Final Exam

EDAP05: Concepts of Programming Languages, HT 2021

2022-01-15

Anonymisation code: ____________________

PLEASE READ THE FOLLOWING CAREFULLY

This final exam consists of 12 questions. You can reach a total of 100 points. If you get 50 points (including your bonus points from the homework assignments), you will pass the exam.

Make sure that the final exam consists of precisely 19 numbered pages. Write down your anonmyisation code on each page. Use a black or blue pen.

Only supply one answer per question. Strike out incorrect answers.

If you run out of space, continue writing on the back of the page. Additional sheets are available.

The following utilities are permitted:

- Paper and writing material
- Calculators that are not capable of wireless connectivity (should not be necessary)
- One sheet of A4 paper with hand-written notes (possibly on both sides)

Make sure to read all questions carefully before starting on your answer!

Good luck!

Hints: Solution hints are listed in shaded boxes. These hints are usually not full solutions (you will have to explain/justify). There are also often alternative solutions that give full credit.

Question:	1	2	3	4	5	6	7	8	9	10	11	12	**Sum**
Max Points:	8	6	3	9	8	7	9	11	11	7	15	6	100
Points Reached:													

MYSTERY grammar (for reference):

⟨*Program*⟩	⟶	⟨*Block*⟩
	|	⟨*Block*⟩ ‘`;`’
⟨*Decls*⟩	⟶	⟨*DeclList*⟩ | ε
⟨*DeclList*⟩	⟶	⟨*Decl*⟩
	|	⟨*Decl*⟩ ‘`;`’ ⟨*DeclList*⟩
⟨*Decl*⟩	⟶	‘`VAR`’ *id* ⟨*OptType*⟩
	|	‘`TYPE`’ *id* ‘=’ ⟨*Type*⟩
	|	⟨*ProcDecl*⟩
⟨*OptType*⟩	⟶	ε | ‘`:`’ ⟨*Type*⟩
⟨*ProcDecl*⟩	⟶	‘`PROC`’ *id* ‘`(`’ ⟨*Formals*⟩ ‘`)`’ ⟨*OptType*⟩ ‘=’ ⟨*Block*⟩
	|	‘`PROC`’ *id* ‘`(`’ ⟨*Formals*⟩ ‘`)`’ ‘=’ ⟨*Block*⟩
⟨*Formals*⟩	⟶	⟨*FormalList*⟩ | ε
⟨*FormalList*⟩	⟶	⟨*Formal*⟩
	|	⟨*FormalList*⟩ ‘`,`’ ⟨*Formal*⟩
⟨*Formal*⟩	⟶	*id* ‘`:`’ ⟨*Type*⟩
⟨*Type*⟩	⟶	‘`INT`’
	|	⟨*SubrTy*⟩
	|	⟨*ArrayTy*⟩
	|	*id*
	|	⟨*ProcTy*⟩
⟨*SubrTy*⟩	⟶	‘`[`’ *number* ‘`TO`’ *number* ‘`]`’
⟨*ArrayTy*⟩	⟶	‘`ARRAY`’ ⟨*SubrTy*⟩ ‘`OF`’ ⟨*Type*⟩
⟨*ProcTy*⟩	⟶	‘`PROC`’ ‘`(`’ ⟨*Formals*⟩ ‘`)`’ ⟨*OptType*⟩
⟨*Block*⟩	⟶	⟨*Decls*⟩ ‘`BEGIN`’ ⟨*Stmts*⟩ ‘`END`’
⟨*Stmts*⟩	⟶	⟨*StmtList*⟩ | ε
⟨*StmtList*⟩	⟶	⟨*Stmt*⟩
	|	⟨*StmtList*⟩ ‘`;`’ ⟨*Stmt*⟩
⟨*Stmt*⟩	⟶	⟨*Assignment*⟩
	|	⟨*Return*⟩
	|	⟨*Block*⟩
	|	⟨*Conditional*⟩
	|	⟨*Iteration*⟩
	|	⟨*Output*⟩
	|	⟨*Expr*⟩
⟨*Assignment*⟩	⟶	⟨*Expr*⟩ ‘`:=`’ ⟨*Expr*⟩
⟨*Return*⟩	⟶	‘`RETURN`’ ⟨*Expr*⟩
⟨*Conditional*⟩	⟶	‘`IF`’ ⟨*Expr*⟩ ‘`THEN`’ ⟨*StmtList*⟩ ‘`ELSE`’ ⟨*StmtList*⟩ ‘`END`’
⟨*Iteration*⟩	⟶	‘`WHILE`’ ⟨*Expr*⟩‘`DO`’ ⟨*StmtList*⟩ ‘`END`’
⟨*Output*⟩	⟶	‘`PRINT`’ ⟨*Expr*⟩
⟨*Expr*⟩	⟶	⟨*Operand*⟩
	|	⟨*Expr*⟩ ⟨*Operator*⟩ ⟨*Operand*⟩
⟨*Operand*⟩	⟶	*number*
	|	*id*
	|	⟨*Operand*⟩ ‘`[`’ ⟨*Expr*⟩ ‘`]`’
	|	⟨*Operand*⟩ ‘`(`’ ⟨*Actuals*⟩ ‘`)`’
	|	‘`(`’ ⟨*Expr*⟩ ‘`)`’
⟨*Operator*⟩	⟶	‘+’ | ‘>’ | ‘==’ | ‘`AND`’
⟨*Actuals*⟩	⟶	⟨*ActualList*⟩ | ε
⟨*ActualList*⟩	⟶	⟨*Expr*⟩
	|	⟨*Actuals*⟩ ‘`,`’ ⟨*Expr*⟩

