Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ОТЧЕТ

к лабораторной работе №4

на тему

**СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР**

Выполнила: студентка гр. 253503

Тимошевич К. С.

Проверил: ассистент кафедры

информатики Гриценко Н. Ю.

Минск 2025

СОДЕРЖАНИЕ

[1 Постановка задачи](#_Toc188619154) 3

[2 Описание работы программы 4](#_Toc188619155)

[3 Ход выполнения программы 5](#_Toc188619156)

[Заключение 6](#_Toc188619157)

[Список литературных источников 7](#_Toc188619158)

[Приложение А](#_Toc188619159) [(обязательное) Листинг программного кода 8](#_Toc188619160)

1. постановка задачи

Цель данной лабораторной работы заключается в разработке семантического анализатора для подмножества языка программирования *C#* [1]. Семантический анализ является важным этапом компиляции, на котором проверяется корректность программы с точки зрения типов данных, областей видимости переменных и соответствия операций правилам языка. В отличие от синтаксического анализа, который проверяет лишь формальную правильность структуры программы, семантический анализ выявляет логические ошибки, связанные с использованием типов, несоответствием операций и другими семантическими ограничениями.

Семантический анализатор работает на основе синтаксического дерева, полученного на предыдущем этапе компиляции. Его задача – проверить, что все операции выполняются над корректными типами данных, переменные объявлены до их использования, а преобразования типов соответствуют правилам языка. Например, при анализе выражения *x* = *y* + 5 необходимо убедиться, что переменная y объявлена и имеет числовой тип, а результат операции сложения может быть присвоен переменной *x*.

Важной частью семантического анализа является контроль типов. В языках со статической типизацией, таких как *C#*, типы всех переменных и выражений должны быть известны на этапе компиляции. Анализатор проверяет, что операции выполняются только над допустимыми комбинациями типов, а в случае неявных преобразований (например, автоматического приведения *int* к *double*) корректно обрабатывает такие ситуации.

Таким образом, задача данной лабораторной работы состоит в разработке семантического анализатора, который на основе синтаксического дерева проводит проверку типов, контролирует области видимости и выявляет семантические ошибки в программе, написанной на заданном подмножестве *C#.* Это позволит обеспечить корректность программы перед переходом к следующим этапам компиляции, таким как оптимизация и генерация кода.

1. описание работы программы

Разработанный семантический анализатор представляет собой компонент компилятора, выполняющий проверку исходного кода на соответствие семантическим правилам языка программирования *C#*. Программа реализована на языке *Erlang* [2] и принимает на вход синтаксическое дерево, сформированное на предыдущем этапе синтаксического анализа. Основная функция анализатора заключается в последовательном обходе дерева и проверке каждого узла на предмет семантической корректности. Анализатор начинает работу с обработки корневого узла программы, после чего рекурсивно проверяет все дочерние элементы, включая объявления переменных, операторы присваивания, арифметические и логические выражения.

Центральным механизмом работы анализатора является система проверки типов, которая обеспечивает контроль за соответствием типов в операциях присваивания и различных выражениях. При обработке объявления переменной анализатор проверяет совместимость типа переменной с типом присваиваемого значения. Для арифметических операций осуществляется проверка допустимости операции для данных типов операндов, а также обработка неявных преобразований типов, характерных для языка *C#*. Важной особенностью реализации является механизм контекста, который хранит информацию об объявленных переменных и их типах в различных областях видимости, что позволяет отслеживать корректность использования идентификаторов на протяжении всей программы.

Работа с ошибками организована таким образом, что анализатор продолжает проверку даже после обнаружения первой ошибки, накапливая все выявленные проблемы для последующего вывода. Это позволяет получить максимально полную информацию о семантических ошибках в коде за один проход анализа. Результатом работы программы является либо подтверждение семантической корректности анализируемого кода, либо список обнаруженных ошибок с указанием их характера нарушений, что делает вывод анализатора удобным для последующего исправления ошибок программистом.

