SyntaxTutor

Generated by Doxygen 1.14.0

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

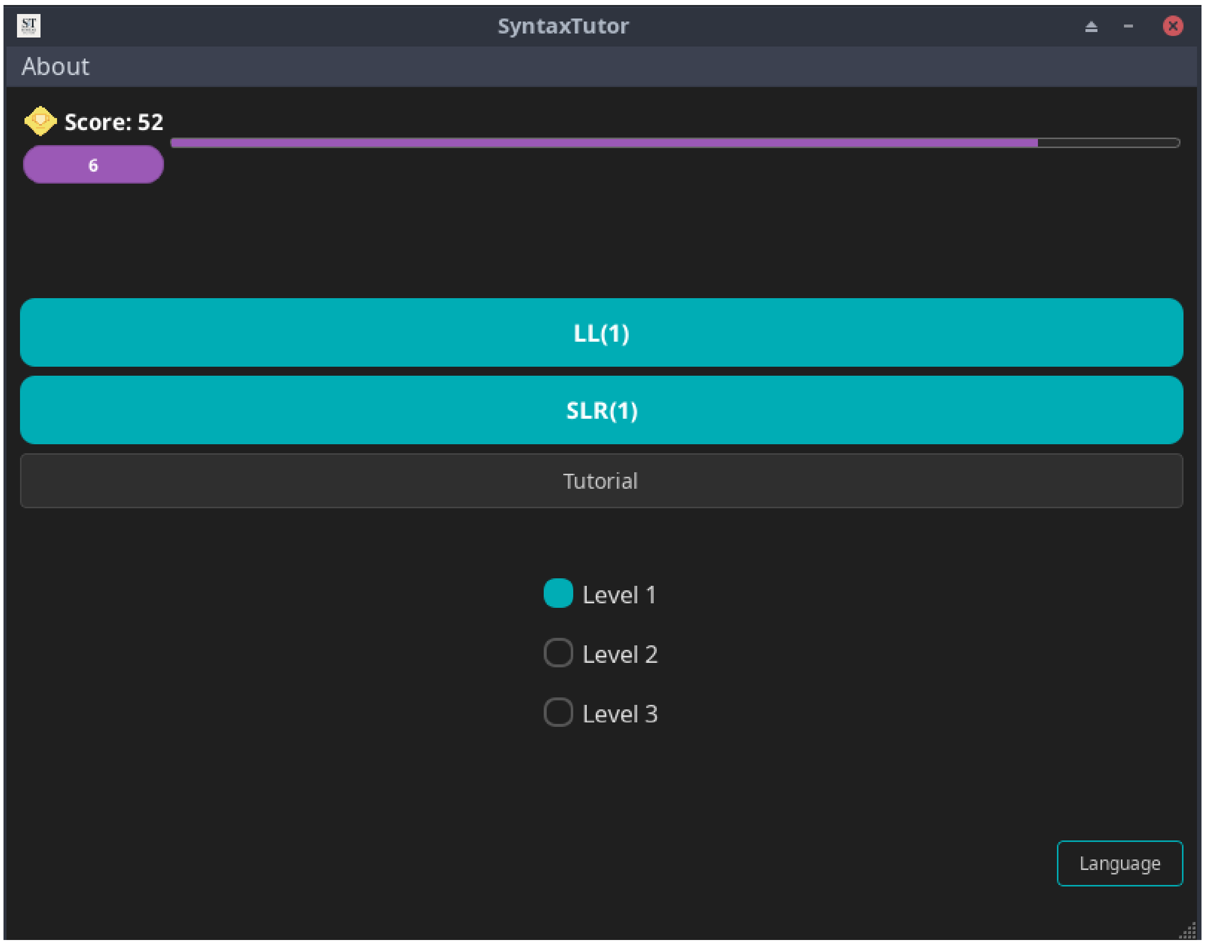
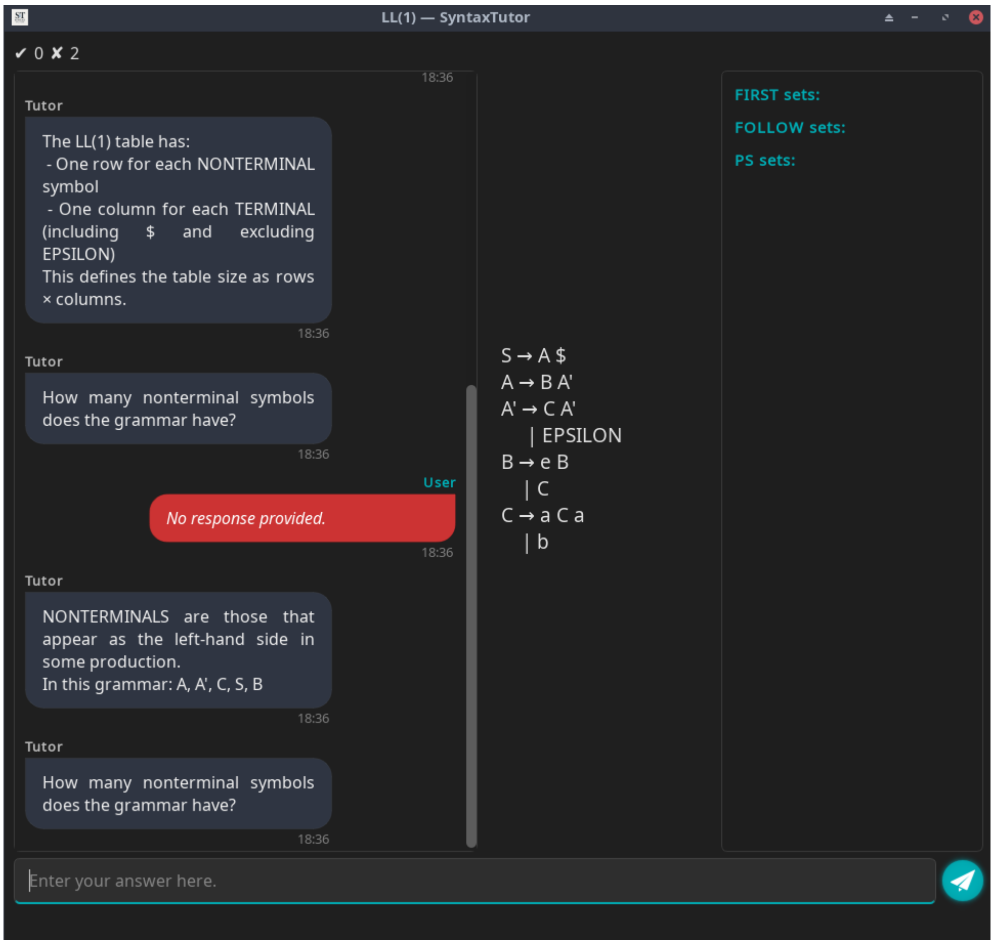


Figure 1.1 Main window

Home screen with gamification, levels, and language options.

1.3.2 LL(1) Learning Mode



18:36

Tutor

The LL(1) table has:
- One row for each NONTERMINAL symbol
- One column for each TERMINAL (including $ and excluding EPSILON)
This defines the table size as rows × columns.

18:36

Tutor

How many nonterminal symbols does the grammar have?

18:36

User

No response provided.

18:36

Tutor

NONTERMINALS are those that appear as the left-hand side in some production.
In this grammar: A, A', C, S, B

18:36

Tutor

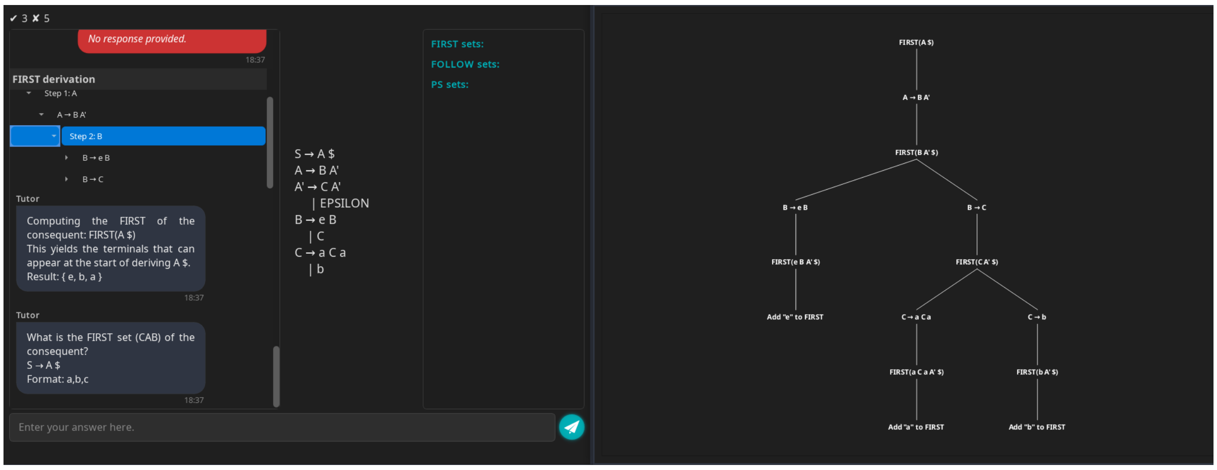
How many nonterminal symbols does the grammar have?

18:36

```
S → A $
A → B A'
A' → C A'
    | EPSILON
B → e B
   | C
C → a C a
   | b
```

Figure 1.2 LL(1) dialog view

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



No response provided.

18:37

Tutor

Computing the FIRST of the consequent: FIRST(A $)
This yields the terminals that can appear at the start of deriving A $.
Result: { e, b, a }

18:37

Tutor

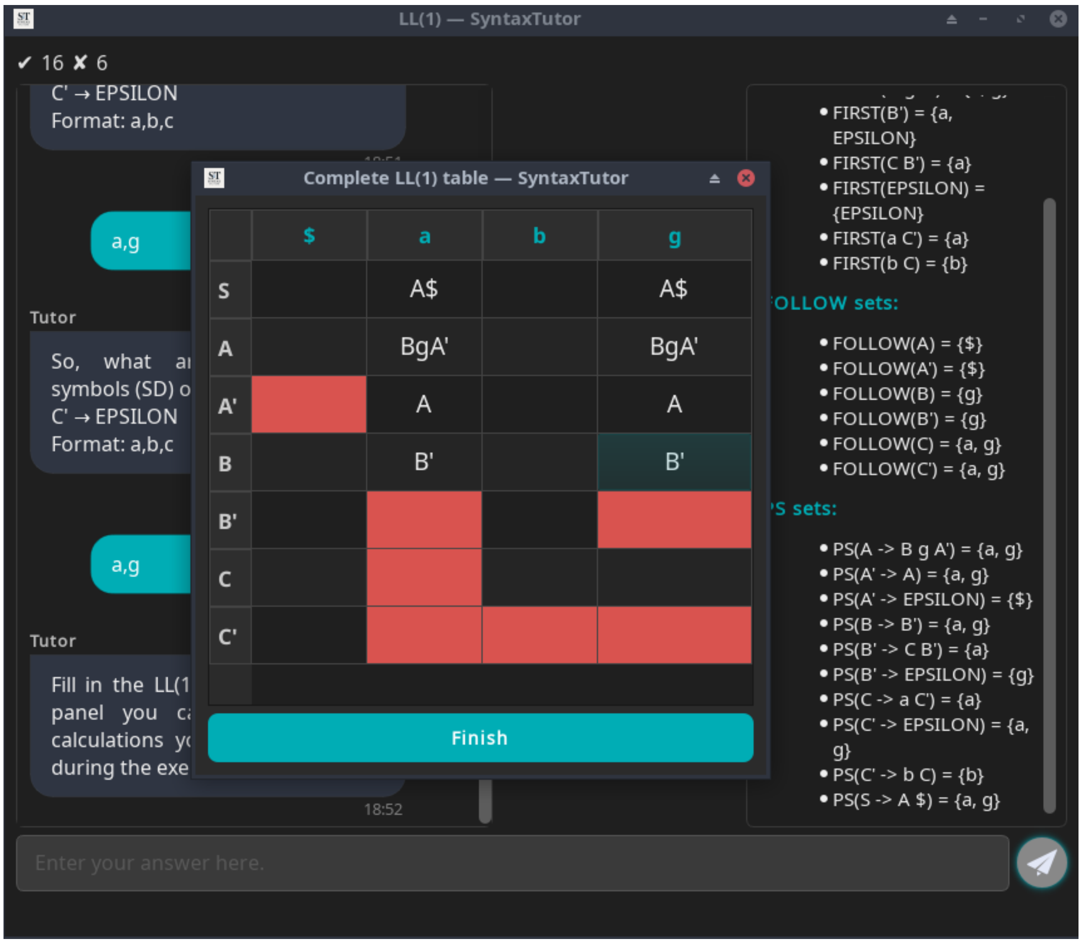
What is the FIRST set (CAB) of the consequent?
S → A $
Format: a,b,c

18:37

```
S → A $
A → B A'
A' → C A'
    | EPSILON
B → e B
   | C
C → a C a
   | b
```

Figure 1.3 LL(1) derivation tree

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



C' → EPSILON
Format: a,b,c

a,g

Tutor
So, what a
symbols (SD) o
C' → EPSILON
Format: a,b,c

a,g

Tutor
Fill in the LL(1
panel you c
calculations y
during the exe

18:52

	$	a	b	g
S		A$		A$
A		BgA'		BgA'
A'	🟥	A		A
B		B'		B'
B'		🟥		🟥
C		🟥		
C'		🟥	🟥	🟥

- FIRST(B') = {a, EPSILON}
- FIRST(C B') = {a}
- FIRST(EPSILON) = {EPSILON}
- FIRST(a C') = {a}
- FIRST(b C) = {b}

OLLOW sets:

- FOLLOW(A) = {$}
- FOLLOW(A') = {$}
- FOLLOW(B) = {g}
- FOLLOW(B') = {g}
- FOLLOW(C) = {a, g}
- FOLLOW(C') = {a, g}

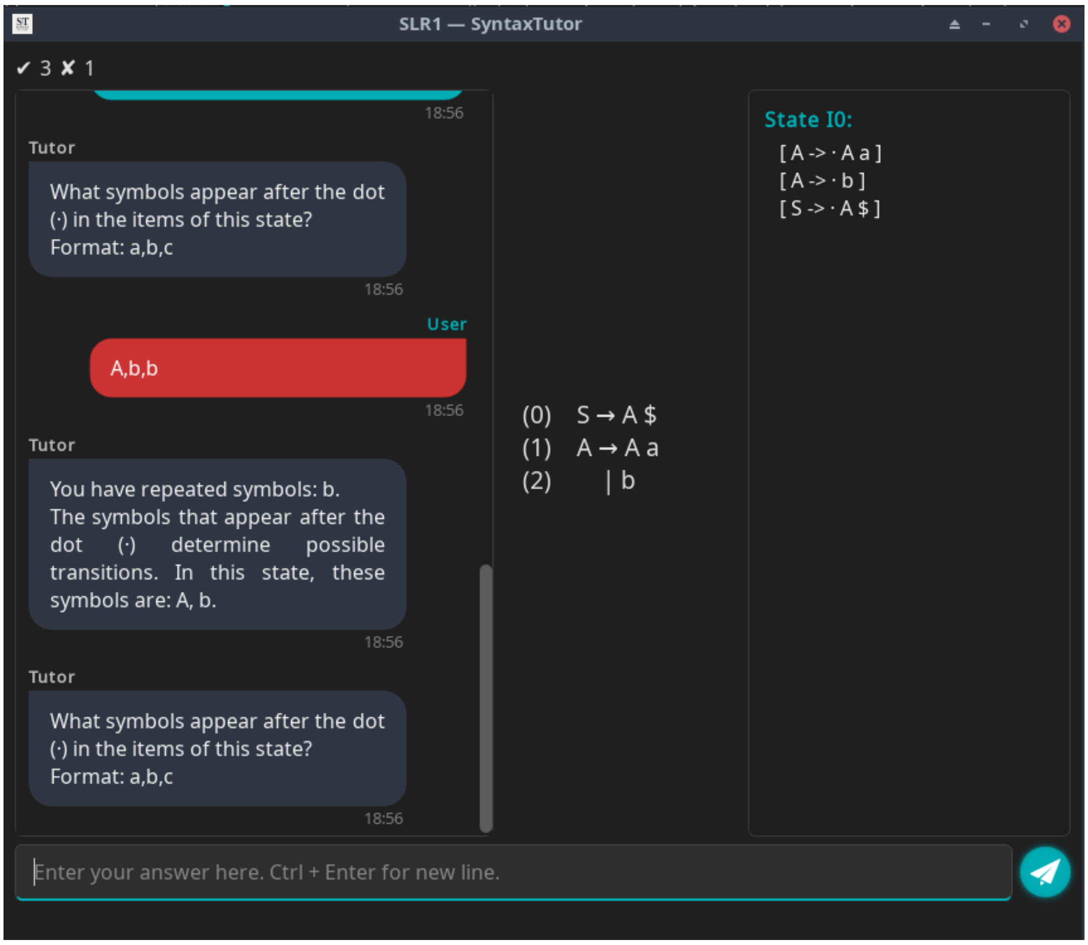
S sets:

- PS(A -> B g A') = {a, g}
- PS(A' -> A) = {a, g}
- PS(A' -> EPSILON) = {$}
- PS(B -> B') = {a, g}
- PS(B' -> C B') = {a}
- PS(B' -> EPSILON) = {g}
- PS(C -> a C') = {a}
- PS(C' -> EPSILON) = {a, g}
- PS(C' -> b C) = {b}
- PS(S -> A $) = {a, g}

Figure 1.4 LL(1) table task

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

1.3.3 SLR(1) Learning Mode



18:56

Tutor

What symbols appear after the dot (·) in the items of this state?
Format: a,b,c

18:56

User

A,b,b

18:56

Tutor

You have repeated symbols: b.
The symbols that appear after the dot (·) determine possible transitions. In this state, these symbols are: A, b.

18:56

Tutor

What symbols appear after the dot (·) in the items of this state?
Format: a,b,c

18:56

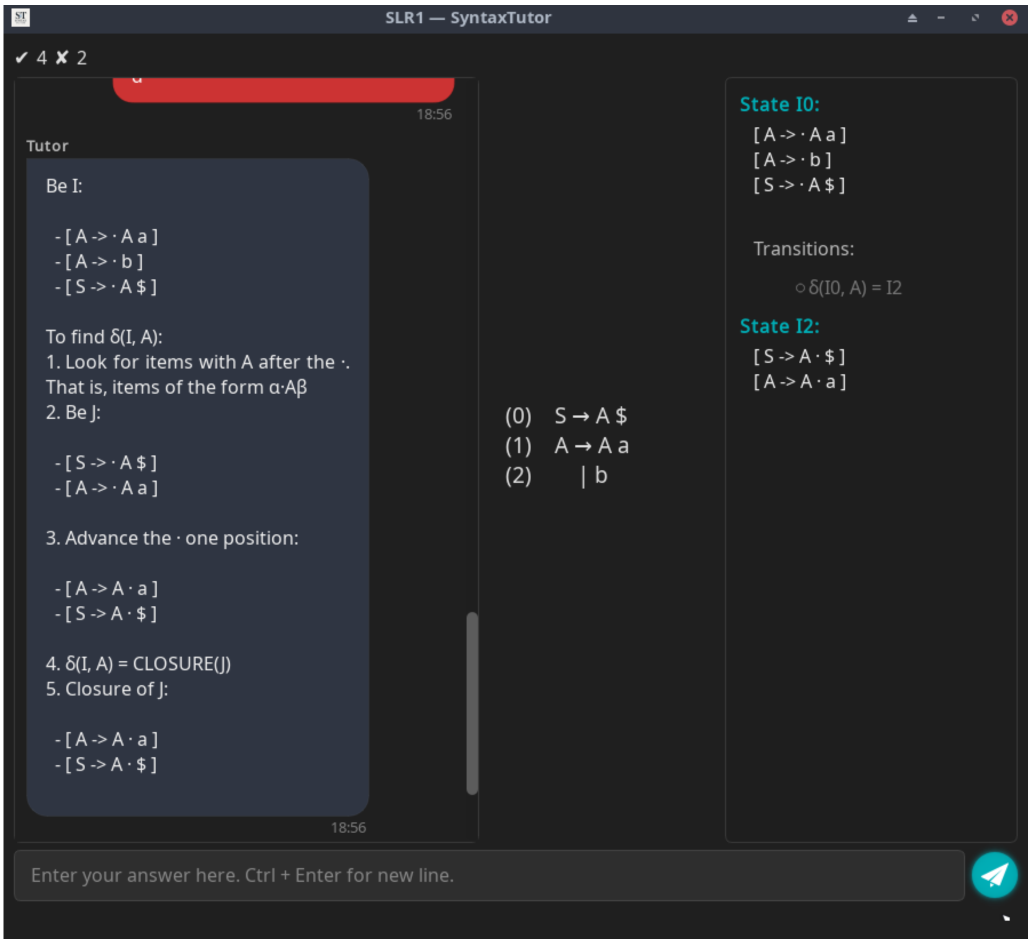
(0) S → A $
(1) A → A a
(2) | b

State I0:
[A -> · A a]
[A -> · b]
[S -> · A $]

Enter your answer here. Ctrl + Enter for new line.

