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.

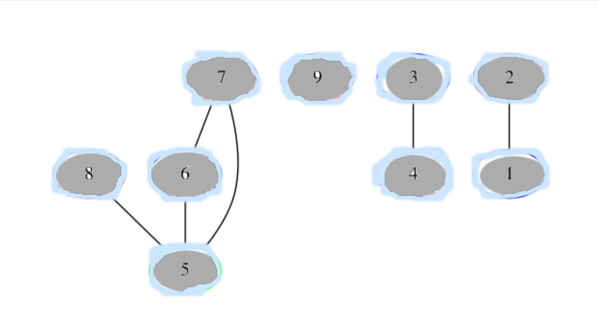
Solution:

1. Code in the array below
2. Here it is:

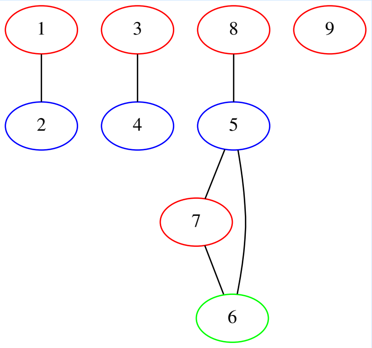
code	t1	t2	t3	t4	t5	t6	t7	t8	t9
`ld t1,[n]`									
`li t2,1`	*								
`add t3,t1,t2`	*	*							
`ld t4,[n]`			*						
`mul t5,t4,t3`			*	*					
`li t6 2`					*				
`ld t7,[n]`					*	*			
`mul t8,t6,t7`					*	*	*		
`add t9,t5,t8`					*		*		
`...`								*	

There are 3 registers alive simultaneously at the control point (*). 3 registers will be sufficient.

3. The interference graph is:



4. With three colors it works.



5. The colouring algorithm with $K = 2$ gives the following result :

tempo	t1	t2	t3	t4	t5	t6	t7	t8	t9
affinity=color	1	0	1	0	SPILL	1	0	0	0

Renaming t5 into two new temporaries : `t5_1` and `t5_2` and generate the spilled 3 address code, with live ranges:

code	t1	t2	t3	t4	t51	t52	t6	t7	t8	t9
`ld t1,[n]`										
`ldi t2,1`	*									
`add t3,t1,t2`	*	*								
`ld t4,[n]`			*							
`mul t51,t4,t3`			*	*						
`st t51,[.]  (spill)`					*					
`ldi t6 2`										
`ld t7,[n]`							*			
`mul t8,t6,t7`							*	*		
`ld t52,[.]  (spillcode)`									*	
`add t9,t52,t8`						*			*	
`...`										*

The colouring algorithm with $K = 2$ now gives:

tempo	t1	t2	t3	t4	t51	t52	t6	t7	t8	t9
register	1	0	1	0	0	1	1	0	0	0

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

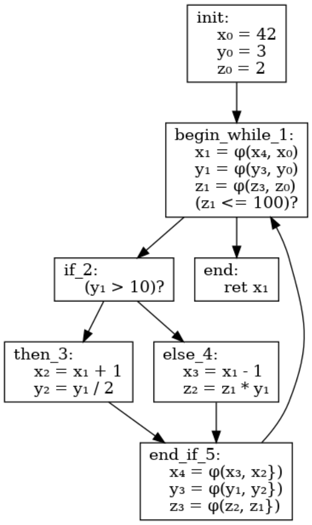


Figure 2: Program 1 in SSA form

Question #1

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

Solution:

x_0	
x_1	x_0, x_4
x_2	x_1
x_3	x_1
x_4	x_2, x_3
y_0	
y_1	y_0, y_3
y_2	y_1
y_3	y_1, y_2
z_0	
z_1	z_0, z_3
z_2	z_1, y_1
z_3	z_1, z_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?

Solution: We take the transitive closure, which yields for y_1: y_0, y_1, y_2, y_3.

The transitive closure as a whole is given by the following table:

x_0	
x_1, x_2, x_3, x_4	x_0, x_1, x_2, x_3, x_4
y_0	
y_1, y_2, y_3	y_0, y_1, y_2, y_3
z_0	
z_1, z_2, z_3	z_0, z_1, z_2, z_3, y_1

even though to answer the question only the part concerning variables y_i is needed.

Question #3

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

Solution: Walk through the CFG under SSA form. At each declaration $v = e$, add an entry $v \mid var(e)$ in the dictionnary.

