Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления (ИУ)**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии (ИУ7)**»

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Разработка компилятора языка tinyc»

Студент группы **ИУ7-21М** _____ **В.А. Иванов**
(Подпись, дата) (И.О. Фамилия)

Руководитель _____ **А.А. Ступников**
(Подпись, дата) (И.О. Фамилия)

***2023** г.*

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
_____ И. В. Рудаков
(И.О.Фамилия)
« _____ » _____ 20 ____ г.

ЗАДАНИЕ
на выполнение курсового проекта

по дисциплине Конструирование компиляторов

Студент группы ИУ7-21М

Иванов Всеволод Алексеевич
(Фамилия, имя, отчество)

Тема курсового проекта Разработка компилятора языка tinyc

Направленность КП (учебный, исследовательский, практический, производственный, др.)
учебный
Источник тематики (кафедра, предприятие, НИР) Кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание Описать грамматику языка tinyc, расширить её поддержкой массивов с помощью грамматики языка Си. Разработать компилятор расширенного языка tinyc. В качестве фронтенда компилятора использовать утилиту ANTLR4 для преобразования исходного кода в синтаксическое дерево в соответствии с описанной грамматикой. В качестве бекенда компилятора использовать LLVM для генерации промежуточного представления на основе составленного синтаксического дерева.

Оформление курсового проекта:

Расчетно-пояснительная записка на 20-30 листах формата А4.
Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую, конструкторскую, технологическую части, заключение и список литературы.

Дата выдачи задания « 2 » марта 2023 г.

Руководитель курсового проекта _____ А.А.Ступников
(Подпись, дата) (И.О.Фамилия)

Студент _____ В.А.Иванов
(Подпись, дата) (И.О.Фамилия)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор — программа, переводящая написанный на языке программирования текст в набор машинных кодов[1].

Целью данного курсового проекта является разработка компилятора для языка программирования tinyc. В данной курсовой работе грамматика данного языка будет расширена за счёт элементов грамматики языка Си. Это делается для удволетворения требований к курсовому проекту по наличию более сложных элементов языка, чем имеющиеся в данной грамматике.

Основные задачи, которые необходимо выполнить в рамках данного проекта:

1) Провести анализ существующей грамматики языка tinyc, и расширить ее поддержкой массивов, используя грамматику языка Си.
2) Разработать лексический и ситнтаксический анализатор с использованием утилиты ANTLR4.
3) Разработать семантический анализатор для генерации промежуточного представления LLVM.
4) Провести тестирование компилятора.

1 Аналитическая часть

1.1 Составляющие компилятора

Компилятор состоит из следующих составляющих подпрограмм:

- Frontend компилятора отвечает за первичную обработку исходного кода и создание внутреннего представления программы. Он состоит из следующих частей:
  - препроцессор;
  - лексический анализатор;
  - синтаксический анализатор;
  - семантический анализатор;
  - генератор промежуточного представления;
- Middle-end компилятора занимается оптимизацией и преобразованием промежуточного представления программы.
- Backend компилятора отвечает за генерацию целевого кода, который может быть выполнен на конкретной аппаратной платформе или виртуальной машине.

В данной работе функции Middle-end и Backend компилятора будут осуществляться библиотекой LLVM, поэтому рассмотрим более подробно Frontend составляющих компилятора.

1.1.1 Препроцессор

Препроцессор компилятора - это компонент компилятора, который выполняет предварительную обработку исходного кода перед фазой фронтенда. Его задача заключается в обработке директив препроцессора и внесении соответствующих изменений в исходный код.

Препроцессор предоставляет набор директив, которые позволяют включать или исключать определенные части исходного кода, задавать макросы для замены текста и включать заголовочные файлы. Примером директив языка Си являются **include**, **define**, **pragma**. После работы препроцессора изменённый

исходный код программы подаётся на вход лексический анализатора.

В данном проекте препроцессор не используется ввиду его избыточности.

1.1.2 Лексический анализатор

Лексический анализатор выполняет первичную обработку исходного кода, разбивая его на лексемы. Лексема - минимальный элемент исходного кода. Примеры: ключевые слова, идентификаторы, операторы, константы и символы пунктуации.

Задачи лексического анализатора:

- разбиение исходного кода на лексемы;
- идентификация типов лексем;
- удаление незначащих символов;
- формирование потока токенов для синтаксического анализатора;

1.1.3 Синтаксический анализатор

Синтаксический анализатор выполняет построение синтаксического дерева из полученного потока токенов, которое представляет иерархическую структуру программы. Обычно это представление выражается в виде абстрактного синтаксического дерева (AST), где каждый внутренний узел является оператором, а дочерние его аргументами.

Задачи синтаксического анализатора:

- проверка синтаксической корректности (соответствие грамматике);
- построение синтаксического дерева;
- обработка ошибок.

Полученное представление программы в виде синтаксического дерева используется на следующем этапе.

1.1.4 Семантический анализ

Семантический анализатор выполняет проверку семантики исходного кода, включая правильное использование типов данных, правила области видимости и согласованность операций.

