CS 591L1: Emb. Langs. & Frmwrks. **Name:** _____
Time Limit: 80 Minutes **BU ID:** _____

Instructions:

This exam contains 9 pages (including this cover page) and 4 problems totalling 100 points.

You have 1 hour and 20 minutes to finish your answers.

Answer the questions in the level of details described. Try to be as concise as possible.

If you do not know the answer to a question, write "I don't know", and you will receive some complimentary grades!

Please fill your answers in the allocated space, you can use the back of the sheet as draft. Anything written outside the allocated space will not be graded.

Grade Table (for teacher use only)

Question	Points	Score
1	20	
2	25	
3	35	
4	20	
Total:	100	

1. (20 points) **BNF Notation and Parsing** Write the syntax definition of the following simple expressions language using BNF.

   The language consists of the following building blocks:

   1. integers.
   2. variable names, all lower case letters.
   3. basic arithmetic operation: +, -, * and / with the usual order of precedence.
   4. parenthesis: they have the highest order of precedence.

Solution: The BNF grammer is as follows:

⟨*integers*⟩ ::= [0-9]+

⟨*variables*⟩ ::= [a-b]+

// start with highest order of precedence going down

⟨*exp1*⟩ ::= ⟨*integers*⟩ | ⟨*variables*⟩ | '`(`' ⟨*expression*⟩ '`)`'

⟨*exp2*⟩ ::= ⟨*exp1*⟩ | ⟨*exp1*⟩ '`*`' ⟨*expression*⟩ | ⟨*exp1*⟩ '`/`' ⟨*expression*⟩

⟨*exp3*⟩ ::= ⟨*exp2*⟩ | ⟨*exp2*⟩ '`+`' ⟨*expression*⟩ | ⟨*exp2*⟩ '`-`' ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*exp3*⟩

2. (25 points) **Type Systems**

Consider a language containing variables, expressions, function definitions, and function calls. Our focus will be on function definitions.

Assume the syntax for function definition is as follows:

function <name> *(* <parameterName> *:* <type>, ... *)* {
<body statements>
return <expression>
}

We will refer to a definition like that as:

def(name, param1, type1, ..., body, returnExpression)

In our language, we will assign such a function the type:

(type1, ...) → typeof(returnExpression)

In other words, the type of the function indicates the type of all the parameters first, then the type of the return value, seperated by an arrow. For example:

f1 = def(add, x, int, y, int, [ ], x + y)

corresponds to:

function add(x: int, y: int) {
return x + y
}

(a) (5 points) According to the above, what do you think the type of the above function f1 should be?

Solution: $(int,\ int) \rightarrow int$

(b) (12 points) Consider the following typing rules on the next page, that attempt to formalize what we described above.

1. Rule 1 assigns a type to a variable x according to the environment/context.
2. Rule 2 assigns types to function definitions according to its return and parameter types.
3. Rule 3 allows typing of two consecutive statements, by typing the first statement, and using its type to assign a type to the second statement.
4. Rule 4 assigns types to return expressions.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (Rule 1)}$$

$$\frac{\{p1 : T1, ...\} \vdash (body; \text{return } retExp) : R}{\Gamma \vdash \text{def(name, p1, T1, ..., body, retExp)} : \ (T1, ...) \to R} \text{ (Rule 2)}$$

$$\frac{st1 := (v = e) \qquad \Gamma \vdash e : T1 \qquad \Gamma \cup \{v : T1\} \vdash st2 : T2}{\Gamma \vdash (st1; st2) : T2} \text{ (Rule 3)}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{return } e : T} \text{ (Rule 4)}$$

To the best of your understanding, answer the following in English:

- Why does Rule 2 use an environment Γ in its conclusion, but does not use it in its premise, and instead uses a new environment, containing only the parameters and their types?
- How does Rule 2 restrict the functions we can define beyond what popular programming languages do?

Solution: The use of a new environment indicates that the scope within the body of a function is different from other scopes outside of it.

In particular, only the parameters of a function are initially visible within its body at the very beginning.

