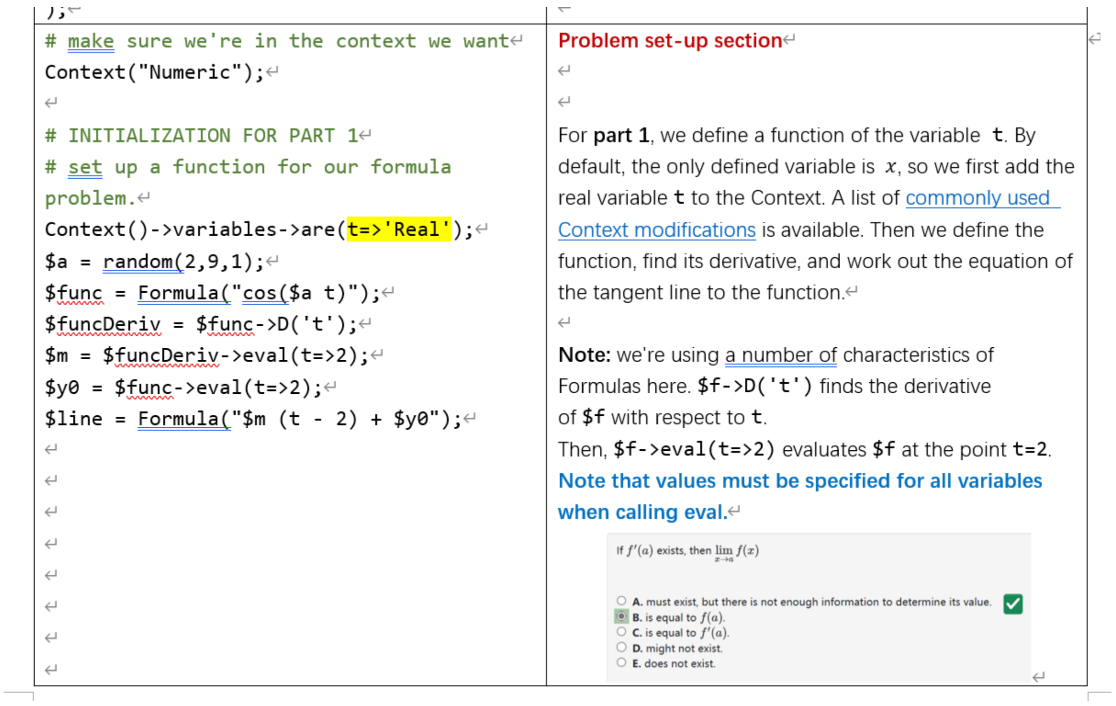
1. What is => in Perl?



```
);
# make sure we're in the context we want
Context("Numeric");

# INITIALIZATION FOR PART 1
# set up a function for our formula
problem.
Context()->variables->are(t=>'Real');
$a = random(2,9,1);
$func = Formula("cos($a t)");
$funcDeriv = $func->D('t');
$m = $funcDeriv->eval(t=>2);
$y0 = $func->eval(t=>2);
$line = Formula("$m (t - 2) + $y0");
```

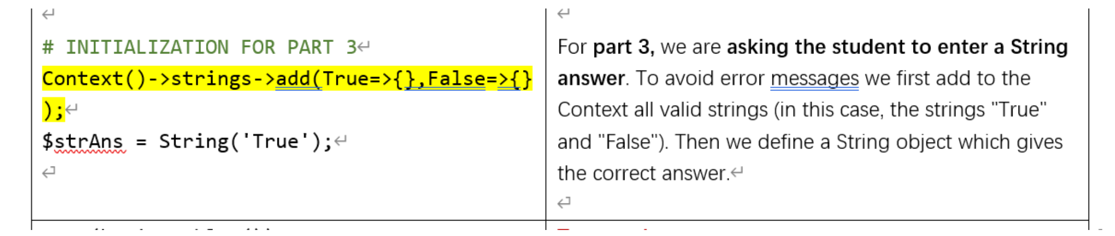
Problem set-up section

For **part 1**, we define a function of the variable **t**. By default, the only defined variable is x, so we first add the real variable **t** to the Context. A list of commonly used Context modifications is available. Then we define the function, find its derivative, and work out the equation of the tangent line to the function.

Note: we're using a number of characteristics of Formulas here. `$f->D('t')` finds the derivative of `$f` with respect to `t`.
Then, `$f->eval(t=>2)` evaluates `$f` at the point `t=2`.
Note that values must be specified for all variables when calling eval.