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.

Solution: Take the transitive closure of $DD(P)$, and access on v.

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?

Solution: We obtain a slice without any z_i or x_i. The loop will spin: control 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).

Solution:

x_0	
x_1	
x_2	z_1, y_1
x_3	z_1, y_1
x_4	z_1
y_0	
y_1	
y_2	z_1, y_1
y_3	z_1
z_0	
z_1	
z_2	z_1, y_1
z_3	z_1

Question #7

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

Solution: All x_i are removed, the control structure remains.

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.

Solution:

1. Compute the transitive closure of $DD(P) \cup CD(P)$ (possible by hypothesis) and evaluate it on v.

   Thus you obtain the transitive dependencies of v: we call this $D(P, v)$.

2. For any assignment to some variable x (including ϕ nodes), keep the assignment if and only if $x \in D(P, v)$.

3. If a test has no variables in $D(P, v)$, remove this test and keep only the negative edge of the jump.

   This is to prevent having two edges from a same block without any branching, we choose arbitrarily.

4. When both edges of some test go to the same block, remove one edge and the test.

5. Replace empty blocks with a predecessor and a successor with an edge from the former to the latter.

6. Remove all remaining empty blocks.

NB: $CD(P)$ is computed using post-dominators.

3 If-conversion and Predicated Instructions

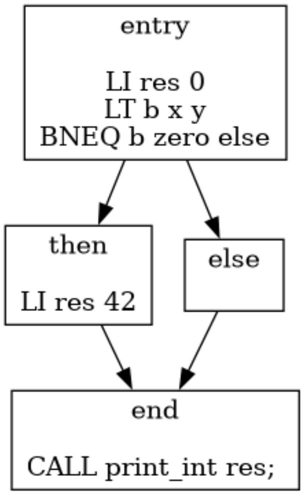
If-conversion is a transformation of the CFG which converts control dependencies (conditional jumps) into data dependencies. It is particularly useful for loop bodies, where performance is critical.

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.

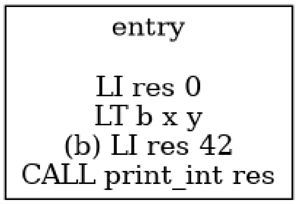
In the following, we assume all instructions can be predicated by a register or temporary.



```
entry:
LI res 0
LT b x y
BNEQ b zero else

then:
LI res 42

end:
CALL print_int res;
```



```
LI res 0
LT b x y
(b) LI res 42
CALL print_int res
```

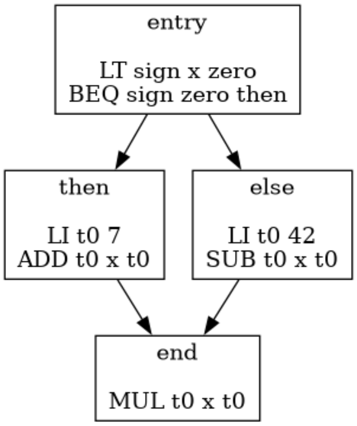
In the rest of this section, we only consider pieces of control flow graphs which are **not in SSA form**.

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:



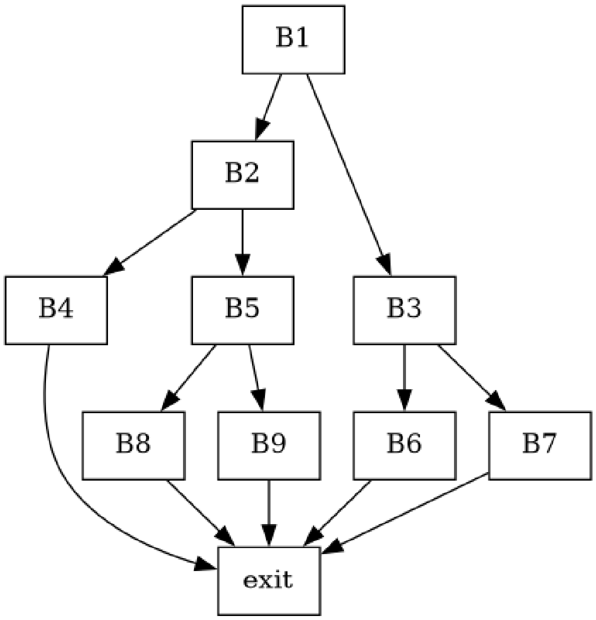
Give *predicate* for each basic bloc and write the if-converted code.

