Exercises (TD)
Compilation and Program Analysis (CAP)

1 Code production and register allocation

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

- A multiplication instruction `mul` t1, t2, t3 that computes t1 := t2*t3.
- An "immediate load" instruction `li` t1, 4.
- A notation $[n]$ for the stack slot in which n is stored.

Question #1

Generate a 3 address-code for E with temporaries using the `ld` instruction to load n. Do it as blindly as possible (no temporary recycling).

Question #2

Draw the liveness intervals without applying liveness analysis. How many registers are sufficient to compute this expression?

Question #3

Draw the interference graph (nodes are variables, edges are liveness conflicts).

Question #4

Color this graph with three colors using the algorithm seen in the course.

Question #5

Give a register allocation with $K = 2$ registers using the iterative register allocation algorithm seen in course.

2 Program Slicing

Let us now consider the following scenario: through testing, we have identified a variable taking an incorrect value. We want to inspect only the part of the program that might influence this variable: such a part of the program is called a "slice". In this exercise, we will design an algorithm to statically compute the slice of an SSA program.

In order to compute the slice with respect to a given variable v, we need the dependencies of v. We consider both direct (i.e. non-transitive) dependencies, and transitive dependencies. We assume we can take the transitive closure of direct dependencies (by denoting it with a star *).

2.1 Data Dependencies

Let us consider a first notion of dependencies.

Definition 1 (Direct data dependencies) *A variable v depends directly on a variable u in a program P if P contains an instruction that reads u and defines v, e.g. $v := u + 1$.*

```
int x = 42;
int y = 3;
int z = 2;
while (z <= 100) {
  if (y > 10) {
    x = x + 1;
    y = y / 2;
  } else {
    x = x - 1;
    z = z * y;
  }
}
return x
```

Figure 1: Program 1

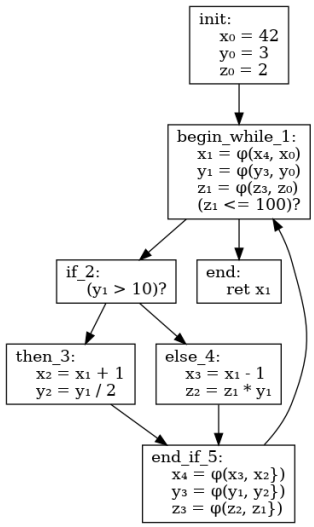


Figure 2: Program 1 in SSA form

Question #1

Give the direct data dependencies of all variables in the SSA representation of Program 1 (given on Figure 2).

Question #2

What are all the variables that influence the value of y_1 in Program 1, according to the direct data dependencies computed in the previous question?

Question #3

Give an algorithm $DD(P)$ computing direct data dependencies in an SSA program P as a dictionary.

Question #4

Give an algorithm $DD^*(P, v)$ computing transitive data dependencies of a variable v in an SSA program P. You can use the transitive closure operation.

Question #5

Remove all the instructions of Program 1 on which y_1 has no transitive data dependencies. Does this slice captures every instruction that might influence the value of y_1? What is missing?

2.2 Control Dependencies

We now consider a new kind of dependencies, to take into account what was missing with the previous notion.

Definition 2 (Direct control dependencies) *A variable u is a direct control dependency of a variable v in a program P if u is used in the predicate of a branch of P that determines a definition of v*

E.g. in "if $u > 0$ then $v = 0$ else $u = 0$", u is a direct control dependency of v. Be careful with how phi-nodes are treated.

Question #6

Give the direct control dependencies of all variables in the SSA representation of Program 1 (given on Figure 2).

Question #7

Compute the slice of Program 1 with respect to y_1 using both (transitive) data and control dependencies.

Question #8

We now assume to have an algorithm $CD(P)$ computing the direct control dependencies in an SSA program P as a dictionary. Write an algorithm $slice(P, v)$ which slices the program P with respect to the variable v using both data and control dependencies.

3 If-conversion and Predicated Instructions

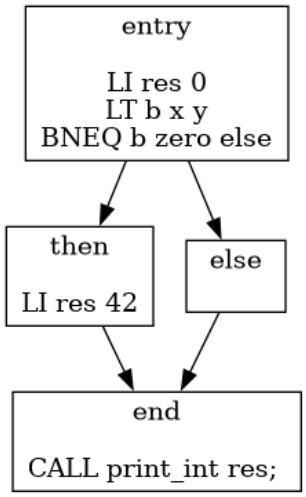
If-conversion is a transformation of the CFG which converts control dependencies (conditional jumps) into data dependencies. It is particularly useful for loop bodies, where performance is critical.

