Übung 07

Arbeitsaufwand insgesamt: 9h

Inhaltsverzeichnis

Inhaltsverzeichnis

Teil 1 - Ausdrucksbaum

Ziel dieser Übung ist die Entwicklung eines Parsers der die Grammatik des Satzsymboles „Programm“ auswerten kann.

Lösungsidee:

Grundsätzlich geht es darum, „Mini-Programme“ mithilfe eines Parsers auszuführen. Laut Angabe sollen Ein- und Ausgaben in diesem Format möglich sein:

```
1 set (a, 1);
2 set (pi, 3.1415);
3 set (k, -(a + 3) * pi^0.5);
4 print (k);
5 print (3);
6 print (4 / (2 * (3 - 1)));
7 set (x, 15) ;
8 print (x * 2);
9 set (x, y^2 - 4);
10 set (y, 6); print (x); print (y); print (x / 2);
11 set (y, 5); print (x); print (y); print (x / 2);
```

```
1 -7.08971
2 3
3 1
4 30
5 32
6 6
7 16
8 21
9 5
10 10.5
```

Die Übung ähnelt stark der letzten, mit dem kleinen Unterschied, dass Variablen nun wieder Ausdrücke enthalten können, was die Verwendung von „SyntaxTrees“ notwendig macht. Dieser Auswertung dieser Bäume wurde bereits vollständig in der Übungseinheit vorgenommen. Der Teil, welcher noch selbstständig zu erledigen ist, ist das eigentliche Aufbauen eines SyntaxTrees aus den gegebenen arithmetischen Ausdrücken in Form von Strings.

Dieser Baum besteht aus mehreren verschiedenen Knoten-Arten:

- Value-Knoten: Enthalten genau einen Wert
- Identifier-Knoten: Enthalten einen Identifier bzw. Variable
    - Diese Variablen werden in einer Namelist (organisiertes Dictionary) gespeichert und können selbst wieder aus Ausdrücken bestehen
- Operator-Knoten: Legen fest, welche Operation zwischen den beiden Kindknoten vorliegt.

Hierfür wäre ich so vorgegangen, dass ich mir die Parse-Funktionen aus der letzten Übung als Grundlage nehme und diese erweitere. Anstatt direkt auszuwerten würde ich die Vorzeichen und Operatoren als entsprechende Knoten erzeugen und die Operanden als Kindknoten hinzufügen. Das sollte ohne weiteres möglich sein, da die Reihenfolge bereits sichergestellt ist und es sich in jedem Fall, aufgrund der Vererbung, um ähnliche Knoten handelt und diese so leicht zusammengehängt werden können.

Die einzige Schwierigkeit, die ich sehe, ist die beliebige Wiederholung von Termen und Faktoren. Hierfür ist oft ein Baum mit mehr als zwei Kindern notwendig. In diesem Fall würde ich immer einen zusätzlichen Operatorknoten auf der rechten Seite einhängen, welcher pro Ebene einen zusätzlichen Wert ermöglicht (3 aneinandergehängte Operatorknoten ermöglichen z.B. 4 Operanden).

Sobald die alte Grammatik (aus der letzten Übung) funktioniert, kann diese sehr leicht auf die neue Grammatik erweitert werden, welche auch Exponenten unterstützt.

Zusätzlich zu Berechnung der Ausdrücke soll es auch möglich sein die erstellten Bäume auszugeben. Diese Funktion print (wie es in der Angabe erwähnt wurde) musste ich auf print_tree umbenennen, da es bereits eine print Funktion gab. Diese Funktion wurde danach public in der SyntaxTree Klasse bereitgestellt. Um ***alle*** Variablen auszugeben, können alle Elemente der Namelist durchiteriert werden und an diese Funktion delegiert werden.

Um das Programm ausführen zu können muss außerdem der Sprachstandard von C++ auf 2017 oder höher gestellt werden.