1. ход выполнения программы

На рисунке 3.1 представлена часть синтаксического дерева разбора тестовой программы, содержащей различные объявления переменных и арифметические выражения.

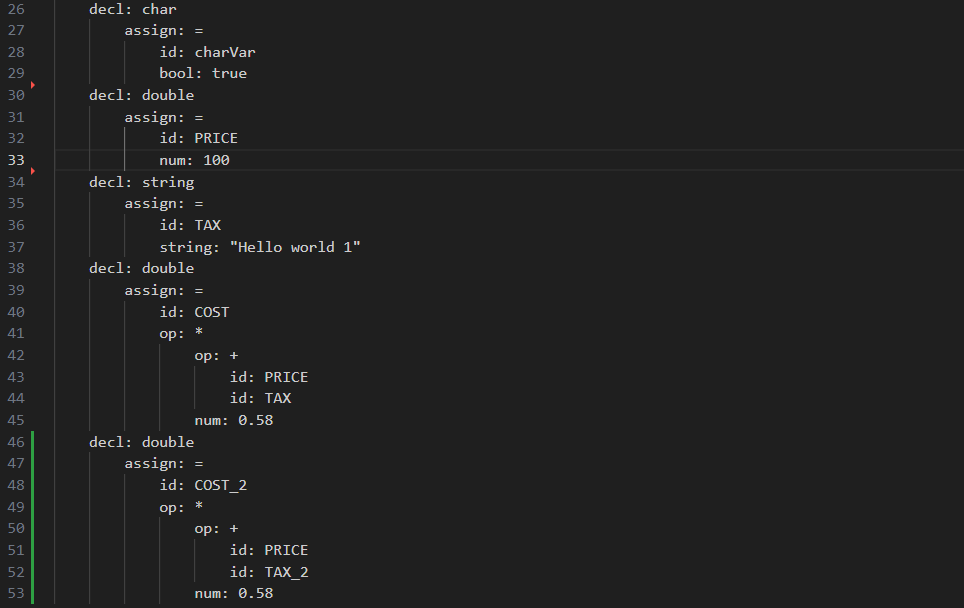


Рисунок 3.1 – Синтаксическое дерево части программы

При выполнении семантического анализа программа последовательно проверяет каждый узел дерева. В процессе анализа выявлены следующие семантические ошибки, отображенные на рисунке 3.2: несоответствие типов при объявлении переменных – попытка присвоения строкового значения переменной *intVar* целочисленного типа, присвоения значения *double* переменной *boolVar* логического типа. Также показаны ошибки в арифметических выражениях: недопустимая операция сложения между переменными *PRICE* (тип *double*) и *TAX* (тип *string*), а также использование необъявленной переменной *TAX\_2* в выражении для вычисления *COST\_2*.

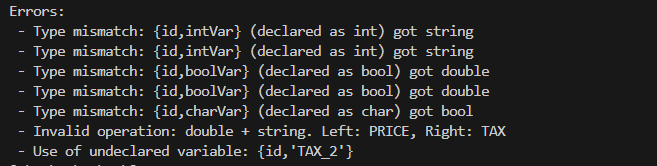


Рисунок 3.2 – Вывод семантических ошибок

Таким образом анализатор обрабатывает допустимые конструкции, такие как, например, объявление переменной *stringVar* с присвоением строкового значения. Для каждого объявления выводится отладочная информация о типе переменной и типе присваиваемого значения. В результате работы программа формирует список всех обнаруженных семантических ошибок.

Заключение

В ходе выполнения лабораторной работы №4 был разработан семантический анализатор для подмножества языка *C#*, реализованный на языке *Erlang* [3]. Программа продемонстрировала свою работоспособость в выявлении семантических ошибок в исходном коде. Анализатор корректно обрабатывает объявления переменных, операции присваивания, арифметические и логические выражения, а также обнаруживает несоответствия типов, недопустимые операции и использование необъявленных идентификаторов.