Figure 1.5 SLR(1) item view

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



Be I:

- [A -> · A a]
- [A -> · b]
- [S -> · A $]

To find δ(I, A):
1. Look for items with A after the ·.
That is, items of the form α·Aβ
2. Be J:

- [S -> · A $]
- [A -> · A a]

3. Advance the · one position:

- [A -> A · a]
- [S -> A · $]

4. δ(I, A) = CLOSURE(J)
5. Closure of J:

- [A -> A · a]
- [S -> A · $]

(0) S → A $
(1) A → A a
(2) | b

State I0:
[A -> · A a]
[A -> · b]
[S -> · A $]

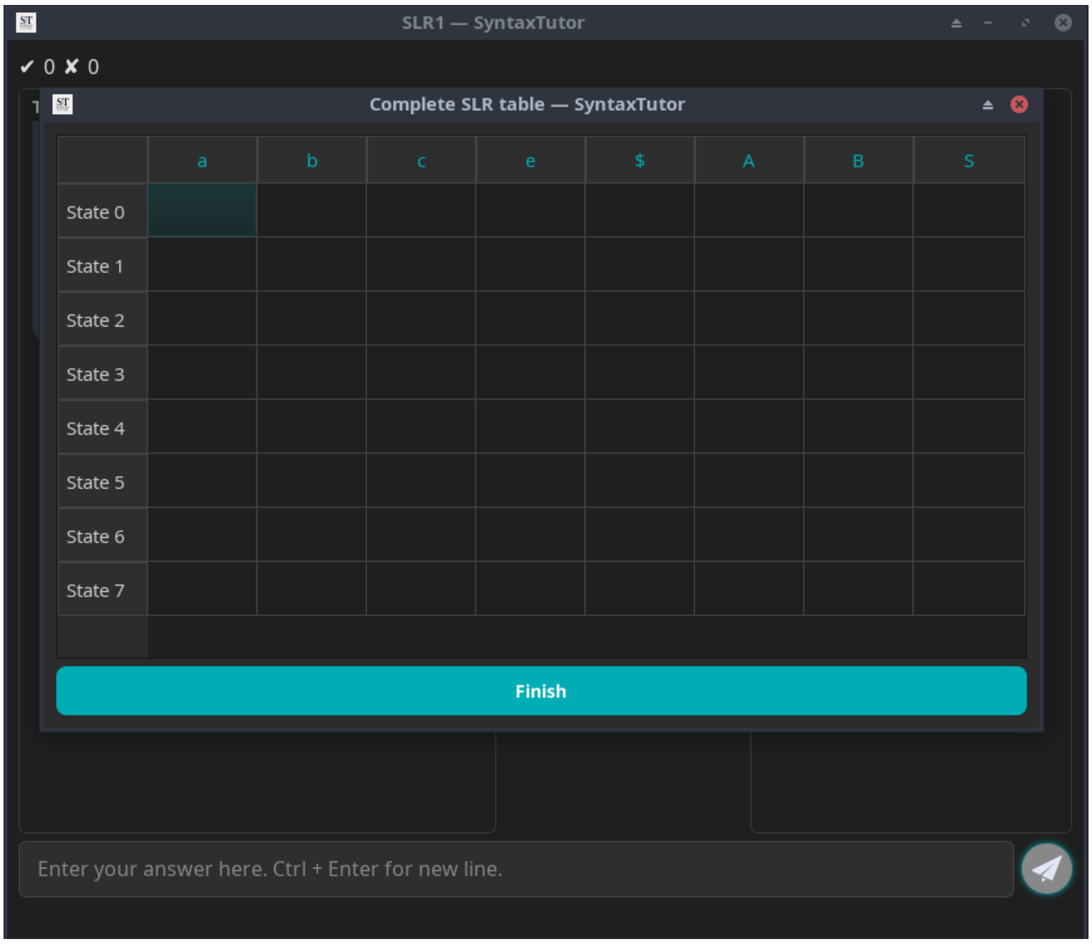
Transitions:

○ δ(I0, A) = I2

State I2:
[S -> A · $]
[A -> A · a]

Figure 1.6 SLR(1) automaton construction

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



SLR1 — SyntaxTutor

✓ 0 ✗ 0

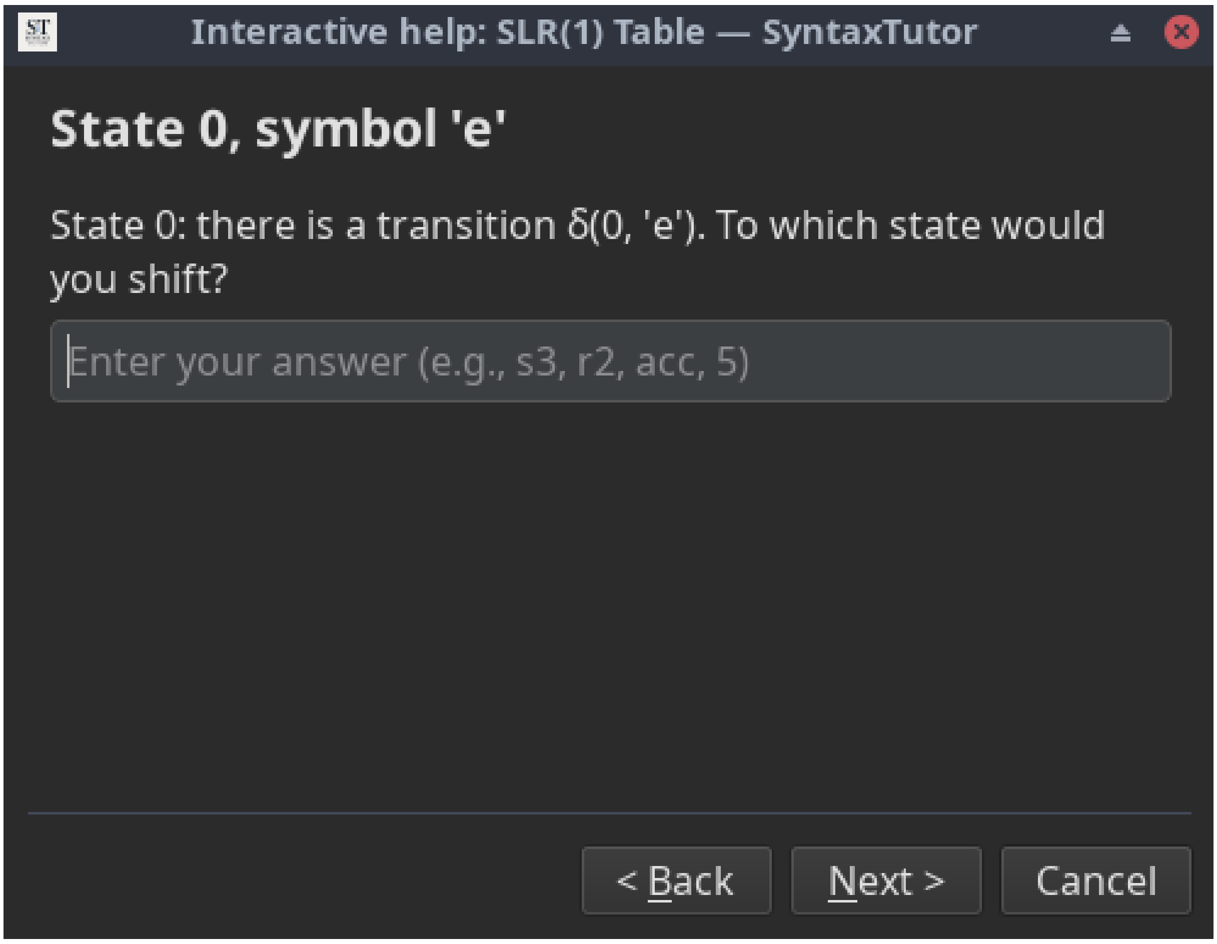
Complete SLR table — SyntaxTutor

	a	b	c	e	$	A	B	S
State 0								
State 1								
State 2								
State 3								
State 4								
State 5								
State 6								
State 7								

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


Interactive help: SLR(1) Table — SyntaxTutor

State 0, symbol 'e'

State 0: there is a transition δ(0, 'e'). To which state would you shift?

Enter your answer (e.g., s3, r2, acc, 5)

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
```

1.6.1 Documentation

Full documentation for the source code is available via Doxygen:

- Online HTML Documentation: https://jose-rzm.github.io/SyntaxTutor/
- PDF Reference Manual: refman.pdf (in the docs/latex/ folder)

The documentation includes:

- Detailed class and function reference
- Graphs of dependencies and inheritance
- Descriptions of parsing algorithms and internal logic

To regenerate it locally, install Doxygen and run:

```
doxygen
```

This will update the contents of the docs/ folder with both HTML and LaTeX output.

Chapter 2

Changelog

All notable changes to this project will be documented in this file.

2.1 [1.0.1] - 2025-06-17

2.1.1 Added

- Added Doxyfile for automatic documentation generation with Doxygen.
- Completed missing translations for multilingual support (English/Spanish).

2.1.2 Fixed

- Corrected a typo in the SLR(1) Quick Reference view.
- EPSILON is no longer shown when exporting LL(1) parse tables to PDF.
- Improved feedback message for the FA question in the SLR module.

2.1.3 Quality

- All changes successfully passed CI (GitHub Actions).
- Test suite: 158 tests passed (100% success rate).
- Maintained high test coverage across modules (most above 90%).

2.2 [1.0.0] - 2025-06-15

2.2.1 Initial Release

- First public version of SyntaxTutor.
- Includes LL(1) and SLR(1) modules with guided exercises.
- Features interactive tutoring, automatic grammar generation, feedback system, and performance tracking.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

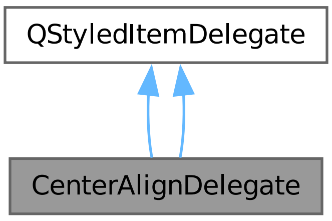
Here is a list of all files with brief descriptions:

Chapter 6

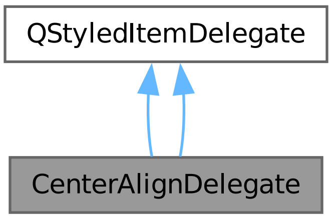
Class Documentation

6.1 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

6.1.1 Member Function Documentation

6.1.1.1 initStyleOption() [1/2]

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

6.1.1.2 initStyleOption() [2/2]

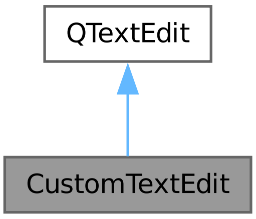
void CenterAlignDelegate::initStyleOption (
    QStyleOptionViewItem ∗ opt,
    const QModelIndex & idx) const [inline], [override]

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

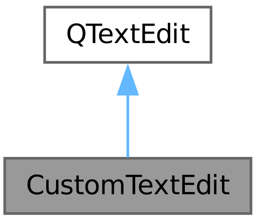
- lltabledialog.cpp
- slrtabledialog.cpp

6.2 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

6.2.1 Constructor & Destructor Documentation

6.2.1.1 CustomTextEdit()

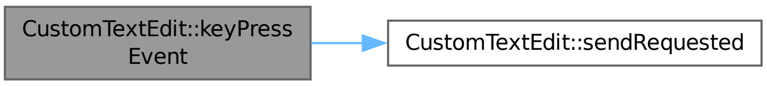
CustomTextEdit::CustomTextEdit (
QWidget ∗ parent = nullptr) [explicit]

6.2.2 Member Function Documentation

6.2.2.1 keyPressEvent()

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

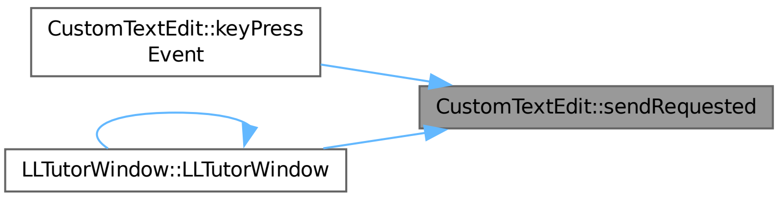
Here is the call graph for this function:



6.2.2.2 sendRequested

void CustomTextEdit::sendRequested () [signal]

Here is the caller graph for this function:


CustomTextEdit::keyPress Event
CustomTextEdit::sendRequested
LLTutorWindow::LLTutorWindow

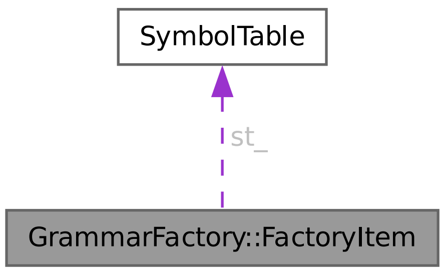
The documentation for this class was generated from the following files:

- customtextedit.h
- customtextedit.cpp

6.3 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.

6.3.1 Detailed Description

Represents an individual grammar item with its associated symbol table.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 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.

6.3.3 Member Data Documentation

6.3.3.1 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.

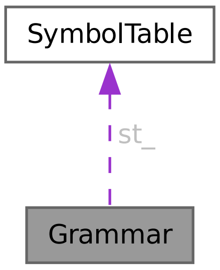
6.3.3.2 st_

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

6.4 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.

6.4.1 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.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 Grammar() [1/2]

Grammar::Grammar () [default]

6.4.2.2 Grammar() [2/2]

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

6.4.3 Member Function Documentation

6.4.3.1 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 $\rightarrow$ 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.

6.4.3.2 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.

6.4.3.3 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.

6.4.3.4 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.

6.4.3.5 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.

6.4.3.6 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.

6.4.4 Member Data Documentation

6.4.4.1 axiom_

std::string Grammar::axiom_

The axiom or entry point of the grammar.

6.4.4.2 g_

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

Stores the grammar rules with each antecedent mapped to a list of productions.

6.4.4.3 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

6.5 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:
- 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.
- void LeftFactorize (Grammar &grammar)

  Performs left factorization. A grammar can be left factorized if it has productions with the same prefix for one non-terminal. For example:
- 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.

6.5.1 Detailed Description

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

6.5.2 Member Function Documentation

6.5.2.1 AdjustTerminals()

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

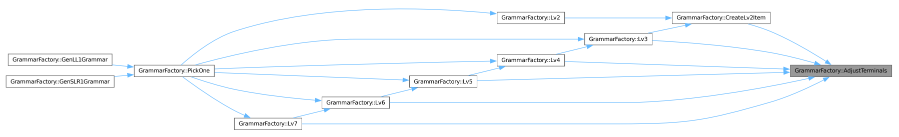
Adjusts the terminal symbols between two grammar items.

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:



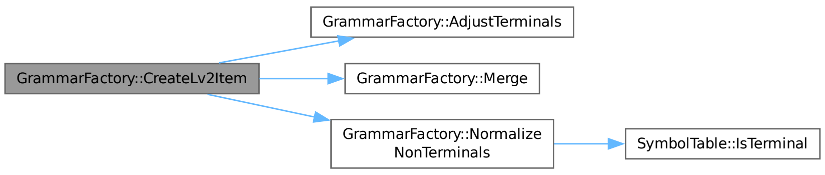
6.5.2.2 CreateLv2Item()

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.

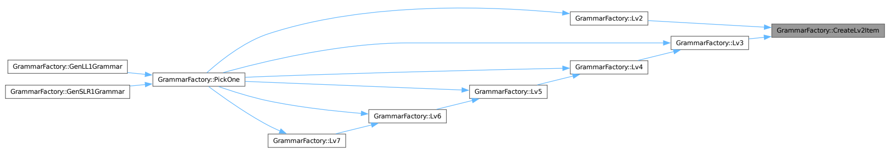
Returns

A FactoryItem representing a Level 2 grammar.

Here is the call graph for this function:


GrammarFactory::CreateLv2Item
GrammarFactory::AdjustTerminals
GrammarFactory::Merge
GrammarFactory::Normalize NonTerminals
SymbolTable::IsTerminal

Here is the caller graph for this function:



6.5.2.3 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.

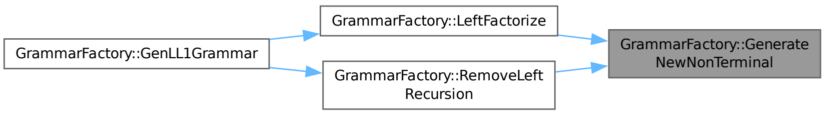
Parameters

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:


GrammarFactory::LeftFactorize
GrammarFactory::GenLL1Grammar
GrammarFactory::Generate NewNonTerminal
GrammarFactory::RemoveLeft Recursion

6.5.2.4 GenLL1Grammar()

Grammar GrammarFactory::GenLL1Grammar (
int level)

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

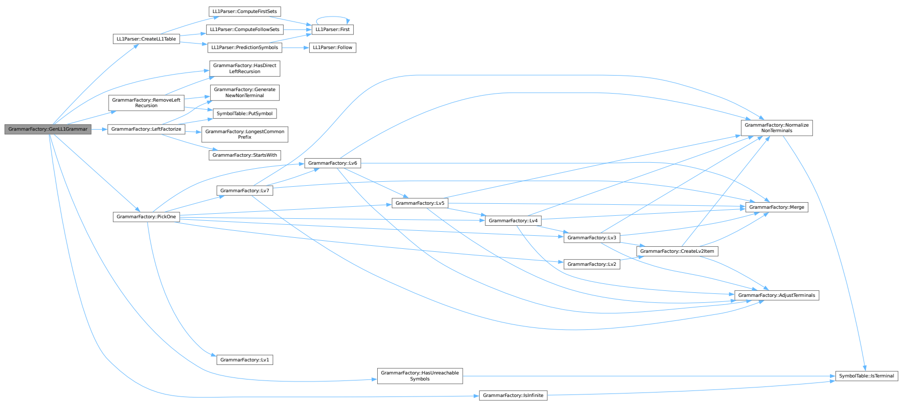
Parameters

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

Returns

A random LL(1) grammar.

Here is the call graph for this function:


LL1Parser::ComputeFirstSets
LL1Parser::ComputeFollowSets
LL1Parser::First
LL1Parser::CreateLL1Table
LL1Parser::PredictionSymbols
LL1Parser::Follow
GrammarFactory::HasDirect LeftRecursion
GrammarFactory::Generate NewNonTerminal
GrammarFactory::RemoveLeft Recursion
SymbolTable::PutSymbol
GrammarFactory::GenLL1Grammar
GrammarFactory::LeftFactorize
GrammarFactory::LongestCommon Prefix
GrammarFactory::Normalize NonTerminals
GrammarFactory::StartsWith
GrammarFactory::Lv6
GrammarFactory::Lv7
GrammarFactory::Lv5
GrammarFactory::Merge
GrammarFactory::PickOne
GrammarFactory::Lv4
GrammarFactory::Lv3
GrammarFactory::CreateLv2Item
GrammarFactory::Lv2
GrammarFactory::AdjustTerminals
GrammarFactory::Lv1
GrammarFactory::HasUnreachable Symbols
SymbolTable::IsTerminal
GrammarFactory::IsInfinite

6.5.2.5 GenSLR1Grammar()

Grammar GrammarFactory::GenSLR1Grammar (
int level)

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

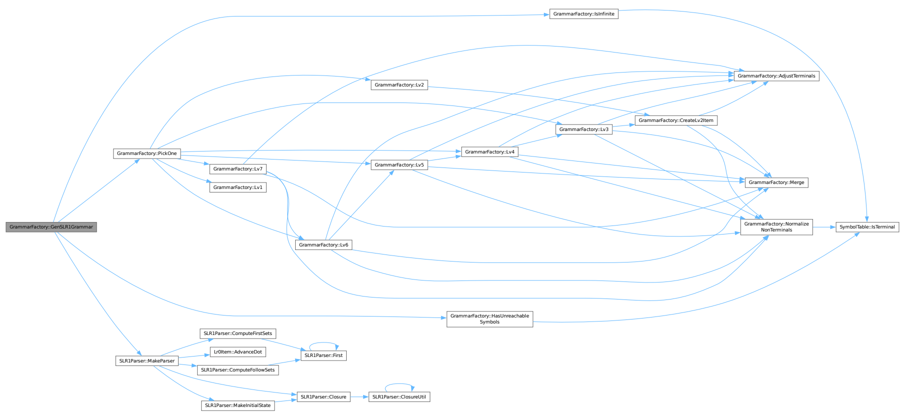
Parameters

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

Returns

A random SLR(1) grammar.

Here is the call graph for this function:



6.5.2.6 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.

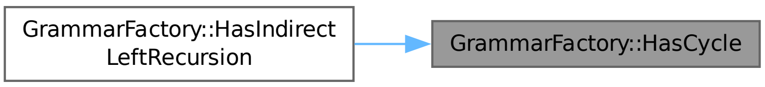
Parameters

graph	The directed graph.

Returns

true if grammar has cycle.

Here is the caller graph for this function:



6.5.2.7 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).

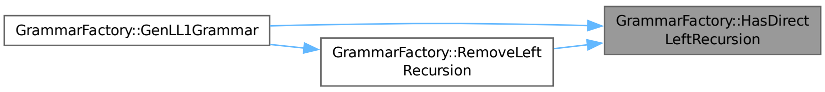
Parameters

grammar	The grammar to check.

Returns

true if there is direct left recursion, false otherwise.

Here is the caller graph for this function:


GrammarFactory::GenLL1Grammar
GrammarFactory::RemoveLeft Recursion
GrammarFactory::HasDirect LeftRecursion

6.5.2.8 HasIndirectLeftRecursion()

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

Checks if a grammar contains indirect left recursion.

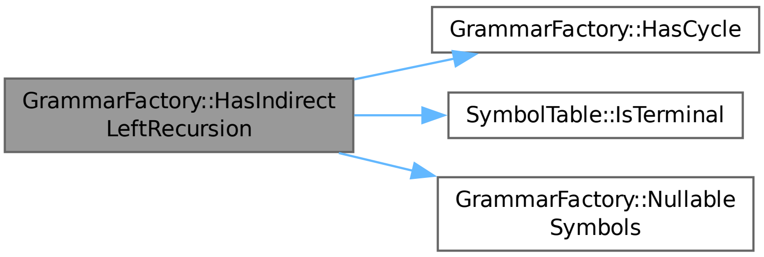
Parameters

grammar	The grammar to check.

Returns

true if there is direct left recursion, false otherwise.

Here is the call graph for this function:


GrammarFactory::HasIndirect LeftRecursion
GrammarFactory::HasCycle
SymbolTable::IsTerminal
GrammarFactory::Nullable Symbols

6.5.2.9 HasUnreachableSymbols()

bool GrammarFactory::HasUnreachableSymbols (
Grammar & grammar) const

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

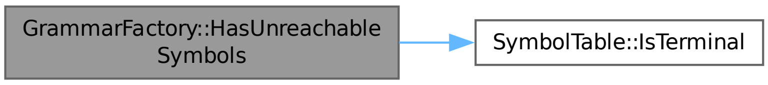
Parameters

grammar	The grammar to check.

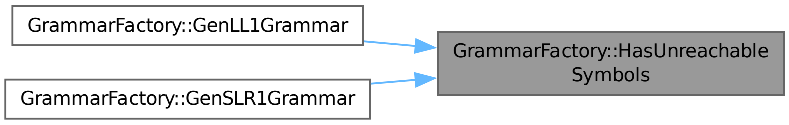
Returns

true if there are unreachable symbols, false otherwise.

Here is the call graph for this function:



Here is the caller graph for this function:


GrammarFactory::GenLL1Grammar
GrammarFactory::GenSLR1Grammar
GrammarFactory::HasUnreachable Symbols

6.5.2.10 Init()

void GrammarFactory::Init ()

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

6.5.2.11 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.

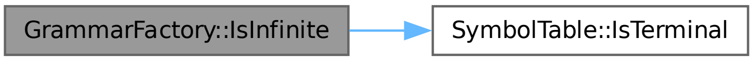
Parameters

grammar	The grammar to check.

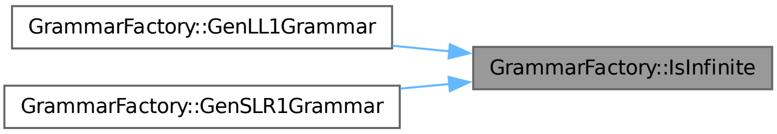
Returns

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:


GrammarFactory::GenLL1Grammar
GrammarFactory::GenSLR1Grammar
GrammarFactory::IsInfinite

6.5.2.12 LeftFactorize()

```
void GrammarFactory::LeftFactorize (
                Grammar & grammar)
```

Performs left factorization. A grammar can be left factorized if it has 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 one production. So:

```
A -> a x | a y
```

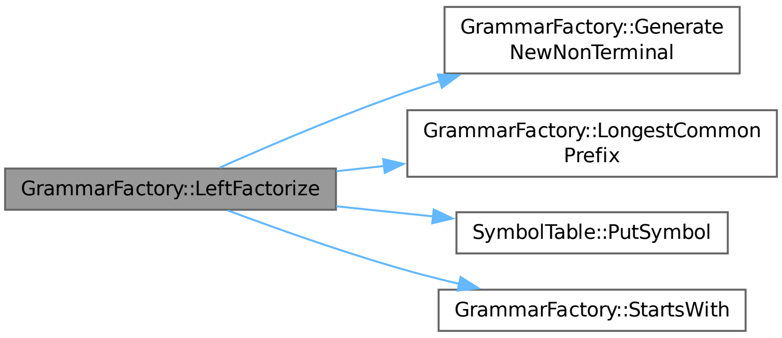
would become:

```
A  -> a A'
A' -> x | y
```

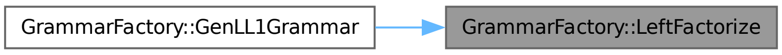
Parameters

grammar	The grammar to be left factorized.

Here is the call graph for this function:


GrammarFactory::Generate NewNonTerminal
GrammarFactory::LongestCommon Prefix
GrammarFactory::LeftFactorize
SymbolTable::PutSymbol
GrammarFactory::StartsWith

Here is the caller graph for this function:



6.5.2.13 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.

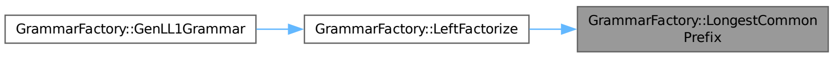
Parameters

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:


GrammarFactory::GenLL1Grammar
GrammarFactory::LeftFactorize
GrammarFactory::LongestCommon Prefix

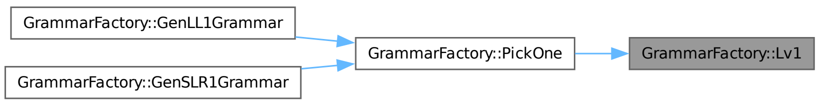
6.5.2.14 Lv1()

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

Returns

A Level 1 grammar.

Here is the caller graph for this function:



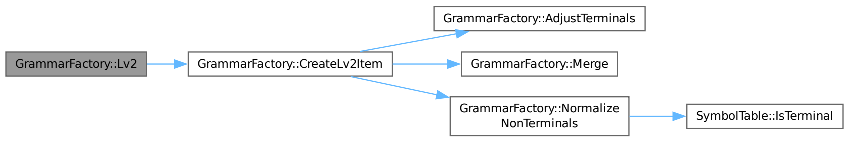
6.5.2.15 Lv2()

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

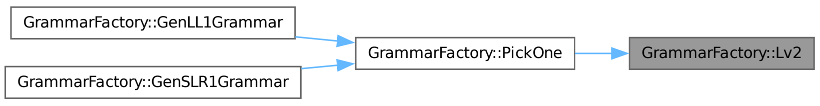
Returns

A Level 2 grammar.

Here is the call graph for this function:


GrammarFactory::Lv2
GrammarFactory::CreateLv2Item
GrammarFactory::AdjustTerminals
GrammarFactory::Merge
GrammarFactory::Normalize NonTerminals
SymbolTable::IsTerminal

Here is the caller graph for this function:



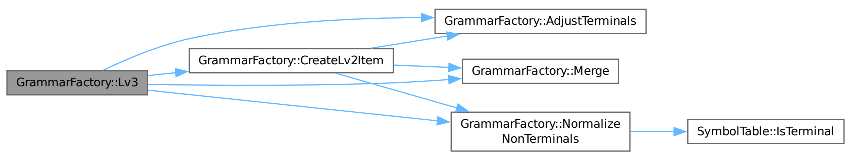
6.5.2.16 Lv3()

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

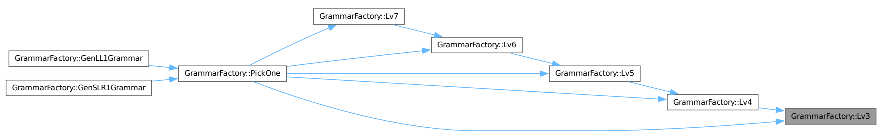
Returns

A Level 3 grammar.

Here is the call graph for this function:


GrammarFactory::Lv3
GrammarFactory::AdjustTerminals
GrammarFactory::CreateLv2Item
GrammarFactory::Merge
GrammarFactory::Normalize NonTerminals
SymbolTable::IsTerminal

Here is the caller graph for this function:



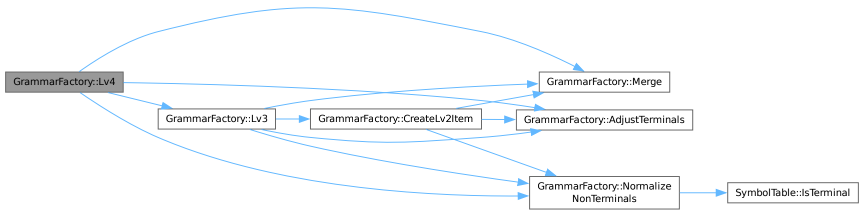
6.5.2.17 Lv4()

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.

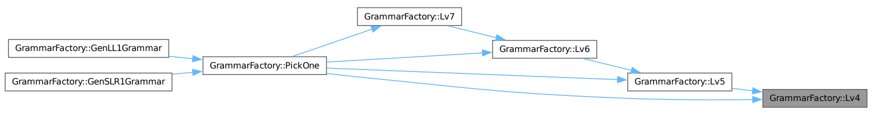
Returns

A Level 4 grammar.

Here is the call graph for this function:


GrammarFactory::Lv4
GrammarFactory::Merge
GrammarFactory::Lv3
GrammarFactory::CreateLv2Item
GrammarFactory::AdjustTerminals
GrammarFactory::Normalize NonTerminals
SymbolTable::IsTerminal

Here is the caller graph for this function:



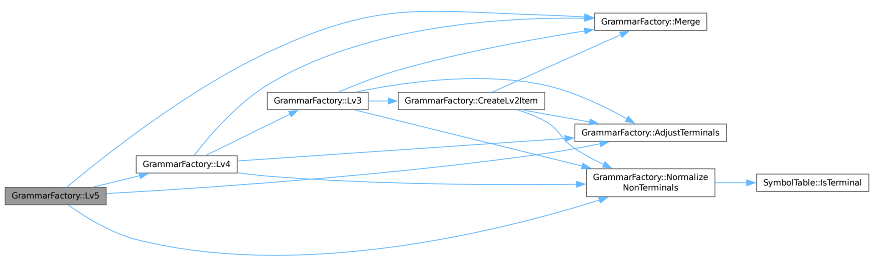
6.5.2.18 Lv5()

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.

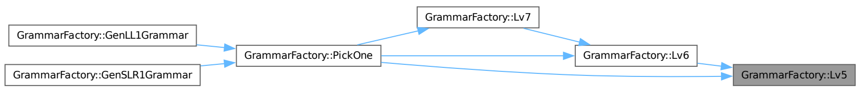
Returns

A Level 5 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



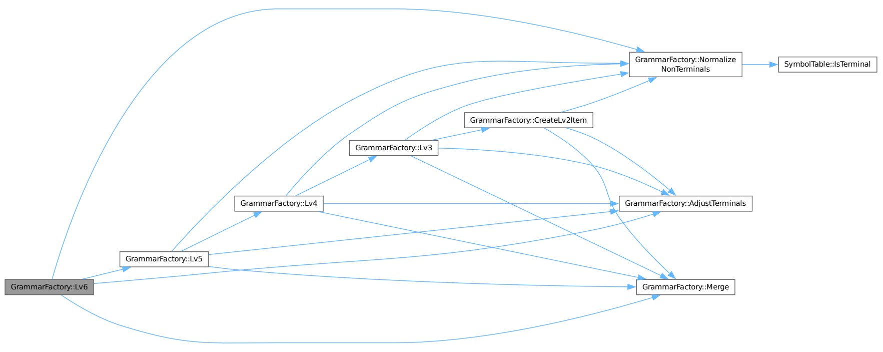
6.5.2.19 Lv6()

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.

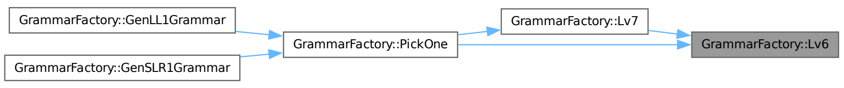
Returns

A Level 6 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.20 Lv7()

Grammar GrammarFactory::Lv7 ()

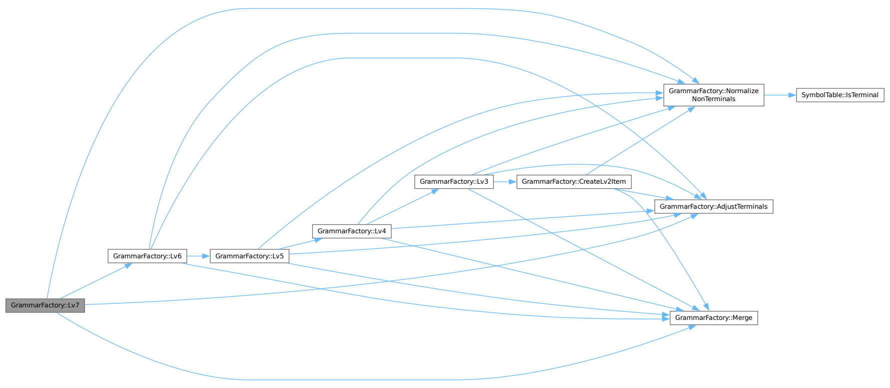
Generates a Level 7 grammar by combining Level 6 and Level 1 items.

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.

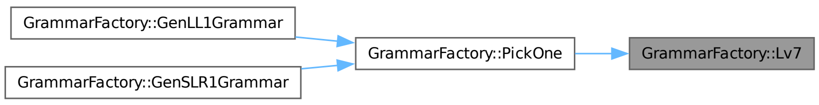
Returns

A Level 7 grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.21 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.

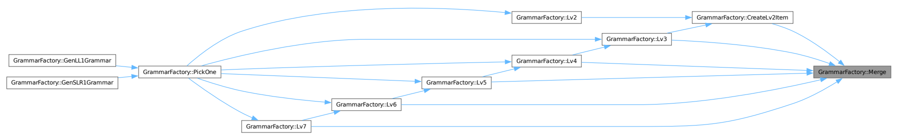
Parameters

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:



6.5.2.22 NormalizeNonTerminals()

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

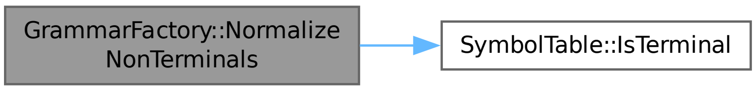
Replaces all non-terminal symbols in a grammar item with a single target non-terminal.

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.

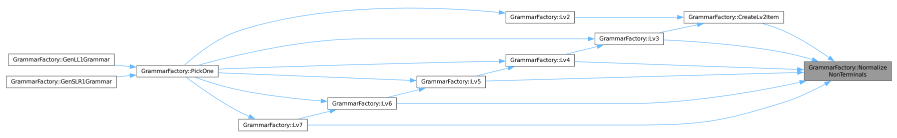
Parameters

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:



6.5.2.23 NullableSymbols()

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

Find nullable symbols in a grammar.

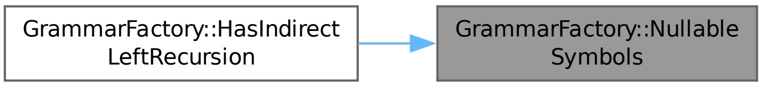
Parameters

grammar	The grammar to check.

Returns

set of nullable symbols.

Here is the caller graph for this function:


GrammarFactory::HasIndirect LeftRecursion
GrammarFactory::Nullable Symbols

6.5.2.24 PickOne()

Grammar GrammarFactory::PickOne (
    int level)

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

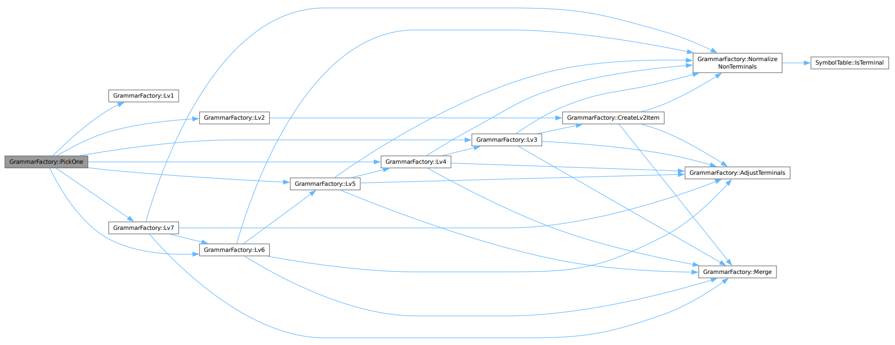
Parameters

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

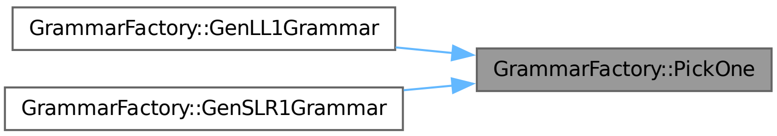
Returns

A randomly picked grammar.

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.25 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

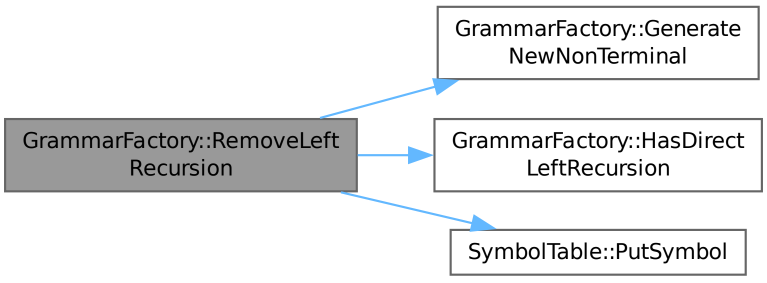
the result would be:

A -> b A'
A' -> a A' | epsilon

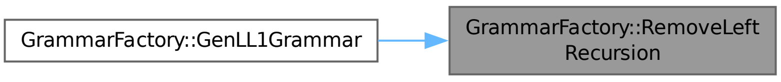
Parameters

grammar	The grammar to remove left recursion.

Here is the call graph for this function:


GrammarFactory::RemoveLeft Recursion
GrammarFactory::Generate NewNonTerminal
GrammarFactory::HasDirect LeftRecursion
SymbolTable::PutSymbol

Here is the caller graph for this function:


GrammarFactory::GenLL1Grammar
GrammarFactory::RemoveLeft Recursion

6.5.2.26 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.

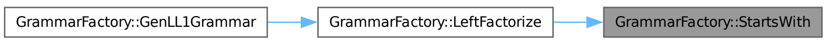
Parameters

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:


GrammarFactory::GenLL1Grammar
GrammarFactory::LeftFactorize
GrammarFactory::StartsWith

6.5.3 Member Data Documentation

6.5.3.1 items

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

6.5.3.2 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.

6.5.3.3 terminal_alphabet_

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

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

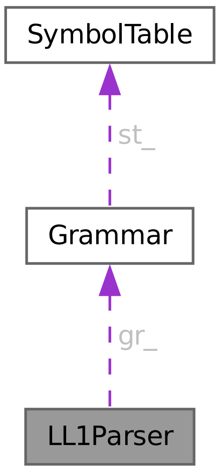
A vector of terminal symbols (alphabet) used in the grammar.
The documentation for this struct was generated from the following files:

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

6.6 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. 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:
- 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.

6.6.1 Constructor & Destructor Documentation

6.6.1.1 LL1Parser() [1/2]

LL1Parser::LL1Parser () [default]

6.6.1.2 LL1Parser() [2/2]

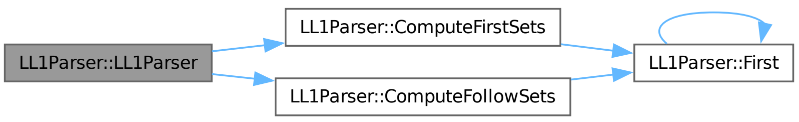
LL1Parser::LL1Parser (
Grammar gr) [explicit]

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:



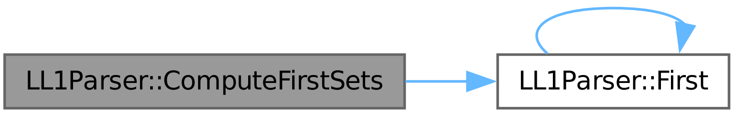
6.6.2 Member Function Documentation

6.6.2.1 ComputeFirstSets()

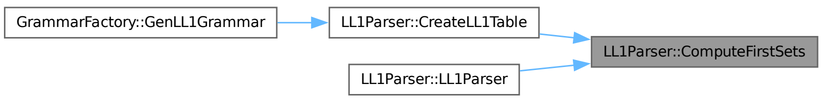
void LL1Parser::ComputeFirstSets ()

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:


GrammarFactory::GenLL1Grammar
LL1Parser::CreateLL1Table
LL1Parser::LL1Parser
LL1Parser::ComputeFirstSets

6.6.2.2 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( ) { } 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.

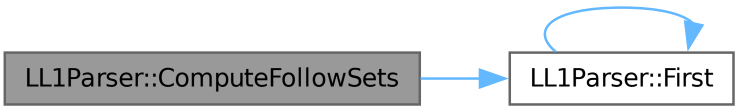
   Note

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

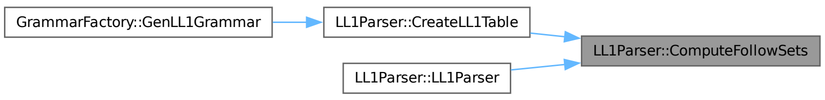
   See also

   > First
   > follow_sets_

Here is the call graph for this function:



Here is the caller graph for this function:


GrammarFactory::GenLL1Grammar
LL1Parser::CreateLL1Table
LL1Parser::LL1Parser
LL1Parser::ComputeFollowSets

6.6.2.3 CreateLL1Table()

bool LL1Parser::CreateLL1Table ()

Creates the LL(1) parsing table for the grammar.

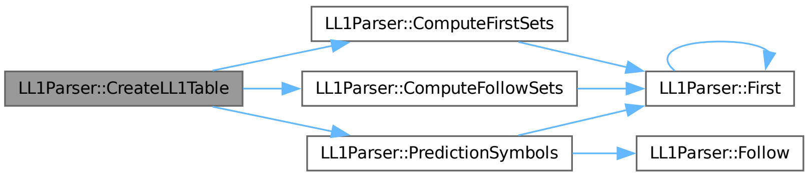
This function constructs the LL(1) parsing table by iterating over each production in the grammar and determining the appropriate cells for each non-terminal and director symbol (prediction symbol) combination. If the grammar is LL(1) compatible, each cell will contain at most one production, indicating 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 prediction symbols using the Prediction↩ Symbols function.
- It then fills the parsing table at the cell corresponding to the non-terminal A and each prediction symbol in the set.
- If a cell already contains a production, this indicates a conflict, meaning the grammar is not LL(1).

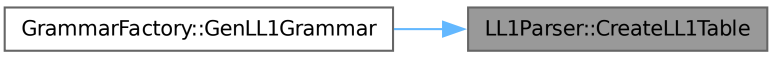
Returns

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:


LL1Parser::CreateLL1Table
LL1Parser::ComputeFirstSets
LL1Parser::ComputeFollowSets
LL1Parser::PredictionSymbols
LL1Parser::First
LL1Parser::Follow

Here is the caller graph for this function:



6.6.2.4 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:

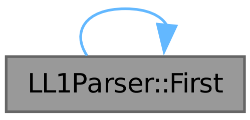
- If a terminal symbol is encountered, it is added directly to the FIRST set, as it is the starting symbol of some derivation.

- 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.

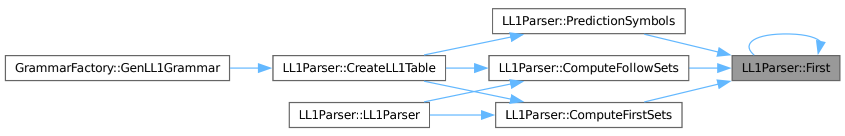
Parameters

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:


LL1Parser::PredictionSymbols
GrammarFactory::GenLL1Grammar
LL1Parser::CreateLL1Table
LL1Parser::ComputeFollowSets
LL1Parser::First
LL1Parser::LL1Parser
LL1Parser::ComputeFirstSets

6.6.2.5 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.

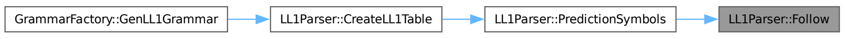
Parameters

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:


GrammarFactory::GenLL1Grammar
LL1Parser::CreateLL1Table
LL1Parser::PredictionSymbols
LL1Parser::Follow

6.6.2.6 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) { } FOLLOW(antecedent).

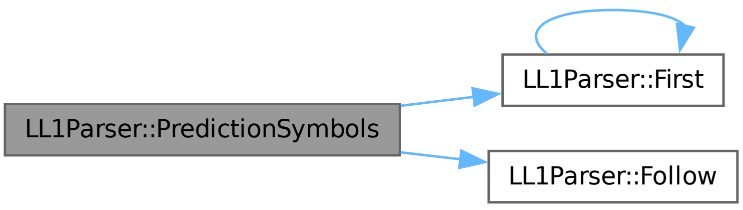
Parameters

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).

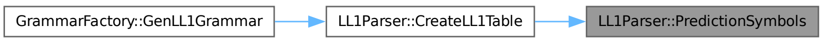
Returns

- 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:


GrammarFactory::GenLL1Grammar
LL1Parser::CreateLL1Table
LL1Parser::PredictionSymbols

6.6.3 Member Data Documentation

6.6.3.1 first_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > LL1Parser::first_sets_
FIRST sets for each non-terminal in the grammar.

6.6.3.2 follow_sets_

std::unordered_map<std::string, std::unordered_set<std::string> > LL1Parser::follow_sets_
FOLLOW sets for each non-terminal in the grammar.

6.6.3.3 gr_

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

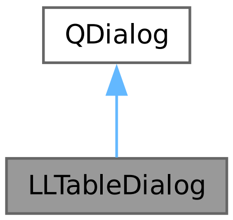
6.6.3.4 ll1_t_

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:

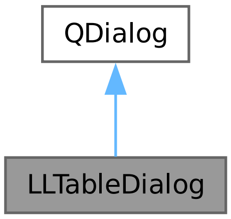
- backend/ll1_parser.hpp
- backend/ll1_parser.cpp

6.7 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.

6.7.1 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.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 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.

6.7.3 Member Function Documentation

6.7.3.1 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.

6.7.3.2 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.

6.7.3.3 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.

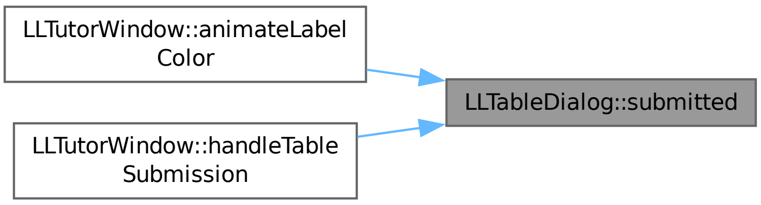
6.7.3.4 submitted

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:

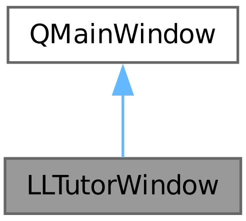

LLTutorWindow::animateLabel Color
LLTutorWindow::handleTable Submission
LLTableDialog::submitted

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

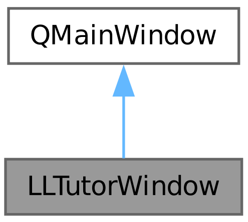
- lltabledialog.h
- lltabledialog.cpp

6.8 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

6.8.1 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.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 LLTutorWindow()

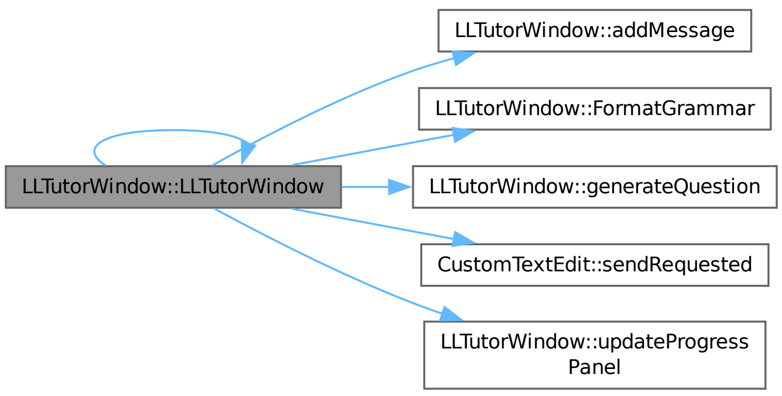
```
LLTutorWindow::LLTutorWindow (
                const Grammar & grammar,
                TutorialManager * tm = nullptr,
                QWidget * parent = nullptr)   [explicit]
```

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

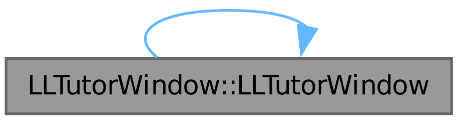
Parameters

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:


LLTutorWindow::LLTutorWindow
LLTutorWindow::addMessage
LLTutorWindow::FormatGrammar
LLTutorWindow::generateQuestion
CustomTextEdit::sendRequested
LLTutorWindow::updateProgressPanel

Here is the caller graph for this function:



6.8.2.2 ~LLTutorWindow()

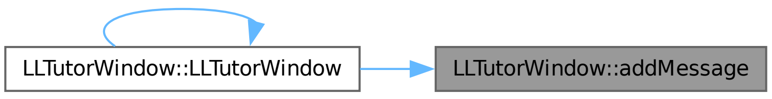
LLTutorWindow::~LLTutorWindow ()

6.8.3 Member Function Documentation

6.8.3.1 addMessage()

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

Here is the caller graph for this function:



6.8.3.2 addWidgetMessage()

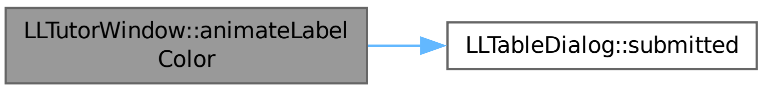
void LLTutorWindow::addWidgetMessage (
    QWidget * widget)

< Add text message to chat

6.8.3.3 animateLabelColor()

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

Here is the call graph for this function:



6.8.3.4 animateLabelPop()

void LLTutorWindow::animateLabelPop (
    QLabel ∗ label)

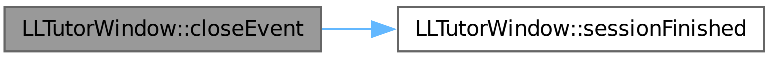
6.8.3.5 buildTreeNode()

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)

6.8.3.6 closeEvent()

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

Here is the call graph for this function:



6.8.3.7 computeSubtreeWidth()

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

6.8.3.8 drawTree()

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

6.8.3.9 eventFilter()

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

6.8.3.10 exportConversationToPdf()

void LLTutorWindow::exportConversationToPdf (
    const QString & filePath)

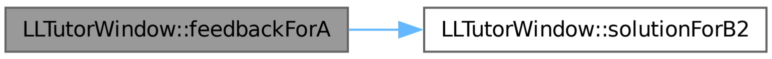
< Add widget (e.g., table, tree)

6.8.3.11 feedback()

QString LLTutorWindow::feedback ()

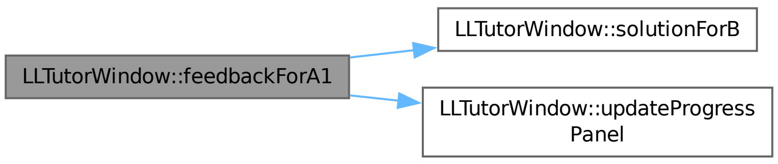
6.8.3.12 feedbackForA()

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



6.8.3.13 feedbackForA1()

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

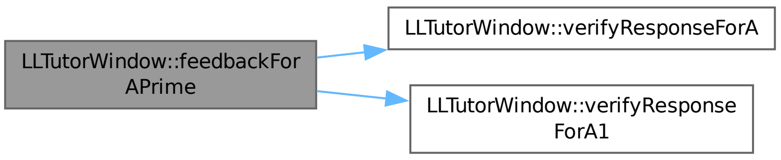

LLTutorWindow::feedbackForA1
LLTutorWindow::solutionForB
LLTutorWindow::updateProgress Panel

6.8.3.14 feedbackForA2()

QString LLTutorWindow::feedbackForA2 ()

6.8.3.15 feedbackForAPrime()

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


LLTutorWindow::feedbackFor APrime
LLTutorWindow::verifyResponseForA
LLTutorWindow::verifyResponse ForA1

6.8.3.16 feedbackForB()

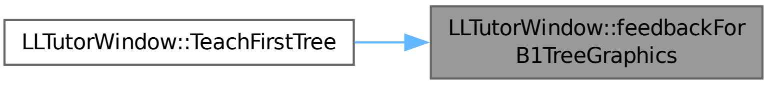
QString LLTutorWindow::feedbackForB ()

6.8.3.17 feedbackForB1()

QString LLTutorWindow::feedbackForB1 ()

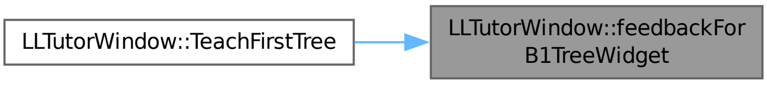
6.8.3.18 feedbackForB1TreeGraphics()

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


LLTutorWindow::TeachFirstTree
LLTutorWindow::feedbackFor B1TreeGraphics

6.8.3.19 feedbackForB1TreeWidget()

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


LLTutorWindow::TeachFirstTree
LLTutorWindow::feedbackFor B1TreeWidget

6.8.3.20 feedbackForB2()

QString LLTutorWindow::feedbackForB2 ()

6.8.3.21 feedbackForBPrime()

QString LLTutorWindow::feedbackForBPrime ()

6.8.3.22 feedbackForC()

QString LLTutorWindow::feedbackForC ()

6.8.3.23 feedbackForCPrime()

QString LLTutorWindow::feedbackForCPrime ()

6.8.3.24 FormatGrammar()

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

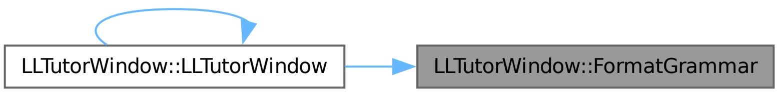
Parameters

grammar	The grammar to format.

Returns

A QString representation.

Here is the caller graph for this function:



6.8.3.25 generateQuestion()

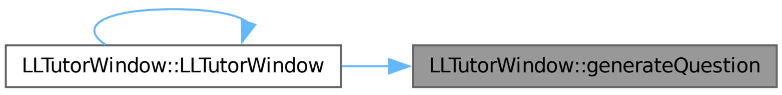
QString LLTutorWindow::generateQuestion ()

Generates a question for the current state of the tutor.

Returns

A formatted question string.

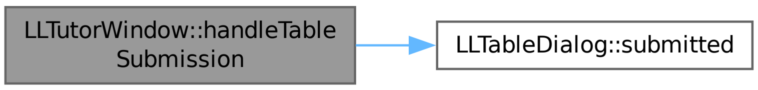
Here is the caller graph for this function:



6.8.3.26 handleTableSubmission()

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

Here is the call graph for this function:



6.8.3.27 markLastUserIncorrect()

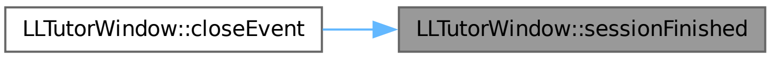
void LLTutorWindow::markLastUserIncorrect ()

Marks last message as incorrect.

6.8.3.28 sessionFinished

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

Here is the caller graph for this function:



6.8.3.29 showTable()

void LLTutorWindow::showTable ()

< Export chat to PDF
Display the full LL(1) table in C ex.

6.8.3.30 showTableForCPrime()

void LLTutorWindow::showTableForCPrime ()

Display the full LL(1) table in C' ex.

6.8.3.31 showTreeGraphics()

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

6.8.3.32 solution()

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

6.8.3.33 solutionForA()

QStringList LLTutorWindow::solutionForA ()

6.8.3.34 solutionForA1()

QString LLTutorWindow::solutionForA1 ()

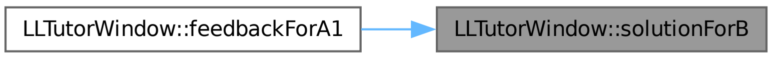
6.8.3.35 solutionForA2()

QString LLTutorWindow::solutionForA2 ()

6.8.3.36 solutionForB()

QSet< QString > LLTutorWindow::solutionForB ()

Here is the caller graph for this function:



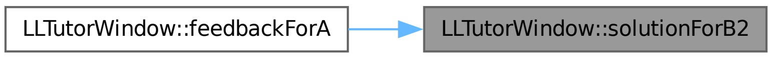
6.8.3.37 solutionForB1()

QSet< QString > LLTutorWindow::solutionForB1 ()

6.8.3.38 solutionForB2()

QSet< QString > LLTutorWindow::solutionForB2 ()

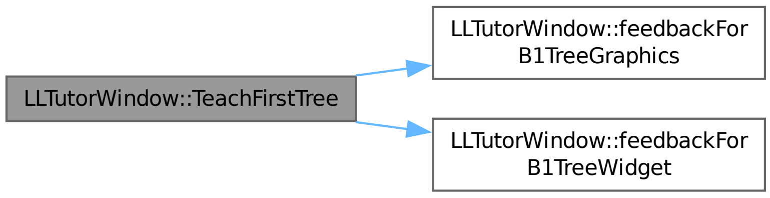
Here is the caller graph for this function:



6.8.3.39 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:


LLTutorWindow::TeachFirstTree
LLTutorWindow::feedbackFor B1TreeGraphics
LLTutorWindow::feedbackFor B1TreeWidget

6.8.3.40 TeachFollow()

QString LLTutorWindow::TeachFollow (
    const QString & nt)

6.8.3.41 TeachLL1Table()

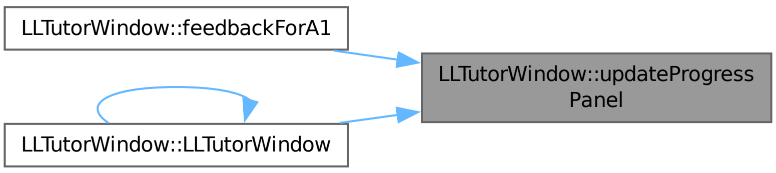
QString LLTutorWindow::TeachLL1Table ()

6.8.3.42 TeachPredictionSymbols()

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

6.8.3.43 updateProgressPanel()

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


LLTutorWindow::feedbackForA1
LLTutorWindow::LLTutorWindow
LLTutorWindow::updateProgress Panel

6.8.3.44 updateState()

void LLTutorWindow::updateState (
    bool isCorrect)
Updates the tutor state after verifying user response.

Parameters

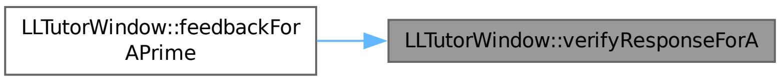
isCorrect	Whether the user answered correctly.

6.8.3.45 verifyResponse()

bool LLTutorWindow::verifyResponse (
    const QString & userResponse)

6.8.3.46 verifyResponseForA()

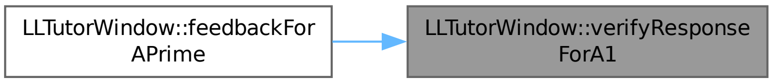
bool LLTutorWindow::verifyResponseForA (
    const QString & userResponse)
Here is the caller graph for this function:


LLTutorWindow::feedbackFor APrime
LLTutorWindow::verifyResponseForA

6.8.3.47 verifyResponseForA1()

bool LLTutorWindow::verifyResponseForA1 (
    const QString & userResponse)

Here is the caller graph for this function:


LLTutorWindow::feedbackFor APrime
LLTutorWindow::verifyResponse ForA1

6.8.3.48 verifyResponseForA2()

bool LLTutorWindow::verifyResponseForA2 (
    const QString & userResponse)

6.8.3.49 verifyResponseForB()

bool LLTutorWindow::verifyResponseForB (
    const QString & userResponse)

6.8.3.50 verifyResponseForB1()

bool LLTutorWindow::verifyResponseForB1 (
    const QString & userResponse)

6.8.3.51 verifyResponseForB2()

bool LLTutorWindow::verifyResponseForB2 (
    const QString & userResponse)

6.8.3.52 verifyResponseForC()

bool LLTutorWindow::verifyResponseForC ()

6.8.3.53 wrongAnimation()

void LLTutorWindow::wrongAnimation ()

Visual shake/flash for incorrect answer.

6.8.3.54 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

6.9 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.

6.9.1 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 ($).

6.9.2 Constructor & Destructor Documentation

6.9.2.1 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.

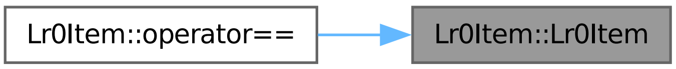
Parameters

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

Parameters

epsilon	The EPSILON symbol.
eol	The end-of-line symbol.

Here is the caller graph for this function:



6.9.2.2 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

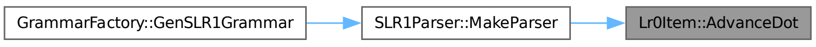
antecedent	The left-hand side non-terminal.
consequent	The right-hand side of the production.
dot	The position of the dot.
epsilon	The EPSILON symbol.
eol	The end-of-line symbol.

6.9.3 Member Function Documentation

6.9.3.1 AdvanceDot()

