Programmer Documentation

In the repository, you can find two projects: **CzeTex** and **CzeTex.Tests**, where **CzeTex** represents the main program and **CzeTex.Tests** is the project that runs the tests.

CzeTex

The project and source code are divided into two parts. The first part deals with input file processing, building helper data structures, and providing utility methods. The second part handles generating the final .pdf file using the **iText**. library.

First Part

The first part includes the following files: **Commander.cs, Files.cs, SetupLoader.cs, Trie.cs** and **Util.cs**. Let's go through these files and their contents one by one.

Commander.cs

The file **Commander.cs** contains a class of the same name, which serves as the core of the program. It calls various functions and classes and ensures consistency throughout execution. It processes the content loaded from the input file and calls the class for processing the setup .json file.

The input file is processed using the method **ReadContent**, which detects a special character (default: /then extracts functions and their parameters from the text, searches the **trie** data structure for the relevant function reference, and calls it.

The class includes several functions such as **ReadRestForParameter, ReadParameter, etc.,** which complement each other in various cases to correctly extract parameters of CzeTex text functions.

Files.cs

Another file is **Files.cs**, which contains a class of the same name. This class is responsible for locating the file based on a user-provided parameter and correctly opening, reading, and closing it.

In addition, the Files class provides methods like **GetPdfPath** for generating the output .pdf path, or **GetBaseName**, to return the base name of the input file, and others.

SetupLoader.cs

Next is **SetupLoader.cs**, which contains two classes: **JsonEntry** and **SetupLoader**.

As the name suggests, **JsonEntry** represents a single entry in the setup .json file. It includes attributes for **addFunction, getFunction**, and optionally whether the function is dynamically generated and the symbol used in such case.

The **SetupLoader** class reads the mapping from the setup JSON file, which is then used to match specific CzeTex text functions with their code equivalents. As mentioned, functions can be divided into dynamically generated and predefined ones. In the **AddFunctions** method, we loop through all entries in the JSON file (already instantiated JsonEntry objects) and determine whether each function should be dynamically generated based on the **dynamicallyGenerated**. flag.

If a function needs to be created, we switch to the method **AddDynamicallyGeneratedFunction**, where delegates for `add` and `get` functions are generated and then added to the **trie** structure. Note: These dynamic functions are used only for functions that render a character, so they are tightly coupled with character generation. However, extending this is simple—just use a different delegate generator.

For code-defined functions, we determine whether they should have both `add` and `get` or just `add`. Then we use the **CreateDelegate** method with necessary parameters to create delegates and add them to the **trie** similarly as with dynamic functions.

Util.cs

This file contains a number of utility classes/functions used throughout the rest of the code. You'll find a generic stack implementation and the corresponding generic linked list node.

It also includes custom error classes that enhance error messages to better indicate where the user made a mistake in their CzeTex code. Furthermore, it contains static classes with string or parameter processing utilities.

Last but not least, it includes the class **FunctionGeneratorForPDF**, which provides methods to generate delegates for both `add` and `get` dynamic functions (as described earlier).

Trie.cs

The last file in this section is **Trie.cs**, which implements the **trie** (prefix tree) data structure used to create a bijective mapping between CzeTex text functions and actual implementation functions.

Besides the tree itself, the class has two arrays—one for `add` functions and another for `get` functions. The bijection is built such that the character path from the root to a node corresponds to the CzeTex function name. This places constraints on the function naming: names must consist of lowercase English letters only. Uniqueness is guaranteed by the setup JSON, which does not allow duplicate keys (function names).

When a node is reached whose character path matches a function name, it stores an unsigned integer representing an index into the add/get arrays. Note: If a function lacks a `get`, the corresponding slot is just `null`, allowing a single index to serve both arrays.

This mapping technique ensures we can find the function mapping in time linear to the function name length, instead of the number of entries. It avoids the upper-bound performance issues of hashing, and since the function name universe is known, hashing would be overkill.

