Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ОТЧЕТ

к лабораторной работе №5

на тему

**ИНТЕРПРЕТАЦИЯ ПРОГРАММЫ**

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 реализации интерпретатора для подмножества языка *C#* на языке *Erlang*. Интерпретация представляет собой заключительный этап цепочки обработки программ, следующих за этапами лексического, синтаксического и семантического анализа, разработанных в рамках предыдущих лабораторных работ. В отличие от компиляции, интерпретация предполагает немедленное выполнение программы на основе её синтаксического дерева (*AST*), без генерации промежуточного или машинного кода.

Интерпретатор должен обрабатывать структуры, характерные для *C#*, такие как объявления и присваивания переменных различных типов (*int, double, bool, char, string, var*), управляющие конструкции (*циклы for, while, do-while*, условные операторы *if-else*, оператор выбора *switch*), а также вызовы методов (в частности, *Console.WriteLine*). Интерпретация выражений должна учитывать текущие значения переменных и выполнять соответствующие арифметические и логические операции, включая инкремент, сравнение и вычисление выражений с приоритетом операций.

В процессе выполнения лабораторной работы необходимо реализовать обработку *AST*-дерева, выполнять семантически корректные действия и выводить информативные сообщения о ходе интерпретации, отражающие изменения в окружении переменных и выполненные команды. Выходные данные должны служить подтверждением корректной интерпретации программы: от объявления переменных до логики выполнения управляющих конструкций.

Таким образом, основная задача лабораторной работы – создать интерпретатор, способный последовательно выполнять инструкции, представленные в виде абстрактного синтаксического дерева, обеспечивая корректное поведение программы и позволяя визуально проследить ход её выполнения.

1. описание работы программы

Разработанный интерпретатор представляет собой программный компонент, реализующий выполнение кода на языке *C#* на основе его абстрактного синтаксического дерева (*AST*), сформированного в ходе предыдущих этапов анализа. Программа реализована на языке *Erlang* и предназначена для интерпретации исходного кода, представленного в виде дерева разбора.

Основная функция интерпретатора заключается в последовательном обходе узлов *AST* и выполнении операций, описанных в этих узлах. Интерпретатор обрабатывает такие конструкции, как объявления переменных, присваивания, арифметические выражения, условные операторы, циклы и вызовы функций. Объявления переменных сопровождаются сохранением их значений в специальном контексте (окружении), реализованном в виде отображения (*map*), где ключами являются имена переменных, а значениями – соответствующие данные.

Важным элементом является система вычисления выражений (*eval\_expr* и *eval\_expr\_node*), способная обрабатывать базовые типы (*int, double, char, string, bool*), арифметические и логические операции (+, -, \*, <, == и др.), а также выполнять инкременты (++) и сравнения. При обработке циклов *for, while, do-while* используются отдельные функции (*loop\_for, loop\_while, loop\_dowhile*), реализующие повторное выполнение тела блока в зависимости от результата вычисления условия. Обработка условных операторов (*if*) реализована через выделение подузлов *condition, if\_body* и *else\_body*, с логгированием результата условия и выбором соответствующей ветки исполнения.

Ветка *switch* реализована с сопоставлением ключей, включая строковые значения, и имеет поддержку ветки *default*. Также предусмотрены вспомогательные функции (*get\_node\_value, get\_node\_block, get\_node\_cases*) для извлечения нужных компонентов из *AST*.

Интерпретатор имеет систему логирования, выводящую пошаговую информацию о ходе выполнения программы: объявления переменных, значения выражений, переходы по условиям и веткам *switch*, а также вызовы вывода на консоль.

Результатом работы интерпретатора является лог, отображающий ход выполнения программы.

1. ход выполнения программы

На рисунке 3.1 представлена часть вывода выполнения программы, которая показывает обработку объявления переменных, конструкций вывода строк на консоль, а также то, что, если переменной было присвоено некоторое арифметическое выражение, то оно обрабатывается и в переменную записывается результат выполнения (*int COST = ( PRICE + TAX ) \* 0.58*).