Ключевым достижением работы стала реализация системы проверки типов, которая автоматически выявляет семантические ошибки, включая несовместимость типов в операциях, некорректные присваивания и попытки использования переменных без предварительного объявления. Механизм контекста, отслеживающий типы переменных в различных областях видимости, обеспечивает точный контроль за корректностью программы.

Разработанный семантический анализатор не только подтверждает соответствие кода правилам языка, но и формирует подробный отчет об ошибках, что значительно упрощает процесс отладки. Результаты тестирования показали, что программа успешно обрабатывает как корректные конструкции, так и выявляет ошибочные случаи, включая недопустимые арифметические операции и попытки неявного преобразования типов.

Таким образом, лабораторная работа позволила получить практический опыт реализации важного этапа компиляции – семантического анализа.

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

ПРИЛОЖЕНИЕ А

(обязательное)  
Листинг программного кода

-module(semantic).

-export([analyze/1, analyze\_file/1]).

analyze(Tree) ->

InitialContext = [{'Scope', [], []}],

case check\_children(Tree, InitialContext, []) of

{ok, \_, Errors} when Errors =/= [] ->

{error, lists:reverse(Errors)};

{ok, \_, \_} ->

ok;

{error, Errors} ->

{error, lists:reverse(Errors)}

end.

analyze\_file(Filename) ->

case parse\_file(Filename) of

{ok, Tree} ->

io:format("[DEBUG] AST: ~p~n", [Tree]),

analyze(Tree);

{error, Reason} -> {error, Reason}

end.

parse\_file(Filename) ->

{ok, Data} = file:read\_file(Filename),

Lines = string:split(unicode:characters\_to\_list(Data), "\n", all),

{Nodes, \_Remaining} = parse\_block(Lines, 0),

{ok, Nodes}.

parse\_block([], \_CurrentIndent) ->

{[], []};

parse\_block([Line | Rest], CurrentIndent) ->

{Indent, Node} = parse\_line(Line),

if

Indent < CurrentIndent ->

{[], [Line | Rest]};

Indent == CurrentIndent ->

{Children, Rem1} = parse\_block(Rest, CurrentIndent + 1),

Node1 = case Children of

[] -> Node;

\_ -> add\_children(Node, Children)

end,

{Siblings, Rem2} = parse\_block(Rem1, CurrentIndent),

{[Node1 | Siblings], Rem2};

Indent > CurrentIndent ->

{[], [Line | Rest]}

end.

add\_children({Type, Value}, Children) ->

{Type, Value, Children}.

parse\_line(Line) ->

LineTrim = string:trim(Line),

IndentLevel = (string:length(Line) - string:length(LineTrim)) div 4,

[TypePart | ValueParts] = string:split(LineTrim, ":", leading),

Type = list\_to\_atom(string:trim(TypePart)),

Value = case ValueParts of

[] -> [];

[V] ->

VTrim = string:trim(V),

case Type of

num ->

case string:chr(VTrim, $.) of

0 -> {int, list\_to\_integer(VTrim)};

\_ -> {double, list\_to\_float(VTrim)}

end;

string -> {string, string:trim(VTrim, both, "\"")};

id -> {id, list\_to\_atom(VTrim)};

bool -> {bool, list\_to\_atom(VTrim)};

\_ -> list\_to\_atom(VTrim)

end

end,

{IndentLevel, {Type, Value}}.

check\_node({assign, '=', Children}, Context, Errors) ->

case Children of

[IdNode, ExprNode] ->

{id, \_} = IdNode,

case check\_expression(ExprNode, Context) of

{ok, \_, NewContext} ->

{ok, NewContext, Errors};

{error, Msg} ->

{ok, Context, [Msg | Errors]}

end;

\_ ->

{ok, Context, [format\_error("Invalid assignment structure", []) | Errors]}

end;

check\_node({op, Op, Left, Right}, Context, Errors) ->

case check\_expression(Left, Context) of

{error, Msg} -> {ok, Context, [Msg | Errors]};