Note: Extending support to uppercase letters would only require changing constants in **TrieConstants** : `smallestAvailableCharacter` to `'A'` and `numberOfChildren` to `58`. However, this would also include special characters and increase memory usage—each node would reserve space for 57 children even if only one is used, which is inefficient for sparse trees.

Second Part

The second part of the source code handles PDF generation and related operations, again split across multiple files.

Fonts.cs and Signs.cs

These two files are quite similar, each containing a class of constants. `Fonts` defines font types, default text size, and paragraph/line spacing. `Signs` defines constants for special characters such as division, implication, quantifiers, etc.

PDFSupport.cs

The **PDFSupport.cs** file contains, as the name suggests, supporting classes and methods—mostly classes for text layers or wrappers around iText's text class to extend its functionality.

There is a base class **TextCharacteristics**, representing text features like font and size. Several subclasses inherit from it and override virtual methods like `Special`, `Add`, etc., to affect text rendering. For example, `UnderLineText` overrides `Special` to apply underlining.

To keep the inheritance structure clear, classes with shared traits inherit from a common parent. For example: **TextCharacteristics ⇒ SpecialTextCharacteristics ⇒ UnderLineText** — so `UnderLineText` inherits from `SpecialTextCharacteristics`, which groups all text features that override the `Special` method.

As mentioned, this file also includes a class extending the iText `Text` class, primarily to fix its suboptimal implementation of vertical shifting ("raising") of text.

PDFStack.cs

In **PDFStack.cs** you'll find the `CharacteristicsStack` class—a stack implementation storing text characteristics. The top layer is used to determine current rendering style.

The stack allows partial definitions; for example, if a new layer defines only the font, other values (like size) are inherited. If the stack is empty, defaults are used.

PDF.cs and PDFMath.cs

The final files are **PDF.cs** and **PDFMath.cs**, containing the **PDF** class, which provides the actual PDF generation functions. The mathematical functions are in PDFMath.cs, the rest in PDF.cs.

In **PDF.cs**, the `PDF` class initializes the PDF structure in its constructor and then modifies it by calling mapped functions. It includes methods for creating text, headings, lists, etc.

In **PDFMath.cs** you'll find math-related text functions, special math symbols (quantifiers, Greek letters), and basic algebraic expressions. These methods are used by the dynamic character rendering functions.

Tests

In the **CzeTex.Tests** project, the **Tests.cs** file contains dozens of tests for both individual functions and full file interpretation.

The **RunScript**, function simulates user terminal input and runs `dotnet run` with specific input parameters. It supports input-only files or input with setup files.

Running Tests,

To run the tests, navigate to **CzeTex.Tests** and run **dotnet tests**.

Note: **RunScript** does not return error messages—it uses the **exit code**. to evaluate behavior. This may vary by platform. The default **exit code** is 134, but if your tests all return a consistent different value, it means the exit code for exceptions is different. Update it by replacing the value of the constant on line **15** in **Tests.cs**.

FOR DETAILED INFORMATION ABOUT ALL CLASSES, METHODS, ETC., SEE THE COMMENTED SOURCE CODE.

Possible Extensions

One of the most straightforward extensions would be to support more complex mathematical expressions. This would require algorithms to calculate the size of notation parts. Since the **iText** library lacks proper math support, this would involve image/vector insertion for rendering formulas.

Another potential extension is support for CzeTex function names with diacritics or special characters. Uppercase support is already discussed in the **Trie**. section.

Other possible additions include support for numbered text or combining multiple formatting styles like underline and strikethrough.

Conclusion

The **CzeTex** application successfully implements both basic and moderately advanced text features like font settings, page breaks, proper spacing, special styles, and basic mathematical notation—algebraic expressions, quantifiers, and math symbols.

It also allows the user to easily customize it via setup .json files to remap function names. Partial setups are supported, so users who don’t need all features (or don’t speak English) can load faster and work more comfortably—for example, using the Czech version.