Задачи семантического анализатора:

- установить семантическую связь между различными частями программы;
- выявить потенциальные ошибки и несоответствия типов.

Семантический анализатор составляет таблицу символов, описывающую хранящиеся типы данных.

1.1.5 Генерация кода

Генерация кода - это фаза компиляции, в которой основываясь на синтаксическом дереве программы и системных таблиц создаётся её код.

Получение машинного кода осуществляется в два этапа.

1) Генерация промежуточного кода - относится к последней фазе frontend компилятора.
2) Генерация машинного кода - относится к middle-end и backend компилятора.

Основные этапы генерации кода включают:

- оптимизация промежуточного представления;
- выбор инструкций целевой платформы, соотвествующие промежуточному представлению;
- связывание данных с именами переменных;
- собственно генерация кода.

Результатом этого этапа является исполняемый на целевой платформе код.

1.2 ANTLR4

В качестве лексического и синтаксического анализатора будет использован ANTLR4 (ANother Tool for Language Recognition). Выбор обосновывается рядом преимуществ и особенностей данного инструмента:

- поддержка генерацию лексических и синтаксических анализаторов для широкого спектра языков программирования;
- удобный и интуитивно понятный синтаксис для описания грамматик языков программирования;
- автоматическая генерация синтаксического дерева;

- широкая и активная пользовательская база и развитое сообщество разработчиков.

1.3 LLVM

В качестве генератора кода используется LLVM (Low Level Virtual Machine). Его выбор обосновывается следующими факторами:

- Поддержка большого количества целевых платформ.
- Поддержка библиотек на различных языках (C, C++, Rust, Python и другие).
- Поддержка основных типы данных: целые числа, числа с плавающей точки различных точностей, массивы, структуры, функции.
- Автоматическая оптимизация сгенерированного промежуточного представления.
- Широкая и активная пользовательская база и развитое сообщество разработчиков.
- Имеется интерпретатор промежуточного представления.

Выводы

В данном разделе был проведён обзор основных фаз компиляции. Обоснованы выборы средств лексического и синтаксического анализа - ANTLR4 и генератора машинного кода - LLVM. Метод работы компилятора будет заключаться в генерации синтаксического дерева и генерации по нему промежуточного представления кода (LLVM IR).