Anonymkod:

Question 1 (8 Points)

In the table below you see pairs of types with a box ☐ in between. Write an X in the box if neither type is a subtype of the other, or draw a $<:$ or $:>$ (suitably) to indicate that one is a subtype of the other.

Use the same assumptions as in class, i.e., that (1) we are using an imperative language (updates are allowed), that (2) the type system enforces strong typing and (3) the type system permits any type to be a subtype of another if and only if doing so will not require dynamic checks.

(a) (4 Points) Fill in as indicated above:

`[5 TO 8]`	$<:$	`[0 TO 8]`
`[5 TO 8]`	$:>$	`[5 TO 7]`
`[5 TO 8]`	X	`[3 TO 7]`
`ARRAY [0 TO 10] OF [0 TO 10]`	$<:$	`ARRAY [5 TO 10] OF [0 TO 10]`
`ARRAY [0 TO 10] OF [5 TO 10]`	X	`ARRAY [0 TO 10] OF [0 TO 10]`

(b) (4 Points) Continue filling in. For the following, assume that `A` is a *supertype* of `B`, and that the type `C[X]` is *contravariant* in type parameter `X`.

`A`	$:>$	`B`
`C[A]`	$<:$	`C[B]`
`A` $\rightarrow$ `A`	X	`B` $\rightarrow$ `B`
`A` $\rightarrow$ `A`	$:>$	`A` $\rightarrow$ `B`
`A` $\rightarrow$ `A`	$<:$	`B` $\rightarrow$ `A`
`A` $\rightarrow$ `B`	$<:$	`B` $\rightarrow$ `A`

Anonymkod:

Question 2 (6 Points)

Consider the following MYSTERY program.

```
1 PROC g(v : INT) =
2   VAR z : INT;
3   PROC R() =
4     BEGIN
5       z := 0        // Assignment-A
6     END;
7   PROC f(i : INT): INT =
8     VAR z : INT
9     BEGIN
10      R();
11      z := z + i;   // Assignment-B
12      RETURN z
13    END
14  BEGIN
15    RETURN f(v);
16  END
17 BEGIN
18   PRINT g(0);
19 END
```

(a) (4 Points) Assuming static or dynamic scoping, which `z` (line 2 or line 8) do Assignment-A and Assignment-B update? Fill in.

Scoping	Assignment	Line
Static	`Assignment-A`	2
Static	`Assignment-B`	8
Dynamic	`Assignment-A`	8
Dynamic	`Assignment-B`	8

(b) (2 Points) Assuming static scoping, what are the scopes of `i` and of the `z` in line 8? Give line numbers or mark in the code above.

Hints:

i, z:7–13 or 7–12

Question 3 (3 Points)

Consider the following MYSTERY program. Assume that all `INT` variables are initialised to `0` by default.

```
1  PROC h(i : INT): INT =
2    VAR y : INT
3    BEGIN
4      y := y + i;
5      RETURN y
6    END
7  BEGIN
8    PRINT h(0);
9    PRINT h(1);
10   PRINT h(2)
11 END
```

When run, the program prints the following:

```
0
1
3
```

(a) (3 Points) What storage binding does `y` (line 2) use? Explain.

> **Hints:** No reason to assume heap-dynamic binding. Stack-dynamic would reset `y` to `0`. The answer should explain how static binding would justify the result we see here.