Solution:

$$predicate(entry) = True \qquad predicate(else) = (x < 0)$$
$$predicate(then) = (x \geq 0) \qquad predicate(end) = True$$

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.

Solution: if-then-else chains:

```
if (x < 0) {
  return 1;
} elif (x > 0) {
  return -1;
} else (x == 0) {
  return 0;
}
```

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.

Solution: $predicate(E) = True$

if $pred(B_i) = \emptyset$, then $predicate(B_i) = True$.

Otherwise, by definition of a tree, $pred(B_i) = \{B_j\}$. if $B_i = br_{true}(B_j)$, then $predicate(B_i) = cond(B_j) \wedge predicate(B_j)$ else, $B_i = br_{false}(B_j)$, then $predicate(B_i) = \neg cond(B_j) \wedge predicate(B_j)$

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 ?

Solution: Any order where parents are before their children works. for instance DFS and BFS (with parents before children).

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.

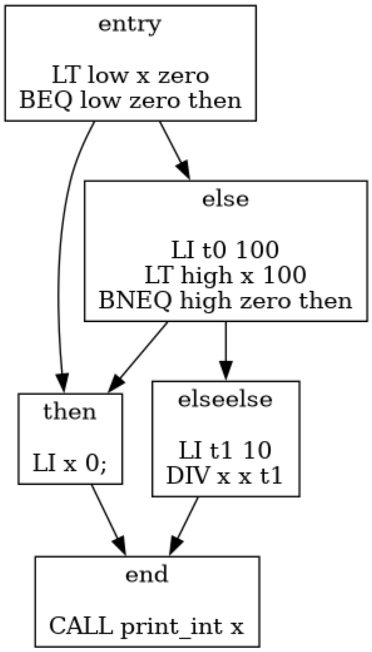
Write the algorithm for if-conversion of arbitrary $Tree_1$ programs. Justify its correctness.

Solution:

```
$L = sortBlocks(P)$
$for\ B \in L$
    $b$ a fresh temporary
    $emit(bool2instr(b, predicate(B)));$
    $for\ I \in B$
        $emit((b)I)$
```

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`".

Solution:

$$predicate(entry) = True \qquad predicate(then) = (x < 0 \vee x > 100)$$
$$predicate(else) = (0 < x) \qquad predicate(elseelse) = (0 < x < 100)$$
$$predicate(end) = True$$

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.

Solution:

$$CD(entry) = \{\} \qquad CD(then) = else$$
$$CD(else) = entry \qquad CD(elseelse) = entry, else$$
$$CD(end) = \{\}$$

Question #8 (Difficult)

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

Solution: $predicate(E) = True$

if $ControlDeps(B_i) = \emptyset$, then $predicate(B_i) = True$.

Otherwise for each $B_j \in ControlDeps(B_i)$. Let $B_{true} = br_{true}(B_j)$ and $B_{false} = br_{false}(B_j)$. By definition of the control dependencies, either B_{true} or B_{false} leads to B_i. Let b_j the condition that does. then $predicate(B_i) = \bigvee_j (b_j \wedge predicate(B_j))$.

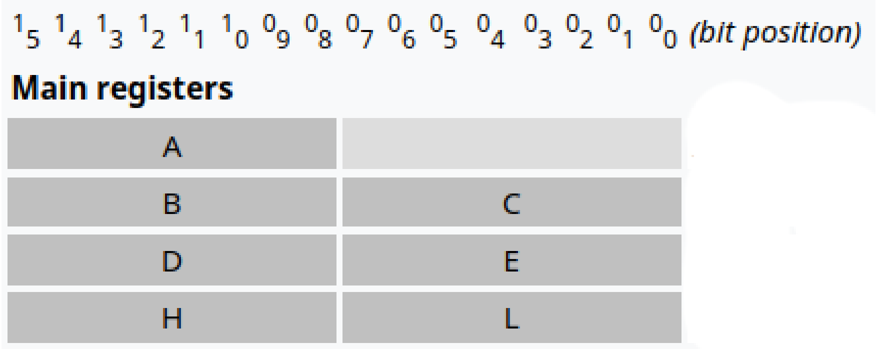
Question #9 (Difficult)

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

Solution: Topological sort

4 Duo-processor for video-game execution

The Game Boy, released in 1989, was equipped with a Z80 processor. This processor was peculiar: it was an hybrid 8bit/16bit processor containing seven 8-bit registers (A,B,C,D,E,H,L). However, some instructions operated on 16bits by pairing together two registers (BC, DE, HL). Memory was addressed by 16bit pointers. This is illustrated by the picture below.