Lösung:

Als C++ Projekt „Teil_1“ im Archiv
Nachfolgender Source-Code

Source-Code:

Main.cpp

```
#include <iostream>
#include "syntax_tree.h"
#include "parser.h"
#include "divide_by_zero_exception.h"
#include<fstream>

using namespace xpr;

void test_functionality()
{
	std::istringstream in("set (a,0.25+0.25); set (b, 9+(9^a*12)/6-5); print (b);"); //
returns 10

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}
void test_divbyzero()
{
	std::istringstream in("set (a,0); set (b, 10/a); print (b);"); // divide by zero
exception

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}
void test_parseexcept()
{
	std::istringstream in("set (a,0+);"); // parse exception

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}
```

```
void test_varnotfound()
{
	std::istringstream in("print (pepehands);"); // variable not found exception

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}
void test_invalidvar()
{
	std::istringstream in("set (1, b);");

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}
void test_printall()
{
	std::istringstream in("set (a, 2^2*4); set (b, a^0.5); set (x, a+b);"); // variable
not found exception

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}

	std::ofstream out("printed_trees.txt");

	pst.print_all_variables(out);

	out.close();
}
```

```
void test_cycle()
{
	// crashes. could be prevented by adding each evaluated variable (per expression)
to a list and checking if it has already been evaluated.
	std::istringstream in("set (a, b); set(b, a); print(b);"); // cycle

	ParseSyntaxTree pst(in);

	try {
		pst.parse();
	}
	catch (std::exception const& e)
	{
		std::cout << e.what() << std::endl;
	}
}

void main()
{
	//test_functionality();
	//test_divbyzero();
	//test_parseexcept();
	//test_varnotfound();
	//test_invalidvar();
	//test_printall();
	test_cycle();
}
```

parser.h

```
#ifndef PARSER_H
#define PARSER_H
#include "pfc_scanner.h"
#include "syntax_tree.h"
#include "name_list.h"

namespace xpr
{
	class ParserException final : public std::runtime_error {
	public:
		explicit ParserException(std::string const& message) :
std::runtime_error{ message } { }
	};

	class Parser
	{
	public:
		Parser()
		{

		}
		virtual ~Parser()
		{}
		virtual void parse() = 0;
	protected:
		pfc::scanner scanner;

		// Terminal beginnings
		bool is_tb_Expression() const
		{
			return is_tb_Term();
		}
		bool is_tb_Term() const
		{
			return is_tb_Factor();
		}
		bool is_tb_Factor() const
		{
			return is_tb_AddOp() || is_tb_UFactor();
		}
		bool is_tb_UFactor() const
		{
			return is_tb_Monom() || is_tb_PExpression();
		}
		bool is_tb_Monom() const
		{
			return is_tb_WMonom();
		}
		bool is_tb_WMonom() const
		{
			return is_tb_Unsigned() || is_tb_Identifier();
		}
		bool is_tb_Identifier() const
		{
			return scanner.is_identifier() && !scanner.is_keyword();
		}
```

```
		bool is_tb_Exponent() const
		{
			return scanner.is('^');
		}
		bool is_tb_AddOp() const
		{
			return scanner.is('+') || scanner.is('-');
		}
		bool is_tb_MultOp() const
		{
			return scanner.is('*') || scanner.is('/');
		}
		bool is_tb_PExpression() const
		{
			return scanner.is('(');
		}
		bool is_tb_Unsigned() const
		{
			return scanner.is_number();
		}


	};

	class ParseSyntaxTree : public Parser
	{
	public:
		typedef StNode<double>* node_t;
		typedef NameList<SyntaxTree<double>*>* name_t;
		ParseSyntaxTree(std::istream& in)
		{
			// register print and set as keywords (to distinguish between them
and variables)
			scanner.register_keyword("print");
			scanner.register_keyword("set");
			scanner.set_istream(in);

			// Create a new namelist to store our syntaxtree variables in
			name_list = new NameListMap<SyntaxTree<double>*>();
		}

		virtual ~ParseSyntaxTree()
		{
			delete name_list;
		}

		void parse() override
		{
			parse_program();
		}

		void print_all_variables(std::ostream& os) const
		{
			name_list->print_name_list(os);
		}

	private:
		name_t name_list;
```

```
// Additional terminal beginnings specific to our program
bool is_tb_Program()
{
	return is_tb_outputs() || is_tb_assignments();
}

bool is_tb_outputs()
{
	return is_tb_output();
}
bool is_tb_output()
{
	return scanner.is_keyword("print");
}

bool is_tb_assignments()
{
	return is_tb_assignment();
}
bool is_tb_assignment()
{
	return scanner.is_keyword("set");
}

void parse_program()
{
	if (!is_tb_Program())
	{
		throw ParseException("Parse exception in Rule 'Program'.");
	}

	while (is_tb_Program())
	{
		if (is_tb_outputs())
		{
			parse_outputs();
		}
		else if (is_tb_assignments())
		{
			parse_assignments();
		}
	}
}

void parse_outputs()
{
	if (!is_tb_outputs())
	{
		throw ParseException("Parse exception in rule 'Outputs'.");
	}

	parse_output();
	while (is_tb_outputs())
	{
		parse_output();
	}
}
```

```
		void parse_output()
		{
			if (!is_tb_output())
			{
				throw ParseException("Parse exception in rule 'Output'.");
			}

			scanner.next_symbol("print");
			scanner.next_symbol('(');
			node_t root = parse_Expression();
			scanner.next_symbol(')');
			scanner.next_symbol(';');

			SyntaxTree<double>* st = new SyntaxTree<double>(root);
			std::cout << st->evaluate(name_list) << std::endl;
			// std::cout << st;
			delete st;
		}

		void parse_assignments()
		{
			if (!is_tb_assignments())
			{
				throw ParseException("Parse exception in rule
'Assignments'.");
			}

			parse_assignment();
			while (is_tb_assignment())
			{
				parse_assignment();
			}
		}
		void parse_assignment()
		{
			if (!is_tb_assignment())
			{
				throw ParseException("Parse exception in rule 'Assignment'.");
			}

			scanner.next_symbol("set");
			scanner.next_symbol('(');
			std::string identifier = scanner.get_identifier();
scanner.next_symbol();
			if (identifier.empty())
			{
				throw InvalidIdentifierException();
			}
			scanner.next_symbol(',');

			node_t root = parse_Expression();
			scanner.next_symbol(')');
			scanner.next_symbol(';');

			name_list->register_variable(identifier, new
SyntaxTree<double>(root));
		}
```

```
		// parse functions


		// Sequence of additions/subtractions
		node_t parse_Expression()
		{
			if (!is_tb_Expression())
			{
				throw ParserException("Error parsing 'Expression'.");
			}

			// The root node needs to be created and its left child can be
populated
			node_t root = nullptr;
			node_t right = nullptr;

			node_t left(parse_Term());
			if (is_tb_AddOp())
			{
				double const sign{ parse_AddOp() };
				root = (sign == 1 ?
new StNodeOperator<double>(StNodeOperator<double>::Operator::ADD) :
new StNodeOperator<double>(StNodeOperator<double>::Operator::SUB));
				root->set_left(left);
				right = parse_Term();
			}

			// We can't populate the right child yet, since it is unclear whether
there are more operands/operators.
			// Thats why we save "right" for the time being.

			node_t ref = root;
			while (is_tb_AddOp())
			{
				// Everytime an AddOp appears, we create the corresponding
Operator Node...
				double const sign{ parse_AddOp() };

				node_t op_node = (sign == 1 ?
new StNodeOperator<double>(StNodeOperator<double>::Operator::ADD) :
new StNodeOperator<double>(StNodeOperator<double>::Operator::SUB));
				// ...And set it as the current reference nodes (the one who
is the farthest to the right) right child node.
				ref->set_right(op_node);

				// The reference node then becomes this operator node and its
left child can be set to the previously parsed "right".
				ref = op_node;
				ref->set_left(right);

				// Last but not least, prepare the next "right" node.
				right = parse_Term();
			}
```

```
                        // If a tree exists, append the "right" node to the bottom right
                node.
                        if (ref != nullptr)
                                ref->set_right(right);
                        // Or if the tree doesn't even exist (= only one value), left becomes
the root node.
                        else
                                root = left;


                        return root;
                }

                // Sequence of multiplications/divisions
                node_t parse_Term()
                {
                        if (!is_tb_Term())
                        {
                                throw ParserException("Error parsing 'Term'.");
                        }

                        // This function works analogous to "parse expression". if questions
arise, consult the other function.
                        node_t root = nullptr;
                        node_t right = nullptr;
                        node_t left(parse_Factor());
                        if (is_tb_MultOp())
                        {
                                // Instead of parsing the AddOp, we now have to consider
MultOp and create Operator Nodes accordingly
                                if (scanner.is('*'))
                                {
                                        scanner.next_symbol('*');
                                        root =
new StNodeOperator<double>(StNodeOperator<double>::Operator::MUL);
                                }
                                else if (scanner.is('/'))
                                {
                                        scanner.next_symbol('/');
                                        root =
new StNodeOperator<double>(StNodeOperator<double>::Operator::DIV);
                                }
                                root->set_left(left);
                                right = parse_Factor();
                        }

                        node_t ref = root;
```

```
                // As long as there are more operators we continue to add them to the
        tree
                while (is_tb_MultOp())
                {
                        node_t op_node = nullptr;
                        if (scanner.is('*'))
                        {
                                scanner.next_symbol('*');
                                op_node = new
StNodeOperator<double>(StNodeOperator<double>::Operator::MUL);
                        }
                        else if (scanner.is('/'))
                        {
                                scanner.next_symbol('/');
                                op_node = new
StNodeOperator<double>(StNodeOperator<double>::Operator::DIV);
                        }
                        ref->set_right(op_node);

                        ref = op_node;
                        ref->set_left(right);

                        right = parse_Factor();
                }

                if (ref != nullptr)
                        ref->set_right(right);
                else
                        root = left;

                return root;
        }

        // Optional sign and UFactor
        node_t parse_Factor() {
                if (!is_tb_Factor())
                {
                        throw ParserException("Error parsing 'Factor'.");
                }

                // optional sign
                double const sign{ is_tb_AddOp() ? parse_AddOp() : 1.0 };

                // Multiplication (to account for the sign) has to be done through
operator nodes.
                node_t root = new
StNodeOperator<double>(StNodeOperator<double>::Operator::MUL);
                root->set_left(new StNodeValue<double>(sign));
                // Then delegate to UFactor
                root->set_right(parse_UFactor());

                return root;
        }
```

```
		// Monom or PExpression
		node_t parse_UFactor()
		{
			if (!is_tb_UFactor())
			{
				throw ParserException("Error parsing 'UFactor'.");
			}

			node_t root = nullptr;

			// UFactor can either be a Monom or PExpression. Delegate to the
right parse function.
			if (is_tb_Monom())
			{
				root = parse_Monom();
			}
			else if (is_tb_PExpression())
			{
				root = parse_PExpression();
			}
			else
			{
				throw ParserException("Error parsing 'Factor'.");
			}

			return root;
		}

		// Number/variable followed by an optional exponent
		node_t parse_Monom() {
			if (!is_tb_Monom())
			{
				throw ParserException("Error parsing 'Monom'.");
			}

			// A Monom is a WMonom (delegation) followed by an optional exponent.
			// --> Operator node required
			node_t root = new
StNodeOperator<double>(StNodeOperator<double>::Operator::EXP);
			root->set_left(parse_WMonom());

			// e^1 = e. If there is no exponent, we jut set it to one. Otherwise
it becomes the stated value.
			node_t exp_node = { is_tb_Exponent() ? parse_Exponent() : new
StNodeValue<double>(1) };

			root->set_right(exp_node);

			return root;
		}
```

```
// exponent sign followed by sign and number
		node_t parse_Exponent() {
			if (!is_tb_Exponent())
			{
				throw ParserException("Error parsing 'Exponent'.");
			}

			node_t root = new
StNodeOperator<double>(StNodeOperator<double>::Operator::MUL);
			double exp = 1.0f;

			// An exponent expression starts with '^'. Skip the character and
parse the exponent and optional sign.
			scanner.next_symbol('^');
			double const sign{ is_tb_AddOp() ? parse_AddOp() : 1.0 };

			// Again, to apply the sign, add both values to the Operator Node.
			root->set_left(new StNodeValue<double>(sign));
			root->set_right(parse_WMonom());

			return root;
		}

		// Number or Variable
		node_t parse_WMonom() {
			if (!is_tb_WMonom())
			{
				throw ParserException("Error parsing 'WMonom'.");
			}

			node_t root = nullptr;

			// WMonom is either an identifer (variable) or an unsigned number.
Parse them accordingly and skip to the next symbol.
			if (is_tb_Identifier())
			{
				root = new
StNodeIdent<double>(scanner.current_symbol().get_identifier()); scanner.next_symbol();
			}
			else if(is_tb_Unsigned())
			{
				root = new StNodeValue<double>(scanner.get_number());
scanner.next_symbol();
			}

			return root;
		}
```

```
double parse_AddOp()
		{
			// parse AddOp remains unchanged and returns -1.0 for - and +1.0 for
+
			if (!is_tb_AddOp())
			{
				throw ParserException("Error parsing 'AddOp'.");
			}

			double sign{ 0.0 };
			if (scanner.is('+'))
			{
				sign = 1;
				scanner.next_symbol();
			}
			else if (scanner.is('-'))
			{
				sign = -1;
				scanner.next_symbol();
			}
			else
			{
				throw ParserException("Error parsing 'AddOp'.");
			}

			return sign;
		}

		// expression in parentheses
		node_t parse_PExpression()
		{
			if (!is_tb_PExpression())
			{
				throw ParserException("Error parsing 'PExpression'.");
			}

			// PExpression also remains unchanged and delegates to the altered
parse_Expression()
			scanner.next_symbol('(');
			node_t root(parse_Expression());
			scanner.next_symbol(')');

			return root;
		}
	};

}

#endif
```

Name_list.h

```
#ifndef NAME_LIST_H
#define NAME_LIST_H

#include<map>
#include<iostream>
#include<string>
#include<algorithm>

using std::map;
using std::string;
using std::ostream;

namespace xpr
{
	// forward declaration
	template<typename T>
	class SyntaxTree;

	template<typename T>
	class NameList
	{
	public:
		NameList()
		{

		}
		virtual ~NameList()
		{

		}
		// searches and if found returns the specified identifier in the namelist
		virtual T lookup_variable(string identifier) = 0;
		// adds a new variable to the name_list
		virtual void register_variable(string identifier, T value) = 0;
		// prints all of the variables in the namelist as trees
		virtual void print_name_list(ostream& os) = 0;
	};

	template<typename T>
	class NameListMap : public NameList<T>
	{
	public:
		NameListMap()
		{

		}

		~NameListMap()
		{
			// Delete all variables (syntaxtrees)
			for (auto it = name_list.begin(); it != name_list.end(); ++it)
			{
				delete it->second;
			}
		}
```

```
T lookup_variable(string identifier) override
		{
			return name_list[identifier];
		}

		void register_variable(string identifier, T value) override
		{
			name_list[identifier] = value;
		}

		void print_name_list(ostream& os) override
		{
			std::for_each(name_list.begin(), name_list.end(),
[&os](std::pair<std::string, T> dp) { dp.second->print_tree(os,dp.first); });
		}

	private:
		map<string, T> name_list;

	};
}

#endif
```

Syntax_tree.h

```
#ifndef SYNTAX_TREE_H
#define SYNTAX_TREE_H
#include "st_node.h"
#include "expr_tree_exceptions.h"
#include <iostream>

namespace xpr
{
	template<typename T>
	class SyntaxTree
	{
	public:
		SyntaxTree(StNode<T>* root) : root(root)
		{

		}

		~SyntaxTree()
		{
			if (root != nullptr)
			{
				delete root;
			}
		}

		T evaluate(NameList<SyntaxTree<T>*>* name_list)
		{
			if (root == nullptr)
			{
				throw EvaluationException();
			}
			// delegate to st_node.evaluate
			return root->evaluate(name_list);
		}

		friend std::ostream& operator<<(std::ostream& lhs, const SyntaxTree<T>*
other)
		{
			other->print(lhs);
			return lhs;
		}

		// Recursive function to print trees vertically
		void print_tree(std::ostream& os, std::string const& name) const
		{
			os << "Printing tree for variable '" + name + "':\n";
			print_tree_worker(root, 0, os);
		}
```

```
		void print_tree_worker(StNode<T>* const& node, size_t const depth,
		std::ostream& os) const
		{
			if (node == nullptr)
			{
				return;
			}

			print_tree_worker(node->get_right(), depth+1, os);
			for (size_t n(0); n < depth; n++)
			{
				os << "   ";
			}

			node->print(os); os << std::endl;
			print_tree_worker(node->get_left(), depth+1, os);
		}

	private:
		StNode<T>* root;
		// Prints the tree on one line
		void print(std::ostream& os) const
		{
			if (root == nullptr)
			{
				throw EvaluationException();
			}
			else
			{
				os << "Print tree in pre-order: ";
				root->print_pre_order(os);
				os << "\n";

				os << "Print tree in in-order: ";
				root->print_in_order(os);
				os << "\n";

				os << "Print tree in post-order: ";
				root->print_post_order(os);
				os << "\n";
			}
		}
	};
}

#endif
```

St_node.h

```
#ifndef ST_NODE_H
#define ST_NODE_H
#include<iostream>
#include "expr_tree_exceptions.h"
#include "divide_by_zero_exception.h"
#include "string"
#include <cmath>
#include "name_list.h"
#include "syntax_tree.h"
#include<vector>

namespace xpr {

	// forward declaration
	template<typename T>
	class SyntaxTree;

	template<typename T>
	class StNode
	{
	public:
		StNode() : left(nullptr), right(nullptr)
		{

		}
		virtual ~StNode()
		{
			if (left != nullptr)
				delete left;
			if (right != nullptr)
				delete right;
		}

		StNode<T>* get_left() const {
			return left;
		}
		StNode<T>* get_right() const {
			return right;
		}

		void set_left(StNode<T>* left)
		{
			this->left = left;
		}
		void set_right(StNode<T>* right)
		{
			this->right = right;
		}

		void print_in_order(std::ostream& os) const
		{
			if (left != nullptr)
			{
				left->print_in_order(os);
			}

			print(os);
```

```
                if (right != nullptr)
                {
                        right->print_in_order(os);
                }
        }

        void print_pre_order(std::ostream& os) const
        {
                print(os);
                if (left != nullptr)
                {
                        left->print_pre_order(os);
                }
                if (right != nullptr)
                {
                        right->print_pre_order(os);
                }
        }

        void print_post_order(std::ostream& os) const
        {
                if (left != nullptr)
                {
                        left->print_post_order(os);
                }
                if (right != nullptr)
                {
                        right->print_post_order(os);
                }
                print(os);
        }

        virtual T evaluate(NameList<SyntaxTree<T>*>* name_list) const = 0;
        virtual void print(std::ostream& os) const = 0;

protected:
        StNode<T>* left{ nullptr };
        StNode<T>* right{ nullptr };

};

template<typename T>
class StNodeValue : public StNode<T>
{
public:
        StNodeValue(const T& value) : value(value)
        {

        }

        virtual T evaluate(NameList<SyntaxTree<T>*>* name_list) const override
        {
                return value;
        }
```

```
	virtual void print(std::ostream& os) const override
		{
			os << value << " ";
		}

	private:
		T value;
	};

	template<typename T>
	class StNodeOperator : public StNode<T>
	{
	public:
		enum Operator { ADD = 0, SUB, MUL, DIV, EXP };
		const std::string operator_rep[5] = { "+","-","*","/","^" };

		StNodeOperator(Operator op) : op(op) {}

		virtual T evaluate(NameList<SyntaxTree<T>*>* name_list) const override
		{
			StNode<T>* l{ this->left };
			StNode<T>* r{ this->right };

			// For some reason i couldn't add this inside the switch
			if (op == DIV && r->evaluate(name_list) == 0)
			{
				throw divide_by_zero_exception();
			}

			switch (op)
			{
			case ADD:
				return l->evaluate(name_list) + r->evaluate(name_list);
				break;
			case SUB:
				return l->evaluate(name_list) - r->evaluate(name_list);
				break;
			case MUL:
				return l->evaluate(name_list) * r->evaluate(name_list);
				break;
			case DIV:
				return l->evaluate(name_list) / r->evaluate(name_list);
				break;
			case EXP:
				return std::pow(l->evaluate(name_list), r-
>evaluate(name_list));
				break;
			default:
				throw EvaluationException();
				break;
			}
		}
		virtual void print(std::ostream& os) const override
		{
			os << operator_rep[op] << " ";
		}
```

```
	private:
		Operator op;
	};

	template<typename T>
	class StNodeIdent : public StNode<T>
	{
	private:
		std::string value;
	public:
		StNodeIdent(const std::string& value) : value(value)
		{

		}

		virtual T evaluate(NameList<SyntaxTree<T>*>* name_list) const override
		{
			SyntaxTree<T>* tree = name_list->lookup_variable(value);
			if (tree != nullptr)
			{
				return tree->evaluate(name_list);
			}
			throw VariableNotFoundException("Variable: " + value + " was not
defined.");
		}

		virtual void print(std::ostream& os) const override
		{
			os << value << " ";
		}
	};


}
#endif
```