2 Конструкторская часть

2.1 IDEF0

Концептуальная модель программы представлена в нотации IDEF0 на рисунке 2.1

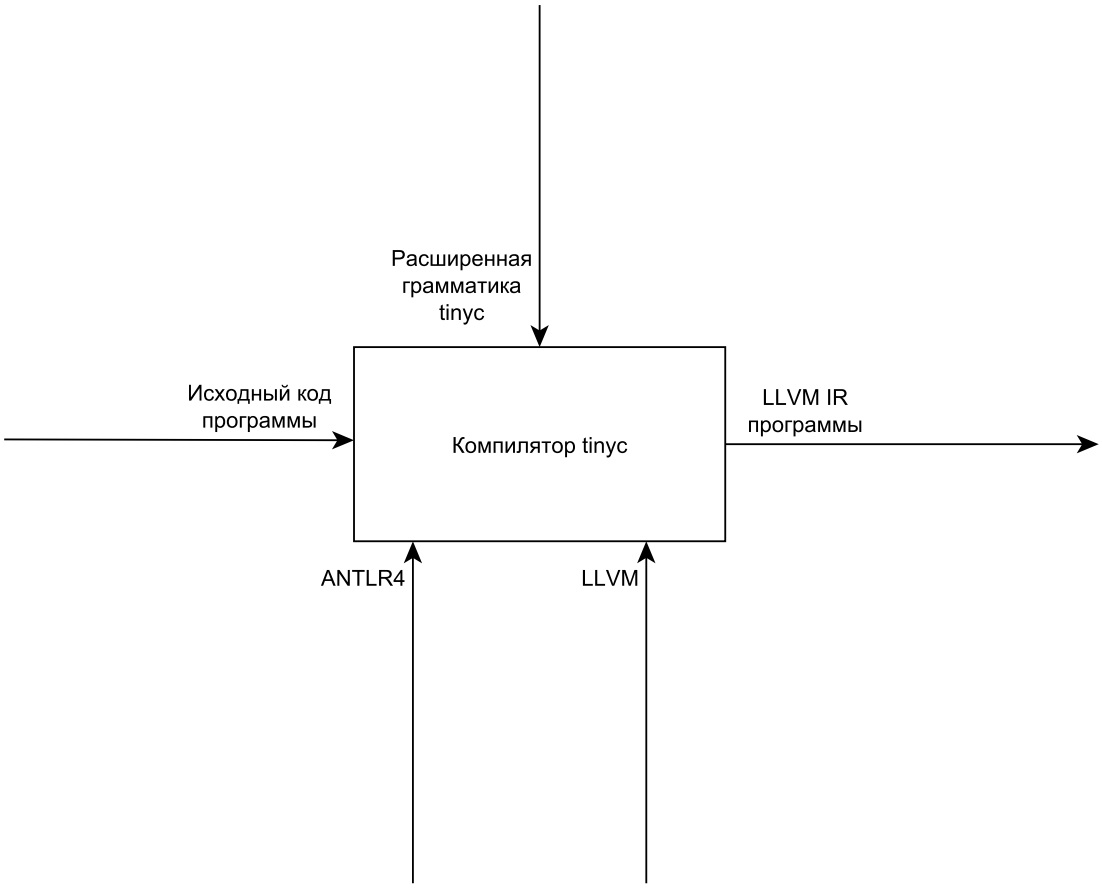

Рисунок 2.1 – Концептуальная модуль системы в нотации IDEF0.

2.2 Грамматика языка tinyc

Грамматика языка tinyc 6 является крайне упрощённым вариантом грамматики языка Си.

Листинг 1: Грамматика языка tinyc

```
1 grammar tinyc;
2 program: statement + EOF;
3 statement
4     : 'if' paren_expr statement
```

```
5     | 'if' paren_expr statement 'else' statement
6     | 'while' paren_expr statement
7     | 'do' statement 'while' paren_expr ';'
8     | '{' statement* '}'
9     | expr ';'
10    | ';'
11    ;
12 paren_expr: '(' expr ')';
13 expr: test | id_ '=' expr;
14 test: sum_ | sum_ '<' sum_;
15 sum_: term | sum_ '+' term | sum_ '-' term;
16 term: id_ | integer | paren_expr;
17
18 id_: STRING;
19 integer: INT;
20 STRING: [a-z]+;
21 INT: [0-9]+;
22 WS: [ \r\n\t] -> skip;
```

Так как эта грамматика не удовлетворяла требованиям курсовой работы, она была дополнена элементами грамматики Си. Были добавлены следующие элементы синтаксиса:

- недостающие арифметические операции с учётом приоритетов действий;
- объявления и инициализация переменных;
- поддержка многомерных статических массивов;
- комментарии.

Полная дополненная грамматика приведена в Приложении А.

2.3 Обход синтаксического дерева

Исходная программа преобразовывается в синтаксическое дерево при помощи кода, сгенерированного ANTLR4 для описанной грамматики. Обход всех узлов данного представления позволяет сгенерировать LLVM IR.

Рассмотрим синтаксическое дерево на примере изображения 2.2.