```
void Lr0Item::AdvanceDot ()
```

Advances the dot one position to the right.
Here is the caller graph for this function:



6.9.3.2 IsComplete()

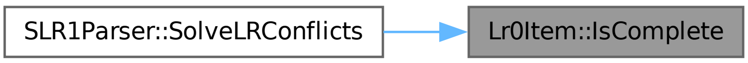
```
bool Lr0Item::IsComplete () const
```

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:



6.9.3.3 NextToDot()

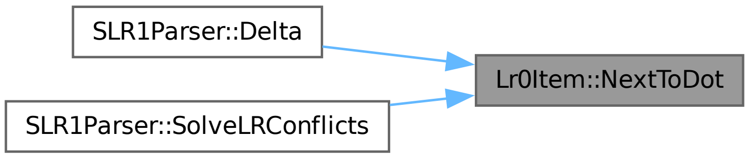
std::string Lr0Item::NextToDot () const

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:



6.9.3.4 operator==()

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

Equality operator for comparing two LR(0) items.

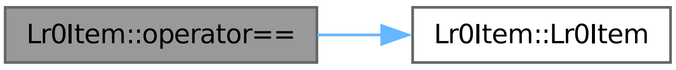
Parameters

other	The item to compare with.

Returns

true if both items are equal; false otherwise.

Here is the call graph for this function:



6.9.3.5 PrintItem()

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

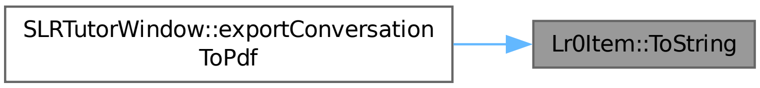
6.9.3.6 ToString()

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:



6.9.4 Member Data Documentation

6.9.4.1 antecedent_

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

6.9.4.2 consequent_

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

6.9.4.3 dot_

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

6.9.4.4 eol_

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

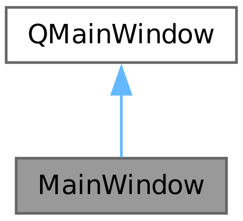
6.9.4.5 epsilon_

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

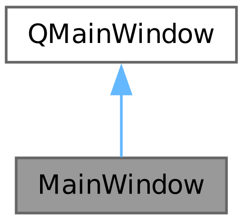
- backend/lr0_item.hpp
- backend/lr0_item.cpp

6.10 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

6.10.1 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.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 MainWindow()

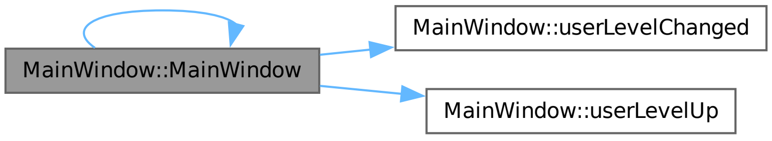
MainWindow::MainWindow (
    QWidget ∗ parent = nullptr)

Constructs the main window.

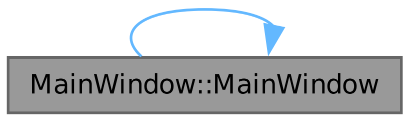
Parameters

parent	Parent widget.

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.2.2 ∼MainWindow()

MainWindow::∼MainWindow ()
Destructor.

6.10.3 Member Function Documentation

6.10.3.1 setUserLevel()

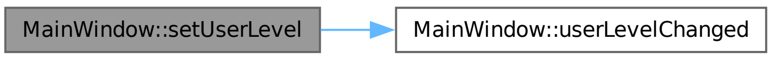
void MainWindow::setUserLevel (
unsigned lvl) [inline]

Sets the user level, clamping it to the allowed maximum.

Parameters

lvl	New level to assign.

Here is the call graph for this function:



6.10.3.2 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.

6.10.3.3 userLevel()

unsigned MainWindow::userLevel () const [inline]

Returns the current user level.

6.10.3.4 userLevelChanged

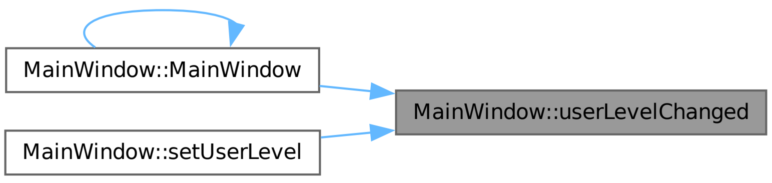
void MainWindow::userLevelChanged (
    unsigned lvl) [signal]

Emitted when the user's level changes.

Parameters

lvl	New user level.

Here is the caller graph for this function:



6.10.3.5 userLevelUp

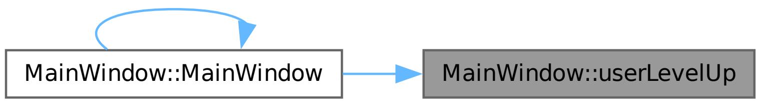
void MainWindow::userLevelUp (
    unsigned newLevel) [signal]

Emitted when the user levels up.

Parameters

newLevel	The new level achieved.

Here is the caller graph for this function:



6.10.4 Property Documentation

6.10.4.1 userLevel

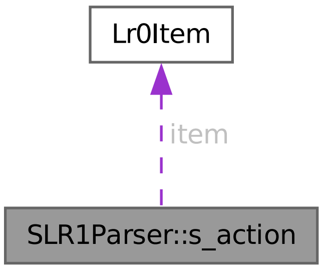
unsigned MainWindow::userLevel [read], [write]

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

- mainwindow.h
- mainwindow.cpp

6.11 SLR1Parser::s_action Struct Reference

#include <slr1_parser.hpp>
Collaboration diagram for SLR1Parser::s_action:



Public Attributes

- const Lr0Item ∗ item
- Action action

6.11.1 Member Data Documentation

6.11.1.1 action

Action SLR1Parser::s_action::action

6.11.1.2 item

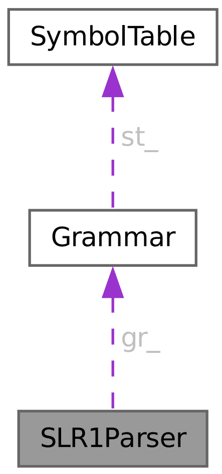
const Lr0Item∗ SLR1Parser::s_action::item
The documentation for this struct was generated from the following file:

- backend/slr1_parser.hpp

6.12 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. 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., $A \rightarrow \alpha \bullet \beta$).
- 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. 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.
- 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. 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:
- 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.

6.12.1 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.

6.12.2 Member Typedef Documentation

6.12.2.1 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.

6.12.2.2 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).

6.12.3 Member Enumeration Documentation

6.12.3.1 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	

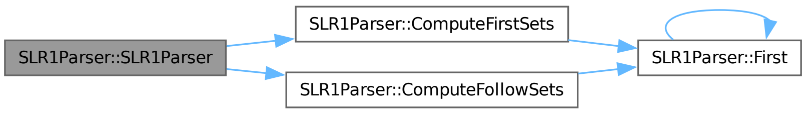
6.12.4 Constructor & Destructor Documentation

6.12.4.1 SLR1Parser() [1/2]

SLR1Parser::SLR1Parser () [default]

6.12.4.2 SLR1Parser() [2/2]

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



6.12.5 Member Function Documentation

6.12.5.1 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., $A \to \alpha \bullet \beta$).

Returns

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

6.12.5.2 Closure()

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

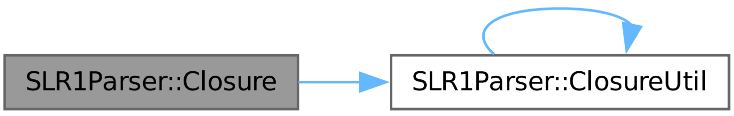
Computes the closure of a set of LR(0) items.

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.

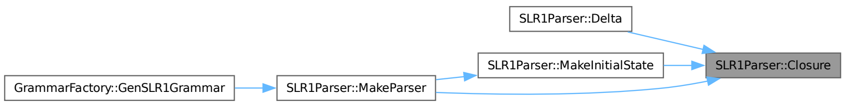
Parameters

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:



6.12.5.3 ClosureUtil()

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

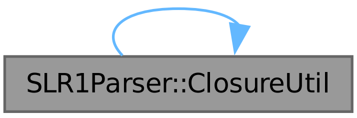
Helper function for computing the closure of LR(0) items.

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.

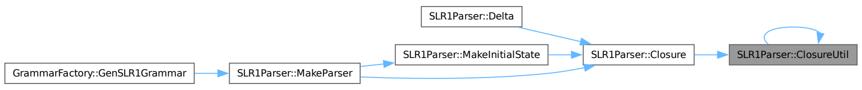
Parameters

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:



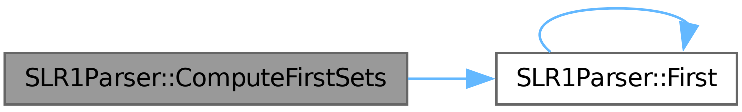
6.12.5.4 ComputeFirstSets()

void SLR1Parser::ComputeFirstSets ()

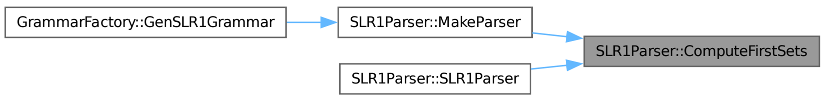
Computes the FIRST sets for all non-terminal symbols in the grammar.

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:



6.12.5.5 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( ) { } 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.

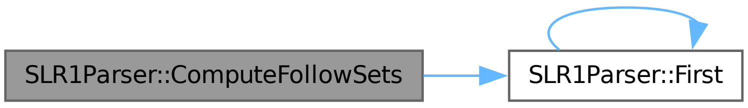
   Note

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

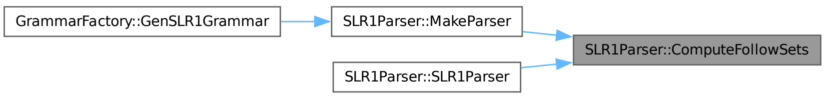
   See also

   First
   follow_sets_

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.5.6 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.

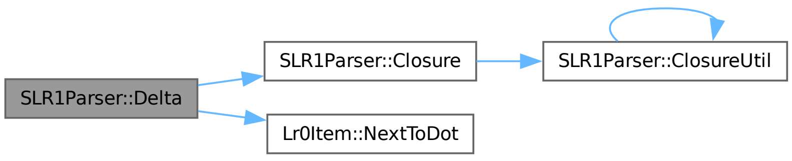
Parameters

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:



6.12.5.7 First()

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

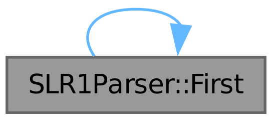
Calculates the FIRST set for a given production rule in a grammar.
The FIRST set of a production rule contains all terminal symbols that can appear at the beginning of any string derived from that rule. If the rule can derive the empty string (epsilon), epsilon is included in the FIRST set.
This function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

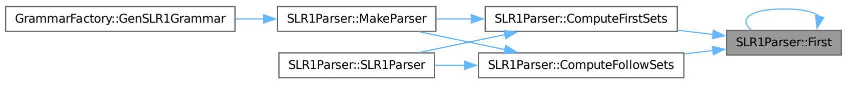
Parameters

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:


GrammarFactory::GenSLR1Grammar
SLR1Parser::MakeParser
SLR1Parser::ComputeFirstSets
SLR1Parser::First
SLR1Parser::SLR1Parser
SLR1Parser::ComputeFollowSets

6.12.5.8 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.

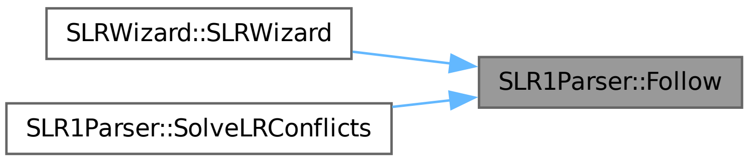
Parameters

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:



6.12.5.9 MakeInitialState()

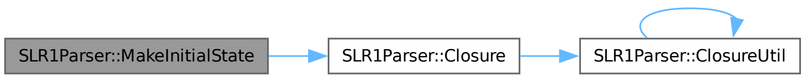
void SLR1Parser::MakeInitialState ()
Creates the initial state of the parser's state machine.
This function initializes the starting state of the parser by computing the closure of the initial set of LR(0) items derived from the grammar's start symbol. The initial state is added to the states_ set, and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

See also

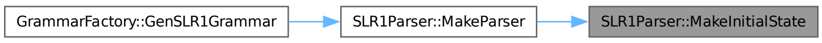
states_

transitions_

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.5.10 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.

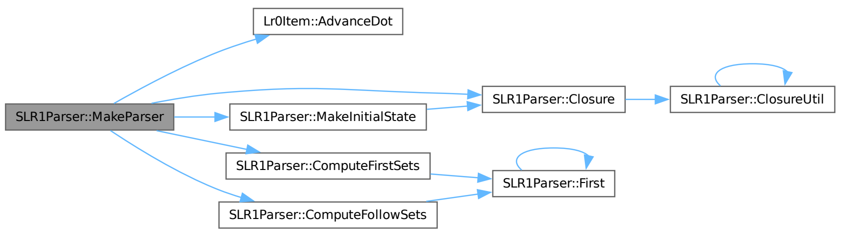
See also

actions_

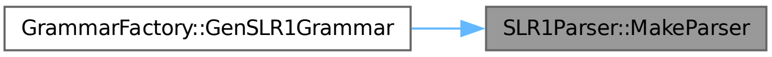
transitions_

states_

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.5.11 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.

6.12.5.12 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.

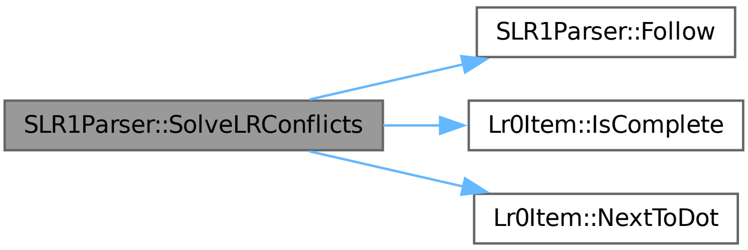
Parameters

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:



6.12.6 Member Data Documentation

6.12.6.1 actions_

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

6.12.6.2 first_sets_

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

6.12.6.3 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.

6.12.6.4 gr_

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

6.12.6.5 states_

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

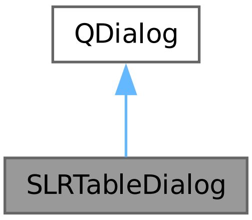
6.12.6.6 transitions_

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:

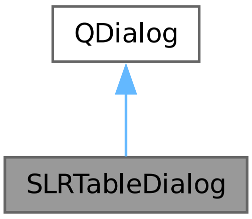
- backend/slr1_parser.hpp
- backend/slr1_parser.cpp

6.13 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.

6.13.1 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.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 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.

6.13.3 Member Function Documentation

6.13.3.1 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.

6.13.3.2 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.

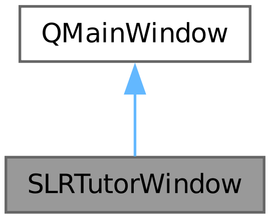
The documentation for this class was generated from the following files:

- slrtabledialog.h
- slrtabledialog.cpp

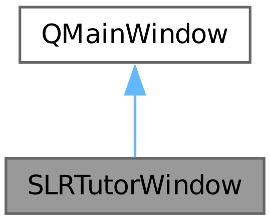
6.14 SLRTutorWindow Class Reference

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

6.14.1 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.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 SLRTutorWindow()

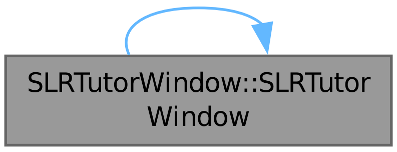
```
SLRTutorWindow::SLRTutorWindow (
                const Grammar & g,
                TutorialManager ∗ tm = nullptr,
                QWidget ∗ parent = nullptr) [explicit]
```

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

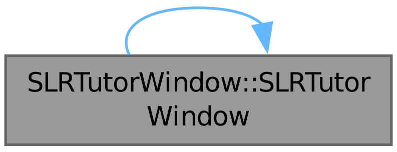
Parameters

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:



6.14.2.2 ~SLRTutorWindow()

SLRTutorWindow::~SLRTutorWindow ()

6.14.3 Member Function Documentation

6.14.3.1 addMessage()

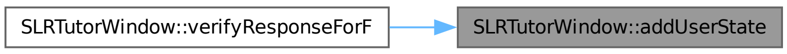
```
void SLRTutorWindow::addMessage (
                    const QString & text,
                    bool isUser)
```

< Prepares grammar in display-friendly format

6.14.3.2 addUserState()