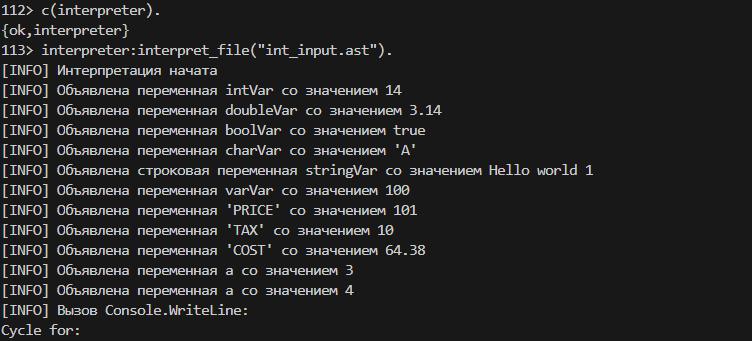


Рисунок 3.1 – Результат выполнения программы

На рисунке 3.2 показан фрагмент выполнения управляющих конструкций и ветвлений. Сначала выполняется цикл *for,* в котором на каждой итерации выводится строка «*Cycle for*» и переменная *i* увеличивается с 0 до 4. Затем демонстрируется *while*-цикл с выводом «*hello from cycle*», после чего *do-while*-цикл с «*Hi from do-while*». Далее условный оператор соответственно выбирает ветку *else* и выводит «*num <= 5*». И конструкция *switch (day)*: интерпретатор пошагово сравнивает значение переменной *day* со строками *Monday* и *Tuesday*, что показывает корректную работу механизма выбора веток по совпадению ключей.

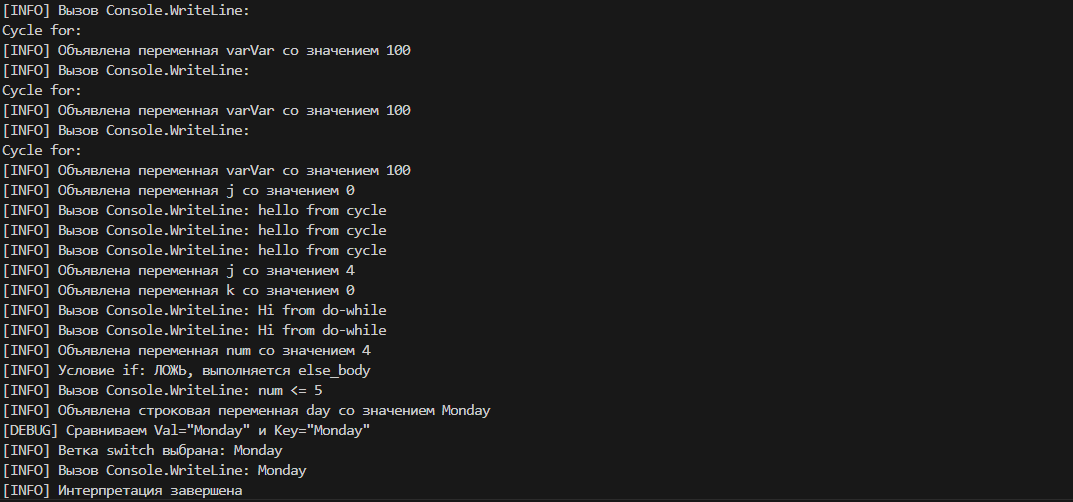


Рисунок 3.2 – Вывод обработки конструкций

В результате формируется подробный пооперационный лог, показывающий ход выполнения программы, изменения окружения и все вызовы ввода-вывода.

Заключение

В ходе выполнения лабораторной работы №5 был разработан интерпретатор для подмножества языка *C#*, реализованный на языке *Erlang*. Интерпретатор осуществляет выполнение исходной программы, представленной в виде абстрактного синтаксического дерева (*AST*), и отражает финальный этап обработки кода после лексического, синтаксического и семантического анализа, реализованных ранее.

Программа корректно обрабатывает объявления переменных различных типов, присваивания, арифметические и логические выражения, а также управляющие конструкции, включая циклы (*for, while, do-while*), условные операторы *if-else* и оператор *switch*. В процессе выполнения интерпретатор последовательно вычисляет выражения, обновляет окружение переменных и выводит подробную информацию о каждом шаге интерпретации.

Одним из ключевых аспектов работы стала реализация механизма окружения, в котором отслеживаются текущие значения переменных. Это позволило достичь корректной интерпретации вложенных выражений и сложных управляющих конструкций, а также упростило отладку за счёт генерации подробного пооперационного лога. Программа также обрабатывает вызовы встроенных методов, таких как *Console.WriteLine*, с выводом соответствующих сообщений.

Результаты тестирования продемонстрировали обработку различных конструкций языка и соответствие поведения интерпретатора ожидаемой логике исполнения программы.

Список литературных источников

1. Erlang/OTP – documentation [Электронный ресурс]. – Режим доступа: https://www.erlang.org/. – Дата доступа: 20.04.2025.
2. Основы C# [Электронный ресурс]. – Режим доступа: https://code-basics.com/ru/languages/csharp. – Дата доступа: 20.04.2025.
3. Разработка интерпретатора. Теоретическая часть [Электронный ресурс]. – Режим доступа https://studfile.net/preview/16490435/. – Дата доступа: 20.04.2025.

ПРИЛОЖЕНИЕ А

(обязательное)  
Листинг программного кода

-module(interpreter).