If $f'(a)$ exists, then $\lim_{x \to a} f(x)$

- ○ **A.** must exist, but there is not enough information to determine its value. ✓
- [✓ ● **B.** is equal to $f(a)$.]
- ○ **C.** is equal to $f'(a)$.
- ○ **D.** might not exist.
- ○ **E.** does not exist.



```
# INITIALIZATION FOR PART 3
Context()->strings->add(True=>{},False=>{}
);
$strAns = String('True');
```

For **part 3,** we are **asking the student to enter a String answer**. To avoid error messages we first add to the Context all valid strings (in this case, the strings "True" and "False"). Then we define a String object which gives the correct answer.

What do the two statements do:

```
Context()->variables->are(t => 'Real');
```

1. **Translation:** "In the current mathematical context, the variable **t** is a real number."
2. **Consequence:** Students must use **t** in their answers, and **t** behaves like a real variable (e.g., **`t^2`** is allowed, but **`sqrt(t)`** might warn if **`t`** is negative).

```
Context()->strings->add(True => {}, False => {});
```

1. **Translation:** "In the current context, the valid string answers are **'True'** and **'False'**, with no extra options."
2. **Consequence:** Students must enter exactly **True** or **False** (case-sensitive by default).

Definition of => in Perl:

In Perl (the language underlying WeBWorK's PG code), the **=>** operator is a **"fat comma"** that behaves similarly to a regular comma " **,** " .

Two key features of =>:

1. Automatic Quoting of the Left-Hand Side

If the left-hand side (the "key") is a **bareword** (a plain identifier without quotes), **=>** automatically quotes it as a string.

For example:

```
t => 'Real'    # Equivalent to ('t', 'Real')
True => {}     # Equivalent to ('True', {})
```

No need to explicitly write **'t'** or **'True'**; **=>** adds the quotes implicitly.

2. Improved Readability for Key-Value Pairs

=> is often used to visually associate keys with values in hashes or parameter lists.

For example:

```
# Without =>
my %hash = ('key1', 'value1', 'key2', 'value2');

# With =>
my %hash = (key1 => 'value1', key2 => 'value2');
# Cleaner and more intuitive
```

In our PG sample:

`Context()->variables->are(t => 'Real');`

`t => 'Real'` defines a key-value pair:

- Key: **`t`** (automatically quoted as **`'t'`**).
- Value: **`'Real'`** (the type of the variable).

This tells the context that the variable **`t`** is a **real number**.

`Context()->strings->add(True => {}, False => {});`

`True => {}` and **`False => {}`** define allowed strings with options:

- Key: **`True`** (automatically quoted as **`'True'`**).
- Value: **`{}`** (an empty hash of options; no special settings for these strings).

This adds **`'True'`** and **`'False'`** as valid string answers (with default settings).

Why Not Use a Regular Comma " , " ?

Technically, we could use " **,** ", but **=>** is preferred for clarity:

```
# Equivalent but less readable:
Context()->variables->are('t', 'Real');
Context()->strings->add('True', {}, 'False', {});
```

What we find so far:

- **=>** is syntactic sugar for readability and automatic quoting.
- It is commonly used in Perl for hashes, function arguments, or any key-value pairs.
- In WeBWorK PG, it is often used to configure contexts (variables, strings, etc.).

Let's go further with some OCaml analogies:

⭑ Perl's => Operator (Fat Comma)

What It Does:

- **=>** is used to define **key-value pairs** (like associating a key with its value).
- Automatically quotes the **left-hand side** (if it's a plain word) as a string.
- Makes code more readable than using commas.

Perl Examples:

```
# Example 1: Defining a hash (dictionary)
my %colors = (
    apple  => 'red',      # Equivalent to ('apple', 'red')
    banana => 'yellow',   # => quotes "banana" as a string
);

# Example 2: Function arguments
Context()->variables->are(t => 'Real');
# Equivalent to: Context()->variables->are('t', 'Real');
```

OCaml Analogy:

```
(* Tuples for key-value pairs *)
let variables = [("t", "Real")]  (* Like t => 'Real' *)

(* Labeled arguments (closer in spirit) *)
let add_string ~key:True ~value:{} = ...
(* Not exact, but similar to True => {} *)
```

⭑ `'Real'` in `Context()->variables->are(t => 'Real');`

✧ **Purpose**: Declares that the variable **t** is a real number (not complex, a string, etc.).
✧ **Why**: Ensures mathematical operations (like derivatives) follow real-number rules.
✧ **Example:**

```
Context()->variables->are(x => 'Real', y => 'Complex');
# x is real, y is complex
```

What Happens if I Simply Omit This?

By default, variables might be treated as complex numbers (depending on context settings). Specifying **'Real'** restricts **t** to real values, which is critical for problems like calculus or real functions.

