Programming Paradigms

—Final Exam—

Department of Computer Science
University of Stuttgart

Winter semester 2019/20, May 16, 2020

Name, first name:_____

Matriculation 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	12	
3	9	
4	10	
5	8	
6	9	
7	8	
Total	60	

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)

   - ☐ A scanner checks if a given grammar describes a context-free language.
   - ☐ A scanner ensures that all identifier names are unambiguous.
   - ☐ A scanner ensures that all tokens in a given program have the same type.
   - ☐ A scanner transforms a sequence of characters into a sequence of tokens.
   - ☐ A scanner transforms a sequence of tokens into a parse tree.

2. Which of the following statements is true? (Only one statement is true.)

   - ☐ Static and dynamic binding yield the same association of names to entities.
   - ☐ The lifetime of an object must be equal to the lifetime of all its bindings.
   - ☐ A binding associates a token with its type.
   - ☐ Most programming languages have a single scope only.
   - ☐ The scope of a binding is the textual region where the binding is active.

3. Which of the following statements is true? (Only one statement is true.)

   - ☐ The fields of a record are usually stored in locations spread randomly across the heap.
   - ☐ The fields of a record must all have the same size.
   - ☐ A record stores related data of the same type together.
   - ☐ A record stores related data of heterogeneous types together.
   - ☐ The purpose of packing is to ensure that all fields in a record have the same type.

4. Which of the following statements is true? (Only one statement is true.)

   - ☐ To prevent a data race between two concurrent memory access, it suffices that one of the access is protected by a lock.
   - ☐ Two concurrent reads of the same memory location may cause a data race.
   - ☐ A data race may occur only if the order of memory accesses is deterministic.
   - ☐ When multiple concurrent threads are each waiting for a lock held by another thread, the system has a data race.
   - ☐ Two concurrent writes to the same memory location may cause a data race.

Part 2 [12 points]

Consider the following grammar, where "baa" is a terminal, "*Goal*" and "*SheepNoise*" are non-terminals, and "*Goal*" is the start symbol.

⟨*Goal*⟩	::=	⟨*SheepNoise*⟩	(rule 1)
⟨*SheepNoise*⟩	::=	baa ⟨*SheepNoise*⟩	(rule 2)
	|	baa	(rule 3)

Your task is to parse the input following using an LR(1) parser: baa baa EOF

The following are the tables required for LR(1) parsing:

Action table:

State	EOF	baa
0		shift 2
1	accept	
2	reduce 3	shift 2
3	reduce 2	

Goto table:

State	*SheepNoise*
0	1
1	
2	3
3	

1. Provide the steps taken by the LR(1) parsing algorithm. For each step, indicate the current stack, the currently remaining input, and the action taken based on the stack and input. Use the following table as a template for your answer.

Stack	Remaining input	Action
EOF, 0	baa, baa, EOF	shift 2

2. Draw the parse tree created by the parser.

Part 3 [9 points]

Recall the following table of precedence levels in C (copied from the lecture slides):

Operator	Meaning
`++, --`	Post-increment, post-decrement
`++, --` `*`	Pre-increment, pre-decrement Pointer dereference
`<, >`	Inequality test
`==, !=`	Equality test
`&&`	Logical and
`||`	Logical or
`=, +=`	Assignment

For each of the three C expressions listed in the following:

- Add parenthesis around *every* expression and subexpression, so that the evaluation order is consistent with the behavior specified by the precedence rules of C.
- Rewrite the expression so that all binary operations use prefix notation. Use a Lisp-like syntax, i.e., where each subexpression is surrounded by parentheses.
- Rewrite the expression so that all binary operations use postfix notation. Use a Lisp-like syntax, i.e., where each subexpression is surrounded by parentheses.

Expression 1: `x > y != z`

- Same expression with parenthesis:

- Same expression with prefix notation for binary operations:

- Same expression with postfix notation for binary operations:

Expression 2: `arr[j++]-- > val--`

- Same expression with parenthesis:

- Same expression with prefix notation for binary operations:

- Same expression with postfix notation for binary operations:

Expression 3: `get_flag() && closed || nb_clients > *ctr--`

- Same expression with parenthesis:

- Same expression with prefix notation for binary operations:

- Same expression with postfix notation for binary operations:

Part 4 [10 points]

Recall the toy language for typed expressions introduced in the lecture. For your reference, the syntax and type rules are reproduced here.

