SyntaxTutor

Generated by Doxygen 1.13.2

Chapter 1

SyntaxTutor: An interactive Tool for Learning Syntax Analysis

SyntaxTutor is an educational application designed to help compiler students understand LL(1) and SLR(1) parsing algorithms. Through a visual and interactive interface, it guides users step-by-step through the computation of FIRST, FOLLOW, CLOSURE, GOTO, predictive parsing tables, and LR automata, offering real-time pedagogical feedback.

Rather than acting as a mere calculator, SyntaxTutor functions as a learning companion. It explains the reasoning behind each step, highlights common mistakes, and encourages students to engage with the theory behind the algorithms.

1.1 Academic Context

SyntaxTutor is part of a Final Degree Project (TFG) developed at the University of Málaga (UMA), in the Computer Engineering program. Its main goal is to offer an educational companion for students learning syntax analysis, going beyond traditional calculators by incorporating guided feedback, visualization, and gamified learning.

1.2 Key Features

- Educational Focus: built to teach, not just compute.
- Visualization: derivation trees, intermediate steps, sets, and tables.
- Exportable Results: useful for reports or coursework.

1.3 Interface Screenshots

1.3.1 Main Menu

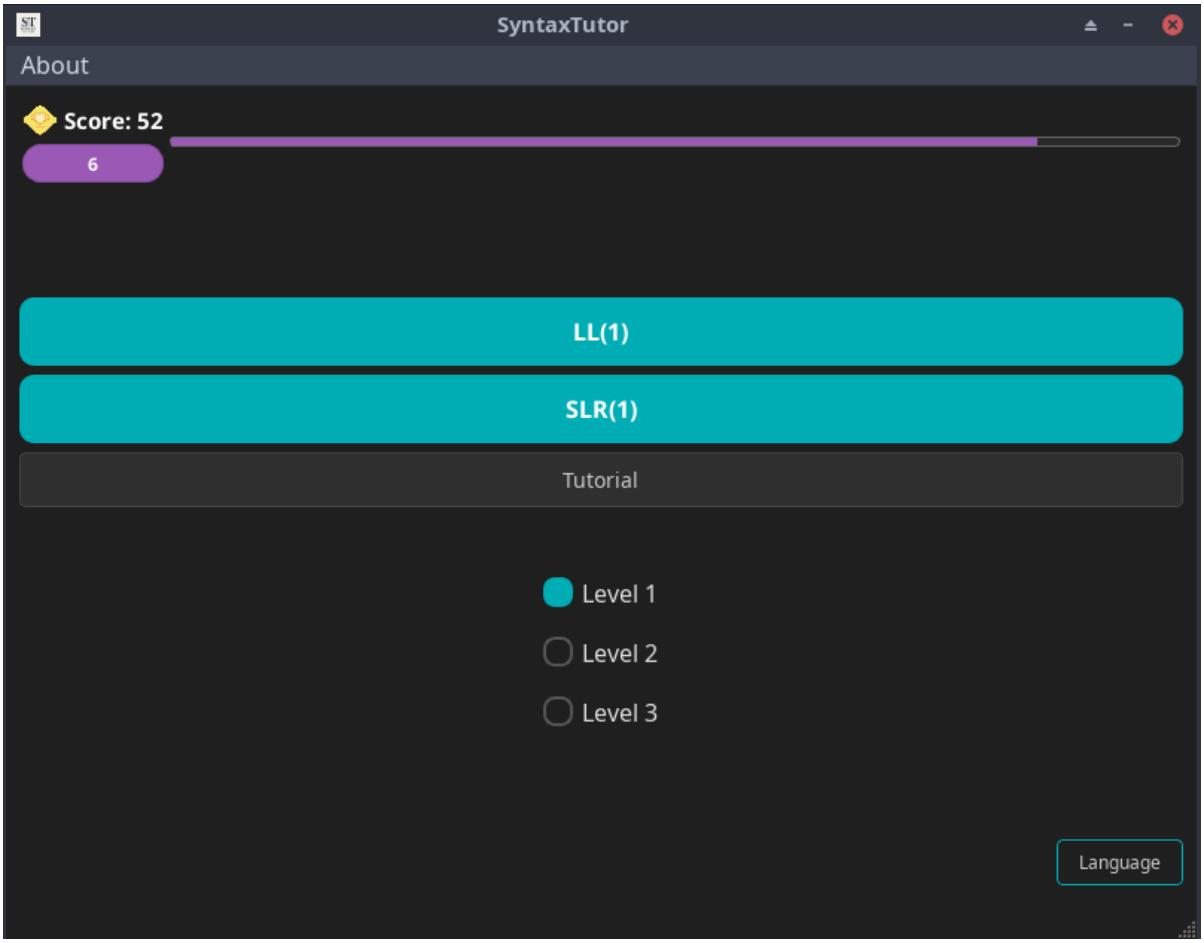


Figure 1.1 Main window

Home screen with gamification, levels, and language options.

1.3.2 LL(1) Learning Mode

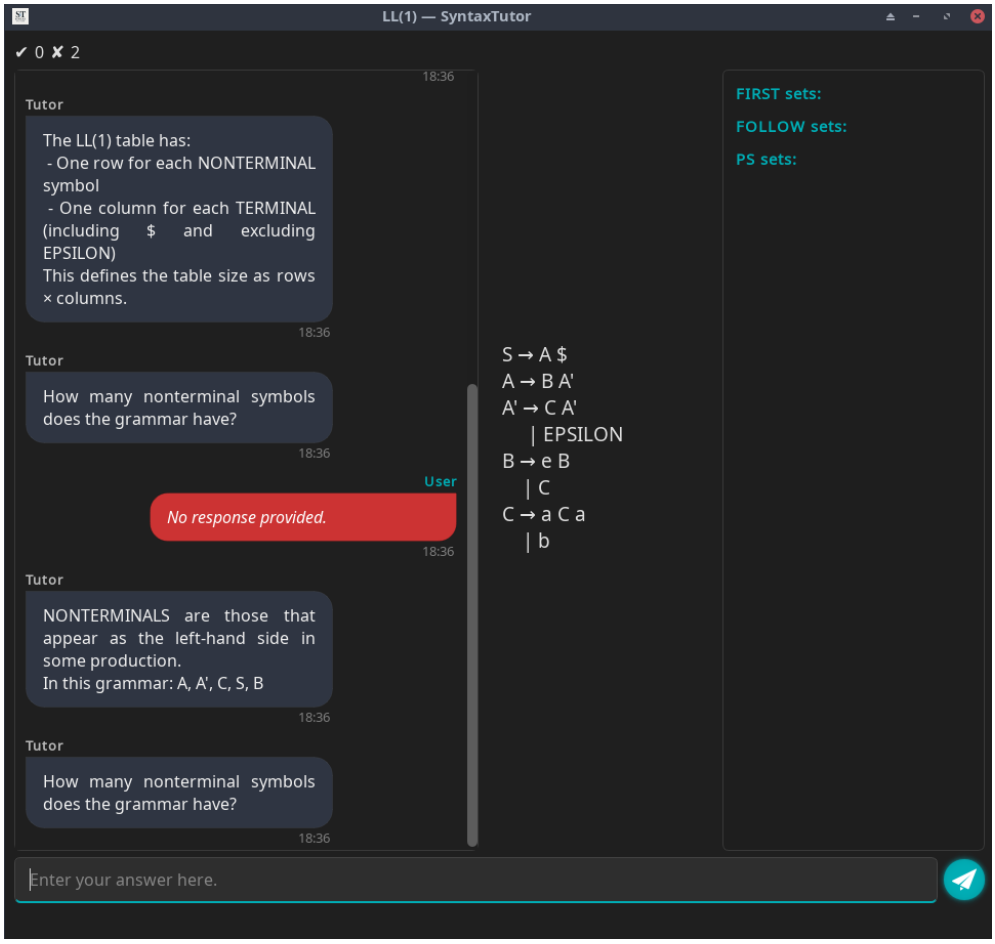


Figure 1.2 LL(1) dialog view

Interactive LL(1) tutor asks questions and provides feedback.

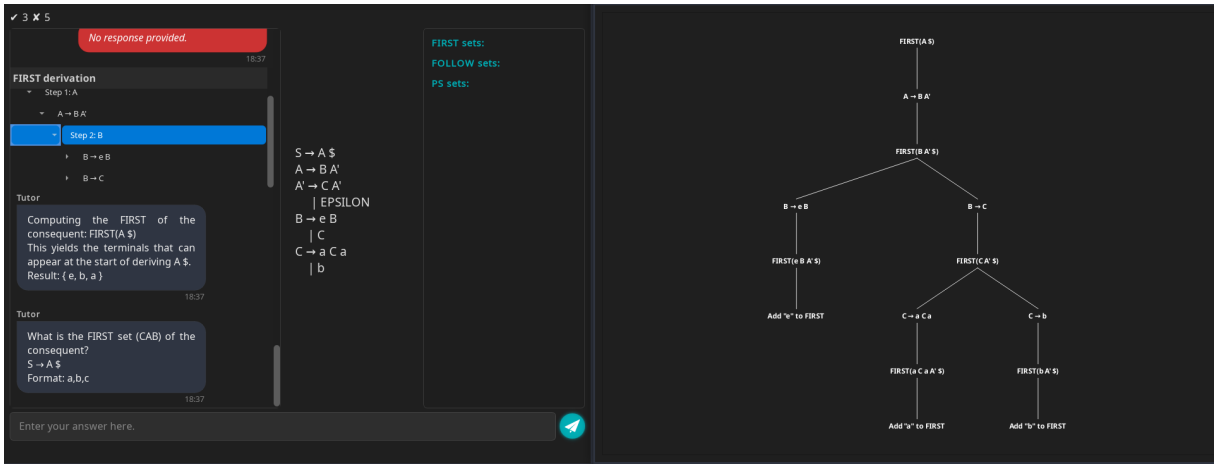


Figure 1.3 LL(1) derivation tree

Derivation tree view showing how FIRST sets are built step-by-step.

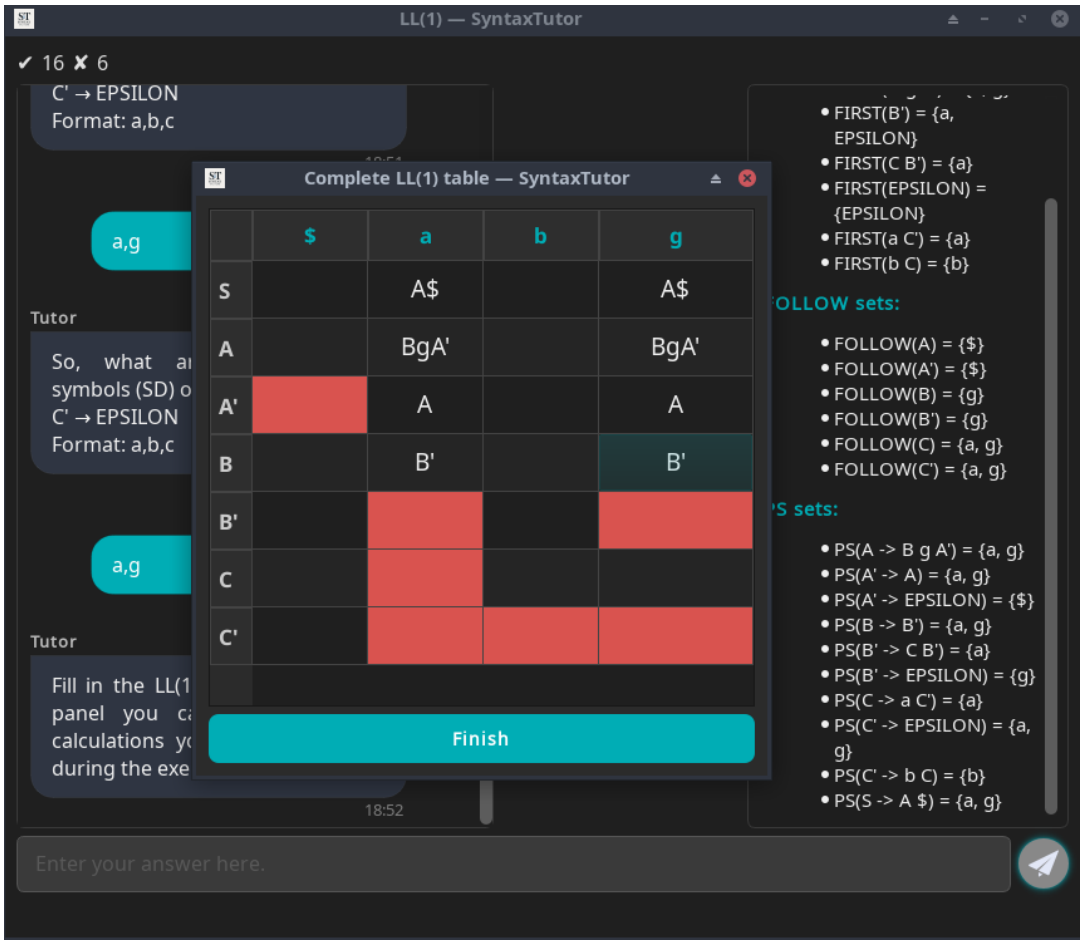


Figure 1.4 LL(1) table task

Completion of the LL(1) predictive table with visual guidance.

1.3.3 SLR(1) Learning Mode

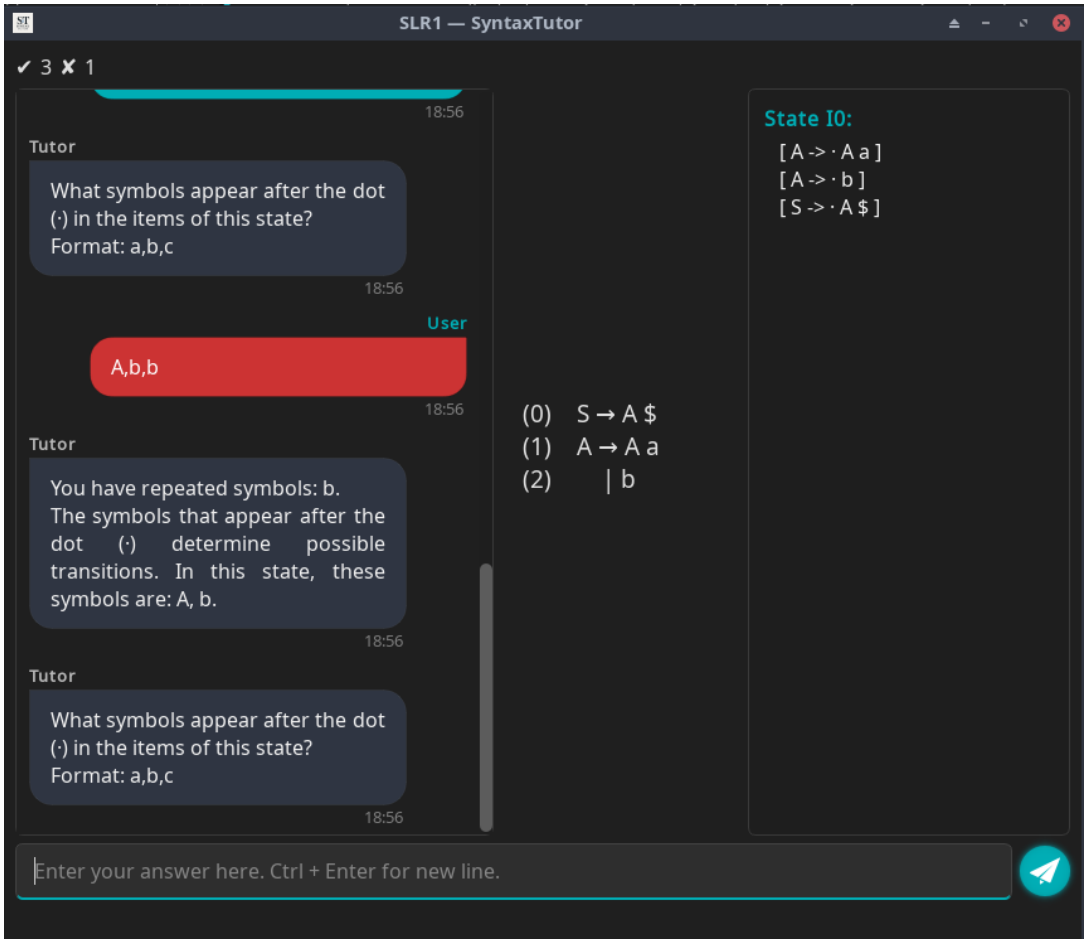


Figure 1.5 SLR(1) item view

User is asked to identify symbols after the dot in an LR(0) item.

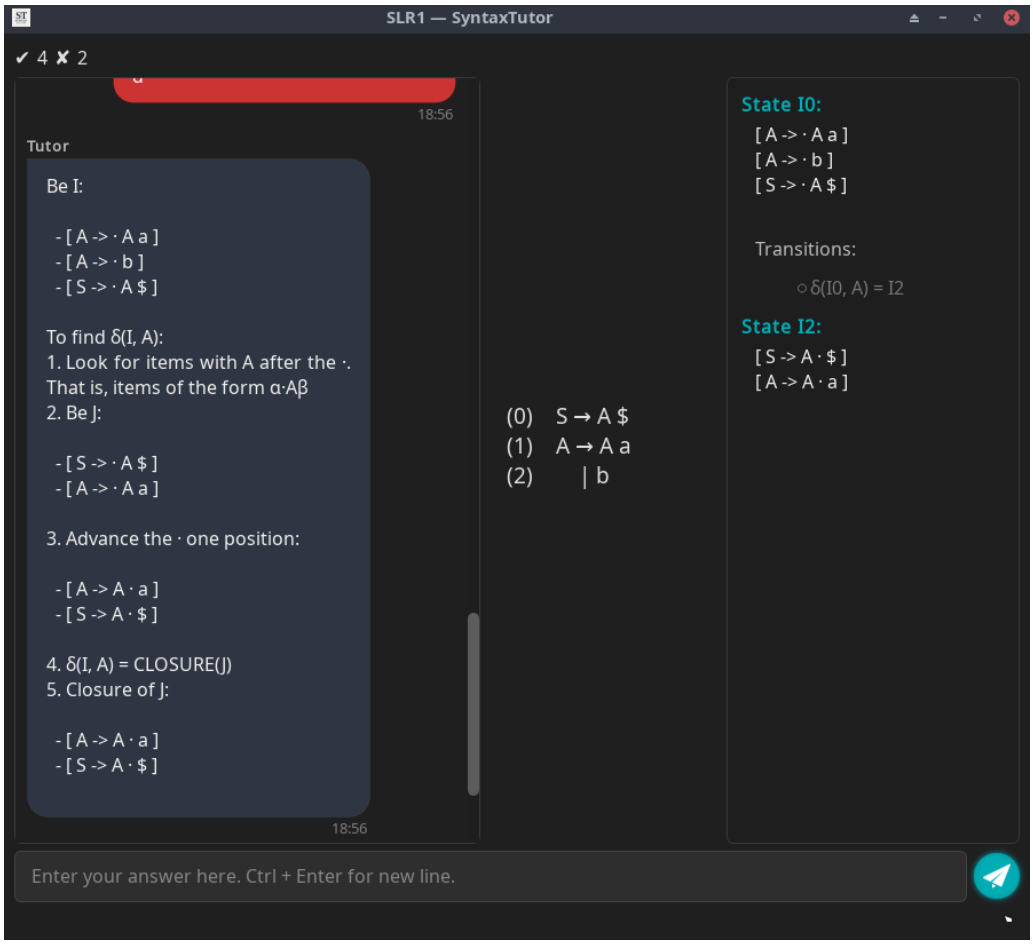


Figure 1.6 SLR(1) automaton construction

Step-by-step explanation of the GOTO/closure construction.

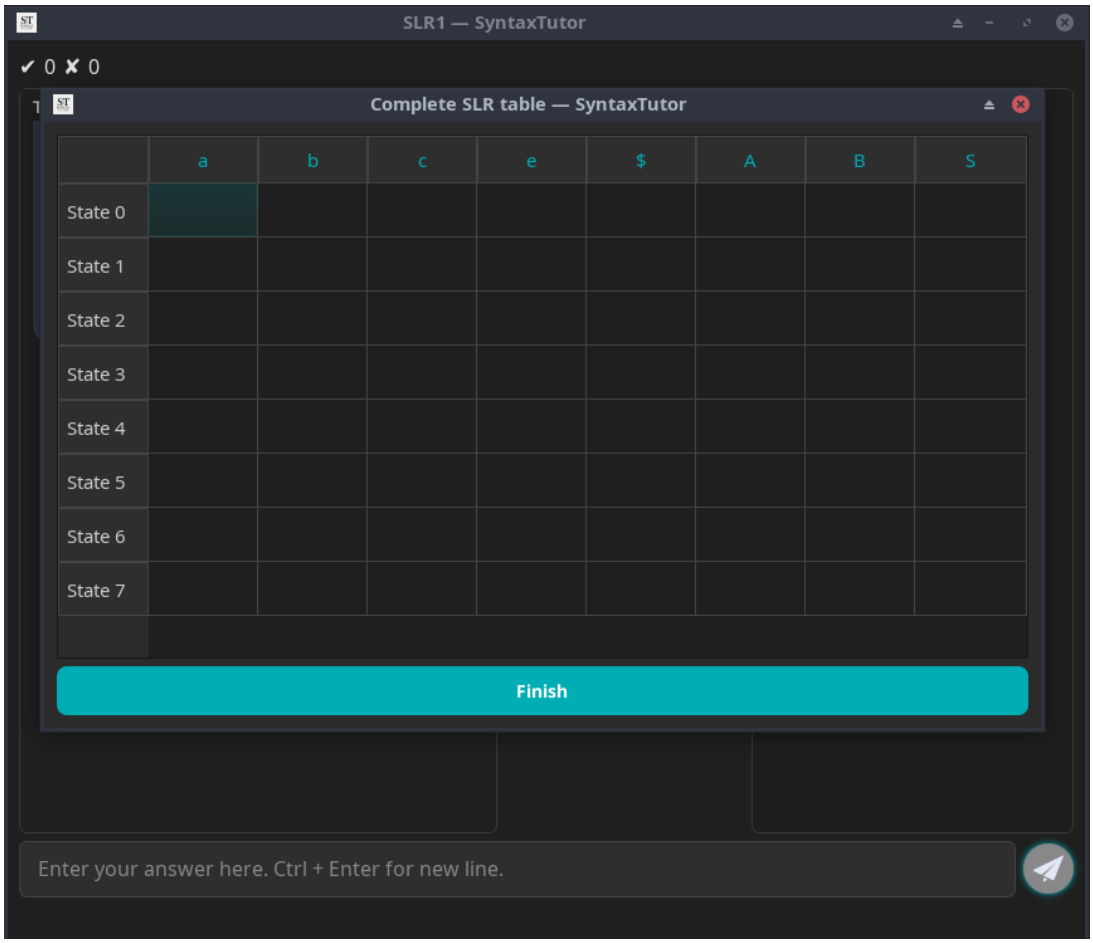


Figure 1.7 SLR(1) table fill-in

Interactive SLR(1) table to complete, with states and terminals/non-terminals.

1.3.4 Assisted Mode: Guided Table Completion

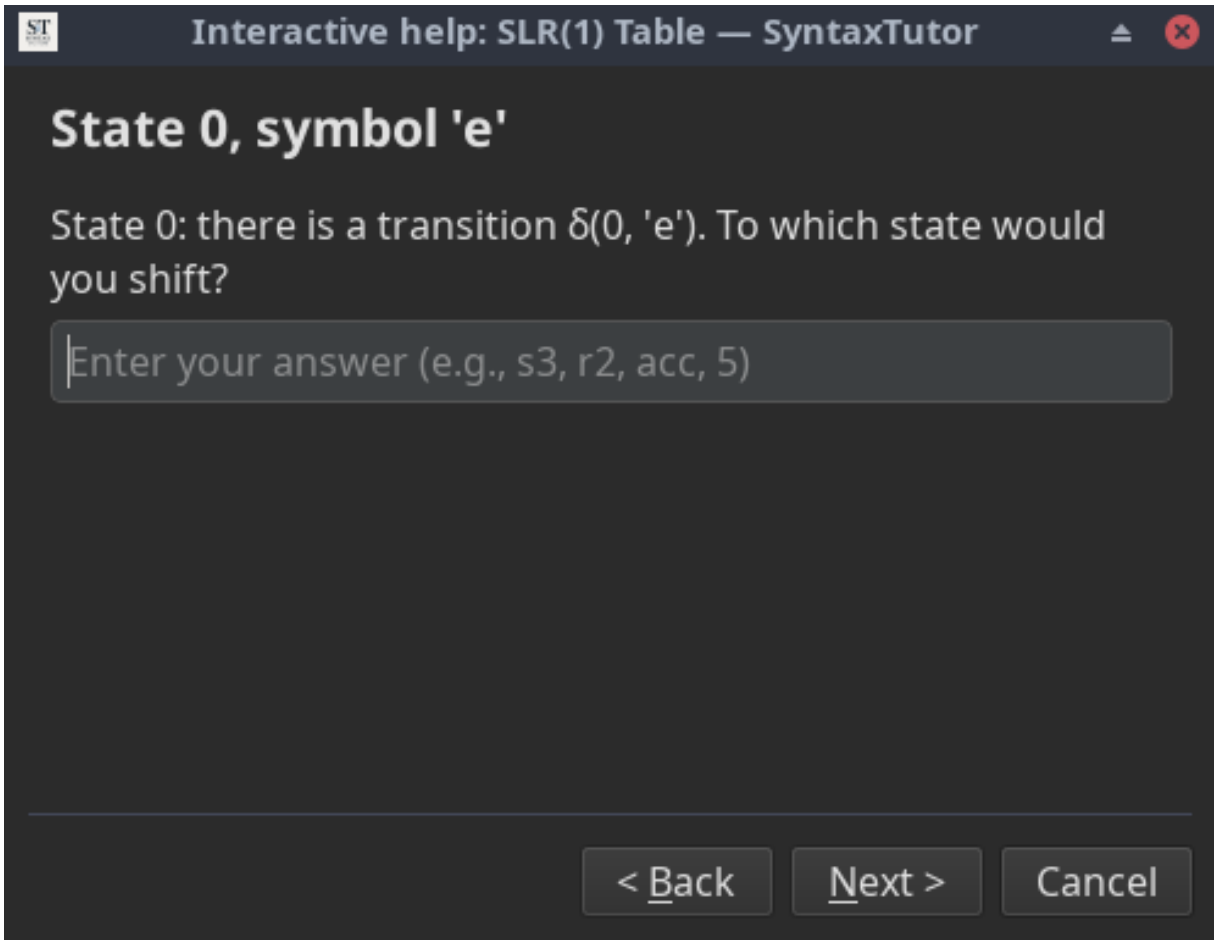


Figure 1.8 SLR(1) guided mode

SyntaxTutor walks the student through each cell in the parsing table with hints and context.

1.4 Technologies Used

- C++: efficient implementation of parsing algorithms
- Qt6: modern, cross-platform graphical user interface.
- Modular architecture: clean separation between logic and UI, designed for easy extensibility.

1.5 Downloads

Precompiled builds of SyntaxTutor are available in the Releases tab:

- Linux (X11): executable AppImage
- Windows: ZIP archive with the .exe
- macOS: .app bundles for both Apple Silicon (ARM) and Intel

Warning

The Windows and macOS versions are not digitally signed. Your operating system may display a warning when running the application. You can bypass it manually if you trust the source.

1.6 Building from Source

To build SyntaxTutor from source, you just need:

- Qt6 (including qmake6)
- A C++20-compliant compiler qmake6 make ` This will generate the executable in te project directory.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

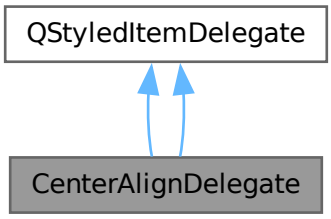
Here is a list of all files with brief descriptions:

Chapter 5

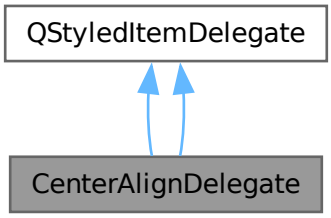
Class Documentation

5.1 CenterAlignDelegate Class Reference

Inheritance diagram for CenterAlignDelegate:



Collaboration diagram for CenterAlignDelegate:



Public Member Functions

- void initStyleOption (QStyleOptionViewItem *opt, const QModelIndex &idx) const override
- void initStyleOption (QStyleOptionViewItem *opt, const QModelIndex &idx) const override

5.1.1 Member Function Documentation

5.1.1.1 initStyleOption() [1/2]

void CenterAlignDelegate::initStyleOption (
QStyleOptionViewItem * opt,
const QModelIndex & idx) const [inline], [override]

5.1.1.2 initStyleOption() [2/2]

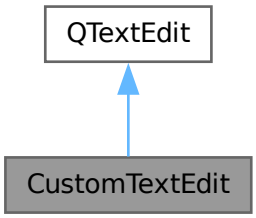
void CenterAlignDelegate::initStyleOption (
QStyleOptionViewItem * opt,
const QModelIndex & idx) const [inline], [override]

The documentation for this class was generated from the following files:

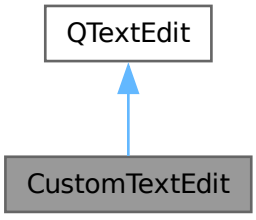
- lltabledialog.cpp
- slrtabledialog.cpp

5.2 CustomTextEdit Class Reference

#include <customtextedit.h>
Inheritance diagram for CustomTextEdit:



Collaboration diagram for CustomTextEdit:



Signals

- void sendRequested ()

Public Member Functions

- CustomTextEdit (QWidget *parent=nullptr)

Protected Member Functions

- void keyPressEvent (QKeyEvent *event) override

5.2.1 Constructor & Destructor Documentation

5.2.1.1 CustomTextEdit()

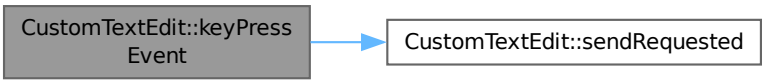
CustomTextEdit::CustomTextEdit (
QWidget * parent = nullptr) [explicit]

5.2.2 Member Function Documentation

5.2.2.1 keyPressEvent()

void CustomTextEdit::keyPressEvent (
QKeyEvent * event) [override], [protected]

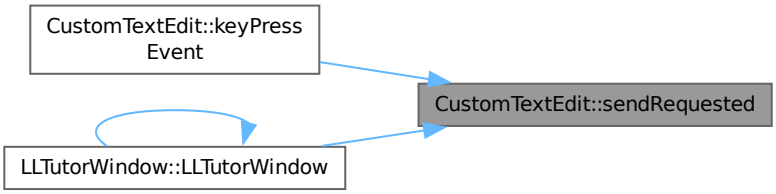
Here is the call graph for this function:



5.2.2.2 sendRequested

void CustomTextEdit::sendRequested () [signal]

Here is the caller graph for this function:



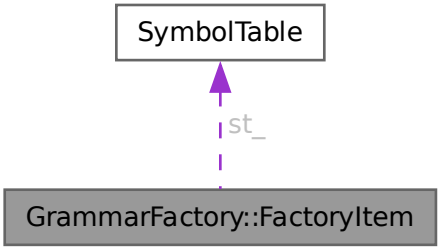
The documentation for this class was generated from the following files:

- customtextedit.h
- customtextedit.cpp

5.3 GrammarFactory::FactoryItem Struct Reference

Represents an individual grammar item with its associated symbol table.
#include <grammar_factory.hpp>

Collaboration diagram for GrammarFactory::FactoryItem:



Public Member Functions

- FactoryItem (const std::unordered_map< std::string, std::vector< production > > &grammar)

  Constructor that initializes a FactoryItem with the provided grammar.

Public Attributes

- std::unordered_map< std::string, std::vector< production > > g_

  Stores the grammar rules where each key is a non-terminal symbol and each value is a vector of production rules.
- SymbolTable st_

  Symbol table associated with this grammar item.

5.3.1 Detailed Description

Represents an individual grammar item with its associated symbol table.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 FactoryItem()

GrammarFactory::FactoryItem::FactoryItem (
const std::unordered_map< std::string, std::vector< production > > & grammar) [explicit]

Constructor that initializes a FactoryItem with the provided grammar.

Parameters

grammar	The grammar to initialize the FactoryItem with.

5.3.3 Member Data Documentation

5.3.3.1 g_

std::unordered_map<std::string, std::vector<production> > GrammarFactory::FactoryItem::g_

Stores the grammar rules where each key is a non-terminal symbol and each value is a vector of production rules.

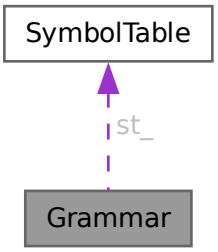
5.3.3.2 st_

SymbolTable GrammarFactory::FactoryItem::st_
Symbol table associated with this grammar item.
The documentation for this struct was generated from the following files:

- backend/grammar_factory.hpp
- backend/grammar_factory.cpp

5.4 Grammar Struct Reference

Represents a context-free grammar, including its rules, symbol table, and starting symbol.
#include <grammar.hpp>
Collaboration diagram for Grammar:



Public Member Functions

- Grammar ()
- Grammar (const std::unordered_map< std::string, std::vector< production > > &grammar)
- void SetAxiom (const std::string &axiom)

  Sets the axiom (entry point) of the grammar.
- bool HasEmptyProduction (const std::string &antecedent) const

  Checks if a given antecedent has an empty production.
- std::vector< std::pair< const std::string, production > > FilterRulesByConsequent (const std::string &arg) const

  Filters grammar rules that contain a specific token in their consequent.
- void Debug () const

  Prints the current grammar structure to standard output.
- void AddProduction (const std::string &antecedent, const std::vector< std::string > &consequent)

  Adds a production rule to the grammar and updates the symbol table.
- std::vector< std::string > Split (const std::string &s)

  Splits a string into grammar symbols using the current symbol table.

Public Attributes

- std::unordered_map< std::string, std::vector< production > > g_

  Stores the grammar rules with each antecedent mapped to a list of productions.
- std::string axiom_

  The axiom or entry point of the grammar.
- SymbolTable st_

  Symbol table of the grammar.

5.4.1 Detailed Description

Represents a context-free grammar, including its rules, symbol table, and starting symbol.
This structure encapsulates all components required to define and manipulate a grammar, including production rules, the associated symbol table, and metadata such as the start symbol. It supports construction, transformation, and analysis of grammars.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Grammar() [1/2]

Grammar::Grammar () [default]

5.4.2.2 Grammar() [2/2]

Grammar::Grammar (
const std::unordered_map< std::string, std::vector< production > > & grammar) [explicit]

5.4.3 Member Function Documentation

5.4.3.1 AddProduction()

void Grammar::AddProduction (
const std::string & antecedent,
const std::vector< std::string > & consequent)
Adds a production rule to the grammar and updates the symbol table.
This function inserts a new production of the form A → into the grammar, where antecedent is the non-terminal A and consequent is the sequence . It also updates the internal symbol table to reflect any new symbols introduced.

Parameters

antecedent	The left-hand side non-terminal of the production.
consequent	The right-hand side sequence of grammar symbols.

5.4.3.2 Debug()

void Grammar::Debug () const
Prints the current grammar structure to standard output.
This function provides a debug view of the grammar by printing out all rules, the axiom, and other relevant details.

5.4.3.3 FilterRulesByConsequent()

std::vector< std::pair< const std::string, production > > Grammar::FilterRulesByConsequent (
const std::string & arg) const
Filters grammar rules that contain a specific token in their consequent.

Parameters

arg	The token to search for within the consequents of the rules.

Returns

std::vector of pairs where each pair contains an antecedent and its respective production that includes the specified token.

Searches for rules in which the specified token is part of the consequent and returns those rules.

5.4.3.4 HasEmptyProduction()

bool Grammar::HasEmptyProduction (
const std::string & antecedent) const
Checks if a given antecedent has an empty production.

Parameters

antecedent	The left-hand side (LHS) symbol to check.

Returns

true if there exists an empty production for the antecedent, otherwise false.

An empty production is represented as <antecedent> -> ;, indicating that the antecedent can produce an empty string.

5.4.3.5 SetAxiom()

void Grammar::SetAxiom (
const std::string & axiom)
Sets the axiom (entry point) of the grammar.

Parameters

axiom	The entry point or start symbol of the grammar.

Defines the starting point for the grammar, which is used in parsing algorithms and must be a non-terminal symbol present in the grammar.

5.4.3.6 Split()

std::vector< std::string > Grammar::Split (
const std::string & s)
Splits a string into grammar symbols using the current symbol table.
This function tokenizes the input string s into a sequence of grammar symbols based on the known entries in the symbol table. It uses a greedy approach, matching the longest valid symbol at each step.

Parameters

s	The input string to split.

Returns

A vector of grammar symbols extracted from the string.

5.4.4 Member Data Documentation

5.4.4.1 axiom_

std::string Grammar::axiom_
The axiom or entry point of the grammar.