{ok, LeftType, Ctx1} ->

case check\_expression(Right, Ctx1) of

{error, Msg} -> {ok, Context, [Msg | Errors]};

{ok, RightType, Ctx2} ->

case check\_operator(Op, Left, Right, LeftType, RightType, Ctx2) of

{ok, \_} -> {ok, Ctx2, Errors};

{error, Msg} -> {ok, Ctx2, [Msg | Errors]}

end

end

end;

check\_children([Node | Rest], Context, Errors) ->

case check\_node(Node, Context, Errors) of

{ok, NewCtx, NewErrors} ->

check\_children(Rest, NewCtx, NewErrors);

{error, NewErrors} ->

{error, NewErrors}

end.

check\_assignment(Type, Id, Expr, Context, Errors) ->

case check\_expression(Expr, Context) of

{ok, ExprType, NewContext} ->

UpdatedContext = add\_variable(Id, Type, NewContext),

io:format("[DEBUG] Declared ~p as ~p. Value type: ~p~n", [Id, Type, ExprType]),

case is\_convertible(ExprType, Type) of

true -> {ok, UpdatedContext, Errors};

false ->

Msg = format\_error("Type mismatch: ~p (declared as ~p) got ~p", [Id, Type, ExprType]),

{ok, UpdatedContext, [Msg | Errors]

end;

{error, Msg} -> {ok, Context, [Msg | Errors]}

end.

check\_expression({op, Op, Left, Right}, Context) ->

case check\_expression(Left, Context) of

{error, Msg} -> {error, Msg};

{ok, LeftType, Ctx1} ->

case check\_expression(Right, Ctx1) of

{error, Msg} -> {error, Msg};

{ok, RightType, Ctx2} ->

case check\_operator(Op, Left, Right, LeftType, RightType, Ctx2) of

{ok, Type} -> {ok, Type, Ctx2};

{error, Msg} -> {error, Msg}

end

end

end;

check\_operator(Op, LeftNode, RightNode, LeftType, RightType, Context) ->

LeftStr = expression\_to\_string(LeftNode, Context),

RightStr = expression\_to\_string(RightNode, Context),

case Op of

'+' ->

case {LeftType, RightType} of

{int, int} -> {ok, int};

{double, double} -> {ok, double};

{int, double} -> {ok, double};

{double, int} -> {ok, double};

{string, string} -> {ok, string};

\_ ->

{error, format\_error("Invalid operation: ~s + ~s. Left: ~s, Right: ~s",

[type\_to\_str(LeftType), type\_to\_str(RightType), LeftStr, RightStr])}

end;

'-' ->

case {LeftType, RightType} of

{int, int} -> {ok, int};

{double, double} -> {ok, double};

{int, double} -> {ok, double};

{double, int} -> {ok, double};

\_ ->

{error, format\_error("Invalid operation: ~s - ~s. Left operand: ~s, Right operand: ~s",[type\_to\_str(LeftType), type\_to\_str(RightType), LeftStr, RightStr])}

end;

\_ ->

{error, format\_error("Invalid operation: ~s \* ~s. Left op: ~s, Right op: ~s", [type\_to\_str(LeftType),type\_to\_str(RightType), LeftStr, RightStr])}

end;

\_ ->

{error, format\_error("Unsupported operator: ~s", [Op])}

end.

find\_variable(Id, [{'Scope', Vars, \_} | Rest]) ->

case proplists:get\_value(Id, Vars) of

undefined -> find\_variable(Id, Rest);

Type -> {ok, Type}

end;

find\_variable(\_, []) -> not\_found.

is\_convertible(From, To) ->

case {From, To} of

{int, int} -> true;

{int, double} -> true;

{double, double} -> true;

{string, string} -> true;

{bool, bool} -> true;

{char, char} -> true;

{var, \_} -> true;

{\_, var} -> true;

\_ -> false

end.

format\_error(Fmt, Args) ->

lists:flatten(io\_lib:format(Fmt, Args)).