(Image from Wikipedia)

In this exercise, we will consider a RISCV-like Instruction Set Assembly with the same register set, both 8bit and 16bits instructions, and the compilation challenge that entails. For simplicity, we will only consider straight code with no branching.

More precisely, all arithmetic instructions (ADD, SUB, DIV, MUL, XOR) come in a 16bit version (ADD16, . . .) and a 8bit version (MUL8, . . .). Immediates (used in LI, for instance) can only be 8-bit wide. LOAD and STORE are done with 16 bits addresses and access 16 bits at a time, with an optional immediate offset.

As a first example, we consider the vector multiplication of two 8 bits 2D vectors $(x_0\ x_1)$ and $(x'_0\ x'_1)$ stored at address HL and (HL+16) respectively. More precisely, we wish to implement $x_0 * x'_0 + x_1 * x'_1$. The assembly program below computes the result in register A. Note that memory access is done inside 16 bits duo-registers (here, BC and DE respectively), whose 8-bits halves are used independently to then multiply and sum the coordinates.

```
LOAD BC HL,offset(0); BC <– LOAD(HL)
LOAD DE HL,offset(16); DE <– LOAD(HL+16)
MUL8 B B D ; B <– B ∗ D
MUL8 C C E ; C <– C ∗ E
ADD8 A B C ; A <– B + C
```

Question #1

Write a program that, given an 8bit 3D vector, computes the 8-bit average of its component. The memory address of the first word of the vector is given in register HL. The result should be in register A.

Solution:

```
LOAD BC HL,offset(0)
```

```
ADD8 A B C
LOAD BC HL,offset(16)
ADD8 A A B
DIV8i A A 3
```

In the rest of this exercise, we wish to compile a single basic bloc in SSA 3-address instructions. Let us consider the following 3-address instructions:

$$
\begin{array}{ll}
i ::= \ t \leftarrow i_8 & \text{(Load 8bits Immediate)} \\
\mid t \leftarrow t' & \text{(Move)} \\
\mid t \leftarrow t' + t'' \mid t' * t'' \mid \ldots & \text{(Arithmetic Operations)} \\
\mid t \leftarrow t'.t'' & \text{(Concatenation of 8bit Words)} \\
\mid t.t' \leftarrow t'' & \text{(Split of 16bit Words)} \\
\mid t \leftarrow load(t', i_8) & \text{(Memory Load)} \\
\mid store(t, t', i_8) & \text{(Memory Store)}
\end{array}
$$

$t \leftarrow t'.t''$ concatenates two 8bits integers into a 16bits integer. $t.t' \leftarrow t''$ splits a 16bits integer in t'' into its upper part in t and its lower part in t'. Load and store take a 16bits address and return a 16bits integer, with an optional immediate offset. Programs must be in single static assignment form: variables can't be redefined.

The example above corresponds to the following 3-address program:

```
t1 <- load(t0,offset(0))
t2 <- load(t0,offset(16))
 t3.t4 <- t1
 t5.t6 <- t2
t7 <- t3 * t5
t8 <- t4 * t6
t9 <- t7 + t8
ret t9
```

Question #2

Let us consider the following program:

```
t1 <- load(t0, offset(0))
 t2.t3 <- t1
t4 <- t3 * 2
t5 <- t2.t4
store(t5, t0, offset(0))
```

What does it do?

What is the purpose of such a program?

Question #3

Recall the goal of instruction selection and instruction scheduling.

Solution: See course

Question #4

Consider the instruction: $t9 \leftarrow t7 + t8$. What is missing from the syntax to write instruction selection for this instruction?

Solution: Size information on the arithmetic operations.

Question #5

We now assume that there exists an analysis that provides a function `size` which, given a temporary, provide the size of the register it should be stored in: $\texttt{size}(t) = 8$ or $\texttt{size}(t) = 16$.

Do the instruction selection for the program in Question 2, with the following size function:

temporary	size
t2, t3, t4	8bits
t0, t1, t5	16bits

Solution:

```
t1 <– load(t0, offset(0))
 t2.t3 <– t1
t4 <– MUL8i t3 2
t5 <– t2.t4
store(t5, t0, offset(0))
```

Question #6

Propose an instruction selection algorithm for the given instruction language.