* `{}` in `Context()->strings->add(True => {}, False => {});`

- ✧ **Purpose:** The `{}` is an empty hash that specifies no additional options for the strings 'True' and 'False'.
- ✧ **Why:** In Perl, the **add** method expects **a list of key-value pairs**, where the value is a hash of options. Empty hashes mean no extra settings (like case sensitivity or aliases).
- ✧ **Example with Options:**

```
# If you wanted 'true' (lowercase) to also be accepted as 'True',
you could add:
Context()->strings->add(
    True => {caseSensitive => 0}, # Allow case-insensitive
    False => {alias => 'F'}       # Allow "F" as an alias for "False"
);
```

OCaml Analogy:

```
(* In OCaml, we might use a record type with optional fields *)
type string_option = { caseSensitive : bool; alias : string option }
let true_opt = { caseSensitive = false; alias = None }
let false_opt = { caseSensitive = true; alias = Some "F" }
```

Here, `{}` in **Perl** is like leaving all fields in the **OCaml record** as defaults.

Overall:

Code Snippet	Purpose	OCaml Analogy
`t => 'Real'`	Declares `t` as a real number variable.	`("t", "Real")` in a list of tuples.
`True => {}`	Adds `'True'` as a valid string answer with no extra options.	`~key:True ~value:default_options`
`{}`	Empty options hash (use defaults).	`{ caseSensitive = true; alias = None }`

* What is a "Hash" in Perl?

Definition: A **hash** in Perl is a **key-value data structure**, similar to a **dictionary in Python** or a **tuple list in OCaml**.

- ✧ Keys are unique strings.
- ✧ Values can be any scalar (number, string, reference, etc.).

Python Analogy:

```
# Perl hash                    # Python dictionary
my %hash = (                   dict = {
    key1 => 'value1',              'key1': 'value1',
    key2 => 'value2',              'key2': 'value2',
);                             }
```

OCaml Analogy:

```
(* Using a list of tuples *)
let hash = [("key1", "value1"); ("key2", "value2")] ;;
```

The **are** Method for Variables in the Context:

The **are** method defines the variables allowed in the current mathematical context and specifies their types.

Syntax:

```
Context()->variables->are(variable1 => 'Type1', variable2 => 'Type2', ...);
```

Purpose:

- ✧ Declares which variables students can use in their answers.
- ✧ Specifies the variables' data types (e.g., real numbers, complex numbers).
- ✧ Overwrites any previous variable declarations in the context.

Overwriting Variables:

```
# Initially, variables x and y are defined
Context()->variables->are(x => 'Real', y => 'Real');

# Later, overwrite to only allow 't'
Context()->variables->are(t => 'Real');
# Now, x and y are invalid!
```

`Context()->variables->are(t => 'Real');`

Effect:

- The variable `t` is now recognized as a real number.
- Students must use `t` in their answers (e.g., `3t+2`).
- Mathematical operations (like derivatives or integrals) treat `t` as a real variable.
- Other variables (e.g., `x`, `y`) are no longer recognized unless explicitly added.

`variables` has add method as well:

```
# Start with x and y
Context()->variables->are(x => 'Real', y => 'Real');

# Add z without removing x and y
Context()->variables->add(z => 'Complex');
# Now x, y, z are valid
```

The add Method for Strings in the Context:

Purpose: The **add** method registers valid **string answers** in the problem's context.

`Context()->strings->add(True => {}, False => {});`

This tells WeBWorK: "Accept **only** the exact strings **'True'** and **'False'** as valid answers."

Syntax:

```
Context()->strings->add(
    StringName => { options },   # Key-value pairs
    ...
);
```

What `{}` Means:

- `{}` is an **empty hash** (no options specified).
- This means **'True'** and **'False'** are **case-sensitive** and **have no aliases**.
- Example with options:

