Problem 1: Parsing (25 pts)

Consider the following grammar which describes a fragment of the OCaml syntax:

\[
S ::= E$
\]

\[
E ::= \text{let } id = E \text{ in } E | E(id) | id # id | id
\]

where \( S \) and \( E \) are nonterminals and $, let, id, =, in, (, ), and # are all terminal symbols. Nonterminal \( S \) is the start symbol and $ is the right endmarker symbol.

a) [7 pts] This grammar is ambiguous, and therefore is not LR(1). Show that the grammar is ambiguous.

b) [7 pts] The following grammar is LR(1) and accepts the same language.

\[
S ::= E$
\]

\[
E ::= \text{let } id = E \text{ in } E | F
\]

\[
F ::= F(id) | id # id | id
\]

Now consider the LR(0) automaton for this grammar. Show the initial state of this automaton and the successor state where the automaton transitions to on nonterminal symbol \( F \). Use the written states to argue that the grammar is not LR(0).

c) [4 pts] The grammar from part b) is not LL(1) either. Give two reasons why the grammar cannot be used to build an LL(1) parser.

d) [7 pts] Write an LL(1) grammar which accepts the same language.

Problem 2: Loop transformations (25 pts)

Answer the following questions with respect to a uniprocessor machine with a two level memory hierarchy consisting of an L1 cache and main memory.

a) [6 pts] Define the terms “spatial locality” and “temporal locality” in the context of programs executing on this machine, and explain briefly why programs might exhibit spatial locality and temporal locality.

Loop permutation (or loop interchange) is an important transformation for enhancing locality in programs. This transformation changes the nesting order of loops in a perfect loop nest, leaving the loop body unchanged.

b) [7 pts] Consider the following pseudo-code for multiplying \( N \times N \) matrices.

\[
\begin{align*}
\text{for } i &= 1, N \\
\text{for } j &= 1, N \\
\text{for } k &= 1, N \\
C(i,j) &= C(i,j) + A(i,k) \times B(k,j)
\end{align*}
\]

It can be shown that all six possible permutations of this loop nest compute exactly the same results. Consider the permutation \( i-j-k \), shown above, and \( i-k-j \). For each of these permutations and for each of the array accesses in the loop body, indicate what kind of locality the access exhibits: temporal, spatial, both, or none. Assume that the matrices are stored in row-major format.
c) [6 pts] In certain cases, the compiler must adjust the loop bounds to account for the change in the nesting order. Consider the following loop nest:

\[
\begin{align*}
\text{for } i & = 1, N \\
\text{for } j & = 1, i \\
\text{sum} & = \text{sum} + A(i,j)
\end{align*}
\]

This loop nest adds up the elements in matrix \( A \) that are below the diagonal. Write down the result of applying loop permutation to this loop nest.

d) [6 pts] The loop permutation transformation is not always legal. Write down a loop nest for which loop permutation is illegal.

---

**Problem 3: Modules (25 pts)**

a) [5 pts] Explain what is a module and name a language that has a module system.

b) [5 pts] Why is it useful to programmers to have a module system in a programming language? How do interfaces (signatures) aid in this? Explain in 2-3 sentences.

Now, consider the following simple module that can be found in a programming language. The program consists of a set of modules and interfaces, where modules must agree with their interfaces of the same name. In the following grammar, modules are described by nonterminal `module`:

\[
\begin{align*}
\text{module} & ::= \text{module } id = \text{begin } decl_1 \ldots decl_n \text{ end} \\
\text{decl} & ::= \text{fundecl} \mid \text{valdecl} \\
\text{fundecl} & ::= \text{fun } id(args) = expr \\
\text{valdecl} & ::= \text{val } id = expr \\
\text{expr} & ::= \ldots \mid expr.id
\end{align*}
\]

The idea here is that the `module` form declares a module, which may contain two kinds of declarations: function declarations introduced by the keyword `fun` and simple value declarations introduced by `val`. The syntactic class `id` represents an identifier, and `args` gives some syntax for function argument declarations. A function declaration `fundecl` contains an expression that defines the body of the function; a value declaration `valdecl` contains an expression that is to be evaluated to obtain the appropriate value to bind to the given identifier. Expressions `expr` include, in addition to some set of ordinary expressions unrelated to modules, an expression of the form `e.x` where `e` is an expression that evaluates to a module and `x` is the name of a module component.

c) [5 pts] Suggest a reasonable syntax for a corresponding *typed* interface declaration. Give it as a context-free grammar similar in form to that above.

d) [5 pts] Adding a module form to a language creates a new way to cause non-termination unless we’re careful, even if modules are not first-class values. Sketch an example of a program that creates the problem, and describe two reasonable ways we might restrict the initialization expression of a `valdecl` to solve this problem.

e) [5 pts] Suppose we want to compile modules separately and that our target is a binary object code format. What information will the object file likely need to contain to ensure that separately compiled modules will combine to make a program that would have been accepted if given to the compiler as a single compilation unit?
Problem 4: Semantics (25 pts)

This problem explores the semantics of multitasking and synchronization constructs in Ada. We start with a core multitasking language which consists of the following syntactic sets:

Tasks: \( t ::= T(c) \)

Commands: \( c ::= \text{skip} \mid X := e \mid c_0; c_1 \mid t_1 || t_2 \)

Expressions: \( e ::= n \mid X \mid c_0 + c_1 \)

where \( T \) ranges over task names, \( X \) over variable names, and \( n \) over integer numbers. Each task \( t = T(c) \) has a task name \( T \) and a task body \( c \). The command \( t_1 || t_2 \) executes the bodies of tasks \( t_1 \) and \( t_2 \) in parallel; its execution finishes when both tasks complete. A program in this language is a task \( T \).

The small-step operational semantics for this language includes the following two evaluation relations:

\[
\langle c, \sigma \rangle \rightarrow_C \langle c', \sigma' \rangle \quad \text{for commands and} \quad \langle t, \sigma \rangle \rightarrow_T \langle t', \sigma' \rangle \quad \text{for tasks.}
\]

A state \( \langle c, \sigma \rangle \) maps variables to their values. The evaluation rules for assignments and sequences are standard, and we omit them. To simplify the evaluation rules of parallel tasks, we introduce an equivalence relation \( \equiv \) which makes parallel tasks commutative and associative:

\[
t_1 || t_2 \equiv t_2 || t_1 \quad \text{and} \quad (t_1 || t_2) || t_3 \equiv t_1 || (t_2 || t_3).
\]

We can use this equivalence to rearrange the structure of parallel threads at any point during the execution of a program. The evaluation of parallel commands and tasks uses the following rules:

\[
\frac{\langle c, \sigma \rangle \rightarrow_C \langle c', \sigma' \rangle}{\langle T(c), \sigma \rangle \rightarrow_T \langle T(c'), \sigma' \rangle} \quad \frac{\langle t_1, \sigma \rangle \rightarrow_T \langle t'_1, \sigma' \rangle}{\langle t_1 || t_2, \sigma \rangle \rightarrow_C \langle t'_1 || t_2, \sigma' \rangle}
\]

The execution of a program terminates when it reaches a final configuration \( \langle T(\text{skip}), \sigma \rangle \).

a) [6 pts] Note that we don’t need an additional rule to reduce the task on the right side of a parallel command because parallel tasks commute. However, we need one more rule to describe how the execution proceeds when parallel tasks complete. Write this remaining rule.

Tasks in Ada can synchronize with each other using the rendezvous mechanism, which is based on the following accept and call commands:

\[
d ::= \ldots \mid \text{accept} (P,c) \mid T.P
\]

where \( P \) ranges over a set of names representing entry points. Whenever a task \( T_1 \) executes an “accept \((P,c)\)” command, it blocks until some other task \( T_2 \) calls this entry using “\( T_1.P \)”. Similarly, when task \( T_2 \) executes “\( T_1.P \)”, it blocks until the execution of \( T_1 \) reaches an “accept \((P,c)\)” command. When the two tasks have reached the corresponding accept and call points, \( T_1 \) starts executing command \( c \). When \( c \) completes, \( T_1 \) proceeds with the command after the accept and \( T_2 \) proceeds with the command after the call. Here is an example of a command which uses a rendezvous:

\[
T_1( \text{accept}(P, Y := X+1) ) || T_2( X := 1; T_1.P; X := Y )
\]

After the execution of this parallel command, both \( X \) and \( Y \) have value 2.

b) [7 pts] To model the semantics of rendezvous constructs, we extend the program configurations with a rendezvous configuration of the form \[ T_1 || T_2 \] which captures the fact that tasks \( t_1 \) and \( t_2 \) currently synchronize at a rendezvous. One of the rules that introduces such a rendezvous configuration is:

\[
\langle T_1(\text{accept} (P,c_{11}); c_{12}) || T_2(T_1.P; c_2), \sigma \rangle \rightarrow_C \langle [T_1(\text{accept} (P,c_{11}); c_{12}), T_2(T_1.P; c_2)], \sigma \rangle.
\]

Write the rules for evaluating the rendezvous \[ [T_1(\text{accept} (P,c_{11}); c_{12}), T_2(T_1.P; c_2)] \].

c) [5 pts] There are many situations where the execution of a program can run into errors, i.e., cannot make progress using our semantic rules. Write three different examples of stuck configurations which don’t contain rendezvous configurations \[ T_1 || T_2 \] or sequences of commands \( c_1; c_2 \).
d) [7 pts] Ada uses a type system which includes *task types*: the type of a task is a set of entry points in that task. Programmers must declare task types before using tasks in the program.

We want to model the task typechecking process in our language. Consider that a type environment $\Gamma$ is a set of pairs $(T, P)$ where $P$ is an entry point of task $T$. We want to derive a typing judgment of the form $T; \Gamma \vdash c : \Gamma'$, where $T$ is the current task, $\Gamma$ contains the declared task types (i.e., the declared entry points), and $\Gamma'$ contains the entry points that explicitly occur in accept commands in $c$. The typing rule for command sequences is:

$$
\frac{T, \Gamma \vdash c_1 : \Gamma_1 \quad T, \Gamma \vdash c_2 : \Gamma_2}{T, \Gamma \vdash c_1 ; c_2 : \Gamma_1 \cup \Gamma_2}
$$

The goal is to guarantee that all accept and call commands use valid entry points, and that all the declared entry points actually occur in the code. A program $t = T(c)$ typechecks if: $T, \Gamma \vdash c : \Gamma$.

Write the typing rules for parallel commands, accept, and call commands.