Solution: Mostly syntax directed (+ size function on the arith expr)

Question #7

Recall the goal of register allocation. What is the usual technique to solve such problem? What are the challenges for register allocation in this architecture?

Question #8

Give a register allocation for the program obtained in Question 5 and compile it to valid assembly.

Solution: Magic! the cast disappear:

```
LOAD BC HL offset(0)
MUL8i C C 2
STORE BC HL offset(0)
```

Question #9

Let us now consider the following graph-coloring problem: given a set $\mathcal{C}$ of colors, a compatibility relation $\sim$ on colors, and $G = (V, E)$ a graph with vertices V and edges E, the algorithm $color(\mathcal{C}, \sim, G)$ returns a map M from V to $\mathcal{C}$ such that for all $(v, v') \in E$, $M(v) \sim M(v')$.

Let B a block in SSA 3-address instructions presented above. Formulate informally how to compute register allocation of B using the *color* algorithm. Demonstrate it on the program from Question 2.

Remark: It is not required to propose an implementation for the *color* algorithm.

Question #10 (Difficult)

Propose informally an analysis that would allow obtaining the `size` function.

5 Exceptional expressions

In this exercise, we consider expressions that can fail, but only locally. More precisely, we enrich a traditional language of expressions, as the one we studied in **WHILE**, with two new constructs.

- `fail`(e) indicates that the execution fails with the value returned by e.
- `guard`(e) evaluates an expression e, catching its potential failure.

These primitives allows one to write expressions like:

$$\texttt{guard}(3 + \texttt{fail}(\texttt{true}))$$

that returns true because the expression fails but the failure is caught.

On the other hand "`guard`$(3 + 4)$" will return 7 as no failure occurred.

Such exceptions are particularly useful when combined with ternary conditions of the form $e \;?\; e_t \;:\; e_e$ that allow for conditional evaluations inside expressions. We will introduce these conditionals in the exercise.

5.1 Syntax and Semantics

The syntax and semantics of expressions is given below. It is similar to expressions in **WHILE**. We consider the new domain of exceptional values $v^* \in \mathcal{V}^*$, which are either a usual value $v \in \mathcal{V}$, or a value returned by raising an exception.

Grammar:

$$\begin{array}{rll} v \in \mathcal{V} ::= & i \in \mathbb{Z} & \text{Integers} \\ & \mid b \in \mathbb{B} & \text{Booleans} \\ v^* \in \mathcal{V}^* ::= & v \in \mathcal{V} & \text{Success} \\ & \mid \textbf{Fail}(v) & \text{Failure} \\ & & \\ e \in \mathcal{E} ::= & c & \text{Constants} \\ & \mid x & \text{Variable} \\ & \mid e + e & \text{Addition} \\ & \mid e \parallel e \mid e \,\&\&\, e & \text{Bool Ops} \\ & \mid \texttt{fail}(e) & \text{Failure} \\ & \mid \texttt{guard}(e) & \text{Guard} \end{array}$$

Evaluation: $Val : \mathcal{E} \times State \to \mathcal{V}^*$

$$Val(c, \sigma) = value(c) \qquad Val(x, \sigma) = \sigma(x)$$

$$Val(e_1 + e_2, \sigma) = Val(e_1, \sigma) +^* Val(e_2, \sigma)$$

$$Val(e_1 \parallel e_2, \sigma) = Val(e_1, \sigma) \parallel^* Val(e_2, \sigma)$$

$$Val(e_1 \,\&\&\, e_2, \sigma) = Val(e_1, \sigma) \,\&\&^*\, Val(e_2, \sigma)$$

$$\frac{Val(e, \sigma) = v}{Val(\texttt{fail}(e), \sigma) = \textbf{Fail}(v)} \qquad \frac{Val(e, \sigma) = v}{Val(\texttt{guard}(e), \sigma) = v}$$

$$\frac{Val(e, \sigma) = \textbf{Fail}(v)}{Val(\texttt{guard}(e), \sigma) = v}$$

We extend all binary operators ($+, \parallel, \&\&$) to exceptional values ($+^*, \parallel^*, \&\&^*$). Below is the definition for $+^*$, the others are similar:

$$n +^* n \equiv n + n \qquad\qquad v_2 +^* \textbf{Fail}(v) \equiv \textbf{Fail}(v) +^* v_2^* \equiv \textbf{Fail}(v)$$

In the examples, we also use comparison operators between integers ($<$, $>$, ...) with the straightforward semantics and its extension to failed values.

