Block	$$\frac{\Gamma_V \vdash D_V \mid \Gamma'_V \quad (\Gamma'_V, \Gamma_P) \vdash D_P \mid \Gamma'_P \quad (\Gamma'_V, \Gamma'_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \texttt{begin}\ D_V\ \ D_P\ \ S\ \texttt{end}}$$
Empty proc. decl.	$$\frac{}{(\Gamma_V, \Gamma_P) \vdash \epsilon \mid \Gamma_P}$$
Non-empty proc. decl.	$$\frac{(\Gamma_V, \Gamma_P) \vdash S \quad (\Gamma_V, \Gamma_P[p \mapsto \texttt{proc}]) \vdash D_P \mid \Gamma'_P \quad p \notin DP(D_P)}{(\Gamma_V, \Gamma_P) \vdash \texttt{proc}\ p\ \texttt{is}\ S\ ;\ D_P \mid \Gamma'_P}$$
Call	$$\frac{\Gamma_P(p) = \texttt{proc}}{(\Gamma_V, \Gamma_P) \vdash\ \texttt{call}\ p}$$

Figure 1: Type-checking rules for procedures

Univ. Grenoble Alpes | **Part II** | **UFR IM²AG**
MoSIG 1 | | **Year 2021-2022**

Programming Language Semantics and Compiler Design

—

Final Exam of Thursday 9 December

- **Duration**: 3h.
- 5 sheets of A4 paper are authorized.
- Any electronic device is forbidden.
- The grading scale is indicative.
- **The care of your submission will be taken into account.**
- Exercises are **independent**.
- If you don't know how to answer to some question, you may assume the result and proceed with the next question.
- The maximal grade is obtained with 20 points.

- **Submit each part on a separate answer sheet (negative point otherwise).**
- **Care will be taken into account (-1 point in case of lack of care).**
- **Unreadable parts will be ignored.**

Answer of exercise 1

1.
   - block statement:

$$\frac{(\Gamma_V, \Gamma_F) \vdash\ D_V \mid \Gamma'_V \qquad (\Gamma'_V, \Gamma_F) \vdash\ D_F \mid \Gamma'_F \qquad (\Gamma'_V, \Gamma'_F) \vdash\ S}{(\Gamma_V, \Gamma_F) \vdash \texttt{begin}\ D_V\ D_F\ \ S\ \texttt{end}}$$

The programm below is rejected by this rule because (for instance !) the block body is not well-typed (it uses an undefined variable `x`)

```
begin x := 0 end
```

   - non-empty function declaration:

$$\frac{(\Gamma_V, \Gamma_F) \vdash\ S \qquad (\Gamma_V, \Gamma_F) \vdash\ e : t \qquad (\Gamma_V, \Gamma_F[f \to t]) \vdash\ D_F \mid \Gamma'_F \qquad f \notin DF(D_F)}{(\Gamma_V, \Gamma_F) \vdash \texttt{func}\ f\ \texttt{is}\ S\ ;\texttt{return}\ e;\ D_F \mid \Gamma'_F}$$

The programm below is rejected by this rule because function declaration `f` is not well-typed (it returns an undefined variable `x`)

```
begin func f is skip ; return x ; end
```

   - empty function declaration:

$$\frac{}{(\Gamma_V, \Gamma_F) \vdash \varepsilon \mid \Gamma_F}$$

This rule is always satisfied ...

   - function call

$$\frac{\Gamma_F(f) = t}{(\Gamma_V, \Gamma_F) \vdash \texttt{call}\ f : t}$$

The program below is rejected by this rule because function `f` is not defined

```
begin call f end
```

2.
   1. Complete the following type checking rules for the assignment:

$$\frac{}{\vdash \texttt{x}\ :=\ e : Void}$$

   2. Complete the following type checking rules for sequential composition

$$\frac{\vdash S_1 : t}{\vdash S_1\ ;\ S_2 : t} \qquad \frac{\vdash S_1 : Void \qquad \vdash S_2 : t}{\vdash S_1\ ;\ S_2 : t}$$

   3. Complete the following type checking rules for conditional statement

$$\frac{\vdash S_1 : t \qquad \vdash S_2 : t}{\vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} : t}$$

Give the proof tree obtained with your rule for the following code example:

```
// statement S3
   if true then
        return true  // returns a bool
   else
        return x+1  // returns an int
   fi
```

This example is not well-typed with respect to this type system since the two alternatives do not return values of the same type:

$$\frac{\vdash\ \texttt{return true : Bool} \qquad \vdash\ \texttt{return}\ \ \texttt{x} + 1\ \texttt{: Int}}{\vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} : \ldots}$$

Since type-checking is performed at compile time, "correct" programs may be rejected. In this example the "else" branch is not executed and the return value is always `true`.