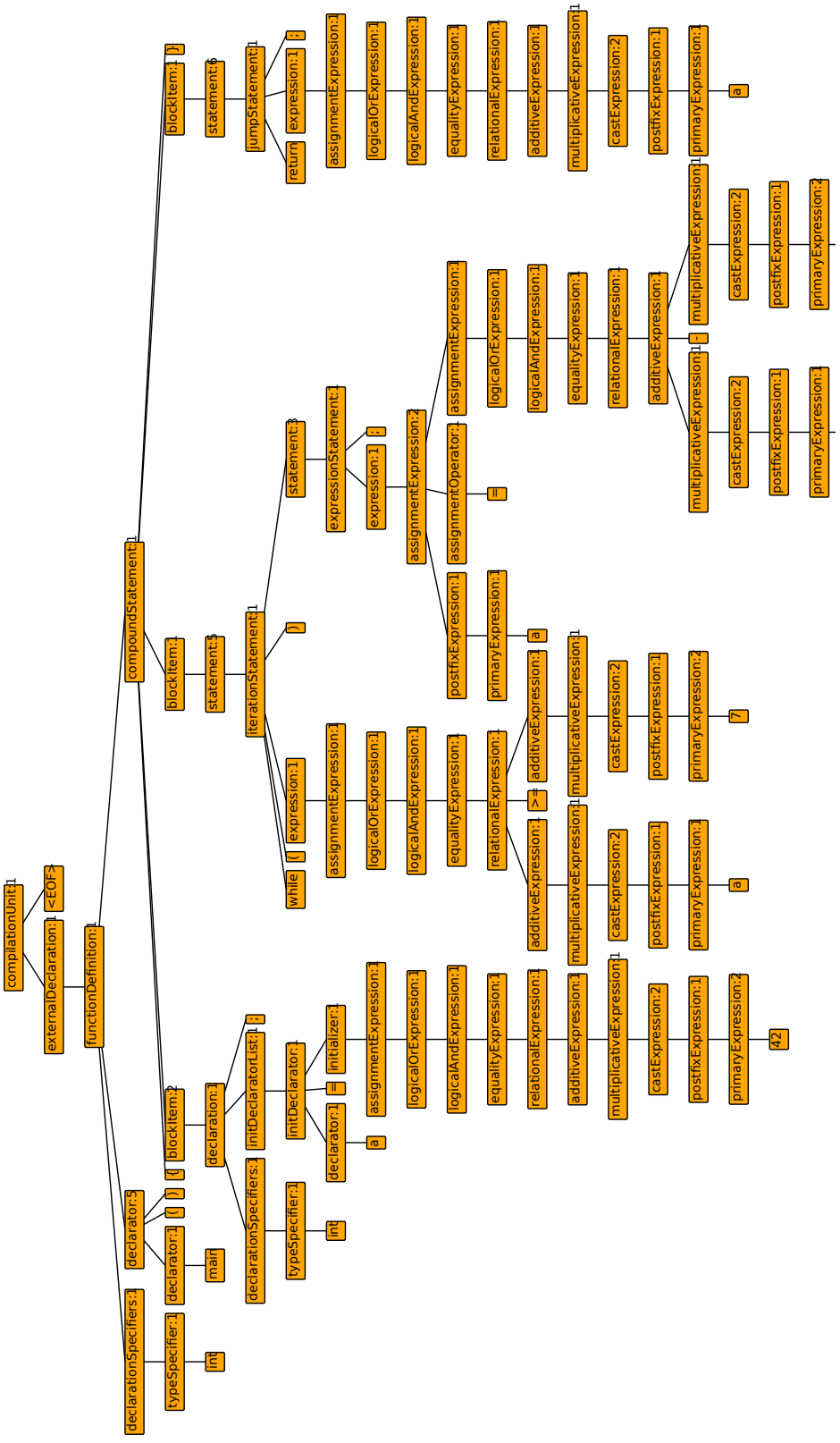

Рисунок 2.2 – Пример визуализации синтаксического дерева.

Данная визуализация получена использованием утилиты **antlr4-parse** для следующей программы, вычисляющей остаток от деления числа на 7.

Листинг 2: Пример программы (остаток от деления на 7)

```
int main() {
    int a = 42;
    while(a >= 7)
        a = a - 7;
    return a;
}
```

2.4 Генерация LLVM IR

Сгенерировать промежуточное представление LLVM можно путём обхода всех узлов синтаксического дерева. Каждый узел может создавать новые инструкции, блоки, функции и т.п. в зависимости от его типа и дочерних узлов. Рассмотрим пример обработки узла **iterationStatement**, грамматика которого показана в выражении 1.

iterationStatement: While ’(’ expression ’)’ statement; (1)

Пример LLVM IR представления, сгенерированного данным алгоритмом по вышеупомянутой программе приведён в листинге 2.4.

Листинг 3: Пример LLVM IR (остаток от деления на 7)

```
define i32 @main() {
main:
    %0 = alloca i32
    store i32 42, i32* %0
    br label %cond
cond:
    %1 = load i32, i32* %0
    %2 = icmp sge i32 %1, 7
    br i1 %2, label %body, label %next
body:
    %3 = load i32, i32* %0
    %4 = sub i32 %3, 7
    store i32 %4, i32* %0
```

```
14     br label %cond
15 next:
16     %5 = load i32, i32* %0
17     ret i32 %5
18 }
```

На рисунке 2.3 приведена схема алгоритма генерации кода для цикла while.