Question 4 (9 Points)

You are using a programming language that has a built-in array type, `array<T>`, where `T` can be any type in the language. You now want to write a subroutine `concat` that concatenates two arrays. Below is a snippet in Java syntax that illustrates the idea:

```
// in Java: assume a static method for the subroutine
static SomeType_0 concat(SomeType_1 array1, SomeType_2 array2) {
    SomeType_3 mergedArray =
        new SomeType_4[array1.length + array2.length];

    // Copy array1 into mergedArray
    ...
    // Copy array2 into mergedArray
    ...

    return mergedArray;
}
```

Assume that we want the type of `concat` to be as general as possible, and that the language is strongly statically typed. In the following, you can use Java, Scala, SML, Rust, or C++ syntax, or explain informally. If you cannot express what you want to express in the syntax that you picked, use English.

(a) (2 Points) Assume that the language supports parametric polymorphism, but not subtype polymorphism. Specify the most general type for `concat` that you can.

Hints: (`array<T>`, `array<T>`) → `array<T>`

(b) (2 Points) Consider the type that you gave in (a). In a language like Java with subtype polymorphism, are there any uses of `concat` that would be type-safe from the perspective of strong typing but that are forbidden by your type? Explain or give an example.

Hints: Example: `concat` with parameters `array<Integer>`, `array<String>` could yield `array<Object>`, but we don't allow that here.

(c) (4 Points) Assume that the language supports subtype polymorphism and bounded parametric polymorphism. Specify the most general type for `concat` that you can.

Hints: $\forall \alpha, \beta, \gamma. \alpha :> \beta, \alpha :> \gamma. (\texttt{array}<\beta>, \texttt{array}<\gamma>) \to \texttt{array}<\alpha>$
(Adequately translated into one of the languages)

(d) (1 Point) Are there any uses of `concat` in this language that would be type-safe from the perspective of strong typing but that are forbidden by your type? Explain or give an example.

Hints: `concat` involving empty lists, here the element type is irrelevant.

Question 5 (8 Points)

What is the relation between variance and parameter passing modes, if any?

Consider the following MYSTERY program:

```
PROC g(f : PROC(x : [0 TO 10]): INT) : INT = //...
```

Here, `f : [1 TO 10] → INT`, and `f` takes a parameter `x : [1 TO 10]`.

In the following sub-questions, *explain why the alternative(s) do not preserve strong typing*. You can give an example. Remember that you can reference your answers to other sub-questions to simplify your answer.

(a) (2 Points) Assume that we pass `x` via *pass-by-value*. What is the most general variance that we can give to the type of `x`, relative to the type of `f`? Explain.

(b) (3 Points) (**Synthesis**) Assume that we pass `x` via *pass-by-result*. What is the most general variance that we can give to the type of `x`, relative to the type of `f`? Explain.

Hints: Treat like a return value.

(c) (3 Points) (**Synthesis**) Assume that we pass `x` via *pass-by-name*. What is the most general variance that we can give to the type of `x`, relative to the type of `f`? Explain.

Hints: Cf. (a) and (possibly) (b), the latter if we assume that we can use `x` as an lvalue (as in MYSTERY and ALGOL).

Anonymkod:

Question 6 (7 Points)

(a) (4 Points) Which of the following is part of the syntax, static semantics, and dynamic semantics? Add check-marks as appropriate.

	Syntax	Static Semantics	Dynamic Semantics
Type Inference		X	
Type Checking		X	X
Operator associativity	X		
Closures anything but Syntax			

(b) (3 Points) (**Synthesis**) Assume that you are given a language that is implemented in a pure interpreter (i.e., not a hybrid implementation). Is it possible that the language uses static typing? Explain your reasoning.

Hints: Yes. Static type checking does not require compilation. Implementing a static type checker requires much of the implementation effort of building a compiler in practice, though.

Question 7 (9 Points)

We are adding a new type to Java. This type, `valset<T>`, is polymorphic over `T`, and represents sets that contain `T` values as elements.

Values of type `valset<T>` cannot be modified: any operation on them instead creates a new value, analogously to integers.

Assume that in the Java syntax, $\langle expr\rangle$ is the Java non-terminal for arbitrary expressions. We extend $\langle expr\rangle$ as follows:

$$
\begin{array}{lcl}
\langle expr\rangle & \longrightarrow & \text{`}\texttt{[[}\text{'}\ \langle exprlist\rangle \text{`}\texttt{]]}\text{'} \\
\langle exprlist\rangle & \longrightarrow & \langle expr\rangle \\
 & | & \langle expr\rangle\ \text{`}\texttt{,}\text{'}\ \langle exprlist\rangle
\end{array}
$$

We also overload the operator +. Let `s1`, `s2 : valset<T>` and `e : T`, then:

a. `s1 + e` evaluates to a set that contains the elements of `s1` and the element `e`.

b. `s1 + s2` evaluates to the union of the two sets `s1` and `s2`.