```
void SLRTutorWindow::addUserState (
                    unsigned id)
```

< Refresh visual progress
Here is the caller graph for this function:


SLRTutorWindow::verifyResponseForF
SLRTutorWindow::addUserState

6.14.3.3 addUserTransition()

void SLRTutorWindow::addUserTransition (
    unsigned fromId,
    const std::string & symbol,
    unsigned toId)

< Register a user-created state

6.14.3.4 animateLabelColor()

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

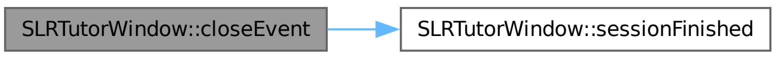
6.14.3.5 animateLabelPop()

void SLRTutorWindow::animateLabelPop (
    QLabel ∗ label)

6.14.3.6 closeEvent()

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

Here is the call graph for this function:

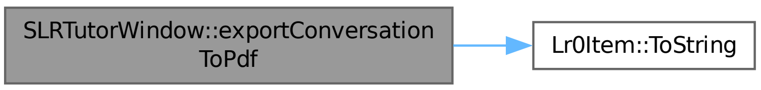


6.14.3.7 exportConversationToPdf()

void SLRTutorWindow::exportConversationToPdf (
    const QString & filePath)

< Add message to chat

Here is the call graph for this function:



6.14.3.8 feedback()

QString SLRTutorWindow::feedback ()

6.14.3.9 feedbackForA()

QString SLRTutorWindow::feedbackForA ()

6.14.3.10 feedbackForA1()

QString SLRTutorWindow::feedbackForA1 ()

6.14.3.11 feedbackForA2()

QString SLRTutorWindow::feedbackForA2 ()

6.14.3.12 feedbackForA3()

QString SLRTutorWindow::feedbackForA3 ()

6.14.3.13 feedbackForA4()

QString SLRTutorWindow::feedbackForA4 ()

6.14.3.14 feedbackForAPrime()

QString SLRTutorWindow::feedbackForAPrime ()

6.14.3.15 feedbackForB()

QString SLRTutorWindow::feedbackForB ()

6.14.3.16 feedbackForB1()

QString SLRTutorWindow::feedbackForB1 ()

6.14.3.17 feedbackForB2()

QString SLRTutorWindow::feedbackForB2 ()

6.14.3.18 feedbackForBPrime()

QString SLRTutorWindow::feedbackForBPrime ()

6.14.3.19 feedbackForC()

QString SLRTutorWindow::feedbackForC ()

6.14.3.20 feedbackForCA()

QString SLRTutorWindow::feedbackForCA ()

6.14.3.21 feedbackForCB()

QString SLRTutorWindow::feedbackForCB ()

6.14.3.22 feedbackForD()

QString SLRTutorWindow::feedbackForD ()

6.14.3.23 feedbackForD1()

QString SLRTutorWindow::feedbackForD1 ()

6.14.3.24 feedbackForD2()

QString SLRTutorWindow::feedbackForD2 ()

6.14.3.25 feedbackForDPrime()

QString SLRTutorWindow::feedbackForDPrime ()

6.14.3.26 feedbackForE()

QString SLRTutorWindow::feedbackForE ()

6.14.3.27 feedbackForE1()

QString SLRTutorWindow::feedbackForE1 ()

6.14.3.28 feedbackForE2()

QString SLRTutorWindow::feedbackForE2 ()

6.14.3.29 feedbackForF()

QString SLRTutorWindow::feedbackForF ()

6.14.3.30 feedbackForFA()

QString SLRTutorWindow::feedbackForFA ()

6.14.3.31 feedbackForG()

QString SLRTutorWindow::feedbackForG ()

6.14.3.32 fillSortedGrammar()

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

6.14.3.33 FormatGrammar()

QString SLRTutorWindow::FormatGrammar (
const Grammar & grammar)

6.14.3.34 generateQuestion()

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

Returns

The formatted question string.

6.14.3.35 launchSLRWizard()

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

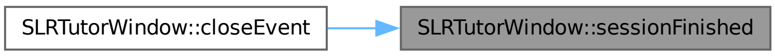
6.14.3.36 markLastUserIncorrect()

void SLRTutorWindow::markLastUserIncorrect ()

6.14.3.37 sessionFinished

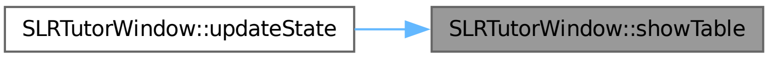
void SLRTutorWindow::sessionFinished (
int cntRight,
int cntWrong) [signal]

Here is the caller graph for this function:



6.14.3.38 showTable()

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

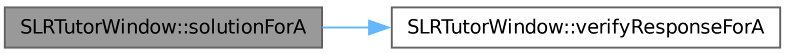


6.14.3.39 solution()

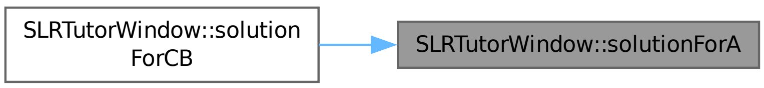
QString SLRTutorWindow::solution (
const std::string & state)

6.14.3.40 solutionForA()

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


SLRTutorWindow::solutionForA
SLRTutorWindow::verifyResponseForA

Here is the caller graph for this function:



6.14.3.41 solutionForA1()

QString SLRTutorWindow::solutionForA1 ()

6.14.3.42 solutionForA2()

QString SLRTutorWindow::solutionForA2 ()

6.14.3.43 solutionForA3()

std::vector< std::pair< std::string, std::vector< std::string > > > SLRTutorWindow::solutionForA3 ()

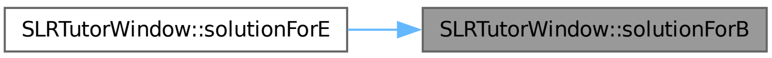
6.14.3.44 solutionForA4()

std::unordered_set< Lr0Item > SLRTutorWindow::solutionForA4 ()

6.14.3.45 solutionForB()

unsigned SLRTutorWindow::solutionForB ()

Here is the caller graph for this function:



6.14.3.46 solutionForC()

unsigned SLRTutorWindow::solutionForC ()

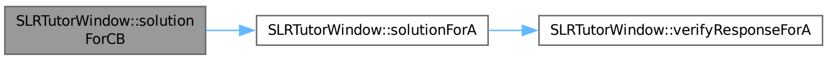
6.14.3.47 solutionForCA()

QStringList SLRTutorWindow::solutionForCA ()

6.14.3.48 solutionForCB()

std::unordered_set< Lr0Item > SLRTutorWindow::solutionForCB ()

Here is the call graph for this function:


SLRTutorWindow::solution ForCB
SLRTutorWindow::solutionForA
SLRTutorWindow::verifyResponseForA

6.14.3.49 solutionForD()

QString SLRTutorWindow::solutionForD ()

6.14.3.50 solutionForD1()

QString SLRTutorWindow::solutionForD1 ()

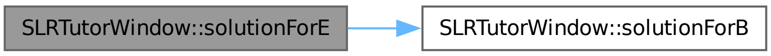
6.14.3.51 solutionForD2()

QString SLRTutorWindow::solutionForD2 ()

6.14.3.52 solutionForE()

std::ptrdiff_t SLRTutorWindow::solutionForE ()

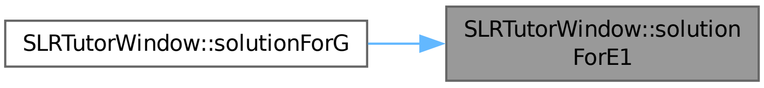
Here is the call graph for this function:



6.14.3.53 solutionForE1()

QSet< unsigned > SLRTutorWindow::solutionForE1 ()

Here is the caller graph for this function:



6.14.3.54 solutionForE2()

QMap< unsigned, unsigned > SLRTutorWindow::solutionForE2 ()

6.14.3.55 solutionForF()

QSet< unsigned > SLRTutorWindow::solutionForF ()

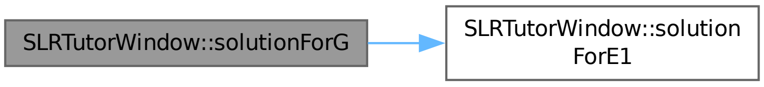
6.14.3.56 solutionForFA()

QSet< QString > SLRTutorWindow::solutionForFA ()

6.14.3.57 solutionForG()

QSet< QString > SLRTutorWindow::solutionForG ()

Here is the call graph for this function:



6.14.3.58 TeachClosure()

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

6.14.3.59 TeachClosureStep()

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

6.14.3.60 TeachDeltaFunction()

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

6.14.3.61 updateProgressPanel()

void SLRTutorWindow::updateProgressPanel ()

6.14.3.62 updateState()

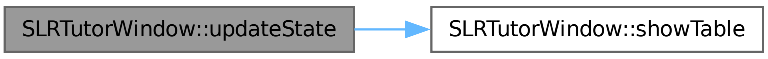
void SLRTutorWindow::updateState (
bool isCorrect)

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:



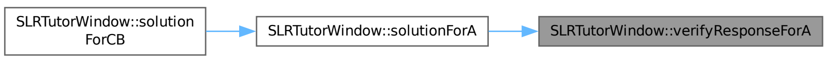
6.14.3.63 verifyResponse()

bool SLRTutorWindow::verifyResponse (
const QString & userResponse)

6.14.3.64 verifyResponseForA()

bool SLRTutorWindow::verifyResponseForA (
const QString & userResponse)

Here is the caller graph for this function:


SLRTutorWindow::solution ForCB
SLRTutorWindow::solutionForA
SLRTutorWindow::verifyResponseForA

6.14.3.65 verifyResponseForA1()

bool SLRTutorWindow::verifyResponseForA1 (
const QString & userResponse)

6.14.3.66 verifyResponseForA2()

bool SLRTutorWindow::verifyResponseForA2 (
const QString & userResponse)

6.14.3.67 verifyResponseForA3()

bool SLRTutorWindow::verifyResponseForA3 (
const QString & userResponse)

6.14.3.68 verifyResponseForA4()

bool SLRTutorWindow::verifyResponseForA4 (
const QString & userResponse)

6.14.3.69 verifyResponseForB()

bool SLRTutorWindow::verifyResponseForB (
const QString & userResponse)

6.14.3.70 verifyResponseForC()

bool SLRTutorWindow::verifyResponseForC (
const QString & userResponse)

6.14.3.71 verifyResponseForCA()

bool SLRTutorWindow::verifyResponseForCA (
const QString & userResponse)

6.14.3.72 verifyResponseForCB()

bool SLRTutorWindow::verifyResponseForCB (
const QString & userResponse)

6.14.3.73 verifyResponseForD()

bool SLRTutorWindow::verifyResponseForD (
const QString & userResponse)

6.14.3.74 verifyResponseForD1()

bool SLRTutorWindow::verifyResponseForD1 (
const QString & userResponse)

6.14.3.75 verifyResponseForD2()

bool SLRTutorWindow::verifyResponseForD2 (
    const QString & userResponse)

6.14.3.76 verifyResponseForE()

bool SLRTutorWindow::verifyResponseForE (
    const QString & userResponse)

6.14.3.77 verifyResponseForE1()

bool SLRTutorWindow::verifyResponseForE1 (
    const QString & userResponse)

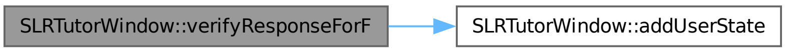
6.14.3.78 verifyResponseForE2()

bool SLRTutorWindow::verifyResponseForE2 (
    const QString & userResponse)

6.14.3.79 verifyResponseForF()

bool SLRTutorWindow::verifyResponseForF (
    const QString & userResponse)

Here is the call graph for this function:


SLRTutorWindow::verifyResponseForF
SLRTutorWindow::addUserState

6.14.3.80 verifyResponseForFA()

bool SLRTutorWindow::verifyResponseForFA (
    const QString & userResponse)

6.14.3.81 verifyResponseForG()

bool SLRTutorWindow::verifyResponseForG (
    const QString & userResponse)

6.14.3.82 verifyResponseForH()

bool SLRTutorWindow::verifyResponseForH ()

6.14.3.83 wrongAnimation()

void SLRTutorWindow::wrongAnimation ()

6.14.3.84 wrongUserResponseAnimation()

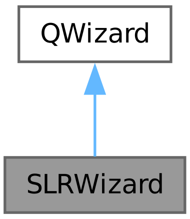
void SLRTutorWindow::wrongUserResponseAnimation ()

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

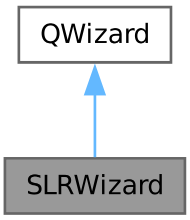
- slrtutorwindow.h
- slrtutorwindow.cpp

6.15 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.

6.15.1 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).

6.15.2 Constructor & Destructor Documentation

6.15.2.1 SLRWizard()

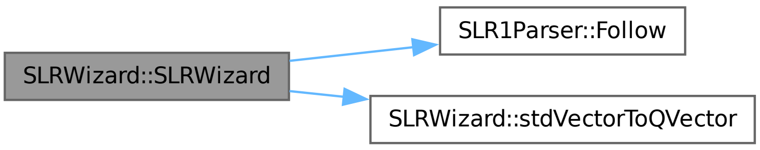
SLRWizard::SLRWizard (
    SLR1Parser & parser,
    const QVector< QVector< QString > > & rawTable,
    const QStringList & colHeaders,
    const QVector< QPair< QString, QVector< QString > > > & sortedGrammar,
    QWidget $*$ parent = nullptr) [inline]

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:



6.15.3 Member Function Documentation

6.15.3.1 stdVectorToQVector()

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

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

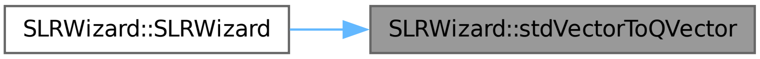
Parameters

vec	The input vector of strings.

Returns

A QVector of QStrings.

Here is the caller graph for this function:

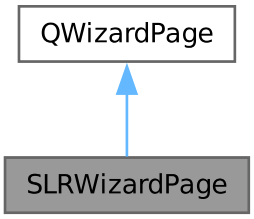


The documentation for this class was generated from the following file:

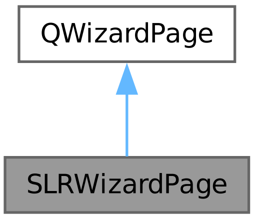
- slrwizard.h

6.16 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.

6.16.1 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.

6.16.2 Constructor & Destructor Documentation

6.16.2.1 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

6.17 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.

6.17.1 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.

6.17.2 Member Function Documentation

6.17.2.1 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.

6.17.3 Member Data Documentation

6.17.3.1 id_

```
unsigned int state::id_
```

Unique identifier of the state.

6.17.3.2 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

6.18 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.

6.18.1 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 $.

6.18.2 Member Function Documentation

6.18.2.1 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.

6.18.2.2 IsTerminal()

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

Checks if a symbol is a terminal.

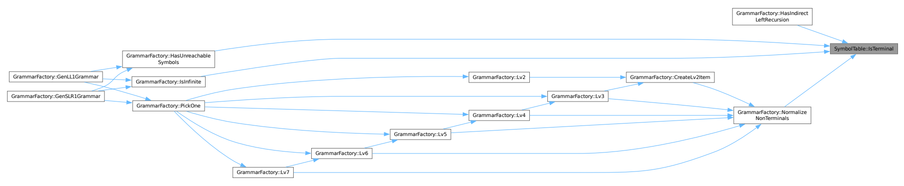
Parameters

s	Symbol identifier to check.

Returns

true if the symbol is terminal, otherwise false.

Here is the caller graph for this function:


GrammarFactory::HasIndirect LeftRecursion
SymbolTable::IsTerminal
GrammarFactory::HasUnreachable Symbols
GrammarFactory::GenLL1Grammar
GrammarFactory::IsInfinite
GrammarFactory::GenSLR1Grammar
GrammarFactory::PickOne
GrammarFactory::Lv2
GrammarFactory::CreateLv2Item
GrammarFactory::Lv3
GrammarFactory::Lv4
GrammarFactory::Normalize NonTerminals
GrammarFactory::Lv5
GrammarFactory::Lv6
GrammarFactory::Lv7

6.18.2.3 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.

6.18.2.4 PutSymbol()

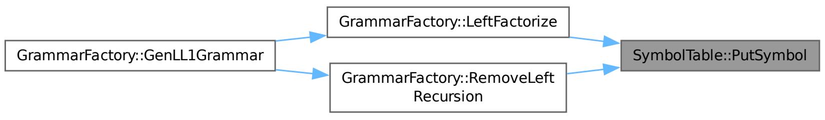
void SymbolTable::PutSymbol (
    const std::string & identifier,
    bool isTerminal)

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:


GrammarFactory::LeftFactorize
GrammarFactory::GenLL1Grammar
SymbolTable::PutSymbol
GrammarFactory::RemoveLeft Recursion

6.18.3 Member Data Documentation

6.18.3.1 EOL_

std::string SymbolTable::EOL_ {"$"}

End-of-line symbol used in parsing, initialized as "$".

6.18.3.2 EPSILON_

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

6.18.3.3 non_terminals_

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

6.18.3.4 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.

6.18.3.5 terminals_

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

6.18.3.6 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

6.19 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

6.19.1 Detailed Description

TreeNode structure used to build derivation trees.

6.19.2 Member Data Documentation

6.19.2.1 children

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

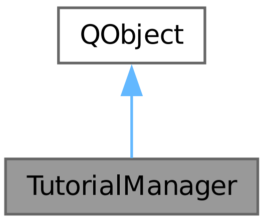
6.19.2.2 label

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

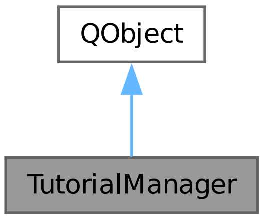
- lltutorwindow.h

6.20 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.

6.20.1 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.

6.20.2 Constructor & Destructor Documentation

6.20.2.1 TutorialManager()