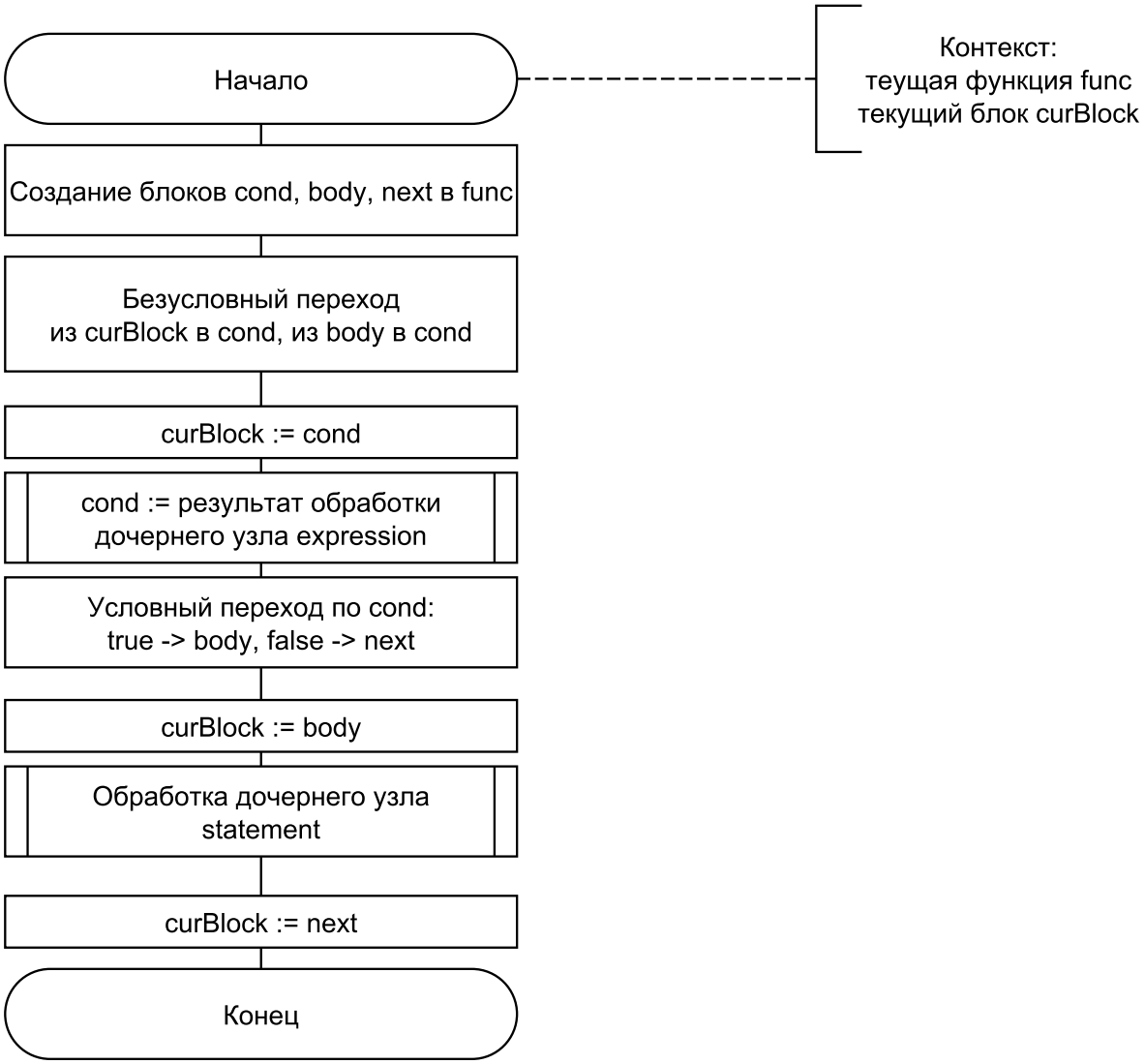

Рисунок 2.3 – Схема алгоритма обработки узла iterationStatement.

Вывод

В данном разделе была предоставлена концептуальная модель метода компиляции в нотации диаграммы IDEF0. Приведена грамматика tinyc, описаны предпринятые расширения. Описаны работы frontend часть разрабатываемого компилятора.

3 Технологическая часть

3.1 Обоснование средств программной реализации

В качестве языка программирования для написания компилятора был выбран Go[2], ввиду следующих причин.

- Имеется личный опыт работы с данным языком на других курсовых проектах.
- ANTLR4[3] и LLVM[4] поддерживают библиотеки на Go. По моей субъективной оценке в данном языке их использование будет наиболее удобным.

3.2 Описание программы

Обход синтаксического дерева в бибиотеке antlr4-go можно осуществить с помощью паттерна Walker или Visitor. Первый состоит в описании методов enter и exit для каждого типа узла дерева. Второй - в описании методов visit для каждого типа узла. Был выбран обход с помощью Visitor, так как он позволяет контролировать порядок вызова обработки дочерних узлов, а также поддерживает возврат значения.

LLVM IR формируется при помощи структуры Module, предоставляющей интерфейс для построения промежуточного представления. Эта и другие переменные контекста (текущий блок, текущая функция, области видимости переменных) хранятся в Visitor и используются по мере обхода дерева.

В результате обхода дерева получается заполненная структура LLVM модуля, которая записывается в текстовом формате в .ll файл. После чего компилируется с помощью утилит **clang** и **lld-link**.

3.3 Тестирование программы

Для тестирования программы были написаны программы на языке Си, удовлетворяющие ранее сформированной грамматике. Тестирование производится в три этапа.

1) Тестовая программа компилируется при помощи написанного модуля.

2) Тестовая программа компилируется при помощи gcc.

3) Полученные исполняемые файлы запускаются. Сравниваются коды возврата данных программ.

Так как грамматика не предусматривает возможность использования функций ввода/вывода из стандартных библиотек Си, единственным путём проверки результатов вычислений остаётся код возврата программмы. Стоит отметить, что такое использование кода возврата не соответствует его основной функции - передачи кода ошибки, с которым завершилась программа. Однако, в условиях данного проекта такое использование возвращаемого значения в целях демонстрации работы программы было сочтено приемлимым.

3.4 Пример работы программы

Для примера работы программы используется один из тестовых примеров - вычисление n-го числа Фибоначчи с использованием статического массива. В следующих листингах 3.4 3.4.

