Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ОТЧЕТ
к лабораторной работе №3
на тему

СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Выполнила: студентка гр. 253503
Тимошевич К. С.

Проверил: ассистент кафедры
информатики Гриценко Н. Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Цель данной лабораторной работы заключается в разработке синтаксического анализатора на языке *Erlang* для подмножества языка программирования *C#* [1]. В ходе синтаксического анализа результатом является построение синтаксического дерева, которое наглядно отражает структуру входной последовательности токенов, полученных на предыдущем этапе лексического анализа. Такое дерево разбора представляет собой иерархическую структуру, где внутренние вершины обозначают выполняемые операции (например, операторы присваивания, арифметические и логические операторы), а листья содержат идентификаторы, литералы и константы, что позволяет провести дальнейшую обработку и синтез вывода.

Синтаксический анализатор должен группировать токены в грамматические конструкции, определяя порядок выполнения операций без необходимости сохранять скобки в итоговом представлении, так как скобки используются лишь для установления порядка вычислений во входном потоке. Например, разбор выражения «*COST* = (*PRICE*+*TAX*)*0.98» должен приводить к построению дерева, где сначала происходит сложение идентификаторов *PRICE* и *TAX*, затем результат умножается на литерал 0.98, а полученное значение присваивается идентификатору *COST*. Такая структура дерева разбора позволяет наглядно отразить последовательность вычислений, соответствующую семантике исходного выражения.

Для реализации синтаксического анализа используется нисходящий метод разбора, реализованный посредством рекурсивного спуска (*LL*-анализатор). В этом подходе каждая функция парсинга отвечает за разбор определённого элемента, вызывая рекурсивно другие функции для обработки вложенных конструкций Исходные данные для синтаксического анализатора представляются в виде текстового файла, содержащего программу на заданном подмножестве *C#*.

В рамках лабораторной работы необходимо обеспечить корректное разбиение выражений, состоящих из литералов, операторов и круглых скобок, на грамматические фразы и их последующее представление в виде дерева.

Таким образом, цель лабораторной работы состоит в разработке синтаксического анализатора, способного на основе входного потока токенов построить корректное синтаксическое дерево, отражающее структуру исходной программы.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В ходе выполнения лабораторной работы был разработан синтаксический анализатор для подмножества языка программирования *C#*, реализованный на языке *Erlang* [2]. Синтаксический анализатор предназначен для построения синтаксического дерева, отражающего иерархическую структуру входной последовательности токенов, полученных на этапе лексического анализа.

Программа начинает свою работу с приема потока токенов, после чего с помощью набора рекурсивных функций (таких как *parse_statement*, *parse_expr, parse_term, parse_factor* и других) выполняется нисходящий разбор, реализованный методом рекурсивного спуска (*LL*-анализ) [3]. Такой подход позволяет группировать токены в грамматические конструкции, определяя последовательность операций, а также правильно обрабатывать вложенные блоки и управляющие конструкции, такие как условные операторы (*if-else*), циклы (*for, while, do-while, foreach*) и конструкции выбора (*switch*). В результате синтаксического анализа формируется синтаксическое дерево, где внутренние узлы представляют операторы и выражения (например, арифметические операции, операции присваивания и вызовы функций), а листья содержат идентификаторы, литералы и константы. Это дерево наглядно демонстрирует структуру исходной программы и порядок выполнения операций.

Для удобства тестирования и демонстрации работы синтаксического анализатора предусмотрена функция, позволяющая считывать исходный текст из файла, выполнять токенизацию, анализировать поток токенов и выводить сформированное синтаксическое дерево в виде иерархической структуры.

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

На рисунке 3.1 представлена часть дерева разбора для следующей строки кода *int COST = (PRICE + TAX) * 0.58*;.

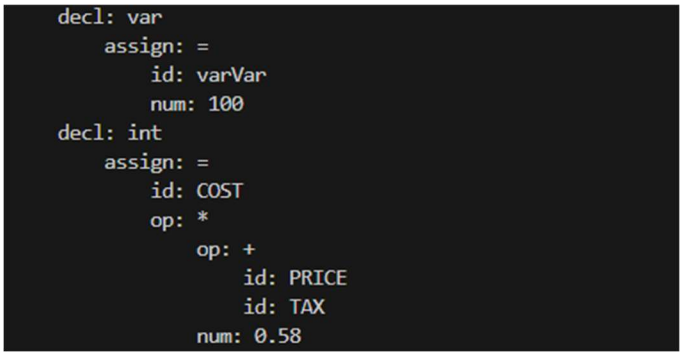

Рисунок 3.1 – Разбор арифметического выражения

На рисунке 3.2 показан вывод для объявления переменных и подключения различных библиотек и *namespace* путем *using*.