```
TutorialManager::TutorialManager (
        QWidget ∗ rootWindow)
```

Constructs a TutorialManager for a given window.

Parameters

rootWindow	The main application window used for relative positioning.

6.20.3 Member Function Documentation

6.20.3.1 addStep()

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

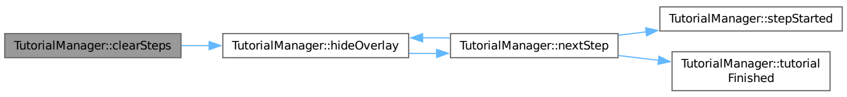
Adds a new step to the tutorial sequence.

Parameters

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

6.20.3.2 clearSteps()

void TutorialManager::clearSteps ()
Clears all steps in the tutorial.
Here is the call graph for this function:

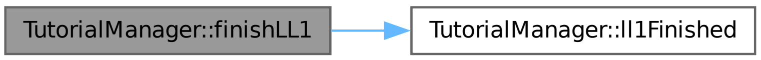

TutorialManager::clearSteps
TutorialManager::hideOverlay
TutorialManager::nextStep
TutorialManager::stepStarted
TutorialManager::tutorial Finished

6.20.3.3 eventFilter()

bool TutorialManager::eventFilter (
QObject * obj,
QEvent * ev) [override], [protected]
Intercepts UI events to handle overlay behavior.

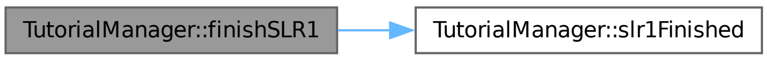
6.20.3.4 finishLL1()

void TutorialManager::finishLL1 ()
Ends the LL(1) tutorial sequence and emits its corresponding signal.
Here is the call graph for this function:



6.20.3.5 finishSLR1()

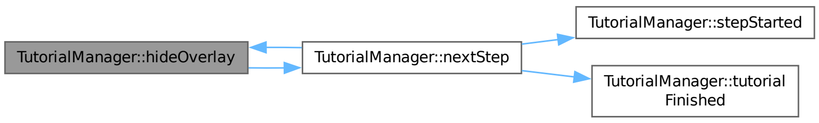
void TutorialManager::finishSLR1 ()
Ends the SLR(1) tutorial sequence and emits its corresponding signal.
Here is the call graph for this function:



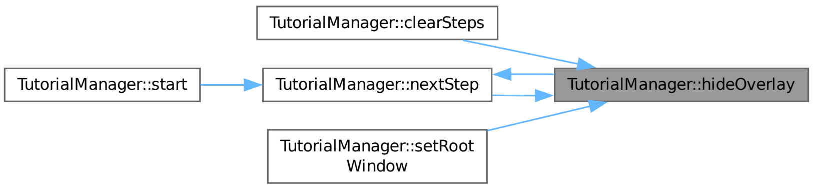
6.20.3.6 hideOverlay()

void TutorialManager::hideOverlay ()
Hides the tutorial overlay immediately.

Here is the call graph for this function:


TutorialManager::hideOverlay
TutorialManager::nextStep
TutorialManager::stepStarted
TutorialManager::tutorial Finished

Here is the caller graph for this function:

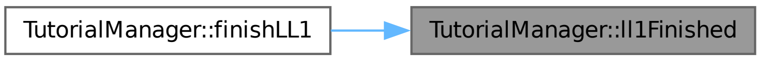

TutorialManager::clearSteps
TutorialManager::start
TutorialManager::nextStep
TutorialManager::hideOverlay
TutorialManager::setRoot Window

6.20.3.7 ll1Finished

void TutorialManager::ll1Finished () [signal]

Emitted when the LL(1) tutorial ends.

Here is the caller graph for this function:

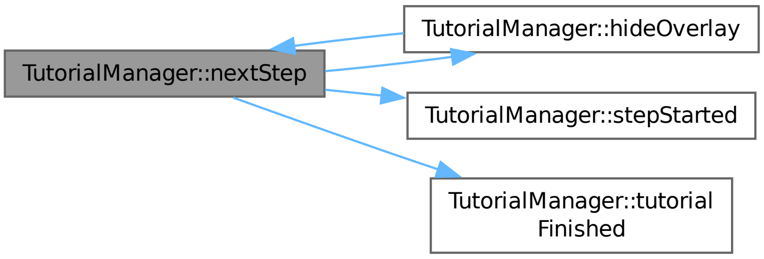


6.20.3.8 nextStep

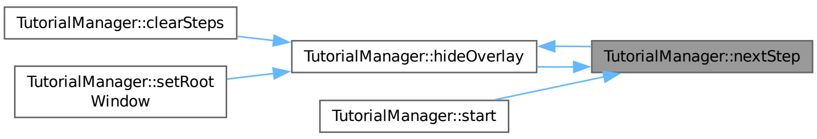
void TutorialManager::nextStep () [slot]

Advances to the next tutorial step.

Here is the call graph for this function:


TutorialManager::nextStep
TutorialManager::hideOverlay
TutorialManager::stepStarted
TutorialManager::tutorial Finished

Here is the caller graph for this function:


TutorialManager::clearSteps
TutorialManager::setRoot Window
TutorialManager::hideOverlay
TutorialManager::start
TutorialManager::nextStep

6.20.3.9 setRootWindow()

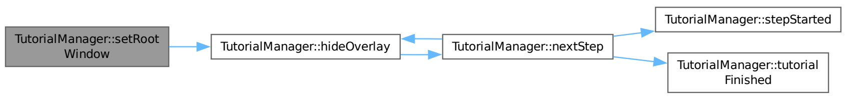
void TutorialManager::setRootWindow (
QWidget ∗ newRoot)

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:

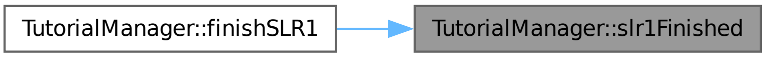

TutorialManager::setRoot Window
TutorialManager::hideOverlay
TutorialManager::nextStep
TutorialManager::stepStarted
TutorialManager::tutorial Finished

6.20.3.10 slr1Finished

void TutorialManager::slr1Finished () [signal]

Emitted when the SLR(1) tutorial ends.

Here is the caller graph for this function:

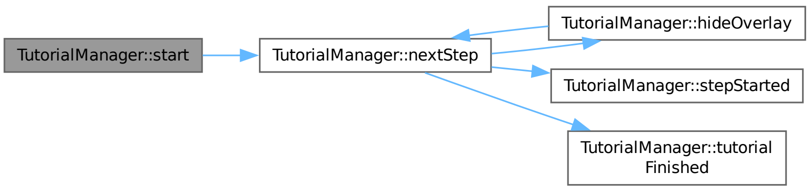


6.20.3.11 start()

void TutorialManager::start ()

Starts the tutorial from the beginning.

Here is the call graph for this function:


TutorialManager::start
TutorialManager::nextStep
TutorialManager::hideOverlay
TutorialManager::stepStarted
TutorialManager::tutorial Finished

6.20.3.12 stepStarted

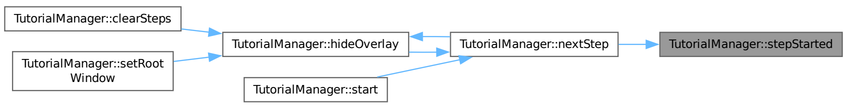
void TutorialManager::stepStarted (
int index) [signal]

Emitted when a new tutorial step starts.

Parameters

index	Index of the current step.

Here is the caller graph for this function:

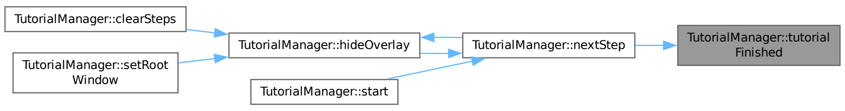

TutorialManager::clearSteps
TutorialManager::setRoot Window
TutorialManager::hideOverlay
TutorialManager::start
TutorialManager::nextStep
TutorialManager::stepStarted

6.20.3.13 tutorialFinished

void TutorialManager::tutorialFinished () [signal]

Emitted when the full tutorial is finished.

Here is the caller graph for this function:


TutorialManager::clearSteps
TutorialManager::setRoot Window
TutorialManager::hideOverlay
TutorialManager::start
TutorialManager::nextStep
TutorialManager::tutorial Finished

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

- tutorialmanager.h
- tutorialmanager.cpp

6.21 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.

6.21.1 Detailed Description

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

6.21.2 Member Data Documentation

6.21.2.1 htmlText

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

6.21.2.2 target

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

- tutorialmanager.h

6.22 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.

6.22.1 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.

6.22.2 Member Function Documentation

6.22.2.1 clear()

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

6.22.2.2 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.

6.22.2.3 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.

6.22.2.4 pop()

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

6.22.2.5 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:

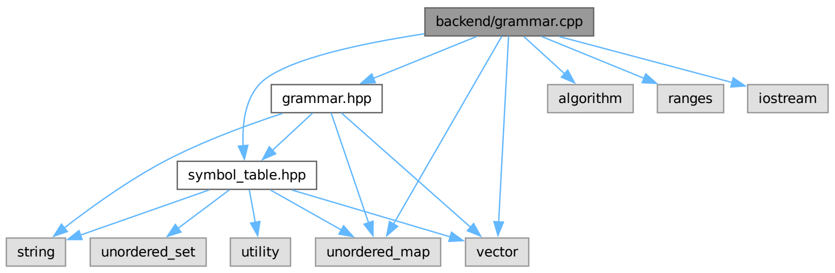
- UniqueQueue.h

Chapter 7

File Documentation

7.1 backend/grammar.cpp File Reference

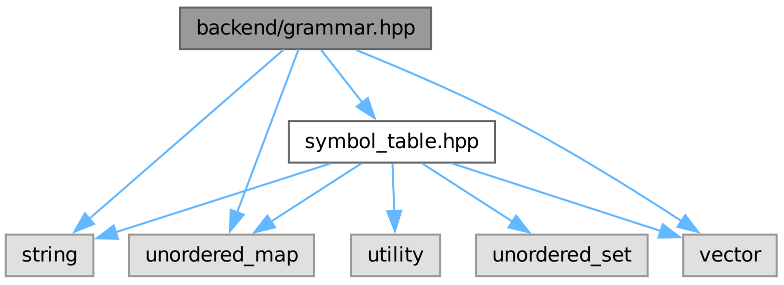
#include "grammar.hpp"
#include "symbol_table.hpp"
#include <algorithm>
#include <ranges>
#include <iostream>
#include <unordered_map>
#include <vector>
Include dependency graph for grammar.cpp:



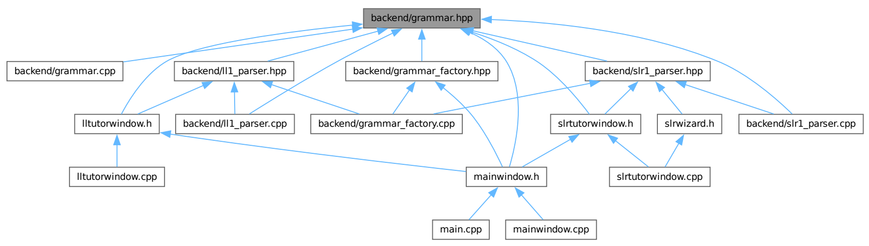
7.2 backend/grammar.hpp File Reference

#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.

7.2.1 Typedef Documentation

7.2.1.1 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"}

7.3 grammar.hpp

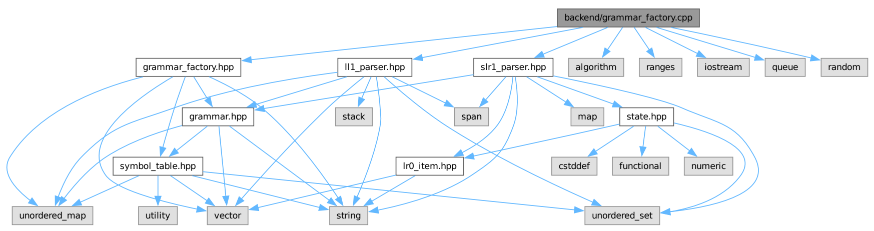
Go to the documentation of this file.

```
#pragma once
#include "symbol_table.hpp"
#include <string>
#include <unordered_map>
#include <vector>
```

```

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

struct Grammar {

    Grammar();
    explicit Grammar(
        const std::unordered_map<std::string, std::vector<production»&
            grammar);


    void SetAxiom(const std::string& axiom);

    bool HasEmptyProduction(const std::string& antecedent) const;

    std::vector<std::pair<const std::string, production» FilterRulesByConsequent(
        const std::string& arg) const;

    void Debug() const; //NOSONAR

    void AddProduction(const std::string&              antecedent,
                       const std::vector<std::string>& consequent);

    std::vector<std::string> Split(const std::string& s);

    std::unordered_map<std::string, std::vector<production» g_;

    std::string axiom_;

    SymbolTable st_;
};
```

7.4 backend/grammar_factory.cpp File Reference

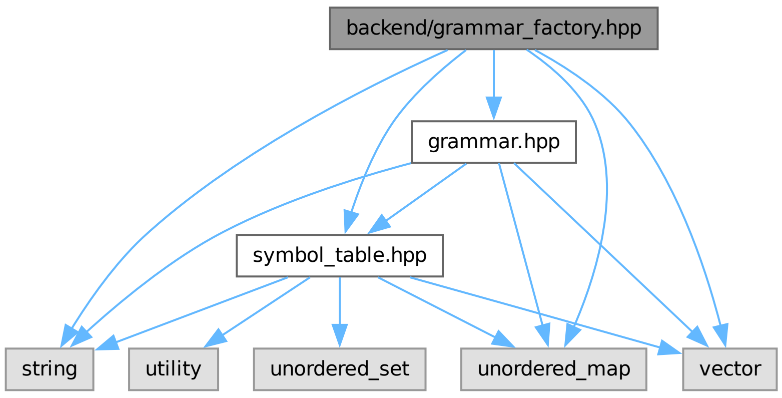
#include "grammar_factory.hpp"
#include "ll1_parser.hpp"
#include "slr1_parser.hpp"
#include <algorithm>
#include <ranges>
#include <iostream>
#include <queue>
#include <random>
Include dependency graph for grammar_factory.cpp:



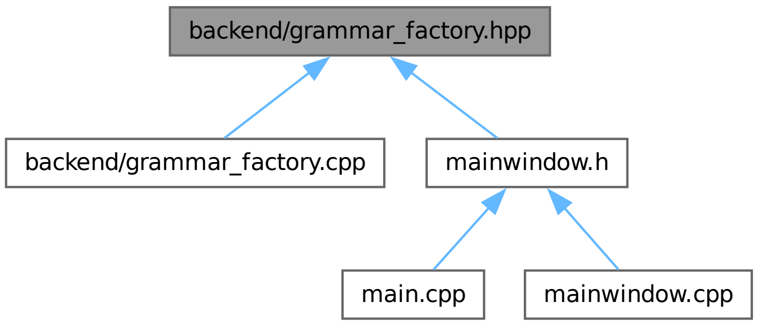
7.5 backend/grammar_factory.hpp File Reference

#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.

7.6 grammar_factory.hpp

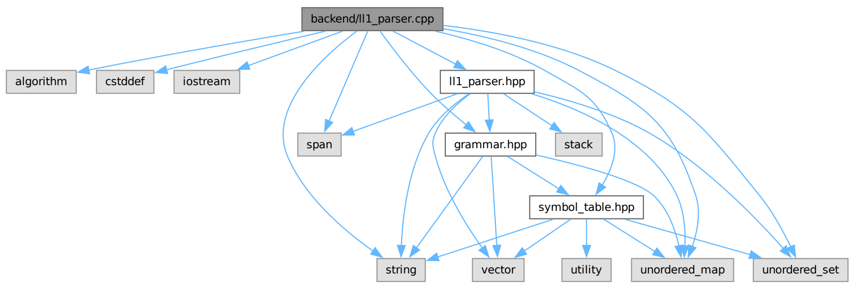
Go to the documentation of this file.

```
#pragma once

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

struct GrammarFactory {

    struct FactoryItem {
        std::unordered_map<std::string, std::vector<production» g_;

        SymbolTable st_;

        explicit FactoryItem(const std::unordered_map<std::string, std::vector<production» &grammar);
    };

    void Init();

    Grammar PickOne(int level);

    Grammar GenLL1Grammar(int level);
    Grammar GenSLR1Grammar(int level);

    Grammar Lv1();

    Grammar Lv2();

    Grammar Lv3();

    Grammar Lv4();

    Grammar Lv5();

    Grammar Lv6();

    Grammar Lv7();

    FactoryItem CreateLv2Item();

    bool HasUnreachableSymbols(Grammar& grammar) const;

    bool IsInfinite(Grammar& grammar) const;

    bool HasDirectLeftRecursion(const Grammar& grammar) const;

    bool HasIndirectLeftRecursion(const Grammar& grammar) const;

    bool HasCycle(const std::unordered_map<std::string, std::unordered_set<std::string»& graph) const;

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



    void RemoveLeftRecursion(Grammar& grammar);



    void LeftFactorize(Grammar& grammar);

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

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

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

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

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

    std::unordered_map<std::string, std::vector<production»
    Merge(const FactoryItem& base, const FactoryItem& cmb) const;

    std::vector<FactoryItem> items;

    std::vector<std::string> terminal_alphabet_{”a”, ”b”, ”c”, ”d”, ”e”, ”f”,
                                                ”g”, ”h”, ”i”, ”j”, ”k”, ”l”};

    std::vector<std::string> non_terminal_alphabet_{”A”, ”B”, ”C”, ”D”,
                                                    ”E”, ”F”, ”G”};
};
```

7.7 backend/ll1_parser.cpp File Reference

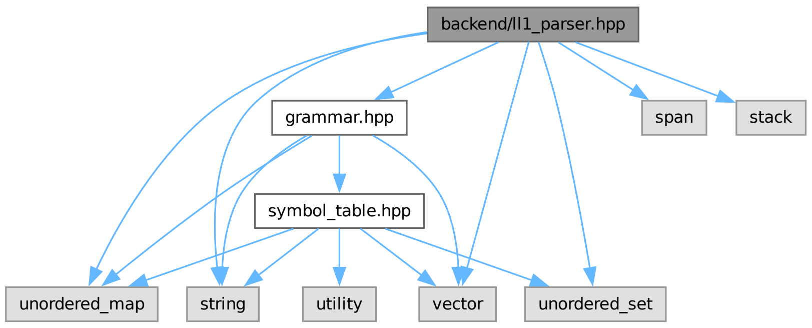
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <span>
#include <string>

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

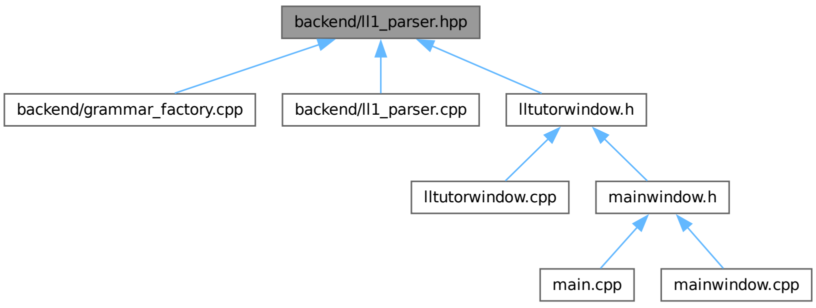


7.8 backend/ll1_parser.hpp File Reference

#include "grammar.hpp"
#include <span>
#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

7.9 ll1_parser.hpp

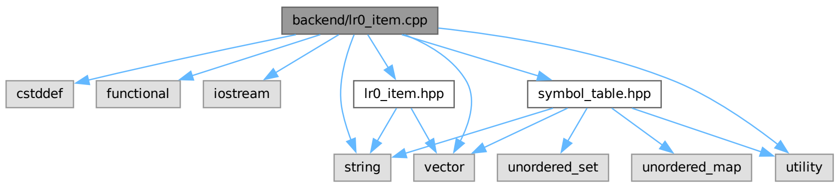
Go to the documentation of this file.

```
#pragma once
#include "grammar.hpp"
#include <span>
#include <stack>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

class LL1Parser {

    using ll1_table = std::unordered_map<
        std::string, std::unordered_map<std::string, std::vector<production»>;

  public:
    LL1Parser() = default;
    explicit LL1Parser(Grammar gr);

    bool CreateLL1Table();

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

    void ComputeFirstSets();

    void ComputeFollowSets();

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

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

    ll1_table ll1_t_;

    Grammar gr_;

    std::unordered_map<std::string, std::unordered_set<std::string»
        first_sets_;

    std::unordered_map<std::string, std::unordered_set<std::string»
        follow_sets_;
};
```

7.10 backend/lr0_item.cpp File Reference

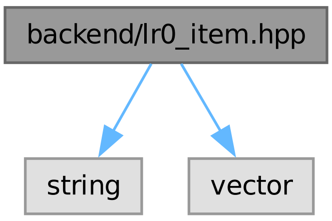
#include <cstddef>
#include <functional>

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

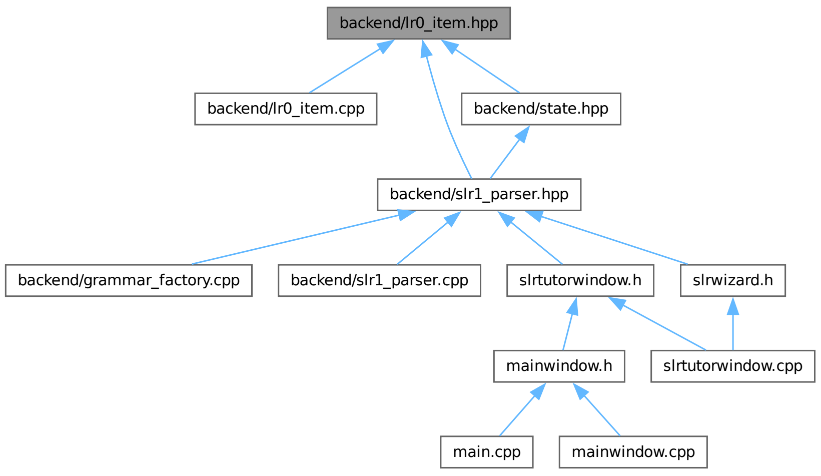


7.11 backend/lr0_item.hpp File Reference

#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.

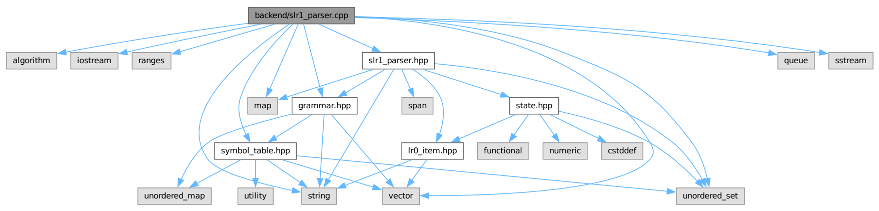
7.12 lr0_item.hpp

Go to the documentation of this file.

