Homework (DM)
Compilation and Program Analysis (CAP)

Synchronisation and Branching

*Refer to the semantic course of CAP (chapter 03) for the semantics of **WHILE***

Instructions:

1. Every single answer must be informally explained AND formally proved.
2. Using LaTeX is NOT mandatory at all.
3. Vous avez le droit de rédiger en Français.

In this Homework, you will consider a parallel extension of the **WHILE** language seen in the course named **BSPWhile**, inspired by the real-world programming model BSP[1] (no prior knowledge on parallel programming or BSP is required for this homework). The main idea of BSP is to structure parallel programming with barriers which acts as synchronisation points for all parallel processes. A program is a set of processes, each executing the same non-terminating loop. A barrier pauses a process until all the other processes also reach a barrier, at which point all the processes progress together to the next statement.

The syntax and rules are similar to **WHILE**, as presented in the course, with some additions for parallelism. The grammar is shown in Figure 1. Expressions have been extended with a reserved variable *pid*, the process identifier, and *nbproc*, the number of processes. Statements now allow placing `barriers`. A program is a $\texttt{while}_{||}\ true\ \{\ S\ \}$ infinite parallel loop.

The semantics is shown in Figure 2. Each process is characterised by a statement and a process identifier (an integer). Consequently, $(S,1)||(S',2)$ is the parallel composition of two processes with identifiers 1 and 2. $\|_{i\in[1..n]}(S,i)$ is used to denote a finite set of such processes;

[1] https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

$||$ is associative and commutative, which allows us to write $(S,1)||(\|_{i\in[1..n]}(S,i))$ to extract one process from a parallel composition. The initial store σ of a parallel configuration maps *nbproc* to the number of parallel running processes, and all other variables to 0 ($\sigma_0(x) = 0$ for all variables x).

1 Dynamic Semantics

Question #1

Explain with a few sentences the behaviour of the example program shown in Figure 3. In particular, what is the value of `result` and `base` when each barrier is reached.

Question #2

In the semantics of **WHILE** we have rules with the signature $(s,\sigma) \Rightarrow \sigma$. Explain there are no toplevel rule with a similar signature in the hypothesis of semantics of parallel programs ($\Rightarrow_{||}$).

From now on and for the next examples, we suppose we can use arrays with statements of the form $A[i] := e$ and $x := A[j]$ and for statements of the form `for x = 1 to n { S }`

To add an input to our program, we suppose that a special variable M is initialised before running the program (and not set to 0.) The semantics of the for is that the `x` variable is not shared between the threads (x should be considered as `x[pid]`).

Question #3

Modify the example in Figure 3 so that after some point `result` contains the sum of the M first integers. The program cannot stop but the `result` variable should not be modified any more.

We consider the following application : given an input array A of size M filled with integers, we want to compute an array B such that $B[0] = A[0]$, $B[M-1] = A[M-1]$. and for all $i \in [1..M-2]$. $B[i] = (A[i-1] + A[i] + A[i+1])/3$.

We want to iterate this "average method" several times (i.e. set as A the previous array B computed and compute B again iteratively).

The computation will be performed by N processes such that $M = k * N$ for some k that is not known.

We suppose that $A[i]$ and M are initialised before running the program.

Question #4

Write the program described above, with a barrier between each computation of B. You can have more barriers if needed. Add a comment "`(* output B*)`" just after the `barrier` statement where a new computation of the average array B could be printed on the screen.

Bonus question: Note that it is important to do input/outputs at a `barrier` statement. Can you explain why in a few words?

