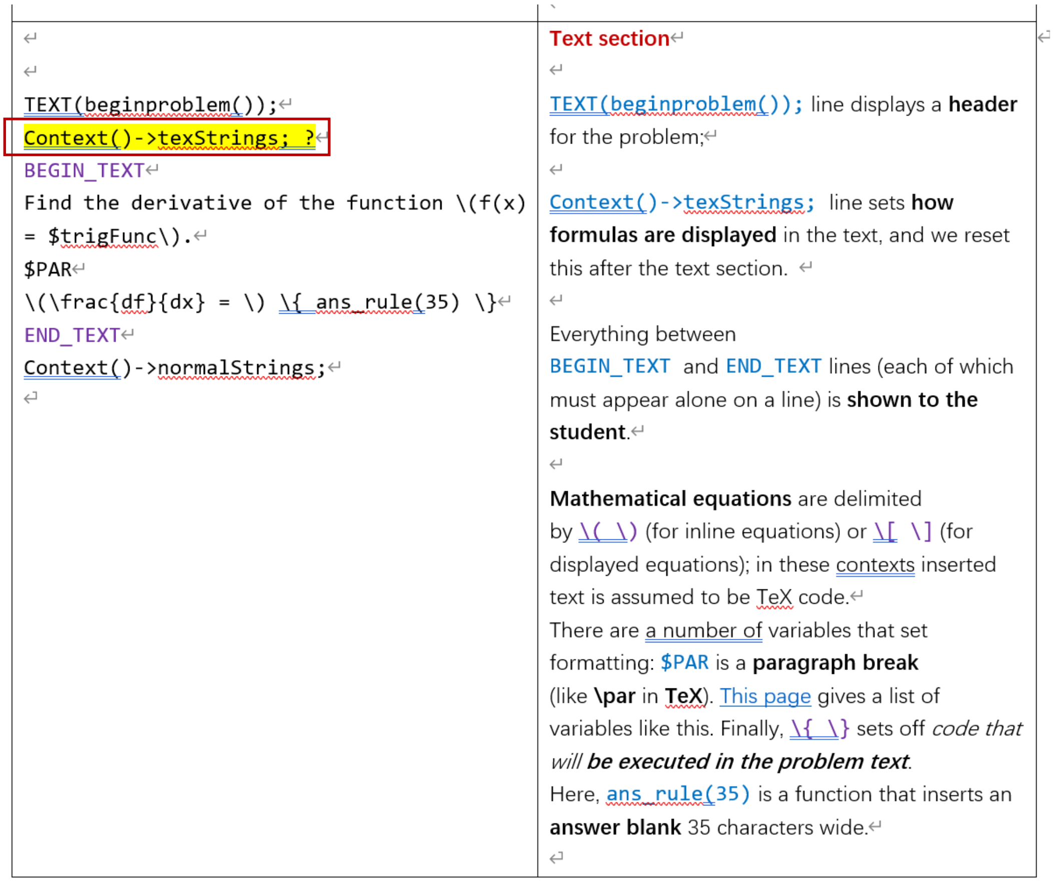
1. Consider Pipeline |> in functional programming:



```
TEXT(beginproblem());
Context()->texStrings;  ?
BEGIN_TEXT
Find the derivative of the function \(f(x)
= $trigFunc\).
$PAR
\(\frac{df}{dx} = \) \{ ans_rule(35) \}
END_TEXT
Context()->normalStrings;
```

Text section

`TEXT(beginproblem());` line displays a **header** for the problem;

`Context()->texStrings;` line sets **how formulas are displayed** in the text, and we reset this after the text section.

Everything between `BEGIN_TEXT` and `END_TEXT` lines (each of which must appear alone on a line) is **shown to the student**.

Mathematical equations are delimited by `\( \)` (for inline equations) or `\[ \]` (for displayed equations); in these contexts inserted text is assumed to be TeX code.
There are a number of variables that set formatting: `$PAR` is a **paragraph break** (like **\par** in TeX). This page gives a list of variables like this. Finally, `\{ \}` sets off *code that will **be executed in the problem text***.
Here, `ans_rule(35)` is a function that inserts an **answer blank** 35 characters wide.

What it does:

The **`Context()->texStrings;`** command switches the rendering context to produce LaTeX-formatted strings.

Purpose:

- **LaTeX Typesetting:** When active, it ensures that MathObjects (e.g., formulas, variables like **`$trigFunc`**) interpolated into text blocks (`BEGIN_TEXT`, `BEGIN_SOLUTION`) are rendered as LaTeX code.
- This allows proper mathematical notation (e.g., $\sin(2x)$ instead of plain-text sin(2x)).
- **`Context()->normalStrings;`** reverts to the default context after the text block, avoiding unintended LaTeX rendering in non-TeX parts (e.g., answer checking with **`ANS()`**).

Why It Matters:

- Without **`texStrings`**, variables might display in a non-LaTeX format (e.g., **sin(2x)**), breaking the problem's visual layout.
- Ensures solutions and problem statements are mathematically typeset for clarity.

What -> Means in Perl:

In Perl, **->** is the **dereference operator**, primarily used for:

1. **Accessing object methods or properties**:
   - ✧ **`Context()`** returns an object (the current "context" in WeBWorK).
   - ✧ **`->texStrings`** calls the **`texStrings`** method on that object, switching the context to output LaTeX-formatted strings.
   - ✧ Example: **`$obj->method()`** invokes **`method()`** on the object **`$obj`**.
     (调用**`$obj`** 里的 **`method()`**)

2. **Dereferencing references**:
   - ✧ It can also dereference array/hash references (e.g., **`$array_ref->[0]`**).

```
Context()->texStrings;
# Call the `texStrings` method on the Context object
```

This is equivalent to:

```
my $context = Context();  # Get the context object
$context->texStrings;     # Call texStrings() on it
# “my $context” creates a new variable named $context
that is local to the current block or scope (e.g., inside
a subroutine, loop, or conditional block).
```

What |> Means in OCaml:

In OCaml, **|>** is the **pipeline operator**, used for **function composition**:

- **`x |> f`** is equivalent to **`f(x)`**, allowing you to "pipe" (feed) a value into a function.
- Example:

```
let result = 5 |> add_one |> square;;
(* Equivalent to square(add_one(5)) *)
```

This chains functions in a readable, **left-to-right order**.

Thus, we can **distinguish -> and |>**:

Feature	**Perl ->**	**OCaml |>**
Purpose	Object method/property access	Function composition
Paradigm	Object-oriented	Functional
Syntax	`$obj->method()`	`x |> f |> g`
Underlying Action	Dereferences an object/reference	Rearranges function arguments

Similarity:

Both operators involve "**chaining**" actions:

- In Perl: `$obj->method1->method2` chains method calls on an object.
- In OCaml: `x |> f |> g` chains function applications.

Example Comparison:

Perl:

```
# Create an object and chain methods
my $result = MathObject->new(5)->add(3)->multiply(2);
```

Here, **`->`** accesses methods on the **`MathObject`** instance.

Final `$result`: 16 (as a `MathObject` instance with value 16).

OCaml:

```
(* Pipe a value through functions *)
let result = 5 |> add_one |> multiply_by 2;;
```

Here, **`|>`** passes **`5`** to **`add_one`**, then the result to **`multiply_by`**.

Final `result`: 12 (as an integer).

2. Create a repository on Github:

https://github.com/Smart-Jason/PG-Code-Generation.git