```
#pragma once

#include <string>
#include <vector>


struct Lr0Item {
    std::string antecedent_;

    std::vector<std::string> consequent_;

    std::string epsilon_;

    std::string eol_;

    unsigned int dot_ = 0;

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

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

    std::string NextToDot() const;

    void PrintItem() const;

    std::string ToString() const;

    void AdvanceDot();

    bool IsComplete() const;

    bool        operator==(const Lr0Item& other) const;
};

namespace std {
template <> struct hash<Lr0Item> {
    size_t operator()(const Lr0Item& item) const;
};
} // namespace std
```

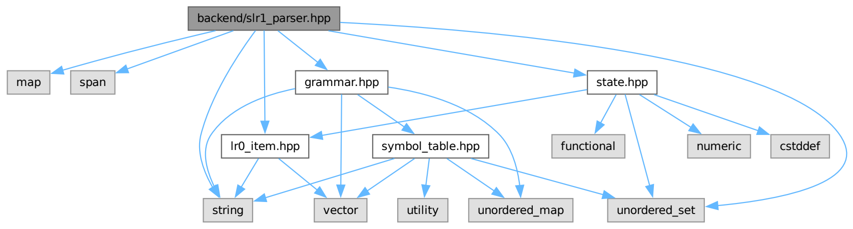
7.13 backend/slr1_parser.cpp File Reference

#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:

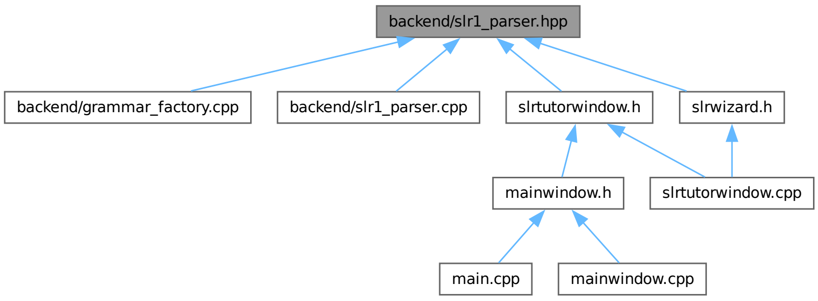


7.14 backend/slr1_parser.hpp File Reference

#include <map>
#include <span>
#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

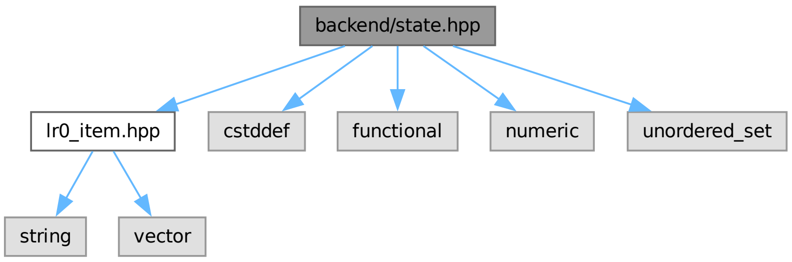
7.15 slr1_parser.hpp

Go to the documentation of this file.

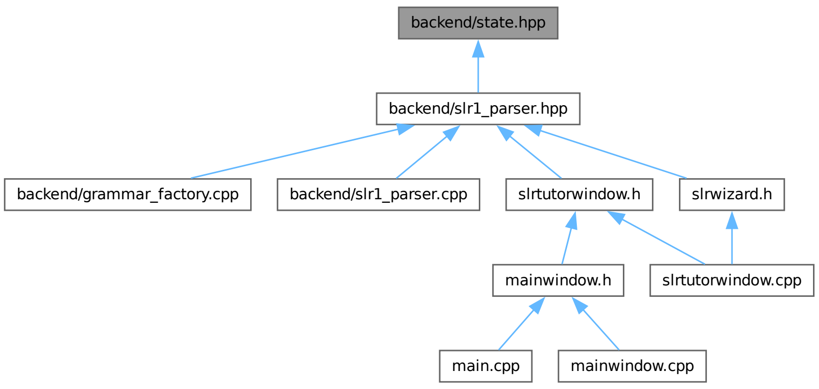
```
#pragma once

#include <map>
#include <span>
#include <string>
#include <unordered_set>

#include ”grammar.hpp”
#include ”lr0_item.hpp”
#include ”state.hpp”

class SLR1Parser {
  public:
    enum class Action { Shift, Reduce, Accept, Empty };

    struct s_action {
        const Lr0Item* item;
        Action         action;
    };

    using action_table =
        std::map<unsigned int, std::map<std::string, SLR1Parser::s_action»;

    using transition_table =
        std::map<unsigned int, std::map<std::string, unsigned int»;

    SLR1Parser() = default;
    explicit SLR1Parser(Grammar gr);

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

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

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

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

    bool SolveLRConflicts(const state& st);

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

    void ComputeFirstSets();

    void ComputeFollowSets();

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

    void MakeInitialState();

    bool MakeParser();

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

    Grammar gr_;

    std::unordered_map<std::string, std::unordered_set<std::string»
        first_sets_;

    std::unordered_map<std::string, std::unordered_set<std::string»
        follow_sets_;

    action_table actions_;

    transition_table transitions_;

    std::unordered_set<state> states_;
};
```

7.16 backend/state.hpp File Reference

#include "lr0_item.hpp"
#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.

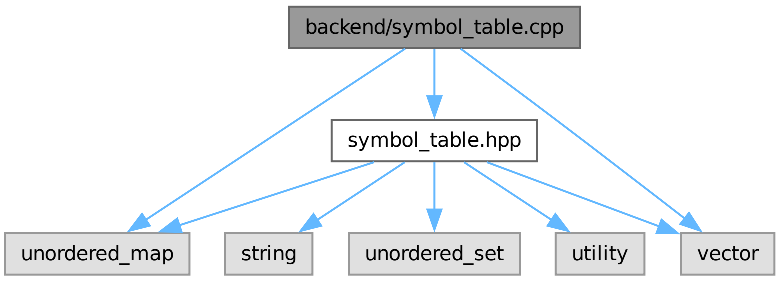
7.17 state.hpp

Go to the documentation of this file.

```
#pragma once
#include "lr0_item.hpp"
#include <cstddef>
#include <functional>
#include <numeric>
#include <unordered_set>

struct state {
    std::unordered_set<Lr0Item> items_;

    unsigned int id_;

    bool operator==(const state& other) const { return other.items_ == items_; }
};

namespace std {
template <> struct hash<state> {
    size_t operator()(const state& st) const {
        size_t seed =
            std::accumulate(st.items_.begin(), st.items_.end(), 0,
                            [](size_t acc, const Lr0Item& item) {
                                return acc ^ (std::hash<Lr0Item>()(item));
                            });
        return seed;
    }
};
} // namespace std
```

7.18 backend/symbol_table.cpp File Reference

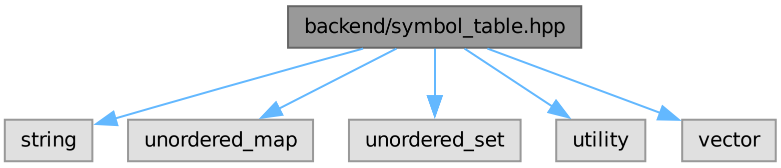
#include "symbol_table.hpp"
#include <unordered_map>
#include <vector>
Include dependency graph for symbol_table.cpp:



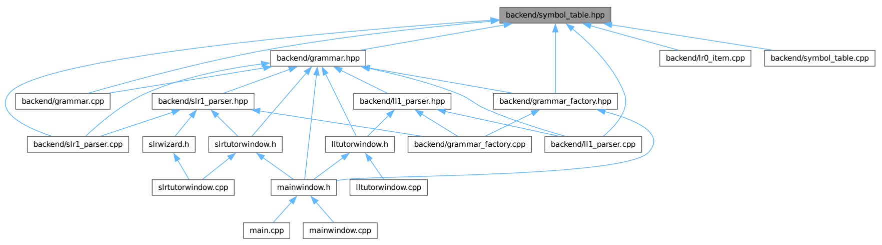
7.19 backend/symbol_table.hpp File Reference

#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.

7.19.1 Enumeration Type Documentation

7.19.1.1 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	

7.20 symbol_table.hpp

Go to the documentation of this file.

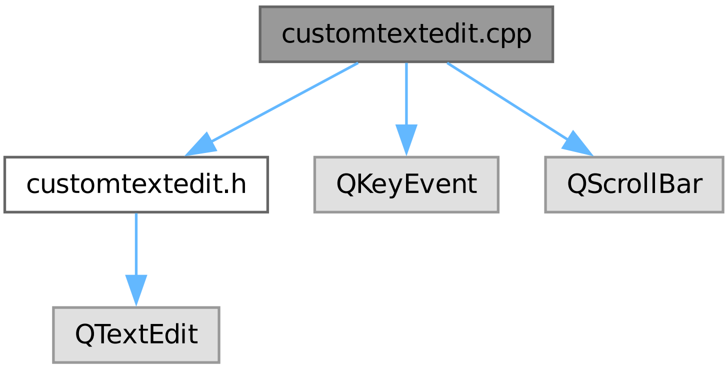
```
#pragma once
#include <string>
```

```
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

enum class symbol_type { NO_TERMINAL, TERMINAL };

struct SymbolTable {
    std::string EOL_{"$"};

    std::string EPSILON_{"EPSILON"};

    std::unordered_map<std::string, symbol_type> st_{{EOL_, symbol_type::TERMINAL},
                                                     {EPSILON_, symbol_type::TERMINAL}};

    std::unordered_set<std::string> terminals_{EOL_};

    std::unordered_set<std::string> terminals_wtho_eol_{};

    std::unordered_set<std::string> non_terminals_;

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

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

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

    bool IsTerminalWthoEol(const std::string& s) const;
};
```

7.21 CHANGELOG.md File Reference

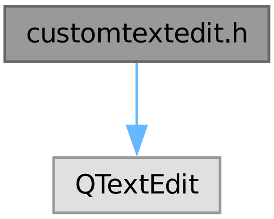
7.22 customtextedit.cpp File Reference

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

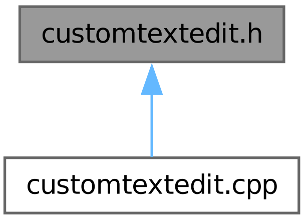


7.23 customtextedit.h File Reference

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



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



Classes

- class CustomTextEdit

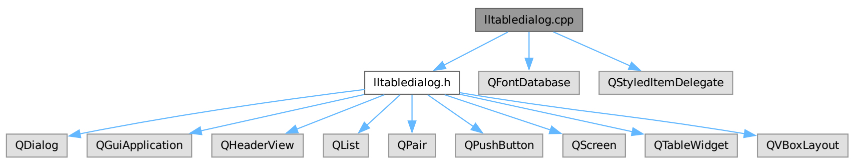
7.24 customtextedit.h

Go to the documentation of this file.

```
#ifndef CUSTOMTEXTEDIT_H
#define CUSTOMTEXTEDIT_H

#include <QTextEdit>

class CustomTextEdit : public QTextEdit {
    Q_OBJECT
  public:
    explicit CustomTextEdit(QWidget* parent = nullptr);

  signals:
    void sendRequested();

  protected:
    void keyPressEvent(QKeyEvent* event) override;
};

#endif // CUSTOMTEXTEDIT_H
```

7.25 lltabledialog.cpp File Reference

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

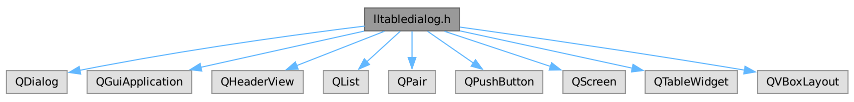


Classes

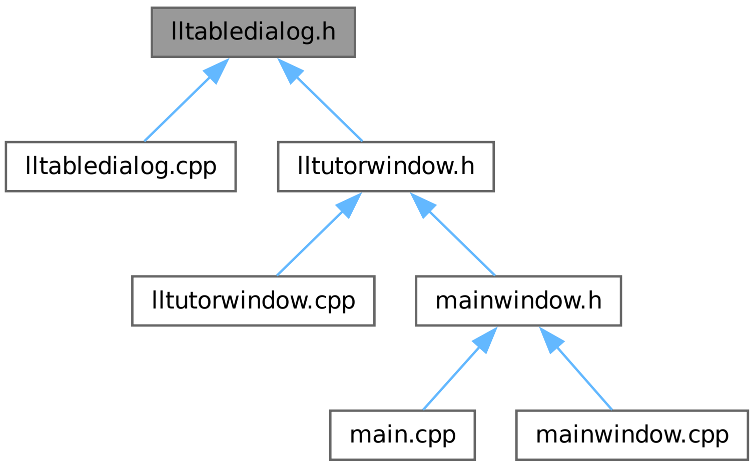
- class CenterAlignDelegate

7.26 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.

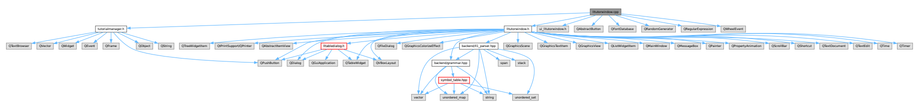
7.27 lltabledialog.h

Go to the documentation of this file.

```
#ifndef LLTABLEDIALOG_H
#define LLTABLEDIALOG_H

#include <QDialog>
#include <QGuiApplication>
#include <QHeaderView>
#include <QList>
#include <QPair>
#include <QPushButton>
#include <QScreen>
#include <QTableWidget>
#include <QVBoxLayout>

class LLTableDialog : public QDialog {
    Q_OBJECT
  public:
    LLTableDialog(const QStringList& rowHeaders, const QStringList& colHeaders,
                  QWidget*                 parent,
                  QVector<QVector<QString>>* initialData = nullptr);

    QVector<QVector<QString>> getTableData() const;

    void setInitialData(const QVector<QVector<QString>>& data);

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

  signals:
    void submitted(const QVector<QVector<QString>>& data);

  private:
    QTableWidget* table;
    QPushButton*  submitButton;
};

#endif // LLTABLEDIALOG_H
```

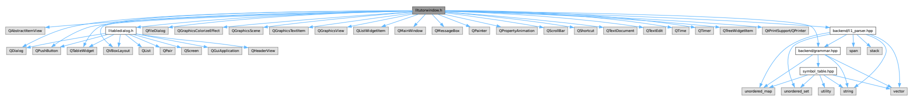
7.28 lltutorwindow.cpp File Reference

#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:

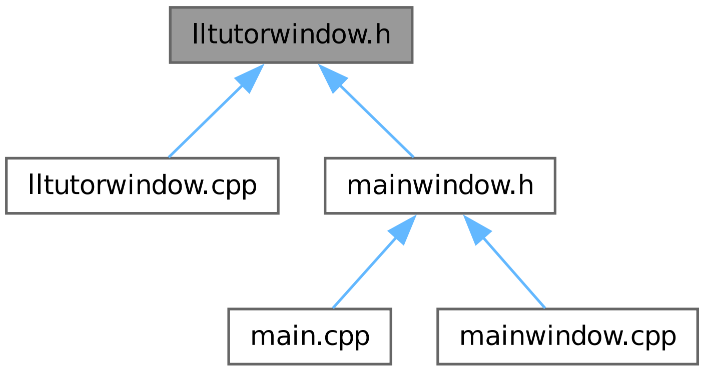


7.29 lltutorwindow.h File Reference

#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 }

7.29.1 Enumeration Type Documentation

7.29.1.1 State

enum class State [strong]

Enumerator

A	
A1	
A2	
A_prime	
B	
B1	
B2	
B_prime	
C	
C_prime	
fin	

7.30 lltutorwindow.h

Go to the documentation of this file.

```
#ifndef LLTUTORWINDOW_H
#define LLTUTORWINDOW_H

#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"

class TutorialManager;

namespace Ui {
class LLTutorWindow;
}

// ====== LL(1) Tutor States ===================================
enum class State { A, A1, A2, A_prime, B, B1, B2, B_prime, C, C_prime, fin };

// ====== LL(1) Tutor Main Class ===============================
class LLTutorWindow : public QMainWindow {
    Q_OBJECT

  public:
    // ====== Derivation Tree (used in TeachFirst) =============
    struct TreeNode {
        QString                                label;
        std::vector<std::unique_ptr<TreeNode>> children;
    };

    // ====== Constructor / Destructor ==========================
    explicit LLTutorWindow(const Grammar&    grammar,
                           TutorialManager* tm      = nullptr,
                           QWidget*         parent = nullptr);
    ~LLTutorWindow();

    // ====== State Machine & Question Logic ====================
    QString generateQuestion();

    void updateState(bool isCorrect);

    QString FormatGrammar(const Grammar& grammar);

    // ====== UI Interaction ====================================
    void addMessage(const QString& text,
                    bool           isUser);
    void addWidgetMessage(QWidget* widget);
    void
    exportConversationToPdf(const QString& filePath);
    void showTable();
    void showTableForCPrime();
    void updateProgressPanel(); // Update progress panel

    // ====== Visual Feedback / Animations ======================
    void animateLabelPop(QLabel* label);
    void animateLabelColor(QLabel* label, const QColor& flashColor);
    void wrongAnimation();
    void
    wrongUserResponseAnimation();
    void markLastUserIncorrect();

    // ====== Tree Generation (TeachFirst mode) =================
    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); // Display derivation tree visually

    // ====== User Response Verification ========================
    bool verifyResponse(const QString& userResponse); // Delegates to current
                                                      // state's verification
    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(); // C is non-textual (checks internal table)

    // ====== Expected Solutions (Auto-generated) ================
    QString       solution(const std::string& state);
    QStringList   solutionForA();
    QString       solutionForA1();
    QString       solutionForA2();
    QSet<QString> solutionForB();
    QSet<QString> solutionForB1();
    QSet<QString> solutionForB2();

    // ====== Feedback (Corrective Explanations) =================
    QString feedback(); // Delegates by state
    QString feedbackForA();
    QString feedbackForA1();
    QString feedbackForA2();
    QString feedbackForAPrime();
    QString feedbackForB();
    QString feedbackForB1();
    QString feedbackForB2();
    QString feedbackForBPrime();
    QString feedbackForC();
    QString feedbackForCPrime();
    void    feedbackForB1TreeWidget();   // TreeWidget of Teach (LL1 TeachFirst)
    void    feedbackForB1TreeGraphics(); // Show derivation tree
    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);
  private slots:
    void on_confirmButton_clicked();
    void on_userResponse_textChanged();

  signals:
    void sessionFinished(int cntRight, int cntWrong);

  protected:
    void closeEvent(QCloseEvent* event) override {
        emit sessionFinished(cntRightAnswers, cntWrongAnswers);
        QWidget::closeEvent(event);
    }

    bool eventFilter(QObject* obj, QEvent* event) override;

  private:
    // ====== Core Objects ======================================
    Ui::LLTutorWindow* ui;
    Grammar            grammar;
    LL1Parser          ll1;

    // ====== State & Grammar Tracking ==========================
    State          currentState;
    size_t         currentRule        = 0;
    const unsigned kMaxHighlightTries = 3;
    const unsigned kMaxTotalTries     = 5;
    unsigned       lltries            = 0;
```

```
    unsigned      cntRightAnswers = 0, cntWrongAnswers = 0;

    using Cell = std::pair<QString, QString>;
    std::vector<Cell> lastWrongCells;
    LLTableDialog*    currentDlg = nullptr;

    QVector<QString>                      sortedNonTerminals;
    QVector<QPair<QString, QVector<QString>>> sortedGrammar;
    QString                               formattedGrammar;

    QMap<QString, QMap<QString, QVector<QString>>> lltable;
    QVector<QVector<QString>>                       rawTable;
    QSet<QString>                                   solutionSet;

    // ====== Conversation Logging ===============================
    struct MessageLog {
        QString message;
        bool    isUser;
        bool    isCorrect = true;
        MessageLog(const QString& message, bool isUser)
            : message(message), isUser(isUser) {}
        void toggleIsCorrect() { isCorrect = false; }
    };

    QVector<MessageLog> conversationLog;
    QWidget*            lastUserMessage       = nullptr;
    qsizetype           lastUserMessageLogIdx = -1;

    QMap<QString, QString> userCAB;
    QMap<QString, QString> userSIG;
    QMap<QString, QString> userSD;

    // ====== Helper Conversions =================================
    std::vector<std::string> qvectorToStdVector(const QVector<QString>& qvec);
    QVector<QString> stdVectorToQVector(const std::vector<std::string>& vec);
    QSet<QString>
    stdUnorderedSetToQSet(const std::unordered_set<std::string>& uset);
    std::unordered_set<std::string>
    qsetToStdUnorderedSet(const QSet<QString>& qset);

    void setupTutorial();

    void
    fillSortedGrammar(); // Populate sortedGrammar from internal representation

    QPropertyAnimation* m_shakeAnimation =
        nullptr; // For interrupting userResponse animation if they spam enter
                 // key

    TutorialManager* tm = nullptr;

    QRegularExpression re{"^\\s+|\\s+$"};
};

#endif // LLTUTORWINDOW_H
```

7.31 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

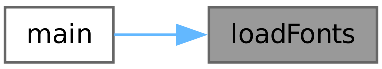
- void loadFonts ()
- int main (int argc, char *argv[])

7.31.1 Function Documentation

7.31.1.1 loadFonts()

void loadFonts ()

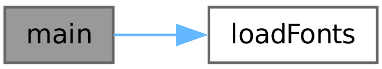
Here is the caller graph for this function:



7.31.1.2 main()

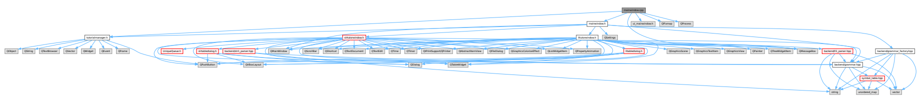
```
int main (
        int argc,
        char ∗ argv[])
```

Here is the call graph for this function:



7.32 mainwindow.cpp File Reference

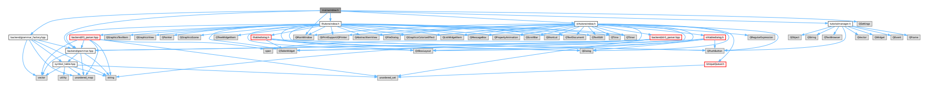
#include "mainwindow.h"
#include "tutorialmanager.h"
#include "ui_mainwindow.h"
#include <QMessageBox>
#include <QPixmap>
#include <QProcess>
Include dependency graph for mainwindow.cpp:



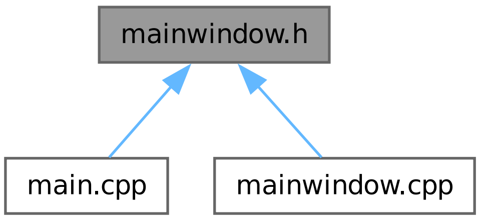
7.33 mainwindow.h File Reference

#include "backend/grammar.hpp"
#include "backend/grammar_factory.hpp"
#include "lltutorwindow.h"
#include "slrtutorwindow.h"
#include "tutorialmanager.h"

#include <QMainWindow>
#include <QSettings>
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.

7.34 mainwindow.h

Go to the documentation of this file.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "backend/grammar.hpp"
#include "backend/grammar_factory.hpp"
#include "lltutorwindow.h"
#include "slrtutorwindow.h"
#include "tutorialmanager.h"
#include <QMainWindow>
#include <QSettings>

static const QVector<QString> levelColors = {
    "#2C3E50", // 1: Navy oscuro
    "#2980B9", // 2: Azul brillante
    "#16A085", // 3: Teal
    "#27AE60", // 4: Verde esmeralda
    "#8E44AD", // 5: Púrpura medio
    "#9B59B6", // 6: Púrpura claro
    "#E67E22", // 7: Naranja
    "#D35400", // 8: Naranja oscuro
    "#CD7F32", // 9: Bronce
    "#FFD700"  // 10: Oro puro
};

QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow {
    Q_OBJECT
    Q_PROPERTY(unsigned userLevel READ userLevel WRITE setUserLevel NOTIFY
                   userLevelChanged)

```

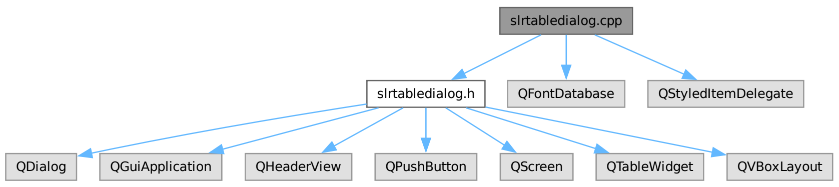
```
 public:
   MainWindow(QWidget* parent = nullptr);

   ~MainWindow();

   unsigned thresholdFor(unsigned level) { return BASE_THRESHOLD * level; }

   unsigned userLevel() const { return m_userLevel; };

   void setUserLevel(unsigned lvl) {
       unsigned clamped = qMin(lvl, MAX_LEVEL);
       if (m_userLevel == clamped)
           return;
       m_userLevel = clamped;
       emit userLevelChanged(clamped);
   }

 private slots:
   void on_lv1Button_clicked(bool checked);
   void on_lv2Button_clicked(bool checked);
   void on_lv3Button_clicked(bool checked);

   void on_pushButton_clicked();

   void on_pushButton_2_clicked();

   void on_tutorial_clicked();

   void on_actionSobre_la_aplicaci_n_triggered();

   void on_actionReferencia_LL_1_triggered();

   void on_actionReferencia_SLR_1_triggered();

   void on_idiom_clicked();

 signals:
   void userLevelChanged(unsigned lvl);

   void userLevelUp(unsigned newLevel);

 private:
   void setupTutorial();

   void restartTutorial();

   void handleTutorFinished(int cntRight, int cntWrong);

   void saveSettings();

   void loadSettings();

   Ui::MainWindow*  ui;
   GrammarFactory   factory;
   int              level = 1;
   TutorialManager* tm    = nullptr;

   static constexpr unsigned MAX_LEVEL = 10;
   static constexpr unsigned MAX_SCORE = 999;

   unsigned  m_userLevel = 1;
   unsigned  userScore   = 0;
   QSettings settings;

   const unsigned BASE_THRESHOLD = 10;
};
#endif // MAINWINDOW_H
```

7.35 README.md File Reference

7.36 slrtabledialog.cpp File Reference

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

Include dependency graph for slrtabledialog.cpp:

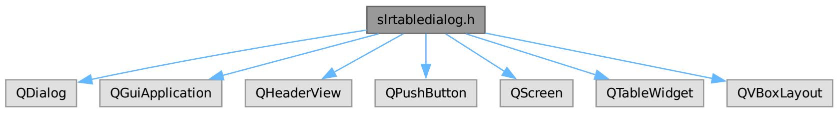


Classes

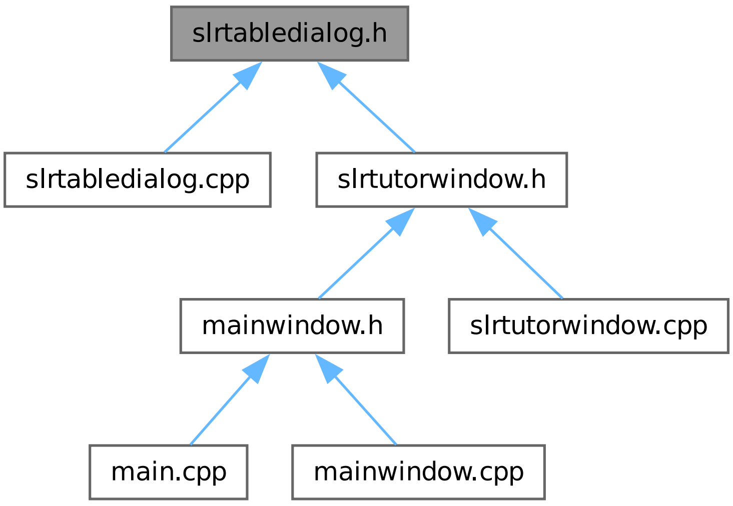
- class CenterAlignDelegate

7.37 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.