Syntax:

$$
\begin{array}{rcl}
\langle t \rangle & ::= & \text{true} \\
& | & \text{false} \\
& | & \text{if } \langle t \rangle \text{ then } \langle t \rangle \text{ else } \langle t \rangle \\
& | & 0 \\
& | & \text{succ } \langle t \rangle \\
& | & \text{pred } \langle t \rangle \\
& | & \text{iszero } \langle t \rangle
\end{array}
$$

Type rules for *Bool*:

$$\frac{}{\text{true} : Bool}\ \text{T-True}$$

$$\frac{}{\text{false} : Bool}\ \text{T-False}$$

$$\frac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}\ \text{T-If}$$

Type rules for *Nat*:

$$\frac{}{0 : Nat}\ \text{T-Zero}$$

$$\frac{t_1 : Nat}{\text{succ } t_1 : Nat}\ \text{T-Succ}$$

$$\frac{t_1 : Nat}{\text{pred } t_1 : Nat}\ \text{T-Pred}$$

$$\frac{t_1 : Nat}{\text{iszero } t_1 : Bool}\ \text{T-IsZero}$$

1. Consider the following typed expression: if iszero (succ 0) then true else iszero 0

   (a) Is this expression type-correct?

   (b) If yes, provide the type derivation tree, including the names of the rules you apply. If no, then explain why not.

2. Consider the following typed expression: pred (iszero false)

   (a) Is this expression type-correct?

   (b) If yes, provide the type derivation tree, including the names of the rules you apply. If no, then explain why not.

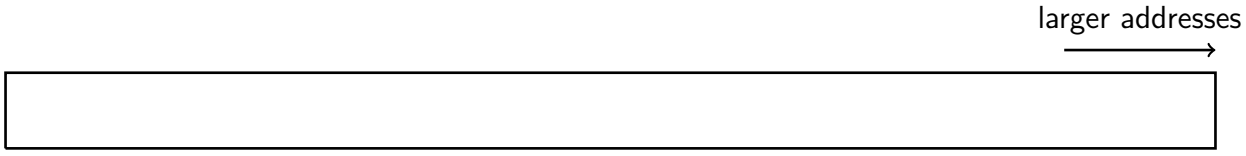
Part 5 [8 points]

Consider the following array declaration in C:

```
1 #define M 3
2 #define N 2
3 int matrix[M][N] = {{1,2},{3,4},{5,6}};
```

(Note: `#define` is a macro that is used here to define the dimensionality of the array.)

1. How is the array represented in memory under the assumption that the language uses contiguous, column-major layout? Use the following template to draw the memory layout, including the values in the array and a pointer to indicate where the array starts.



2. Provide two logically controlled, nested loops in C that traverse the array and sum up all elements. Again under the assumption that the language uses contiguous, column-major layout, ensure that the loops traverse the array efficiently.

3. Explain why you have chosen the control flow structure above.

Part 6 [9 points]

This part is based on a toy language with Python-inspired syntax. It has the following features:

- The `coroutine` keyword defines a coroutine.
- Calling a coroutine `c` with `c()` works like a regular function call that returns once the routine has ended.
- Calling a coroutine `c` with `co c()` creates and immediately returns a coroutine object to which control can be later transferred.
- A `yield c_ref` statement passes control to the coroutine object `c_ref`.
- The `my_ref` keyword is a reference to the current subroutine object.

Consider the following program written in our toy language:

```
1  a_ref = b_ref = c_ref = e_ref = None
2
3  coroutine main():
4    a()
5
6  coroutine a():
7    b_ref = co b()
8    c()
9
10 coroutine b():
11   d()
12
13 coroutine c():
14   c_ref = my_ref
15   yield b_ref
16   e_ref = co e()
17   yield e_ref
18
19 coroutine d():
20   yield c_ref
21   // other stmts
22
23 coroutine e():
24   f()
25
26 coroutine f():
27   // here
28
29 main()
```

Draw the stack of the program's execution when it reaches line 27. Mark the stack frame where control is when reaching this point.

Part 7 [8 points]

Consider the following Scheme program:

```
1  (define (p)
2    (p)
3  )
4
5  (define (test x y)
6    (if (= x 0)
7      0
8      y
9    )
10 )
11
12 (test 0 (p))
```

1. Provide a step-by-step evaluation of the expression at line 12 under *applicative-order* evaluation. Use the following template to provide your solution.

   ```
   (test 0 (p))
   ```

   ⇒ _____

   ⇒ _____

   ⇒ _____

   ⇒ _____ (Show at most four evaluation steps.)

2. Provide a step-by-step evaluation of the expression at line 12 under *normal-order* evaluation. Use the following template to provide your solution.

   ```
   (test 0 (p))
   ```

   ⇒ _____

   ⇒ _____

   ⇒ _____

   ⇒ _____ (Show at most four evaluation steps.)

3. Do both evaluation orders yield the same result? Explain your answer.