Expressions:

$$
\begin{array}{rll}
e \in \mathcal{E} ::= & \texttt{true} \mid \texttt{false} \mid n & \text{Constants} \\
\mid & x & \text{Variable} \\
\mid & (b \parallel b) \mid (b \,\&\&\, b) & \text{Bool Op} \\
\mid & (e == e') & \text{Equality} \\
\mid & e + e & \text{Addition} \\
\mid & pid & \textbf{Process identifier} \\
\mid & nbproc & \textbf{Number of processes}
\end{array}
$$

Statements:

$$
\begin{array}{rll}
S \in Stm ::= & (x := e) & \text{Assign} \\
\mid & \texttt{skip} & \text{Do nothing} \\
\mid & S_1; S_2 & \text{Sequence} \\
\mid & \texttt{if } e \texttt{ \{ } S_1 \texttt{ \} else \{ } S_2 \texttt{ \}} & \text{Test} \\
\mid & \texttt{while } e \texttt{ \{ } S \texttt{ \}} & \text{Loop} \\
\mid & \texttt{barrier} & \text{Barrier}
\end{array}
$$

Programs:

$$
P \in Prog ::= \texttt{while}_{||}\ \texttt{true \{ } S \texttt{ \}} \quad \text{Main parallel loop}
$$

Figure 1: Grammar of the **WHILE** language

Evaluation of Expressions: $Val : \mathcal{E} \times State \to \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L}$

$$
\begin{array}{ll}
Val(n, \sigma) = value(n) & Val(x, \sigma) = \sigma(x) \\
Val(e_1 + e_2, \sigma) = Val(e_1, \sigma) + Val(e_2, \sigma) & \ldots
\end{array}
$$

Evaluation of Statements $(Stm, State) \Rightarrow (Stm, State) \mid State$

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e, \sigma)] \qquad (\texttt{skip}, \sigma) \Rightarrow \sigma \qquad \frac{(S_1, \sigma) \Rightarrow \sigma'}{((S_1; S_2), \sigma) \Rightarrow (S_2, \sigma')}$$

$$\frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \Rightarrow (S_1'; S_2, \sigma')} \qquad \frac{Val(b, \sigma) = \texttt{true}}{(\texttt{if } b \texttt{ \{ } S_1 \texttt{ \} else \{ } S_2 \texttt{ \}}, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = \texttt{false}}{(\texttt{if } b \texttt{ \{ } S_1 \texttt{ \} else \{ } S_2 \texttt{ \}}, \sigma) \Rightarrow (S_2, \sigma)} \qquad \frac{Val(b, \sigma) = \texttt{true}}{(\texttt{while } b \texttt{ \{ } S \texttt{ \}}, \sigma) \Rightarrow (S; \texttt{while } b \texttt{ \{ } S \texttt{ \}}, \sigma)}$$

$$\frac{Val(b, \sigma) = \texttt{false}}{(\texttt{while } b \texttt{ \{ } S \texttt{ \}}, \sigma) \Rightarrow (\texttt{skip}, \sigma)}$$

Evaluation of Programs and parallel statements $(\|_{i \in I}(Stm, n), State) \Rightarrow_{||} (\|_{i \in J}(Stm, n), State)$ where I and J are finite sets of indices.

$$\frac{(S, \sigma[pid \mapsto n]) \Rightarrow (S', \sigma')}{((S, n)||(\|_{i \in I}(S_i, n_i)), \sigma) \Rightarrow_{||} ((S', n)||(\|_{i \in I}(S_i, n_i)).\sigma')}$$

$$(\|_{i \in I}(\texttt{barrier}; S_i, n_i), \sigma) \Rightarrow_{||} (\|_{i \in I}(S_i, n_i), \sigma)$$

Given a program $P = \texttt{while}_{||}\ \texttt{true \{ } S \texttt{ \}}$ and a number n of processes to be spawned, an initial configuration is:

$$\left(\|_{i \in [1..n]}(\texttt{while true \{ } S \texttt{ \}}, i), \sigma_0[nbproc \mapsto n]\right)$$

Figure 2: Semantic of the **BSPWhile** language

```
1 while|| true {
2   result := result+base+pid
3   barrier;
4   if (pid==0) { base := base+nbproc } else { skip } ;
5   barrier;
6 }
```

Figure 3: Example of **BSPWhile** programs

1.1 Language extensions

Initialisation is difficult in the setting we defined, we now want programs to be of the form $P_{init} ::= S_{init}; \texttt{while}_{||}\ \texttt{true}\ \{\ S\ \}$. In this case, S_{init} should only be executed once by a single process (the one with $pid = 0$).