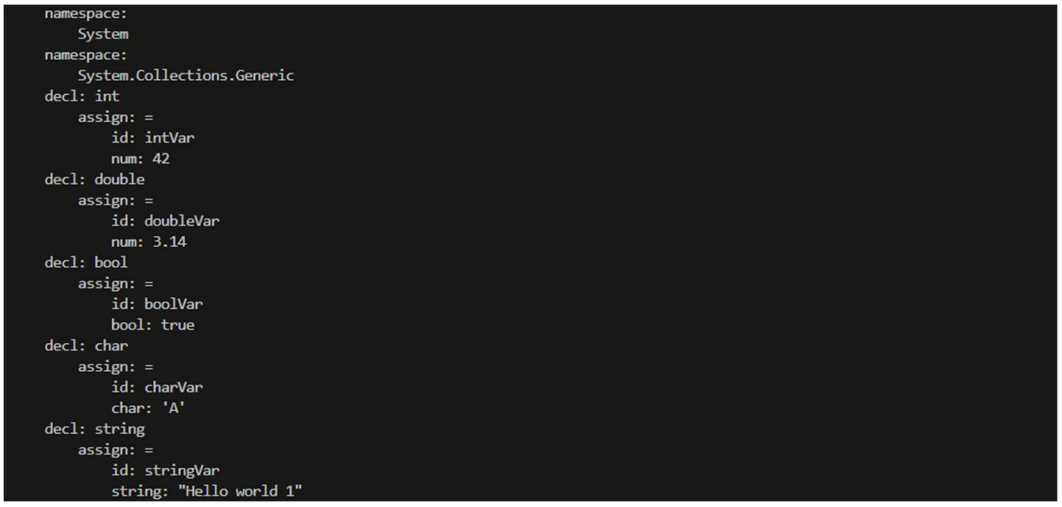

Рисунок 3.2 – Разбор объявления переменных и *namespace*

Пример обработки цикла *while* них представлен на рисунке 3.3.

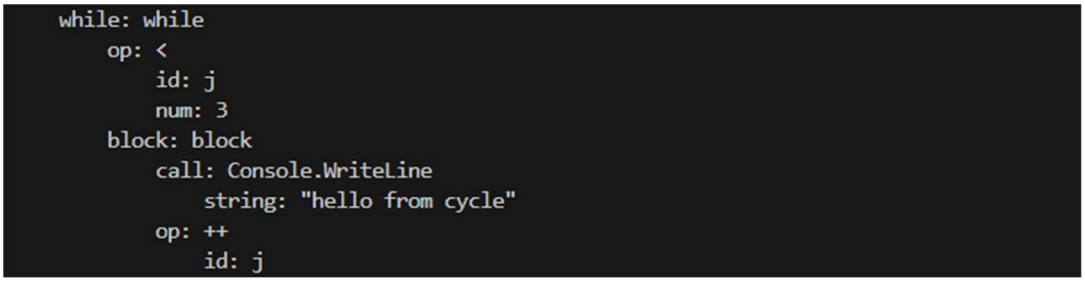

Рисунок 3.3 – Разбор цикла *while*

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы был разработан синтаксический анализатор на языке *Erlang* для подмножества языка программирования *C#*. Разработанный анализатор строит синтаксическое дерево, наглядно отражающее структурную организацию входного потока токенов. Для реализации синтаксического анализа использован метод рекурсивного спуска (*LL*-анализатор), что позволило разбить исходный текст программы на грамматические фразы и правильно определить последовательность выполняемых операций.

Синтаксический анализатор корректно обрабатывает различные элементы языка, такие как операторы присваивания, арифметические операции, управляющие конструкции (*if, for, while, do-while, foreach, switch*), вызовы функций и так далее, группируя их в единую иерархическую структуру. Реализованный алгоритм синтаксического анализа не только обеспечивает правильное построение синтаксического дерева, но и закладывает основу для дальнейших этапов обработки программ, таких как семантический анализ и следующие за ним этапы.