5.4.4.2 g_

std::unordered_map<std::string, std::vector<production> > Grammar::g_
Stores the grammar rules with each antecedent mapped to a list of productions.

5.4.4.3 st_

SymbolTable Grammar::st_
Symbol table of the grammar.
The documentation for this struct was generated from the following files:

- backend/grammar.hpp
- backend/grammar.cpp

5.5 GrammarFactory Struct Reference

Responsible for creating and managing grammar items and performing checks on grammars.
#include <grammar_factory.hpp>

Classes

- struct FactoryItem

  Represents an individual grammar item with its associated symbol table.

Public Member Functions

- void Init ()

  Initializes the GrammarFactory and populates the items vector with initial grammar items.
- Grammar PickOne (int level)

  Picks a random grammar based on the specified difficulty level (1, 2, or 3).
- Grammar GenLL1Grammar (int level)

  Generates a LL(1) random grammar based on the specified difficulty level.
- Grammar GenSLR1Grammar (int level)

  Generates a SLR(1) random grammar based on the specified difficulty lefel.
- Grammar Lv1 ()

  Generates a Level 1 grammar.
- Grammar Lv2 ()

  Generates a Level 2 grammar by combining Level 1 items.
- Grammar Lv3 ()

  Generates a Level 3 grammar by combining a Level 2 item and a Level 1 item.
- Grammar Lv4 ()

  Generates a Level 4 grammar by combining Level 3 and Level 1 items.
- Grammar Lv5 ()

  Generates a Level 5 grammar by combining Level 4 and Level 1 items.
- Grammar Lv6 ()

  Generates a Level 6 grammar by combining Level 5 and Level 1 items.
- Grammar Lv7 ()

  Generates a Level 7 grammar by combining Level 6 and Level 1 items.
- FactoryItem CreateLv2Item ()

  Creates a Level 2 grammar item for use in grammar generation.
- bool HasUnreachableSymbols (Grammar &grammar) const

  Checks if a grammar contains unreachable symbols (non-terminals that cannot be derived from the start symbol).
- bool IsInfinite (Grammar &grammar) const

  Checks if a grammar is infinite, meaning there are non-terminal symbols that can never derive a terminal string. This happens when a production leads to an infinite recursion or an endless derivation without reaching terminal symbols. For example, a production like: S -> A A -> a A | B B -> c B could lead to an infinite derivation of non-terminals.
- bool HasDirectLeftRecursion (const Grammar &grammar) const

Checks if a grammar contains direct left recursion (a non-terminal can produce itself on the left side of a production in one step).

- bool HasIndirectLeftRecursion (const Grammar &grammar) const

  Checks if a grammar contains indirect left recursion.
- bool HasCycle (const std::unordered_map< std::string, std::unordered_set< std::string > > &graph) const

  Checks if directed graph has a cycle using topological sort.
- std::unordered_set< std::string > NullableSymbols (const Grammar &grammar) const

  Find nullable symbols in a grammar.
- void RemoveLeftRecursion (Grammar &grammar)

  Removes direct left recursion in a grammar. A grammar has direct left recursion when one of its productions is A -> A a, where A is a non terminal symbol and "a" the rest of the production. The procedure removes direct left recursion by adding a new non terminal. So, if the productions with left recursion are A -> A a | b, the result would be A -> b A'; A'-> a A' | EPSILON.
- void LeftFactorize (Grammar &grammar)

  Perfoms left factorization. A grammar could be left factorized if it have productions with the same prefix for one non terminal. For example, A -> a x | a y; could be left factorized because it has "a" as the common prefix. The left factorization is done by adding a new non terminal symbol that contains the uncommon part, and by unifying the common prefix in a one producion. So, A -> a x | a y would be A -> a A'; A' -> x | y.
- std::vector< std::string > LongestCommonPrefix (const std::vector< production > &productions)

  Finds the longest common prefix among a set of productions.
- bool StartsWith (const production &prod, const std::vector< std::string > &prefix)

  Checks if a production starts with a given prefix.
- std::string GenerateNewNonTerminal (Grammar &grammar, const std::string &base)

  Generates a new non-terminal symbol that is unique in the grammar.
- void NormalizeNonTerminals (FactoryItem &item, const std::string &nt) const

  Replaces all non-terminal symbols in a grammar item with a single target non-terminal.
- void AdjustTerminals (FactoryItem &base, const FactoryItem &cmb, const std::string &target_nt) const

  Adjusts the terminal symbols between two grammar items.
- std::unordered_map< std::string, std::vector< production > > Merge (const FactoryItem &base, const FactoryItem &cmb) const

  Merges the grammar rules of two grammar items into a single grammar.

Public Attributes

- std::vector< FactoryItem > items

  A vector of FactoryItem objects representing different level 1 grammar items created by the Init method.
- std::vector< std::string > terminal_alphabet_

  A vector of terminal symbols (alphabet) used in the grammar.
- std::vector< std::string > non_terminal_alphabet_

  A vector of non-terminal symbols (alphabet) used in the grammar.

5.5.1 Detailed Description

Responsible for creating and managing grammar items and performing checks on grammars.

5.5.2 Member Function Documentation

5.5.2.1 AdjustTerminals()

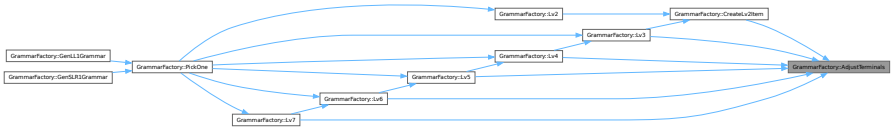
```
void GrammarFactory::AdjustTerminals (
            FactoryItem & base,
            const FactoryItem & cmb,
            const std::string & target_nt) const
```

Adjusts the terminal symbols between two grammar items.
This function modifies the terminal symbols of a base grammar item so that they do not conflict with those of the item being combined. It also renames terminals to ensure consistency and inserts the target non-terminal where appropriate.

Parameters

base	The base grammar item to adjust.
cmb	The grammar item being combined with the base.
target_nt	The target non-terminal symbol used for replacement.

Here is the caller graph for this function:



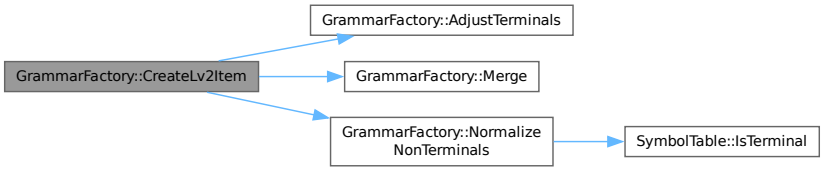
5.5.2.2 CreateLv2Item()

GrammarFactory::FactoryItem GrammarFactory::CreateLv2Item ()
Creates a Level 2 grammar item for use in grammar generation.
This function generates a Level 2 grammar item, which can be used as a building block for creating more complex grammars.

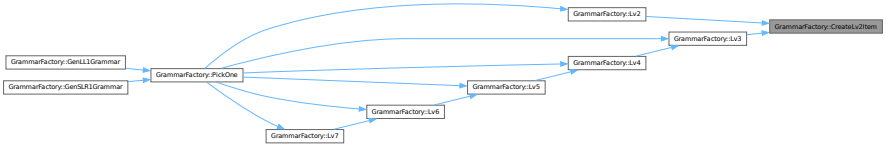
Returns

A FactoryItem representing a Level 2 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.2.3 GenerateNewNonTerminal()

std::string GrammarFactory::GenerateNewNonTerminal (
Grammar & grammar,
const std::string & base)

Generates a new non-terminal symbol that is unique in the grammar.
This function creates a new non-terminal symbol by appending a prime symbol (') to the base name until the resulting symbol is not already present in the grammar's symbol table. It is used during left factorization to introduce new non-terminals for factored productions.

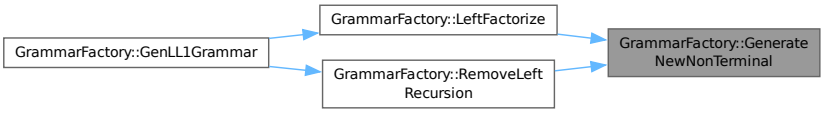
Parameters

grammar	The grammar in which the new non-terminal will be added.
base	The base name for the new non-terminal.

Returns

A unique non-terminal symbol derived from the base name.

Here is the caller graph for this function:



5.5.2.4 GenLL1Grammar()

Grammar GrammarFactory::GenLL1Grammar (
int level)

Generates a LL(1) random grammar based on the specified difficulty level.

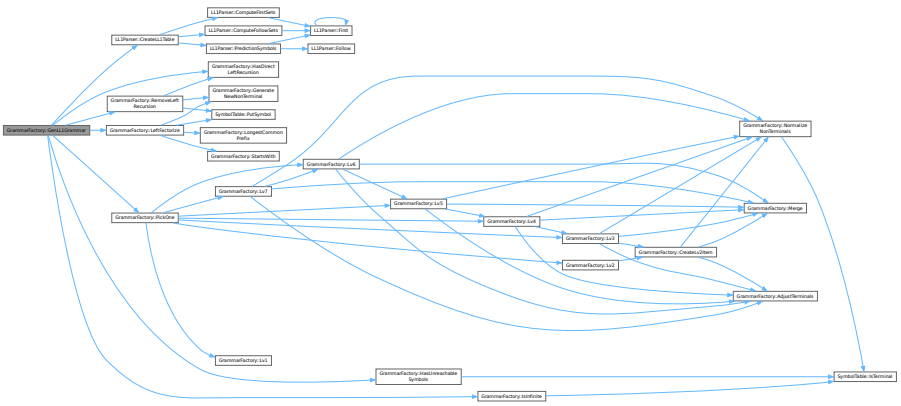
Parameters

level	The difficulty level (1, 2, or 3)

Returns

A random LL(1) grammar.

Here is the call graph for this function:



5.5.2.5 GenSLR1Grammar()

Grammar GrammarFactory::GenSLR1Grammar (
int level)

Generates a SLR(1) random grammar based on the specified difficulty lefel.

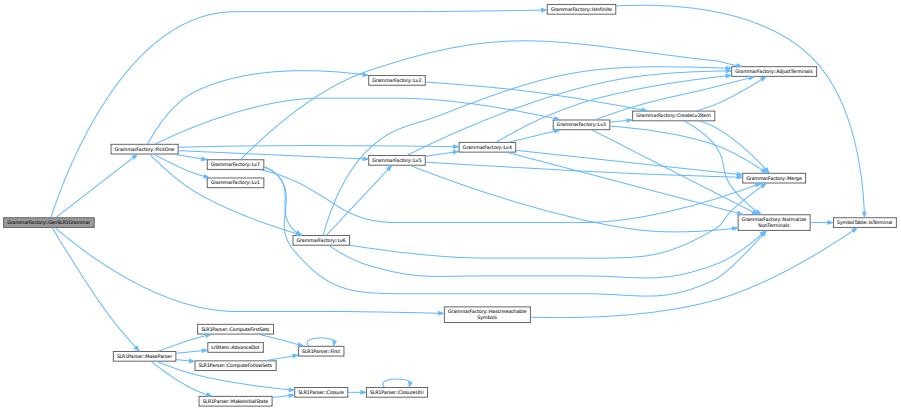
Parameters

level	The difficulty level (1, 2, or 3)

Returns

A random SLR(1) grammar.

Here is the call graph for this function:



5.5.2.6 HasCycle()

bool GrammarFactory::HasCycle (
const std::unordered_map< std::string, std::unordered_set< std::string > > & graph) const

Checks if directed graph has a cycle using topological sort.

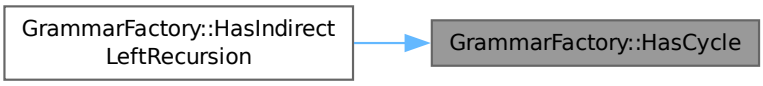
Parameters

graph	The directed graph.

Returns

true if grammar has cycle.

Here is the caller graph for this function:



5.5.2.7 HasDirectLeftRecursion()

bool GrammarFactory::HasDirectLeftRecursion (
const Grammar & grammar) const

Checks if a grammar contains direct left recursion (a non-terminal can produce itself on the left side of a production in one step).

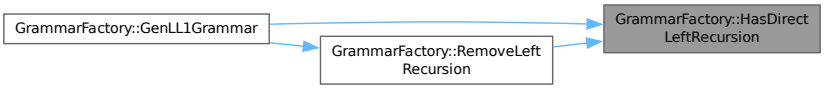
Parameters

grammar	The grammar to check.

Returns

true if there is direct left recursion, false otherwise.

Here is the caller graph for this function:



5.5.2.8 HasIndirectLeftRecursion()

bool GrammarFactory::HasIndirectLeftRecursion (
const Grammar & grammar) const

Checks if a grammar contains indirect left recursion.

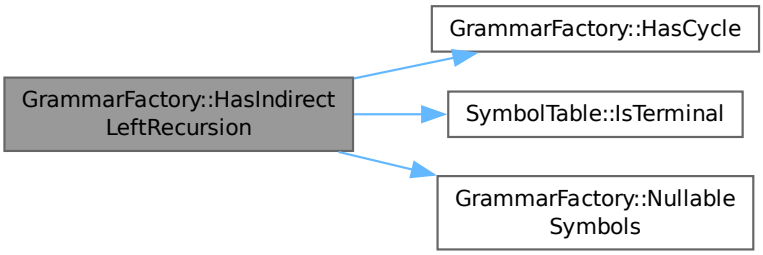
Parameters

grammar	The grammar to check.

Returns

true if there is direct left recursion, false otherwise.

Here is the call graph for this function:



5.5.2.9 HasUnreachableSymbols()

bool GrammarFactory::HasUnreachableSymbols (
Grammar & grammar) const

Checks if a grammar contains unreachable symbols (non-terminals that cannot be derived from the start symbol).

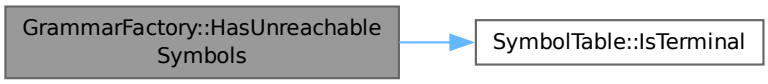
Parameters

grammar	The grammar to check.

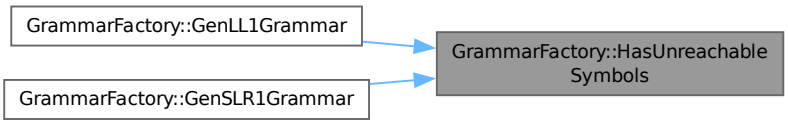
Returns

true if there are unreachable symbols, false otherwise.

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.2.10 Init()

void GrammarFactory::Init ()

Initializes the GrammarFactory and populates the items vector with initial grammar items.

5.5.2.11 IsInfinite()

bool GrammarFactory::IsInfinite (
Grammar & grammar) const

Checks if a grammar is infinite, meaning there are non-terminal symbols that can never derive a terminal string. This happens when a production leads to an infinite recursion or an endless derivation without reaching terminal symbols. For example, a production like: S -> A A -> a A | B B -> c B could lead to an infinite derivation of non-terminals.

Parameters

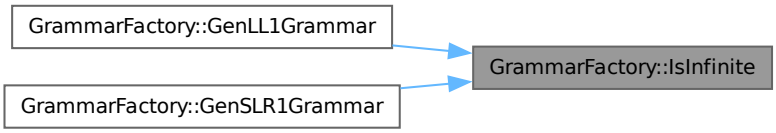
grammar	The grammar to check.

Returns

true if the grammar has infinite derivations, false otherwise.

Here is the call graph for this function:

Here is the caller graph for this function:



5.5.2.12 LeftFactorize()

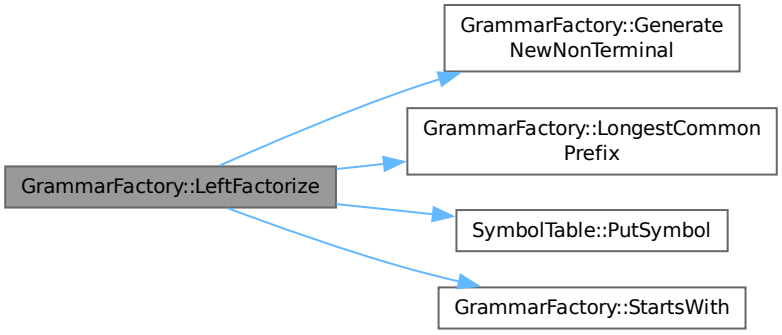
void GrammarFactory::LeftFactorize (
Grammar & grammar)

Perfoms left factorization. A grammar could be left factorized if it have productions with the same prefix for one non terminal. For example, A -> a x | a y; could be left factorized because it has "a" as the common prefix. The left factorization is done by adding a new non terminal symbol that contains the uncommon part, and by unifying the common prefix in a one producion. So, A -> a x | a y would be A -> a A'; A' -> x | y.

Parameters

grammar	The grammar to be left factorized.

Here is the call graph for this function:



Here is the caller graph for this function:

5.5.2.13 LongestCommonPrefix()

std::vector< std::string > GrammarFactory::LongestCommonPrefix (
const std::vector< production > & productions)

Finds the longest common prefix among a set of productions.

This function computes the longest sequence of symbols that is common to the beginning of all productions in the given vector. It is used during left factorization to identify common prefixes that can be factored out.

Parameters

productions	A vector of productions to analyze.

Returns

A vector of strings representing the longest common prefix. If no common prefix exists, an empty vector is returned.

Here is the caller graph for this function:

5.5.2.14 Lv1()

Grammar GrammarFactory::Lv1 ()
Generates a Level 1 grammar.

Returns

A Level 1 grammar.

Here is the caller graph for this function:

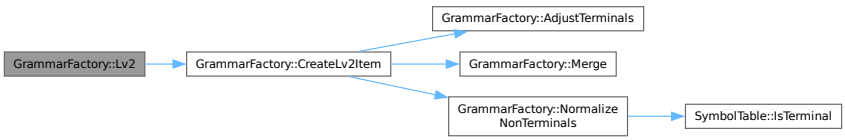
5.5.2.15 Lv2()

Grammar GrammarFactory::Lv2 ()
Generates a Level 2 grammar by combining Level 1 items.

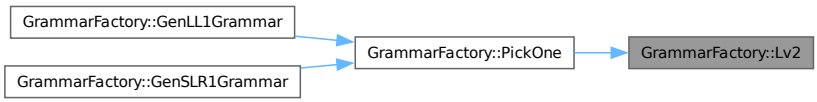
Returns

A Level 2 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



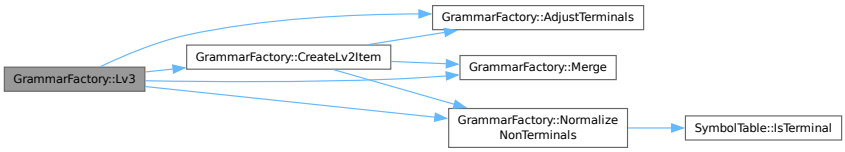
5.5.2.16 Lv3()

Grammar GrammarFactory::Lv3 ()
Generates a Level 3 grammar by combining a Level 2 item and a Level 1 item.

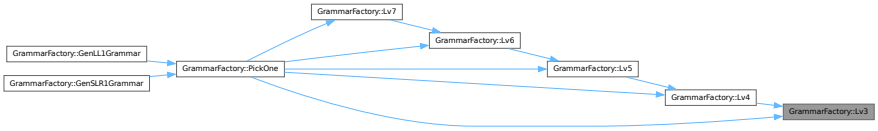
Returns

A Level 3 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



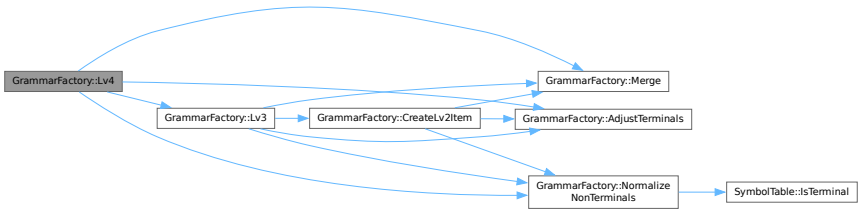
5.5.2.17 Lv4()

Grammar GrammarFactory::Lv4 ()
Generates a Level 4 grammar by combining Level 3 and Level 1 items.
This function creates a more complex grammar by combining elements from Level 3 and Level 1 grammars. It is used to generate grammars with increased complexity for testing or parsing purposes.

Returns

A Level 4 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:

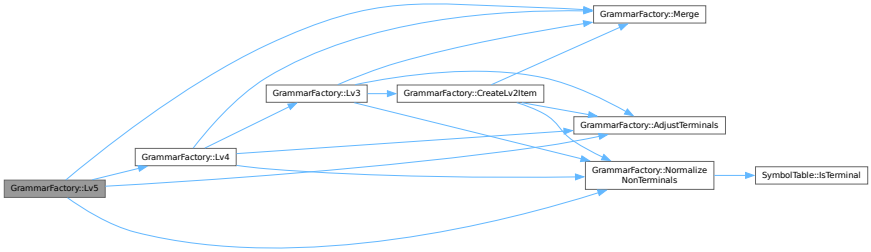
5.5.2.18 Lv5()

Grammar GrammarFactory::Lv5 ()
Generates a Level 5 grammar by combining Level 4 and Level 1 items.
This function creates a more advanced grammar by combining elements from Level 4 and Level 1 grammars. It is used to generate grammars with higher complexity for testing or parsing purposes.

Returns

A Level 5 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:

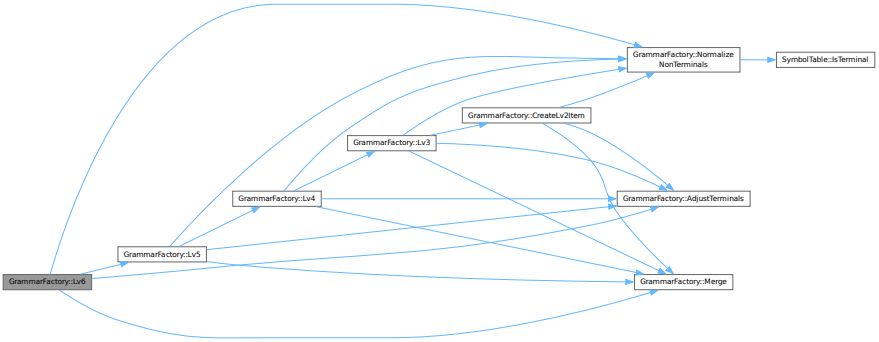
5.5.2.19 Lv6()

Grammar GrammarFactory::Lv6 ()
Generates a Level 6 grammar by combining Level 5 and Level 1 items.
This function creates a highly complex grammar by combining elements from Level 5 and Level 1 grammars. It is used to generate grammars with advanced structures for testing or parsing purposes.

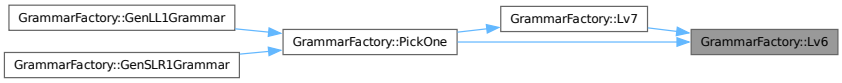
Returns

A Level 6 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



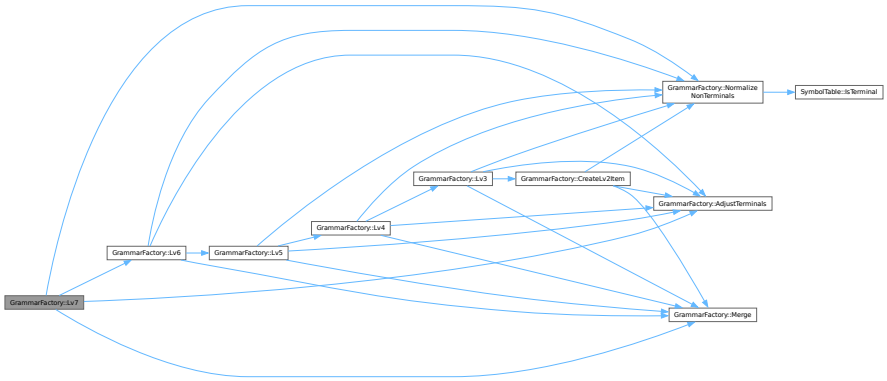
5.5.2.20 Lv7()

Grammar GrammarFactory::Lv7 ()
Generates a Level 7 grammar by combining Level 6 and Level 1 items.
This function creates a very complex grammar by combining elements from Level 6 and Level 1 grammars. It is used to generate grammars with highly advanced structures for testing or parsing purposes.

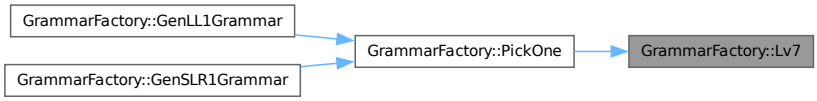
Returns

A Level 7 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.2.21 Merge()

std::unordered_map< std::string, std::vector< production > > GrammarFactory::Merge (
const FactoryItem & base,
const FactoryItem & cmb) const
Merges the grammar rules of two grammar items into a single grammar.
This function performs a raw combination of the production rules from both grammar items, resulting in a single grammar map that contains all productions.

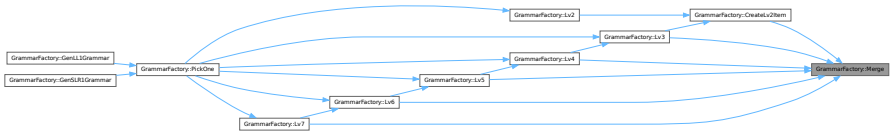
Parameters

base	The first grammar item.
cmb	The second grammar item.

Returns

A merged grammar map containing all production rules from both inputs.

Here is the caller graph for this function:



5.5.2.22 NormalizeNonTerminals()

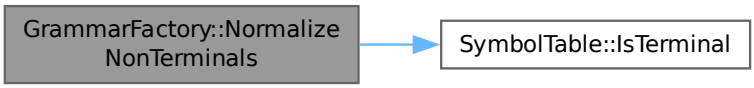
void GrammarFactory::NormalizeNonTerminals (
FactoryItem & item,
const std::string & nt) const

Replaces all non-terminal symbols in a grammar item with a single target non-terminal.
This function is used during grammar combination to normalize the non-terminal symbols in a given FactoryItem, so that they are consistent and compatible with another item.

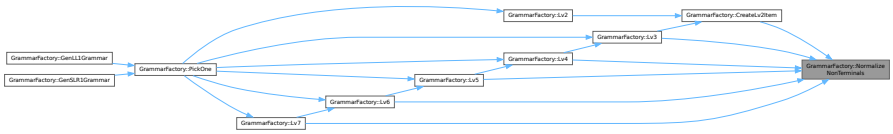
Parameters

item	The grammar item whose non-terminals will be renamed.
nt	The new non-terminal symbol that will replace all existing ones.

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.2.23 NullableSymbols()

std::unordered_set< std::string > GrammarFactory::NullableSymbols (
const Grammar & grammar) const

Find nullable symbols in a grammar.

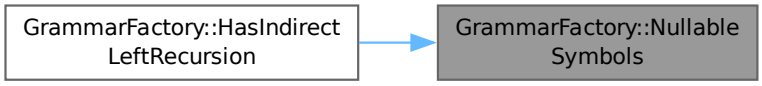
Parameters

grammar	The grammar to check.

Returns

set of nullable symbols.

Here is the caller graph for this function:



5.5.2.24 PickOne()

Grammar GrammarFactory::PickOne (
int level)

Picks a random grammar based on the specified difficulty level (1, 2, or 3).

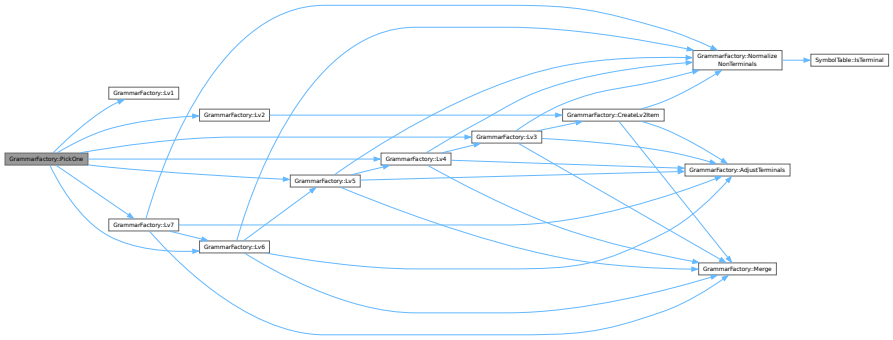
Parameters

level	The difficulty level (1, 2, or 3).

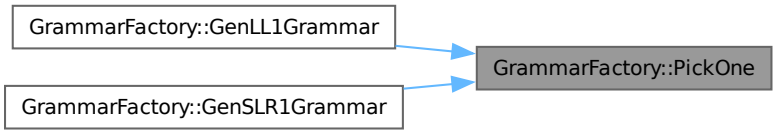
Returns

A randomly picked grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.2.25 RemoveLeftRecursion()

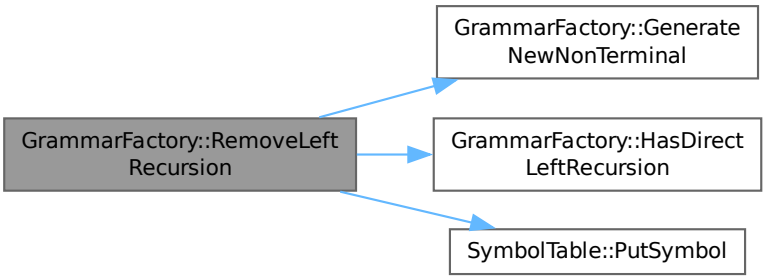
void GrammarFactory::RemoveLeftRecursion (
Grammar & grammar)

Removes direct left recursion in a grammar. A grammar has direct left recursion when one of its productions is A -> A a, where A is a non terminal symbol and ”a” the rest of the production. The procedure removes direct left recursion by adding a new non terminal. So, if the productions with left recursion are A -> A a | b, the result would be A -> b A'; A'-> a A' | EPSILON.

Parameters

grammar	The grammar to remove left recursion

Here is the call graph for this function:



Here is the caller graph for this function:

5.5.2.26 StartsWith()

bool GrammarFactory::StartsWith (
const production & prod,
const std::vector< std::string > & prefix)

Checks if a production starts with a given prefix.
This function determines whether the symbols in a production match the provided prefix sequence at the beginning. It is used during left factorization to identify productions that share a common prefix.

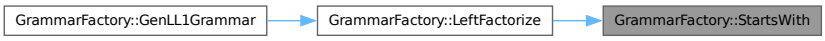
Parameters

prod	The production to check.
prefix	The sequence of symbols to compare against the beginning of the production.

Returns

true if the production starts with the prefix, false otherwise.

Here is the caller graph for this function:



5.5.3 Member Data Documentation

5.5.3.1 items

std::vector<FactoryItem> GrammarFactory::items

A vector of FactoryItem objects representing different level 1 grammar items created by the Init method.

5.5.3.2 non_terminal_alphabet_

std::vector<std::string> GrammarFactory::non_terminal_alphabet_

Initial value:

{"A", "B", "C", "D",
"E", "F", "G"}

A vector of non-terminal symbols (alphabet) used in the grammar.

5.5.3.3 terminal_alphabet_

std::vector<std::string> GrammarFactory::terminal_alphabet_

Initial value:

{"a", "b", "c", "d", "e", "f",
"g", "h", "i", "j", "k", "l"}

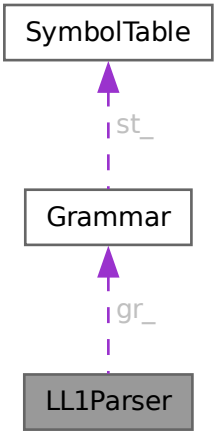
A vector of terminal symbols (alphabet) used in the grammar.
The documentation for this struct was generated from the following files:

- backend/grammar_factory.hpp
- backend/grammar_factory.cpp

5.6 LL1Parser Class Reference

#include <ll1_parser.hpp>

Collaboration diagram for LL1Parser:



Public Member Functions

- LL1Parser ()=default
- LL1Parser (Grammar gr)

  Constructs an LL1Parser with a grammar object and an input file.
- bool CreateLL1Table ()

  Creates the LL(1) parsing table for the grammar.
- void First (std::span< const std::string > rule, std::unordered_set< std::string > &result)

  Calculates the FIRST set for a given production rule in a grammar.
- void ComputeFirstSets ()

  Computes the FIRST sets for all non-terminal symbols in the grammar.
- void ComputeFollowSets ()

  Computes the FOLLOW sets for all non-terminal symbols in the grammar.
- std::unordered_set< std::string > Follow (const std::string &arg)

  Computes the FOLLOW set for a given non-terminal symbol in the grammar.
- std::unordered_set< std::string > PredictionSymbols (const std::string &antecedent, const std::vector< std::string > &consequent)

  Computes the prediction symbols for a given production rule.

Public Attributes

- ll1_table ll1_t_

  The LL(1) parsing table, mapping non-terminals and terminals to productions.
- Grammar gr_

  Grammar object associated with this parser.
- std::unordered_map< std::string, std::unordered_set< std::string > > first_sets_

  FIRST sets for each non-terminal in the grammar.
- std::unordered_map< std::string, std::unordered_set< std::string > > follow_sets_

  FOLLOW sets for each non-terminal in the grammar.

5.6.1 Constructor & Destructor Documentation

5.6.1.1 LL1Parser() [1/2]

LL1Parser::LL1Parser () [default]

5.6.1.2 LL1Parser() [2/2]

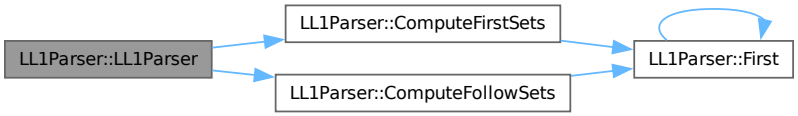
LL1Parser::LL1Parser (
Grammar gr) [explicit]

Constructs an LL1Parser with a grammar object and an input file.

Parameters

gr	Grammar object to parse with

Here is the call graph for this function:



5.6.2 Member Function Documentation

5.6.2.1 ComputeFirstSets()

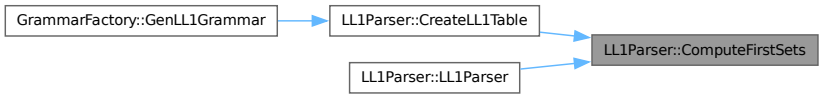
void LL1Parser::ComputeFirstSets ()

Computes the FIRST sets for all non-terminal symbols in the grammar.

This function calculates the FIRST set for each non-terminal symbol in the grammar by iteratively applying a least fixed-point algorithm. This approach ensures that the FIRST sets are fully populated by repeatedly expanding and updating the sets until no further changes occur (i.e., a fixed-point is reached).

Here is the call graph for this function:

Here is the caller graph for this function:



5.6.2.2 ComputeFollowSets()

void LL1Parser::ComputeFollowSets ()
Computes the FOLLOW sets for all non-terminal symbols in the grammar.
The FOLLOW set of a non-terminal symbol A contains all terminal symbols that can appear immediately after A in any sentential form derived from the grammar's start symbol. Additionally, if A can be the last symbol in a derivation, the end-of-input marker ($) is included in its FOLLOW set.
This function computes the FOLLOW sets using the following rules:

1. Initialize FOLLOW(S) = { $ }, where S is the start symbol.
2. For each production rule of the form A → B :
   - Add FIRST( ) (excluding) to FOLLOW(B).
   - If FIRST( ), add FOLLOW(A) to FOLLOW(B).
3. Repeat step 2 until no changes occur in any FOLLOW set.

The computed FOLLOW sets are cached in the follow_sets_ member variable for later use by the parser.

Note

This function assumes that the FIRST sets for all symbols have already been computed and are available in the first_sets_ member variable.

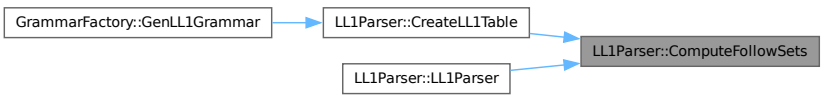
See also

First

follow_sets_

Here is the call graph for this function:

Here is the caller graph for this function:



5.6.2.3 CreateLL1Table()

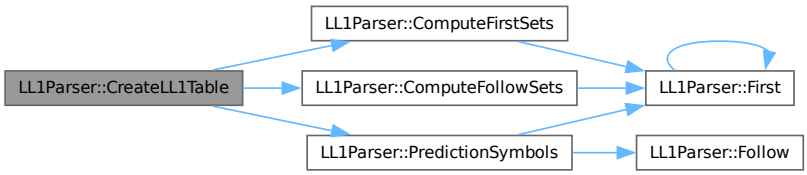
bool LL1Parser::CreateLL1Table ()
Creates the LL(1) parsing table for the grammar.
This function constructs the LL(1) parsing table by iterating over each production in the grammar and determining the appropriate cells for each non-terminal and director symbol (prediction symbol) combination. If the grammar is LL(1) compatible, each cell will contain at most one production, indicating no conflicts. If conflicts are found, the function will return false, signaling that the grammar is not LL(1).