Question #5

Define a "translation semantics", i.e. a function $[\![P_{init}]\!]$ that takes as input a program P_{init} with an initialisation statement into a program P without an initialisation statement that behaves exactly the same. You should make sure the initialisation statement is executed only once and only by process with *pid* 0.

Explain informally why it works in one or two sentences.

We extend the language with a new statement called exit that terminates the current process:

$$\begin{array}{rll} S \in Stm ::= & \ldots & \text{previous statements} \\ & \mid exit & \text{terminates the current process} \end{array}$$

It has the following semantics:

$$((exit, n) \mid\mid (\|_{i \in I}(S_i, n_i)), \sigma) \Rightarrow_{||} ((\|_{i \in I}(S_i, n_i)), \sigma)$$

Question #6

Explain this semantic rule, what is the final configuration for a program that terminates correctly (i.e. a program that terminates and is not stuck on a barrier)?

Question #7

Modify the example in Figure 3 so that it computes the sum of M first integers and stops. You should probably start from the solution to Question 3.

2 Barrier analysis

2.1 An easy sub-language

Definition 1 (Blocked barrier) *We say that a runtime configuration C contains a blocked barrier if it is of the form*

$$C = ((\texttt{barrier}; S, n) \mid\mid (\|_{i \in I}(S_i, n_i)), \sigma)$$

and for all configurations C' such that $C \Rightarrow^*_{||} C'$, C' is not of the form

$$C' = \left((\|_{i \in J}(\texttt{barrier}; S'_i, n'_i)), \sigma\right)$$

In other words, a blocked barrier is a process that is stuck on a barrier statement and will never progress further because at least one of the other processes will never reach a barrier.

Question #8

Write a simple example of a program that systematically reaches a blocked barrier.

For now, we place ourself in a language without the following statements: `if .. then .. else`, `exit`, `while`. Programs still contain a single `while||` statement at the top level. Intermediary reduction steps might have `while` constructs (due to the semantics of `while||`).

Given an execution with only two processes, we consider the following theorem:

Theorem 1 (Reachable configurations for 2 processes) *All reachable configurations are of the form:*

$$(S_3; \texttt{while } \textit{true } \{\ S\ \} \mid\mid S_2; S_3; \texttt{while } \textit{true } \{\ S\ \}, \sigma) \text{ with } S = S_1; S_2; S_3$$

Or the symmetric case, with S_1, S_2, S_3 possibly empty.

Also if $S_2 = S'; \texttt{barrier}; S''$ then there is no barrier statement in S_3 and S_1.

We thus only consider one of these three cases.

Question #9

Prove formally Theorem 1.

Question #10

State the generalisation of Theorem 1 to n processes.

Question #11

Specialize the definition of blocked barrier to 2 processes. Use Theorem 1 to formally prove that there is no blocked barrier in our reduced language in the case of two processes.

Can this be generalized to n processes ?

2.2 Designing an Analysis

In order to handle additional statements, we design an analysis that checks whether barriers are well-placed in a program. Our objective is that if the program is validated by the analysis then no blocked barrier can appear at runtime. Our idea is to count the barriers and ensure that all execution paths will produce the same number of barriers.

In a first time we only consider conditional statements (if then else). We will discuss exit and while later. We want to enforce that each branch of the `if` has the same number of barriers, for this we define a judgement $\vdash S : n$ that states that: All execution paths of S contain the same number of branches, and this number is equal to n.

The analysis rule for the if statement is:

$$\frac{\vdash S : n \qquad \vdash S' : n}{\vdash \texttt{if}\ e\ \{\ S\ \}\ \texttt{else}\ \{\ S'\ \} : n}$$

Question #12

Define the analysis rules for the other statements of the language.

Question #13

State formally the correctness theorem that ensures that all analysed programs have no blocked barriers

Explain informally in 10 lines maximum 1) why this theorem is true; and 2) how you would prove it.

Question #14

Define an example program that is rejected by the analysis but has no blocked barrier at runtime.

Question #15 (Difficult)

Extend the analysis to deal with the exit statement (you should probably change slightly the signature of the judgement).

Explain how it works, in particular how your example provided in Question #7 is valid according to the analysis.