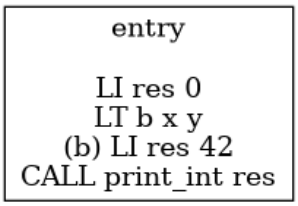
Below is an example of if-conversion from the program on the left-hand side, with a branch, to the program on the right-hand side, without branches, but a dependency on `b` instead.

For this purpose, we consider predicated instructions: such instructions can have a boolean guard (`b` below) indicating if the instruction should be executed or not.

For instance in the program below on the right, we first put the boolean value of $x < y$ in the `b` register using the `LT` (less than) instruction. Then `(b) LI res 42` loads the immediate 42 in the temporary `res` only if `b` contains 1, the value representing "true". Such instructions are available in many architecture, such as X86.

In the following, we assume all instructions can be predicated by a register or temporary.





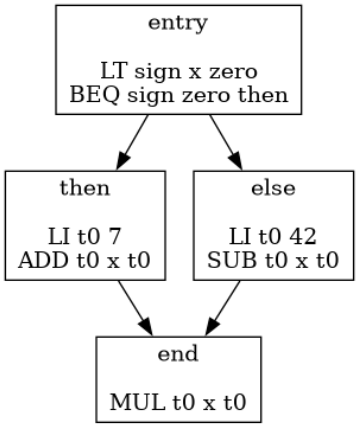
In the rest of this section, we only consider pieces of control flow graphs which are **not in SSA form**.

We note $pred(B)$ and $succ(B)$ the immediate predecessors and successors of B. We note $cond(B)$ the condition of the jump at the end of B, $br_{true}(B)$ (resp $br_{false}(B)$) the branches if the condition is true (resp false).

For a given node B, we note $predicate(B)$ a logical expression that predicates its execution given by a function *predicate*. For instance in the program above, $predicate(\texttt{else}) = x < y$.

Question #1

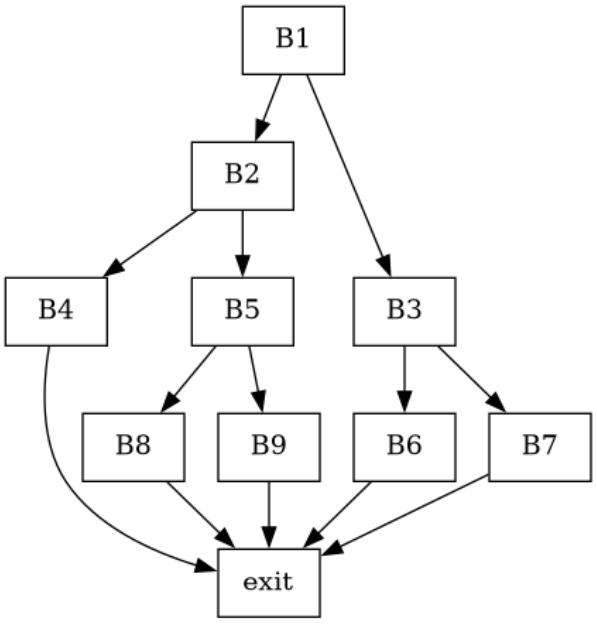
We first consider the following program:



Give *predicate* for each basic bloc and write the if-converted code.

3.1 Tree programs

We now consider a first class of CFG: trees with single entry, single output, and outdegree 2 for any other block, noted $Tree_1$. Let $B_0, \ldots, B_n, E$ be some basic blocks such that the B_i form a tree and all leaves lead to E. Here is an example:



Question #2

In which cases can such program appear ? Give an example WHILE program that would result in such a CFG.

Question #3

Given a program $P = B_0, \ldots, B_n, E$ in $Tree_1$:

(a) What is $predicate(E)$?

(b) How to compute $predicate(B_i)$ for any B_i for a program in $Tree_1$? Give an algorithm and justify it. HINT: Proceed by induction on the tree.