For brevity, we omit other useful operators that such a set should have.

We define the semantics of set constructions with the following natural semantics rules:

$$
\frac{\texttt{[[}\ \ell\ \texttt{]]} \Downarrow v_\ell \quad e \Downarrow v_e}{\texttt{[[}\ e\texttt{,}\ \ell\ \texttt{]]} \Downarrow v_\ell \cup \{v_e\}}\ (\textit{literal-singleton}) \qquad \frac{e \Downarrow v_e}{\texttt{[[}\ e\ \texttt{]]} \Downarrow \{v_e\}}\ (\textit{literal-multi})
$$

To define the semantics of the overloaded + operator, we have the choice between two options:

Option A:

$$
\frac{\texttt{[[}\ \ell\ \texttt{]]} \Downarrow v_\ell \quad e \neq \texttt{[[}\ \ell'\ \texttt{]]} \text{ for any } \ell' \quad e \Downarrow v_e}{\texttt{[[}\ \ell\ \texttt{]]}\ \texttt{+}\ e \Downarrow v_\ell \cup \{v_e\}}\ (\textit{add-element-A})
$$

$$
\frac{\ell_1 \Downarrow v_{\ell_1} \quad \ell_2 \Downarrow v_{\ell_2}}{\texttt{[[}\ \ell_1\ \texttt{]]}\ \texttt{+}\ \texttt{[[}\ \ell_2\ \texttt{]]} \Downarrow v_{\ell_1} \cup v_{\ell_2}}\ (\textit{union-A})
$$

Option B:

$$
\frac{s : \texttt{valset<T>} \quad s \Downarrow v_s \quad e : \texttt{T} \quad e \Downarrow v_e}{s\ \texttt{+}\ e \Downarrow v_s \cup \{v_e\}}\ (\textit{add-element-B})
$$

$$
\frac{s_1 : \texttt{valset<T>} \quad s_2 : \texttt{valset<T>} \quad s_1 \Downarrow v_{s_1} \quad s_2 \Downarrow v_{s_2}}{s_1\ \texttt{+}\ s_2 \Downarrow v_{s_1} \cup v_{s_2}}\ (\textit{union-B})
$$

The following questions explicitly provide any Java-specific knowledge you need.

(Questions on the next page.)

(a) (6 Points) Do the natural semantics of **Option A** and **Option B** result in different behaviour in practice? Give an example to support your claim.

Hints: Option A operates purely syntactically, so it cannot correctly handle e.g. a variable of valset type. An answer should highlight this with an example.

(b) (3 Points) (**Synthesis**) Assume that we add boxing/unboxing coercions to Java that can box `valset<T>` into the object type `ValueSet<T>` and unbox accordingly. Thus, e : `valset<T>` whenever e : `ValueSet<T>`.

Review the two semantic options **Option A** and **Option B**. Which of them, if any, are affected by the boxing/unboxing coercions?

Hint: Java uses subtyping, and all object types have `Object` as a supertype.

Since all valsets are now objects, Option B will now be ambiguous for `valset<Object>`.

Question 8 (11 Points)

We define the language L1 via the non-terminal ⟨*expr*⟩ in the following grammar:

$$
\begin{aligned}
\langle expr \rangle &\longrightarrow \langle con \rangle \\
&\quad\mid\ \text{`X'} \\
&\quad\mid\ \langle con \rangle\ \text{`@'}\ \langle expr \rangle \\
\langle con \rangle &\longrightarrow \langle lit \rangle \\
&\quad\mid\ \langle lit \rangle\ \text{`+'}\ \langle con \rangle \\
&\quad\mid\ \text{`?'}\ \langle expr \rangle \langle proc \rangle \\
\langle proc \rangle &\longrightarrow \text{`A'} \langle con \rangle \\
&\quad\mid\ \text{`STOP'} \\
\langle lit \rangle &\longrightarrow nat
\end{aligned}
$$

where *nat* describes the natural numbers (ℕ).

(a) (3 Points) For each of the following token sequences, check-mark whether they are productions of the L1 grammar:

	Yes	**No**
`? 1 + 2 STOP @ X`		
`1 @ 2 @ 3 + X`		
`? 1 A X @ 2`		

(b) (4 Points) L1 has two binary operators, '@' and '+'. What is the *associativity* of '+'? Explain by giving a parse tree.

right-associative

(c) (4 Points) L1 has two binary operators, '@' and '+'. What is the *precedence* of '+'? Explain by giving a parse tree.