- For each production rule A -> , the function calculates the director symbols using the director_↩ symbols function.
- It then fills the parsing table at the cell corresponding to the non-terminal A and each director symbol in the set.
- If a cell already contains a production, this indicates a conflict, meaning the grammar is not LL(1).

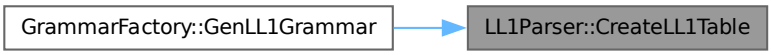
Returns

true if the table is created successfully, indicating the grammar is LL(1) compatible; false if any conflicts are detected, showing that the grammar does not meet LL(1) requirements.

Here is the call graph for this function:



Here is the caller graph for this function:



5.6.2.4 First()

void LL1Parser::First (
std::span< const std::string > rule,
std::unordered_set< std::string > & result)

Calculates the FIRST set for a given production rule in a grammar.
The FIRST set of a production rule contains all terminal symbols that can appear at the beginning of any string derived from that rule. If the rule can derive the empty string (epsilon), epsilon is included in the FIRST set.
This function computes the FIRST set by examining each symbol in the production rule:

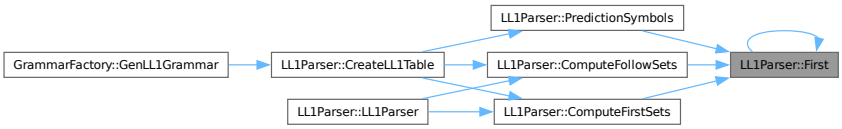
- If a terminal symbol is encountered, it is added directly to the FIRST set, as it is the starting symbol of some derivation.
- If a non-terminal symbol is encountered, its FIRST set is recursively computed and added to the result, excluding epsilon unless it is followed by another symbol that could also lead to epsilon.
- If the entire rule could derive epsilon (i.e., each symbol in the rule can derive epsilon), then epsilon is added to the FIRST set.

Parameters

rule	A span of strings representing the production rule for which to compute the FIRST set. Each string in the span is a symbol (either terminal or non-terminal).
result	A reference to an unordered set of strings where the computed FIRST set will be stored. The set will contain all terminal symbols that can start derivations of the rule, and possibly epsilon if the rule can derive an empty string.

Here is the call graph for this function:

Here is the caller graph for this function:



5.6.2.5 Follow()

std::unordered_set< std::string > LL1Parser::Follow (
const std::string & arg)

Computes the FOLLOW set for a given non-terminal symbol in the grammar.
The FOLLOW set for a non-terminal symbol includes all symbols that can appear immediately to the right of that symbol in any derivation, as well as any end-of-input markers if the symbol can appear at the end of derivations. FOLLOW sets are used in LL(1) parsing table construction to determine possible continuations after a non-terminal.

Parameters

arg	Non-terminal symbol for which to compute the FOLLOW set.

Returns

An unordered set of strings containing symbols that form the FOLLOW set for arg.

Here is the caller graph for this function:

5.6.2.6 PredictionSymbols()

std::unordered_set< std::string > LL1Parser::PredictionSymbols (
const std::string & antecedent,
const std::vector< std::string > & consequent)

Computes the prediction symbols for a given production rule.
The prediction symbols for a rule, determine the set of input symbols that can trigger this rule in the parsing table. This function calculates the prediction symbols based on the FIRST set of the consequent and, if epsilon (the empty symbol) is in the FIRST set, also includes the FOLLOW set of the antecedent.

- If the FIRST set of the consequent does not contain epsilon, the prediction symbols are simply the FIRST symbols of the consequent.
- If the FIRST set of the consequent contains epsilon, the prediction symbols are computed as (FIRST(consequent) - {epsilon}) FOLLOW(antecedent).

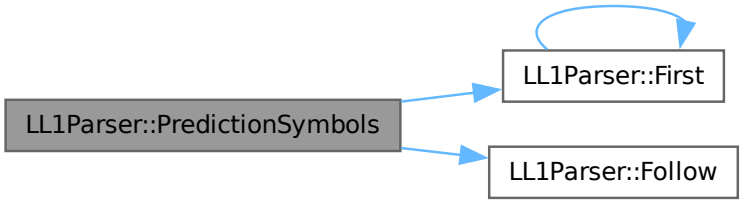
Parameters

antecedent	The left-hand side non-terminal symbol of the rule.
consequent	A vector of symbols on the right-hand side of the rule (production body).

Returns

An unordered set of strings containing the prediction symbols for the specified rule.

Here is the call graph for this function:



Here is the caller graph for this function:

5.6.3 Member Data Documentation

5.6.3.1 first_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > LL1Parser::first_sets_

FIRST sets for each non-terminal in the grammar.

5.6.3.2 follow_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > LL1Parser::follow_sets_

FOLLOW sets for each non-terminal in the grammar.

5.6.3.3 gr_

Grammar LL1Parser::gr_
Grammar object associated with this parser.

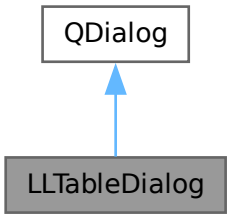
5.6.3.4 ll1_t_

ll1_table LL1Parser::ll1_t_
The LL(1) parsing table, mapping non-terminals and terminals to productions.
The documentation for this class was generated from the following files:

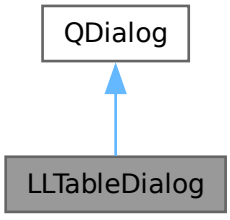
- backend/ll1_parser.hpp
- backend/ll1_parser.cpp

5.7 LLTableDialog Class Reference

Dialog for filling and submitting an LL(1) parsing table.
#include <lltabledialog.h>
Inheritance diagram for LLTableDialog:



Collaboration diagram for LLTableDialog:



Signals

- void submitted (const QVector< QVector< QString > > &data)

  Signal emitted when the user submits the table.

Public Member Functions

- LLTableDialog (const QStringList &rowHeaders, const QStringList &colHeaders, QWidget *parent, QVector< QVector< QString > > *initialData=nullptr)

  Constructs the LL(1) table dialog with given headers and optional initial data.
- QVector< QVector< QString > > getTableData () const

  Returns the contents of the table filled by the user.
- void setInitialData (const QVector< QVector< QString > > &data)

  Pre-fills the table with existing user data.
- void highlightIncorrectCells (const QList< QPair< int, int > > &coords)

  Highlights cells that are incorrect based on provided coordinates.

5.7.1 Detailed Description

Dialog for filling and submitting an LL(1) parsing table.
This class represents a dialog window that displays a table for users to complete the LL(1) parsing matrix. It provides functionality to initialize the table with data, retrieve the user's input, and highlight incorrect answers.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 LLTableDialog()

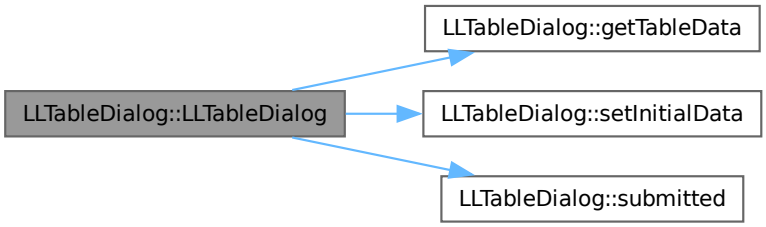
LLTableDialog::LLTableDialog (
const QStringList & rowHeaders,
const QStringList & colHeaders,
QWidget * parent,
QVector< QVector< QString > > * initialData = nullptr)

Constructs the LL(1) table dialog with given headers and optional initial data.

Parameters

rowHeaders	Row labels (non-terminal symbols).
colHeaders	Column labels (terminal symbols).
parent	Parent widget.
initialData	Optional initial table data to pre-fill cells.

Here is the call graph for this function:



5.7.3 Member Function Documentation

5.7.3.1 getTableData()

QVector< QVector< QString > > LLTableDialog::getTableData () const

Returns the contents of the table filled by the user.

Returns

A 2D vector representing the LL(1) table.

Here is the caller graph for this function:

5.7.3.2 highlightIncorrectCells()

void LLTableDialog::highlightIncorrectCells (
const QList< QPair< int, int > > & coords)

Highlights cells that are incorrect based on provided coordinates.

Parameters

coords	A list of (row, column) pairs to highlight as incorrect.

5.7.3.3 setInitialData()

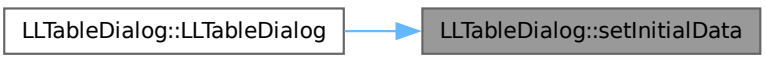
void LLTableDialog::setInitialData (
const QVector< QVector< QString > > & data)

Pre-fills the table with existing user data.
This is used to populate the table with a previous (possibly incorrect) answer when retrying a task or providing feedback.

Parameters

data	A 2D vector of strings representing the initial cell values.

Here is the caller graph for this function:



5.7.3.4 submitted

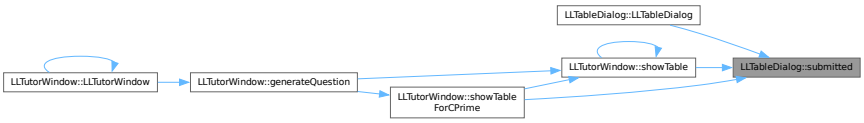
void LLTableDialog::submitted (
const QVector< QVector< QString > > & data) [signal]

Signal emitted when the user submits the table.

Parameters

data	The filled table data submitted by the user.

Here is the caller graph for this function:

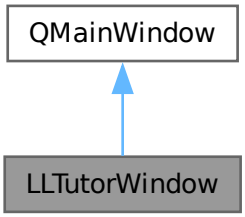


The documentation for this class was generated from the following files:

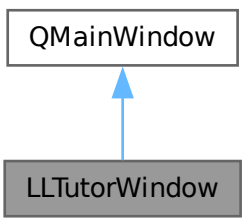
- lltabledialog.h
- lltabledialog.cpp

5.8 LLTutorWindow Class Reference

Main window for the LL(1) interactive tutoring mode in SyntaxTutor.
#include <lltutorwindow.h>
Inheritance diagram for LLTutorWindow:



Collaboration diagram for LLTutorWindow:



Classes

- struct TreeNode

  TreeNode structure used to build derivation trees.

Signals

- void sessionFinished (int cntRight, int cntWrong)

Public Member Functions

- LLTutorWindow (const Grammar &grammar, TutorialManager *tm=nullptr, QWidget *parent=nullptr)

  Constructs the LL(1) tutor window with a given grammar.
- ~LLTutorWindow ()
- QString generateQuestion ()

  Generates a question for the current state of the tutor.
- void updateState (bool isCorrect)

  Updates the tutor state after verifying user response.
- QString FormatGrammar (const Grammar &grammar)

  Formats a grammar for display in the chat interface.
- void addMessage (const QString &text, bool isUser)
- void addWidgetMessage (QWidget *widget)

  < Add text message to chat
- void exportConversationToPdf (const QString &filePath)

  < Add widget (e.g., table, tree)
- void showTable ()

  < Export chat to PDF
- void showTableForCPrime ()

  Display the full LL(1) table in C' ex.
- void updateProgressPanel ()
- void animateLabelPop (QLabel *label)
- void animateLabelColor (QLabel *label, const QColor &flashColor)
- void wrongAnimation ()

  Visual shake/flash for incorrect answer.
- void wrongUserResponseAnimation ()

  Animation specific to user chat input.
- void markLastUserIncorrect ()

  Marks last message as incorrect.
- void TeachFirstTree (const std::vector< std::string > &symbols, std::unordered_set< std::string > &first_set, int depth, std::unordered_set< std::string > &processing, QTreeWidgetItem *parent)
- std::unique_ptr< TreeNode > buildTreeNode (const std::vector< std::string > &symbols, std::unordered_set< std::string > &first_set, int depth, std::vector< std::pair< std::string, std::vector< std::string > > > &active_derivations)
- int computeSubtreeWidth (const std::unique_ptr< TreeNode > &node, int hSpacing)
- void drawTree (const std::unique_ptr< TreeNode > &root, QGraphicsScene *scene, QPointF pos, int hSpacing, int vSpacing)
- void showTreeGraphics (std::unique_ptr< TreeNode > root)
- bool verifyResponse (const QString &userResponse)
- bool verifyResponseForA (const QString &userResponse)
- bool verifyResponseForA1 (const QString &userResponse)
- bool verifyResponseForA2 (const QString &userResponse)
- bool verifyResponseForB (const QString &userResponse)
- bool verifyResponseForB1 (const QString &userResponse)
- bool verifyResponseForB2 (const QString &userResponse)

- bool verifyResponseForC ()
- QString solution (const std::string &state)
- QStringList solutionForA ()
- QString solutionForA1 ()
- QString solutionForA2 ()
- QSet< QString > solutionForB ()
- QSet< QString > solutionForB1 ()
- QSet< QString > solutionForB2 ()
- QString feedback ()
- QString feedbackForA ()
- QString feedbackForA1 ()
- QString feedbackForA2 ()
- QString feedbackForAPrime ()
- QString feedbackForB ()
- QString feedbackForB1 ()
- QString feedbackForB2 ()
- QString feedbackForBPrime ()
- QString feedbackForC ()
- QString feedbackForCPrime ()
- void feedbackForB1TreeWidget ()
- void feedbackForB1TreeGraphics ()
- QString TeachFollow (const QString &nt)
- QString TeachPredictionSymbols (const QString &ant, const production &conseq)
- QString TeachLL1Table ()
- void handleTableSubmission (const QVector< QVector< QString > > &raw, const QStringList &colHeaders)

Protected Member Functions

- void closeEvent (QCloseEvent *event) override
- bool eventFilter (QObject *obj, QEvent *event) override

5.8.1 Detailed Description

Main window for the LL(1) interactive tutoring mode in SyntaxTutor.
This class guides students through the construction and analysis of LL(1) parsing tables. It uses a finite-state sequence to present progressively more complex tasks, verifies user responses, provides corrective feedback, and supports visualizations like derivation trees.
The tutor is designed to teach the student how the LL(1) table is built, not just test it — including interactive tasks, animated feedback, and hints.
Key features include:

- Interactive question flow based on grammar analysis.
- Derivation tree generation (TeachFirst).
- Step-by-step verification of FIRST, FOLLOW, prediction symbols, and table entries.
- Exportable conversation log for grading or review.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 LLTutorWindow()

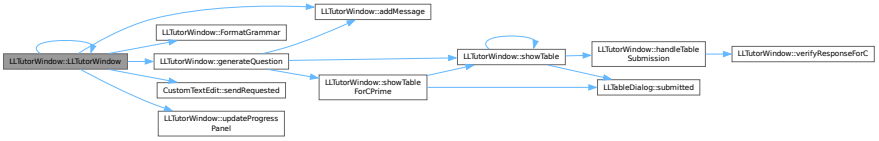
```
LLTutorWindow::LLTutorWindow (
            const Grammar & grammar,
            TutorialManager * tm = nullptr,
            QWidget * parent = nullptr)   [explicit]
```

Constructs the LL(1) tutor window with a given grammar.

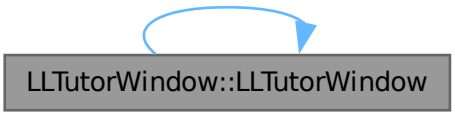
Parameters

grammar	The grammar to use during the session.
tm	Optional pointer to the tutorial manager (for help overlays).
parent	Parent widget.

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.2 ~LLTutorWindow()

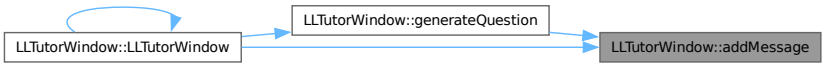
LLTutorWindow::~LLTutorWindow ()

5.8.3 Member Function Documentation

5.8.3.1 addMessage()

void LLTutorWindow::addMessage (
const QString & text,
bool isUser)

Here is the caller graph for this function:



5.8.3.2 addWidgetMessage()

void LLTutorWindow::addWidgetMessage (
QWidget * widget)

< Add text message to chat

Here is the caller graph for this function:

5.8.3.3 animateLabelColor()

void LLTutorWindow::animateLabelColor (
QLabel * label,
const QColor & flashColor)

5.8.3.4 animateLabelPop()

void LLTutorWindow::animateLabelPop (
QLabel * label)

5.8.3.5 buildTreeNode()

std::unique_ptr< LLTutorWindow::TreeNode > LLTutorWindow::buildTreeNode (
const std::vector< std::string > & symbols,
std::unordered_set< std::string > & first_set,
int depth,
std::vector< std::pair< std::string, std::vector< std::string > > > & active_derivations)

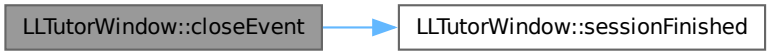
Here is the call graph for this function:

Here is the caller graph for this function:

5.8.3.6 closeEvent()

void LLTutorWindow::closeEvent (
QCloseEvent * event) [inline], [override], [protected]

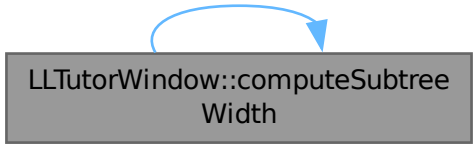
Here is the call graph for this function:



5.8.3.7 computeSubtreeWidth()

int LLTutorWindow::computeSubtreeWidth (
const std::unique_ptr< TreeNode > & node,
int hSpacing)

Here is the call graph for this function:

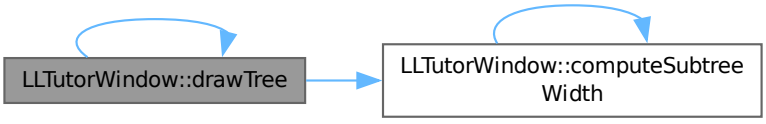


Here is the caller graph for this function:

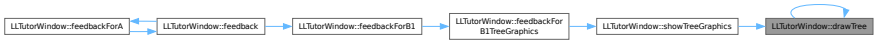
5.8.3.8 drawTree()

void LLTutorWindow::drawTree (
const std::unique_ptr< TreeNode > & root,
QGraphicsScene * scene,
QPointF pos,
int hSpacing,
int vSpacing)

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.3.9 eventFilter()

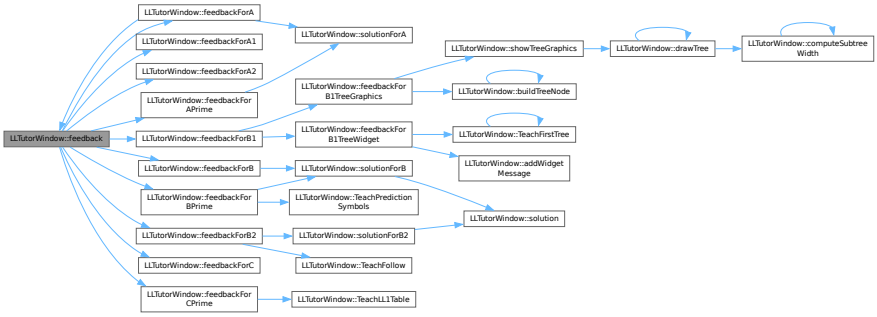
bool LLTutorWindow::eventFilter (
QObject * obj,
QEvent * event) [override], [protected]

5.8.3.10 exportConversationToPdf()

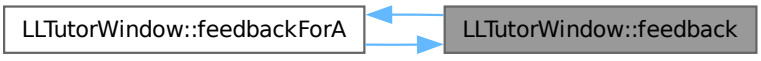
void LLTutorWindow::exportConversationToPdf (
const QString & filePath)
< Add widget (e.g., table, tree)

5.8.3.11 feedback()

QString LLTutorWindow::feedback ()
Here is the call graph for this function:



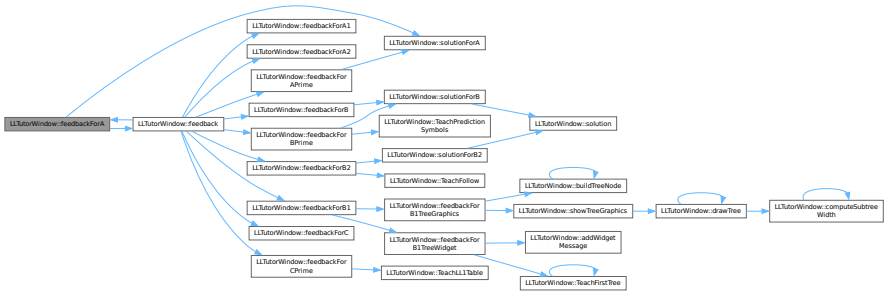
Here is the caller graph for this function:



5.8.3.12 feedbackForA()

QString LLTutorWindow::feedbackForA ()

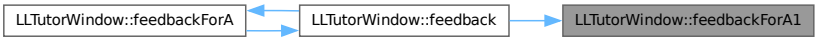
Here is the call graph for this function:



Here is the caller graph for this function:

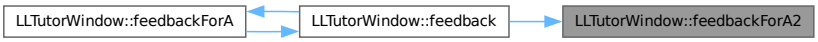
5.8.3.13 feedbackForA1()

QString LLTutorWindow::feedbackForA1 ()
Here is the caller graph for this function:



5.8.3.14 feedbackForA2()

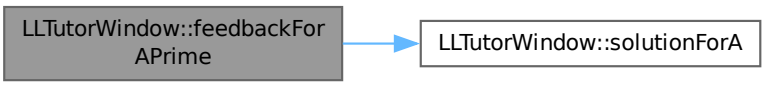
QString LLTutorWindow::feedbackForA2 ()
Here is the caller graph for this function:



5.8.3.15 feedbackForAPrime()

QString LLTutorWindow::feedbackForAPrime ()

Here is the call graph for this function:

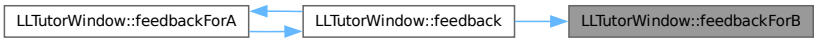


Here is the caller graph for this function:

5.8.3.16 feedbackForB()

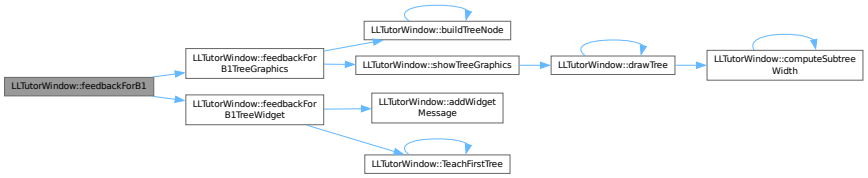
QString LLTutorWindow::feedbackForB ()
Here is the call graph for this function:

Here is the caller graph for this function:

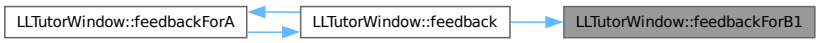


5.8.3.17 feedbackForB1()

QString LLTutorWindow::feedbackForB1 ()
Here is the call graph for this function:

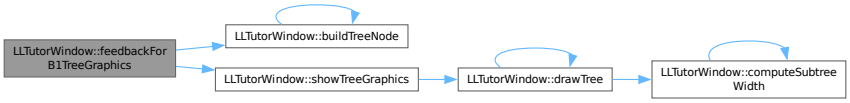


Here is the caller graph for this function:



5.8.3.18 feedbackForB1TreeGraphics()

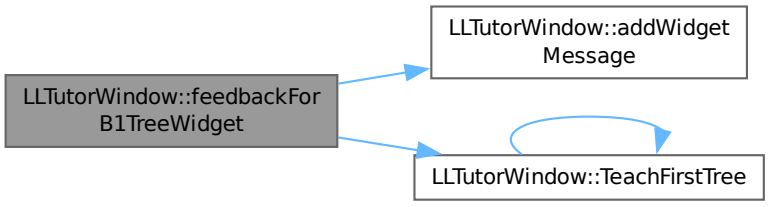
void LLTutorWindow::feedbackForB1TreeGraphics ()
Here is the call graph for this function:



Here is the caller graph for this function:

5.8.3.19 feedbackForB1TreeWidget()

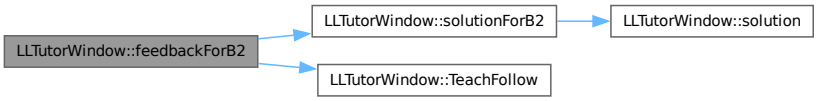
void LLTutorWindow::feedbackForB1TreeWidget ()
Here is the call graph for this function:



Here is the caller graph for this function:

5.8.3.20 feedbackForB2()

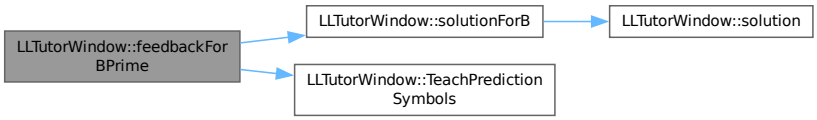
QString LLTutorWindow::feedbackForB2 ()
Here is the call graph for this function:



Here is the caller graph for this function:

5.8.3.21 feedbackForBPrime()

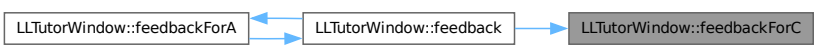
QString LLTutorWindow::feedbackForBPrime ()
Here is the call graph for this function:



Here is the caller graph for this function:

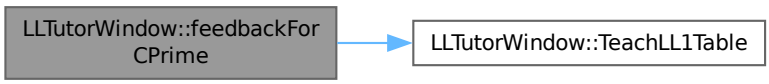
5.8.3.22 feedbackForC()

QString LLTutorWindow::feedbackForC ()
Here is the caller graph for this function:

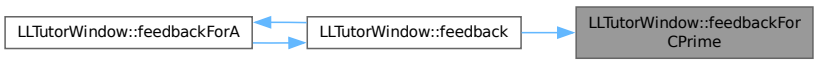


5.8.3.23 feedbackForCPrime()

QString LLTutorWindow::feedbackForCPrime ()
Here is the call graph for this function:



Here is the caller graph for this function:



5.8.3.24 FormatGrammar()

QString LLTutorWindow::FormatGrammar (
const Grammar & grammar)
Formats a grammar for display in the chat interface.

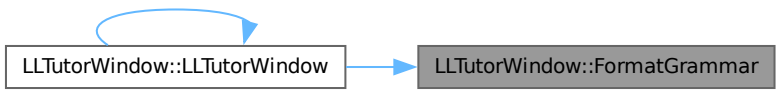
Parameters

grammar	The grammar to format.

Returns

A QString representation.

Here is the caller graph for this function:



5.8.3.25 generateQuestion()

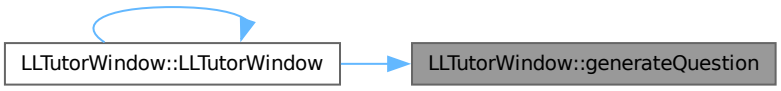
QString LLTutorWindow::generateQuestion ()
Generates a question for the current state of the tutor.

Returns

A formatted question string.

Here is the call graph for this function:

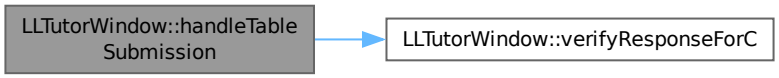
Here is the caller graph for this function:



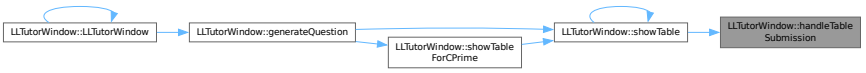
5.8.3.26 handleTableSubmission()

void LLTutorWindow::handleTableSubmission (
const QVector< QVector< QString > > & raw,
const QStringList & colHeaders)

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.3.27 markLastUserIncorrect()

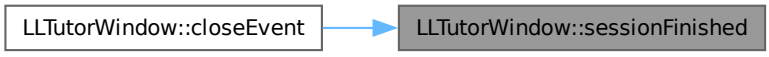
void LLTutorWindow::markLastUserIncorrect ()

Marks last message as incorrect.

5.8.3.28 sessionFinished

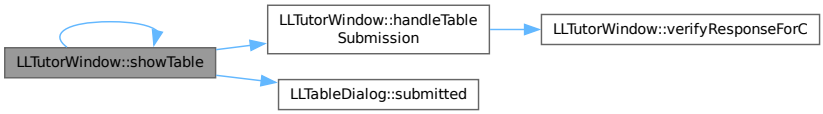
void LLTutorWindow::sessionFinished (
int cntRight,
int cntWrong) [signal]

Here is the caller graph for this function:

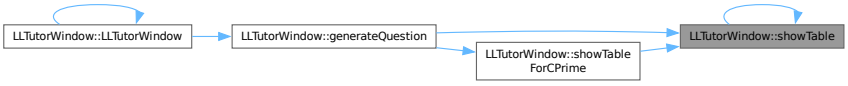


5.8.3.29 showTable()

void LLTutorWindow::showTable ()
< Export chat to PDF
Display the full LL(1) table in C ex. Here is the call graph for this function:

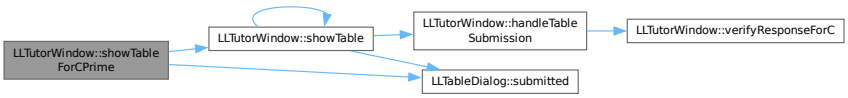


Here is the caller graph for this function:



5.8.3.30 showTableForCPrime()

void LLTutorWindow::showTableForCPrime ()
Display the full LL(1) table in C' ex.
Here is the call graph for this function:



Here is the caller graph for this function:

5.8.3.31 showTreeGraphics()

void LLTutorWindow::showTreeGraphics (
std::unique_ptr< TreeNode > root)

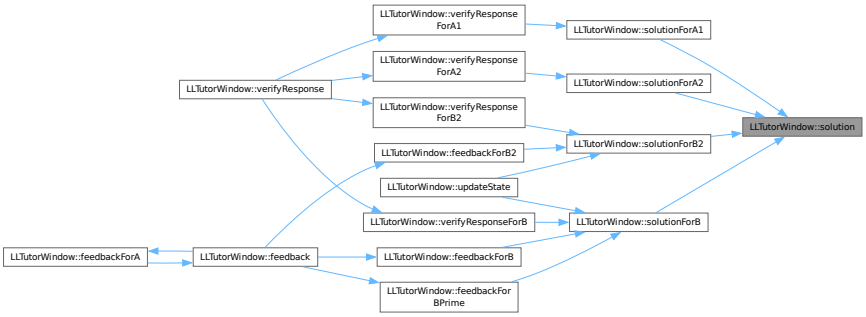
Here is the call graph for this function:

Here is the caller graph for this function:

5.8.3.32 solution()

QString LLTutorWindow::solution (
const std::string & state)

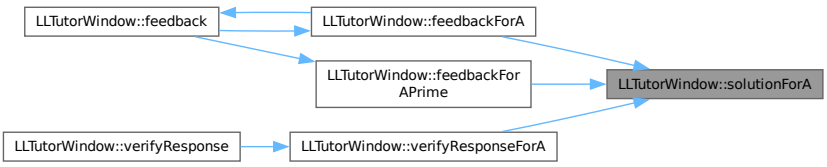
Here is the caller graph for this function:



5.8.3.33 solutionForA()

QStringList LLTutorWindow::solutionForA ()

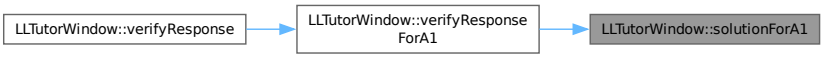
Here is the caller graph for this function:



5.8.3.34 solutionForA1()

QString LLTutorWindow::solutionForA1 ()
Here is the call graph for this function:

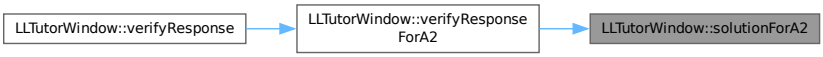
Here is the caller graph for this function:



5.8.3.35 solutionForA2()

QString LLTutorWindow::solutionForA2 ()
Here is the call graph for this function:

Here is the caller graph for this function:

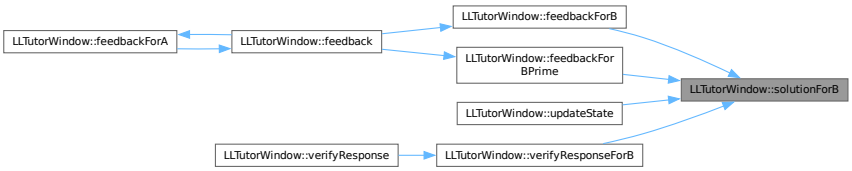


5.8.3.36 solutionForB()

QSet< QString > LLTutorWindow::solutionForB ()

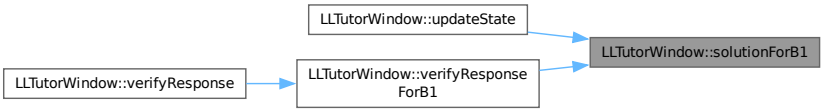
Here is the call graph for this function:

Here is the caller graph for this function:



5.8.3.37 solutionForB1()

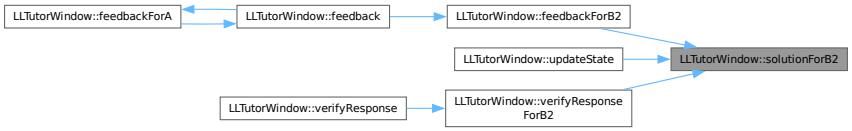
QSet< QString > LLTutorWindow::solutionForB1 ()
Here is the caller graph for this function:



5.8.3.38 solutionForB2()

QSet< QString > LLTutorWindow::solutionForB2 ()
Here is the call graph for this function:

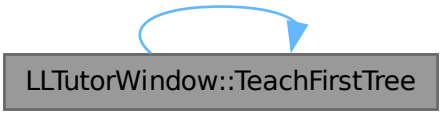
Here is the caller graph for this function:



5.8.3.39 TeachFirstTree()

void LLTutorWindow::TeachFirstTree (
const std::vector< std::string > & symbols,
std::unordered_set< std::string > & first_set,
int depth,
std::unordered_set< std::string > & processing,
QTreeWidgetItem * parent)

Here is the call graph for this function:

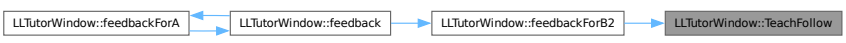


Here is the caller graph for this function:

5.8.3.40 TeachFollow()

QString LLTutorWindow::TeachFollow (
const QString & nt)

Here is the caller graph for this function:



5.8.3.41 TeachLL1Table()

QString LLTutorWindow::TeachLL1Table ()

Here is the caller graph for this function:

5.8.3.42 TeachPredictionSymbols()

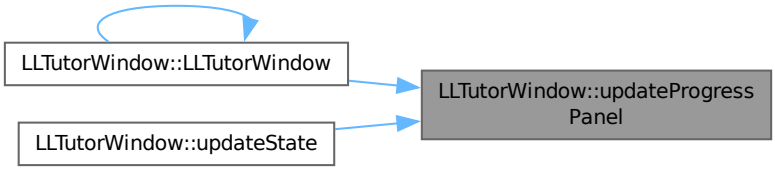
QString LLTutorWindow::TeachPredictionSymbols (
const QString & ant,
const production & conseq)

Here is the caller graph for this function:

5.8.3.43 updateProgressPanel()

void LLTutorWindow::updateProgressPanel ()

Here is the caller graph for this function:



5.8.3.44 updateState()

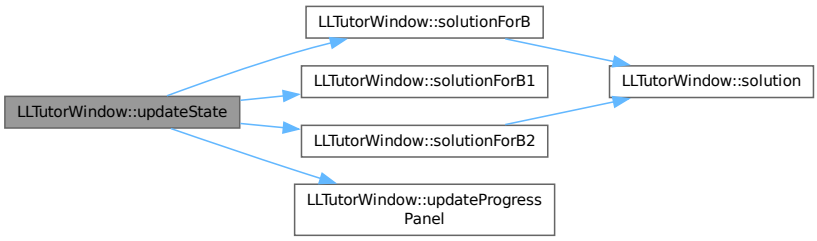
void LLTutorWindow::updateState (
bool isCorrect)

Updates the tutor state after verifying user response.

Parameters

isCorrect	Whether the user answered correctly.