Таким образом, данная лабораторная работа позволила получить практический опыт разработки модульного синтаксического анализатора, демонстрирующего эффективность структурирования исходного кода, что является важным шагом на пути создания полнофункциональных компиляторов.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Erlang/OTP – documentation [Электронный ресурс]. – Режим доступа: https://www.erlang.org/. – Дата доступа: 01.03.2025.

[2] Основы C# [Электронный ресурс]. – Режим доступа: https://code-basics.com/ru/languages/csharp. – Дата доступа: 01.03.2025.

[3] Синтаксический анализ: типы анализа компилятором [Электронный ресурс]. – Режим доступа https://www.guru99.com/ru/syntax-analysis-parsing-types.html. – Дата доступа: 01.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программного кода

```
-module(parser).
-export([parse/1, print_tree/1, parse_file/1]).

node(Type, Value, Children) -> {Type, Value, Children}.

parse_statements([";" | Rest], Acc) ->
    parse_statements(Rest, Acc);  % скип лишних ;

parse_statements(Tokens, Acc) ->
    {Stmt, Rest} = parse_statement(Tokens),
    parse_statements(Rest, [Stmt | Acc]).

parse_statement(["for" | Rest]) ->
    parse_for(["for" | Rest]);
parse_statement(["do" | Rest]) ->
    parse_dowhile(["do" | Rest]);
parse_statement(["while" | Rest]) ->
    parse_while(["while" | Rest]);
parse_statement(["foreach" | Rest]) ->
    parse_foreach(["foreach" | Rest]);
parse_statement(["if" | Rest]) ->
    parse_if(["if" | Rest]);
parse_statement(["switch" | Rest]) ->
    parse_switch(["switch" | Rest]);

parse_statement([Token | Rest]) ->
    case Token of
        "using" ->
            {Namespace, [";" | Rest1]} = parse_namespace(Rest),
            {node(using, "using", [Namespace]), Rest1};
        _ ->
            case is_function_call(Token) of
                true ->
                    {Call, Rest1} = parse_function_call(Token, Rest),
                    {Call, Rest1};
                false ->
                    parse_assignment([Token | Rest])
            end
    end.

parse_namespace(Tokens) ->
    {Namespace, Rest} = parse_identifier(Tokens, []),
    {Namespace, Rest}.


parse_identifier([], Acc) -> {lists:reverse(Acc), []};
parse_identifier([";" | Rest], Acc) -> {lists:reverse(Acc), Rest};
parse_identifier([Token | Rest], Acc) ->
    parse_identifier(Rest, [Token | Acc]).

parse_for(["for", "(" | Rest]) ->
    {Init, [";" | Rest1]} = parse_statement(Rest),
    {Cond, [";" | Rest2]} = parse_expr(Rest1),
    {Incr, [")" | Rest3]} = parse_increment(Rest2),

    {Body, Rest4} = parse_block(Rest3),
    {node(for, "for", [Init, Cond, Incr, Body]), Rest4}.
```

```
parse_dowhile(["do" | Rest]) ->
    {Body, ["while", "(" | Rest1]} = parse_block(Rest),
    {Cond, [")" | Rest2]} = parse_expr(Rest1),
    {node(dowhile, "do-while", [Body, Cond]), Rest2};
parse_dowhile(_) ->
    error(invalid_do_while_syntax).
parse_while(["while", "(" | Rest]) ->
    {Cond, [")" | Rest1]} = parse_expr(Rest),
    {Body, Rest2} = parse_block(Rest1),
    {node(while, "while", [Cond, Body]), Rest2};
parse_while(_) ->
    error(invalid_while_syntax).

parse_foreach(["foreach", "(" | Rest]) ->
    {ForeachDecl, Rest1} = parse_foreach_declaration(Rest),
    case Rest1 of
        ["in" | Rest2] ->
            {Collection, [")" | Rest3]} = parse_expr(Rest2),
            {Body, Rest4} = parse_block(Rest3),
            {node(foreach, "foreach", [ForeachDecl, Collection, Body]),
Rest4};
        _ ->
            error(invalid_foreach_syntax)
    end;

parse_if(["if", "(" | Rest]) ->
    {Cond, [")" | Rest1]} = parse_expr(Rest),