7.38 slrtabledialog.h

Go to the documentation of this file.

```
#ifndef SLRTABLEDIALOG_H
#define SLRTABLEDIALOG_H

#include <QDialog>
#include <QGuiApplication>
#include <QHeaderView>
#include <QPushButton>
#include <QScreen>
#include <QTableWidget>
#include <QVBoxLayout>

class SLRTableDialog : public QDialog {
    Q_OBJECT
  public:
    SLRTableDialog(int rowCount, int colCount, const QStringList& colHeaders,
                   QWidget*                   parent      = nullptr,
                   QVector<QVector<QString>>* initialData = nullptr);

    QVector<QVector<QString>> getTableData() const;

    void setInitialData(const QVector<QVector<QString>>& data);

  private:
    QTableWidget* table;
    QPushButton*  submitButton;
};

#endif // SLRTABLEDIALOG_H
```

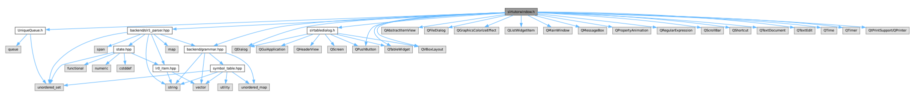
7.39 slrtutorwindow.cpp File Reference

#include ”slrtutorwindow.h”
#include ”tutorialmanager.h”

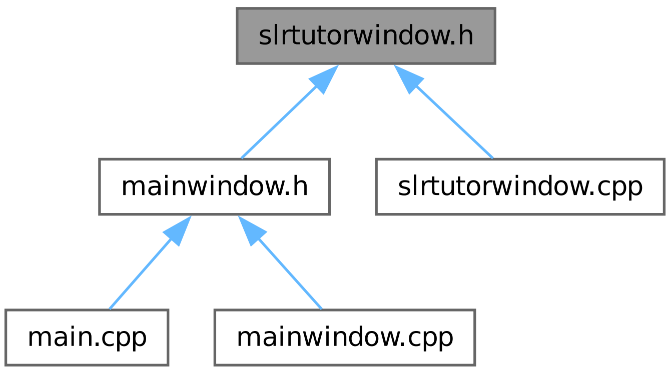
#include "ui_slrtutorwindow.h"
#include <QEasingCurve>
#include <QFontDatabase>
#include <sstream>
#include "slrwizard.h"
Include dependency graph for slrtutorwindow.cpp:

7.40 slrtutorwindow.h File Reference

#include "UniqueQueue.h"
#include "backend/grammar.hpp"
#include "backend/slr1_parser.hpp"
#include "slrtabledialog.h"
#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 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 }

7.40.1 Enumeration Type Documentation

7.40.1.1 StateSlr

enum class StateSlr [strong]

Enumerator

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

Enumerator

D2	
D_prime	
E	
E1	
E2	
F	
FA	
G	
H	
H_prime	
fin	

7.41 slrtutorwindow.h

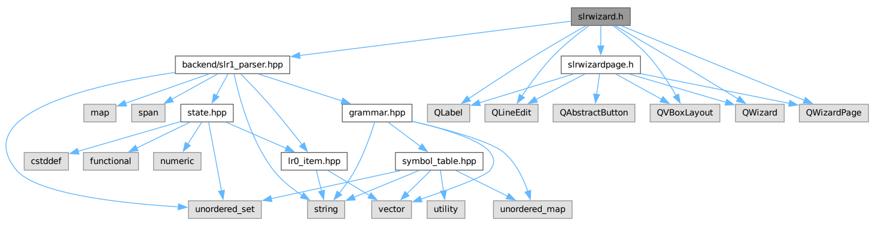
Go to the documentation of this file.

```
#ifndef SLRTUTORWINDOW_H
#define SLRTUTORWINDOW_H

#include "UniqueQueue.h"
#include "backend/grammar.hpp"
#include "backend/slr1_parser.hpp"
#include "slrtabledialog.h"
#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>

namespace Ui {
class SLRTutorWindow;
}

// ====== SLR(1) Tutor States =======================================
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
};
```

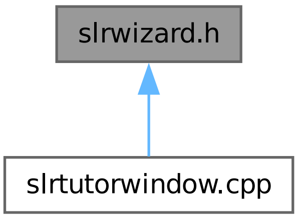
```

class TutorialManager;

// ====== Main Tutor Class for SLR(1) ===============================
class SLRTutorWindow : public QMainWindow {
    Q_OBJECT

  public:
    // ====== Constructor / Destructor ===============================
    explicit SLRTutorWindow(const Grammar& g, TutorialManager* tm = nullptr,
                            QWidget* parent = nullptr);
    ~SLRTutorWindow();

    // ====== Core Flow Control ========================================
    QString generateQuestion();

    void updateState(bool isCorrect);
    QString
    FormatGrammar(const Grammar& grammar);
    void fillSortedGrammar();

    // ====== UI Interaction ===========================================
    void addMessage(const QString& text, bool isUser);
    void exportConversationToPdf(
        const QString& filePath);
    void showTable();
    void launchSLRWizard();
    void updateProgressPanel();
    void addUserState(unsigned id);
    void addUserTransition(unsigned fromId, const std::string& symbol,
                           unsigned toId); // Register a user-created transition

    // ====== Visual Feedback & Animations ============================
    void animateLabelPop(QLabel* label);
    void animateLabelColor(QLabel* label, const QColor& flashColor);
    void wrongAnimation();             // Label animation for incorrect answer
    void wrongUserResponseAnimation(); // Message widget animation for incorrect
                                       // answer
    void markLastUserIncorrect();

    // ====== Response Verification ==================================
    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();

    // ====== Correct Solutions (Auto-generated) =====================
    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();

    // ====== Pedagogical Feedback ===================================
    QString feedback(); // Delegates to appropriate feedback based on state
    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);
  private slots:
    void on_confirmButton_clicked();
    void on_userResponse_textChanged();

  signals:
    void sessionFinished(int cntRight, int cntWrong);

  protected:
    void closeEvent(QCloseEvent* event) override {
        emit sessionFinished(cntRightAnswers, cntWrongAnswers);
        QWidget::closeEvent(event);
    }

  private:
    // ====== Helper Functions ========================================
    std::vector<std::string> qvectorToStdVector(const QVector<QString>& qvec);
    QVector<QString> stdVectorToQVector(const std::vector<std::string>& vec);
    QSet<QString>
    stdUnorderedSetToQSet(const std::unordered_set<std::string>& uset);
    std::unordered_set<std::string>
    qsetToStdUnorderedSet(const QSet<QString>& qset);
    std::unordered_set<Lr0Item> ingestUserItems(const QString& userResponse);
    std::vector<std::pair<std::string, std::vector<std::string»>
         ingestUserRules(const QString& userResponse);
    void setupTutorial();
    // ====== Core Components ===========================================
    Ui::SLRTutorWindow* ui;
    Grammar             grammar;
    SLR1Parser          slr1;

    // ====== State and Grammar Tracking ===============================
    StateSlr                              currentState;
    QVector<QString>                              sortedNonTerminals;
    QVector<QPair<QString, QVector<QString»> sortedGrammar;
    QString                               formattedGrammar;

    unsigned cntRightAnswers = 0;
    unsigned cntWrongAnswers = 0;

    // ====== State Machine Runtime Variables =========================
    std::unordered_set<state> userMadeStates; // All states the user has created
    std::unordered_map<unsigned, std::unordered_map<std::string, unsigned»
        userMadeTransitions; // Transitions made by the user
    UniqueQueue<unsigned>
             statesIdQueue; // States to be processed in B-C-CA-CB loop
    unsigned currentStateId = 0;
    state    currentSlrState;

    QStringList  followSymbols; // Used in CA-CB loop
    qsizetype    currentFollowSymbolsIdx = 0;
    unsigned int nextStateId             = 0;

    QVector<const state*> statesWithLr0Conflict; // Populated in F
    std::queue<unsigned>  conflictStatesIdQueue;
    unsigned              currentConflictStateId = 0;
    state                 currentConflictState;

    std::queue<unsigned>
             reduceStatesIdQueue; // States without conflicts but with reduce
    unsigned currentReduceStateId = 0;
    state    currentReduceState;

    struct ActionEntry {
        enum Type { Shift, Reduce, Accept, Goto } type;
        int              target;
        static ActionEntry makeShift(int s) { return {Shift, s}; }
        static ActionEntry makeReduce(int r) { return {Reduce, r}; }
        static ActionEntry makeAccept() { return {Accept, 0}; }
        static ActionEntry makeGoto(int g) { return {Goto, g}; }
    };

    QMap<int, QMap<QString, ActionEntry» slrtable;
    QVector<QVector<QString»             rawTable;

    // ====== Conversation Log ========================================
    struct MessageLog {
        QString message;
        bool    isUser;
        bool    isCorrect = true;

        MessageLog(const QString& message, bool isUser)
            : message(message), isUser(isUser) {}

        void toggleIsCorrect() { isCorrect = false; }
    };

    QVector<MessageLog> conversationLog;
    QWidget*            lastUserMessage        = nullptr;
    qsizetype           lastUserMessageLogIdx = -1;

    QPropertyAnimation* m_shakeAnimation =
        nullptr; // For interrupting userResponse animation if they spam enter
                 // key

    TutorialManager* tm;

    QRegularExpression re{"^\\s+|\\s+$"};
};

#endif // SLRTUTORWINDOW_H
```

7.42 slrwizard.h File Reference

#include "backend/slr1_parser.hpp"
#include "slrwizardpage.h"
#include <QLabel>
#include <QLineEdit>
#include <QVBoxLayout>
#include <QWizard>
#include <QWizardPage>
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.

7.43 slrwizard.h

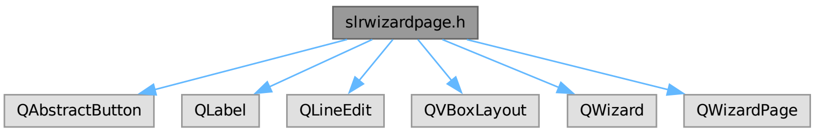
Go to the documentation of this file.

```
#ifndef SLRWIZARD_H
#define SLRWIZARD_H

#include ”backend/slr1_parser.hpp”
#include ”slrwizardpage.h”
#include <QLabel>
#include <QLineEdit>
#include <QVBoxLayout>
#include <QWizard>
#include <QWizardPage>

class SLRWizard : public QWizard {
    Q_OBJECT
  public:
    SLRWizard(SLR1Parser& parser, const QVector<QVector<QString»& rawTable,
              const QStringList&                              colHeaders,
              const QVector<QPair<QString, QVector<QString»>& sortedGrammar,
              QWidget*                                      parent = nullptr)
        : QWizard(parent) {
        setWindowTitle(tr(”Ayuda interactiva: Tabla SLR(1)”));

        const int nTerm =
            parser.gr_.st_.terminals_.contains(parser.gr_.st_.EPSILON_)
                ? parser.gr_.st_.terminals_.size() - 1
                : parser.gr_.st_.terminals_.size();
        SLRWizardPage* last = nullptr;
        // Generar explicación y páginas
        int rows = rawTable.size();
        int cols = colHeaders.size();
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                QString sym = colHeaders[j];
                QString expected;
                QString explanation;
                if (j < nTerm) {
                    auto itAct = parser.actions_.at(i).find(sym.toStdString());
                    SLR1Parser::s_action act =
                        (itAct != parser.actions_.at(i).end()
                             ? itAct->second
                             : SLR1Parser::s_action{nullptr,
                                                    SLR1Parser::Action::Empty});
                    switch (act.action) {
                    case SLR1Parser::Action::Shift: {
                        unsigned to =
                            parser.transitions_.at(i).at(sym.toStdString());
                        expected    = QString(”s%1”).arg(to);
                        explanation = tr(”Estado %1: existe transición  (%1, ”
                                         ”'%2'). ¿A qué ”
                                         ”estado harías shift?”)
```

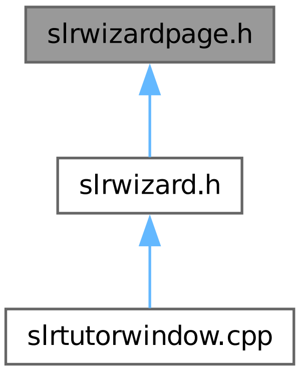
```
                                          .arg(i)
                                          .arg(sym);
                        break;
                    }
                    case SLR1Parser::Action::Reduce: {
                        int idx = -1;
                        for (int k = 0; k < sortedGrammar.size(); ++k) {
                            auto& rule = sortedGrammar[k];
                            if (rule.first.toStdString() ==
                                    act.item->antecedent_ &&
                                stdVectorToQVector(act.item->consequent_) ==
                                    rule.second) {
                                idx = k;
                                break;
                            }
                        }
                        expected = QString("r%1").arg(idx);
                        // explicación con FOLLOW
                        std::unordered_set<std::string> F;
                        F = parser.Follow(act.item->antecedent_);
                        QStringList followList;
                        for (auto& t : F)
                            followList << QString::fromStdString(t);
                        explanation = tr("Estado %1: contiene el ítem [%2 → "
                                         "... · ] y '%3'  "
                                         "SIG(%2). ¿Qué regla usas para "
                                         "reducir (0, 1, ...)?")
                                          .arg(i)
                                          .arg(QString::fromStdString(
                                              act.item->antecedent_))
                                          .arg(colHeaders[j]);
                        break;
                    }
                    case SLR1Parser::Action::Accept:
                        expected    = "acc";
                        explanation = tr("Estado %1: contiene [S → A  ·  $]. "
                                         "¿Qué palabra clave "
                                         "usas para aceptar?")
                                          .arg(i);
                        break;
                    case SLR1Parser::Action::Empty:
                    default:
                        continue;
                    }
                } else {
                    // GOTO sobre no terminal
                    auto nonT = sym.toStdString();
                    if (!parser.transitions_.contains(i)) {
                        continue;
                    }
                    auto itGo = parser.transitions_.at(i).find(nonT);
                    if (itGo != parser.transitions_.at(i).end()) {
                        expected    = QString::number(itGo->second);
                        explanation = tr("Estado %1:  (%1, '%2') existe. ¿A "
                                         "qué estado va "
                                         "la transición? (pon solo el número)")
                                          .arg(i)
                                          .arg(sym);
                    } else {
                        continue;
                    }
                }

                SLRWizardPage* page =
                    new SLRWizardPage(i, sym, explanation, expected, this);
                last = page;
                addPage(page);
            }
        }
        if (last) {
            last->setFinalPage(true);
        }
    }

    QVector<QString> stdVectorToQVector(const std::vector<std::string>& vec) {
        QVector<QString> result;
        result.reserve(vec.size());
        for (const auto& str : vec) {
            result.push_back(QString::fromStdString(str));
        }
        return result;
    }
};

#endif // SLRWIZARD_H
```

7.44 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.

7.45 slrwizardpage.h

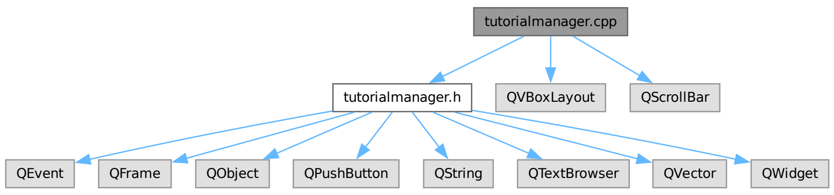
Go to the documentation of this file.

```
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
00023 class SLRWizardPage : public QWizardPage {
00024     Q_OBJECT
```

```
 public:
  SLRWizardPage(int state, const QString& symbol, const QString& explanation,
         const QString& expected, QWidget* parent = nullptr)
    : QWizardPage(parent), m_state(state), m_symbol(symbol),
     m_expected(expected) {
    setTitle(tr("Estado %1, símbolo '%2'").arg(state).arg(symbol));

    QLabel* lbl = new QLabel(explanation, this);
    lbl->setWordWrap(true);

    m_edit = new QLineEdit(this);
    m_edit->setPlaceholderText(
      tr("Escribe tu respuesta (p.ej. s3, r2, acc, 5)"));

    QVBoxLayout* layout = new QVBoxLayout(this);
    layout->addWidget(lbl);
    layout->addWidget(m_edit);
    setLayout(layout);

    connect(m_edit, &QLineEdit::textChanged, this,
        &SLRWizardPage::onTextChanged);
  }
 private slots:
  void onTextChanged(const QString& text) {
    bool correct = (text.trimmed() == m_expected);
    setComplete(correct);
    if (correct) {
      setSubTitle(
        tr("  Respuesta correcta, pasa a la siguiente pregunta"));
    } else {
      setSubTitle(tr("  Incorrecto, revisa el enunciado. Consulta los "
             "estados que has construido."));
    }
    wizard()->button(QWizard::NextButton)->setEnabled(correct);
  }

 private:
  void setComplete(bool complete) {
    m_isComplete = complete;
    emit completeChanged();
  }

  bool isComplete() const override { return m_isComplete; }

  int     m_state;
  QString   m_symbol;
  QString   m_expected;
  QLineEdit* m_edit;
  bool     m_isComplete =
    false;
};

#endif // SLRWIZARDPAGE_H
```

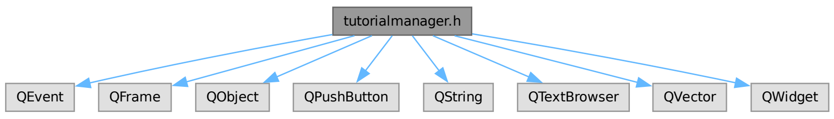
7.46 tutorialmanager.cpp File Reference

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

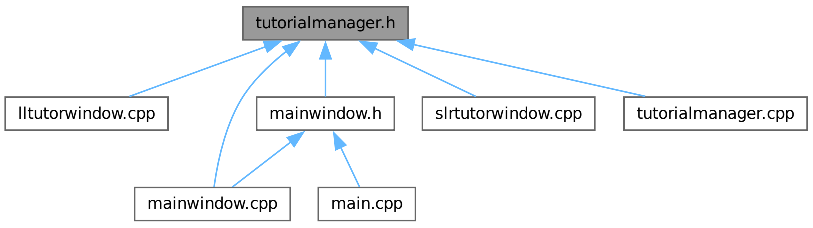


7.47 tutorialmanager.h File Reference

#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.

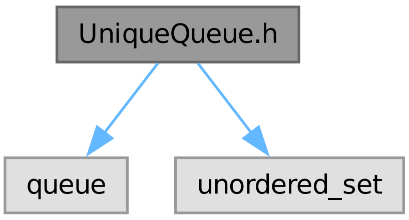
7.48 tutorialmanager.h

Go to the documentation of this file.

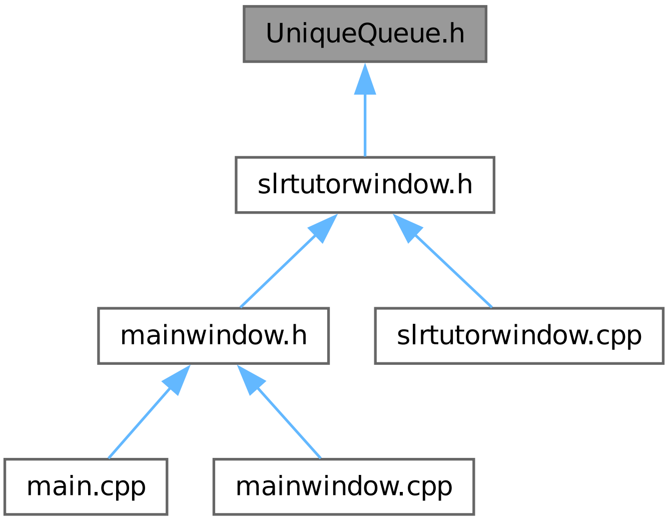
```
#ifndef TUTORIALMANAGER_H
#define TUTORIALMANAGER_H

#include <QEvent>
#include <QFrame>
#include <QObject>
#include <QPushButton>
#include <QString>
#include <QTextBrowser>
#include <QVector>
#include <QWidget>

struct TutorialStep {
    QWidget* target;
    QString  htmlText;
};

class TutorialManager : public QObject {
    Q_OBJECT
public:
    TutorialManager(QWidget* rootWindow);

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

    void start();

    void setRootWindow(QWidget* newRoot);

    void clearSteps();

    void hideOverlay();

    void finishLL1();

    void finishSLR1();

protected:
    bool eventFilter(QObject* obj, QEvent* ev) override;

signals:
    void stepStarted(int index);

    void tutorialFinished();

    void ll1Finished();

    void slr1Finished();

public slots:
    void nextStep();

private:
    void showOverlay();

    void repositionOverlay();

    QWidget*              m_root;
    QVector<TutorialStep> m_steps;
    int                   m_index = -1;

    QWidget* m_overlay = nullptr;
    QFrame*  m_highlight =
        nullptr;
    QTextBrowser* m_textBox = nullptr;
    QPushButton*  m_nextBtn = nullptr;
};

#endif // TUTORIALMANAGER_H
```

7.49 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.

7.50 UniqueQueue.h

Go to the documentation of this file.

```
#ifndef UNIQUEQUEUE_H
#define UNIQUEQUEUE_H
#include <queue>
#include <unordered_set>

template <typename T> class UniqueQueue {
  public:
    void push(const T& value) {
        if (seen_.insert(value).second) {
            queue_.push(value);
        }
    }

    void pop() {
        if (!queue_.empty()) {
            queue_.pop();
        }
    }

    const T& front() const { return queue_.front(); }

    bool empty() const { return queue_.empty(); }

    void clear() {
        while (!queue_.empty())
            queue_.pop();
        seen_.clear();
    }

  private:
    std::queue<T>          queue_;
    std::unordered_set<T> seen_;
};
#endif // UNIQUEQUEUE_H
```