Here is the call graph for this function:



5.8.3.45 verifyResponse()

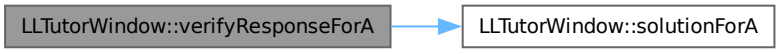
bool LLTutorWindow::verifyResponse (
const QString & userResponse)

Here is the call graph for this function:

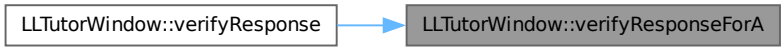
5.8.3.46 verifyResponseForA()

bool LLTutorWindow::verifyResponseForA (
const QString & userResponse)

Here is the call graph for this function:



Here is the caller graph for this function:

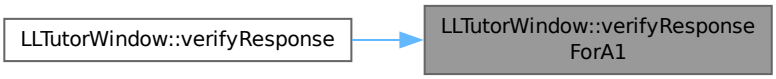


5.8.3.47 verifyResponseForA1()

bool LLTutorWindow::verifyResponseForA1 (
const QString & userResponse)

Here is the call graph for this function:

Here is the caller graph for this function:

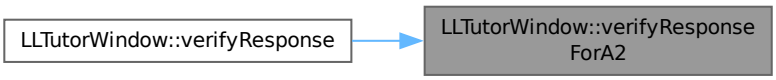


5.8.3.48 verifyResponseForA2()

bool LLTutorWindow::verifyResponseForA2 (
const QString & userResponse)

Here is the call graph for this function:

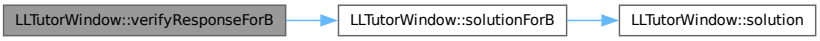
Here is the caller graph for this function:



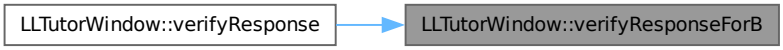
5.8.3.49 verifyResponseForB()

bool LLTutorWindow::verifyResponseForB (
const QString & userResponse)

Here is the call graph for this function:



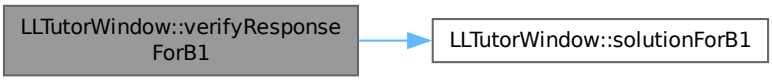
Here is the caller graph for this function:



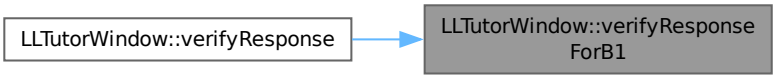
5.8.3.50 verifyResponseForB1()

bool LLTutorWindow::verifyResponseForB1 (
const QString & userResponse)

Here is the call graph for this function:



Here is the caller graph for this function:

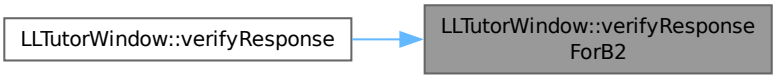


5.8.3.51 verifyResponseForB2()

bool LLTutorWindow::verifyResponseForB2 (
const QString & userResponse)

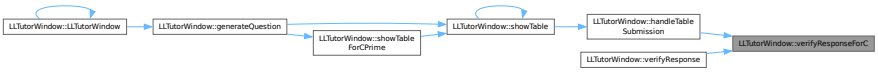
Here is the call graph for this function:

Here is the caller graph for this function:



5.8.3.52 verifyResponseForC()

bool LLTutorWindow::verifyResponseForC ()
Here is the caller graph for this function:



5.8.3.53 wrongAnimation()

void LLTutorWindow::wrongAnimation ()
Visual shake/flash for incorrect answer.

5.8.3.54 wrongUserResponseAnimation()

void LLTutorWindow::wrongUserResponseAnimation ()
Animation specific to user chat input.
The documentation for this class was generated from the following files:

- lltutorwindow.h
- lltutorwindow.cpp

5.9 Lr0Item Struct Reference

Represents an LR(0) item used in LR automata construction.
#include <lr0_item.hpp>

Public Member Functions

- Lr0Item (std::string antecedent, std::vector< std::string > consequent, std::string epsilon, std::string eol)

  Constructs an LR(0) item with the dot at position 0.
- Lr0Item (std::string antecedent, std::vector< std::string > consequent, unsigned int dot, std::string epsilon, std::string eol)

Constructs an LR(0) item with a custom dot position.

- std::string NextToDot () const

  Returns the symbol immediately after the dot, or empty if the dot is at the end.
- void PrintItem () const

  Prints the LR(0) item to the standard output in a human-readable format.
- std::string ToString () const

  Converts the item to a string representation, including the dot position.
- void AdvanceDot ()

  Advances the dot one position to the right.
- bool IsComplete () const

  Checks whether the dot has reached the end of the production.
- bool operator== (const Lr0Item &other) const

  Equality operator for comparing two LR(0) items.

Public Attributes

- std::string antecedent_

  The non-terminal on the left-hand side of the production.
- std::vector< std::string > consequent_

  The sequence of symbols on the right-hand side of the production.
- std::string epsilon_

  The symbol representing the empty string ( ).
- std::string eol_

  The symbol representing end-of-line or end-of-input ($).
- unsigned int dot_ = 0

  The position of the dot (·) in the production.

5.9.1 Detailed Description

Represents an LR(0) item used in LR automata construction.
An LR(0) item has a production of the form A → · , where the dot indicates the current parsing position. This structure tracks the antecedent (left-hand side), consequent (right-hand side), the dot position, and special symbols like EPSILON and end-of-line ($).

5.9.2 Constructor & Destructor Documentation

5.9.2.1 Lr0Item() [1/2]

```
Lr0Item::Lr0Item (
            std::string antecedent,
            std::vector< std::string > consequent,
            std::string epsilon,
            std::string eol)
```

Constructs an LR(0) item with the dot at position 0.

Parameters

antecedent	The left-hand side non-terminal.
consequent	The right-hand side of the production.
epsilon	The EPSILON symbol.
eol	The end-of-line symbol.

Here is the caller graph for this function:

5.9.2.2 Lr0Item() [2/2]

Lr0Item::Lr0Item (
std::string antecedent,
std::vector< std::string > consequent,
unsigned int dot,
std::string epsilon,
std::string eol)

Constructs an LR(0) item with a custom dot position.

Parameters

antecedent	The left-hand side non-terminal.
consequent	The right-hand side of the production.
dot	The position of the dot.
epsilon	The EPSILON symbol.
eol	The end-of-line symbol.

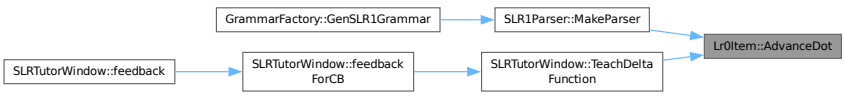
5.9.3 Member Function Documentation

5.9.3.1 AdvanceDot()

void Lr0Item::AdvanceDot ()

Advances the dot one position to the right.

Here is the caller graph for this function:



5.9.3.2 IsComplete()

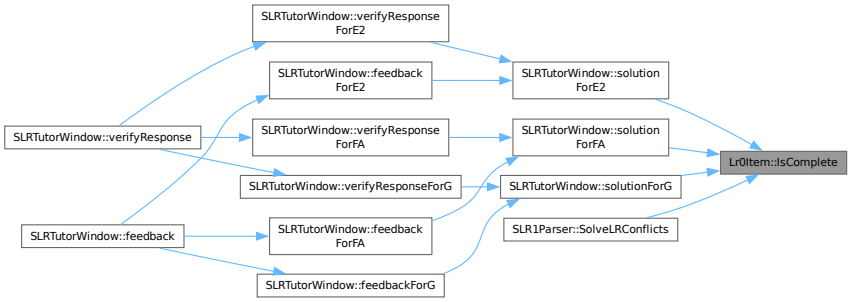
bool Lr0Item::IsComplete () const

Checks whether the dot has reached the end of the production.

Returns

true if the item is complete; false otherwise.

Here is the caller graph for this function:



5.9.3.3 NextToDot()

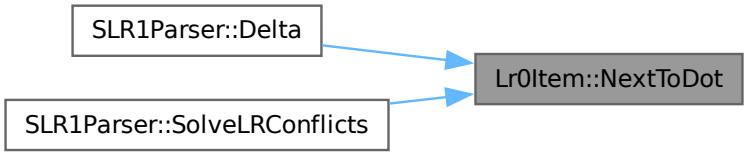
std::string Lr0Item::NextToDot () const

Returns the symbol immediately after the dot, or empty if the dot is at the end.

Returns

The symbol after the dot, or an empty string.

Here is the caller graph for this function:



5.9.3.4 operator==()

bool Lr0Item::operator== (
const Lr0Item & other) const

Equality operator for comparing two LR(0) items.

Parameters

other	The item to compare with.

Returns

true if both items are equal; false otherwise.

Here is the call graph for this function:

5.9.3.5 PrintItem()

void Lr0Item::PrintItem () const
Prints the LR(0) item to the standard output in a human-readable format.

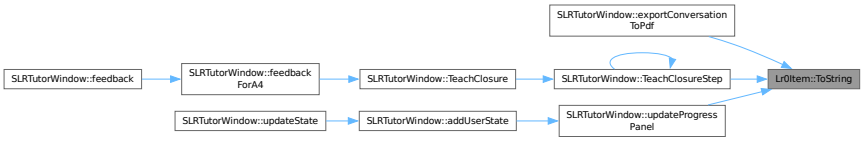
5.9.3.6 ToString()

std::string Lr0Item::ToString () const
Converts the item to a string representation, including the dot position.

Returns

A string representation of the item.

Here is the caller graph for this function:



5.9.4 Member Data Documentation

5.9.4.1 antecedent_

std::string Lr0Item::antecedent_
The non-terminal on the left-hand side of the production.

5.9.4.2 consequent_

std::vector<std::string> Lr0Item::consequent_
The sequence of symbols on the right-hand side of the production.

5.9.4.3 dot_

unsigned int Lr0Item::dot_ = 0
The position of the dot (·) in the production.

5.9.4.4 eol_

std::string Lr0Item::eol_
The symbol representing end-of-line or end-of-input ($).

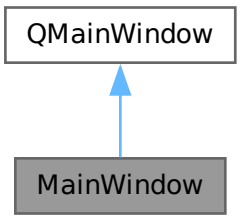
5.9.4.5 epsilon_

std::string Lr0Item::epsilon_
The symbol representing the empty string ( ).
The documentation for this struct was generated from the following files:

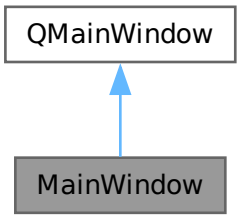
- backend/lr0_item.hpp
- backend/lr0_item.cpp

5.10 MainWindow Class Reference

Main application window of SyntaxTutor, managing levels, exercises, and UI state.
#include <mainwindow.h>
Inheritance diagram for MainWindow:



Collaboration diagram for MainWindow:



Signals

- void userLevelChanged (unsigned lvl)

  Emitted when the user's level changes.
- void userLevelUp (unsigned newLevel)

  Emitted when the user levels up.

Public Member Functions

- MainWindow (QWidget *parent=nullptr)

  Constructs the main window.

- ~MainWindow ()
  Destructor.
- unsigned thresholdFor (unsigned level)
  Returns the required score threshold to unlock a level.
- unsigned userLevel () const
  Returns the current user level.
- void setUserLevel (unsigned lvl)
  Sets the user level, clamping it to the allowed maximum.

Properties

- unsigned userLevel

5.10.1 Detailed Description

Main application window of SyntaxTutor, managing levels, exercises, and UI state.
This class serves as the central hub of the application. It handles level selection, navigation to LL(1) and SLR(1) exercises, tutorial management, settings persistence, and emits signals for user progress. It also includes UI logic for dynamic behavior like unlocking levels and changing language.

5.10.2 Constructor & Destructor Documentation

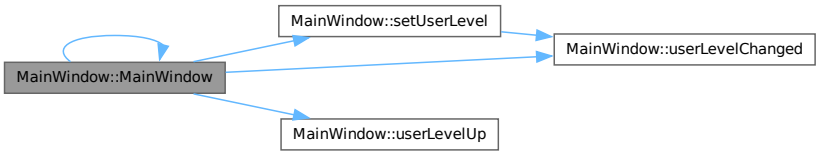
5.10.2.1 MainWindow()

MainWindow::MainWindow (
QWidget * parent = nullptr)
Constructs the main window.

Parameters

parent	Parent widget.

Here is the call graph for this function:



Here is the caller graph for this function:

5.10.2.2 ~MainWindow()

MainWindow::~MainWindow ()
Destructor.

5.10.3 Member Function Documentation

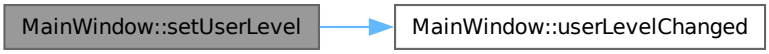
5.10.3.1 setUserLevel()

void MainWindow::setUserLevel (
unsigned lvl) [inline]
Sets the user level, clamping it to the allowed maximum.

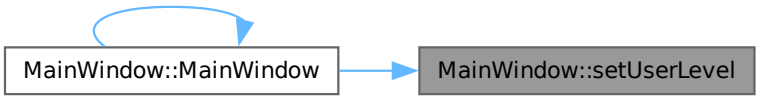
Parameters

lvl	New level to assign.

Here is the call graph for this function:



Here is the caller graph for this function:



5.10.3.2 thresholdFor()

unsigned MainWindow::thresholdFor (
unsigned level) [inline]
Returns the required score threshold to unlock a level.

Parameters

level	The level number.

Returns

The score needed to unlock the given level.

5.10.3.3 userLevel()

unsigned MainWindow::userLevel () const [inline]
Returns the current user level.

5.10.3.4 userLevelChanged

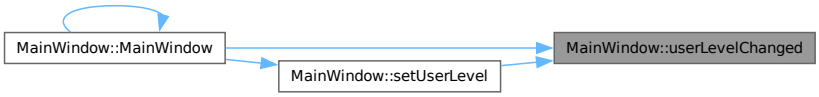
void MainWindow::userLevelChanged (
unsigned lvl) [signal]

Emitted when the user's level changes.

Parameters

lvl	New user level.

Here is the caller graph for this function:



5.10.3.5 userLevelUp

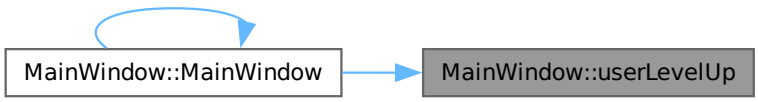
void MainWindow::userLevelUp (
unsigned newLevel) [signal]

Emitted when the user levels up.

Parameters

newLevel	The new level achieved.

Here is the caller graph for this function:



5.10.4 Property Documentation

5.10.4.1 userLevel

unsigned MainWindow::userLevel [read], [write]

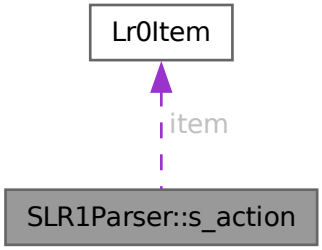
The documentation for this class was generated from the following files:

- mainwindow.h
- mainwindow.cpp

5.11 SLR1Parser::s_action Struct Reference

#include <slr1_parser.hpp>

Collaboration diagram for SLR1Parser::s_action:



Public Attributes

- const Lr0Item * item
- Action action

5.11.1 Member Data Documentation

5.11.1.1 action

Action SLR1Parser::s_action::action

5.11.1.2 item

const Lr0Item* SLR1Parser::s_action::item

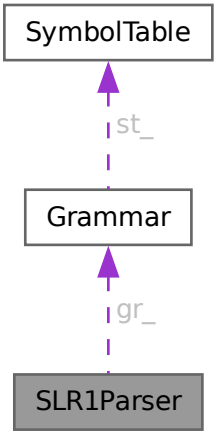
The documentation for this struct was generated from the following file:

- backend/slr1_parser.hpp

5.12 SLR1Parser Class Reference

Implements an SLR(1) parser for context-free grammars.
#include <slr1_parser.hpp>

Collaboration diagram for SLR1Parser:



Classes

- struct s_action

Public Types

- enum class Action { Shift , Reduce , Accept , Empty }

  Represents the possible actions in the SLR(1) parsing table.
- using action_table

  Represents the action table for the SLR(1) parser.
- using transition_table

  Represents the transition table for the SLR(1) parser.

Public Member Functions

- SLR1Parser ()=default
- SLR1Parser (Grammar gr)
- std::unordered_set< Lr0Item > AllItems () const

  Retrieves all LR(0) items in the grammar.
- void Closure (std::unordered_set< Lr0Item > &items)

  Computes the closure of a set of LR(0) items.
- void ClosureUtil (std::unordered_set< Lr0Item > &items, unsigned int size, std::unordered_set< std::string > &visited)

  Helper function for computing the closure of LR(0) items.
- std::unordered_set< Lr0Item > Delta (const std::unordered_set< Lr0Item > &items, const std::string &str)

  Computes the GOTO transition ( ) for a given set of LR(0) items and a symbol.
- bool SolveLRConflicts (const state &st)

  Resolves LR conflicts in a given state.
- void First (std::span< const std::string > rule, std::unordered_set< std::string > &result)

  Calculates the FIRST set for a given production rule in a grammar.
- void ComputeFirstSets ()

Computes the FIRST sets for all non-terminal symbols in the grammar.

- void ComputeFollowSets ()

  Computes the FOLLOW sets for all non-terminal symbols in the grammar.
- std::unordered_set< std::string > Follow (const std::string &arg)

  Computes the FOLLOW set for a given non-terminal symbol in the grammar.
- void MakeInitialState ()

  Creates the initial state of the parser's state machine.
- bool MakeParser ()

  Constructs the SLR(1) parsing tables (action and transition tables).
- std::string PrintItems (const std::unordered_set< Lr0Item > &items) const

  Returns a string representation of a set of LR(0) items.

Public Attributes

- Grammar gr_

  The grammar being processed by the parser.
- std::unordered_map< std::string, std::unordered_set< std::string > > first_sets_

  Cached FIRST sets for all symbols in the grammar.
- std::unordered_map< std::string, std::unordered_set< std::string > > follow_sets_

  Cached FOLLOW sets for all non-terminal symbols in the grammar.
- action_table actions_

  The action table used by the parser to determine shift/reduce actions.
- transition_table transitions_

  The transition table used by the parser to determine state transitions.
- std::unordered_set< state > states_

  The set of states in the parser's state machine.

5.12.1 Detailed Description

Implements an SLR(1) parser for context-free grammars.
This class builds an SLR(1) parsing table and LR(0) automaton from a given grammar. It provides methods for computing closure sets, GOTO transitions, constructing states, and performing syntax analysis using the generated table.

5.12.2 Member Typedef Documentation

5.12.2.1 action_table

using SLR1Parser::action_table
Initial value:

```
std::map<unsigned int, std::map<std::string, SLR1Parser::s_action»
```

Represents the action table for the SLR(1) parser.
The action table is a map that associates each state and input symbol with a specific action (Shift, Reduce, Accept, or Empty). It is used to determine the parser's behavior during the parsing process.
The table is structured as:

- Outer map: Keys are state IDs (unsigned int).
- Inner map: Keys are input symbols (std::string), and values are s_action structs representing the action to take.

5.12.2.2 transition_table

using SLR1Parser::transition_table
Initial value:

```
std::map<unsigned int, std::map<std::string, unsigned int»
```

Represents the transition table for the SLR(1) parser.
The transition table is a map that associates each state and symbol with the next state to transition to.
It is used to guide the parser's state transitions during the parsing process.
The table is structured as:

- Outer map: Keys are state IDs (unsigned int).
- Inner map: Keys are symbols (std::string), and values are the next state IDs (unsigned int).

5.12.3 Member Enumeration Documentation

5.12.3.1 Action

enum class SLR1Parser::Action [strong]
Represents the possible actions in the SLR(1) parsing table.
This enumeration defines the types of actions that can be taken by the parser during the parsing process:

- Shift: Shift the input symbol onto the stack and transition to a new state.
- Reduce: Reduce a production rule and pop symbols from the stack.
- Accept: Accept the input as a valid string in the grammar.
- Empty: No action is defined for the current state and input symbol.

Enumerator

Shift	
Reduce	
Accept	
Empty	

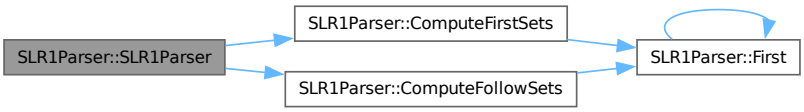
5.12.4 Constructor & Destructor Documentation

5.12.4.1 SLR1Parser() [1/2]

SLR1Parser::SLR1Parser () [default]

5.12.4.2 SLR1Parser() [2/2]

SLR1Parser::SLR1Parser (
Grammar gr) [explicit]
Here is the call graph for this function:



5.12.5 Member Function Documentation

5.12.5.1 AllItems()

std::unordered_set< Lr0Item > SLR1Parser::AllItems () const

Retrieves all LR(0) items in the grammar.

This function returns a set of all LR(0) items derived from the grammar's productions. Each LR(0) item represents a production with a marker indicating the current position in the production (e.g., A → •).

Returns

A set of all LR(0) items in the grammar.

5.12.5.2 Closure()

void SLR1Parser::Closure (
std::unordered_set< Lr0Item > & items)

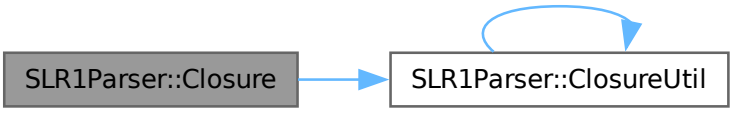
Computes the closure of a set of LR(0) items.

This function computes the closure of a given set of LR(0) items by adding all items that can be derived from the current items using the grammar's productions. The closure operation ensures that all possible derivations are considered when constructing the parser's states.

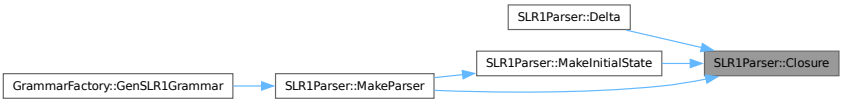
Parameters

items	The set of LR(0) items for which to compute the closure.

Here is the call graph for this function:



Here is the caller graph for this function:



5.12.5.3 ClosureUtil()

void SLR1Parser::ClosureUtil (
std::unordered_set< Lr0Item > & items,
unsigned int size,
std::unordered_set< std::string > & visited)

Helper function for computing the closure of LR(0) items.

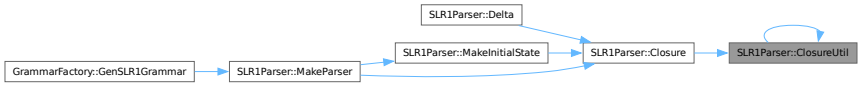
This function recursively computes the closure of a set of LR(0) items by adding items derived from non-terminal symbols. It avoids redundant work by tracking visited non-terminals and stopping when no new items are added.

Parameters

items	The set of LR(0) items being processed.
size	The size of the items set at the start of the current iteration.
visited	A set of non-terminals that have already been processed.

Here is the call graph for this function:

Here is the caller graph for this function:



5.12.5.4 ComputeFirstSets()

void SLR1Parser::ComputeFirstSets ()

Computes the FIRST sets for all non-terminal symbols in the grammar.

This function calculates the FIRST set for each non-terminal symbol in the grammar by iteratively applying a least fixed-point algorithm. This approach ensures that the FIRST sets are fully populated by repeatedly expanding and updating the sets until no further changes occur (i.e., a fixed-point is reached).

Here is the call graph for this function:

Here is the caller graph for this function:

5.12.5.5 ComputeFollowSets()

void SLR1Parser::ComputeFollowSets ()

Computes the FOLLOW sets for all non-terminal symbols in the grammar.

The FOLLOW set of a non-terminal symbol A contains all terminal symbols that can appear immediately after A in any sentential form derived from the grammar's start symbol. Additionally, if A can be the last symbol in a derivation, the end-of-input marker ($) is included in its FOLLOW set.

This function computes the FOLLOW sets using the following rules:

1. Initialize FOLLOW(S) = { $ }, where S is the start symbol.
2. For each production rule of the form A → B :
   - Add FIRST( ) (excluding) to FOLLOW(B).
   - If FIRST( ), add FOLLOW(A) to FOLLOW(B).
3. Repeat step 2 until no changes occur in any FOLLOW set.

The computed FOLLOW sets are cached in the follow_sets_ member variable for later use by the parser.

Note

This function assumes that the FIRST sets for all symbols have already been computed and are available in the first_sets_ member variable.

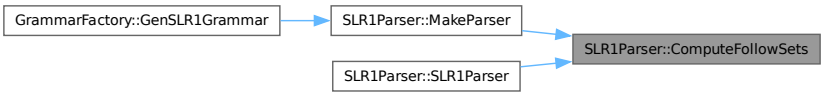
See also

First

follow_sets_

Here is the call graph for this function:

Here is the caller graph for this function:



5.12.5.6 Delta()

std::unordered_set< Lr0Item > SLR1Parser::Delta (
const std::unordered_set< Lr0Item > & items,
const std::string & str)

Computes the GOTO transition ( ) for a given set of LR(0) items and a symbol.
This function is equivalent to the (I, X) function in LR parsing, where it computes the set of items reached from a state I via symbol X.

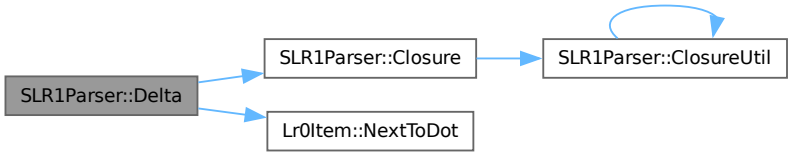
Parameters

items	The current set of LR(0) items (state).
str	The grammar symbol used for the transition.

Returns

The resulting item set after the GOTO transition.

Here is the call graph for this function:



5.12.5.7 First()

void SLR1Parser::First (
std::span< const std::string > rule,
std::unordered_set< std::string > & result)

Calculates the FIRST set for a given production rule in a grammar.
The FIRST set of a production rule contains all terminal symbols that can appear at the beginning of any string derived from that rule. If the rule can derive the empty string (epsilon), epsilon is included in the FIRST set.
This function computes the FIRST set by examining each symbol in the production rule:

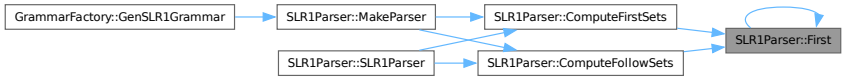
- If a terminal symbol is encountered, it is added directly to the FIRST set, as it is the starting symbol of some derivation.
- If a non-terminal symbol is encountered, its FIRST set is recursively computed and added to the result, excluding epsilon unless it is followed by another symbol that could also lead to epsilon.
- If the entire rule could derive epsilon (i.e., each symbol in the rule can derive epsilon), then epsilon is added to the FIRST set.

Parameters

rule	A span of strings representing the production rule for which to compute the FIRST set. Each string in the span is a symbol (either terminal or non-terminal).
result	A reference to an unordered set of strings where the computed FIRST set will be stored. The set will contain all terminal symbols that can start derivations of the rule, and possibly epsilon if the rule can derive an empty string.

Here is the call graph for this function:

Here is the caller graph for this function:



5.12.5.8 Follow()

std::unordered_set< std::string > SLR1Parser::Follow (
const std::string & arg)

Computes the FOLLOW set for a given non-terminal symbol in the grammar.
The FOLLOW set for a non-terminal symbol includes all symbols that can appear immediately to the right of that symbol in any derivation, as well as any end-of-input markers if the symbol can appear at the end of derivations. FOLLOW sets are used in LL(1) parsing table construction to determine possible continuations after a non-terminal.

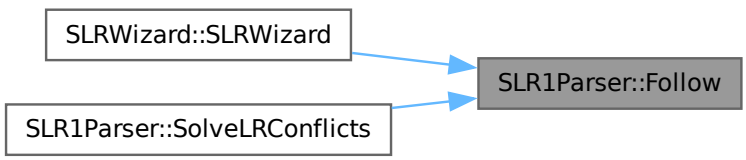
Parameters

arg	Non-terminal symbol for which to compute the FOLLOW set.

Returns

An unordered set of strings containing symbols that form the FOLLOW set for arg.

Here is the caller graph for this function:



5.12.5.9 MakeInitialState()

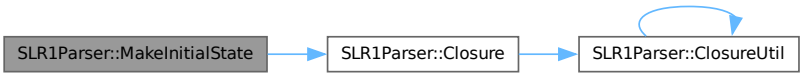
void SLR1Parser::MakeInitialState ()
Creates the initial state of the parser's state machine.
This function initializes the starting state of the parser by computing the closure of the initial set of LR(0) items derived from the grammar's start symbol. The initial state is added to the states_ set, and its transitions are prepared for further processing in the parser construction.

See also

states_

transitions_

Here is the call graph for this function:



Here is the caller graph for this function:

5.12.5.10 MakeParser()

bool SLR1Parser::MakeParser ()
Constructs the SLR(1) parsing tables (action and transition tables).
This function builds the SLR(1) parsing tables by computing the canonical collection of LR(0) items, generating the action and transition tables, and resolving conflicts (if any). It returns true if the grammar is SLR(1) and the tables are successfully constructed, or false if a conflict is detected that cannot be resolved.

Returns

true if the parsing tables are successfully constructed, false if the grammar is not SLR(1) or a conflict is encountered.

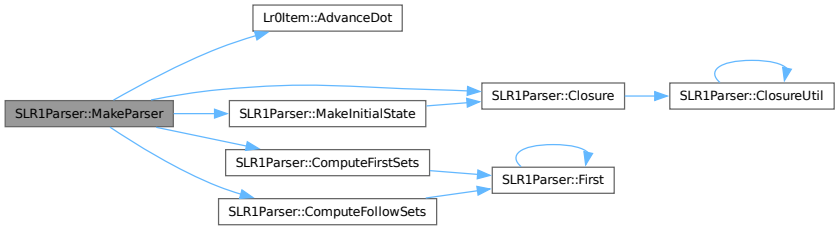
See also

actions_

transitions_

states_

Here is the call graph for this function:



Here is the caller graph for this function:

5.12.5.11 PrintItems()

std::string SLR1Parser::PrintItems (
const std::unordered_set< Lr0Item > & items) const

Returns a string representation of a set of LR(0) items.
This function converts a set of LR(0) items into a human-readable string, including dot positions, to help visualize parser states.

Parameters

items	The set of LR(0) items to print.

Returns

A formatted string representation of the items.

5.12.5.12 SolveLRConflicts()

bool SLR1Parser::SolveLRConflicts (
const state & st)

Resolves LR conflicts in a given state.
This function attempts to resolve shift/reduce or reduce/reduce conflicts in a given state using SLR(1) parsing rules. It checks the FOLLOW sets of non-terminals to determine the correct action and updates the action table accordingly.

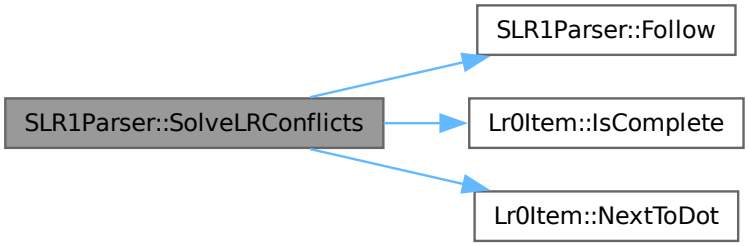
Parameters

st	The state in which to resolve conflicts.

Returns

true if all conflicts are resolved, false if an unresolvable conflict is detected.

Here is the call graph for this function:



5.12.6 Member Data Documentation

5.12.6.1 actions_

action_table SLR1Parser::actions_
The action table used by the parser to determine shift/reduce actions.

5.12.6.2 first_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > SLR1Parser::first_sets_
Cached FIRST sets for all symbols in the grammar.

5.12.6.3 follow_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > SLR1Parser::follow_sets_
Cached FOLLOW sets for all non-terminal symbols in the grammar.

5.12.6.4 gr_

Grammar SLR1Parser::gr_
The grammar being processed by the parser.

5.12.6.5 states_

std::unordered_set<state> SLR1Parser::states_
The set of states in the parser's state machine.

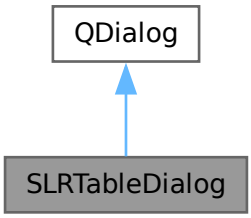
5.12.6.6 transitions_

transition_table SLR1Parser::transitions_
The transition table used by the parser to determine state transitions.
The documentation for this class was generated from the following files:

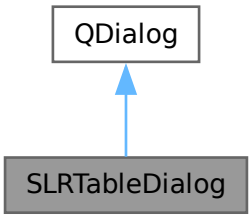
- backend/slr1_parser.hpp
- backend/slr1_parser.cpp

5.13 SLRTableDialog Class Reference

Dialog window for completing and submitting an SLR(1) parsing table.
#include <slrtabledialog.h>
Inheritance diagram for SLRTableDialog:



Collaboration diagram for SLRTableDialog:



Public Member Functions

- SLRTableDialog (int rowCount, int colCount, const QStringList &colHeaders, QWidget *parent=nullptr, QVector< QVector< QString > > *initialData=nullptr)

  Constructs the SLR(1) table dialog.
- QVector< QVector< QString > > getTableData () const

  Retrieves the content of the table after user interaction.
- void setInitialData (const QVector< QVector< QString > > &data)

  Fills the table with existing data.

5.13.1 Detailed Description

Dialog window for completing and submitting an SLR(1) parsing table.
This class displays a table-based UI for students to fill in the ACTION and GOTO parts of the SLR(1) parsing table. It supports initializing the table with data, retrieving user input, and integrating with correction logic in tutorial or challenge mode.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 SLRTableDialog()

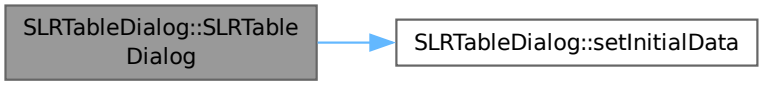
```
SLRTableDialog::SLRTableDialog (
            int rowCount,
            int colCount,
            const QStringList & colHeaders,
            QWidget * parent = nullptr,
            QVector< QVector< QString > > * initialData = nullptr)
```

Constructs the SLR(1) table dialog.

Parameters

rowCount	Number of rows (usually equal to number of LR(0) states).
colCount	Number of columns (symbols = terminals + non-terminals).
colHeaders	Header labels for the columns.
parent	Parent widget.
initialData	Optional initial data to pre-fill the table.

Here is the call graph for this function:



5.13.3 Member Function Documentation

5.13.3.1 getTableData()

```
QVector< QVector< QString > > SLRTableDialog::getTableData () const
```

Retrieves the content of the table after user interaction.

Returns

A 2D vector representing the current table values.

5.13.3.2 setInitialData()

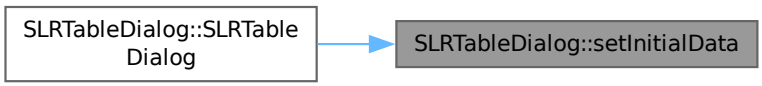
```
void SLRTableDialog::setInitialData (
            const QVector< QVector< QString > > & data)
```

Fills the table with existing data.
This method is used to show a previous user submission (e.g., during retries or feedback).

Parameters

data	2D vector containing the table data to display.

Here is the caller graph for this function:

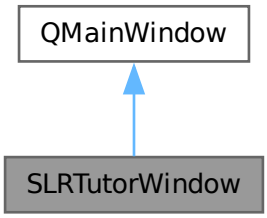


The documentation for this class was generated from the following files:

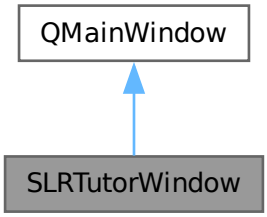
- slrtabledialog.h
- slrtabledialog.cpp

5.14 SLRTutorWindow Class Reference

Main window for the SLR(1) interactive tutoring mode in SyntaxTutor.
#include <slrtutorwindow.h>
Inheritance diagram for SLRTutorWindow:



Collaboration diagram for SLRTutorWindow:



Signals

- void sessionFinished (int cntRight, int cntWrong)

Public Member Functions

- SLRTutorWindow (const Grammar &g, TutorialManager *tm=nullptr, QWidget *parent=nullptr)

  Constructs the SLR(1) tutor window with a given grammar.
- ~SLRTutorWindow ()
- QString generateQuestion ()

  Generates a new question for the current tutor state.
- void updateState (bool isCorrect)

  Updates tutor state based on whether the last answer was correct.
- QString FormatGrammar (const Grammar &grammar)
- void fillSortedGrammar ()

  < Utility for displaying grammar
- void addMessage (const QString &text, bool isUser)

  < Prepares grammar in display-friendly format
- void exportConversationToPdf (const QString &filePath)

  < Add message to chat
- void showTable ()

  < Export full interaction
- void launchSLRWizard ()

  < Render SLR(1) table
- void updateProgressPanel ()
- void addUserState (unsigned id)

  < Refresh visual progress
- void addUserTransition (unsigned fromId, const std::string &symbol, unsigned toId)

  < Register a user-created state
- void animateLabelPop (QLabel *label)
- void animateLabelColor (QLabel *label, const QColor &flashColor)
- void wrongAnimation ()
- void wrongUserResponseAnimation ()
- void markLastUserIncorrect ()
- bool verifyResponse (const QString &userResponse)
- bool verifyResponseForA (const QString &userResponse)
- bool verifyResponseForA1 (const QString &userResponse)
- bool verifyResponseForA2 (const QString &userResponse)
- bool verifyResponseForA3 (const QString &userResponse)
- bool verifyResponseForA4 (const QString &userResponse)
- bool verifyResponseForB (const QString &userResponse)
- bool verifyResponseForC (const QString &userResponse)
- bool verifyResponseForCA (const QString &userResponse)
- bool verifyResponseForCB (const QString &userResponse)
- bool verifyResponseForD (const QString &userResponse)
- bool verifyResponseForD1 (const QString &userResponse)
- bool verifyResponseForD2 (const QString &userResponse)
- bool verifyResponseForE (const QString &userResponse)
- bool verifyResponseForE1 (const QString &userResponse)
- bool verifyResponseForE2 (const QString &userResponse)
- bool verifyResponseForF (const QString &userResponse)
- bool verifyResponseForFA (const QString &userResponse)
- bool verifyResponseForG (const QString &userResponse)
- bool verifyResponseForH ()

- QString solution (const std::string &state)
- std::unordered_set< Lr0Item > solutionForA ()
- QString solutionForA1 ()
- QString solutionForA2 ()
- std::vector< std::pair< std::string, std::vector< std::string > > > solutionForA3 ()
- std::unordered_set< Lr0Item > solutionForA4 ()
- unsigned solutionForB ()
- unsigned solutionForC ()
- QStringList solutionForCA ()
- std::unordered_set< Lr0Item > solutionForCB ()
- QString solutionForD ()
- QString solutionForD1 ()
- QString solutionForD2 ()
- std::ptrdiff_t solutionForE ()
- QSet< unsigned > solutionForE1 ()
- QMap< unsigned, unsigned > solutionForE2 ()
- QSet< unsigned > solutionForF ()
- QSet< QString > solutionForFA ()
- QSet< QString > solutionForG ()
- QString feedback ()
- QString feedbackForA ()
- QString feedbackForA1 ()
- QString feedbackForA2 ()
- QString feedbackForA3 ()
- QString feedbackForA4 ()
- QString feedbackForAPrime ()
- QString feedbackForB ()
- QString feedbackForB1 ()
- QString feedbackForB2 ()
- QString feedbackForBPrime ()
- QString feedbackForC ()
- QString feedbackForCA ()
- QString feedbackForCB ()
- QString feedbackForD ()
- QString feedbackForD1 ()
- QString feedbackForD2 ()
- QString feedbackForDPrime ()
- QString feedbackForE ()
- QString feedbackForE1 ()
- QString feedbackForE2 ()
- QString feedbackForF ()
- QString feedbackForFA ()
- QString feedbackForG ()
- QString TeachDeltaFunction (const std::unordered_set< Lr0Item > &items, const QString &symbol)
- void TeachClosureStep (std::unordered_set< Lr0Item > &items, unsigned int size, std::unordered_set< std::string > &visited, int depth, QString &output)
- QString TeachClosure (const std::unordered_set< Lr0Item > &initialItems)

Protected Member Functions

- void closeEvent (QCloseEvent *event) override

5.14.1 Detailed Description

Main window for the SLR(1) interactive tutoring mode in SyntaxTutor.
This class implements an interactive, step-by-step tutorial to teach students how to construct SLR(1) parsing tables, including closure, GOTO, automaton construction, FOLLOW sets, and the final table.
It supports animated feedback, pedagogical guidance, error correction, and export of the tutoring session.
The tutor follows a finite-state flow (StateSlr) to structure learning, with corrective explanations and automatic evaluation at each step.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 SLRTutorWindow()

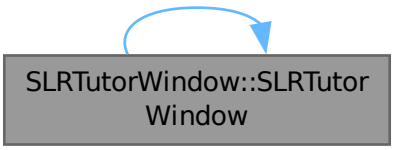
SLRTutorWindow::SLRTutorWindow (
const Grammar & g,
TutorialManager * tm = nullptr,
QWidget * parent = nullptr) [explicit]

Constructs the SLR(1) tutor window with a given grammar.

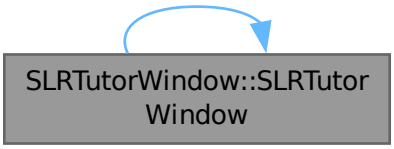
Parameters

g	The grammar used for the session.
tm	Optional pointer to the tutorial manager (for guided tour).
parent	Parent widget.

Here is the call graph for this function:



Here is the caller graph for this function:



5.14.2.2 ~SLRTutorWindow()

SLRTutorWindow::~SLRTutorWindow ()

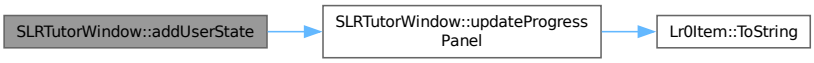
5.14.3 Member Function Documentation

5.14.3.1 addMessage()

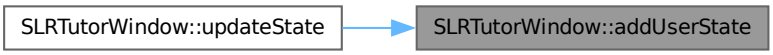
void SLRTutorWindow::addMessage (
const QString & text,
bool isUser)
< Prepares grammar in display-friendly format

5.14.3.2 addUserState()

void SLRTutorWindow::addUserState (
unsigned id)
< Refresh visual progress
Here is the call graph for this function:

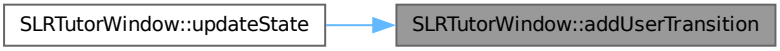


Here is the caller graph for this function:



5.14.3.3 addUserTransition()

void SLRTutorWindow::addUserTransition (
unsigned fromId,
const std::string & symbol,
unsigned toId)
< Register a user-created state
Here is the caller graph for this function:



5.14.3.4 animateLabelColor()

void SLRTutorWindow::animateLabelColor (
QLabel * label,
const QColor & flashColor)

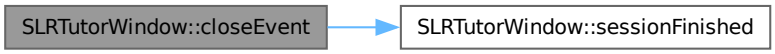
5.14.3.5 animateLabelPop()

void SLRTutorWindow::animateLabelPop (
QLabel * label)

5.14.3.6 closeEvent()

void SLRTutorWindow::closeEvent (
QCloseEvent * event) [inline], [override], [protected]

Here is the call graph for this function:



5.14.3.7 exportConversationToPdf()

void SLRTutorWindow::exportConversationToPdf (
const QString & filePath)

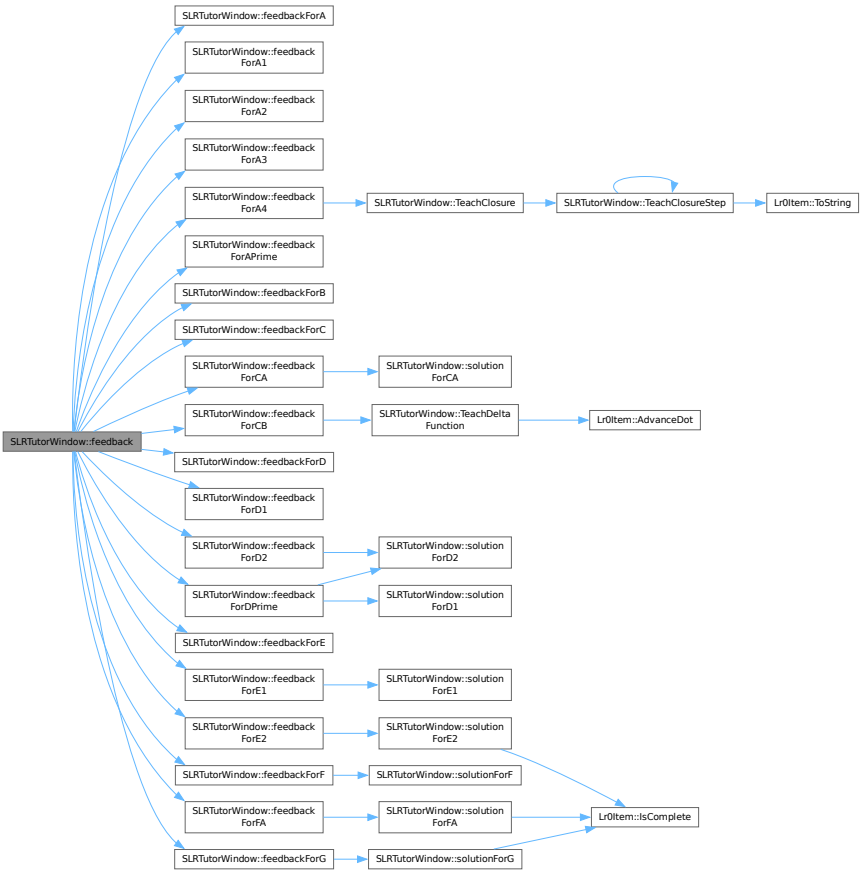
< Add message to chat

Here is the call graph for this function:

5.14.3.8 feedback()

QString SLRTutorWindow::feedback ()

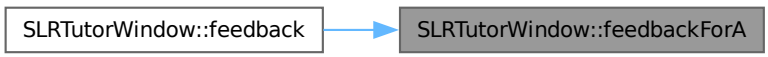
Here is the call graph for this function:



5.14.3.9 feedbackForA()

QString SLRTutorWindow::feedbackForA ()

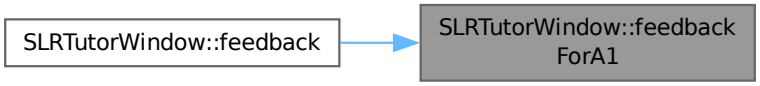
Here is the caller graph for this function:



5.14.3.10 feedbackForA1()

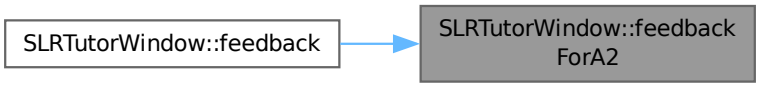
QString SLRTutorWindow::feedbackForA1 ()

Here is the caller graph for this function:



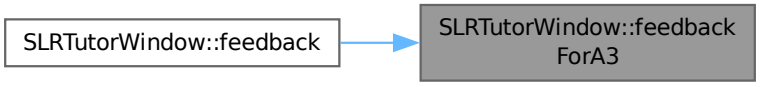
5.14.3.11 feedbackForA2()

QString SLRTutorWindow::feedbackForA2 ()
Here is the caller graph for this function:



5.14.3.12 feedbackForA3()

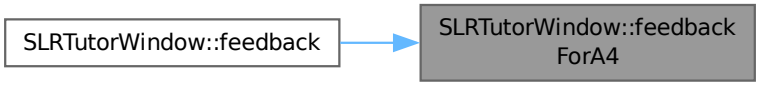
QString SLRTutorWindow::feedbackForA3 ()
Here is the caller graph for this function:



5.14.3.13 feedbackForA4()

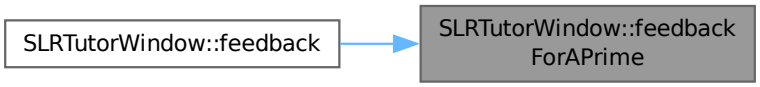
QString SLRTutorWindow::feedbackForA4 ()
Here is the call graph for this function:

Here is the caller graph for this function:



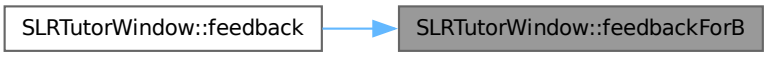
5.14.3.14 feedbackForAPrime()

QString SLRTutorWindow::feedbackForAPrime ()
Here is the caller graph for this function:



5.14.3.15 feedbackForB()

QString SLRTutorWindow::feedbackForB ()
Here is the caller graph for this function:



5.14.3.16 feedbackForB1()

QString SLRTutorWindow::feedbackForB1 ()

5.14.3.17 feedbackForB2()

QString SLRTutorWindow::feedbackForB2 ()

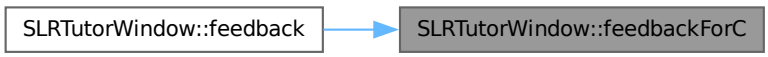
5.14.3.18 feedbackForBPrime()

QString SLRTutorWindow::feedbackForBPrime ()

5.14.3.19 feedbackForC()

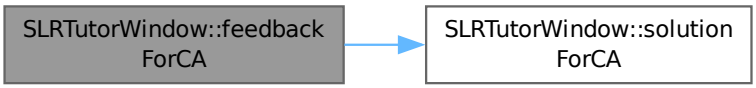
QString SLRTutorWindow::feedbackForC ()

Here is the caller graph for this function:

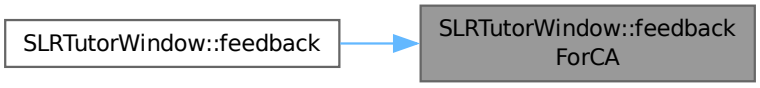


5.14.3.20 feedbackForCA()

QString SLRTutorWindow::feedbackForCA ()
Here is the call graph for this function:



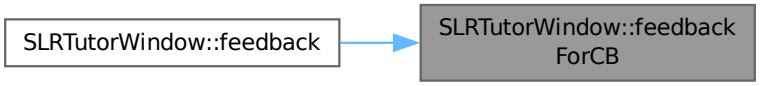
Here is the caller graph for this function:



5.14.3.21 feedbackForCB()

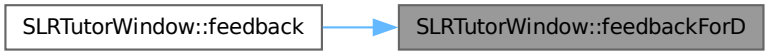
QString SLRTutorWindow::feedbackForCB ()
Here is the call graph for this function:

Here is the caller graph for this function:



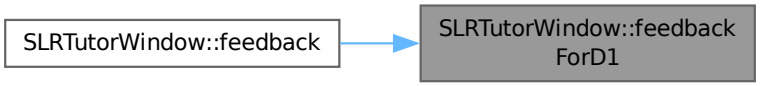
5.14.3.22 feedbackForD()

QString SLRTutorWindow::feedbackForD ()
Here is the caller graph for this function:



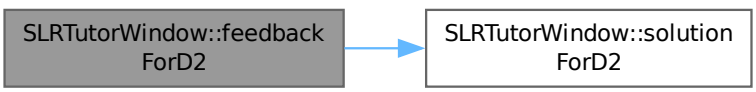
5.14.3.23 feedbackForD1()

QString SLRTutorWindow::feedbackForD1 ()
Here is the caller graph for this function:

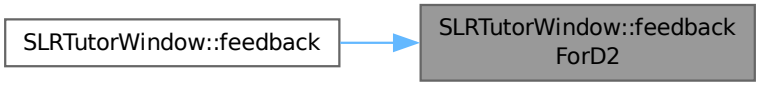


5.14.3.24 feedbackForD2()

QString SLRTutorWindow::feedbackForD2 ()
Here is the call graph for this function:

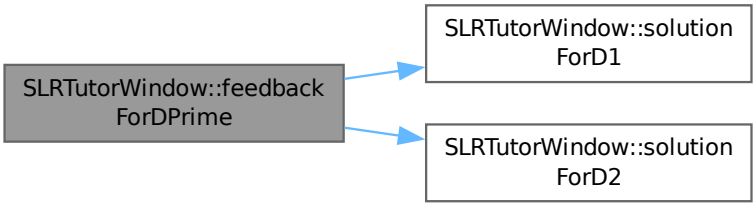


Here is the caller graph for this function:

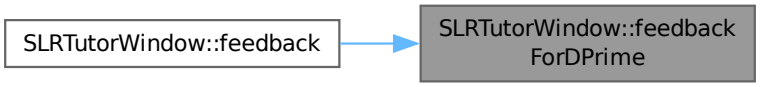


5.14.3.25 feedbackForDPrime()

QString SLRTutorWindow::feedbackForDPrime ()
Here is the call graph for this function:

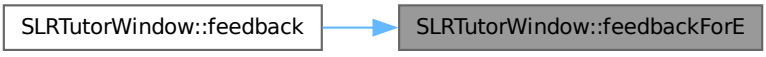


Here is the caller graph for this function:



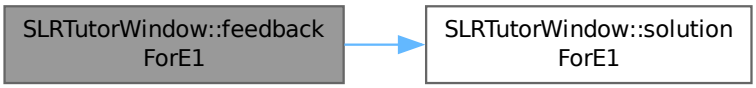
5.14.3.26 feedbackForE()

QString SLRTutorWindow::feedbackForE ()
Here is the caller graph for this function:

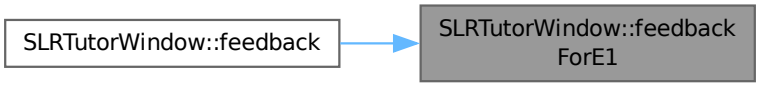


5.14.3.27 feedbackForE1()

QString SLRTutorWindow::feedbackForE1 ()
Here is the call graph for this function:

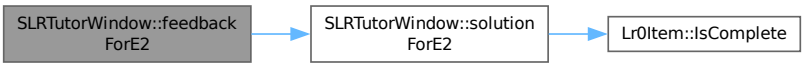


Here is the caller graph for this function:

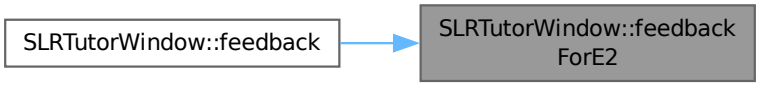


5.14.3.28 feedbackForE2()

QString SLRTutorWindow::feedbackForE2 ()
Here is the call graph for this function:



Here is the caller graph for this function:

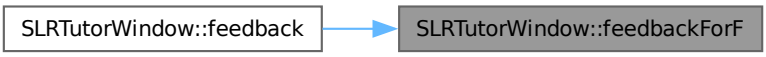


5.14.3.29 feedbackForF()

QString SLRTutorWindow::feedbackForF ()

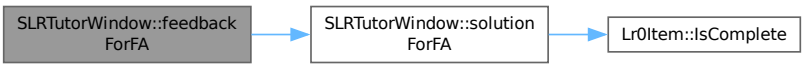
Here is the call graph for this function:

Here is the caller graph for this function:

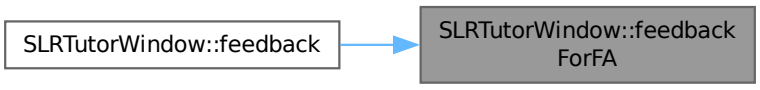


5.14.3.30 feedbackForFA()

QString SLRTutorWindow::feedbackForFA ()
Here is the call graph for this function:

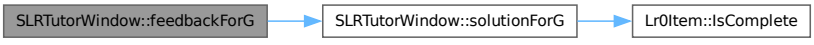


Here is the caller graph for this function:

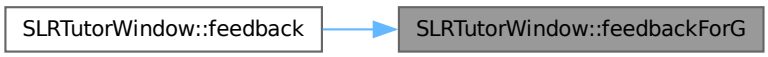


5.14.3.31 feedbackForG()

QString SLRTutorWindow::feedbackForG ()
Here is the call graph for this function:



Here is the caller graph for this function:



5.14.3.32 fillSortedGrammar()

void SLRTutorWindow::fillSortedGrammar ()
< Utility for displaying grammar

5.14.3.33 FormatGrammar()

QString SLRTutorWindow::FormatGrammar (
const Grammar & grammar)

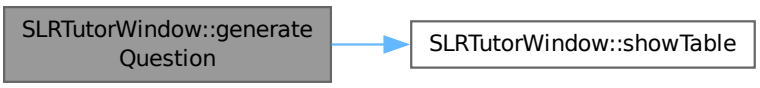
5.14.3.34 generateQuestion()

QString SLRTutorWindow::generateQuestion ()
Generates a new question for the current tutor state.

Returns

The formatted question string.

Here is the call graph for this function:



5.14.3.35 launchSLRWizard()

void SLRTutorWindow::launchSLRWizard ()
< Render SLR(1) table

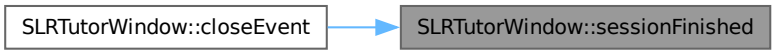
5.14.3.36 markLastUserIncorrect()

void SLRTutorWindow::markLastUserIncorrect ()

5.14.3.37 sessionFinished

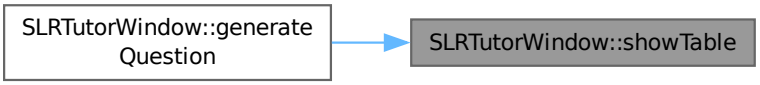
void SLRTutorWindow::sessionFinished (
int cntRight,
int cntWrong) [signal]

Here is the caller graph for this function:



5.14.3.38 showTable()

void SLRTutorWindow::showTable ()
< Export full interaction
Here is the caller graph for this function:

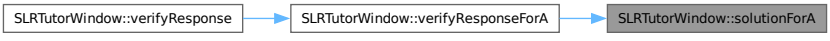


5.14.3.39 solution()

QString SLRTutorWindow::solution (
const std::string & state)

5.14.3.40 solutionForA()

std::unordered_set< Lr0Item > SLRTutorWindow::solutionForA ()
Here is the caller graph for this function:



5.14.3.41 solutionForA1()

QString SLRTutorWindow::solutionForA1 ()
Here is the caller graph for this function:

5.14.3.42 solutionForA2()

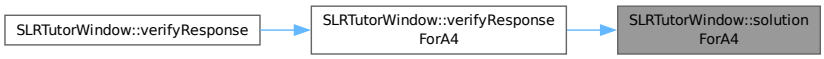
QString SLRTutorWindow::solutionForA2 ()
Here is the caller graph for this function:

5.14.3.43 solutionForA3()

std::vector< std::pair< std::string, std::vector< std::string > > > SLRTutorWindow::solutionForA3 ()
Here is the caller graph for this function:

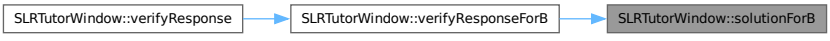
5.14.3.44 solutionForA4()

std::unordered_set< Lr0Item > SLRTutorWindow::solutionForA4 ()
Here is the caller graph for this function:



5.14.3.45 solutionForB()

unsigned SLRTutorWindow::solutionForB ()
Here is the caller graph for this function:



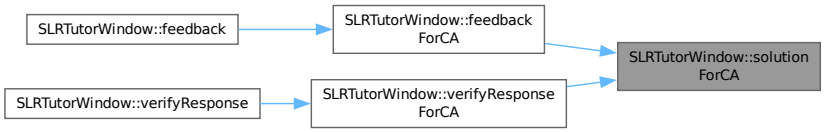
5.14.3.46 solutionForC()

unsigned SLRTutorWindow::solutionForC ()

Here is the caller graph for this function:

5.14.3.47 solutionForCA()

QStringList SLRTutorWindow::solutionForCA ()
Here is the caller graph for this function:

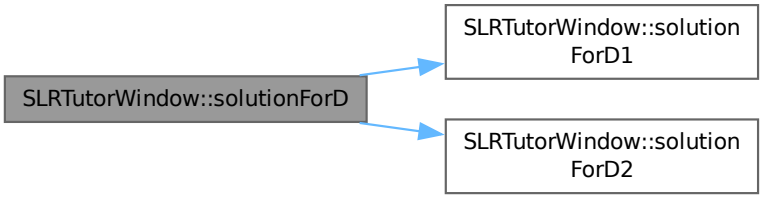


5.14.3.48 solutionForCB()

std::unordered_set< Lr0Item > SLRTutorWindow::solutionForCB ()
Here is the caller graph for this function:

5.14.3.49 solutionForD()

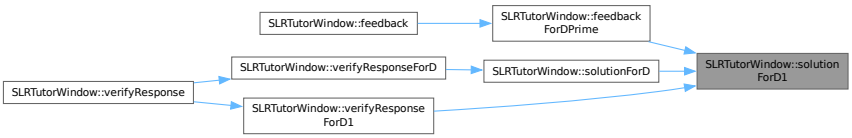
QString SLRTutorWindow::solutionForD ()
Here is the call graph for this function:



Here is the caller graph for this function:

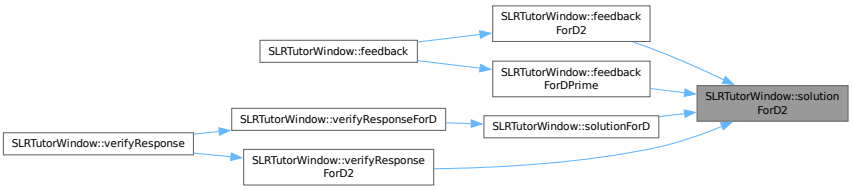
5.14.3.50 solutionForD1()

QString SLRTutorWindow::solutionForD1 ()
Here is the caller graph for this function:



5.14.3.51 solutionForD2()

QString SLRTutorWindow::solutionForD2 ()
Here is the caller graph for this function:



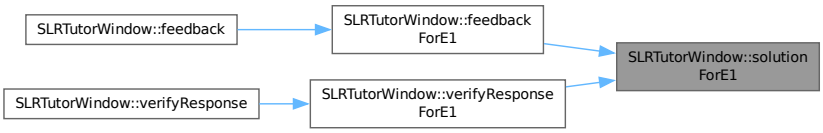
5.14.3.52 solutionForE()

std::ptrdiff_t SLRTutorWindow::solutionForE ()
Here is the caller graph for this function:

5.14.3.53 solutionForE1()

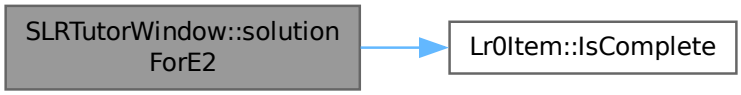
QSet< unsigned > SLRTutorWindow::solutionForE1 ()

Here is the caller graph for this function:

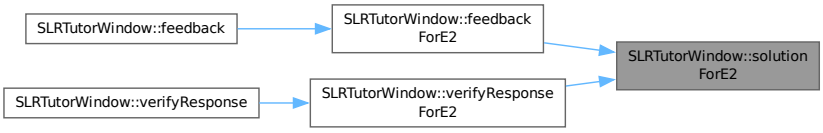


5.14.3.54 solutionForE2()

QMap< unsigned, unsigned > SLRTutorWindow::solutionForE2 ()
Here is the call graph for this function:

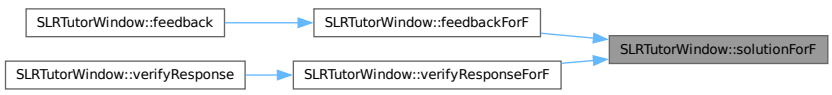


Here is the caller graph for this function:



5.14.3.55 solutionForF()

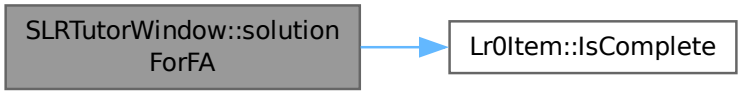
QSet< unsigned > SLRTutorWindow::solutionForF ()
Here is the caller graph for this function:



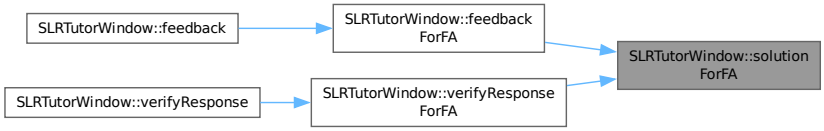
5.14.3.56 solutionForFA()

QSet< QString > SLRTutorWindow::solutionForFA ()

Here is the call graph for this function:



Here is the caller graph for this function:



5.14.3.57 solutionForG()

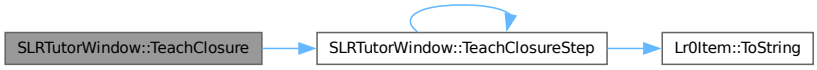
QSet< QString > SLRTutorWindow::solutionForG ()
Here is the call graph for this function:

Here is the caller graph for this function:

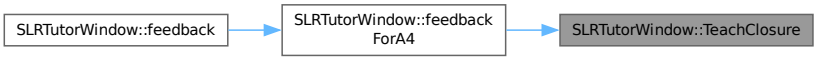
5.14.3.58 TeachClosure()

QString SLRTutorWindow::TeachClosure (
const std::unordered_set< Lr0Item > & initialItems)

Here is the call graph for this function:



Here is the caller graph for this function:



5.14.3.59 TeachClosureStep()

void SLRTutorWindow::TeachClosureStep (
std::unordered_set< Lr0Item > & items,
unsigned int size,
std::unordered_set< std::string > & visited,
int depth,
QString & output)

Here is the call graph for this function:

Here is the caller graph for this function:

5.14.3.60 TeachDeltaFunction()

QString SLRTutorWindow::TeachDeltaFunction (
const std::unordered_set< Lr0Item > & items,
const QString & symbol)

Here is the call graph for this function:

Here is the caller graph for this function:

5.14.3.61 updateProgressPanel()

void SLRTutorWindow::updateProgressPanel ()
Here is the call graph for this function:

Here is the caller graph for this function:

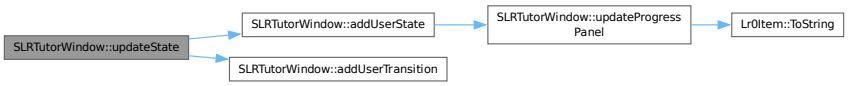
5.14.3.62 updateState()

void SLRTutorWindow::updateState (
bool isCorrect)
Updates tutor state based on whether the last answer was correct.

Parameters

isCorrect	Whether the user's answer was correct.

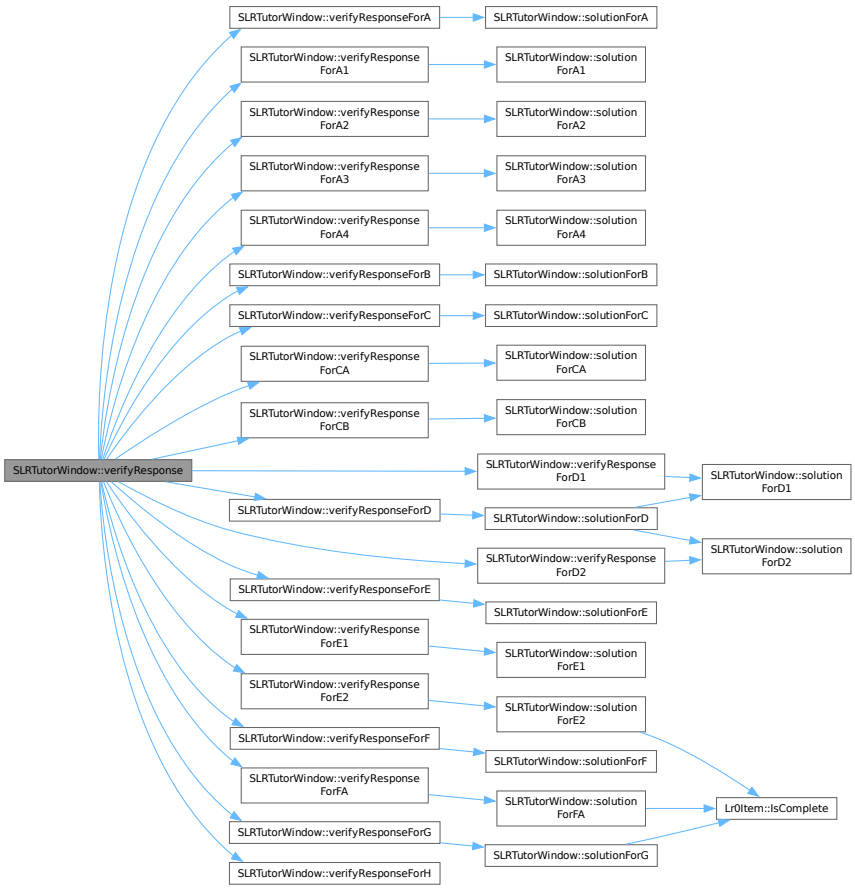
Here is the call graph for this function:



5.14.3.63 verifyResponse()

bool SLRTutorWindow::verifyResponse (
const QString & userResponse)

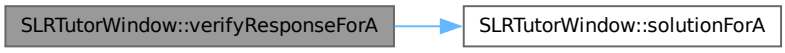
Here is the call graph for this function:



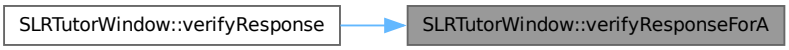
5.14.3.64 verifyResponseForA()

bool SLRTutorWindow::verifyResponseForA (
const QString & userResponse)

Here is the call graph for this function:



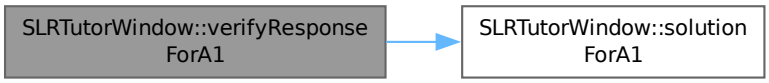
Here is the caller graph for this function:



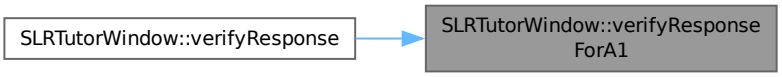
5.14.3.65 verifyResponseForA1()

bool SLRTutorWindow::verifyResponseForA1 (
const QString & userResponse)

Here is the call graph for this function:



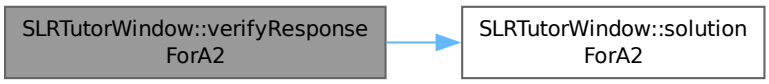
Here is the caller graph for this function:



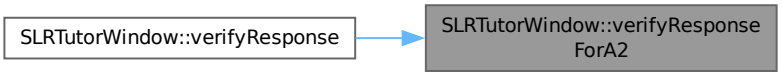
5.14.3.66 verifyResponseForA2()

bool SLRTutorWindow::verifyResponseForA2 (
const QString & userResponse)

Here is the call graph for this function:



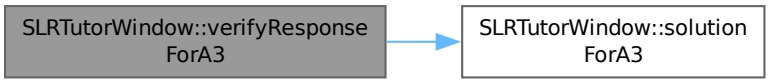
Here is the caller graph for this function:



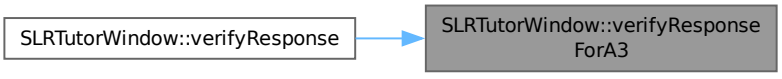
5.14.3.67 verifyResponseForA3()

bool SLRTutorWindow::verifyResponseForA3 (
const QString & userResponse)

Here is the call graph for this function:



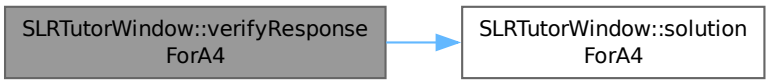
Here is the caller graph for this function:



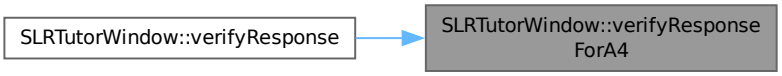
5.14.3.68 verifyResponseForA4()

bool SLRTutorWindow::verifyResponseForA4 (
const QString & userResponse)

Here is the call graph for this function:



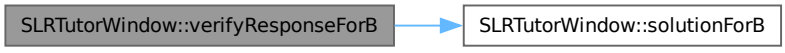
Here is the caller graph for this function:



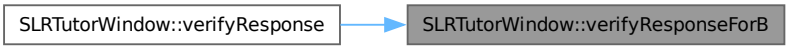
5.14.3.69 verifyResponseForB()

bool SLRTutorWindow::verifyResponseForB (
const QString & userResponse)

Here is the call graph for this function:



Here is the caller graph for this function:

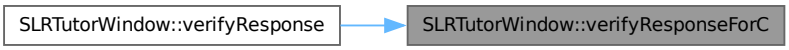


5.14.3.70 verifyResponseForC()

bool SLRTutorWindow::verifyResponseForC (
const QString & userResponse)

Here is the call graph for this function:

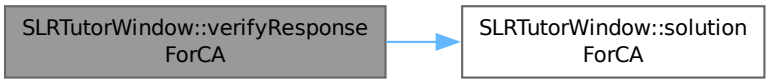
Here is the caller graph for this function:



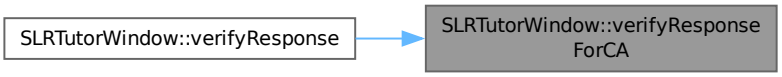
5.14.3.71 verifyResponseForCA()

bool SLRTutorWindow::verifyResponseForCA (
const QString & userResponse)

Here is the call graph for this function:



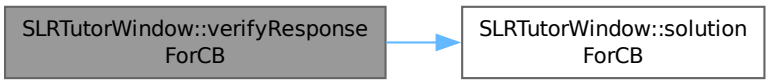
Here is the caller graph for this function:



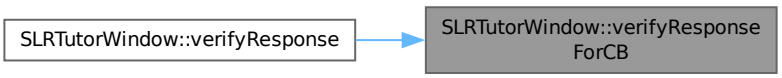
5.14.3.72 verifyResponseForCB()

bool SLRTutorWindow::verifyResponseForCB (
const QString & userResponse)

Here is the call graph for this function:



Here is the caller graph for this function:

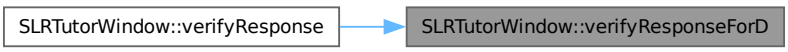


5.14.3.73 verifyResponseForD()

bool SLRTutorWindow::verifyResponseForD (
const QString & userResponse)

Here is the call graph for this function:

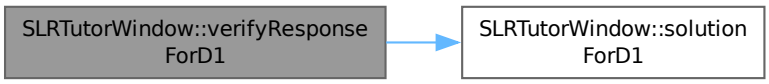
Here is the caller graph for this function:



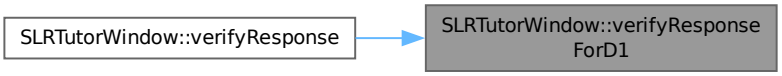
5.14.3.74 verifyResponseForD1()

bool SLRTutorWindow::verifyResponseForD1 (
const QString & userResponse)

Here is the call graph for this function:



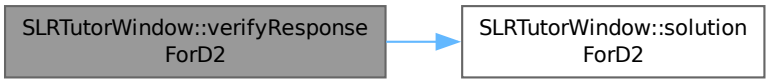
Here is the caller graph for this function:



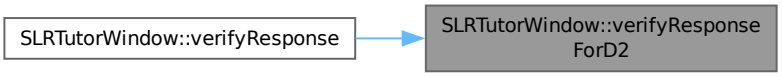
5.14.3.75 verifyResponseForD2()

bool SLRTutorWindow::verifyResponseForD2 (
const QString & userResponse)

Here is the call graph for this function:



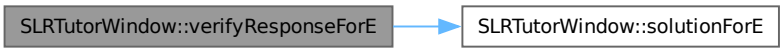
Here is the caller graph for this function:



5.14.3.76 verifyResponseForE()

bool SLRTutorWindow::verifyResponseForE (
const QString & userResponse)

Here is the call graph for this function:

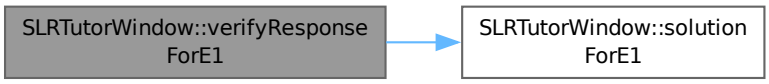


Here is the caller graph for this function:

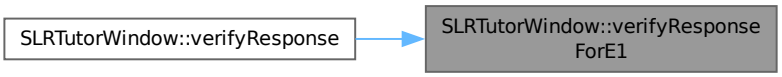
5.14.3.77 verifyResponseForE1()

bool SLRTutorWindow::verifyResponseForE1 (
const QString & userResponse)

Here is the call graph for this function:



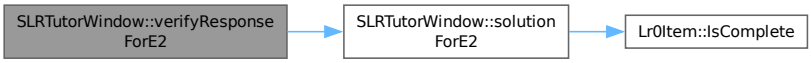
Here is the caller graph for this function:



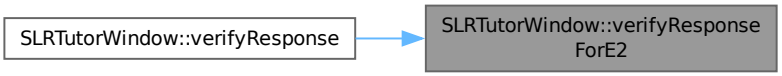
5.14.3.78 verifyResponseForE2()

bool SLRTutorWindow::verifyResponseForE2 (
const QString & userResponse)

Here is the call graph for this function:



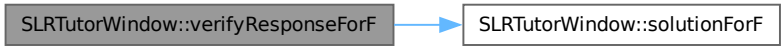
Here is the caller graph for this function:



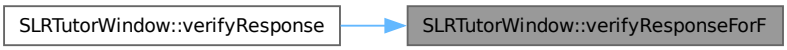
5.14.3.79 verifyResponseForF()

bool SLRTutorWindow::verifyResponseForF (
const QString & userResponse)

Here is the call graph for this function:



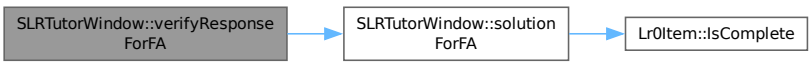
Here is the caller graph for this function:



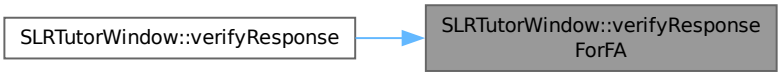
5.14.3.80 verifyResponseForFA()

bool SLRTutorWindow::verifyResponseForFA (
const QString & userResponse)

Here is the call graph for this function:



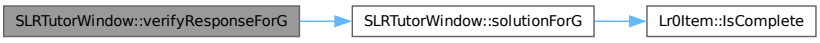
Here is the caller graph for this function:



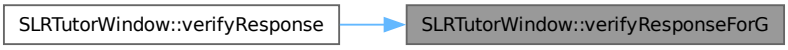
5.14.3.81 verifyResponseForG()

bool SLRTutorWindow::verifyResponseForG (
const QString & userResponse)

Here is the call graph for this function:



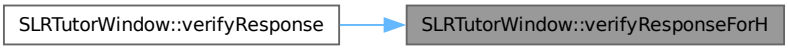
Here is the caller graph for this function:



5.14.3.82 verifyResponseForH()

bool SLRTutorWindow::verifyResponseForH ()

Here is the caller graph for this function:



5.14.3.83 wrongAnimation()

void SLRTutorWindow::wrongAnimation ()

5.14.3.84 wrongUserResponseAnimation()

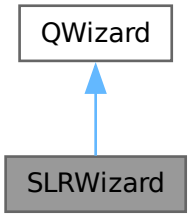
void SLRTutorWindow::wrongUserResponseAnimation ()

The documentation for this class was generated from the following files:

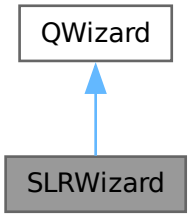
- slrtutorwindow.h
- slrtutorwindow.cpp

5.15 SLRWizard Class Reference

Interactive assistant that guides the student step-by-step through the SLR(1) parsing table.
#include <slrwizard.h>
Inheritance diagram for SLRWizard:



Collaboration diagram for SLRWizard:



Public Member Functions

- SLRWizard (SLR1Parser &parser, const QVector< QVector< QString > > &rawTable, const QStringList &colHeaders, const QVector< QPair< QString, QVector< QString > > > &sorted↩ Grammar, QWidget *parent=nullptr)

  Constructs the SLR(1) wizard with all necessary parsing context.
- QVector< QString > stdVectorToQVector (const std::vector< std::string > &vec)

  Converts a std::vector<std::string> to QVector<QString> for UI compatibility.

5.15.1 Detailed Description

Interactive assistant that guides the student step-by-step through the SLR(1) parsing table.
This wizard-based dialog presents the user with one cell of the SLR(1) parsing table at a time, asking them to deduce the correct ACTION or GOTO entry based on the LR(0) automaton and FOLLOW sets. It is designed as an educational aid to explain the reasoning behind each parsing decision.
Each page includes:

- The current state and symbol (terminal or non-terminal).
- A guided explanation based on the grammar and LR(0) state.
- The expected entry (e.g., s3, r1, acc, or a state number).

5.15.2 Constructor & Destructor Documentation

5.15.2.1 SLRWizard()

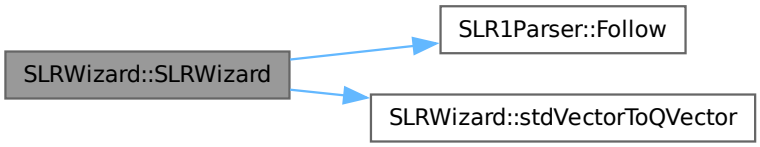
SLRWizard::SLRWizard (
SLR1Parser & parser,
const QVector< QVector< QString > > & rawTable,
const QStringList & colHeaders,
const QVector< QPair< QString, QVector< QString > > > & sortedGrammar,
QWidget * parent = nullptr) [inline]

Constructs the SLR(1) wizard with all necessary parsing context.

Parameters

parser	The SLR(1) parser instance containing the LR(0) states and transitions.
rawTable	The target parsing table (student version or reference).
colHeaders	Header symbols (terminals and non-terminals).
sortedGrammar	Ordered list of grammar rules for reduce explanations.
parent	Parent widget.

Here is the call graph for this function:



5.15.3 Member Function Documentation

5.15.3.1 stdVectorToQVector()

QVector< QString > SLRWizard::stdVectorToQVector (
const std::vector< std::string > & vec) [inline]

Converts a std::vector<std::string> to QVector<QString> for UI compatibility.

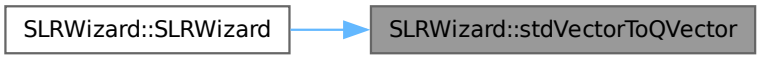
Parameters

vec	The input vector of strings.

Returns

A QVector of QStrings.

Here is the caller graph for this function:

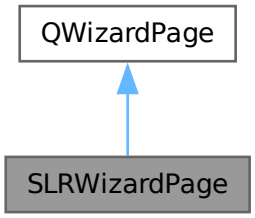


The documentation for this class was generated from the following file:

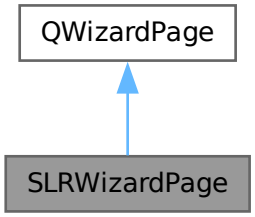
- slrwizard.h

5.16 SLRWizardPage Class Reference

A single step in the SLR(1) guided assistant for table construction.
#include <slrwizardpage.h>
Inheritance diagram for SLRWizardPage:



Collaboration diagram for SLRWizardPage:



Public Member Functions

- SLRWizardPage (int state, const QString &symbol, const QString &explanation, const QString &expected, QWidget *parent=nullptr)

  Constructs a page for a specific cell in the SLR(1) table.

5.16.1 Detailed Description

A single step in the SLR(1) guided assistant for table construction.
This wizard page presents a specific (state, symbol) cell in the SLR(1) parsing table, and prompts the student to enter the correct ACTION or GOTO value.
The page checks the user's input against the expected answer and provides immediate feedback, disabling the ”Next” button until the correct response is entered.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 SLRWizardPage()

SLRWizardPage::SLRWizardPage (
int state,
const QString & symbol,
const QString & explanation,
const QString & expected,
QWidget * parent = nullptr) [inline]

Constructs a page for a specific cell in the SLR(1) table.

Parameters

state	The state ID (row index in the table).
symbol	The grammar symbol (column header).
explanation	A pedagogical explanation shown to the user.
expected	The expected answer (e.g., "s2", "r1", "acc", or a state number).
parent	The parent widget.

The documentation for this class was generated from the following file:

- slrwizardpage.h

5.17 state Struct Reference

Represents a state in the LR(0) automaton.
#include <state.hpp>

Public Member Functions

- bool operator== (const state &other) const
  Equality operator for comparing states based on their items.

Public Attributes

- std::unordered_set< Lr0Item > items_
  The set of LR(0) items that make up this state.
- unsigned int id_
  Unique identifier of the state.

5.17.1 Detailed Description

Represents a state in the LR(0) automaton.
Each state consists of a unique identifier and a set of LR(0) items that define its core. States are used to build the SLR(1) parsing table.

5.17.2 Member Function Documentation

5.17.2.1 operator==()

bool state::operator== (
const state & other) const [inline]

Equality operator for comparing states based on their items.

Parameters

other	The state to compare with.

Returns

true if both states have the same item set; false otherwise.

5.17.3 Member Data Documentation

5.17.3.1 id_

unsigned int state::id_
Unique identifier of the state.

5.17.3.2 items_

std::unordered_set<Lr0Item> state::items_
The set of LR(0) items that make up this state.
The documentation for this struct was generated from the following file:

- backend/state.hpp

5.18 SymbolTable Struct Reference

Stores and manages grammar symbols, including their classification and special markers.
#include <symbol_table.hpp>

Public Member Functions

- void PutSymbol (const std::string &identifier, bool isTerminal)

  Adds a non-terminal symbol to the symbol table.
- bool In (const std::string &s) const

  Checks if a symbol exists in the symbol table.
- bool IsTerminal (const std::string &s) const

  Checks if a symbol is a terminal.
- bool IsTerminalWthoEol (const std::string &s) const

  Checks if a symbol is a terminal excluding EOL.

Public Attributes

- std::string EOL_ {"$"}

  End-of-line symbol used in parsing, initialized as "$".
- std::string EPSILON_ {"EPSILON"}

  Epsilon symbol, representing empty transitions, initialized as "EPSILON".
- std::unordered_map< std::string, symbol_type > st_

  Main symbol table, mapping identifiers to a pair of symbol type and its regex.
- std::unordered_set< std::string > terminals_ {EOL_}

  Set of all terminal symbols (including EOL).
- std::unordered_set< std::string > terminals_wtho_eol_ {}

  Set of terminal symbols excluding the EOL symbol ($).
- std::unordered_set< std::string > non_terminals_

  Set of all non-terminal symbols.

5.18.1 Detailed Description

Stores and manages grammar symbols, including their classification and special markers.
This structure holds information about all terminals and non-terminals used in a grammar, as well as special symbols such as EPSILON and the end-of-line marker ($). It supports symbol classification, membership checks, and filtered views such as terminals excluding $.

5.18.2 Member Function Documentation

5.18.2.1 In()

bool SymbolTable::In (
const std::string & s) const

Checks if a symbol exists in the symbol table.

Parameters

s	Symbol identifier to search.

Returns

true if the symbol is present, otherwise false.

5.18.2.2 IsTerminal()

bool SymbolTable::IsTerminal (
const std::string & s) const

Checks if a symbol is a terminal.

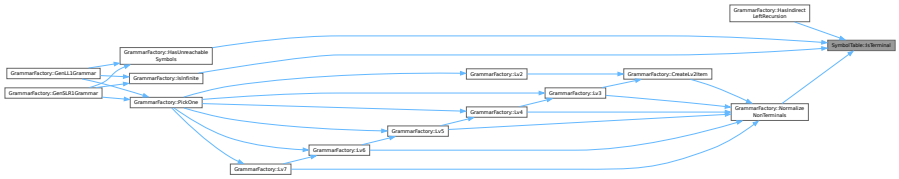
Parameters

s	Symbol identifier to check.

Returns

true if the symbol is terminal, otherwise false.

Here is the caller graph for this function:



5.18.2.3 IsTerminalWthoEol()

bool SymbolTable::IsTerminalWthoEol (
const std::string & s) const

Checks if a symbol is a terminal excluding EOL.

Parameters

s	Symbol identifier to check.

Returns

true if the symbol is terminal, otherwise false.

5.18.2.4 PutSymbol()

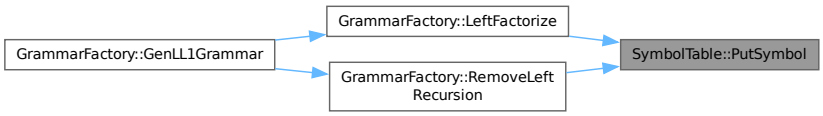
void SymbolTable::PutSymbol (
const std::string & identifier,
bool isTerminal)

Adds a non-terminal symbol to the symbol table.

Parameters

identifier	Name of the symbol.
isTerminal	True if the identifier is a terminal symbol

Here is the caller graph for this function:



5.18.3 Member Data Documentation

5.18.3.1 EOL_

std::string SymbolTable::EOL_ {"$"}
End-of-line symbol used in parsing, initialized as "$".

5.18.3.2 EPSILON_

std::string SymbolTable::EPSILON_ {"EPSILON"}
Epsilon symbol, representing empty transitions, initialized as "EPSILON".

5.18.3.3 non_terminals_

std::unordered_set<std::string> SymbolTable::non_terminals_
Set of all non-terminal symbols.

5.18.3.4 st_

std::unordered_map<std::string, symbol_type> SymbolTable::st_
Initial value:
{{EOL_, symbol_type::TERMINAL},
{EPSILON_, symbol_type::TERMINAL}}
Main symbol table, mapping identifiers to a pair of symbol type and its regex.

5.18.3.5 terminals_

std::unordered_set<std::string> SymbolTable::terminals_ {EOL_}
Set of all terminal symbols (including EOL).

5.18.3.6 terminals_wtho_eol_

std::unordered_set<std::string> SymbolTable::terminals_wtho_eol_ {}
Set of terminal symbols excluding the EOL symbol ($).
The documentation for this struct was generated from the following files:

- backend/symbol_table.hpp
- backend/symbol_table.cpp

5.19 LLTutorWindow::TreeNode Struct Reference

TreeNode structure used to build derivation trees.
#include <lltutorwindow.h>

Public Attributes

- QString label
- std::vector< std::unique_ptr< TreeNode > > children

5.19.1 Detailed Description

TreeNode structure used to build derivation trees.

5.19.2 Member Data Documentation

5.19.2.1 children

std::vector<std::unique_ptr<TreeNode> > LLTutorWindow::TreeNode::children

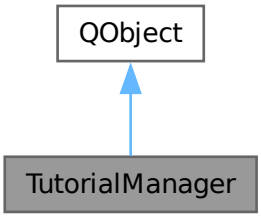
5.19.2.2 label

QString LLTutorWindow::TreeNode::label
The documentation for this struct was generated from the following file:

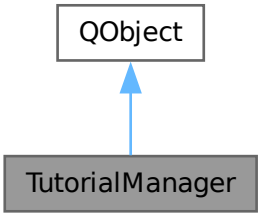
- lltutorwindow.h

5.20 TutorialManager Class Reference

Manages interactive tutorials by highlighting UI elements and guiding the user.
#include <tutorialmanager.h>
Inheritance diagram for TutorialManager:



Collaboration diagram for TutorialManager:



Public Slots

- void nextStep ()

  Advances to the next tutorial step.

Signals

- void stepStarted (int index)

  Emitted when a new tutorial step starts.
- void tutorialFinished ()

  Emitted when the full tutorial is finished.
- void ll1Finished ()

  Emitted when the LL(1) tutorial ends.
- void slr1Finished ()

  Emitted when the SLR(1) tutorial ends.

Public Member Functions

- TutorialManager (QWidget *rootWindow)

  Constructs a TutorialManager for a given window.
- void addStep (QWidget *target, const QString &htmlText)

  Adds a new step to the tutorial sequence.
- void start ()

  Starts the tutorial from the beginning.
- void setRootWindow (QWidget *newRoot)

  Sets the root window (used for repositioning the overlay).
- void clearSteps ()

  Clears all steps in the tutorial.
- void hideOverlay ()

  Hides the tutorial overlay immediately.
- void finishLL1 ()

  Ends the LL(1) tutorial sequence and emits its corresponding signal.
- void finishSLR1 ()

  Ends the SLR(1) tutorial sequence and emits its corresponding signal.

Protected Member Functions

- bool eventFilter (QObject *obj, QEvent *ev) override

  Intercepts UI events to handle overlay behavior.

5.20.1 Detailed Description

Manages interactive tutorials by highlighting UI elements and guiding the user.
This class implements a step-by-step overlay system that visually highlights widgets and shows textual instructions to guide the user through the interface. It supports multiple tutorials (e.g., for LL(1) and SLR(1) modes), with custom steps and signals for tutorial completion.

5.20.2 Constructor & Destructor Documentation

5.20.2.1 TutorialManager()

TutorialManager::TutorialManager (
QWidget * rootWindow)

Constructs a TutorialManager for a given window.

Parameters

rootWindow	The main application window used for relative positioning.

5.20.3 Member Function Documentation

5.20.3.1 addStep()

void TutorialManager::addStep (
QWidget * target,
const QString & htmlText)

Adds a new step to the tutorial sequence.

Parameters

target	The widget to highlight during the step.
htmlText	The instructional HTML message for the step.

5.20.3.2 clearSteps()

void TutorialManager::clearSteps ()

Clears all steps in the tutorial.

Here is the call graph for this function:

5.20.3.3 eventFilter()

bool TutorialManager::eventFilter (
QObject * obj,
QEvent * ev) [override], [protected]

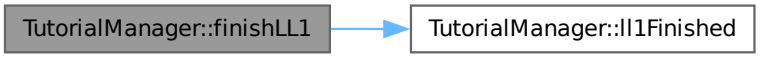
Intercepts UI events to handle overlay behavior.

5.20.3.4 finishLL1()

void TutorialManager::finishLL1 ()

Ends the LL(1) tutorial sequence and emits its corresponding signal.

Here is the call graph for this function:

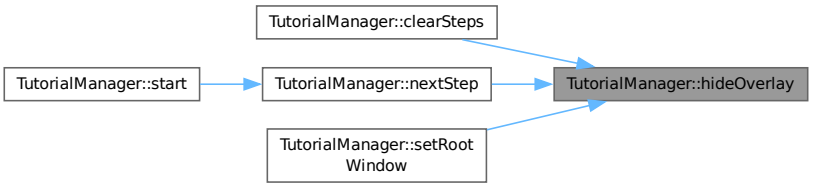


5.20.3.5 finishSLR1()

void TutorialManager::finishSLR1 ()
Ends the SLR(1) tutorial sequence and emits its corresponding signal.
Here is the call graph for this function:

5.20.3.6 hideOverlay()

void TutorialManager::hideOverlay ()
Hides the tutorial overlay immediately.
Here is the caller graph for this function:



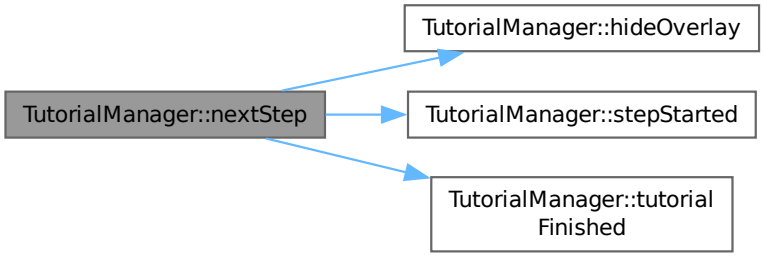
5.20.3.7 ll1Finished

void TutorialManager::ll1Finished () [signal]
Emitted when the LL(1) tutorial ends.
Here is the caller graph for this function:

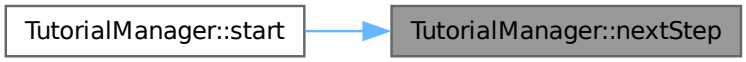
5.20.3.8 nextStep

void TutorialManager::nextStep () [slot]
Advances to the next tutorial step.

Here is the call graph for this function:



Here is the caller graph for this function:



5.20.3.9 setRootWindow()

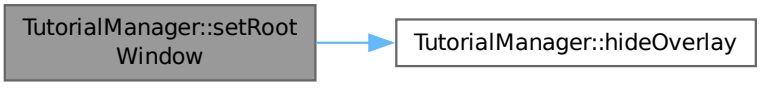
void TutorialManager::setRootWindow (
QWidget * newRoot)

Sets the root window (used for repositioning the overlay).

Parameters

newRoot	The new main window to reference.

Here is the call graph for this function:



5.20.3.10 slr1Finished

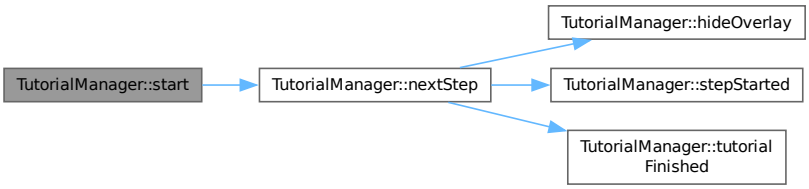
void TutorialManager::slr1Finished () [signal]

Emitted when the SLR(1) tutorial ends.

Here is the caller graph for this function:

5.20.3.11 start()

void TutorialManager::start ()
Starts the tutorial from the beginning.
Here is the call graph for this function:



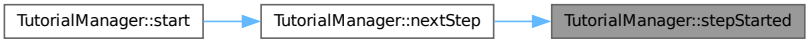
5.20.3.12 stepStarted

void TutorialManager::stepStarted (
int index) [signal]
Emitted when a new tutorial step starts.

Parameters

index	Index of the current step.

Here is the caller graph for this function:



5.20.3.13 tutorialFinished

void TutorialManager::tutorialFinished () [signal]
Emitted when the full tutorial is finished.

Here is the caller graph for this function:

The documentation for this class was generated from the following files:

- tutorialmanager.h
- tutorialmanager.cpp

5.21 TutorialStep Struct Reference

Represents a single step in the tutorial sequence.
#include <tutorialmanager.h>

Public Attributes

- QWidget * target

  Widget to highlight during the tutorial step.
- QString htmlText

  HTML text to show as instruction or explanation.

5.21.1 Detailed Description

Represents a single step in the tutorial sequence.
Each step highlights a target widget and displays an associated HTML-formatted message.

5.21.2 Member Data Documentation

5.21.2.1 htmlText

QString TutorialStep::htmlText
HTML text to show as instruction or explanation.

5.21.2.2 target

QWidget* TutorialStep::target
Widget to highlight during the tutorial step.
The documentation for this struct was generated from the following file:

- tutorialmanager.h

5.22 UniqueQueue< T > Class Template Reference

A queue that ensures each element is inserted only once.
#include <UniqueQueue.h>

Public Member Functions

- void push (const T &value)

  Pushes an element to the queue if it hasn't been inserted before.
- void pop ()

  Removes the front element from the queue.

- const T & front () const

  Accesses the front element of the queue.
- bool empty () const

  Checks whether the queue is empty.
- void clear ()

  Clears the queue and the set of seen elements.

5.22.1 Detailed Description

template<typename T>
class UniqueQueue< T >

A queue that ensures each element is inserted only once.
This data structure behaves like a standard FIFO queue but prevents duplicate insertions.
Internally, it uses a std::queue for ordering and a std::unordered_set to track seen elements.

Template Parameters

T	The type of elements stored in the queue. Must be hashable and comparable.

5.22.2 Member Function Documentation

5.22.2.1 clear()

template<typename T>
void UniqueQueue< T >::clear () [inline]
Clears the queue and the set of seen elements.

5.22.2.2 empty()

template<typename T>
bool UniqueQueue< T >::empty () const [inline]
Checks whether the queue is empty.

Returns

true if the queue is empty; false otherwise.

5.22.2.3 front()

template<typename T>
const T & UniqueQueue< T >::front () const [inline]
Accesses the front element of the queue.

Returns

A reference to the front element.

5.22.2.4 pop()

template<typename T>
void UniqueQueue< T >::pop () [inline]
Removes the front element from the queue.

5.22.2.5 push()

template<typename T>
void UniqueQueue< T >::push (
const T & value) [inline]
Pushes an element to the queue if it hasn't been inserted before.

Parameters

value	The element to insert.

The documentation for this class was generated from the following file:

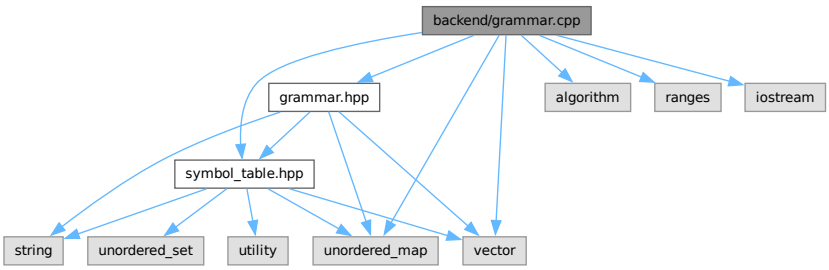
- UniqueQueue.h

Chapter 6

File Documentation

6.1 backend/grammar.cpp File Reference

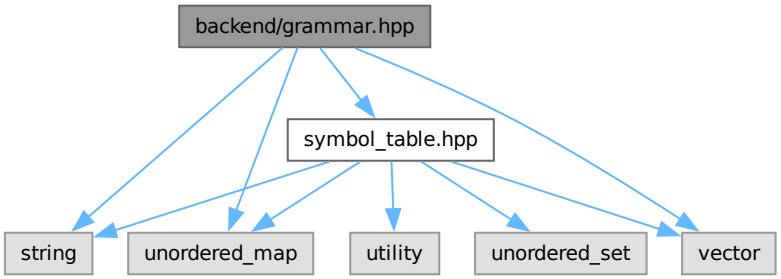
#include ”grammar.hpp”
#include ”symbol_table.hpp”
#include <algorithm>
#include <ranges>
#include <iostream>
#include <unordered_map>
#include <vector>
Include dependency graph for grammar.cpp:



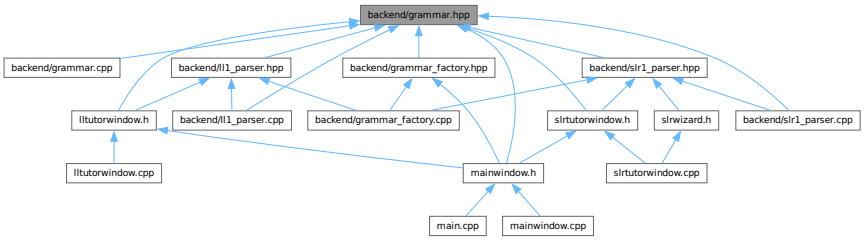
6.2 backend/grammar.hpp File Reference

#include ”symbol_table.hpp”
#include <string>
#include <unordered_map>
#include <vector>

Include dependency graph for grammar.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct Grammar

  Represents a context-free grammar, including its rules, symbol table, and starting symbol.

Typedefs

- using production = std::vector<std::string>

  Represents the right-hand side of a grammar rule.

6.2.1 Typedef Documentation

6.2.1.1 production

using production = std::vector<std::string>
Represents the right-hand side of a grammar rule.
A production is a sequence of grammar symbols (terminals or non-terminals) that can be derived from a non-terminal symbol in the grammar.
For example, in the rule A → a B c, the production would be: {"a", "B", "c"}

6.3 grammar.hpp

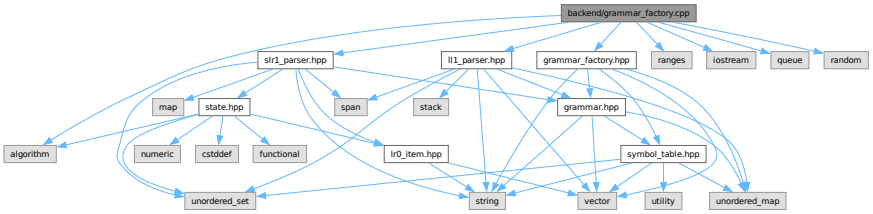
Go to the documentation of this file.

```
00001 #pragma once
00002 #include "symbol_table.hpp"
00003 #include <string>
00004 #include <unordered_map>
00005 #include <vector>
00006
```

```
00017 using production = std::vector<std::string>;
00018
00027 struct Grammar {
00028
00029     Grammar();
00030     explicit Grammar(
00031         const std::unordered_map<std::string, std::vector<production»&
00032             grammar);
00033
00034
00043     void SetAxiom(const std::string& axiom);
00044
00055     bool HasEmptyProduction(const std::string& antecedent) const;
00056
00068     std::vector<std::pair<const std::string, production» FilterRulesByConsequent(
00069         const std::string& arg) const;
00070
00077     void Debug() const; //NOSONAR
00078
00089     void AddProduction(const std::string&              antecedent,
00090                        const std::vector<std::string>& consequent);
00091
00102     std::vector<std::string> Split(const std::string& s);
00103
00108     std::unordered_map<std::string, std::vector<production» g_;
00109
00113     std::string axiom_;
00114
00118     SymbolTable st_;
00119 };
```

6.4 backend/grammar_factory.cpp File Reference

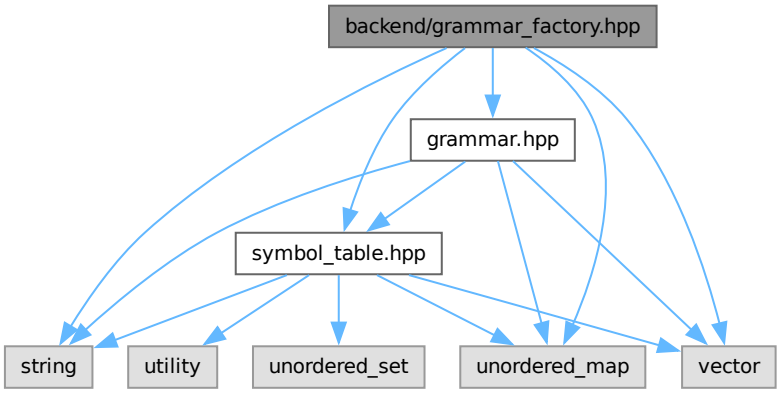
#include ”grammar_factory.hpp”
#include ”ll1_parser.hpp”
#include ”slr1_parser.hpp”
#include <algorithm>
#include <ranges>
#include <iostream>
#include <queue>
#include <random>
Include dependency graph for grammar_factory.cpp:



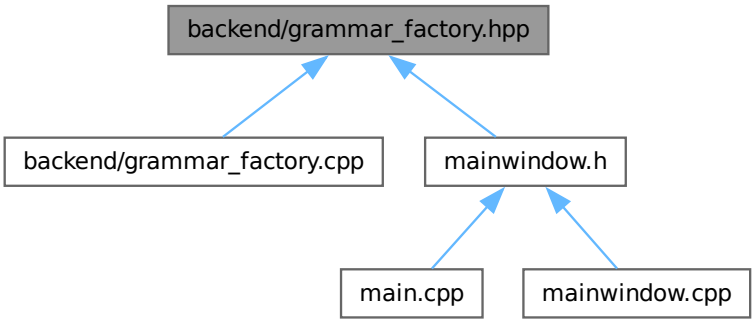
6.5 backend/grammar_factory.hpp File Reference

#include ”grammar.hpp”
#include ”symbol_table.hpp”
#include <string>
#include <unordered_map>
#include <vector>

Include dependency graph for grammar_factory.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct GrammarFactory

  Responsible for creating and managing grammar items and performing checks on grammars.
- struct GrammarFactory::FactoryItem

  Represents an individual grammar item with its associated symbol table.

6.6 grammar_factory.hpp

Go to the documentation of this file.

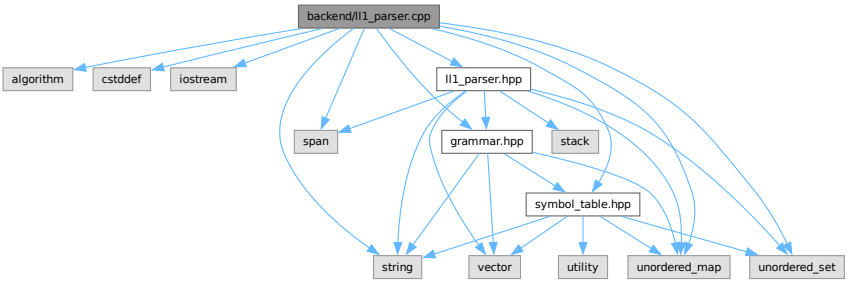
```
00001 #pragma once
00002
00003 #include "grammar.hpp"
00004 #include "symbol_table.hpp"
00005 #include <string>
00006 #include <unordered_map>
00007 #include <vector>
00008
00014 struct GrammarFactory {
00015
```

```
00021     struct FactoryItem {
00026         std::unordered_map<std::string, std::vector<production» g_;
00027
00031         SymbolTable st_;
00032
00038         explicit FactoryItem(const std::unordered_map<std::string, std::vector<production» &grammar);
00039     };
00040
00045     void Init();
00046
00053     Grammar PickOne(int level);
00054
00061     Grammar GenLL1Grammar(int level);
00068     Grammar GenSLR1Grammar(int level);
00069
00074     Grammar Lv1();
00075
00080     Grammar Lv2();
00081
00087     Grammar Lv3();
00088
00099     Grammar Lv4();
00100
00111     Grammar Lv5();
00112
00123     Grammar Lv6();
00124
00135     Grammar Lv7();
00136
00145     FactoryItem CreateLv2Item();
00146
00147     // -------- SANITY CHECKS --------
00148
00155     bool HasUnreachableSymbols(Grammar& grammar) const;
00156
00168     bool IsInfinite(Grammar& grammar) const;
00169
00176     bool HasDirectLeftRecursion(const Grammar& grammar) const;
00177
00183     bool HasIndirectLeftRecursion(const Grammar& grammar) const;
00184
00190     bool HasCycle(const std::unordered_map<std::string, std::unordered_set<std::string»& graph) const;
00191
00197     std::unordered_set<std::string> NullableSymbols(const Grammar& grammar) const;
00198
00199     // -------- TRANSFORMATIONS --------
00209     void RemoveLeftRecursion(Grammar& grammar);
00210
00221     void LeftFactorize(Grammar& grammar);
00222
00234     std::vector<std::string>
00235     LongestCommonPrefix(const std::vector<production>& productions);
00236
00250     bool StartsWith(const production&               prod,
00251                     const std::vector<std::string>& prefix);
00252
00265     std::string GenerateNewNonTerminal(Grammar&           grammar,
00266                                        const std::string& base);
00267
00278     void NormalizeNonTerminals(FactoryItem& item, const std::string& nt) const;
00279
00291     void AdjustTerminals(FactoryItem& base, const FactoryItem& cmb,
00292                          const std::string& target_nt) const;
00293
00304     std::unordered_map<std::string, std::vector<production»
00305     Merge(const FactoryItem& base, const FactoryItem& cmb) const;
00306
00311     std::vector<FactoryItem> items;
00312
00316     std::vector<std::string> terminal_alphabet_{"a", "b", "c", "d", "e", "f",
00317                                                 "g", "h", "i", "j", "k", "l"};
00318
00322     std::vector<std::string> non_terminal_alphabet_{"A", "B", "C", "D",
00323                                                     "E", "F", "G"};
00324 };
```

6.7 backend/ll1_parser.cpp File Reference

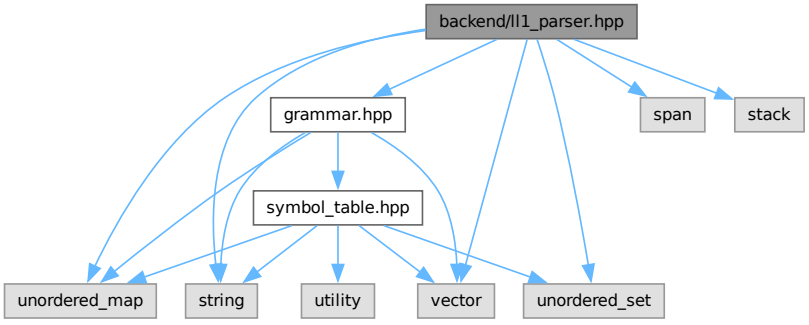
```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <span>
```

#include <string>
#include <unordered_map>
#include <unordered_set>
#include "grammar.hpp"
#include "ll1_parser.hpp"
#include "symbol_table.hpp"
Include dependency graph for ll1_parser.cpp:

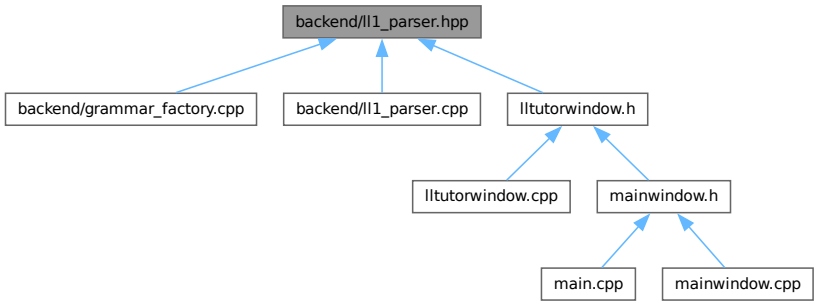


6.8 backend/ll1_parser.hpp File Reference

#include "grammar.hpp"
#include
#include <stack>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>
Include dependency graph for ll1_parser.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class LL1Parser

6.9 ll1_parser.hpp

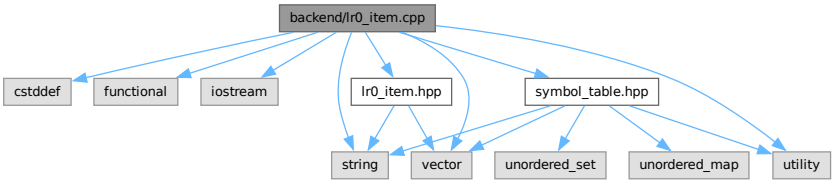
Go to the documentation of this file.

```
00001 #pragma once
00002 #include "grammar.hpp"
00003 #include <span>
00004 #include <stack>
00005 #include <string>
00006 #include <unordered_map>
00007 #include <unordered_set>
00008 #include <vector>
00009
00010 class LL1Parser {
00011
00027     using ll1_table = std::unordered_map<
00028         std::string, std::unordered_map<std::string, std::vector<production»>>;
00029
00030   public:
00031     LL1Parser() = default;
00037     explicit LL1Parser(Grammar gr);
00038
00060     bool CreateLL1Table();
00061
00088     void First(std::span<const std::string>      rule,
00089                std::unordered_set<std::string>& result);
00090
00101     void ComputeFirstSets();
00102
00130     void ComputeFollowSets();
00131
00146     std::unordered_set<std::string> Follow(const std::string& arg);
00147
00170     std::unordered_set<std::string>
00171     PredictionSymbols(const std::string&              antecedent,
00172                       const std::vector<std::string>& consequent);
00173
00176     ll1_table ll1_t_;
00177
00179     Grammar gr_;
00180
00182     std::unordered_map<std::string, std::unordered_set<std::string»
00183         first_sets_;
00184
00186     std::unordered_map<std::string, std::unordered_set<std::string»
00187         follow_sets_;
00188 };
```

6.10 backend/lr0_item.cpp File Reference

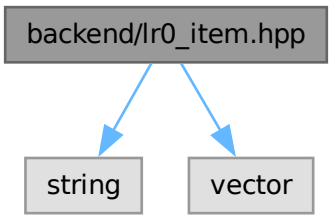
#include <cstddef>
#include <functional>

#include <iostream>
#include <string>
#include <utility>
#include <vector>
#include "lr0_item.hpp"
#include "symbol_table.hpp"
Include dependency graph for lr0_item.cpp:

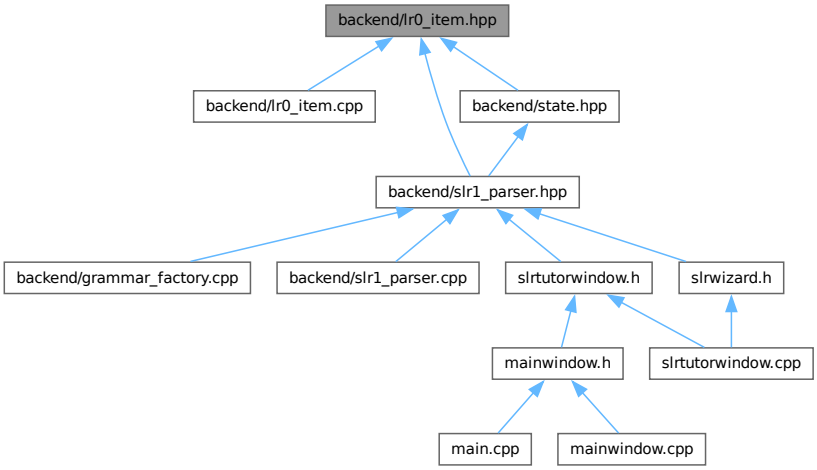


6.11 backend/lr0_item.hpp File Reference

#include <string>
#include <vector>
Include dependency graph for lr0_item.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct Lr0Item

  Represents an LR(0) item used in LR automata construction.

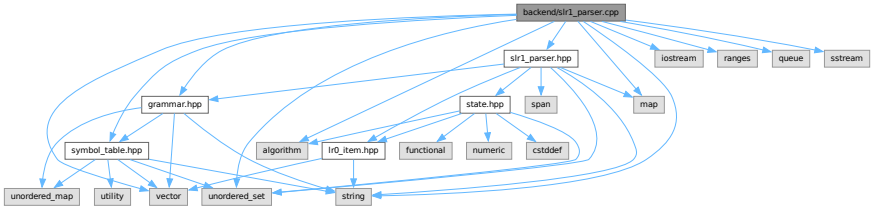
6.12 lr0_item.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <vector>
00005
00014 struct Lr0Item {
00018     std::string antecedent_;
00019
00023     std::vector<std::string> consequent_;
00024
00028     std::string epsilon_;
00029
00033     std::string eol_;
00034
00038     unsigned int dot_ = 0;
00039
00047     Lr0Item(std::string antecedent, std::vector<std::string> consequent,
00048             std::string epsilon, std::string eol);
00049
00058     Lr0Item(std::string antecedent, std::vector<std::string> consequent,
00059             unsigned int dot, std::string epsilon, std::string eol);
00060
00065     std::string NextToDot() const;
00066
00070     void PrintItem() const;
00071
00076     std::string ToString() const;
00077
00081     void AdvanceDot();
00082
00087     bool IsComplete() const;
00088
00094     bool        operator==(const Lr0Item& other) const;
00095 };
00096
00097 namespace std {
00098 template <> struct hash<Lr0Item> {
00099     size_t operator()(const Lr0Item& item) const;
00100 };
00101 } // namespace std
```

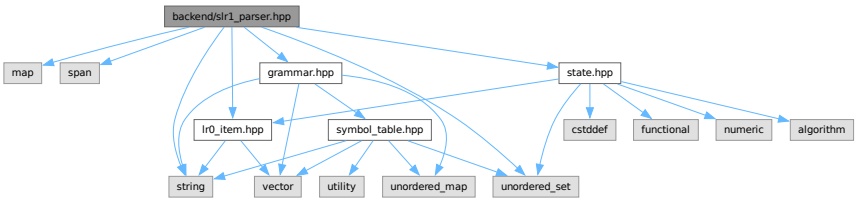
6.13 backend/slr1_parser.cpp File Reference

#include <algorithm>
#include <iostream>
#include <ranges>
#include <map>
#include <queue>
#include <string>
#include <sstream>
#include <unordered_set>
#include <vector>
#include "grammar.hpp"
#include "slr1_parser.hpp"
#include "symbol_table.hpp"
Include dependency graph for slr1_parser.cpp:

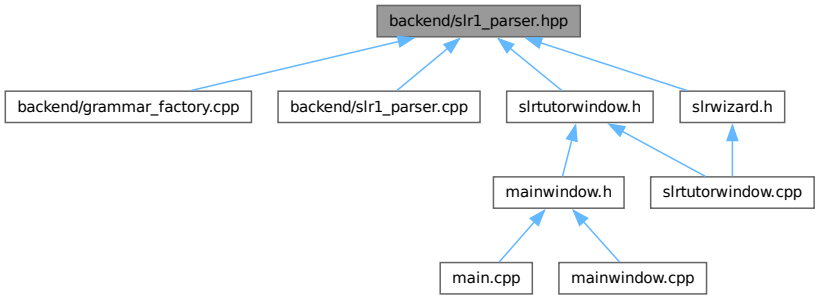


6.14 backend/slr1_parser.hpp File Reference

#include <map>
#include
#include <string>
#include <unordered_set>
#include "grammar.hpp"
#include "lr0_item.hpp"
#include "state.hpp"
Include dependency graph for slr1_parser.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class SLR1Parser

  Implements an SLR(1) parser for context-free grammars.
- struct SLR1Parser::s_action

6.15 slr1_parser.hpp

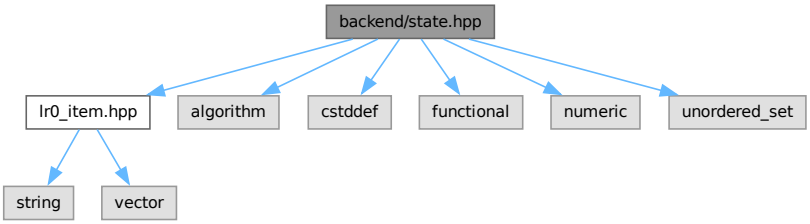
Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <map>
00004 #include <span>
00005 #include <string>
00006 #include <unordered_set>
00007
00008 #include "grammar.hpp"
00009 #include "lr0_item.hpp"
00010 #include "state.hpp"
00011
00020 class SLR1Parser {
00021   public:
00033     enum class Action { Shift, Reduce, Accept, Empty };
00034
00046     struct s_action {
00047         const Lr0Item* item;
00048         Action         action;
00049     };
00050
00063     using action_table =
00064         std::map<unsigned int, std::map<std::string, SLR1Parser::s_action»;
00065
00078     using transition_table =
00079         std::map<unsigned int, std::map<std::string, unsigned int»;
00080
00081     SLR1Parser() = default;
00082     explicit SLR1Parser(Grammar gr);
00083
00093     std::unordered_set<Lr0Item> AllItems() const;
00094
00105     void Closure(std::unordered_set<Lr0Item>& items);
00106
00120     void ClosureUtil(std::unordered_set<Lr0Item>& items, unsigned int size,
00121                      std::unordered_set<std::string>& visited);
00122
00133     std::unordered_set<Lr0Item> Delta(const std::unordered_set<Lr0Item>& items,
00134                                       const std::string&                 str);
00135
00148     bool SolveLRConflicts(const state& st);
00149
00176     void First(std::span<const std::string>      rule,
00177                std::unordered_set<std::string>& result);
00188     void ComputeFirstSets();
00189
00217     void ComputeFollowSets();
00218
00233     std::unordered_set<std::string> Follow(const std::string& arg);
```

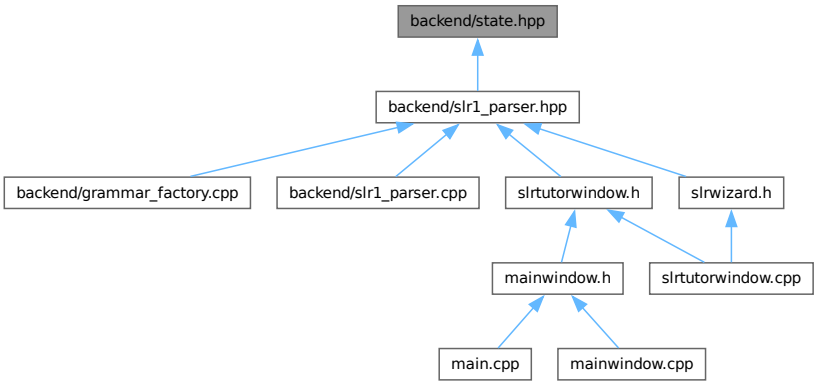
```
00234
00247     void MakeInitialState();
00248
00266     bool MakeParser();
00267
00277     std::string PrintItems(const std::unordered_set<Lr0Item>& items) const;
00278
00280     Grammar gr_;
00281
00283     std::unordered_map<std::string, std::unordered_set<std::string»
00284         first_sets_;
00285
00287     std::unordered_map<std::string, std::unordered_set<std::string»
00288         follow_sets_;
00289
00292     action_table actions_;
00293
00296     transition_table transitions_;
00297
00299     std::unordered_set<state> states_;
00300 };
```

6.16 backend/state.hpp File Reference

#include ”lr0_item.hpp”
#include <algorithm>
#include <cstddef>
#include <functional>
#include <numeric>
#include <unordered_set>
Include dependency graph for state.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct state

  Represents a state in the LR(0) automaton.

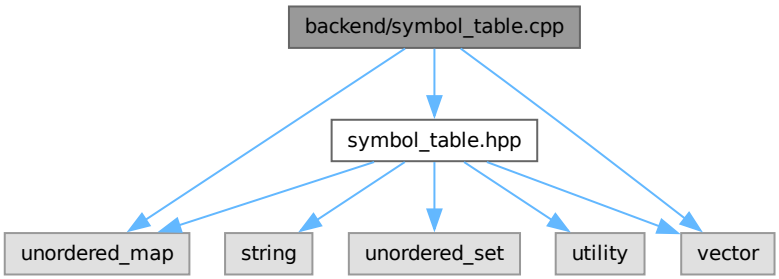
6.17 state.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002 #include "lr0_item.hpp"
00003 #include <algorithm>
00004 #include <cstddef>
00005 #include <functional>
00006 #include <numeric>
00007 #include <unordered_set>
00008
00016 struct state {
00020     std::unordered_set<Lr0Item> items_;
00021
00025     unsigned int id_;
00026
00032     bool operator==(const state& other) const { return other.items_ == items_; }
00033 };
00034
00035 namespace std {
00036 template <> struct hash<state> {
00037     size_t operator()(const state& st) const {
00038         size_t seed =
00039             std::accumulate(st.items_.begin(), st.items_.end(), 0,
00040                             [](size_t acc, const Lr0Item& item) {
00041                                 return acc ^ (std::hash<Lr0Item>()(item));
00042                             });
00043         return seed;
00044     }
00045 };
00046 } // namespace std
```

6.18 backend/symbol_table.cpp File Reference

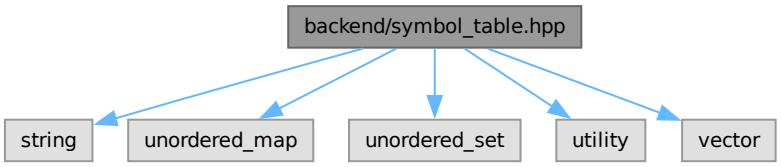
#include "symbol_table.hpp"
#include <unordered_map>
#include <vector>
Include dependency graph for symbol_table.cpp:



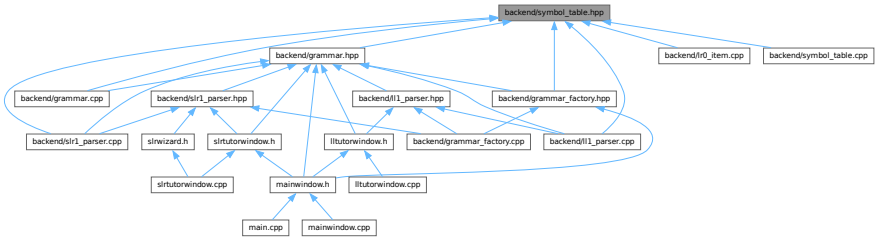
6.19 backend/symbol_table.hpp File Reference

#include <string>
#include <unordered_map>
#include <unordered_set>
#include <utility>

#include <vector>
Include dependency graph for symbol_table.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct SymbolTable

  Stores and manages grammar symbols, including their classification and special markers.

Enumerations

- enum class symbol_type { NO_TERMINAL , TERMINAL }

  Represents the type of a grammar symbol.

6.19.1 Enumeration Type Documentation

6.19.1.1 symbol_type

enum class symbol_type [strong]
Represents the type of a grammar symbol.
This enum distinguishes between terminal and non-terminal symbols within the grammar and the symbol table.

Enumerator

NO_TERMINAL	
TERMINAL	

6.20 symbol_table.hpp

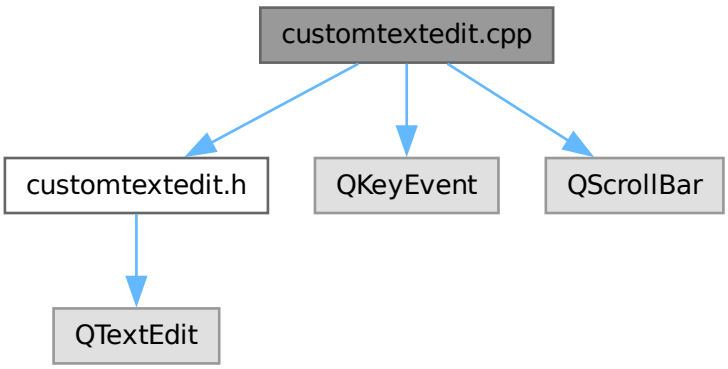
Go to the documentation of this file.

```
00001 #pragma once
00002 #include <string>
00003 #include <unordered_map>
```

```
00004 #include <unordered_set>
00005 #include <utility>
00006 #include <vector>
00007
00015 enum class symbol_type { NO_TERMINAL, TERMINAL };
00016
00026 struct SymbolTable {
00028     std::string EOL_{"$"};
00029
00032     std::string EPSILON_{"EPSILON"};
00033
00036     std::unordered_map<std::string, symbol_type> st_{{EOL_, symbol_type::TERMINAL},
00037                                                      {EPSILON_, symbol_type::TERMINAL}};
00038
00042     std::unordered_set<std::string> terminals_{EOL_};
00043
00047     std::unordered_set<std::string> terminals_wtho_eol_{};
00048
00052     std::unordered_set<std::string> non_terminals_;
00053
00060     void PutSymbol(const std::string& identifier, bool isTerminal);
00061
00068     bool In(const std::string& s) const;
00069
00076     bool IsTerminal(const std::string& s) const;
00077
00084     bool IsTerminalWthoEol(const std::string& s) const;
00085 };
```

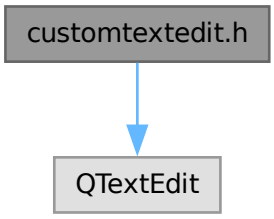
6.21 customtextedit.cpp File Reference

#include "customtextedit.h"
#include <QKeyEvent>
#include <QScrollBar>
Include dependency graph for customtextedit.cpp:

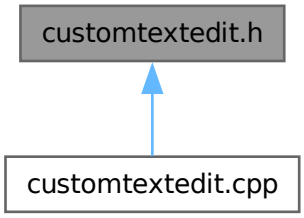


6.22 customtextedit.h File Reference

#include <QTextEdit>
Include dependency graph for customtextedit.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CustomTextEdit

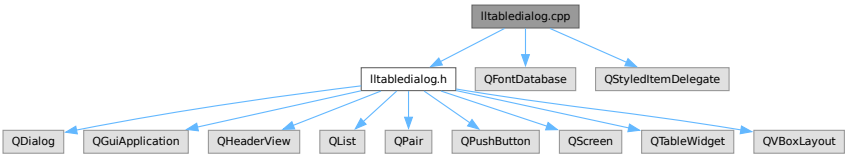
6.23 customtextedit.h

Go to the documentation of this file.

```
00001 #ifndef CUSTOMTEXTEDIT_H
00002 #define CUSTOMTEXTEDIT_H
00003
00004 #include <QTextEdit>
00005
00006 class CustomTextEdit : public QTextEdit
00007 {
00008     Q_OBJECT
00009 public:
00010     explicit CustomTextEdit(QWidget *parent = nullptr);
00011
00012 signals:
00013     void sendRequested();
00014
00015 protected:
00016     void keyPressEvent(QKeyEvent *event) override;
00017 };
00018
00019 #endif // CUSTOMTEXTEDIT_H
```

6.24 lltabledialog.cpp File Reference

#include ”lltabledialog.h”
#include <QFontDatabase>
#include <QStyledItemDelegate>
Include dependency graph for lltabledialog.cpp:

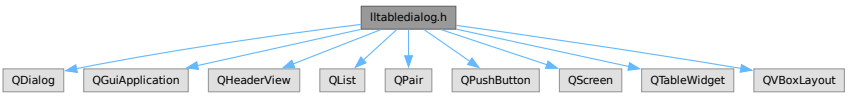


Classes

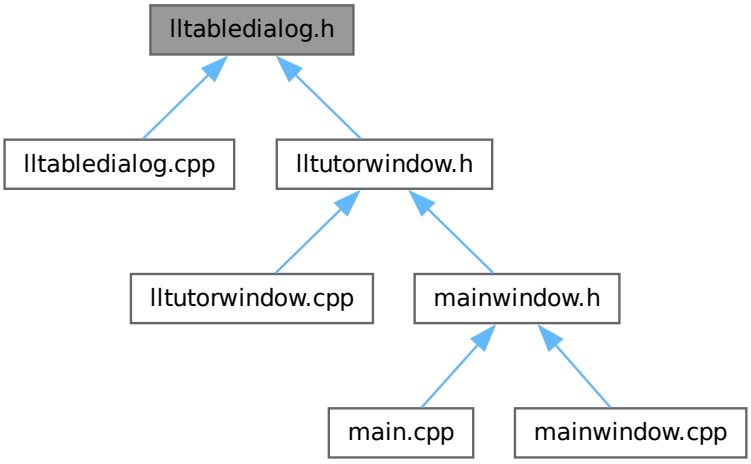
- class CenterAlignDelegate

6.25 lltabledialog.h File Reference

#include <QDialog>
#include <QGuiApplication>
#include <QHeaderView>
#include <QList>
#include <QPair>
#include <QPushButton>
#include <QScreen>
#include <QTableWidget>
#include <QVBoxLayout>
Include dependency graph for lltabledialog.h:



This graph shows which files directly or indirectly include this file:



Classes

- class LLTableDialog

  Dialog for filling and submitting an LL(1) parsing table.

6.26 lltabledialog.h

Go to the documentation of this file.

```
00001 #ifndef LLTABLEDIALOG_H
00002 #define LLTABLEDIALOG_H
00003
00004 #include <QDialog>
00005 #include <QGuiApplication>
00006 #include <QHeaderView>
00007 #include <QList>
00008 #include <QPair>
00009 #include <QPushButton>
00010 #include <QScreen>
00011 #include <QTableWidget>
00012 #include <QVBoxLayout>
00013
00022 class LLTableDialog : public QDialog
00023 {
00024     Q_OBJECT
00025 public:
00033     LLTableDialog(const QStringList &rowHeaders,
00034                   const QStringList &colHeaders,
00035                   QWidget *parent,
00036                   QVector<QVector<QString>> *initialData = nullptr);
00037
00042     QVector<QVector<QString>> getTableData() const;
00043
00052     void setInitialData(const QVector<QVector<QString>> &data);
00053
00058     void highlightIncorrectCells(const QList<QPair<int, int>> &coords);
00059
00060 signals:
00065     void submitted(const QVector<QVector<QString>> &data);
00066
00067 private:
00068     QTableWidget *table;
00069     QPushButton *submitButton;
00070 };
00071
00072 #endif // LLTABLEDIALOG_H
```

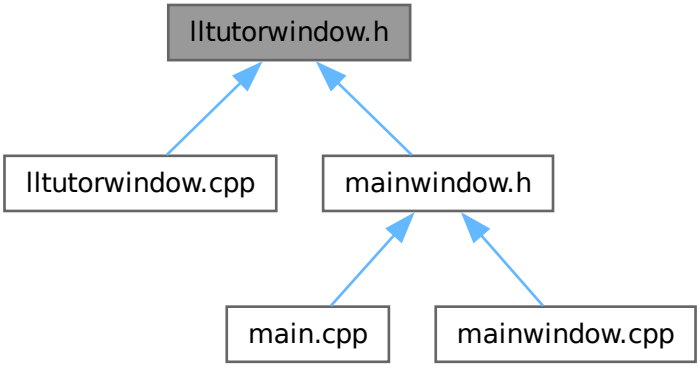
6.27 lltutorwindow.cpp File Reference

#include ”lltutorwindow.h”
#include ”tutorialmanager.h”
#include ”ui_lltutorwindow.h”
#include <QAbstractButton>
#include <QFontDatabase>
#include <QRandomGenerator>
#include <QRegularExpression>
#include <QWheelEvent>
Include dependency graph for lltutorwindow.cpp:

6.28 lltutorwindow.h File Reference

#include <QAbstractItemView>
#include <QDialog>
#include <QFileDialog>
#include <QGraphicsColorizeEffect>
#include <QGraphicsScene>
#include <QGraphicsTextItem>
#include <QGraphicsView>
#include <QListWidgetItem>
#include <QMainWindow>
#include <QMessageBox>
#include <QPainter>
#include <QPropertyAnimation>
#include <QPushButton>
#include <QScrollBar>
#include <QShortcut>
#include <QTableWidget>
#include <QTextDocument>
#include <QTextEdit>
#include <QTime>
#include <QTimer>
#include <QTreeWidgetItem>
#include <QVBoxLayout>
#include <QtPrintSupport/QPrinter>
#include ”backend/grammar.hpp”
#include ”backend/ll1_parser.hpp”
#include ”lltabledialog.h”
Include dependency graph for lltutorwindow.h:

This graph shows which files directly or indirectly include this file:



Classes

- class LLTutorWindow

  Main window for the LL(1) interactive tutoring mode in SyntaxTutor.
- struct LLTutorWindow::TreeNode

  TreeNode structure used to build derivation trees.

Enumerations

- enum class State {
  A , A1 , A2 , A_prime ,
  B , B1 , B2 , B_prime ,
  C , C_prime , fin }

6.28.1 Enumeration Type Documentation

6.28.1.1 State

enum class State [strong]

Enumerator

A	
A1	
A2	
A_prime	
B	
B1	
B2	
B_prime	
C	
C_prime	
fin	

6.29 lltutorwindow.h

Go to the documentation of this file.

```
00001 #ifndef LLTUTORWINDOW_H
00002 #define LLTUTORWINDOW_H
00003
00004 #include <QAbstractItemView>
00005 #include <QDialog>
00006 #include <QFileDialog>
00007 #include <QGraphicsColorizeEffect>
00008 #include <QGraphicsScene>
00009 #include <QGraphicsTextItem>
00010 #include <QGraphicsView>
00011 #include <QListWidgetItem>
00012 #include <QMainWindow>
00013 #include <QMessageBox>
00014 #include <QPainter>
00015 #include <QPropertyAnimation>
00016 #include <QPushButton>
00017 #include <QScrollBar>
00018 #include <QShortcut>
00019 #include <QTableWidget>
00020 #include <QTextDocument>
00021 #include <QTextEdit>
00022 #include <QTime>
00023 #include <QTimer>
00024 #include <QTreeWidgetItem>
00025 #include <QVBoxLayout>
00026 #include <QtPrintSupport/QPrinter>
00027
00028 #include "backend/grammar.hpp"
00029 #include "backend/ll1_parser.hpp"
00030 #include "lltabledialog.h"
00031
00032 class TutorialManager;
00033
00034 namespace Ui {
00035 class LLTutorWindow;
00036 }
00037
00038 // ====== LL(1) Tutor States ====================================
00039 enum class State { A, A1, A2, A_prime, B, B1, B2, B_prime, C, C_prime, fin };
00040
00041 // ====== LL(1) Tutor Main Class ================================
00060 class LLTutorWindow : public QMainWindow
00061 {
00062     Q_OBJECT
00063
00064 public:
00065     // ====== Derivation Tree (used in TeachFirst) ==============
00069     struct TreeNode
00070     {
00071         QString label;
00072         std::vector<std::unique_ptr<TreeNode» children;
00073     };
00074
00075     // ====== Constructor / Destructor ==========================
00082     explicit LLTutorWindow(const Grammar &grammar,
00083                            TutorialManager *tm = nullptr,
00084                            QWidget *parent = nullptr);
00085     ~LLTutorWindow();
00086
00087     // ====== State Machine & Question Logic ====================
00092     QString generateQuestion();
00093
00098     void updateState(bool isCorrect);
00099
00105     QString FormatGrammar(const Grammar &grammar);
00106
00107     // ====== UI Interaction ====================================
00108     void addMessage(const QString &text, bool isUser);
00109     void addWidgetMessage(QWidget *widget);
00110     void exportConversationToPdf(const QString &filePath);
00111     void showTable();
00112     void showTableForCPrime();
00113     void updateProgressPanel(); // Update progress panel
00114
00115     // ====== Visual Feedback / Animations ======================
00116     void animateLabelPop(QLabel *label);
00117     void animateLabelColor(QLabel *label, const QColor &flashColor);
00118     void wrongAnimation();
00119     void wrongUserResponseAnimation();
00120     void markLastUserIncorrect();
00121
00122     // ====== Tree Generation (TeachFirst mode) =================
00123     void TeachFirstTree(const std::vector<std::string> &symbols,
```

```
00124                       std::unordered_set<std::string> &first_set,
00125                       int depth,
00126                       std::unordered_set<std::string> &processing,
00127                       QTreeWidgetItem *parent);
00128
00129     std::unique_ptr<TreeNode> buildTreeNode(
00130         const std::vector<std::string> &symbols,
00131         std::unordered_set<std::string> &first_set,
00132         int depth,
00133         std::vector<std::pair<std::string, std::vector<std::string>> &active_derivations);
00134
00135     int computeSubtreeWidth(const std::unique_ptr<TreeNode> &node, int hSpacing);
00136     void drawTree(const std::unique_ptr<TreeNode> &root,
00137                   QGraphicsScene *scene,
00138                   QPointF pos,
00139                   int hSpacing,
00140                   int vSpacing);
00141
00142     void showTreeGraphics(std::unique_ptr<TreeNode> root); // Display derivation tree visually
00143
00144     // ====== User Response Verification ========================
00145     bool verifyResponse(const QString &userResponse); // Delegates to current state's verification
00146     bool verifyResponseForA(const QString &userResponse);
00147     bool verifyResponseForA1(const QString &userResponse);
00148     bool verifyResponseForA2(const QString &userResponse);
00149     bool verifyResponseForB(const QString &userResponse);
00150     bool verifyResponseForB1(const QString &userResponse);
00151     bool verifyResponseForB2(const QString &userResponse);
00152     bool verifyResponseForC(); // C is non-textual (checks internal table)
00153
00154     // ====== Expected Solutions (Auto-generated) ================
00155     QString solution(const std::string &state);
00156     QStringList solutionForA();
00157     QString solutionForA1();
00158     QString solutionForA2();
00159     QSet<QString> solutionForB();
00160     QSet<QString> solutionForB1();
00161     QSet<QString> solutionForB2();
00162
00163     // ====== Feedback (Corrective Explanations) =================
00164     QString feedback(); // Delegates by state
00165     QString feedbackForA();
00166     QString feedbackForA1();
00167     QString feedbackForA2();
00168     QString feedbackForAPrime();
00169     QString feedbackForB();
00170     QString feedbackForB1();
00171     QString feedbackForB2();
00172     QString feedbackForBPrime();
00173     QString feedbackForC();
00174     QString feedbackForCPrime();
00175     void feedbackForB1TreeWidget();   // TreeWidget of Teach (LL1 TeachFirst)
00176     void feedbackForB1TreeGraphics(); // Show derivation tree
00177     QString TeachFollow(const QString &nt);
00178     QString TeachPredictionSymbols(const QString &ant, const production &conseq);
00179     QString TeachLL1Table();
00180
00181     void handleTableSubmission(const QVector<QVector<QString>> &raw, const QStringList &colHeaders);
00182 private slots:
00183     void on_confirmButton_clicked();
00184     void on_userResponse_textChanged();
00185
00186 signals:
00187     void sessionFinished(int cntRight, int cntWrong);
00188
00189 protected:
00190     void closeEvent(QCloseEvent *event) override
00191     {
00192         emit sessionFinished(cntRightAnswers, cntWrongAnswers);
00193         QWidget::closeEvent(event);
00194     }
00195
00196     bool eventFilter(QObject *obj, QEvent *event) override;
00197
00198 private:
00199     // ====== Core Objects ======================================
00200     Ui::LLTutorWindow *ui;
00201     Grammar grammar;
00202     LL1Parser ll1;
00203
00204     // ====== State & Grammar Tracking ==========================
00205     State currentState;
00206     size_t currentRule = 0;
00207     const unsigned kMaxHighlightTries = 3;
00208     const unsigned kMaxTotalTries = 5;
00209     unsigned lltries = 0;
00210     unsigned cntRightAnswers = 0, cntWrongAnswers = 0;
```

```
00211
00212     using Cell = std::pair<QString, QString>;
00213     std::vector<Cell> lastWrongCells;
00214     LLTableDialog *currentDlg = nullptr;
00215
00216     QVector<QString> sortedNonTerminals;
00217     QVector<QPair<QString, QVector<QString»> sortedGrammar;
00218     QString formattedGrammar;
00219
00220     QMap<QString, QMap<QString, QVector<QString»> lltable;
00221     QVector<QVector<QString» rawTable;
00222     QSet<QString> solutionSet;
00223
00224     // ====== Conversation Logging ================================
00225     struct MessageLog
00226     {
00227         QString message;
00228         bool isUser;
00229         bool isCorrect = true;
00230         MessageLog(const QString &message, bool isUser)
00231             : message(message)
00232             , isUser(isUser)
00233         {}
00234         void toggleIsCorrect() { isCorrect = false; }
00235     };
00236
00237     QVector<MessageLog> conversationLog;
00238     QWidget *lastUserMessage = nullptr;
00239     qsizetype lastUserMessageLogIdx = -1;
00240
00241     QMap<QString, QString> userCAB;
00242     QMap<QString, QString> userSIG;
00243     QMap<QString, QString> userSD;
00244
00245     // ====== Helper Conversions ==================================
00246     std::vector<std::string> qvectorToStdVector(const QVector<QString> &qvec);
00247     QVector<QString> stdVectorToQVector(const std::vector<std::string> &vec);
00248     QSet<QString> stdUnorderedSetToQSet(const std::unordered_set<std::string> &uset);
00249     std::unordered_set<std::string> qsetToStdUnorderedSet(const QSet<QString> &qset);
00250
00251     void setupTutorial();
00252
00253     void fillSortedGrammar(); // Populate sortedGrammar from internal representation
00254
00255     QPropertyAnimation *m_shakeAnimation
00256         = nullptr; // For interrupting userResponse animation if they spam enter key
00257
00258     TutorialManager *tm = nullptr;
00259
00260     QRegularExpression re{"^\\s+|\\s+$"};
00261 };
00262
00263 #endif // LLTUTORWINDOW_H
```

6.30 main.cpp File Reference

#include "mainwindow.h"
#include <QApplication>
#include <QFont>
#include <QFontDatabase>
#include <QImageReader>
#include <QSettings>
#include <QTranslator>
Include dependency graph for main.cpp:

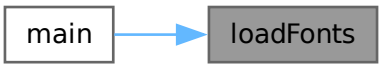
Functions

- void loadFonts ()
- int main (int argc, char *argv[])

6.30.1 Function Documentation

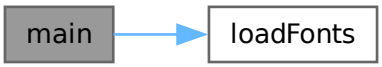
6.30.1.1 loadFonts()

void loadFonts ()
Here is the caller graph for this function:



6.30.1.2 main()

int main (
int argc,
char ∗ argv[])
Here is the call graph for this function:



6.31 mainwindow.cpp File Reference

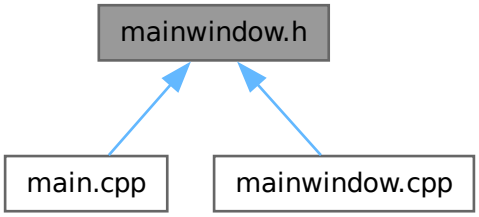
#include ”mainwindow.h”
#include ”tutorialmanager.h”
#include ”ui_mainwindow.h”
#include <QMessageBox>
#include <QPixmap>
#include <QProcess>
Include dependency graph for mainwindow.cpp:

6.32 mainwindow.h File Reference

#include <QMainWindow>
#include <QSettings>
#include ”backend/grammar.hpp”
#include ”backend/grammar_factory.hpp”
#include ”lltutorwindow.h”

#include ”slrtutorwindow.h”
#include ”tutorialmanager.h”
Include dependency graph for mainwindow.h:

This graph shows which files directly or indirectly include this file:



Classes

- class MainWindow

  Main application window of SyntaxTutor, managing levels, exercises, and UI state.

6.33 mainwindow.h

Go to the documentation of this file.

```
00001 #ifndef MAINWINDOW_H
00002 #define MAINWINDOW_H
00003
00004 #include <QMainWindow>
00005 #include <QSettings>
00006 #include ”backend/grammar.hpp”
00007 #include ”backend/grammar_factory.hpp”
00008 #include ”lltutorwindow.h”
00009 #include ”slrtutorwindow.h”
00010 #include ”tutorialmanager.h”
00011
00012 static const QVector<QString> levelColors = {
00013     ”#2C3E50”, // 1: Navy oscuro
00014     ”#2980B9”, // 2: Azul brillante
00015     ”#16A085”, // 3: Teal
00016     ”#27AE60”, // 4: Verde esmeralda
00017     ”#8E44AD”, // 5: Púrpura medio
00018     ”#9B59B6”, // 6: Púrpura claro
00019     ”#E67E22”, // 7: Naranja
00020     ”#D35400”, // 8: Naranja oscuro
00021     ”#CD7F32”, // 9: Bronce
00022     ”#FFD700”  // 10: Oro puro
00023 };
00024
00025 QT_BEGIN_NAMESPACE
00026 namespace Ui {
00027 class MainWindow;
00028 }
00029 QT_END_NAMESPACE
00030
00040 class MainWindow : public QMainWindow
00041 {
00042     Q_OBJECT
00043     Q_PROPERTY(unsigned userLevel READ userLevel WRITE setUserLevel NOTIFY userLevelChanged)
00044
```

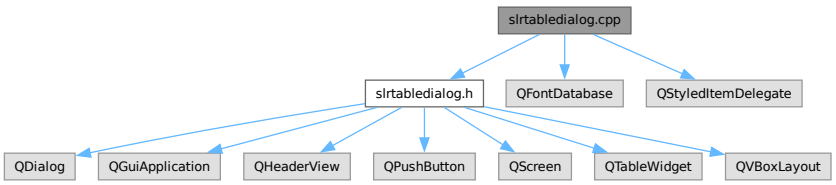
```
00045 public:
00050     MainWindow(QWidget *parent = nullptr);
00051
00053     ~MainWindow();
00054
00060     unsigned thresholdFor(unsigned level) { return BASE_THRESHOLD * level; }
00061
00065     unsigned userLevel() const { return m_userLevel; };
00066
00071     void setUserLevel(unsigned lvl)
00072     {
00073         unsigned clamped = qMin(lvl, MAX_LEVEL);
00074         if (m_userLevel == clamped)
00075             return;
00076         m_userLevel = clamped;
00077         emit userLevelChanged(clamped);
00078     }
00079
00080 private slots:
00084     void on_lv1Button_clicked(bool checked);
00085     void on_lv2Button_clicked(bool checked);
00086     void on_lv3Button_clicked(bool checked);
00087
00091     void on_pushButton_clicked();
00092
00096     void on_pushButton_2_clicked();
00097
00101     void on_tutorial_clicked();
00102
00106     void on_actionSobre_la_aplicaci_n_triggered();
00107
00111     void on_actionReferencia_LL_1_triggered();
00112
00116     void on_actionReferencia_SLR_1_triggered();
00117
00121     void on_idiom_clicked();
00122
00123 signals:
00128     void userLevelChanged(unsigned lvl);
00129
00134     void userLevelUp(unsigned newLevel);
00135
00136 private:
00140     void setupTutorial();
00141
00145     void restartTutorial();
00146
00152     void handleTutorFinished(int cntRight, int cntWrong);
00153
00157     void saveSettings();
00158
00162     void loadSettings();
00163
00164     Ui::MainWindow *ui;
00165     GrammarFactory factory;
00166     int level = 1;
00167     TutorialManager *tm = nullptr;
00168
00169     static constexpr unsigned MAX_LEVEL = 10;
00170     static constexpr unsigned MAX_SCORE = 999;
00171
00172     unsigned m_userLevel = 1;
00173     unsigned userScore = 0;
00174     QSettings settings;
00175
00176     const unsigned BASE_THRESHOLD = 10;
00177 };
00178 #endif // MAINWINDOW_H
```

6.34 README.md File Reference

6.35 slrtabledialog.cpp File Reference

#include ”slrtabledialog.h”
#include <QFontDatabase>
#include <QStyledItemDelegate>

Include dependency graph for slrtabledialog.cpp:

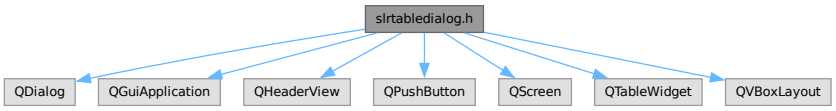


Classes

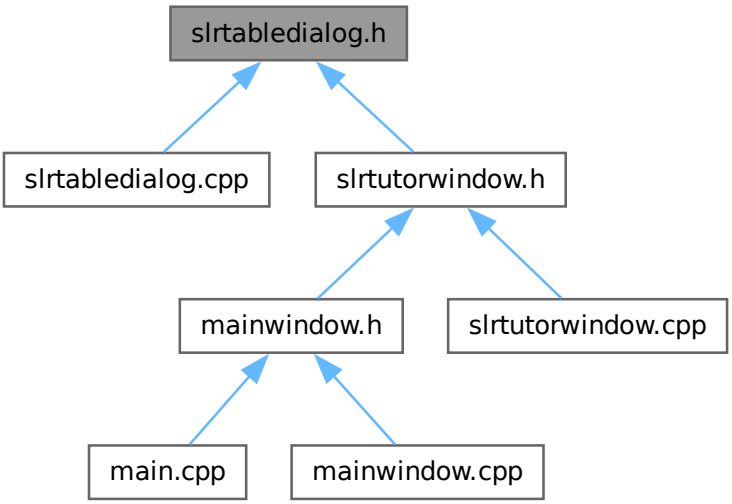
- class CenterAlignDelegate

6.36 slrtabledialog.h File Reference

#include <QDialog>
#include <QGuiApplication>
#include <QHeaderView>
#include <QPushButton>
#include <QScreen>
#include <QTableWidget>
#include <QVBoxLayout>
Include dependency graph for slrtabledialog.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SLRTableDialog

  Dialog window for completing and submitting an SLR(1) parsing table.

6.37 slrtabledialog.h

Go to the documentation of this file.

```
00001 #ifndef SLRTABLEDIALOG_H
00002 #define SLRTABLEDIALOG_H
00003
00004 #include <QDialog>
00005 #include <QGuiApplication>
00006 #include <QHeaderView>
00007 #include <QPushButton>
00008 #include <QScreen>
00009 #include <QTableWidget>
00010 #include <QVBoxLayout>
00011
00020 class SLRTableDialog : public QDialog
00021 {
00022     Q_OBJECT
00023 public:
00033     SLRTableDialog(int rowCount,
00034                    int colCount,
00035                    const QStringList &colHeaders,
00036                    QWidget *parent = nullptr,
00037                    QVector<QVector<QString>> *initialData = nullptr);
00038
00043     QVector<QVector<QString>> getTableData() const;
00044
00052     void setInitialData(const QVector<QVector<QString>> &data);
00053
00054 private:
00055     QTableWidget *table;
00056     QPushButton *submitButton;
00057 };
00058
00059 #endif // SLRTABLEDIALOG_H
```

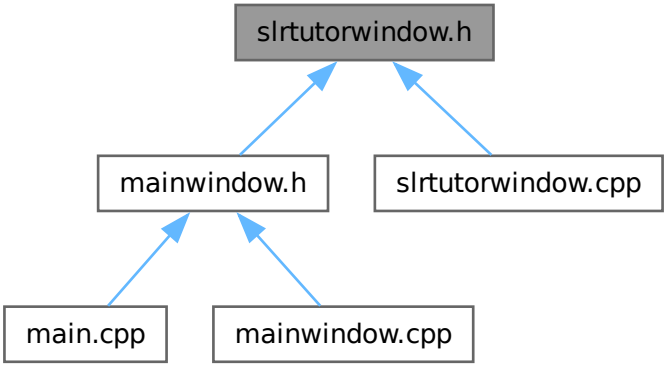
6.38 slrtutorwindow.cpp File Reference

#include ”slrtutorwindow.h”
#include ”tutorialmanager.h”
#include ”ui_slrtutorwindow.h”
#include <QEasingCurve>
#include <QFontDatabase>
#include <sstream>
#include ”slrwizard.h”
Include dependency graph for slrtutorwindow.cpp:

6.39 slrtutorwindow.h File Reference

#include <QAbstractItemView>
#include <QDialog>
#include <QFileDialog>
#include <QGraphicsColorizeEffect>
#include <QListWidgetItem>
#include <QMainWindow>
#include <QMessageBox>
#include <QPropertyAnimation>
#include <QPushButton>
#include <QRegularExpression>
#include <QScrollBar>
#include <QShortcut>
#include <QTableWidget>
#include <QTextDocument>
#include <QTextEdit>
#include <QTime>
#include <QTimer>
#include <QVBoxLayout>
#include <QtPrintSupport/QPrinter>
#include ”UniqueQueue.h”
#include ”backend/grammar.hpp”
#include ”backend/slr1_parser.hpp”
#include ”slrtabledialog.h”
Include dependency graph for slrtutorwindow.h:

This graph shows which files directly or indirectly include this file:



Classes

- class SLRTutorWindow

  Main window for the SLR(1) interactive tutoring mode in SyntaxTutor.

Enumerations

- enum class StateSlr {
  A , A1 , A2 , A3 ,
  A4 , A_prime , B , C ,
  CA , CB , D , D1 ,
  D2 , D_prime , E , E1 ,
  E2 , F , FA , G ,
  H , H_prime , fin }

6.39.1 Enumeration Type Documentation

6.39.1.1 StateSlr

enum class StateSlr [strong]

Enumerator

A	
A1	
A2	
A3	
A4	
A_prime	
B	
C	
CA	
CB	
D	
D1	
D2	

Enumerator

D_prime	
E	
E1	
E2	
F	
FA	
G	
H	
H_prime	
fin	

6.40 slrtutorwindow.h

Go to the documentation of this file.

```
00001 #ifndef SLRTUTORWINDOW_H
00002 #define SLRTUTORWINDOW_H
00003
00004 #include <QAbstractItemView>
00005 #include <QDialog>
00006 #include <QFileDialog>
00007 #include <QGraphicsColorizeEffect>
00008 #include <QListWidgetItem>
00009 #include <QMainWindow>
00010 #include <QMessageBox>
00011 #include <QPropertyAnimation>
00012 #include <QPushButton>
00013 #include <QRegularExpression>
00014 #include <QScrollBar>
00015 #include <QShortcut>
00016 #include <QTableWidget>
00017 #include <QTextDocument>
00018 #include <QTextEdit>
00019 #include <QTime>
00020 #include <QTimer>
00021 #include <QVBoxLayout>
00022 #include <QtPrintSupport/QPrinter>
00023 #include "UniqueQueue.h"
00024 #include "backend/grammar.hpp"
00025 #include "backend/slr1_parser.hpp"
00026 #include "slrtabledialog.h"
00027
00028 namespace Ui {
00029 class SLRTutorWindow;
00030 }
00031
00032 // ====== SLR(1) Tutor States ======================================
00033 enum class StateSlr {
00034     A,
00035     A1,
00036     A2,
00037     A3,
00038     A4,
00039     A_prime,
00040     B,
00041     C,
00042     CA,
00043     CB,
00044     D,
00045     D1,
00046     D2,
00047     D_prime,
00048     E,
00049     E1,
00050     E2,
00051     F,
00052     FA,
00053     G,
00054     H,
00055     H_prime,
00056     fin
00057 };
00058
```

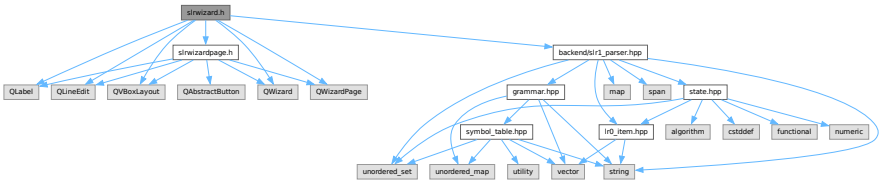
```
00059 class TutorialManager;
00060
00061 // ====== Main Tutor Class for SLR(1) ================================
00076 class SLRTutorWindow : public QMainWindow
00077 {
00078     Q_OBJECT
00079
00080 public:
00081     // ====== Constructor / Destructor ================================
00088     explicit SLRTutorWindow(const Grammar &g,
00089                             TutorialManager *tm = nullptr,
00090                             QWidget *parent = nullptr);
00091     ~SLRTutorWindow();
00092
00093     // ====== Core Flow Control ========================================
00098     QString generateQuestion();
00099
00104     void updateState(bool isCorrect);
00105     QString FormatGrammar(const Grammar &grammar);
00106     void fillSortedGrammar();
00107
00108     // ====== UI Interaction ===========================================
00109     void addMessage(const QString &text, bool isUser);
00110     void exportConversationToPdf(const QString &filePath);
00111     void showTable();
00112     void launchSLRWizard();
00113     void updateProgressPanel();
00114     void addUserState(unsigned id);
00115     void addUserTransition(unsigned fromId,
00116                            const std::string &symbol,
00117                            unsigned toId); // Register a user-created transition
00118
00119     // ====== Visual Feedback & Animations ============================
00120     void animateLabelPop(QLabel *label);
00121     void animateLabelColor(QLabel *label, const QColor &flashColor);
00122     void wrongAnimation();              // Label animation for incorrect answer
00123     void wrongUserResponseAnimation(); // Message widget animation for incorrect answer
00124     void markLastUserIncorrect();
00125
00126     // ====== Response Verification ====================================
00127     bool verifyResponse(const QString &userResponse);
00128     bool verifyResponseForA(const QString &userResponse);
00129     bool verifyResponseForA1(const QString &userResponse);
00130     bool verifyResponseForA2(const QString &userResponse);
00131     bool verifyResponseForA3(const QString &userResponse);
00132     bool verifyResponseForA4(const QString &userResponse);
00133     bool verifyResponseForB(const QString &userResponse);
00134     bool verifyResponseForC(const QString &userResponse);
00135     bool verifyResponseForCA(const QString &userResponse);
00136     bool verifyResponseForCB(const QString &userResponse);
00137     bool verifyResponseForD(const QString &userResponse);
00138     bool verifyResponseForD1(const QString &userResponse);
00139     bool verifyResponseForD2(const QString &userResponse);
00140     bool verifyResponseForE(const QString &userResponse);
00141     bool verifyResponseForE1(const QString &userResponse);
00142     bool verifyResponseForE2(const QString &userResponse);
00143     bool verifyResponseForF(const QString &userResponse);
00144     bool verifyResponseForFA(const QString &userResponse);
00145     bool verifyResponseForG(const QString &userResponse);
00146     bool verifyResponseForH();
00147
00148     // ====== Correct Solutions (Auto-generated) ======================
00149     QString solution(const std::string &state);
00150     std::unordered_set<Lr0Item> solutionForA();
00151     QString solutionForA1();
00152     QString solutionForA2();
00153     std::vector<std::pair<std::string, std::vector<std::string>>> solutionForA3();
00154     std::unordered_set<Lr0Item> solutionForA4();
00155     unsigned solutionForB();
00156     unsigned solutionForC();
00157     QStringList solutionForCA();
00158     std::unordered_set<Lr0Item> solutionForCB();
00159     QString solutionForD();
00160     QString solutionForD1();
00161     QString solutionForD2();
00162     std::ptrdiff_t solutionForE();
00163     QSet<unsigned> solutionForE1();
00164     QMap<unsigned, unsigned> solutionForE2();
00165     QSet<unsigned> solutionForF();
00166     QSet<QString> solutionForFA();
00167     QSet<QString> solutionForG();
00168
00169     // ====== Pedagogical Feedback ====================================
00170     QString feedback(); // Delegates to appropriate feedback based on state
00171     QString feedbackForA();
00172     QString feedbackForA1();
00173     QString feedbackForA2();
```

```
00174     QString feedbackForA3();
00175     QString feedbackForA4();
00176     QString feedbackForAPrime();
00177     QString feedbackForB();
00178     QString feedbackForB1();
00179     QString feedbackForB2();
00180     QString feedbackForBPrime();
00181     QString feedbackForC();
00182     QString feedbackForCA();
00183     QString feedbackForCB();
00184     QString feedbackForD();
00185     QString feedbackForD1();
00186     QString feedbackForD2();
00187     QString feedbackForDPrime();
00188     QString feedbackForE();
00189     QString feedbackForE1();
00190     QString feedbackForE2();
00191     QString feedbackForF();
00192     QString feedbackForFA();
00193     QString feedbackForG();
00194     QString TeachDeltaFunction(const std::unordered_set<Lr0Item> &items, const QString &symbol);
00195     void TeachClosureStep(std::unordered_set<Lr0Item> &items,
00196                           unsigned int size,
00197                           std::unordered_set<std::string> &visited,
00198                           int depth,
00199                           QString &output);
00200     QString TeachClosure(const std::unordered_set<Lr0Item> &initialItems);
00201 private slots:
00202     void on_confirmButton_clicked();
00203     void on_userResponse_textChanged();
00204
00205 signals:
00206     void sessionFinished(int cntRight, int cntWrong);
00207
00208 protected:
00209     void closeEvent(QCloseEvent *event) override
00210     {
00211         emit sessionFinished(cntRightAnswers, cntWrongAnswers);
00212         QWidget::closeEvent(event);
00213     }
00214
00215 private:
00216     // ====== Helper Functions ========================================
00217     std::vector<std::string> qvectorToStdVector(const QVector<QString> &qvec);
00218     QVector<QString> stdVectorToQVector(const std::vector<std::string> &vec);
00219     QSet<QString> stdUnorderedSetToQSet(const std::unordered_set<std::string> &uset);
00220     std::unordered_set<std::string> qsetToStdUnorderedSet(const QSet<QString> &qset);
00221     std::unordered_set<Lr0Item> ingestUserItems(const QString &userResponse);
00222     std::vector<std::pair<std::string, std::vector<std::string>>> ingestUserRules(
00223         const QString &userResponse);
00224     void setupTutorial();
00225     // ====== Core Components =========================================
00226     Ui::SLRTutorWindow *ui;
00227     Grammar grammar;
00228     SLR1Parser slr1;
00229
00230     // ====== State and Grammar Tracking ==============================
00231     StateSlr currentState;
00232     QVector<QString> sortedNonTerminals;
00233     QVector<QPair<QString, QVector<QString>>> sortedGrammar;
00234     QString formattedGrammar;
00235
00236     unsigned cntRightAnswers = 0;
00237     unsigned cntWrongAnswers = 0;
00238
00239     // ====== State Machine Runtime Variables =========================
00240     std::unordered_set<state> userMadeStates; // All states the user has created
00241     std::unordered_map<unsigned, std::unordered_map<std::string, unsigned>
00242         userMadeTransitions;            // Transitions made by the user
00243     UniqueQueue<unsigned> statesIdQueue; // States to be processed in B-C-CA-CB loop
00244     unsigned currentStateId = 0;
00245     state currentSlrState;
00246
00247     QStringList followSymbols; // Used in CA-CB loop
00248     qsizetype currentFollowSymbolsIdx = 0;
00249     unsigned int nextStateId = 0;
00250
00251     QVector<const state *> statesWithLr0Conflict; // Populated in F
00252     std::queue<unsigned> conflictStatesIdQueue;
00253     unsigned currentConflictStateId = 0;
00254     state currentConflictState;
00255
00256     std::queue<unsigned> reduceStatesIdQueue; // States without conflicts but with reduce
00257     unsigned currentReduceStateId = 0;
00258     state currentReduceState;
00259
00260     struct ActionEntry
```

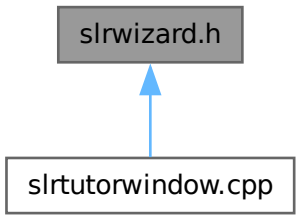
```
00261     {
00262         enum Type { Shift, Reduce, Accept, Goto } type;
00263         int target;
00264         static ActionEntry makeShift(int s) { return {Shift, s}; }
00265         static ActionEntry makeReduce(int r) { return {Reduce, r}; }
00266         static ActionEntry makeAccept() { return {Accept, 0}; }
00267         static ActionEntry makeGoto(int g) { return {Goto, g}; }
00268     };
00269
00270     QMap<int, QMap<QString, ActionEntry» slrtable;
00271     QVector<QVector<QString» rawTable;
00272
00273     // ====== Conversation Log ========================================
00274     struct MessageLog
00275     {
00276         QString message;
00277         bool isUser;
00278         bool isCorrect;
00279
00280         MessageLog(const QString &message, bool isUser)
00281             : message(message)
00282             , isUser(isUser)
00283         {}
00284
00285         void toggleIsCorrect() { isCorrect = false; }
00286     };
00287
00288     QVector<MessageLog> conversationLog;
00289     QWidget *lastUserMessage = nullptr;
00290     qsizetype lastUserMessageLogIdx = -1;
00291
00292     QPropertyAnimation *m_shakeAnimation
00293         = nullptr; // For interrupting userResponse animation if they spam enter key
00294
00295     TutorialManager *tm;
00296
00297     QRegularExpression re{"^\\s+|\\s+$"};
00298 };
00299
00300 #endif // SLRTUTORWINDOW_H
```

6.41 slrwizard.h File Reference

#include <QLabel>
#include <QLineEdit>
#include <QVBoxLayout>
#include <QWizard>
#include <QWizardPage>
#include "backend/slr1_parser.hpp"
#include "slrwizardpage.h"
Include dependency graph for slrwizard.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SLRWizard

  Interactive assistant that guides the student step-by-step through the SLR(1) parsing table.

6.42 slrwizard.h

Go to the documentation of this file.

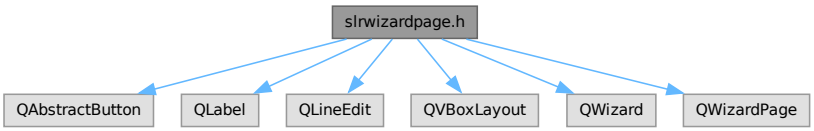
```
00001 #ifndef SLRWIZARD_H
00002 #define SLRWIZARD_H
00003
00004 #include <QLabel>
00005 #include <QLineEdit>
00006 #include <QVBoxLayout>
00007 #include <QWizard>
00008 #include <QWizardPage>
00009 #include "backend/slr1_parser.hpp"
00010 #include "slrwizardpage.h"
00011
00025 class SLRWizard : public QWizard
00026 {
00027     Q_OBJECT
00028 public:
00038     SLRWizard(SLR1Parser &parser,
00039               const QVector<QVector<QString>> &rawTable,
00040               const QStringList &colHeaders,
00041               const QVector<QPair<QString, QVector<QString>>> &sortedGrammar,
00042               QWidget *parent = nullptr)
00043         : QWizard(parent)
00044     {
00045         setWindowTitle(tr("Ayuda interactiva: Tabla SLR(1)"));
00046
00047         const int nTerm = parser.gr_.st_.terminals_.contains(parser.gr_.st_.EPSILON_)
00048                               ? parser.gr_.st_.terminals_.size() - 1
00049                               : parser.gr_.st_.terminals_.size();
00050         SLRWizardPage *last = nullptr;
00051         // Generar explicación y páginas
00052         int rows = rawTable.size();
00053         int cols = colHeaders.size();
00054         for (int i = 0; i < rows; ++i) {
00055             for (int j = 0; j < cols; ++j) {
00056                 QString sym = colHeaders[j];
00057                 QString expected;
00058                 QString explanation;
00059                 if (j < nTerm) {
00060                     auto itAct = parser.actions_.at(i).find(sym.toStdString());
00061                     SLR1Parser::s_action act
00062                         = (itAct != parser.actions_.at(i).end()
00063                                ? itAct->second
00064                                : SLR1Parser::s_action{nullptr, SLR1Parser::Action::Empty});
00065                     switch (act.action) {
00066                     case SLR1Parser::Action::Shift: {
00067                         unsigned to = parser.transitions_.at(i).at(sym.toStdString());
00068                         expected = QString("s%1").arg(to);
00069                         explanation = tr("Estado %1: existe transición  (%1, '%2'). ¿A qué "
00070                                          "estado harías shift?")
00071                                           .arg(i)
```

```
00072                                   .arg(sym);
00073                 break;
00074             }
00075             case SLR1Parser::Action::Reduce: {
00076                 int idx = -1;
00077                 for (int k = 0; k < sortedGrammar.size(); ++k) {
00078                     auto &rule = sortedGrammar[k];
00079                     if (rule.first.toStdString() == act.item->antecedent_
00080                         && stdVectorToQVector(act.item->consequent_) == rule.second) {
00081                         idx = k;
00082                         break;
00083                     }
00084                 }
00085                 expected = QString("r%1").arg(idx);
00086                 // explicación con FOLLOW
00087                 std::unordered_set<std::string> F;
00088                 F = parser.Follow(act.item->antecedent_);
00089                 QStringList followList;
00090                 for (auto &t : F)
00091                     followList << QString::fromStdString(t);
00092                 explanation = tr("Estado %1: contiene el ítem [%2 → ... · ] y '%3'   "
00093                                  "SIG(%2). ¿Qué regla usas para reducir (0, 1, ...)?")
00094                                  .arg(i)
00095                                  .arg(QString::fromStdString(act.item->antecedent_))
00096                                  .arg(colHeaders[j]);
00097                 break;
00098             }
00099             case SLR1Parser::Action::Accept:
00100                 expected = "acc";
00101                 explanation = tr("Estado %1: contiene [S → A  ·  $]. ¿Qué palabra clave "
00102                                  "usas para aceptar?")
00103                                  .arg(i);
00104                 break;
00105             case SLR1Parser::Action::Empty:
00106             default:
00107                 continue;
00108             }
00109         } else {
00110             // GOTO sobre no terminal
00111             auto nonT = sym.toStdString();
00112             if (!parser.transitions_.contains(i)) {
00113                 continue;
00114             }
00115             auto itGo = parser.transitions_.at(i).find(nonT);
00116             if (itGo != parser.transitions_.at(i).end()) {
00117                 expected = QString::number(itGo->second);
00118                 explanation = tr("Estado %1:  (%1, '%2') existe. ¿A qué estado va "
00119                                  "la transición? (pon solo el número)")
00120                                  .arg(i)
00121                                  .arg(sym);
00122             } else {
00123                 continue;
00124             }
00125         }
00126
00127         SLRWizardPage *page = new SLRWizardPage(i, sym, explanation, expected, this);
00128         last = page;
00129         addPage(page);
00130     }
00131 }
00132 if (last) {
00133     last->setFinalPage(true);
00134 }
00135 }
00136
00142 QVector<QString> stdVectorToQVector(const std::vector<std::string> &vec)
00143 {
00144     QVector<QString> result;
00145     result.reserve(vec.size());
00146     for (const auto &str : vec) {
00147         result.push_back(QString::fromStdString(str));
00148     }
00149     return result;
00150 }
00151 };
00152
00153 #endif // SLRWIZARD_H
```

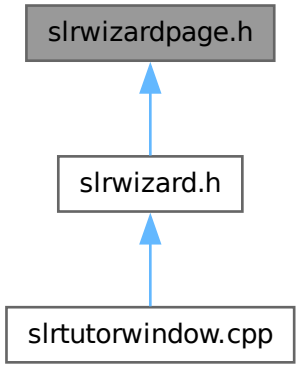
6.43 slrwizardpage.h File Reference

#include <QAbstractButton>
#include <QLabel>
#include <QLineEdit>

#include <QVBoxLayout>
#include <QWizard>
#include <QWizardPage>
Include dependency graph for slrwizardpage.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SLRWizardPage

  A single step in the SLR(1) guided assistant for table construction.

6.44 slrwizardpage.h

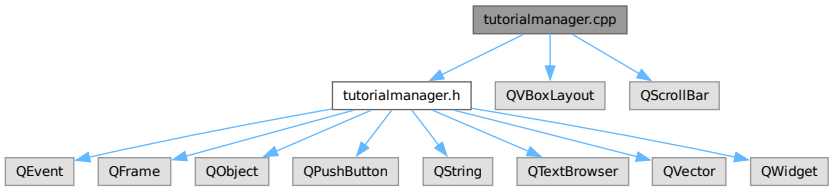
Go to the documentation of this file.

```
00001 #ifndef SLRWIZARDPAGE_H
00002 #define SLRWIZARDPAGE_H
00003
00004 #include <QAbstractButton>
00005 #include <QLabel>
00006 #include <QLineEdit>
00007 #include <QVBoxLayout>
00008 #include <QWizard>
00009 #include <QWizardPage>
00010
00021 class SLRWizardPage : public QWizardPage
00022 {
00023     Q_OBJECT
00024 public:
00034     SLRWizardPage(int state,
00035                   const QString &symbol,
00036                   const QString &explanation,
00037                   const QString &expected,
00038                   QWidget *parent = nullptr)
00039         : QWizardPage(parent)
```

```
00040         , m_state(state)
00041         , m_symbol(symbol)
00042         , m_expected(expected)
00043     {
00044         setTitle(tr("Estado %1, símbolo '%2'").arg(state).arg(symbol));
00045 
00046         QLabel *lbl = new QLabel(explanation, this);
00047         lbl->setWordWrap(true);
00048 
00049         m_edit = new QLineEdit(this);
00050         m_edit->setPlaceholderText(tr("Escribe tu respuesta (p.ej. s3, r2, acc, 5)"));
00051 
00052         QVBoxLayout *layout = new QVBoxLayout(this);
00053         layout->addWidget(lbl);
00054         layout->addWidget(m_edit);
00055         setLayout(layout);
00056 
00057         connect(m_edit, &QLineEdit::textChanged, this, &SLRWizardPage::onTextChanged);
00058     }
00059 private slots:
00064     void onTextChanged(const QString &text)
00065     {
00066         bool correct = (text.trimmed() == m_expected);
00067         setComplete(correct);
00068         if (correct) {
00069             setSubTitle(tr("  Respuesta correcta, pasa a la siguiente pregunta"));
00070         } else {
00071             setSubTitle(
00072                 tr("  Incorrecto, revisa el enunciado. Consulta los estados que has construido."));
00073         }
00074         wizard()->button(QWizard::NextButton)->setEnabled(correct);
00075     }
00076 
00077 private:
00082     void setComplete(bool complete)
00083     {
00084         m_isComplete = complete;
00085         emit completeChanged();
00086     }
00087 
00092     bool isComplete() const override { return m_isComplete; }
00093 
00094     int m_state;
00095     QString m_symbol;
00096     QString m_expected;
00097     QLineEdit *m_edit;
00098     bool m_isComplete = false;
00099 };
00100 
00101 #endif // SLRWIZARDPAGE_H
```

6.45 tutorialmanager.cpp File Reference

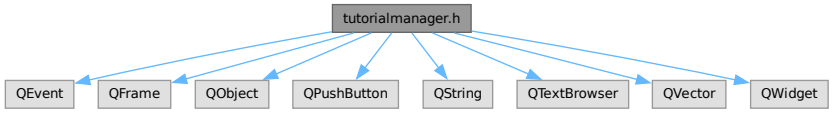
#include "tutorialmanager.h"
#include <QVBoxLayout>
#include <QScrollBar>
Include dependency graph for tutorialmanager.cpp:



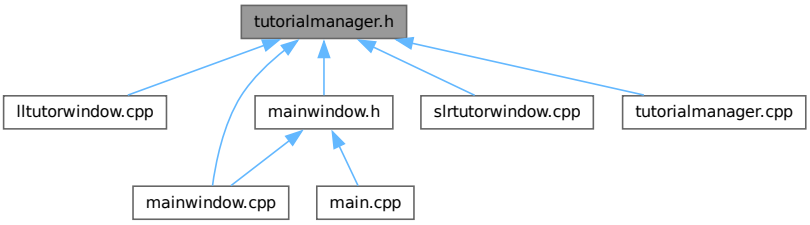
6.46 tutorialmanager.h File Reference

#include <QEvent>
#include <QFrame>
#include <QObject>

#include <QPushButton>
#include <QString>
#include <QTextBrowser>
#include <QVector>
#include <QWidget>
Include dependency graph for tutorialmanager.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct TutorialStep

  Represents a single step in the tutorial sequence.
- class TutorialManager

  Manages interactive tutorials by highlighting UI elements and guiding the user.

6.47 tutorialmanager.h

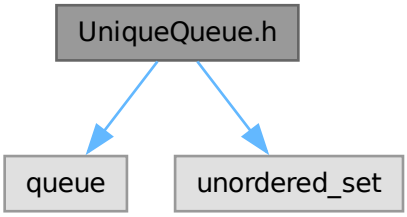
Go to the documentation of this file.

```
00001 #ifndef TUTORIALMANAGER_H
00002 #define TUTORIALMANAGER_H
00003
00004 #include <QEvent>
00005 #include <QFrame>
00006 #include <QObject>
00007 #include <QPushButton>
00008 #include <QString>
00009 #include <QTextBrowser>
00010 #include <QVector>
00011 #include <QWidget>
00012
00019 struct TutorialStep
00020 {
00021     QWidget *target;
00022     QString htmlText;
00023 };
00024
00034 class TutorialManager : public QObject
00035 {
00036     Q_OBJECT
00037 public:
00042     TutorialManager(QWidget *rootWindow);
00043
00049     void addStep(QWidget *target, const QString &htmlText);
```

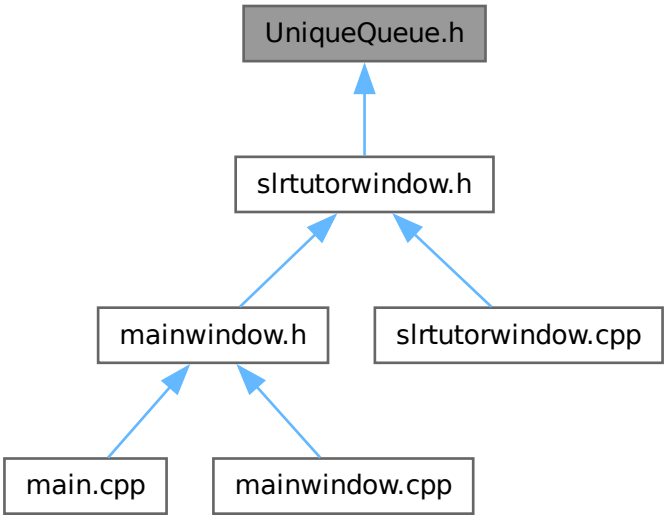
```
00050
00054     void start();
00055
00060     void setRootWindow(QWidget *newRoot);
00061
00065     void clearSteps();
00066
00070     void hideOverlay();
00071
00075     void finishLL1();
00076
00080     void finishSLR1();
00081
00082 protected:
00086     bool eventFilter(QObject *obj, QEvent *ev) override;
00087
00088 signals:
00093     void stepStarted(int index);
00094
00098     void tutorialFinished();
00099
00103     void ll1Finished();
00104
00108     void slr1Finished();
00109
00110 public slots:
00114     void nextStep();
00115
00116 private:
00120     void showOverlay();
00121
00125     void repositionOverlay();
00126
00127     QWidget *m_root;
00128     QVector<TutorialStep> m_steps;
00129     int m_index = -1;
00130
00131     QWidget *m_overlay = nullptr;
00132     QFrame *m_highlight = nullptr;
00133     QTextBrowser *m_textBox = nullptr;
00134     QPushButton *m_nextBtn = nullptr;
00135 };
00136
00137 #endif // TUTORIALMANAGER_H
```

6.48 UniqueQueue.h File Reference

#include <queue>
#include <unordered_set>
Include dependency graph for UniqueQueue.h:



This graph shows which files directly or indirectly include this file:



Classes

- class UniqueQueue< T >

  A queue that ensures each element is inserted only once.

6.49 UniqueQueue.h

Go to the documentation of this file.

```
00001 #ifndef UNIQUEQUEUE_H
00002 #define UNIQUEQUEUE_H
00003 #include <queue>
00004 #include <unordered_set>
00005
00016 template<typename T>
00017 class UniqueQueue {
00018 public:
00023     void push(const T& value) {
00024         if (seen_.insert(value).second) {
00025             queue_.push(value);
00026         }
00027     }
00028
00032     void pop() {
00033         if (!queue_.empty()) {
00034             queue_.pop();
00035         }
00036     }
00037
00042     const T& front() const {
00043         return queue_.front();
00044     }
00045
00050     bool empty() const {
00051         return queue_.empty();
00052     }
00053
00057     void clear() {
00058         while(!queue_.empty()) queue_.pop();
00059         seen_.clear();
00060     }
00061
00062 private:
00063     std::queue<T> queue_;
```

```
00064     std::unordered_set<T> seen_;
00065 };
00066 #endif // UNIQUEQUEUE_H
```