+ has higher precedence than @

Question 9 (11 Points)

Consider the following custom-defined SML datatype, which describes a list that can contain both `int` and `string` elements:

```
datatype polylist = END
                  | INT of int * polylist
                  | STR of string * polylist
```

With this type, we can e.g. list the values `1, "a", 2, "b"` in order. We would write this list `pl0` as follows:

```
val pl0 = INT(1, STR("a", INT(2, STR("b", END))))
```

(a) (5 Points) Write an SML function `rmstr : polylist → polylist` that removes all STR elements from a `polylist`, unless that element is the very last element in the list. For example, `rmstr(pl0)` should yield `INT(1, INT(2, STR("b", END)))`.

Hints: Example:

```
fun rmstr (STR(s, END)) = STR(s, END)
  | rmstr (INT(i, tl))  = INT(i, rmstr(tl))
  | rmstr (STR(i, tl))  = rmstr(tl)
  | rmstr END           = END
```

Anonymkod:


```
(* repeated from the previous page *)
datatype polylist = END
                  | INT of int * polylist
                  | STR of string * polylist
```

(b) (1 Point) Given our definition of `polylist`, what is the type of `INT`?

(c) (5 Points) (**Synthesis**) Consider the following SML function, `org`:

```
val org : polylist -> polylist =
  let fun proc (intl, strl) =
      let fun mkint (i) (ir) = intl(INT (i, ir))
          fun mkstr (s) (sr) = strl(STR (s, sr))
          fun sub (END)         = intl(strl(END))
            | sub (INT (i, r)) = proc (mkint i, strl) (r)
            | sub (STR (s, r)) = proc (intl, mkstr s) (r)
      in sub
      end
    fun id (x:polylist) = x
  in proc (id, id)
  end
```

SML uses type inference to find the most general type for each function and variable. Using your knowledge of SML, determine the types of the following functions and variables:

`id`	: `polylist` → `polylist`
`intl`	: `polylist` → `polylist`
`strl`	: `polylist` → `polylist`
`mkint`	: `int` → `polylist` → `polylist`
`mkstr`	: `string` → `polylist` → `polylist`
`sub`	: `polylist` → `polylist`
`proc`	: (`polylist` → `polylist` × `polylist` → `polylist`) → `polylist` → `polylist`

(d) (**Optional**: 3 points (bonus)) What does the function `org` from (b) compute?

Hints: Shuffle all INT entries before all STR entries. (Uses continuation passing style.)

Anonymkod:

Question 10 (7 Points)

(a) Specify a *generic abstract datatype* for arrays in a language of your choice, with operations at least for:

a. reading array element
b. updating array elements
c. determining the array length

The abstract datatype does not have to match the array operations already provided by the language that you have selected. Specify which language you used.

You may use language features that your language of choice does not provide, if you explain them, and you may deviate from the language syntax as long as your meaning is clear.

Hints: Typeclass or interface definition (or abstract class / C++ purely virtual class) that takes one type parameter for the element type.

(b) In what sense is your specification an abstract datatype?

Hints: No implementation, but specifies contracts.

(c) In what sense is your specification a *generic* abstract datatype?

Hints: Parametric over element types.

Question 11 (15 Points)

Explain the differences between various concepts that we have discussed throughout the course.

(a) (3 Points) What are the differences between pointers and references?

Hints: pointer arithmetic ⇒ no strong typing

(b) (4 Points) What is the difference between reference equality vs structural equality? Explain with an example.

Anonymkod:

(c) (4 Points) What is the difference between an enumeration type and an algebraic datatype? Explain with an example.

Hints: algebraic datatypes strictly subsume enums. Example `polylist`: The `INT` and `STR` constructors are not expressible in enums.

(d) (4 Points) What is the difference between overloading and overriding? Explain with one example for each.

Anonymkod:

Question 12 (6 Points)

(Synthesis)

Consider the following Mystery program:

```
  PROC P(a : INT, b : INT) : INT =
  BEGIN
    IF a
    THEN RETURN b
    ELSE RETURN a
    END
  END;
  PROC Q(x : INT): INT =
  BEGIN
    PRINT x;
    RETURN x
  END
BEGIN
  P(Q(1) AND Q(0), Q(0) AND Q(1))
END
```

Upon running the program, you observe the following output:

```
0
0
```

Assume static scoping, stack-dynamic storage binding, and that we are not using pass-by-result.

(a) Given this output, what can we say about the semantics of the **AND** operator? Explain your answer.

Hints: short-circuit right-to-left.

(b) Given this output, what can we say about subroutine parameter evaluation order? Explain your answer.

Hints: Answer should point out that “a” is evaluated more than once.

Anonymkod: