Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра ПОИТ

Дисциплина «Компиляторные технологии»

ОТЧЁТ
к лабораторной работе №3

РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА, ОСНОВАННОГО НА МЕТОДЕ РЕКУРСИВНОГО СПУСКА

Вариант 9

Студент группы
№351001
Ушаков А.Д.

Преподаватель
Болтак С.В.

Минск 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Синтаксический анализ – это этап трансляции, задача которого заключается в построении древовидного представления анализируемого текста (например, исходного кода программы). Как правило, на вход синтаксическому анализатору (который также называют *парсером*) подаются токены, получаемые в результате обработки анализируемого текста лексическим анализатором.

Теоретически возможно записать грамматику, терминалами в которой будут символы в той или иной кодировке, однако такая грамматика для большинства применяемых на практике компьютерных языков окажется излишне громоздкой, т.к. должна будет учитывать возможность появления пробельных символов или комментариев между любыми двумя лексемами.

Построение дерева разбора может выполняться в одном из двух направлений: от корня к листьям или от листьев к корню. По этому принципу методы синтаксического анализа принято разделять на две большие группы:

- нисходящие: дерево строится от корня к листьям;
- восходящие: дерево строится от листьев к корню.

Независимо от порядка построения листья в деревьях разбора содержат терминальные символы грамматики, промежуточные узлы (в т.ч. и корневой) – нетерминалы. При этом любой фрагмент дерева разбора, состоящий из промежуточного узла и его дочерних элементов, в точности соответствует одной из продукций грамматики — той, в левой части которой записан нетерминал из выбранного промежуточного узла, а в правой – последовательно заданы терминалы и нетерминалы дочерних узлов.

Очевидным недостатком деревьев разбора является зависимость их структуры от структуры грамматики. Фактически это означает, что любое изменение в грамматике может существенно изменить структуру дерева разбора, что потребует внесения изменений в код, отвечающий за последующие этапы трансляции. По этой причине более целесообразным, как правило, оказывается использование вместо деревьев разбора так называемых абстрактных синтаксических деревьев (abstract syntax trees – AST).

AST, как правило, представляет собой отражение логической структуры анализируемого текста, не зависящей от состава продукций грамматики.

Метод рекурсивного спуска является простым в реализации и вместе с тем эффективным методом синтаксического анализа. Суть данного метода заключается в том, что начиная со стартового символа грамматики поочерёдно перебираются все возможные комбинации продукций до тех пор, пока не будет найдено порождение, соответствующее анализируемому тексту, либо пока не окажется, что такого порождения не существует.

1 ОПИСАНИЕ ЛАБОРАТОРНОЙ РАБОТЫ

1.1 Задание к лабораторной работе

В девятом варианте лабораторной работы объектом для проверки является код на PHP. Необходимо разработать лексический анализатор, грамматику языка как минимум для трёх базовых конструкций и программное средство на языке C, проверяющее исходный код на соответствие грамматике.

При создании лексического анализатора допускается использование утилит-генераторов. Поддерживаемое подмножество языка согласовать с преподавателем.

На каждом этапе анализа программное средство должно выводить сообщение с номером строки, в которой была допущена ошибка. Также необходимо вывести все встретившиеся лексемы.

Ввод и вывод осуществляются в текстовых файлах.

1.2 Базовые допустимые конструкции языка

Для синтаксического анализа были выбраны следующие конструкции:

– операторы вывода строковых литералов и идентификаторов на экран print и echo;

– оператор присваивания строковых литералов или идентификаторов;

– пустой оператор;

– ассоциативные массивы;

– блочный оператор foreach, допускающий ассоциативные массивы, любую вложенность операторов и пустые операторы;

– сам скрипт на PHP является конструкцией, допускающей любые вложенность и количество операторов, а также имеющей открывающие и закрывающие лексемы.

1.3 Использованные лексемы

К задействованным в синтаксическом анализе лексемам относятся идентификатор (Identifier), строковый литерал (Literal), символ «равно» (Equal), открывающие и закрывающие скобки (Open и Close – обычные, Open_block и Close_block – для блока), точка с запятой (Ddot), стрелка “=>” в ассоциативном массиве (Dir), зарезервированные слова (Foreach, Echo, Print, As). В скобках на английском языке указаны имена этих лексем, определённые в программе перечислимым типом TOKEN.

1.4 Эскиз ожидаемого результата

Допустим, дан следующий скрипт на PHP:

```
<?php
        foreach ($array as $key => $value) {
                ;
        }
?>
```

Тогда результат работы программы будет выглядеть следующим образом:

Lexical analysis stage: #Этап лексического анализа

Found the token Open_php: <?php
Found the token Foreach: foreach
Found the token Open: (
Found the token Identifier: $array
Found the token As: as
Found the token Identifier: $key
Found the token Dir: =>
Found the token Identifier: $value
Found the token Close:)
Found the token Open_block: {
Found the token Ddot: ;
Found the token Close_block: }
Found the token Close_php: ?> #Найденные лексемы

Syntactic analysis stage: #Этап синтаксического анализа

#Обработанный поток токенов:

Open_php Foreach Open Identifier As Identifier Dir Identifier Close Open_block Ddot Close_block Close_php

Syntax is correct #Сообщение результата

2 РЕАЛИЗАЦИЯ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

2.1 Грамматика для базовых конструкций

PHP –> Open_php Operators Close_php

Operators –> Operator*
Operator –> Op_Input | Op_Equals | Op_Foreach

Op_Input –> Op_Echo | Op_Print
Op_Foreach –> Foreach Open Identifier As Assoc_OR_Indent Close Open_block Op_Exist Close_block
Op_Equals –> Identifier Equal Param Ddot

Op_Echo –> Echo Param Ddot
Op_Print –> Print Param Ddot
Assoc_OR_Indent –> Association | Identifier
Association –> Identifier Dir Identifier
Op_Exist –> Operators | Ddot
Param –> Identifier | Literal

Open_php –> ‘<?php ’
Close_php –> ‘ ?>’
Foreach –> ‘foreach ’
Open –> ‘(’
Identifier –> ‘$’[a-zA-Z]+[a-zA-Z0-9_]*
As –> ‘ as ’
Close –> ‘)’
Open_block –> ‘{’
Close_bock –> ‘}’
Equal –> ‘=’
Echo –> ‘echo ’
Print –> ‘print ’
Dir –> ‘=>’
Ddot –> ‘;’
Literal –> ‘"’([^‘"’]*)‘"’

2.2 Построение дерева разбора

Допустим, на вход дан следующий скрипт на PHP:

```
<?php
      $par1 = "Hello";
      echo "Hello";
      foreach ($arr as $key => $value) {
            ;
      }
?>
```

На рисунке 2.2.1 изображено дерево разбора для данного скрипта.

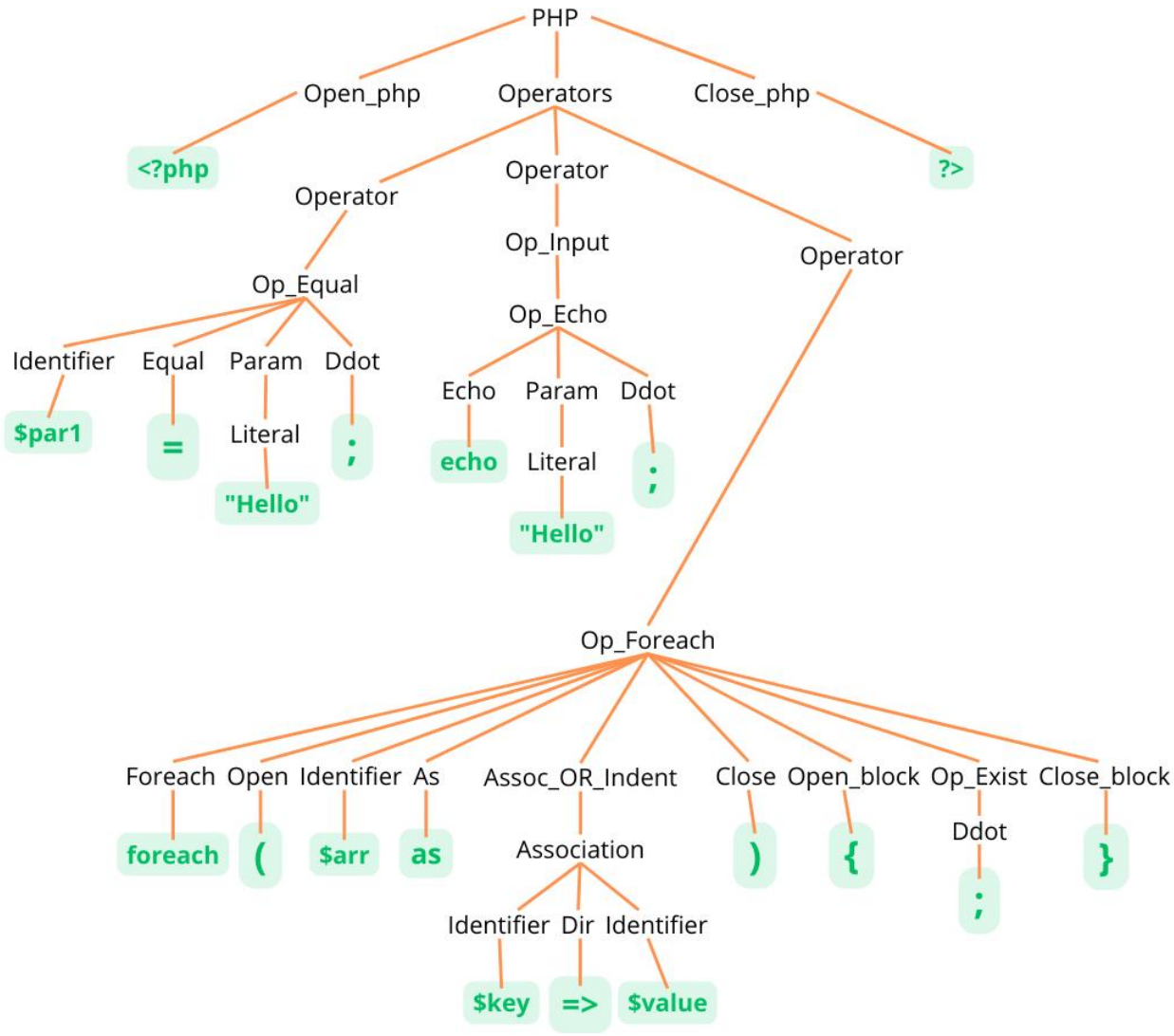

Рисунок 2.2.1 – Дерево разбора кода на PHP

2.3 Код и запуск программы

Программное средство было написано на языке C. Однако представленный код не является файлом с расширением .c в привычном понимании. Код написан под утилиту-генератор flex. После этого программа сгенерировала lex.yy.c, который впоследствии был скомпилирован в исполняемый файл лексического анализатора lexer.exe. Чтобы запустить программу для анализа текста в файле input.txt, ожидая увидеть результат в output.txt, необходимо в cmd ввести команду lexer < input.txt > output.txt, предварительно перейдя в директорию расположения исполняемого файла.

Код программы выглядит следующим образом:

```
%option noyywrap

%{

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <stdbool.h>

    #define TOKEN_COUNT 500
    #define LINE_COUNT 100

    enum TOKEN {
        Identifier,
        Literal,
        Equal,
        Open,
        Close,
        Dot,
        Ddot,
        Open_php,
        Close_php,
        Dir,
        Open_block,
        Close_block,
        Foreach,
        Echo,
        Print,
        As
    };

    const char* tokenNames[] = {
        "Identifier",
        "Literal",
        "Equal",
```

```
        "Open",
        "Close",
        "Dot",
        "Ddot",
        "Open_php",
        "Close_php",
        "Dir",
        "Open_block",
        "Close_block",
        "Foreach",
        "Echo",
        "Print",
        "As"
    };

    struct TOKENS {
        enum TOKEN tokenType;
        int errorLine;
    };


    struct TOKENS currentStr[TOKEN_COUNT];
    int nextTokenIndex = 0;
    int tokenIndexCount = 0;
    int lexLine = 1;
    int errorLine;
    bool isError = false;
    bool startSynt = true;

    void printTokens() {
        printf("Tokens in currentStr:\n");
        for (int i = 0; i < tokenIndexCount; i++) {
            struct TOKENS saveTOKEN = currentStr[i];
            printf("Token %d: %s %i\n", i,
tokenNames[saveTOKEN.tokenType], saveTOKEN.errorLine);
        }
    }

    bool isPHP();
    bool isOperators(enum TOKEN);
    bool isOperator();
    bool isOp_Input();
    bool isOp_Foreach();
    bool isOp_Equals();
    bool isOp_Echo();
    bool isOp_Print();
    bool isOp_Exist();
    bool isAssos_OR_Ident();
    bool isAssociation();
    bool isParam();
    bool isExpression();
```

```
    bool checkTerm(enum TOKEN);

    bool isPHP() {
        return checkTerm(Open_php) &&
               isOperators(Close_php) &&
               checkTerm(Close_php);
    }

    bool isOperators(enum TOKEN stopConditionToken) {
        bool result = true;
        while(result && (currentStr[nextTokenIndex].tokenType !=
stopConditionToken)) {
            result = isOperator();
        }
        return result;
    }

    bool isOperator() {
        int currentTokenIndex = nextTokenIndex;
        return  (nextTokenIndex = currentTokenIndex, isOp_Input()) ||
                (nextTokenIndex = currentTokenIndex, isOp_Equals()) ||
                (nextTokenIndex = currentTokenIndex, isOp_Foreach());
    }

    bool isOp_Input() {
        int currentTokenIndex = nextTokenIndex;
        return (nextTokenIndex = currentTokenIndex, isOp_Echo()) ||
               (nextTokenIndex = currentTokenIndex, isOp_Print());
    }

    bool isOp_Exist() {
        int currentTokenIndex = nextTokenIndex;
        return   (nextTokenIndex =
currentTokenIndex,  checkTerm(Ddot)) ||
                 (nextTokenIndex = currentTokenIndex,
isOperators(Close_block));
    }

    bool isOp_Foreach() {
        return checkTerm(Foreach) &&
               checkTerm(Open) &&
               checkTerm(Identifier) &&
               checkTerm(As) &&
               isAssos_OR_Ident() &&
               checkTerm(Close) &&
               checkTerm(Open_block) &&
               isOp_Exist() &&
               checkTerm(Close_block);
    }

    bool isOp_Equals() {
```

```
        return checkTerm(Identifier) &&
               checkTerm(Equal) &&
               isParam() &&
               checkTerm(Ddot);
    }

    bool isOp_Echo() {
        return checkTerm(Echo) && isParam() && checkTerm(Ddot);
    }

    bool isOp_Print() {
        return checkTerm(Print) && isParam() && checkTerm(Ddot);
    }

    bool isAssos_OR_Ident() {
        int currentTokenIndex = nextTokenIndex;
        return  (nextTokenIndex = currentTokenIndex, isAssociation())
||
                (nextTokenIndex = currentTokenIndex,
checkTerm(Identifier));
    }

    bool isAssociation() {
        return checkTerm(Identifier) &&
               checkTerm(Dir) &&
               checkTerm(Identifier);
    }

    bool isParam() {
        int currentTokenIndex = nextTokenIndex;
        return  (nextTokenIndex = currentTokenIndex,
checkTerm(Identifier)) ||
                (nextTokenIndex = currentTokenIndex,
checkTerm(Literal));
    }

    bool checkTerm(enum TOKEN currentToken) {
        bool result;
        enum TOKEN saveToken = currentStr[nextTokenIndex].tokenType;

        /*printf("Checking token: %s (expected: %s) on line %d\n",
           tokenNames[saveToken], tokenNames[currentToken],
currentStr[nextTokenIndex].errorLine);*/

        if (saveToken == currentToken) {
            printf("%s ", tokenNames[saveToken]);
            result = true;
        } else {
            result = false;
            errorLine = currentStr[nextTokenIndex].errorLine;
        }
```

```
        if (nextTokenIndex < tokenIndexCount) nextTokenIndex++;

        return result;
    }

%}

%%

\"([^\"]*)\" {

    printf("Found the token Literal: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Literal;

}

\$[a-zA-Z]+[a-zA-Z0-9_]* {

    printf("Found the token Identifier: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Identifier;

}

"=>" {

    printf("Found the token Dir: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Dir;

}

= {

    printf("Found the token Equal: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Equal;

}

[{] {

    printf("Found the token Open_block: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Open_block;

}

[}] {
```

```
    printf("Found the token Close_block: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Close_block;

}

[(] {

    printf("Found the token Open: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Open;

}

[)] {

    printf("Found the token Close: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Close;

}

\. {

    printf("Found the token Dot: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Dot;

}

; {

    printf("Found the token Ddot: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Ddot;

}

"echo " {

    printf("Found the token Echo: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Echo;

}

"print " {

    printf("Found the token Print: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
```

```
    currentStr[tokenIndexCount++].tokenType = Print;

}

"foreach" {

    printf("Found the token Foreach: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Foreach;

}

" as " {

    printf("Found the token As: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = As;

}

"<?php" {

    printf("Found the token Open_php: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Open_php;

}

"?>" {

    printf("Found the token Close_php: %s\n", yytext);
    currentStr[tokenIndexCount].errorLine = lexLine;
    currentStr[tokenIndexCount++].tokenType = Close_php;

}

[ \t] { }

. {

    isError = true;
    startSynt = false;

}

\r?\n {
    if (isError) {
        printf("\nError was found in the line %d\n\n", lexLine);
    }
    isError = false;
    lexLine++;
```

```
}

%%

int main(void) {

    printf("Lexical analysis stage:\n\n\n");
    yylex();

    printf("\n\n\n\nSyntactic analysis stage:\n\n\n");
    if (startSynt) {
        bool syntCorrect = true;
        while (nextTokenIndex < tokenIndexCount && syntCorrect) {
            syntCorrect = isPHP();
            /*printf("\n %d %d \n", nextTokenIndex, nextTokenIndex);*/
        }
        if (syntCorrect) {
            printf("\n\nSyntax is correct\n\n");
        } else {
            printf("\nError was found in line: %d\n\nSyntax is
incorrect\n\n", errorLine);
        }
    } else {
        printf("Errors were found in the lexis analysis - syntax
analysis was not started\n\n");
    }

    return 0;
}
```

2.4 Тестирование программы

В таблице 1 пердставлено некоторое количесвто тестов для отладки программы и отражено их соответствие ожидаемому результату.

Таблица 1 – Тестирование программы

Код исходного файла	Ожидаемый результат работы программы	Фактический результат работы программы	Результат теста
<?php echho "Hello"; ?>	Errors were found in the lexis analysis in the line 2 - syntax analysis was not started	Errors were found in the lexis analysis in the line 2 - syntax analysis was not started	Пройден
<?php echo "Hello"; ?>	Syntax is correct	Syntax is correct	Пройден
<?php echo ; ?>	Error was found in line: 2 Syntax is incorrect	Error was found in line: 2 Syntax is incorrect	Пройден
<?php echo $a; ?> <?php echo $b; ?>	Syntax is correct	Syntax is correct	Пройден
<?php echo $a; ?> foreach	Error was found in line: 2 Syntax is incorrect	Error was found in line: 2 Syntax is incorrect	Пройден

Продолжение таблицы 1

<?php
foreach ($par1 as $arr) {
foreach ($par1 as $arr => $arr1) {
foreach ($par1 as $arr) {
foreach ($par1 as $arr) {
print $app;
$par3 = "Literal";
foreach ($par1 as $par2 => $par4) { ; }
}
}
}
}
?>
	Syntax is correct	Syntax is correct	Пройден

<?php
foreach ($par1 as $arr) {
foreach ($par1 as $arr => $arr1) {
foreach ($par1 as $arr) {
foreach ($par1 as $arr) {
print $app;
$par3 = "Literal";
foreach ($par1 as $par2 => $par4)
}
}
}
}
?>
	Error was found in line: 9
	
	Syntax is incorrect
		Error was found in line: 9
		
		Syntax is incorrect
			Пройден

2.5 Вывод по второй главе

В разработки программного средства был успешно реализован синтаксический анализатор, использующий метод рекурсивного спуска для обработки входных последовательностей. Анализатор способен распознавать заданный контекстно-свободный язык, обеспечивая корректный разбор синтаксических конструкций. Реализация показала высокую эффективность и простоту в расширении, что позволяет легко добавлять новые правила грамматики. Основные преимущества данного метода включают интуитивность структуры кода и возможность отладки, однако стоит отметить, что он может сталкиваться с проблемами, связанными с левой рекурсией и неразрешимыми конфликтами.

Метод рекурсивного спуска эффективно реализует синтаксический анализ для контекстно-свободных грамматик (Тип 2 по классификации Хомского). Он позволяет строить синтаксические деревья, преобразуя правила грамматики в рекурсивные функции, что делает процесс анализа интуитивно понятным и удобным. Классификация Хомского помогает четко понять, какие языки могут быть описаны различными типами грамматик и какие методы анализа подходят для них.

Данный метод отлично подходит для языков программирования, так как обеспечивает необходимую эффективность и простоту реализации. В то же время, он демонстрирует свои ограничения при работе с более сложными грамматиками, такими как контекстно-зависимые, для которых необходимы другие подходы.

Таким образом, разработка подтверждает, что метод рекурсивного спуска является мощным инструментом для синтаксического анализа, особенно в контексте контекстно-свободных языков, и подчеркивает важность классификации Хомского для выбора адекватных методов анализа формальных языков. В дальнейшем рекомендуется рассмотреть возможности оптимизации и расширения функционала анализатора для поддержки более сложных языковых конструкций.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы по созданию синтаксического анализатора методом рекурсивного спуска был изучен и проанализирован этап синтаксического анализа как ключевая часть трансляции исходного кода программы. На этом этапе на основе токенов, переданных от лексического анализатора, строится древовидное представление структуры кода. Особенностью синтаксического анализа является его зависимость от грамматики языка программирования, что делает построение корректного дерева разбора (parse tree) важной, но сложной задачей.

Метод рекурсивного спуска относится к нисходящим методам синтаксического анализа, где построение дерева разбора начинается от корня и идет к листьям. Такой подход позволяет реализовать парсер как набор взаимно вызываемых функций, каждая из которых соответствует одному нетерминалу грамматики. При этом каждое правило грамматики преобразуется в последовательность вызовов функций или проверок терминальных символов. Основным преимуществом данного метода является его простота и ясность реализации, однако он требует строго определённой LL(1)-грамматики, что может ограничивать его применимость в случае более сложных грамматик.

Отдельное внимание в работе было уделено сравнению деревьев разбора и абстрактных синтаксических деревьев (AST). Несмотря на то что parse tree точно отражает структуру грамматики, его использование не всегда удобно из-за зависимости структуры дерева от изменений в грамматике. В свою очередь, AST является упрощённым представлением логической структуры программы, игнорирующим избыточные детали грамматики, такие как пробелы или комментарии, что делает его более устойчивым к изменениям и удобным для последующей обработки.

Реализация синтаксического анализатора методом рекурсивного спуска позволила на практике применить теоретические знания о построении деревьев разбора и работе с грамматиками. Итоговая программа успешно справляется с задачей анализа и демонстрирует, как правила грамматики трансформируются в структуру дерева. Полученные навыки работы с грамматиками, деревьями разбора и абстрактными синтаксическими деревьями могут быть полезны в дальнейших исследованиях и разработке трансляторов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие. — СПб.: Питер, 2007. — 638 с.: ил.

[2] Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. — М.: ООО «И.Д. Вильямс», 2008. — 1184 с.: ил. — Парал. тит. англ.

[3] Маккиман У., Хорнинг Дж., Уортман Д. Генератор компиляторов / Пер. с англ. С.М. Круговой; Под ред. и с предисл. В.М. Савинкова. — М.: Статистика, 1980. — 527 с., ил.

[4] Modern Compiler implementation in C / Andrew W. Appel with Maia Ginsburg. — Rev. and expandeded. 544 p.: ill.

[5] Оношко Д.Е., Шостак Е.В., Марина И.М. Основы построения трансляторов языков программирования. Учебно-методическое пособие : пособие / Оношко Д.Е., Шостак Е.В., Марина И.М. – Минск : БГУИР, 2018. – 61 с.