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

interpret(Tree) ->

io:format("[INFO] Интерпретация начата~n"),

Env0 = #{},

interpret\_nodes(Tree, Env0),

io:format("[INFO] Интерпретация завершена~n").

interpret\_file(Filename) ->

case semantic:parse\_file(Filename) of

{ok, Tree} -> interpret(Tree);

{error, Reason} ->

io:format("[ERROR] Не удалось прочитать AST из файла: ~p~n", [Reason])

end.

interpret\_nodes([], Env) -> Env;

interpret\_nodes([{program, \_, Nodes} | Rest], Env) ->

Env1 = interpret\_nodes(Nodes, Env),

interpret\_nodes(Rest, Env1);

interpret\_nodes([Node | Rest], Env) ->

Env1 = interpret\_node(Node, Env),

interpret\_nodes(Rest, Env1).

interpret\_node({decl, char, [{assign, '=', [{id, {id, Id}}, Expr]}]}, Env) ->

{Value, Env1} = eval\_expr(Expr, Env),

io:format("[INFO] Объявлена переменная ~p со значением '~s'~n", [Id, Value]),

maps:put(Id, Value, Env1);

interpret\_node({decl, string, [{assign, '=', [{id, {id, Id}}, Expr]}]}, Env) ->

{Value, Env1} = eval\_expr(Expr, Env),

io:format("[INFO] Объявлена строковая переменная ~p со значением ~s~n", [Id, Value]),

maps:put(Id, Value, Env1);

interpret\_node({decl, \_Type, [{assign, '=', [{id, {id, Id}}, Expr]}]}, Env) ->

{Value, Env1} = eval\_expr(Expr, Env),

io:format("[INFO] Объявлена переменная ~p со значением ~p~n", [Id, Value]),

maps:put(Id, Value, Env1);

interpret\_node({assign, '=', [{id, {id, Id}}, Expr]}, Env) ->

{Value, Env1} = eval\_expr(Expr, Env),

io:format("[INFO] Присвоение переменной ~p значения ~p~n", [Id, Value]),

maps:put(Id, Value, Env1);

interpret\_node({call, 'Console.WriteLine', Children}, Env) when is\_list(Children) ->

Values = lists:map(

fun(Node) ->

{V,\_Env2} = eval\_expr(Node, Env),

V

end,

Children

),

Out = lists:flatten(Values),

io:format("[INFO] Вызов Console.WriteLine: ~s~n", [Out]),

Env;

interpret\_node({for, \_Label, [Init, Cond, Inc, {block, \_, Body}]}, Env) ->

Env1 = interpret\_node(Init, Env),

loop\_for(Cond, Inc, Body, Env1);

interpret\_node({op, '++', [{id,{id,Id}}]}, Env) ->

do\_increment({op,'++', [{id,{id,Id}}]}, Env);

interpret\_node({while, \_Label, [CondNode, {block, \_, Body}]}, Env) ->

loop\_while(CondNode, Body, Env);

interpret\_node({dowhile, \_Label, [ {block, \_, Body}, CondNode ]}, Env) ->

loop\_dowhile(Body, CondNode, Env);

interpret\_node({Tag, \_, Children}, Env) when Tag =:= 'if' ->

Cond = get\_node\_value(condition, Children),

ThenBlock = get\_node\_block(if\_body, Children),

ElseBlock = get\_node\_block(else\_body, Children),

{CondVal, Env1} = eval\_expr\_node(Cond, Env),

case CondVal of

true ->

io:format("[INFO] Условие if: ИСТИНА, выполняется if\_body~n"),

interpret\_nodes(ThenBlock, Env1);

false ->

io:format("[INFO] Условие if: ЛОЖЬ, выполняется else\_body~n"),

interpret\_nodes(ElseBlock, Env1)

end;

interpret\_node({switch, \_, Ch}, Env) ->

RawExpr = get\_node\_value(expr, Ch),

ExprAST = case RawExpr of

Atom when is\_atom(Atom) -> {id,{id,Atom}};

Other -> Other

end,

{Val, Env1} = eval\_expr\_node(ExprAST, Env),

Cases = get\_node\_cases(cases, Ch),

do\_switch(Val, Cases, Env1);

interpret\_node(\_, Env) ->

Env.

do\_switch(\_Val, [], Env) ->

Env;

do\_switch(Val, [{ 'case', KeyAST, Actions } | Rest], Env) ->

Key = case KeyAST of

{string,{string,S}} -> S;

Atom when is\_atom(Atom) ->

string:trim(atom\_to\_list(Atom), both, "\"");

\_ ->

io:format("[WARN] Нестандартный ключ case: ~p~n", [KeyAST]),

""

end,

ValFixed =

case Val of

[Str] when is\_list(Str) -> Str;