Листинг 4: Пример программы

```
int main() {
    int n = 10;

    if (n > 100) {
        return -1;
    }

    int fib[100];
    fib[0] = 0;
    fib[1] = 1;

    int i = 2;
    while (i <= n) {
        fib[i] = fib[i-1] + fib[i-2];
        i = i + 1;
    }

    return fib[n];
}
```

Листинг 5: Пример LLVM IR

```
define i32 @main() {
main:
    %0 = alloca i32
    store i32 10, i32* %0
    %1 = load i32, i32* %0
    %2 = icmp sgt i32 %1, 100
    %3 = alloca [100 x i32]
    %4 = alloca i32
    br i1 %2, label %if-1, label %else-1

if-1:
    %5 = sub i32 0, 1
    ret i32 %5

else-1:
    br label %main-1

main-1:
    %6 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 0
    store i32 0, i32* %6
    %7 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 1
    store i32 1, i32* %7
    store i32 2, i32* %4
    br label %while.cond-4

while.cond-4:
    %8 = load i32, i32* %4
    %9 = load i32, i32* %0
    %10 = icmp sle i32 %8, %9
    br i1 %10, label %while.body-4, label %main-4

while.body-4:
    %11 = load i32, i32* %4
    %12 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 %11
    %13 = load i32, i32* %4
    %14 = sub i32 %13, 1
    %15 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 %14
    %16 = load i32, i32* %4
    %17 = sub i32 %16, 2
```

```
40      %18 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 %17
41      %19 = load i32, i32* %15
42      %20 = load i32, i32* %18
43      %21 = add i32 %19, %20
44      store i32 %21, i32* %12
45      %22 = load i32, i32* %4
46      %23 = add i32 %22, 1
47      store i32 %23, i32* %4
48      br label %while.cond-4
49
50  main-4:
51      %24 = load i32, i32* %0
52      %25 = getelementptr [100 x i32], [100 x i32]* %3, i32 0, i32 %24
53      %26 = load i32, i32* %25
54      ret i32 %26
55  }
56
57  define i32 @mainCRTStartup() {
58  0:
59      %1 = call i32 @main()
60      ret i32 %1
61  }
```

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19781-83 // Вычислительная техника. Терминология: Справочное пособие. Выпуск 1 / Рецензент канд. техн. наук Ю. П. Селиванов. — М.: Издательство стандартов, 1989. — 168 с. — 55 000 экз. — ISBN 5-7050-0155-X.;

2. The Go Programming Language [Электронный ресурс]: официальный сайт языка программирования Go. - Режим доступа: https://go.dev/ (дата обращения: 25.06.2023).

3. ANTLR4 Go Runtime Module Repo [Электронный ресурс]: репозиторий модуля ANTLR4 Go Runtime в GitHub. - Режим доступа: <https://pkg.go.dev/github.com/antlr4-go/antlr/v4@v4.13.0> (дата обращения: 25.06.2023).

4. LLVM [Электронный ресурс]: репозиторий LLVM в GitHub. - Режим доступа: <https://pkg.go.dev/github.com/llir/llvm@v0.3.6> (дата обращения: 25.06.2023).

ПРИЛОЖЕНИЕ А

Листинг 6: Дополненная грамматика языка tinyc

```
grammar tinyc;
primaryExpression
    :   Identifier
    |   Constant
    |   StringLiteral+
    |   '(' expression ')'
    ;

genericAssocList
    :   genericAssociation (',' genericAssociation)*
    ;

genericAssociation
    :   typeName ':' assignmentExpression
    ;

postfixExpression
    :
    (   primaryExpression
    |   '(' typeName ')' '{' initializerList ','? '}'
    )
    ('[' expression ']'
    | '(' argumentExpressionList? ')'
    )*
    ;

argumentExpressionList
    :   assignmentExpression (',' assignmentExpression)*
    ;

unaryExpression: unaryOperator castExpression;

unaryOperator: '+' | '-' | '!';

castExpression
    :   '(' typeName ')' castExpression
    |   postfixExpression
    |   unaryExpression
    ;

multiplicativeExpression
    :   castExpression (('*'|'/'|'%') castExpression)*
    ;

additiveExpression
    :   multiplicativeExpression (('+'|'-') multiplicativeExpression)*
    ;

relationalExpression
    :   additiveExpression (('<'|'>'|'<='|'>=') additiveExpression)*
    ;

equalityExpression
    :   relationalExpression (('=='| '!=') relationalExpression)*
    ;

logicalAndExpression
    :   equalityExpression ('&&' equalityExpression)*
    ;

logicalOrExpression
    :   logicalAndExpression ( '||' logicalAndExpression)*
```

```
63      ;
64
65  assignmentExpression
66      :   logicalOrExpression
67      |   postfixExpression assignmentOperator assignmentExpression
68      ;
69
70  assignmentOperator: '=';
71
72  expression
73      :   assignmentExpression (',' assignmentExpression)*
74      ;
75
76  declaration
77      :   declarationSpecifiers initDeclaratorList? ';'
78      ;
79
80  declarationSpecifiers
81      :   typeSpecifier+
82      ;
83
84  initDeclaratorList
85      :   initDeclarator (',' initDeclarator)*
86      ;
87
88  initDeclarator
89      :   declarator ('=' initializer)?
90      ;
91
92  typeSpecifier
93      :   'char'
94      |   'int'
95      |   'float'
96      ;
97
98  specifierQualifierList
99      :   typeSpecifier specifierQualifierList?
100     ;
101
102 structDeclaratorList
103     :   structDeclarator (',' structDeclarator)*
104     ;
105
106 structDeclarator
107     :   declarator
108     |   declarator? ':' logicalOrExpression
109     ;
110
111 declarator
112     :   Identifier
113     |   '(' declarator ')'
114     |   declarator '[' assignmentExpression? ']'
115     |   declarator '(' parameterTypeList ')'
116     |   declarator '(' identifierList? ')'
117     ;
118
119
120 gccAttribute
121     :   ~(',' | '(' | ')') // relaxed def for "identifier or reserved word"
122         ('(' argumentExpressionList? ')')?
123     ;
124
125 nestedParenthesesBlock
126     :   (   ~('(' | ')')
127         |   '(' nestedParenthesesBlock ')'
128         )*
129     ;
130
131
```

```
parameterTypeList
    :   parameterList (',' '...')?
    ;

parameterList
    :   parameterDeclaration (',' parameterDeclaration)*
    ;

parameterDeclaration
    :   declarationSpecifiers (declarator | abstractDeclarator?)
    ;

identifierList
    :   Identifier (',' Identifier)*
    ;

typeName
    :   specifierQualifierList abstractDeclarator?
    ;

abstractDeclarator
    :   '(' abstractDeclarator ')'
    |   '[' assignmentExpression? ']'
    |   '[' '*' ']'
    |   '(' parameterTypeList? ')'
    |   abstractDeclarator '[' assignmentExpression? ']'
    |   abstractDeclarator '[' '*' ']'
    |   abstractDeclarator '(' parameterTypeList? ')'
    ;

initializer
    :   assignmentExpression
    |   '{' initializerList ','? '}'
    ;

initializerList
    :   designation? initializer (',' designation? initializer)*
    ;

designation
    :   designatorList '='
    ;

designatorList
    :   designator+
    ;

designator
    :   '[' logicalOrExpression ']'
    |   '.' Identifier
    ;

statement
    :   labeledStatement
    |   compoundStatement
    |   expressionStatement
    |   selectionStatement
    |   iterationStatement
    |   jumpStatement
    ;

labeledStatement
    :   Identifier ':' statement
    ;

compoundStatement
    :   '{' blockItem* '}'
    ;

```

```
blockItem
    :   statement
    |   declaration
    ;

expressionStatement
    :   expression? ';'
    ;

selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
    ;

iterationStatement
    :   While '(' expression ')' statement
    ;

jumpStatement: 'return' expression ';' ;

compilationUnit
    :   externalDeclaration+ EOF
    ;

externalDeclaration
    :   functionDefinition
    |   declaration
    |   ';'
    ;

functionDefinition
    :   declarationSpecifiers? declarator declarationList? compoundStatement
    ;

declarationList
    :   declaration+
    ;

Auto : 'auto';
Break : 'break';
Char : 'char';
Const : 'const';
Continue : 'continue';
Else : 'else';
Extern : 'extern';
Float : 'float';
Goto : 'goto';
If : 'if';
Inline : 'inline';
Int : 'int';
Register : 'register';
Restrict : 'restrict';
Return : 'return';
Sizeof : 'sizeof';
Struct : 'struct';
Switch : 'switch';
Typedef : 'typedef';
Union : 'union';
While : 'while';

Noreturn : '_Noreturn';

LeftParen : '(';
RightParen : ')';
LeftBracket : '[';
RightBracket : ']';
LeftBrace : '{';
RightBrace : '}';

Less : '<';
```

```
LessEqual : '<=';
Greater : '>';
GreaterEqual : '>=';
LeftShift : '<<';
RightShift : '>>';

Plus : '+';
PlusPlus : '++';
Minus : '-';
MinusMinus : '--';
Star : '*';
Div : '/';
Mod : '%';

And : '&';
Or : '|';
AndAnd : '&&';
OrOr : '||';
Caret : '^';
Not : '!';
Tilde : '~';

Question : '?';
Colon : ':';
Semi : ';';
Comma : ',';

Assign : '=';

Equal : '==';
NotEqual : '!=';

Arrow : '->';
Dot : '.';
Ellipsis : '...';

Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

fragment
IdentifierNondigit
    :   Nondigit
    |   UniversalCharacterName
    ;

fragment
Nondigit
    :   [a-zA-Z_]
    ;

fragment
Digit
    :   [0-9]
    ;

fragment
UniversalCharacterName
    :   '\\u' HexQuad
    |   '\\U' HexQuad HexQuad
    ;

fragment
HexQuad
    :   HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
    ;
```

```
339
340 Constant
341     :   IntegerConstant
342     |   FloatingConstant
343     |   CharacterConstant
344     ;
345
346 fragment
347 IntegerConstant
348     :   DecimalConstant
349     |   OctalConstant
350     |   HexadecimalConstant
351     |   BinaryConstant
352     ;
353
354 fragment
355 BinaryConstant
356     :   '0' [bB] [0-1]+
357     ;
358
359 fragment
360 DecimalConstant
361     :   NonzeroDigit Digit*
362     ;
363
364 fragment
365 OctalConstant
366     :   '0' OctalDigit*
367     ;
368
369 fragment
370 HexadecimalConstant
371     :   HexadecimalPrefix HexadecimalDigit+
372     ;
373
374 fragment
375 HexadecimalPrefix
376     :   '0' [xX]
377     ;
378
379 fragment
380 NonzeroDigit
381     :   [1-9]
382     ;
383
384 fragment
385 OctalDigit
386     :   [0-7]
387     ;
388
389 fragment
390 HexadecimalDigit
391     :   [0-9a-fA-F]
392     ;
393
394 fragment
395 FloatingConstant
396     :   DecimalFloatingConstant
397     |   HexadecimalFloatingConstant
398     ;
399
400 fragment
401 DecimalFloatingConstant
402     :   FractionalConstant ExponentPart? FloatingSuffix?
403     |   DigitSequence ExponentPart FloatingSuffix?
404     ;
405
406 fragment
407 HexadecimalFloatingConstant
```

```
408     :   HexadecimalPrefix (HexadecimalFractionalConstant | HexadecimalDigitSequence) BinaryExponentPart FloatingSuffix?
409     ;
410
411 fragment
412 FractionalConstant
413     :   DigitSequence? '.' DigitSequence
414     |   DigitSequence '.'
415     ;
416
417 fragment
418 ExponentPart
419     :   [eE] Sign? DigitSequence
420     ;
421
422 fragment
423 Sign
424     :   [+-]
425     ;
426
427 DigitSequence
428     :   Digit+
429     ;
430
431 fragment
432 HexadecimalFractionalConstant
433     :   HexadecimalDigitSequence? '.' HexadecimalDigitSequence
434     |   HexadecimalDigitSequence '.'
435     ;
436
437 fragment
438 BinaryExponentPart
439     :   [pP] Sign? DigitSequence
440     ;
441
442 fragment
443 HexadecimalDigitSequence
444     :   HexadecimalDigit+
445     ;
446
447 fragment
448 FloatingSuffix
449     :   [flFL]
450     ;
451
452 fragment
453 CharacterConstant
454     :   '\'' CCharSequence '\''
455     |   'L\'' CCharSequence '\''
456     |   'u\'' CCharSequence '\''
457     |   'U\'' CCharSequence '\''
458     ;
459
460 fragment
461 CCharSequence
462     :   CChar+
463     ;
464
465 fragment
466 CChar
467     :   ~['\\\r\n]
468     |   EscapeSequence
469     ;
470
471 fragment
472 EscapeSequence
473     :   SimpleEscapeSequence
474     |   OctalEscapeSequence
475     |   HexadecimalEscapeSequence
476     |   UniversalCharacterName
```

```
477     ;
478
479 fragment
480 SimpleEscapeSequence
481     :   '\\' ['"?abfnrtv\\]
482     ;
483
484 fragment
485 OctalEscapeSequence
486     :   '\\' OctalDigit OctalDigit? OctalDigit?
487     ;
488
489 fragment
490 HexadecimalEscapeSequence
491     :   '\\x' HexadecimalDigit+
492     ;
493
494 StringLiteral
495     :   EncodingPrefix? '"' SCharSequence? '"'
496     ;
497
498 fragment
499 EncodingPrefix
500     :   'u8'
501     |   'u'
502     |   'U'
503     |   'L'
504     ;
505
506 fragment
507 SCharSequence
508     :   SChar+
509     ;
510
511 fragment
512 SChar
513     :   ~["\\\r\n]
514     |   EscapeSequence
515     |   '\\\n'   // Added line
516     |   '\\\r\n' // Added line
517     ;
518
519 ComplexDefine
520     :   '#' Whitespace? 'define'  ~[#\r\n]*
521         -> skip
522     ;
523
524 IncludeDirective
525     :   '#' Whitespace? 'include' Whitespace? ('"' ~[\r\n]* '"' | '<' ~[\r\n]* '>' ) Whitespace? Newline
526         -> skip
527     ;
528
529 AsmBlock
530     :   'asm' ~'{'* '{' ~'}'* '}'
531     -> skip
532     ;
533
534 // ignore the lines generated by c preprocessor
535 // sample line : '#line 1 "/home/dm/files/dk1.h" 1'
536 LineAfterPreprocessing
537     :   '#line' Whitespace* ~[\r\n]*
538         -> skip
539     ;
540
541 LineDirective
542     :   '#' Whitespace? DecimalConstant Whitespace? StringLiteral ~[\r\n]*
543         -> skip
544     ;
545
```

```
546 PragmaDirective
547     :   '#' Whitespace? 'pragma' Whitespace ~[\r\n]*
548         -> skip
549     ;
550
551 Whitespace
552     :   [ \t]+
553         -> skip
554     ;
555
556 Newline
557     :   (   '\r' '\n'?
558         |   '\n'
559         )
560         -> skip
561     ;
562
563 BlockComment
564     :   '/*' .*? '*/'
565         -> skip
566     ;
567
568 LineComment
569     :   '//' ~[\r\n]*
570         -> skip
571     ;
```