    {IfBody, Rest2} = parse_block(Rest1),
    case Rest2 of
        ["else", "if", "(" | Rest3] ->
            {ElseIfCond, [")" | Rest4]} = parse_expr(Rest3),
            {ElseIfBody, Rest5} = parse_block(Rest4),
            {ElseBody, Rest6} = parse_if(["if", "(" | Rest5]),
            {node('if', "if", [
                node(condition, "condition", [Cond]),
                node(if_body, "if_body", [IfBody]),
                node(else_if, "else_if", [
                    node(condition, "condition", [ElseIfCond]),
                    node(else_if_body, "else_if_body", [ElseIfBody]),
                    ElseBody
                ])
            ]), Rest6};
        ["else" | Rest3] ->
            {ElseBody, Rest4} = parse_block(Rest3),
            {node('if', "if", [
                node(condition, "condition", [Cond]),
                node(if_body, "if_body", [IfBody]),
                node(else_body, "else_body", [ElseBody])
            ]), Rest4};
        _ ->
            {node('if', "if", [
                node(condition, "condition", [Cond]),
                node(if_body, "if_body", [IfBody])
            ]), Rest2}
    end;

parse_switch(["switch", "(", Expr, ")", "{" | Rest]) ->
    {Cases, Rest1} = parse_cases(Rest, []),
    {node(switch, "switch", [node(expr, Expr, []), node(cases, "cases",
Cases)]), Rest1};
parse_switch(_) ->
    error(invalid_switch_syntax).

parse_cases(["}" | Rest], Acc) ->
```

```
    {lists:reverse(Acc), Rest};
parse_cases(["case" | Rest], Acc) ->
    {RawValue, [":" | Rest1]} = parse_expr(Rest),
    {Body, Rest2} = parse_case_body(Rest1, []),
    parse_cases(Rest2, [node('case', RawValue, Body) | Acc]);

parse_cases(["default", ":" | Rest], Acc) ->
    {Body, Rest1} = parse_case_body(Rest, []),
    parse_cases(Rest1, [node(default, "default", Body) | Acc]);
parse_cases(["default:" | Rest], Acc) ->
    {Body, Rest1} = parse_case_body(Rest, []),
    parse_cases(Rest1, [node(default, "default", Body) | Acc]);
parse_cases(_, _) ->
    error(invalid_case_syntax).

print_tree(Node, Indent) when is_list(Node) ->
    io:format("~s~s~n", [spaces(Indent), Node]);
print_tree({Type, Value, Children}, Indent) ->
    FormattedValue =
        case Value of
            {string, S, []} -> S;
            _ -> Value
        end,
    io:format("~s~s: ~s~n", [spaces(Indent), Type, FormattedValue]),
    lists:foreach(fun(Child) -> print_tree(Child, Indent + 4) end, Children).

tokenize(Content) ->
    Tokens = tokenize(Content, [], false, []),
    [T || T <- Tokens, T =/= ""].

tokenize("do" ++ Rest, Acc, false, []) ->
    case Rest of
        [] ->
            tokenize(Rest, ["do" | Acc], false, []);
        [NextChar | _] ->
            case is_delimiter(NextChar) of
                true -> tokenize(Rest, ["do" | Acc], false, []);
                false -> tokenize("o" ++ Rest, Acc, false, "d")
            end
    end;

tokenize("while" ++ Rest, Acc, false, []) ->
    case Rest of
        [] ->
            tokenize(Rest, ["while" | Acc], false, []);
        [NextChar | _] ->
            case is_delimiter(NextChar) of
                true -> tokenize(Rest, ["while" | Acc], false, []);
                false -> tokenize("hile" ++ Rest, Acc, false, "w")
            end
    end;

tokenize("foreach" ++ Rest, Acc, false, []) ->
    case Rest of
        [] ->
            tokenize(Rest, ["foreach" | Acc], false, []);
        [NextChar | _] ->
            case is_delimiter(NextChar) of
                true -> tokenize(Rest, ["foreach" | Acc], false, []);
                false -> tokenize("oreach" ++ Rest, Acc, false, "f")
            end
    end;
```