Question #16 (Difficult)

Explain informally in two sentences how you would deal with a `for i=1 to n` statement in this analysis.

Discuss in 2 sentences the difficulties and what could be done for a `while` statement. You should be able to identify a simple case where the analysis is possible.

3 Optimizing conditionals

We now want to send our **BSPWhile** program to a massively parallel execution platform **SUPERBSP**. This platform accepts **WHILE** programs, with the following additions around conditionals:

- Barrier instructions are accepted, but only at the toplevel.
- Assignments can be predicated by a boolean variable.

Let us consider the example below. The original **BSPWhile** program is shown on the left. To execute it on the **SUPERBSP** platform, we modify it to the program on the right. In this new program, we lifted the barrier to the toplevel, and used a predicated assignments for the assignments present before the barrier in each branches of the conditional. The predicated assignments, denoted `test ? a[pid] := 10` here, is only executed if `test` is `true`. It avoids introducing an additional `if` due to lifting the barrier to the toplevel. This is advantageous here, since such a if would be small.

```
while|| {
  if (b[pid] == 0) {
    a[pid] := 100;
    barrier;
    b[pid] := 1;
  } else {
    barrier;
    a[pid] :=
      (a[pid-1]+a[pid]+a[pid+1])/3
    b[pid] := 0;
  }
}
```

```
while|| {
  test = (b[pid] == 0)
  test ? a[pid] := 100;
  barrier;
  if (test) {
    b[pid] = 1
  } else {
    a[pid] =
      (a[pid-1]+a[pid]+a[pid+1])/3
    b[pid] = 0;
  }
}
```

In the rest of this section, we will transform our program to first lift the barriers to the toplevel, then optimise our program using predicated assignments. Unless indicated otherwise we consider only programs without `exit` and without `while`, but with conditionals and the top-level $\texttt{while}_{||}$.

Question #17

Let us now remark that if a conditional statement is well typed with a given integer n, i.e. $\vdash \texttt{if}\ e\ \{\ S\ \}\ \texttt{else}\ \{\ S'\ \} : n$, then S and S' must have a similar shape when it comes to barrier. More precisely.

$$\vdash \texttt{if}\ e\ \{\ S\ \}\ \texttt{else}\ \{\ S'\ \} : n \implies \exists S_1, S_2, S'_1, S'_2. \begin{vmatrix} S = S_1; \texttt{barrier}; S_2 & \vdash S_1 : 0 \\ S' = S'_1; \texttt{barrier}; S'_2 & \vdash S'_1 : 0 \\ \vdash \texttt{if}\ e\ \{\ S_2\ \}\ \texttt{else}\ \{\ S'_2\ \} : n-1 \end{vmatrix}$$

Give a formal proof.

Question #18

Use this result to design a procedure `lift` such that, for any well typed **BSPWhile** program p, `lift`(p) is a program that has no barriers inside conditional statements.

Explain and justify it informally.

We now want to optimize small ifs which might appear after lifting barriers by using predicated assignments. Predicated assignments are of the form *test* ? $x \leftarrow e$ where *test* is a boolean variable. They are similar to normal assignment, but are executed only if *test* is true.

Question #19

Design a procedure `ifconversion` such that, for any well typed **BSPWhile** program p without barriers, `ifconversion`(p) has no conditionals, and instead use predicated assignments. The produced program should of course behave similarly

Question #20

Assume that a predicated assignment is 10% more expensive than an normal statement to execute. When should predicated assignments be used ?

Write a heuristics that decide when to apply the `ifconversion` procedure.

Question #21 (Difficult)

`exit` is also treated specially by our platform **SUPERBSP**: `exits` can only be at toplevel, but can be predicated: `test ? exit`.

Update the `lift` and `ifconversion` procedure to lift the exits to toplevel with predicated exits. Justify informally their correction in this new context.

Question #22

We now consider the `while` case. Consider the following piece of code with arbitrary statements s_1, s_2, s_3. Present an equivalent program with barriers only at the toplevel.

```
1 while|| {
2   s1;
3   while true {
4     s2;
5     barrier;
6     s3;
7   }
8 }
```

Question #23 (Difficult)

Open Question

Propose extensions of `lift` and `ifconversion` to lift barriers outside `while true` statements in more general cases. What are the issues? Give an example, and informally propose appropriate restrictions.

3.1 The WHILE language

We present below the syntax of **WHILE** as seen in the course The syntax and rules are the same as present in the course, with some additions built-in operations for dynamic tests and lists.

The grammar is shown in Figure 4. Expressions have been extended with lists of integer including the empty list `[]`, the list with a single element `[`n`]`, and a concatenation operator. We also have operators `is_int`, `is_bool` and `is_list` to check the type of an expression.

The semantics is shown in Figure 5 and follows the one given in the course. No operator is available for mixed types: addition only works with two integers, concatenation $++$ with two lists and equality $==$ for any two expressions of the same type.

Expressions:

$e \in \mathcal{E}_w ::= \texttt{true} \mid \texttt{false} \mid n \mid \texttt{[]}$	Constants
$\mid \texttt{[}e\texttt{]}$	Singletons
$\mid x$	Variable
$\mid (b \mid\mid b) \mid (b \;\&\&\; b)$	Bool Op
$\mid (e == e')$	Equality
$\mid e + e$	Addition
$\mid \texttt{concat}(e, e')$	List concat
$\mid \texttt{is\_int} \mid \texttt{is\_bool} \mid \texttt{is\_list}$	Type checks

Statements:

$S \in Stm_w ::= (x := e)$	Assign
$\mid x, y := pop(e)$	Pop
$\mid skip$	Do nothing
$\mid S_1; S_2$	Sequence
$\mid \texttt{if}\ e\ \{\ S_1\ \}\ \texttt{else}\ \{\ S_2\ \}$	Test
$\mid \texttt{while}\ e\ \{\ S\ \}$	Loop
$\mid \texttt{Fail}$	Fail

Figure 4: Grammar of the **WHILE** language

Evaluation of Expressions: $Val : \mathcal{E} \times State \to \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L}$

$$Val(n, \sigma) = value(n) \qquad Val(\texttt{is\_int}(e), \sigma) = \texttt{true} \quad \text{if } Val(e, \sigma) \in \mathbb{Z}$$
$$Val(x, \sigma) = \sigma(x) \qquad Val(\texttt{is\_int}(e), \sigma) = \texttt{false} \quad \text{otherwise}$$
$$Val(e_1 + e_2, \sigma) = Val(e_1, \sigma) + Val(e_2, \sigma) \qquad Val(\texttt{is\_list}(e), \sigma) = \texttt{true} \quad \text{if } Val(e, \sigma) \in \mathbb{L}$$
$$Val(concat(e_1, e_2), \sigma) = Val(e_1, \sigma) ++ Val(e_2, \sigma) \qquad Val(\texttt{is\_list}(e), \sigma) = \texttt{false} \quad \text{otherwise}$$
$$\ldots \qquad \ldots$$

Evaluation of Statements $(Stm, State) \Rightarrow (Stm, State) \mid State \mid Error$

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e, \sigma)] \qquad (\texttt{skip}, \sigma) \Rightarrow \sigma \qquad (\texttt{Fail}, \sigma) \Rightarrow Error$$

$$\frac{Val(e, \sigma) = \texttt{[}v\texttt{]} ++ L}{(x, y := \texttt{pop}(e), \sigma) \Rightarrow \sigma[x \mapsto v][y \mapsto L]} \qquad \frac{(S_1, \sigma) \Rightarrow \sigma'}{((S_1; S_2), \sigma) \Rightarrow (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \Rightarrow (S_1'; S_2, \sigma')}$$

$$\frac{(S_1, \sigma) \Rightarrow Error}{((S_1; S_2), \sigma) \Rightarrow Error} \qquad \frac{Val(b, \sigma) = \texttt{true}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = \texttt{false}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_2, \sigma)}$$

$$(\texttt{while } b \texttt{ do } S \texttt{ done}, \sigma) \Rightarrow (\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S \texttt{ done}) \texttt{ else skip}, \sigma)$$

Figure 5: Semantic of the **WHILE** language