

```
var x, y : int ;  
while (not x = y) do  
    if x > y then x := x - y else y := y - x fi  
od
```

1. Generate code.

Environment	
Procedure	Shifting
main	x → 4
	y → 8

Exercise 110

1. Extend the code generation function to handle the following construct:

```
for (var = lbound; var < ubound; step) {  
    S(variable)  
}
```

where:

- **lbound** and **ubound** are respectively the lower and upper bounds, given by arithmetical expressions,
- **S(var)** is a statement that depends on **variable**, and
- **step** is an expression that is added to **variable** at the end of each iteration.

2. For the program below, generate assembly code.

```
int t1[10];  
int i;  
for(i=0;i<10;i++) { t1[i] = 0 }
```

3. For the program below, generate assembly code.

```
int t1[10][10] ;
int i,j ;
for(i=0; i<10; i++) {
    for(j=0; j<10; j++)
        t1[i][j] = 0;
}
```

Exercise 111

Extend the code generation function seen for While.

1. To consider statements of the form **repeat** *S* **until** *b* ,
2. To consider Boolean expressions of the form *b1* **xor** *b2*,
3. To consider arithmetical expressions of the form *b* ? *e1* : *e2*.

Exercise 112

Let us consider the two following programs with their symbol tables. For each of them, generate code.

```
begin
    var x : int ; var y : int ;
    procedure p is
        y := x ;
    end
    x := 1 ;
    call p ;
end
```

Environment

Procedure	Shifting
main	x → 4 y → 8
main.p	

Exercise 113

We consider the following program.

```
begin
    var x : int ; var y : int ; var z : int :
    procedure p is
        begin var x : int ; var t : int :
            x := 4 ;
            y := 5 ;
            z := x + y ;
        end
        x := 5;
        call p ;
    end
```

Environment

Procedure	Shifting
main	x ↦ 4 y ↦ 8 z ↦ 12
main.p	x ↦ 4 t ↦ 8

1. Generate the corresponding assembly code.

Exercise 114

Let us consider the following program:

```
var x : int ; var y : int ; var z : int ; var a : int ; var b : int ; var c : int ;
c := 4 ;
x := a+b+c;
y := a * b + 2 * c ;
```

1. Generate 3-address code and then assembly code by supposing an unlimited number of registers.
2. Generate 3-address code and then assembly code by supposing that we have only 4 registers R_1, R_2, R_3, R_4 .

Exercise 115

We consider the C program in Figure 9.1.

1. Build the environment (symbol table)

```
void main() {
    int *x1 ;
    void p1() {
        int x2 ;
        void p2(int *x) { x=&x2; }
        void q2(int *x) { p2(x1); }
        q2(x1);
    }
    x1=malloc(4);
    *x1=1 ;
    p1();
}
```

Figure 9.1: Program for Exercise 115

2. Build the call-tree.
3. Generate code.

Exercise 116

For the C program in Figure 9.2.

1. Build the environment (symbol table)
2. Build the call-tree.
3. Generate code.

Exercise 117

We consider the program in Figure 9.3.

1. Draw the execution stack when executing **g3** by representing only the dynamic and static chaining.
2. We consider procedure **f1**. Give the code snippet that performs **y = h2(32)**.
3. We consider procedure **f1**. Give the code snippet that performs **return y**.
4. We consider procedure **f2**. Give the code snippet that performs **x = 11**
5. We consider procedure **f2**. Give the code snippet that performs **y=y+ f3(p)**
6. We consider procedure **f3**. Give the code snippet that performs **return p(y+x)**.
7. We consider procedure **g3**. Give the code snippet that performs **return p**.
8. We consider procedure **g2**. Give the code snippet that performs **z = x+y**.
9. We consider procedure **g2**. Give the code snippet that performs **return ((g3(g2))(x-1))**.

```

int main () {
    int x1;
    void P1 () { /*static nesting 1*/
        int x2 ; int y2 ; int z2 ;
        int Y (int x) {
            printf ("%d\n", x+1); return (x+1);
        } /* Y */
        void R2 ( int b3, int p(int) ) { /*static nesting 2*/
            int y3 ;
            void R3 ( int p(int) ) {
                y2 = 2 ;
                x2 = y2+x1+b3;
                y3 = 1+x2 ;
                b3 = p(x1);
            } /* R3 */
            R3(p);
        } /* R2 */
        void P2 ( int a, int p (int) ) { /*static nesting 2*/
            int x3;
            void P3 ( int p(int) ) {
                x3 = p(x1);
                R2(x3,Y);
            } /* P3 */;
            x2 = 1+a ;
            P3(p);
        } /* P2 */
        int X (int x) {
            printf ("%d\n", x); return x;
        } /* X */
        void Q () { /*static nesting 2*/
            int z3;
            y2 = 2 ;
            z2 = 3 ;
            z3 = x2 + y2 ;
            P2(z3,X);
        } /* Q */
        x1 = 11 ;
        Q();
    } /* P1 */
    P1();
} /* prog */

```

Figure 9.2: Program for Exercise [116](#)

```

#include <stdio.h>
typedef int (*intprocint) (int);
typedef int (*intprocintint) (int, int);
typedef int (*intprocvoid) (void);

main(){
    int x;
    int f1 () {
        int x ;
        int y ;

        int f2(int x,intprocint p) {
            int f3 (intprocint p){ return p(y+x); }
            x = 11;
            y = y + f3(p);
            return y;
        }

        int g2(int y) {
            int z ;
            intprocint g3(intprocint p){ return (p) ; }
            z = x + y;
            if (x>0) return ((g3(g2))(x-1));
        }

        int h2(int z){
            y = f2(3,g2);
            return(z+y);
        }
        x = 33;
        y = h2(32);
        return y;
    }
    x = 0;
    x = f1();
}

```

Figure 9.3: Program for Exercise [117](#)

PROGRAMMING-LANGUAGE SEMANTICS AND COMPILER DESIGN
(SÉMANTIQUE DES LANGAGES DE PROGRAMMATION ET COMPILATION)