Question #4

To emit linear code that contains all blocks, we must order them appropriately. Let us note $sortBlock(P)$ the list of blocks in such order.

What is a sufficient condition on the list of blocks for the resulting linear code to be correct after if-conversion ?

Question #5

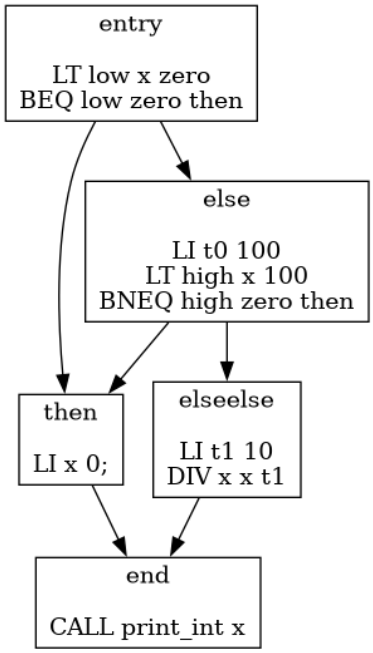
Consider we are given the *predicate* and *sortBlocks* functions from the last two questions. We also provide two utilities: `emit(L)` takes a list of (potentially predicated) instructions L and

emits them. `bool2instr(b,e)` turns a boolean expression e (such as a predicate) into a list of instructions and assigns it to the temporary b. Recall that instructions are predicated by a register or temporary.

Write the algorithm for if-conversion of arbitrary $Tree_1$ programs. Justify its correctness.

3.2 DAG programs

We now consider a richer class, DAG_1, of directed acyclic graphs with one entry and one exit and outdegree 2 for any other block. Here is an example:



Question #6

Give *predicate* on each basic block. Justify your answer on the block "`then`".

We consider the notion of control-dependence on blocks: A is control-dependent on B if, given U and V the successors of B, without order,

- There exists a path from U to A
- No paths from V lead to A

We note $ControlDeps(A)$ the control dependencies of A.

Question #7

In the program above, we have $ControlDeps(\texttt{then}) = \{\texttt{else}\}$. Justify it.

Compute and justify the control dependencies for all other blocks.

Question #8 (Difficult)

For B a block in a program P, express $predicate(B)$ in term of $ControlDeps$.

Question #9 (Difficult)

In which order should the blocks be linearized ? Give a graph walk that achieves it.

4 Security levels

4.1 Introduction

Security conscientious developers must always take care to distinguish two types of data: public data, which can be printed and sent on the network, and private data, which should never be printed, stored, or revealed in any way. As the last few years of security vulnerability have shown, ensuring that no private data can be revealed is cumbersome and error prone to do by hand.

In this exercise, we will consider how to help developers identify if their program can leak information, through a static analysis. The general approach will be to "tag" values with either `HIGH` security or `LOW` security, and prevent mixing of security levels. The purpose of the analysis performed is to prevent `HIGH` level security information to leak toward `LOW` level context. In real life a typical leak would happen through printing but we only consider assignments to public variables here. Values with `HIGH` security must not be disclosed and thus any operation that uses a `HIGH` security value must produce a `HIGH` security value.

We consider the following syntax for a **WHILE** language with security (without functions). We denote $\mathcal{S} = \{HIGH, LOW\}$ and let s range in $\mathcal{S}$ ($s \in \{HIGH, LOW\}$).

We first define expressions e (only constants are changed wrt. **WHILE**):

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

Then we redefine statements S (unchanged wrt. **WHILE**):

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

And finally programs P with variable declarations are defined as follows (only types are changed wrt. **WHILE**):

$$\begin{array}{rcll} P & ::= & D; S & \text{program} \\ D & ::= & \textit{var } x : \tau_s \mid D; D & \text{variable declaration} \end{array}$$

τ_s are types with security information, their syntax is as follows:

$$\tau_s ::= int^s \mid bool^s$$

4.2 Small-step semantics and static analysis for private and public data

We define the static semantic for a language enforcing the absence of information leak, based on the **WHILE** language. Each value is a couple of a normal value and a security tag. Variable declarations are also tagged.

Additionally to the store σ that maps variables to values and has no security information, we have a mapping from variable names to security levels: $\rho : Var \to \mathcal{S}$. The small-step semantics now has the form: $(S, \sigma, \rho) \Rightarrow (S', \sigma', \rho')$ or $(S, \sigma, \rho) \Rightarrow (\sigma', \rho')$ for the last stage. We insist that σ is unchanged and has no security information.

