LR0Item

```
//
// Created by popca on 30/11/2022.
//

#pragma once
#include <deque>
#include <string>
#include <list>
#include <iostream>
#include <boost/functional/hash.hpp>

class LR0Item {
public:
    // string representing the left side of the item and the start at the same time
    std::string start;

    /*
     * deque representing the left side of the right hand side of the item, thus the part
before the dot
     * we are using a deque so that popping from the end is theta(1) when shifting
     */
    std::deque<std::string> lhs;

    /*
     * deque representing the right side of the right hand side of the item, thus the
part after the dot
     * we are using a deque so that adding to the beginning is theta(1) when shifting
     */
    std::deque<std::string> rhs;

    LR0Item(const std::string& start,
            const std::list<std::string>& lhs,
            const std::list<std::string>& rhs);

    LR0Item(const std::string& start,
            const std::deque<std::string>& lhs,
            const std::deque<std::string>& rhs);

    /*
     * we create a new LR0Item by copying the current lhs and rhs, but
     * we move the dot once to the right (pop the last element from lhs and add it to the
beginning of the rhs)
     */
    LR0Item shift_dot_right() const;

    bool operator==(const LR0Item& other) const;
    friend struct std::hash<LR0Item>;
    friend std::ostream &operator<<(std::ostream &os, const LR0Item& object);
};
```

```cpp
/*
 * implement hash for a LR0Item such that we can use unordered sets with LR0Items
 * order is important in this case, so we can use hash combine from the boost library
 * for hashing, we take each value from the lhs and rhs
 */
template<>
struct std::hash<LR0Item>
{
    std::size_t operator()(LR0Item const& item) const noexcept {
        std::size_t seed = 0;
        boost::hash_combine(seed, item.start);
        for (const std::string& lhs_item: item.lhs) {
            boost::hash_combine(seed, lhs_item);
        }
        for (const std::string& rhs_item: item.rhs) {
            boost::hash_combine(seed, rhs_item);
        }
        return seed;
    }
};
```

State

```
//
// Created by popca on 30/11/2022.
//

#pragma once
#include <unordered_set>
#include "LR0Item.h"

class State {
public:
    /*
     * a State consists of multiple, different items, so we will use an unordered set to
store them
     * thus adding a LR0Item and checking if a LR0Item exists in a State will be theta(1)
     */
     std::unordered_set<LR0Item> items;
    bool operator==(const State& other) const;
    State(const std::unordered_set<LR0Item>& items);
    State();

    // check whether there are no items in the set of LR0Items
    bool empty() const;
    friend std::ostream &operator<<(std::ostream &os, const State& object);
};

/*
 * implement hash for a State such that we can use unordered sets with State
 * order is NOT important in this case, so we will add all the hash values of the
LR0Items
 * (we are required to use a commutative operator, and boost combine is not one of them)
 */
template<>
struct std::hash<State>
{
    std::size_t operator()(State const& state) const noexcept {
        std::size_t seed = 0;
        std::hash<LR0Item> item_hash{};
        for (const LR0Item& item: state.items) {
            seed += item_hash.operator()(item);
        }
        return seed;
    }
};
```

ParsingNode

```
//
// Created by popca on 14/12/2022.
//

#pragma once
#include <string>
#include <iostream>

/*
 * represents an entry in the SyntaxTree
 * contains the following:
 *          index, int representing its position in the table
 *          value, string representing the symbol it was expanded to
 *          parent_index, int representing the index of the parent or -1 if it is a root
node
 *          left_sibling_index, int representing the index of its left sibling or -1 if
it has no left siblings
 */
class ParsingNode {
public:
    ParsingNode(int index, std::string value, int parent_index, int left_sibling_index);
    bool operator==(const ParsingNode& other) const;
    friend std::ostream &operator<<(std::ostream &os, const ParsingNode& object);

public:
    int index;
    std::string value;
    int parent_index;
    int left_sibling_index;

};
```