This allows programmers to use the same name for different variables (with different types) inside different functions without ambiguity.

This is a bit more restrictive than popular languages, since our type system excludes the use of variables inside a function if they are defined outside of it (e.g. global variables).

(c) (8 points) Apply the rules above, as well as the following rule, to show that the type of the function **f1** indeed has the type you answered in part (a).

$$\frac{\Gamma \vdash e1 : int \qquad \vdash e2 : int}{\Gamma \vdash e1 + e2 : int} \text{ (Rule 5)}$$

Hint: You do not need to use Rule 3, because the body of the function is just a single statement.

Hint: Use Rule 1 twice, first to type x and y. Followed by Rule 5, Rule 4, and finally Rule 2.

Hint: Make sure that the environment you use in Rule 1 is the same as the premise of Rule 2, so that all the uses hook up together.

Solution: Let $\bar{\Gamma} := \{x : int,\ y : int\}$

Using rule 1 with $\bar{\Gamma}$ twice we get:

$$\frac{x : int \in \bar{\Gamma}}{\bar{\Gamma} \vdash x : int} \text{ (Step 1)}$$

$$\frac{y : int \in \bar{\Gamma}}{\bar{\Gamma} \vdash y : int} \text{ (Step 2)}$$

Using rule 5 on steps 1 and 2 we get:

$$\frac{\bar{\Gamma} \vdash x : int \qquad \bar{\Gamma} \vdash y : int}{\bar{\Gamma} \vdash x + y : int} \text{ (Step 3)}$$

Using rule 4 on step 3 we get:

$$\frac{\bar{\Gamma} \vdash x + y : int}{\bar{\Gamma} \vdash \text{return } x + y : int} \text{ (Step 4)}$$

Finally, we can use rule 2 to get what we want.

$$\frac{\bar{\Gamma} \vdash (\text{return } x + y) : int}{Any\ \Gamma \vdash \text{def(add, x, int, y, int, [ ], x + y)} :\ (int,\ int) \to int} \text{ (Step 5)}$$

Note: $\bar{\Gamma}$ is setup to be exactly what Rule 5 required it to be. The final conclusion applies for any environment Γ, since the type of our function does not depend on the context in which it is defined, because global variables are disallowed.

3. (35 points) **Programming Paradigms & Program Analysis**

   In this problem, we will try to validate that a recursive program terminates.

   To simplify the problem, we will assume that all the functions we will consider have a single parameter, which is always a whole non-negative number. The ideas from this problem can be generalized to apply to multi-parameter functions with richer types.

   One way to check that a recursive function terminates, is to check that the recursive calls it makes are all applied to strictly decreasing parameters.

   In our context, this means that all the recursive calls a function makes are passed a number **strictly less** than the initial given argument to that function. Since the parameter is non-negative, this means that eventually it will reach 0 and the recursion will stop.

   Your task is to provide a high level description (i.e. in English) of a solution design that performs this validation dynamically (i.e. via both instrumentation and aspect-oriented programming) and statically, and contrast their trade-offs.

   (a) (5 points) Assume we only allow a recursive function to make one recursive call to itself (e.g. Fibonacci), and no calls to other functions. Explain, in a paragraph, how a solution based on instrumentation can implement the above check and produce an error whenever it fails when we suspect that the function may not terminate.

   Solution: We parse the body of the function (say using *ast* in Python).

   Since the code may change the value of variables and parameters, we modify the function to store an untouched copy of its parameter at the very beginning.

   We find the recursive call the function makes to itself. We insert an if condition right before that function call that checks that the argument to that call is less than the unmodified copy of its parameter, and raise an error if it is not.

   (b) (5 points) Assume we allow the function to make several recursive calls to itself (e.g. Merge sort). Explain, in a paragraph, how your previous solution can be extended to handle this case.

   Solution: The solution remains mostly the same, however we now have to insert several if statements, one before every recursive call.

   Each if statement checks that the argument passed to its associated recursive call is smaller than the original parameter, and raises an error if it is not.

(c) (10 points) Explain, in a paragraph, how you can implement an alternative solution using Aspect-oriented programming.

