UNIVERSITÉ Grenoble Alpes

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

Tutorials / Travaux Dirigés (Part 2)

Univ. Grenoble Alpes — UFR IM^{2}AG

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

Academic Year 2024 - 2025

CONTENTS

SERIES

ONE

SEMANTIC ANALYSIS - TYPING

Exercise 1 — Program correctly typed or not?

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

1. Program 1:

   $x_1 := 3;$
   `while` $\neg x_3$ `do`
   $x_1 := x_2 + 1;$
   $x_3 := x_3$ `and true`
   `od`

2. Program 2:
   $x_1 := 3 * x_1 + 1;$
   `if` x_2 `and` $\neg x_3$ `then`
   $x_1 := x_2 + 1$
   `else`
   $x_1 := x_2;$
   `fi`

Exercise 2 — 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 3 — 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 `Int2Float` type conversion operator.

Exercise 4 — Other forms of variable declarations

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

1. Initialized un-typed variables: `var` $x := e$
2. Initialized typed variables: `var` $x := e : t$

Exercise 5 — Collateral evalution of declarations

Consider the sequence of initialized and un-typed variable declarations.

$$D_V = \texttt{var } x_1 := 3; \texttt{var } x_2 := 2 * x_1 + 1; \texttt{var } x_3 := \texttt{true}$$

1. Compute the resulting environment by updating an empty environment $\Gamma_V = [\,]$ with D_V using *sequential* evaluation.
2. We now propose a so-called *collateral* way of evaluating a list of variable declaration D_V, where each declaration is considered in isolation, without taking into account the previous ones in the list[1].
   - Give the corresponding typing rules.
   - Using these rules compute the resulting environment by updating an empty environment Γ_V with D_V using *collateral* evaluation.
   - Some question where $\Gamma_V = [x_1 \mapsto Int]$.

Exercise 6 — 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 and therefore x has to exist in the current environment.

Exercise 7 — Procedures without parameters

We consider the three programs below. Applying the rules, determine the variable environment at observation points (1) and (2).

C1
```
global x:Int, y:Int in
    proc swap_x_y var t:Int in
            /* (1) */
    end
 /* (2) */
 swap_x_y
```

C2
```
global x:Int, y:Int, t:int in
    proc swap_x_y var x:Int, y:Int in
            /* (1) */
    end
 /* (2) */
 swap_x_y
```

C3
```
global x:Bool, y;Bool  in
    proc swap_x_y var x:int, y:Int, t:Int in
            /* (1) */
    end
 /* (2) */
 swap_x_y
```

[1] **Rk:** this variant corresponds for instance to the CAML construct `let x=... and y=... in ...`

Exercise 8 — (Mutually-)recursive procedures

We consider the two following sequences of procedure declarations `D1` and `D2`:

```
D1=
   proc p is call p end
```

```
D2=
   proc p1 is call p2 end ;
   proc p2 is call p1 end ;
```

1. Show that, with the type system defined in the lecture, these declarations are *incorrect* when starting from an empty procedure environment.
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 9 — 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 **at compile-time** 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 procedures).
2. Apply the type system to the following code snippet, using $\Gamma = \emptyset$:
   a) $x := 1$; `if` $x = 0$ `then` $y := x + 1$ `else` $y := x - 1$ `fi`
   b) $x := 1$; `if` $x = 0$ `then` $x := x + 1$ `else` $y := x - 1$ `fi`,
   c) $x := 1$; `while` $x \leq 10$ `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.
4. Another option would be to perform this check **at runtime**, as it could be the case for instance in Python. Propose an extension of the **(natural) operationnal semantics** of the Whilelanguage to take into account this dymanic typing rule.
   Indication: you can extend the statement configurations with a new element Γ where Γ is the set of initialized variables at each program location.
5. Check that the program rejected by static typing is now no longer rejected ...

Exercise 10 — Procedures with parameters

Using the rules seen during the lectures, check the type correctness of the two following programs:

```
global r:Int, x:Int in
   proc add_r(x:Int, y;Int) is var s:Int in s:=x+y ; r:=s end
add_r(x,x+2)
```

```
global r:Int in
   proc foo (x:Int) is var x:Bool in r:=x+1 ;
foo(x)
```

Exercise 11 — 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 While.
2. Extend the type system of While accordingly.

Exercise 12 — input/output parameters to procedures in the type system

We aim at extending the While language to consider *input and/or output parameters* for 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 13 — 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):

$$\begin{array}{rcl} S & ::= & \cdots \mid \texttt{begin}\ D_T\ ;\ D_V\ ;\ S\ \texttt{end} \\ D_T & ::= & \texttt{type}\ t\ \texttt{extends}\ B_T; D_T \mid \varepsilon \\ B_T & ::= & \texttt{Top} \mid \texttt{Int} \mid \texttt{Bool} \mid t \end{array}$$

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. This question requires to be familiar with the natural operational semantics of While. 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.