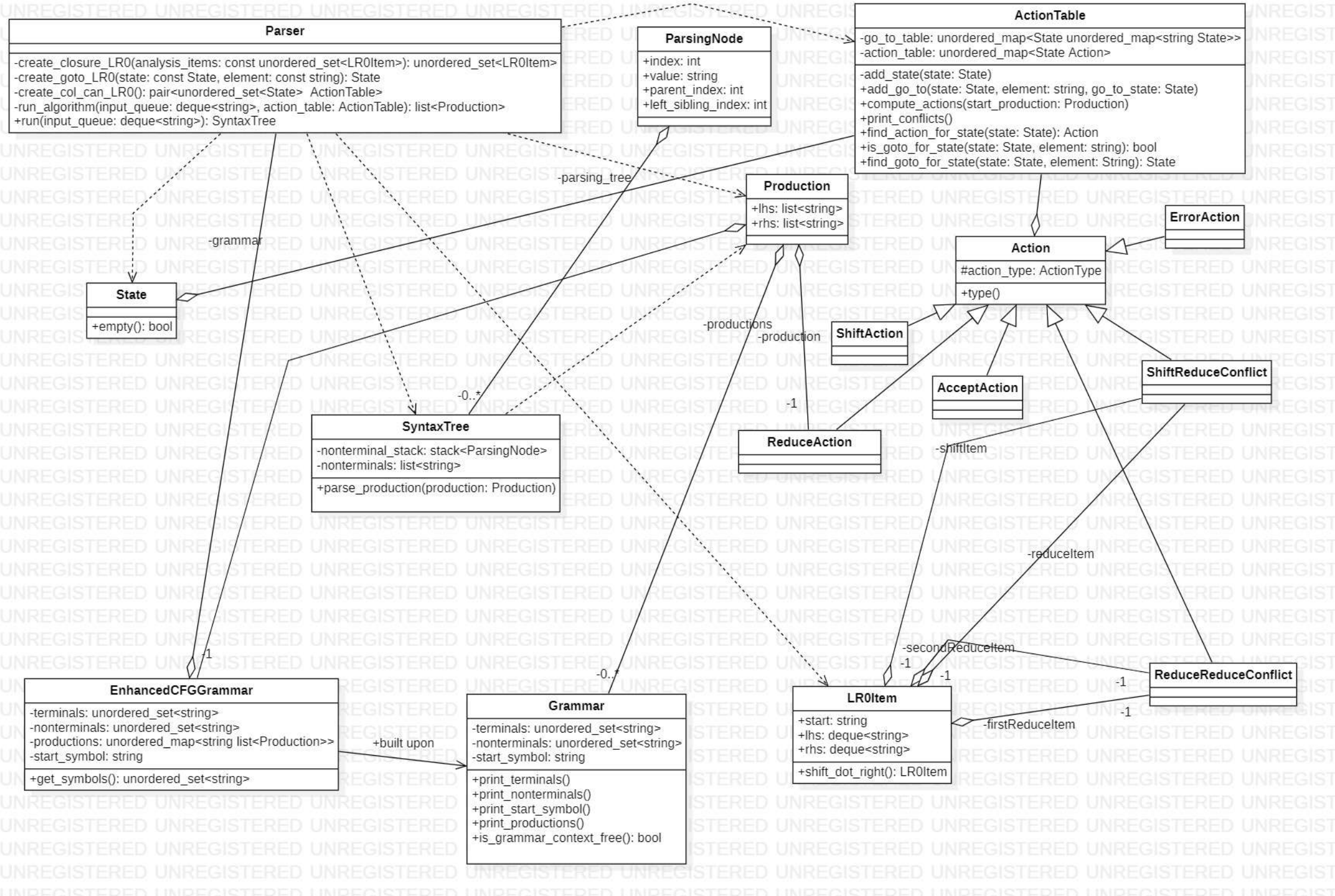
SyntaxTree

```
//
// Created by popca on 14/12/2022.
//

#pragma once

#include "../../model/header/ParsingNode.h"
#include "../../model/header/Production.h"
#include <list>
#include <stack>

class SyntaxTree {
private:
    // parsing tree should be a list, as adding to the end should be theta(1)
    std::list<ParsingNode> parsing_tree;

    /*
     * we use an additional nonterminal stack in order to make finding the right-most
     * nonterminal theta(1)
     * when we transform a production into multiple symbols, we add the nonterminals to
this stack in
     * the order we find them
     */
    std::stack<ParsingNode> nonterminal_stack;

    // we need the set of nonterminals so that checking whether a symbol is terminal or
nonterminal is theta(1)
    std::unordered_set<std::string> nonterminals;

public:
    SyntaxTree(const std::string& start_point, std::unordered_set<std::string>
nonterminals);

    // given a production, we parse it by expanding it
    void parse_production(const Production& production);

    friend std::ostream &operator<<(std::ostream &os, const SyntaxTree& object);
};
```

