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)**»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

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 (АСД), 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

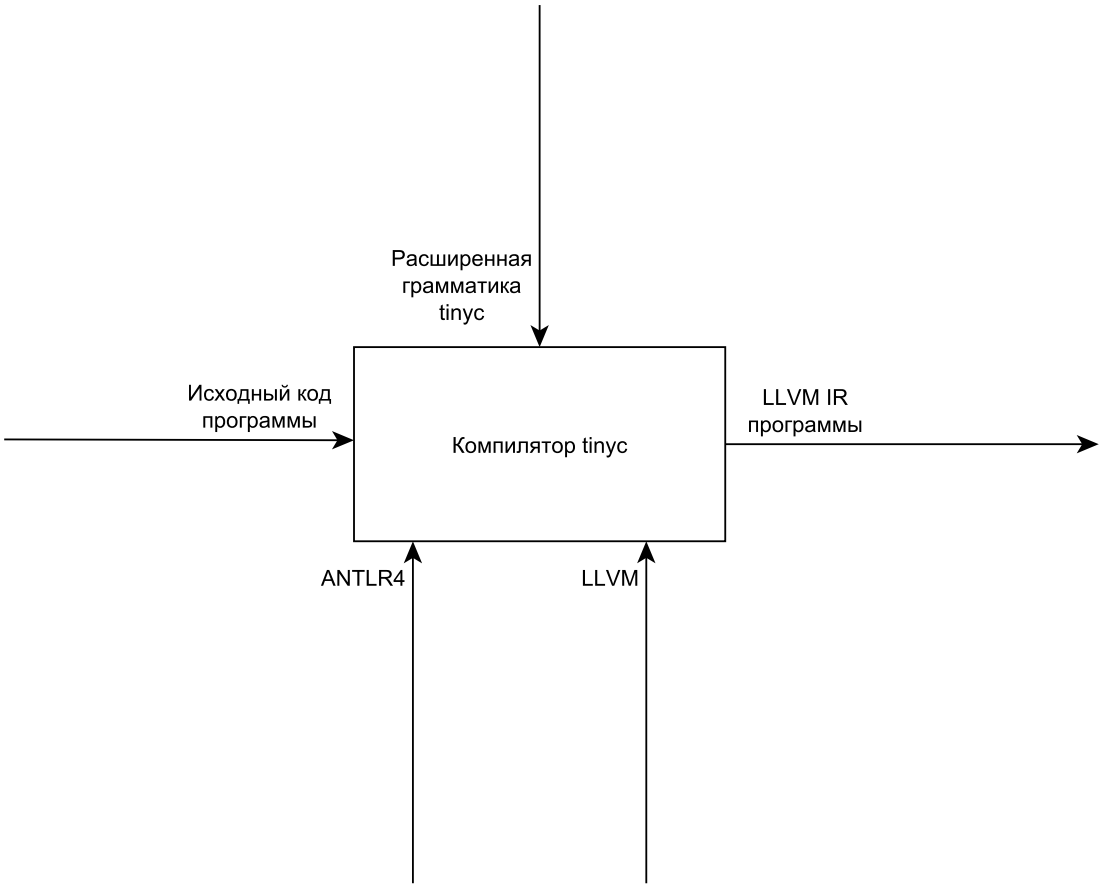

Рисунок 2.1 – Концептуальная модуль системы в нотации IDEF0.

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
```

```
    | 'if' paren_expr statement 'else' statement
    | 'while' paren_expr statement
    | 'do' statement 'while' paren_expr ';'
    | '{' statement* '}'
    | expr ';'
    | ';'
    ;
paren_expr: '(' expr ')';
expr: test | id_ '=' expr;
test: sum_ | sum_ '<' sum_;
sum_: term | sum_ '+' term | sum_ '-' term;
term: id_ | integer | paren_expr;

id_: STRING;
integer: INT;
STRING: [a-z]+;
INT: [0-9]+;
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 из Приложения Б. 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
    br label %cond
next:
    %5 = load i32, i32* %0
```

```
17      ret i32 %5
18  }
```

На рисунке 2.2 приведена схема алгоритма генерации кода для цикла while.

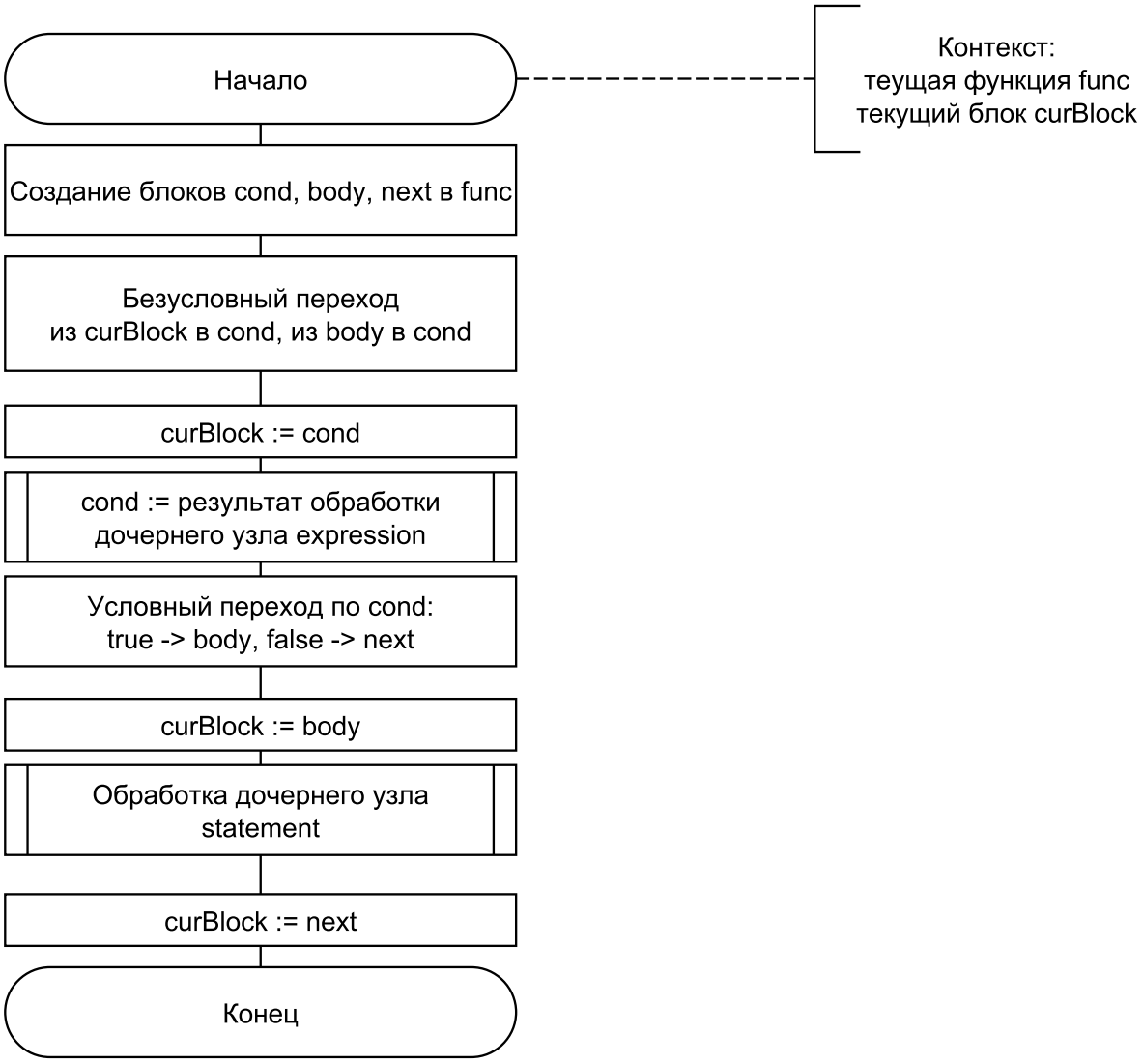

Рисунок 2.2 – Схема алгоритма обработки узла iterationStatement.

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 ')'
    ;
postfixExpression: primaryExpression ('[' expression ']' | funcCall)*;
funcCall: '(' (assignmentExpression (',' assignmentExpression)*)? ')';
unaryExpression: unaryOperator castExpression;
unaryOperator: '+' | '-' | '!';
castExpression
    :   postfixExpression
    |   unaryExpression
    ;
multiplicativeExpression: castExpression (('*'|'/'|'%') castExpression)*;
additiveExpression: multiplicativeExpression (('+'|'-') multiplicativeExpression)*;
relationalExpression:   additiveExpression (('<'|'>'|'<='|'>=') additiveExpression)*;
equalityExpression:   relationalExpression (('=='| '!=') relationalExpression)*;
logicalAndExpression:   equalityExpression ('&&' equalityExpression)*;
logicalOrExpression:   logicalAndExpression ( '||' logicalAndExpression)*;
assignmentExpression
    :   logicalOrExpression
    |   postfixExpression assignmentOperator assignmentExpression
    ;
assignmentOperator: '=';
expression:   assignmentExpression (',' assignmentExpression)*;
declaration:   declarationSpecifiers initDeclaratorList? ';';
declarationSpecifiers:   typeSpecifier+;
initDeclaratorList:   initDeclarator (',' initDeclarator)*;
initDeclarator:   declarator ('=' initializer)?;
typeSpecifier
    :   'char'
    |   'int'
    |   'float'
    ;
structDeclaratorList:   structDeclarator (',' structDeclarator)*;
structDeclarator
    :   declarator
    |   declarator? ':' logicalOrExpression
    ;
declarator
    :   Identifier
    |   '(' declarator ')'
    |   declarator '[' assignmentExpression? ']'
    |   declarator '(' identifierList? ')'
    ;

parameterList
    :   parameterDeclaration (',' parameterDeclaration)*
    |
    ;
parameterDeclaration:   declarationSpecifiers declarator;
identifierList:   Identifier (',' Identifier)*;
initializer
    :   assignmentExpression
    |   '{' initializerList ','? '}'
    ;
initializerList:   designation? initializer (',' designation? initializer)*;
designation:   designatorList '=';
designatorList:   designator+;
designator
```

```
63      :   '[' logicalOrExpression ']'
64      |   '.' Identifier
65      ;
66  statement
67      :   labeledStatement
68      |   compoundStatement
69      |   expressionStatement
70      |   selectionStatement
71      |   iterationStatement
72      |   jumpStatement
73      ;
74  labeledStatement:   Identifier ':' statement;
75  compoundStatement
76      :   '{' blockItem* '}'
77      ;
78  blockItem
79      :   statement
80      |   declaration
81      ;
82  expressionStatement:   expression? ';';
83
84  selectionStatement
85      :   'if' '(' expression ')' statement ('else' statement)?
86      ;
87
88  iterationStatement
89      :   While '(' expression ')' statement
90      ;
91
92  jumpStatement: 'return' expression ';' ;
93
94  compilationUnit
95      :   externalDeclaration+ EOF
96      ;
97
98  externalDeclaration
99      :   functionDefinition
100     |   declaration
101     |   ';' // stray ;
102     ;
103
104 functionDefinition
105     :   declarationSpecifiers? Identifier '(' parameterList ')' declarationList? compoundStatement
106     ;
107
108 declarationList
109     :   declaration+
110     ;
111
112 Char : 'char';
113 Const : 'const';
114 Else : 'else';
115 Float : 'float';
116 If : 'if';
117 Int : 'int';
118 Return : 'return';
119 While : 'while';
120 LeftParen : '(';
121 RightParen : ')';
122 LeftBracket : '[';
123 RightBracket : ']';
124 LeftBrace : '{';
125 RightBrace : '}';
126 Less : '<';
127 LessEqual : '<=';
128 Greater : '>';
129 GreaterEqual : '>=';
130 Plus : '+';
131 Minus : '-';
```

```
132 Star : '*';
133 Div : '/';
134 Mod : '%';
135 And : '&';
136 Or : '|';
137 AndAnd : '&&';
138 OrOr : '||';
139 Caret : '^';
140 Not : '!';
141 Tilde : '~';
142 Question : '?';
143 Colon : ':';
144 Semi : ';';
145 Comma : ',';
146 Assign : '=';
147 Equal : '==';
148 NotEqual : '!=';
149
150 Identifier
151     :   IdentifierNondigit
152         (   IdentifierNondigit
153         |   Digit
154         )*
155     ;
156
157 fragment
158 IdentifierNondigit
159     :   Nondigit
160     |   UniversalCharacterName
161     //|   // other implementation-defined characters...
162     ;
163
164 fragment
165 Nondigit
166     :   [a-zA-Z_]
167     ;
168
169 fragment
170 Digit
171     :   [0-9]
172     ;
173
174 fragment
175 UniversalCharacterName
176     :   '\\u' HexQuad
177     |   '\\U' HexQuad HexQuad
178     ;
179
180 fragment
181 HexQuad
182     :   HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
183     ;
184
185 Constant
186     :   IntegerConstant
187     |   FloatingConstant
188     |   CharacterConstant
189     ;
190
191 fragment
192 IntegerConstant
193     :   DecimalConstant
194     |   OctalConstant
195     |   HexadecimalConstant
196     |   BinaryConstant
197     ;
198
199 fragment
200 BinaryConstant
```

```
201     :   '0' [bB] [0-1]+
202     ;
203
204 fragment
205 DecimalConstant
206     :   NonzeroDigit Digit*
207     ;
208
209 fragment
210 OctalConstant
211     :   '0' OctalDigit*
212     ;
213
214 fragment
215 HexadecimalConstant
216     :   HexadecimalPrefix HexadecimalDigit+
217     ;
218
219 fragment
220 HexadecimalPrefix
221     :   '0' [xX]
222     ;
223
224 fragment
225 NonzeroDigit
226     :   [1-9]
227     ;
228
229 fragment
230 OctalDigit
231     :   [0-7]
232     ;
233
234 fragment
235 HexadecimalDigit
236     :   [0-9a-fA-F]
237     ;
238
239 fragment
240 FloatingConstant
241     :   DecimalFloatingConstant
242     |   HexadecimalFloatingConstant
243     ;
244
245 fragment
246 DecimalFloatingConstant
247     :   FractionalConstant ExponentPart? FloatingSuffix?
248     |   DigitSequence ExponentPart FloatingSuffix?
249     ;
250
251 fragment
252 HexadecimalFloatingConstant
253     :   HexadecimalPrefix (HexadecimalFractionalConstant | HexadecimalDigitSequence) BinaryExponentPart FloatingSuffix?
254     ;
255
256 fragment
257 FractionalConstant
258     :   DigitSequence? '.' DigitSequence
259     |   DigitSequence '.'
260     ;
261
262 fragment
263 ExponentPart
264     :   [eE] Sign? DigitSequence
265     ;
266
267 fragment
268 Sign
269     :   [+-]
```

```
270     ;
271
272 DigitSequence
273     :   Digit+
274     ;
275
276 fragment
277 HexadecimalFractionalConstant
278     :   HexadecimalDigitSequence? '.' HexadecimalDigitSequence
279     |   HexadecimalDigitSequence '.'
280     ;
281
282 fragment
283 BinaryExponentPart
284     :   [pP] Sign? DigitSequence
285     ;
286
287 fragment
288 HexadecimalDigitSequence
289     :   HexadecimalDigit+
290     ;
291
292 fragment
293 FloatingSuffix
294     :   [flFL]
295     ;
296
297 fragment
298 CharacterConstant:   '\'' CCharSequence '\'';
299
300 fragment
301 CCharSequence
302     :   CChar+
303     ;
304
305 fragment
306 CChar
307     :   ~['\\\r\n]
308     |   EscapeSequence
309     ;
310
311 fragment
312 EscapeSequence
313     :   SimpleEscapeSequence
314     |   OctalEscapeSequence
315     |   HexadecimalEscapeSequence
316     |   UniversalCharacterName
317     ;
318
319 fragment
320 SimpleEscapeSequence
321     :   '\\' ['"?abfnrtv\\]
322     ;
323
324 fragment
325 OctalEscapeSequence
326     :   '\\' OctalDigit OctalDigit? OctalDigit?
327     ;
328
329 fragment
330 HexadecimalEscapeSequence
331     :   '\\x' HexadecimalDigit+
332     ;
333
334 StringLiteral
335     :   '"' SCharSequence? '"'
336     ;
337
338 fragment
```

```
339 SCharSequence
340     :   SChar+
341     ;
342
343 fragment
344 SChar
345     :   ~["\\\r\n]
346     |   EscapeSequence
347     |   '\\\n'   // Added line
348     |   '\\\r\n' // Added line
349     ;
350
351
352 LineDirective
353     :   '#' Whitespace? DecimalConstant Whitespace? StringLiteral ~[\r\n]*
354         -> skip
355     ;
356
357 Whitespace
358     :   [ \t]+
359         -> skip
360     ;
361
362 Newline
363     :   (   '\r' '\n'?
364         |   '\n'
365         )
366         -> skip
367     ;
368
369 BlockComment
370     :   '/*' .*? '*/'
371         -> skip
372     ;
373
374 LineComment
375     :   '//' ~[\r\n]*
376         -> skip
377     ;
```

ПРИЛОЖЕНИЕ Б

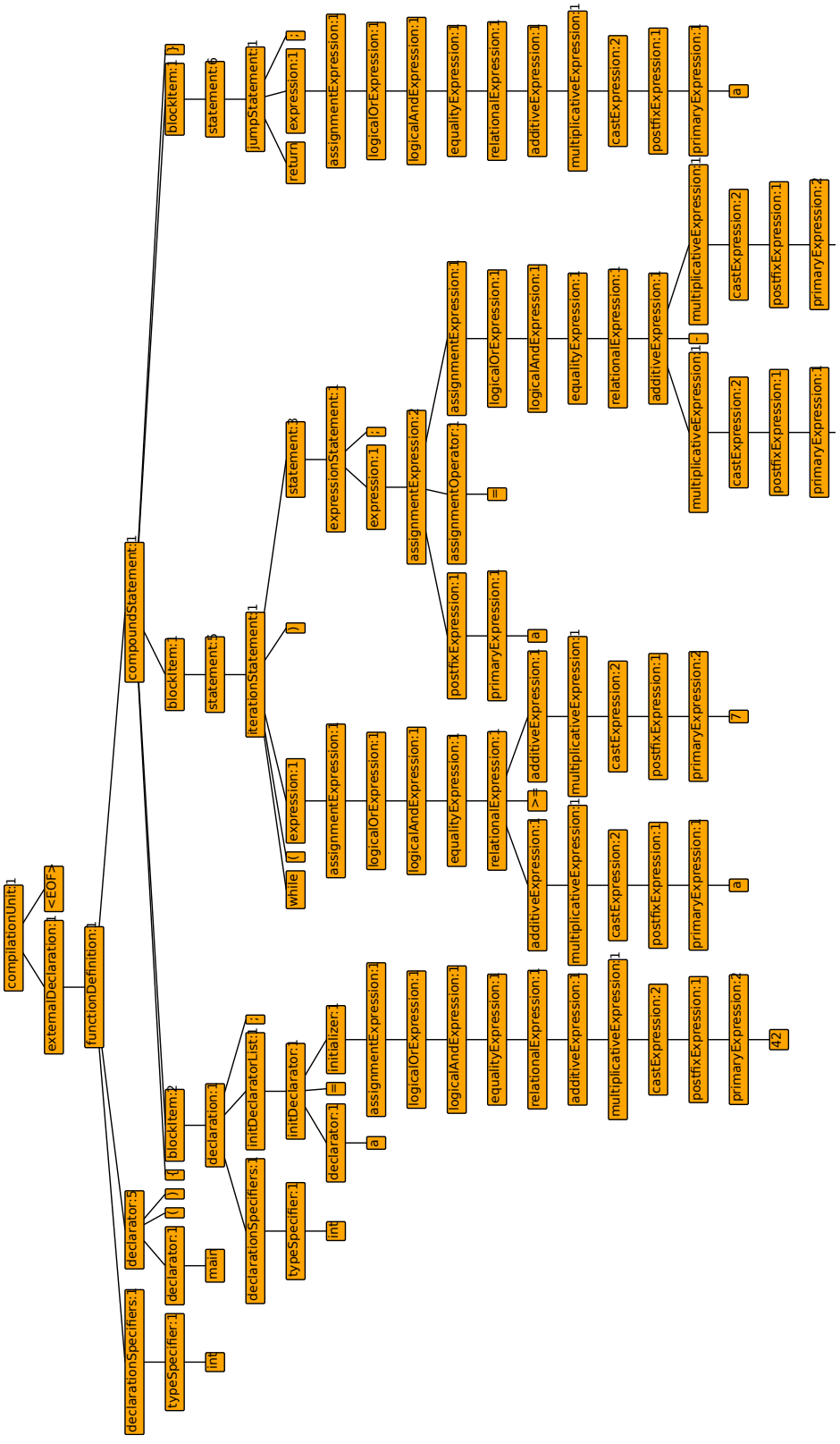

Рисунок 3.1 – Пример визуализации синтаксического дерева.