Expression evaluation is defined as $Val(e, \sigma)$ like for **WHILE**. When σ is not given, it is $\emptyset$.

Note that fail expressions are evaluated when they are encountered and are propagated up in the expression.

Question #1

What does the expression below evaluates to? Give its derivation.

$$\texttt{guard}(3 + \texttt{fail}(\mathsf{true} \,\&\&\, \mathsf{false})) \,\&\&\, \mathsf{true}$$

Solution: false +give derivation

Question #2

We now add a ternary conditional operator $e \; ? \; e_{then} \; : \; e_{else}$. It evaluates e to a boolean. If it is true, it evaluates e_{then}, otherwise e_{else}. If any of these expressions fail, it fails as well. If the condition fails none of the branches is evaluated.

Give the evaluation rules for this new construct.

Solution:

$$\frac{val(eb) = true \qquad val(ethen) = v*}{Val(eb)?ethen : eelse = v*} \qquad\qquad \frac{val(eb) = fail(v)}{Val(eb)?ethen : eelse = fail(v)}$$

+ false case

Question #3

Do the provide rules allow to evaluate $\texttt{guard}(\texttt{fail}(3 + \texttt{fail}(4)))$? Propose an appropriate reduction rule that ensures that nested fails are evaluated to an exceptional value.

Solution:

$$\frac{Val(e) = Fail(v)}{Val(Fail(e)) = Fail(v)}$$

Question #4

Given the environment σ which associates x to an integer, what is the evaluation of (depending on the value of x):

$$\texttt{guard}(x > 0\ ?\ \texttt{fail}(\texttt{true})\ :\ \texttt{fail}(-x)\ ) + 10$$

What is the problem ?

Solution: if x is positive it reduces to true + 10 that is not well typed and is stuck

5.2 Typing

We suppose that the evaluation rules you defined in the preceding questions are now part of the semantics.

We want to design a type system to ensure that failure values are type-compatible, and that guards return a coherent type. The purpose of the type system is to be able to reject expressions like (provided x has type $\mathbb{Z}$):

$$\texttt{guard}(x > 0 \,?\, \texttt{fail}(\texttt{true}) \;:\; \texttt{fail}(-x)\;) + 10 \qquad (a)$$

But accept expressions like:

$$\texttt{guard}((x < 2 \,?\, 3 \;:\; \texttt{fail}(\texttt{true})\;) > 5) \qquad (b)$$

We consider the types $\tau = \mathbb{B} \mid \mathbb{Z} \mid \bot$ and the typing judgement $\Gamma \vdash e : \tau_{success}, \tau_{fail}$ when expression e, with environment Γ, evaluates to a value of type $\tau_{success}$ or fails with a value of type τ_{fail}. The new type $\bot$ is used to denote types that cannot exist, like the failure type for a guarded expression for example.

We have two operations on types: $\leq$ and $\sqcup$ such that

- for all τ, $\bot \leq \tau$, and $\tau \leq \tau$;
- $\mathbb{B} \not\leq \mathbb{Z}$ and $\mathbb{Z} \not\leq \mathbb{B}$;
- $\tau_1 \sqcup \tau_2 = \tau_2$ if $\tau_1 \leq \tau_2$ and symmetrically $\tau_1 \sqcup \tau_2 = \tau_1$ if $\tau_2 \leq \tau_1$
- otherwise $\tau_1 \sqcup \tau_2$ is undefined

Selected typing rules are provided below.

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau, \bot} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1, \tau_{f1} \qquad \Gamma \vdash e_2 : \tau_2, \tau_{f2} \\ \tau_1 \leq \mathbb{Z} \qquad \tau_2 \leq \mathbb{Z} \qquad \tau_f = \tau_{f1} \sqcup \tau_{f2}\end{array}}{\Gamma \vdash e_1 + e_2 : \mathbb{Z}, \tau_f} \qquad \frac{\Gamma \vdash e : \tau_1, \tau_2 \qquad \tau = \tau_1 \sqcup \tau_2}{\Gamma \vdash \texttt{guard}(e) : \tau, \bot}$$

Question #5

Propose a typing rule for $\texttt{fail}(e)$. Illustrate it by showing the derivation for $3 + \texttt{guard}(\texttt{fail}(5))$

Solution:

$$\frac{e : \tau, \tau'}{\texttt{fail}(e) : \bot, \tau \sqcup \tau'}$$

Question #6

Show the derivation for: $\texttt{guard}(3 + \texttt{guard}(\texttt{fail}(5)))$

Question #7

Give the typing rule for the ternary conditional.