Parser

```
//
// Created by popca on 30/11/2022.
//

#pragma once
#include <unordered_set>
#include "../../model/header/LR0Item.h"
#include "../../grammar/header/Grammar.h"
#include "../../grammar/header/EnhancedCFGGrammar.h"
#include "../../model/header/State.h"
#include "../../data_structure/header/ActionTable.h"
#include "../../syntax_tree/header/SyntaxTree.h"
#include <queue>

class Parser {
private:
    // each parser requires a CFG grammar which needs to be enhanced
    EnhancedCFGGrammar grammar;

    /*
     * a closure is an unordered set of LR0Items, that is a closure should not contain
     * two identical LR0Items
     * the analysis items, similarly, is an unordered set of LR0Items
     */
    std::unordered_set<LR0Item> create_closure_LR0(const std::unordered_set<LR0Item>&
analysis_items);

    // given a state and a symbol, we find the state where it goes to
    State create_goto_LR0(const State& state, const std::string& element);

    /*
     * we create the canonical collection, that is an unordered set of States
     * as it should not contain two identical States
     * while creating the canonical collection, we construct the goto table for
efficiency
     */
    std::pair<std::unordered_set<State>, ActionTable> create_col_can_LR0();

    /*
     * we run the algorithm using an input as deque
     * result should be a list because we need adding to the end to be theta(1)
     * input should be a deque because we always take the first character to process it
     * and sometimes we need to add something in front of the queue
     */
    std::list<Production> run_algorithm(std::deque<std::string>& input_queue,
ActionTable& action_table);

public:
    friend class TestParser;
    Parser(EnhancedCFGGrammar grammar);
```

```
    /*
     * given the input, and having an enhanced grammar, we create the action table,
     * then run the algorithm and transform it into a SyntaxTree
     */
    SyntaxTree run(std::deque<std::string>& input_queue);
};
```


Parser
-create_closure_LR0(analysis_items: const unordered_set<LR0Item>): unordered_set<LR0Item>
-create_goto_LR0(state: const State, element: const string): State
-create_col_can_LR0(): pair<unordered_set<State> ActionTable>
-run_algorithm(input_queue: deque<string>, action_table: ActionTable): list<Production>
+run(input_queue: deque<string>): SyntaxTree
ParsingNode
+index: int
+value: string
+parent_index: int
+left_sibling_index: int
ActionTable
-go_to_table: unordered_map<State unordered_map<string State>>
-action_table: unordered_map<State Action>
-add_state(state: State)
+add_go_to(state: State, element: string, go_to_state: State)
+compute_actions(start_production: Production)
+print_conflicts()
+find_action_for_state(state: State): Action
+is_goto_for_state(state: State, element: string): bool
+find_goto_for_state(state: State, element: String): State
-parsing_tree
Production
+lhs: list<string>
+rhs: list<string>
ErrorAction
Action
#action_type: ActionType
+type()
-grammar
State
+empty(): bool
-productions
-production
ShiftAction
AcceptAction
ShiftReduceConflict
-0..*
-1
SyntaxTree
-nonterminal_stack: stack<ParsingNode>
-nonterminals: list<string>
+parse_production(production: Production)
ReduceAction
-shiftItem
-reduceItem
-secondReduceItem
-1
-1
-1
-1
-1
-1
-1
-0..*
EnhancedCFGGrammar
-terminals: unordered_set<string>
-nonterminals: unordered_set<string>
-productions: unordered_map<string list<Production>>
-start_symbol: string
+get_symbols(): unordered_set<string>
+built upon
Grammar
-terminals: unordered_set<string>
-nonterminals: unordered_set<string>
-start_symbol: string
+print_terminals()
+print_nonterminals()
+print_start_symbol()
+print_productions()
+is_grammar_context_free(): bool
LR0Item
+start: string
+lhs: deque<string>
+rhs: deque<string>
+shift_dot_right(): LR0Item
ReduceReduceConflict
-firstReduceItem