SERIES

TWO

INTERMEDIATE-CODE OPTIMIZATION USING DATA-FLOW ANALYSIS

2.1 Defined and Used variables

We recall the syntax of the While language:

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

2.1.1 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 14 — 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 15 — Applying `Def` and `Use`

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

2.1.2 Adding an operator $\|$ for parallel execution

We extend the syntax of statements in the following way:

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

Exercise 16 — Parallelism in Natural Operational Semantics

We consider operator $\|$ for parallelizing statements.

1. Recall the natural operational semantics of this operator.

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

Exercise 17 — Applying the parallelism operator

Applying natural operational semantics rules extended with the rules for $\|$, 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 18 — 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 19

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

Exercise 20

1. Apply the previously defined semantic rules to the statement in Exercise 17.

2.2 Preliminaries

2.2.1 Lattice, recursive equations, fix-points

Exercise 21

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 22

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: $F1(x) = 2^{\{a,b,c\}} \setminus x$
- "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 $F_2(\ldots F_2(F_2(\emptyset)))$.

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

2.2.2 Control-flow Graph (CFG)

Exercise 23

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 24

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.

2.3 Available Expressions

Exercise 25

We consider the program in Exercise 24 and its CFG.

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

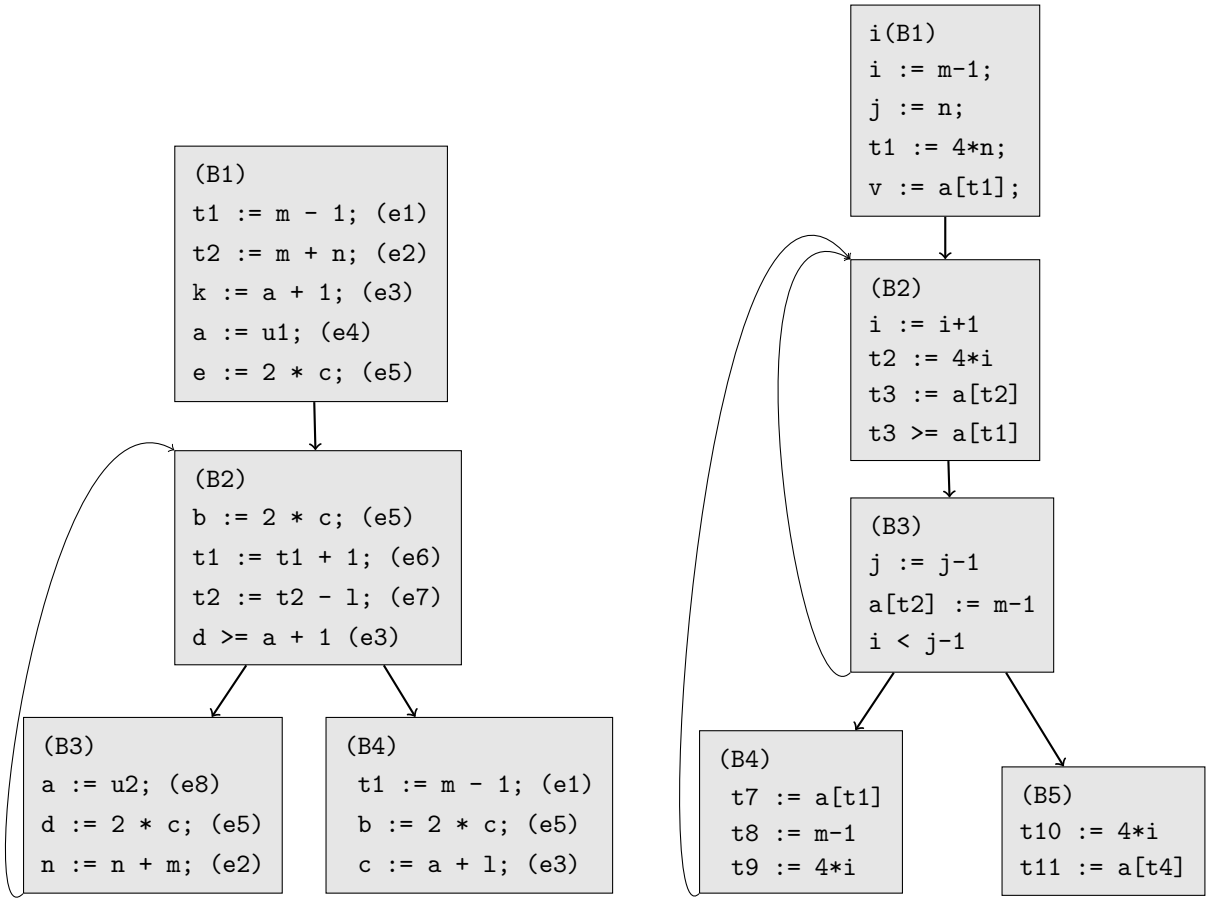

Figure 2.1: CFGs for Exercise 26

Exercise 26

For each of the CFGs in Figure 2.1:

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

Exercise 27

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
```

2.4 Live Variables

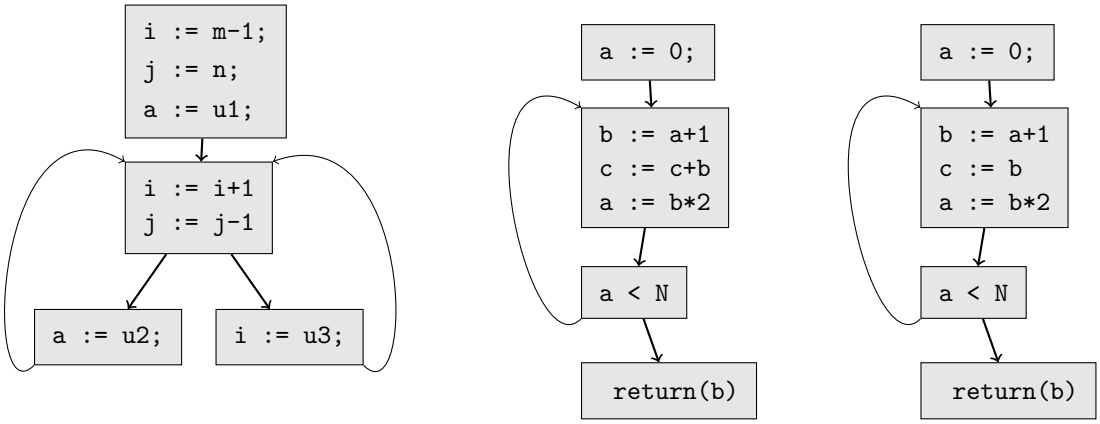

Figure 2.2: CFGs for Exercise 29

Exercise 28

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 29

For each of the CFGs depicted in Fig. 2.2 (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*.

2.5 Constant Propagation

Exercise 30

We consider the CFGs in Figure 2.3.

1. Modify these CFGs by performing constant propagation.

Exercise 31

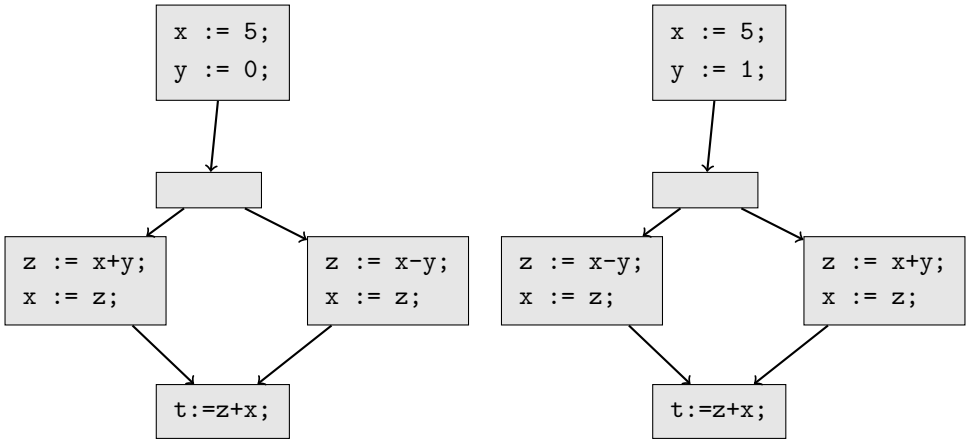

Figure 2.3: CFGs for Exercise 30

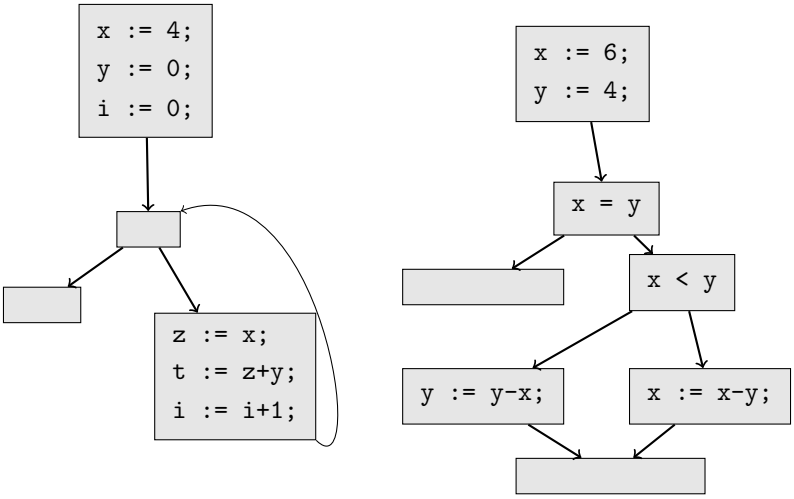

Figure 2.4: CFGs for Exercise 31

We consider the CFGs in Figure 2.4.

1. Modify these CFGs by performing constant propagation.