Solution:

$$\frac{e : \tau_1, \tau_1' \qquad e' : \tau_2, \tau_2' \qquad e'' : \tau_3, \tau_3' \qquad \tau_1 \leq \mathbb{B}}{e \ ?\ e' \ :\ e'' \ : \tau_2 \sqcup \tau_3, \tau_1' \sqcup \tau_2' \sqcup \tau_3',}$$

Note : we could be more precise in some cases like if $\tau_1 = \bot$

Question #8

Give the typing derivation for the expression (a) and (b) above (before the definition of the type system).

Question #9

State but do not prove a soundness theorem of your typing. It should state something about the evaluation of the well-typed expression, and something about what values can be obtained.

Solution:

$$\Gamma \vdash e : \tau.\tau' \wedge \Gamma \text{ and } \sigma \text{ agree on the types of variables } \implies$$
$$\exists v^*.\, Val(e, \sigma) = v^* \wedge (\emptyset \vdash v^* : \tau, \bot \vee \emptyset \vdash v^* : \bot, \tau') \quad (1)$$

Alternative is to say that v^* is either a value or fail of a value of the right types.

Question #10 (Difficult)

Explain in two sentences how the proof should be done. Give a formal proof of the `guard`(e) case only.

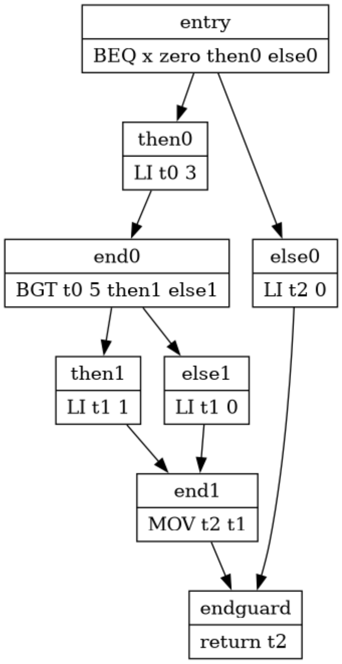
5.3 Code generation

We now want to generate a control flow graph for such expressions. For instance, the expression

$$\texttt{guard}((x \mathbin{?} 3 \;:\; \texttt{fail}(\mathsf{false})\;) > 5)$$

yields the RISCV assembly shown below, with its control flow graph. We can see that the "else" branch of the conditional "$x \mathbin{?} 3 \;:\; \texttt{fail}(\mathsf{false})$ ", which is a fail, is implemented by a direct jump to the end of program, where the guard is resolved, and thus bypassing the computation of the expression "$\ldots > 5$".

```
    BEQ x zero lbl_else0
    LI  t0  3
    J  lbl_endif0
lbl_else0 :
    LI  t2  0
    J  lbl_endguard
lbl_endif0:
    BGT t0 5 lbl_else1
    LI  t1  1
    J  lbl_endif1
lbl_else1 :
    LI  t1  0
lbl_endif1:
    MOV t2 t1
lbl_endguard:
    return  t2
```



Similarly to the course on the **WHILE** language, we consider syntax-directed rules to implement a code generation function $code(e, t)$ which emits the assembly code for e and puts its result in temporary t.

Remark: Selected code generation rules for **WHILE** are available in the companion sheet.

Question #11

Recall (informally) the steps to obtain control flow graph from a **WHILE** program.

Question #12

Propose a code generation rule for $e \mathbin{?} e_{then} \;:\; e_{else}$.

Question #13

Extend the code generation function to the context of exceptional expressions, and propose code generation rules for $\texttt{guard}(e)$ and $\texttt{fail}(e)$.

Hint: You can add new arguments to the *code* function.

Question #14

What about nested guards? Apply your rules on the example from Question 6 and show the control flow graph.

Question #15 (Difficult)

We now consider a more general exception construction where guard and fail specify a label ℓ: `fail`$(\ell(e))$ raises the exception ℓ with payload e, and `guard`(e) with ℓ catches only the ℓ-exceptions.

Propose a compilation strategy for this extension.

Question #16 (Difficult)

What are the differences in execution between our exception construct and the one in general purpose languages, such as Java, C++ or OCaml ? Could the two be used conjointly ?