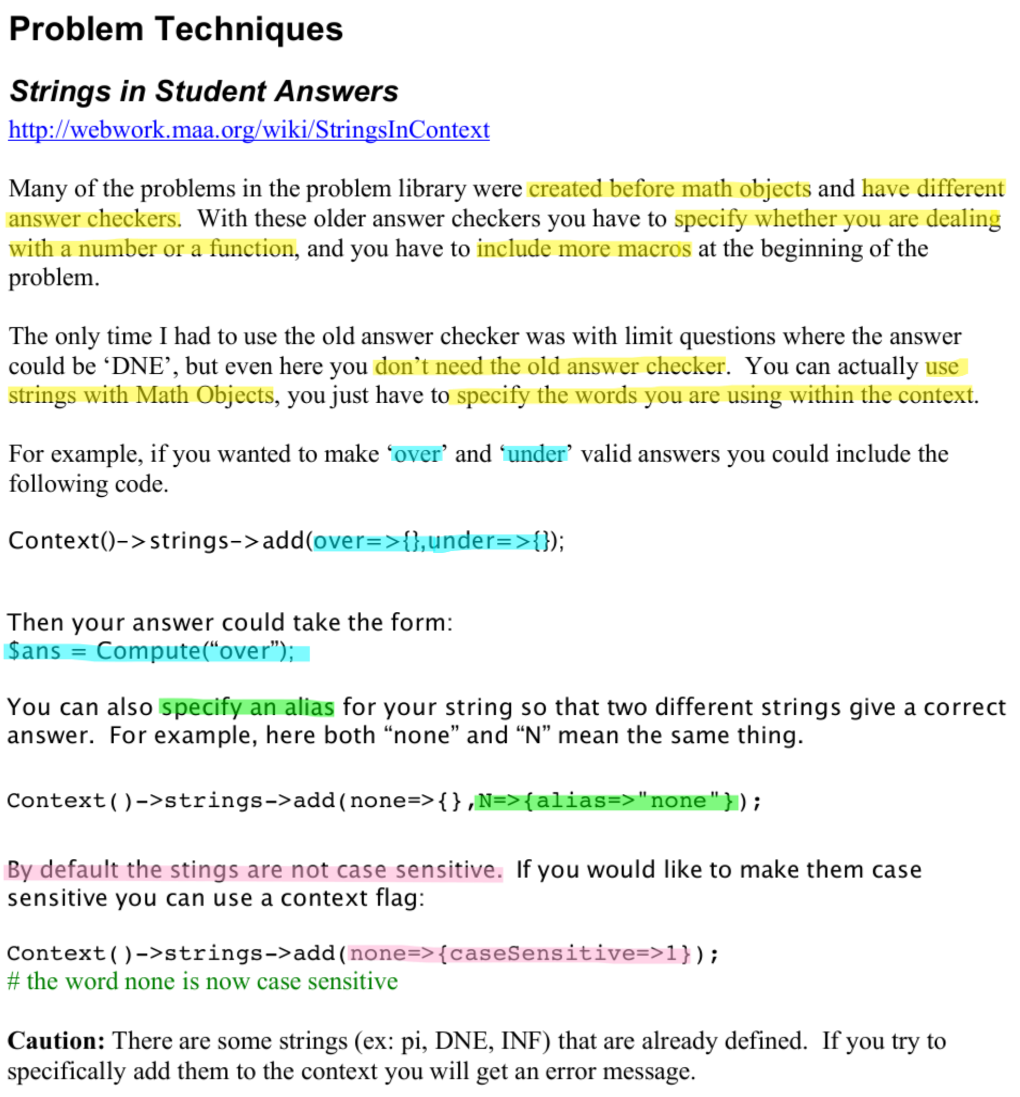
```
Context()->strings->add(
    True => { caseSensitive => 0 },  # Allow "TRUE", "true", etc.
    False => { alias => ['No', '0'] }  # Accept "No" or "0" as
aliases for "False"
);
```

* The `strings` Method in `Context()`:

- **What is `Context()`?**
  The `Context()` object in WeBWorK defines the **mathematical environment** for a problem. It controls:
  - **Variables** (e.g., `t => 'Real'`).
  - **Constants** (e.g., `pi`).
  - **Strings** (valid answers like `'True'`).
  - **Operators** (e.g., allowing/disallowing vector operations).

- **`strings` Method:**
  - `Context()->strings` refers to the **string-answer configuration** within the context.
  - `add` is a method of this configuration to define allowed strings.

UBC_Math_WeBWorK_Manual.pdf: Problem Techniques, p10:



Problem Techniques

Strings in Student Answers

http://webwork.maa.org/wiki/StringsInContext

Many of the problems in the problem library were created before math objects and have different answer checkers. With these older answer checkers you have to specify whether you are dealing with a number or a function, and you have to include more macros at the beginning of the problem.

The only time I had to use the old answer checker was with limit questions where the answer could be 'DNE', but even here you don't need the old answer checker. You can actually use strings with Math Objects, you just have to specify the words you are using within the context.

For example, if you wanted to make 'over' and 'under' valid answers you could include the following code.

```
Context()->strings->add(over=>{},under=>{});
```

Then your answer could take the form:

```
$ans = Compute("over");
```

You can also specify an alias for your string so that two different strings give a correct answer. For example, here both "none" and "N" mean the same thing.

```
Context()->strings->add(none=>{},N=>{alias=>"none"});
```

By default the stings are not case sensitive. If you would like to make them case sensitive you can use a context flag:

```
Context()->strings->add(none=>{caseSensitive=>1});
# the word none is now case sensitive
```

Caution: There are some strings (ex: pi, DNE, INF) that are already defined. If you try to specifically add them to the context you will get an error message.

Add method:

StringsInContext - WeBWorK_wiki

2. Another sample

See **PG_language_3.docx**.

~~3. Basic Perl syntax - WeBWorK_wiki~~

4. Build a Github team

~~超详细！Github 团队协作教程（Gitkraken 版） - thousfeet - 博客园~~
如何使用 Github 实现协同工作（例子：两人合作写代码）_协作写代码方式-CSDN 博客
Manage access