Expr_tree_exceptions.h

```
#ifndef EXPR_TREE_EXCEPTIONS_H
#define EXPR_TREE_EXCEPTIONS_H
#include<exception>

namespace xpr {

	class EvaluationException : public std::exception {
	public:
		virtual const char* what() const noexcept
		{
			return "Evaluation Exception: Cannot evaluate expression tree \n";
		}
	};

	class InvalidIdentifierException : public std::exception {
	public:
		virtual const char* what() const noexcept
		{
			return "Invalid Identifier Exception: Cannot set variable due to
missing identifier. \n";
		}
	};

	class VariableNotFoundException : public std::exception {
	private:
		std::string message;
	public:
		VariableNotFoundException(std::string message) : message(message)
		{

		}

		virtual const char* what() const noexcept
		{
			return message.c_str();
		}

	};

	class ParseException : public std::exception {
	private:
		std::string message;
	public:
		ParseException(std::string message) : message(message)
		{

		}

		virtual const char* what() const noexcept
		{
			return message.c_str();
		}

	};

}

#endif
```

Divide_by_zero_exception.h

```
#ifndef DIVIDE_BY_ZERO_EXCEPTION_H
#define DIVIDE_BY_ZERO_EXCEPTION_H
#include<exception>

class divide_by_zero_exception : public std::exception
{
public:
	virtual const char* what() const throw()
	{
		return "Divide by zero exception: Cannot divide by zero.\n";
	}
};

#endif
```

Testfälle

- Normale Benützung
- Division by Zero
- Ungültiger Ausdruck
- Nicht definierte Variable
- Ungültiger Variablenname
- Alle Variablen ausgeben
- Zyklen mit Variablen

Normale Benützung:

```
10
```

“set (a,0.25+0.25); set (b, 9+(9^a*12)/6-5); print (b);”

Division by Zero:

```
Divide by zero exception: Cannot divide by zero.
```

"set (a,0); set (b, 10/a); print (b);"

Ungültiger Ausdruck/Parserfehler:

```
Error parsing 'Term'.
```

“set (a,0+);”

Nicht definierte Variable:

```
Variable: pepehands was not defined.
```

"print (pepehands);"

Ungültiger Variablenname:

```
Invalid Identifier Exception: Cannot set variable due to missing identifier.
```

"set (1, b);"

Alle Variablen ausgeben

Zur Übersichtlichkeit wurde die Ausgabe ins File „printed_trees.txt“ im Projektordner umgeleitet.

"set (a, 2^2*4); set (b, a^0.5); set (x, a+b);"

Zyklen mit Variablen:

```
- // Crash
```

"set (a, b); set(b, a); print(b);"

Ich habe versucht die Zyklen zu verhindern, hatte aber einige Probleme mit dem Linker (vermutlich in Kombination mit Template Classes). Meine Lösungsidee wäre dazu gewesen, dass ich immer, wenn ich einen StNodeIdent evaluiere, den Namen des Identifiers in einem Vektor (static in StNodeIdent) speichere. Wenn dann innerhalb eines Programmschritts (Set/Print) mehrmals der gleiche Identifier evaluiert wird (d.h. der Identifier ist bereits im Vektor), dann deutet das auf zyklisches Verhalten hin. Dann sollte eigentlich eine Exception geworfen werden. Natürlich muss der Vektor dann auch nach jedem Set bzw. Print geleert werden.