Expression evaluation should be a function of the form $Val(e, \sigma, \rho) = v^s$ where v is an integer or boolean value; note that *Val* now also returns a security level s.

Question #1

Inspired by the semantics for expression evaluation (in the companion sheet) write the semantics for expression evaluation with security. You can use a function max (resp. min) that takes the maximum (resp. minimum) of two security tags (we consider that `HIGH` > `LOW`). It should encode the fact that `HIGH` security values must not be disclosed and thus any operation that uses a `HIGH` security value must produce a `HIGH` security value.

Note: recall that $\sigma(x)$ is a value with no security information.

The semantics of the **WHILE** language is unchanged except:

1. The security levels are remembered at runtime in the ρ mapping.
2. Assignment checks the compatibility between the assigned value and the variable storing the value.

Question #2

Write the inference rules that build ρ from the set of variable declarations D. It should build judgments of the form $\vdash D \mapsto \rho$.

Question #3

Explain the following rule (small-step semantics for assignment) in 2-3 lines.

$$\frac{Val(e, \sigma, \rho) = v^s \qquad \rho(x) \geq s}{(x := e, \sigma, \rho) \Rightarrow (\sigma[x \mapsto v], \rho)}$$

Question #4

Write the rules for evaluating the sequence (small-step).

Question #5

Evaluate the following program according to your rules (we use `^H` and `^L` for the security tags in the code snippets). What is the configuration reached at the end? Explain.

```
var x:int^L;
x:= 5^L+3^L;
x:= 5^L-2^H
```

Question #6

Modify the semantics so that any invalid assignment due to a security reason reaches a new configuration called `SecurityError`. What is changed in the evaluation of the program above with your new semantics?

Question #7

Inspired by the type system of **WHILE**, write a "security" type system for our new language. You do not have to check standard typing. You should provide a security type judgment for expressions, statements and programs. Security judgments should use $\Vdash$ and have the form $\rho \Vdash e : s$ (e has security level s under ρ), $\rho \Vdash S$ (S is well-typed for security under ρ) and $\Vdash P$ (P is well-typed for security).

Question #8

Show that the following property does not hold, you should explain why and provide a counter example:

$$\rho \Vdash e : s \implies \exists v,\ Val(e, \sigma, \rho) = v^s$$

Question #9

Using the type-system we have seen in the course, i.e. supposing $\Gamma \vdash e : \tau$ for the right Γ, state a property that ensures that $Val(e, \sigma, \rho)$ reduces to a value of the right security level.

Question #10

Prove the following property above for a reduced syntax for expression, which contains only constants c^s, addition $e + e$, and comparison $e < e$:

$$\rho \Vdash e : s \wedge Val(e, \sigma, \rho) = v^{s'} \implies s = s'$$

Question #11

Express a simple preservation property for your "security" type system and informally explain why it is true (it is simpler to prove it here than for the classical type system).

Question #12

Express formally the property of the form "well-typed programs do not go wrong" guaranteed by your "security" type system. Explain informally what this guarantees.

Question #13

Prove the property you stated in the question above in the case of a single assignment statement.

Question #14

A type system always has to be conservative and reject programs that would not lead to an error. Give an example of a program that does not reach `SecurityError` but is rejected by your "security" type system.

Information could leak due to a conditional test: consider the program in Figure 3.

Question #15

What is wrong with this program? Explain how confidential information can flow without raising an error.

```
var confidential:int^H;
var b:bool^L;
confidential:=5^H;
if (confidential > 3^L) then
  b:=true^L
else
  b:=false^L
```

Figure 3: A program branching on secrets

Question #16 (Difficult)

Propose an extension of the semantics for **WHILE** with security that tracks such information flow (based on the security level of the if condition) and raise an error in the case above. Explain your changes and only provide formal definitions for rules that are significantly changed.

You should make sure that you take into account nested ifs. You should make sure that the while condition is correctly tracked too.

Hint: one possible solution is to extend the syntax for statements and tag some of the statements with security information.

Question #17 (Difficult)

Propose an extension of your "security" type system that allows you to guarantee that no `SecurityError` configuration is reachable.