   4. Complete the following type checking rules for iterative statement:

A first solution is to simply check that the loop body satisfies P (i.e., it contains a "return" statement on each of its execution paths):

$$\frac{\vdash S : t}{\vdash \texttt{while}\ e\ \texttt{do}\ S\ \texttt{od} : t}$$

According to this solution P is not satisfied if the loop is never executed, and the loop body cannot be executed more than once (since each of its execution path contains a return statement). A better solution could be to consider that iterative statements **never** satisfy P, and hence cannot be used as the last statement of a function ...:

$$\frac{}{\vdash \texttt{while}\ e\ \texttt{do}\ S\ \texttt{od} : Void}$$

3. Rewrite the non-empty function declaration rule taking into account this new syntactic definition of functions:

$$\frac{(\Gamma_V, \Gamma_F) \vdash\ S \qquad \vdash S : t \qquad (\Gamma_V, \Gamma_F[f \to t]) \vdash\ D_F \mid \Gamma'_F \qquad f \notin DF(D_F)}{(\Gamma_V, \Gamma_F) \vdash \texttt{func}\ f\ \texttt{is}\ S;\ D_F \mid \Gamma'_F}$$

4. To reject programs containing "dead code", i.e., code lying after a return statement, we need to use (only) the following rule when type-checking a sequential composition:

$$\frac{\vdash S_1 : Void \qquad \vdash S_2 : t}{\vdash S_1\ ;\ S_2 : t}$$

According to this rule only the **last** statement of a block may return a value.

```
void main() {
      int x ;
      int F1(int u) {
         int y ;
         void G2 (int t) {
            int z ;
            z=4;
            x = y+x+z+t;
         } /* end G2 */
         void F2() {
            y=3;
            G2 (y);
         } /*  end F2 */
         F2();
         return (u+1);
      } /* end F1 */
      x=2 ;
      x=3+F1(x);
   }/* end main */
```

Figure 2: Program for exercise **??**

Answer of exercise 2

1. see the stack layout on Figure 3 below.
2. In procedure `F2`, give the sequence of instructions associated with `G2(y)`.

```
// @y = Env(F1)-4
// LS(G2)=LS(F2)=Env(F1)

LD R1, [FP+8]      // R1 = LS(F2) = Env(F1)
LD R2, [R1-4]      // R2 = y
push(y)            // push parameter y
push(R1)           // push Env(F1) = LS(G2)
CALL G2
ADD SP, SP, #8     // clean the stack
```

3. In procedure `G2`, give the sequence of instructions associated with `x=y+x+z+t`.

```
// @x = Env(main)-4
// @y = Env(F1)-4
// @z = Env(G2)-4
// @t = Env(G2)+12

LD R1, [FP+8]      // R1 = LS(G2) = Env(F1)
LD R2, [R1-4]      // R2 = y
LD R3, [R1+8]      // R3 = LS(F1) = Env(main)
LD R4, [R3-4]      // R4 = x
LD R5, [FP-4]      // R5 = z
LD R6, [FP+12]     // R6 = t
ADD R7, R2, R4    // R7 = y+x
ADD R8, R7, R5    // R8 = y+x+z
ADD R9, R8, R6    // R9 = y+x+z+t
ST R9, [R3-4]      // x = R9
```


4. In function `F1`, give the sequence of instructions associated with **`return(u+1)`**.

```
// @u = Env(F1)+16 (instead of 12, due to the return value of F1 !)

LD R1, [FP+16]     // R1 = u
ADD R2, R1, #1    // R2 = u+1
ST R2, [FP+8]      // return value for F1 ...
Epilogue
RET
```

5. In procedure **`main`**, give the sequence of instructions associated with `x=3+ F1(x)`.

```
// @x = Env(main)-4

LD R1, [FP-4]     // R1 = x
push(R1)          // push param x
push(FP)          // push static link of F1 (= Env(Main) )
ADD SP, SP, #4    // allocate space for F1 return value ...
CALL F1
LD R2, [SP]       // R2 = F1(x)
ADD SP, SP, #12   // clean the stack ...
ADD R3, R2, #3    // R3 = 3 + F1(x)
ST R3, [FP-4]     // x = R3
```

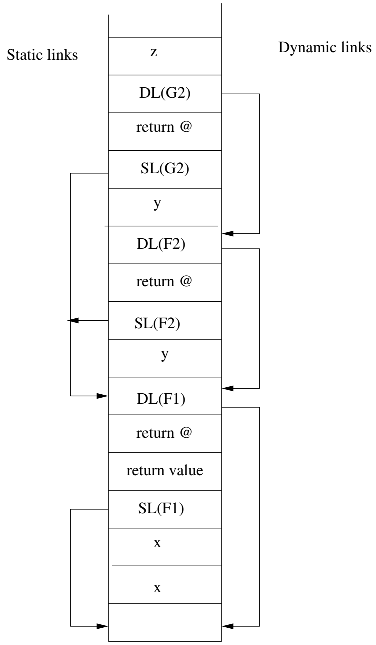

Figure 3: Stack layout when `G2` is executed