Solution: We apply an aspect to the recursive function. Every call to the function (including the initial call, and all following recursive calls) will trigger our aspect. Our aspect function will maintain a **global** list that is initially empty.

Whenever the aspect function is triggered (because of some function call), it checks that the current argument is less than the last element in that list and raises an error if it is not.

The aspect function then adds the argument to the end of the list, and proceeds with the underlying function call regularly. This ensures that future recursive calls will be compared against this call's argument.

When the function call returns, the last element in the list is removed. This ensures that this call's argument is no longer considered after the call is done.

(d) (10 points) How can you implement an alternative solution statically, i.e. via parsing the code of the recursive function and analyzing it, without running it? Are there any additional challenges in this case? Contrast this with the previous dynamic approach: which is more accurate? Which do you think is easier to implement? Which has less overhead when executing the recursive function?

Solution: We parse the function code (say using *ast* in Python), and look for all recursive calls it makes. We then identify the arguments passed to these recursive calls.

We can try to check if these arguments are less than the initial parameter. Since we are doing this statically, we do not know the exact values of these arguments. We can try instead to check if it is smaller symbolically (e.g. if the recursive argument is equal to the initial argument $- 1$).

This is difficult to implement because it requires to symbolically compare variables statically. It is also less accurate because such a comparison is impossible to do in general. However, it introduces no overhead when running the code, since it is all static, as opposed to the previous solutions.

(e) (5 points) Determining whether a general purpose program, or Turing Machine, halts or not is undecidable (the famous halting problem). General recursive functions are as powerful as Turing Machines. Does any of the solution ideas above mean that we somehow solved the halting problem (and broke maths)? Why?

Solution: No. It is impossible to solve an undecidable problem with 100% accuracy on all cases.

Our solutions above are all conservative and inaccurate. They accept only a subset of terminating programs (the most obvious ones), and will reject many terminating programs that may increase the argument temporarily.

In other words, if our solution accepts some recursive function, then it must be terminating. But if it threw an error, that does not mean that the function was really not terminating, only that it is suspicious.

This is similar to how the null-safe check from project 2 works, and is a common idea frequently used to produce useful (but imperfect) tools for tasks where perfect solutions are impossible.

4. (20 points) **Embedding Categories**

   You are given examples of embedded sub-languages and frameworks from the real world. Your task is to **identify** which type of embedding was used to implement the given example, **justify** why you think that is the type of embedding, and **give** a *single advantage* of using this type of embedding in the example context as opposed to another form of embedding.

   (a) (10 points) The promise API in JavaScript is a very popular way to implement asynchronous and event driven code. A promise has two states: pending or resolved. A pending promise indicates a pending asynchronous task or a pending event. A resolved promise indicates that the associated event has occurred. A promise is a *custom JavaScript object* with several methods, including a "*.then*" method that takes a callback function, executes it when the promise is resolved, and returns a new promise. The promise API provides constructs to create a promise and determine how and when it gets resolved.

   Solution: This is a **deep embedding**, because the constructs of the promise API are **datatypes** (in particular *Promise* objects) in JavaScript.

   This type of embedding offers many advantages. Because promises are objects, they can be passed to or returned from functions. They can be chained easily and in arbitrary ways. Further more, they can be defined in one place, used several other places, and resolved in yet another place in the code.

   (b) (10 points) HTML and JavaScript are closely intertwined when developing web applications. In particular, the HTML document is automatically parsed by the interpreter into a Document Object Model (DOM) object that is accessible via JavaScript. JavaScript code can query the DOM to retrieve information from the HTML document, or can modify the DOM to insert information or change the styling of the HTML document. Similarly, JavaScript code is embedded in HTML documents via script tags.

   Solution: HTML is a separate language external to JavaScript. However, it is automatically embedded into JavaScript as a DOM datatype by the browser. Therefore, this is an **external embedding**.

   The main advantage of using this embedding is friendliness. Developers can use JavaScript to interact with the webpage interface (by manipulating the DOM), while being able to define the interface using the friendly and popular syntax of HTML.