Programming Paradigms
—Final Exam—

Department of Computer Science
University of Stuttgart

Winter Semester 2024/2025, March 26, 2025

Name, first name:______________________________________

Matriculation number:____________________

Seat number:____________________

General Guidelines and Information

1. You have 60 minutes and there are 60 points. Use the number of points as guidance on how much time to spend on a question.
2. For multiple choice questions, you get the indicated number of points if your answer is correct, and zero points otherwise (i.e., no negative points for incorrect answers).
3. You should write your answers directly on the test. Use a ballpoint pen or similar, do not use a pencil. Use the space provided (if you need more space your answer is probably too long).
4. Do not provide multiple solutions to a question.
5. Your answers can be given either in English or in German.

To be filled out by the correctors:

Part	Points	Score
1	4	
2	10	
3	12	
4	12	
5	6	
6	10	
7	6	
Total	60	

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)
   - [ ] Liskov's Substitutability Principle states that a subtype should invoke supertype methods only when absolutely necessary.
   - [ ] Liskov's Substitutability Principle states that a subtype should behave like the supertype when being used through the supertype's interface.
   - [ ] Liskov's Substitutability Principle states that a supertype's constructor must always be invoked before the subtype's constructor.
   - [ ] Liskov's Substitutability Principle states that subclasses may reduce the visibility of inherited members.
   - [ ] Liskov's Substitutability Principle states that a subclass may delete methods of its superclass.

2. Which of the following statements is true? (Only one statement is true.)
   - [ ] Coroutines always execute concurrently with at least one other coroutine.
   - [ ] Coroutines are a form of preemptive multi-tasking.
   - [ ] Coroutines are a form of non-preemptive multi-tasking.
   - [ ] Coroutines are a form of cooperative single-tasking.
   - [ ] Coroutines always resume execution at the same position.

3. Which of the following statements is true? (Only one statement is true.)
   - [ ] In a case/switch statement in C++, the default arm is always executed.
   - [ ] Case/switch statement in C++ are a form of logically controlled loops.
   - [ ] In a case/switch statement in C++, the arms of multiple case may execute unless stopped by a break statement.
   - [ ] In a case/switch statement in C++, the compiler ensures that at least one case will always match.
   - [ ] Case/switch statements in C++ are a form of enumeration-controlled loops.

4. Which of the following statements is true? (Only one statement is true.)
   - [ ] In the Scheme language, names are bound to values with the `be` keyword.
   - [ ] In the Scheme language, programs and lists share the same syntax.
   - [ ] In the Scheme language, evaluating a lambda expression yields a conditional.
   - [ ] In the Scheme language, every function must be annotated with argument types and a return type.
   - [ ] In the Scheme language, recursive calls are the only way to express repeated computations.

Part 2 [10 points]

The following is about a DFA-based, switch statement-style scanner. Suppose the tokens of the programming language can be described by this regular expression:

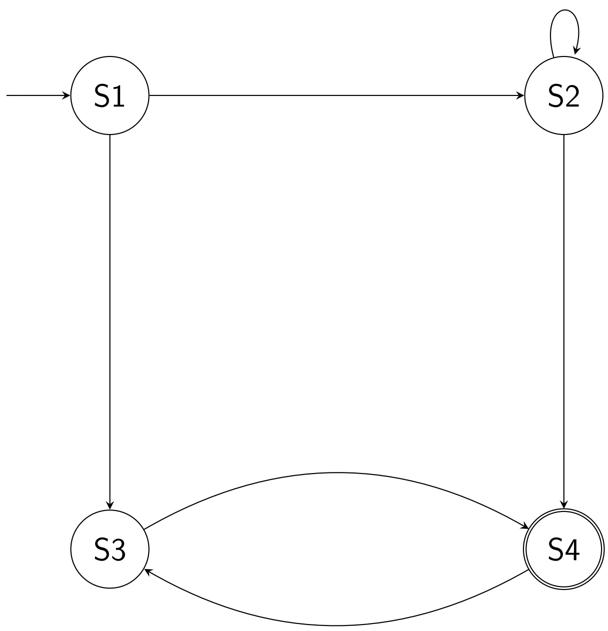
$$y(yz)^*y \mid xx^*(y \mid yz(yz)^*y)$$

1. Give two examples of legal tokens of this language and two examples of illegal tokens.

   - Legal tokens:

   - Illegal tokens:

2. Provide a deterministic finite automaton (DFA) that describes all legal tokens. Use the following template to provide your solution:



3. The following Java code as a switch statement-style implementation of the scanner. Fill in the gaps with the appropriate code so that the recognizeNextToken method takes a non-empty string and returns the next token as defined in the language, or "ERROR" when no token can be recognized. There are ten gaps in total, each marked with __________.

```
String recognizeNextToken(String input) {
    int state = __________;
    String token = "";
    char nextChar = input.charAt(token.length());
    while (true) {
        switch (__________) {
        case 1:
            switch (nextChar) {
            case __________:
                state = 2;
                break;
            case ’y’:
                state = __________;
                break;
            default:
                return "ERROR";
            }
            break;
        case 2:
            switch (nextChar) {
            case ’x’:
                state = 2;
                break;
            case ’y’:
                state = __________;
                break;
            default:
                return "ERROR";
            }
            break;
        case 3:
            switch (__________) {
            case ’y’:
                state = 4;
                __________
            default:
                return "ERROR";
            }
            break;
        case 4:
            switch (nextChar) {
            case ’z’:
                state = 3;
                break;
            case ’ ’:
                return __________;
            default:
                return __________;
            }
            break;
        }
        token = token + __________;
        nextChar = token.length() == input.length() ? ’ ’ : input.charAt(token.length());
    }
}
```

Part 3 [12 points]

This task is about computing the parse table for an LL(1) parser. Suppose the following grammar of a simple programming language:

(1) $\textbf{Function} \rightarrow fun\ id\ (\ \textbf{Args}\ )\ begin\ \textbf{Stmts}\ end$
(2) $\textbf{Args} \rightarrow \epsilon$
(3) $\textbf{Args} \rightarrow id\ \textbf{MoreArgs}$
(4) $\textbf{MoreArgs} \rightarrow \epsilon$
(5) $\textbf{MoreArgs} \rightarrow\ ,\ \textbf{Args}$
(6) $\textbf{Stmts} \rightarrow \epsilon$
(7) $\textbf{Stmts} \rightarrow \textbf{Stmt}\ ;\ \textbf{Stmts}$
(8) $\textbf{Stmt} \rightarrow \textbf{Decl}$
(9) $\textbf{Stmt} \rightarrow \textbf{Call}$
(10) $\textbf{Decl} \rightarrow var\ id$
(11) $\textbf{Call} \rightarrow id\ (\ \textbf{Args}\ )$

The non-terminals of the language, printed in bold font, are:
Function, **Args**, **MoreArgs**, **Stmts**, **Stmt**, **Decl**, **Call**

The terminals of the language are:
fun, *id*, (,), *begin*, *end*, ,, ;, *var*

1. Compute the FIRST and FOLLOW sets of all non-terminals. Provide your solution in the following table:

Non-terminal	FIRST	FOLLOW
Function		
Args		
MoreArgs		
Stmts		
Stmt		
Decl		
Call		

2. Compute the PREDICT set of each grammar rule. Provide your solution in the following table:

Rule	PREDICT
(1)	
(2)	
(3)	
(4)	
(5)	
(6)	
(7)	
(8)	
(9)	
(10)	
(11)	

3. Compute a parse table suitable for a table-based, predictive parser. Provide your solution in the following table:

Non-terminal	Terminal								
	fun	id	(	)	$begin$	end	,	;	var
Function									
Args									
MoreArgs									
Stmts									
Stmt									
Decl									
Call									

Part 4 [12 points]

The following questions are about composite types and pointers in C.
Hints about the first argument given to the `printf` function:

- `%d` means to format the additional argument as a decimal integer
- `%f` means to format the additional argument as a decimal floating pointer number
- `%td` means to format the additional argument as an integer that gives the difference between two pointers

1. Consider the following C code.

```
struct my_struct
{
    int n;
    float f;
    struct my_struct *p;
};

struct my_struct s1 = {100, 3.14, NULL};
struct my_struct s2 = {200, 6.28, NULL};
s1.p = &s2;
```

   (a) What does the following statement print?
   `printf("%f\n", s1.f);`

   (b) What does the following statement print?
   `printf("%p\n", s1.p);`

   (c) What does the following statement print?
   `printf("%f\n", s2.p);`

   (d) Fill in the blank so that the following statement prints `200`.

   `printf("%d\n", __________);?`

2. Consider the following C code.

```
union
{
    int n;  // assume: 4 bytes
    char f[4];
} my_union;
my_union.n = 5;
```

(a) What does the following statement print?

```
printf("%d\n", my_union.n);
```

(b) What the following code print? Explain your answer.

```
for (int i = 0; i < 4; i++)
{
    printf("%d ", my_union.f[i]);
}
```

(c) Suppose we now execute the following code.

```
my_union.f[2] = 7;
printf("%d\n", my_union.n);
```

Explain what the code prints. (No need to give the exact value, but explain how the value is computed.)

3. Consider the following C code.

```
int my_array[4] = { 1, 2, 3, 4 };
int *my_ptr1 = my_array;
int *my_ptr2 = &my_array[3];
```

(a) What does the following statement print?

```
printf("%d\n", 2[my_array]);
```

(b) What does the following statement print?

```
printf("%d\n", *my_ptr1);
```

(c) What does the following statement print?

```
printf("%d\n", *my_ptr2);
```

(d) What does the following statement print?

```
printf("%td\n", my_ptr2 - my_ptr1);
```

Part 5 [6 points]

This task is about static and dynamic method binding in a class-based language.
Consider the following source code, which is written in a toy language with Java-inspired syntax.
Note that the semantics of the language is *not* as in Java, but as given below.

```
class Foo {
  virtual void m() {}
  void n() {}
}

class Bar extends Foo {
  void m() {}
  void n() {}
  void o() {}
}

Foo x = new Foo();
Bar y = new Bar();
Foo z = y;

x.m();  // Call 1
x.n();  // Call 2
y.m();  // Call 3
y.n();  // Call 4
y.o();  // Call 5
z.m();  // Call 6
z.n();  // Call 7
z.o();  // Call 8
```

Suppose three different semantics for method binding:

1. *Static*. The language uses static method binding.
2. *Dynamic*. The language uses dynamic method binding.
3. *Static except for "virtual"*. The language uses static method binding, except when the overridden method is declared with `virtual`, in which case dynamic method binding is used for calls to this method and any methods that override it.

For each of the eight calls at the end of the program, indicate the method that gets called under a specific semantics. For example, if method `m` of class `Foo` gets called, then write `Foo.m`. If the language does not allow a specific call, please indicate "error". Use the following table to provide your solution:

Call	Static	Dynamic	Static except for “virtual”
Call 1			
Call 2			
Call 3			
Call 4			
Call 5			
Call 6			
Call 7			
Call 8			

Part 6 [10 points]

The following is about C code that uses OpenMP. Some hints about OpenMP:

- `#pragma omp parallel for` creates a parallel loop.
- `#pragma omp parallel` creates a parallel region.
- `omp_get_thread_num()` returns the thread identifier of the currently executing thread.
- `#pragma omp barrier` creates a barrier.
- `#pragma omp critical` creates a critical section.

1. Consider the following code:

```
int x = 42;
int y = 0;
#pragma omp parallel for private(y)
for (int i = 0; i < 3; i++) {
    x = omp_get_thread_num();
    y = x + 4;
}
printf("x=%d, y=%d\n", x, y);
```

   (a) What are the possible outputs of the code?

2. Consider an extended version of the above code:

```
int x = 42;
int y = 0;
int z = 8;
#pragma omp parallel for private(y)
for (int i = 0; i < 3; i++) {
    x = omp_get_thread_num();
    y = x + 4;
    z = y;
}
printf("x=%d, y=%d, z=%d\n", x, y, z);
```

   (a) What are the possible outputs of the code?

   (b) How to adapt the pragma in the code so that it the printed output will contain `y=23`? Give the updated pragma and explain why it works.

3. In the following, assume that the maximum number of parallel threads is two, i.e., `OMP_NUM_THREADS=2`. Consider the following code:

```
1      int x = 3;
2      #pragma omp parallel
3      {
4      #pragma omp critical
5          {
6              x = omp_get_thread_num();
7              x += 2;
8              printf("x=%d\n", x);
9          }
10     #pragma omp barrier
11         x = 0;
12
13     }
14     printf("x=%d\n", x);
```

(a) What are the possible outputs of the code?

(b) Assume that the pragma at line 4 gets removed. Describe a behavior that is possible with the modified code but that cannot occur if the pragma is present.

Part 7 [6 points]

1. What do the following terms in the lambda calculus evaluate to?

   (a) $(\lambda y\ .\ if\ iszero\ y\ then\ succ\ 0\ else\ 0)\ 0$

   (b) $(\lambda x\ .\ \lambda y\ .\ succ\ y)\ false$

   (c) $(\lambda a\ .\ a\ true)\ (\lambda b\ .\ if\ true\ then\ false\ else\ true)$

2. Now, consider the typed lambda calculus. What are the types of the following lambda terms? In case the given term is not type-correct, explain why.

   (a) $\lambda a : Bool\ .\ false$

   (b) $(\lambda b : Bool\ .\ if\ b\ then\ true\ else\ false)\ (\lambda x\ .\ x)$

   (c) $(\lambda c : Bool \rightarrow Bool\ .\ c\ false)\ (\lambda c : Bool\ .\ true)$