\_ -> Val

end,

io:format("[DEBUG] Сравниваем Val=~p и Key=~p~n", [ValFixed, Key]),

if

ValFixed =:= Key ->

io:format("[INFO] Ветка switch выбрана: ~s~n", [Key]),

interpret\_nodes(Actions, Env);

true ->

do\_switch(Val, Rest, Env)

end;

do\_switch(\_Val, [{default, \_, Actions} | \_], Env) ->

io:format("[INFO] Ветка switch: default~n", []),

interpret\_nodes(Actions, Env).

get\_node\_value(Key, [{Key, \_, [V]} | \_]) -> V;

get\_node\_value(Key, [{Key, V} | \_]) -> V;

get\_node\_value(Key, [\_ | R]) -> get\_node\_value(Key, R).

get\_node\_block(Key, [{Key, \_, [{block, \_, Block}]} | \_]) -> Block;

get\_node\_block(Key, [\_ | Rest]) -> get\_node\_block(Key, Rest).

get\_node\_cases(Key, [{Key, \_, Cs} | \_]) -> Cs;

get\_node\_cases(Key, [\_ | R]) -> get\_node\_cases(Key, R).

loop\_for(CondNode, IncNode, Body, Env) ->

{CondVal, Env1} = eval\_expr\_node(CondNode, Env),

case CondVal of

true ->

Env2 = interpret\_nodes(Body, Env1),

Env3 = do\_increment(IncNode, Env2),

loop\_for(CondNode, IncNode, Body, Env3);

false ->

Env1

end.

loop\_while(CondNode, Body, Env) ->

{CondVal, Env1} = eval\_expr\_node(CondNode, Env),

case CondVal of

true ->

Env2 = interpret\_nodes(Body, Env1),

loop\_while(CondNode, Body, Env2);

false ->

Env1

end.

loop\_dowhile(Body, CondNode, Env) ->

Env1 = interpret\_nodes(Body, Env),

{CondVal, Env2} = eval\_expr\_node(CondNode, Env1),

case CondVal of

true -> loop\_dowhile(Body, CondNode, Env2);

false -> Env2

end.

eval\_expr\_node({op, Op, L, R}, Env) ->

eval\_expr({op, Op, L, R}, Env);

eval\_expr\_node({op, Op, Args}, Env) when is\_list(Args) ->

eval\_expr({op, Op, Args}, Env);

eval\_expr\_node(Node, Env) ->

eval\_expr(Node, Env).

do\_increment({op, '++', [{id,{id,Id}}]}, Env) ->

Cur = maps:get(Id, Env),

maps:put(Id, Cur + 1, Env);

do\_increment(\_, Env) ->

Env.

eval\_expr({num, {int, N}}, Env) -> {N, Env};

eval\_expr({num, {double, N}}, Env) -> {N, Env};

eval\_expr({string, {string, S}}, Env) ->

{unescape(S), Env};

eval\_expr({bool, {bool, B}}, Env) ->

{B, Env};

eval\_expr({char, {char, C}}, Env) -> normalize\_char(C, Env);

eval\_expr({char, C}, Env) -> normalize\_char(C, Env);

eval\_expr({id, IdAtom}, Env) when is\_atom(IdAtom) ->

eval\_expr({id, {id, IdAtom}}, Env);

eval\_expr({id, {id, Id}}, Env) ->

case maps:get(Id, Env, undefined) of

undefined ->

io:format("[ERROR] Не определена переменная ~p~n", [Id]),

{undefined, Env};

Val ->

{Val, Env}

end;

eval\_expr({op, '<', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV < RV, Env2};

eval\_expr({op, '>', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV > RV, Env2};

eval\_expr({op, '<=', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV =< RV, Env2};

eval\_expr({op, '>=', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV >= RV, Env2};

eval\_expr({op, '==', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV =:= RV, Env2};

eval\_expr({op, '!=', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV =/= RV, Env2};

eval\_expr({op, '+', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV + RV, Env2};

eval\_expr({op, '-', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV - RV, Env2};

eval\_expr({op, '\*', L, R}, Env) ->

{LV, Env1} = eval\_expr(L, Env),

{RV, Env2} = eval\_expr(R, Env1),

{LV \* RV, Env2};

eval\_expr({op, Op, [L, R]}, Env) ->

eval\_expr({op, Op, L, R}, Env);

eval\_expr(Unknown, Env) ->

io:format("[WARN] Не удалось вычислить узел: ~p~n", [Unknown]),

{undefined, Env}.

normalize\_char(C, Env) ->

Base = case C of

Atom when is\_atom(Atom) -> atom\_to\_list(Atom);

L when is\_list(L) -> L

end,

Stripped = string:trim(Base, both, "'"),

{Stripped, Env}.

unescape(Str) when is\_list(Str) ->

Step1 = string:replace(Str, "\\n", "\n", all),

Step1.