СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор является программой, которая переводит текст, написанный на языке программирования в машинный язык. Целью данной работы является написание компилятора для языка Clojure на языке C#. Для достижения поставленной цели необходимо выполнить ряд задач:

- проанализировать предметную область;
- разработать блок лексического и синтаксического анализа с явным построением дерева разбора для заданного исходного кода языка Clojure;
- разработать блок семантического анализа для составления символьных таблиц.

1. Аналитический раздел

В данном разделе будут рассмотрены основные составляющие компиляторов, а также методы реализации каждого из них.

1.1. Этапы компиляции

Компиляция состоит из четырех основных этапов [1]:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- кодогенерация

Рассмотрим каждый из этих этапов подробнее.

1.1.1. Лексический анализ

На первом этапе компиляции исходный код программы преобразуется в набор лексем - токенов. Обычно [1], процесс лексического анализа организовывается следующим образом: входной поток текста компилируемой программы рассматривается как поток символов, выборкой которых может управлять сам процесс токенизации. Распознавание же отдельных лексем производится с помощью идентификации токенов, которые были определены в грамматике языка. Задачей лексического анализа является аналитический разбор входной последовательности символов с целью получения на выходе последовательности «токенов», которые характеризуются определенными типом и значением.

Лексический анализатор функционирует в соответствии с некоторыми правилами построения допустимых входных последовательностей. Данные правила могут быть определены, например, в виде детерминированного конечного автомата, регулярного выражения или праволинейной

грамматики. С практической точки зрения наиболее удобным способом является формализация работы лексического анализатора с помощью грамматики [2]. Благодаря правилам, задаваемым грамматикой, можно проверить корректность входных цепочек данных. В процессе такой проверки обнаруживаются лексические ошибки - простейшие ошибки компиляции, связанные с наличием в тексте программы недопустимых символов, некорректной записью идентификаторов, числовых констант и пр.

Лексический анализ может быть представлен и как самостоятельная фаза трансляции, так и как составная часть фазы синтаксического анализа. В первом случае лексический анализатор реализуется в виде отдельного модуля, который принимает последовательность символов, составляющих текст компилируемой программы, и выдает список обнаруженных лексем. Во втором случае лексический анализатор фактически является подпрограммой, вызываемой синтаксическим анализатором для получения очередной лексемы [1].

Основной целью лексического анализа является подготовка исходного текста программы для дальнейшей обработки, а также выявление лексических ошибок.

1.1.2. Синтаксический анализ

Синтаксический анализ, или разбор – это процесс сопоставления линейной последовательности токенов исходного языка с его формальной грамматикой [1]. Результатом обычно является дерево разбора (или абстрактное синтаксическое дерево).

Синтаксический анализатор фиксирует синтаксические ошибки, т.е. ошибки, связанные с нарушением принятой структуры программы.

1.1.3. Семантический анализ и кодогенерация

В процессе семантического анализа дерево разбора обрабатывается с целью установления его семантики. На данном этапе производится привязка идентификаторов к их объявлениям, типам данных, проверка совместимости, определение типов выражений и т. д. Результат обычно

[1] называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то еще, удобным для дальнейшей обработки.

Таким образом, семантический анализатор предназначен для нахождения семантических ошибок и накопление данных о переменных, функциях и используемых типах для генерации кода. Информация об используемых объектах сохраняется в иерархические структуры данных т.н. таблицы символов.

1.2. Методы реализации лексического и синтаксического анализаторов

Лексический и синтаксический анализаторы могут быть разработаны с использованием стандартных алгоритмов анализа, а могут быть получены с помощью инструментов генерации анализаторов. Сравним два представленных способа получения необходимых анализаторов.

1.2.1. Алгоритмы анализа

Существуют две основные стратегии синтаксического анализа: нисходящий анализ и восходящий анализ. В нисходящем анализе дерево вывода цепочки строится от корня к листьям, т.е. дерево вывода «реконструируется» в прямом порядке, и аксиома грамматики «развертывается» в цепочку. В общем виде нисходящий анализ представлен в анализе методом рекурсивного спуска, который может использовать откаты, т.е. производить повторный просмотр считанных символов [1]. В восходящем анализе дерево вывода строится от листьев к корню и анализируемая цепочка «сворачивается» в аксиому. На каждом шаге свертки некоторая подстрока, соответствующая правой части продукции, замещается левым символом данной продукции. Примерами восходящих синтаксических анализаторов являются синтаксические анализаторы приоритета операторов, LR-анализаторы (SLR, LALR) [1].

1.2.2. Генерация анализатора

Имеется множество различных стандартных средств для построения синтаксических анализаторов: Lex и Yacc, Coco/R, ANTLR, JavaCC и др.

Генератор Yacc предназначен для построения синтаксического анализатора контекстно-свободного языка. Результатом работы Yacc'а является программа на Си, реализующая восходящий LALR(1) распознаватель. Как правило, Yacc используется в связке с Lex – стандартным генератором лексических анализаторов. Для обоих этих инструментов существуют свободные реализации – Bison и Flex.

Coco/R читает файл с атрибутивной грамматикой исходного языка в расширенной форме и создает файлы лексического и синтаксического анализаторов. Лексический анализатор работает как конечный автомат. Синтаксический анализатор использует методику нисходящего рекурсивного спуска.

ANTLR (ANother Tool for Language Recognition) – это генератор синтаксических анализаторов для чтения, обработки или трансляции как структурированных текстовых, так и бинарных файлов. ANTLR широко используется для разработки компиляторов, прикладных программных инструментов и утилит. На основе заданной грамматики языка ANTLR генерирует код синтаксического анализатора, который может строить абстрактное синтаксического дерево и производить его обход [3]. Принимая во внимание эффективность и простоту использования ANTLR, для построения кода синтаксического анализатора было решено применить данное средство.

В качестве входных данных для ANTLR выступает файл с описанием грамматики исходного языка [3]. Данный файл содержит только правила грамматики без добавления кода, исполнение которого соответствует применению определённых правил. Подобное разделение позволяет использовать один и тот же файл грамматики для построения различных приложений (например, компиляторов, генерирующих код для различных сред исполнения). На основе правил заданной грамматики языка ANTLR генерирует класс нисходящего рекурсивного синтаксического

анализатора. Для каждого правила грамматики в полученном классе имеется свой рекурсивный метод. Разбор входной последовательности начинается с корня синтаксического дерева и заканчивается в листьях.

2. Конструкторский раздел

В данном разделе рассмотрена структура компилятора, генерируемые классы, а также алгоритм работы разрабатываемой программы.

2.1. Структура компилятора

На рисунке 1 приведен нулевой уровень структуры разрабатываемого компилятора.

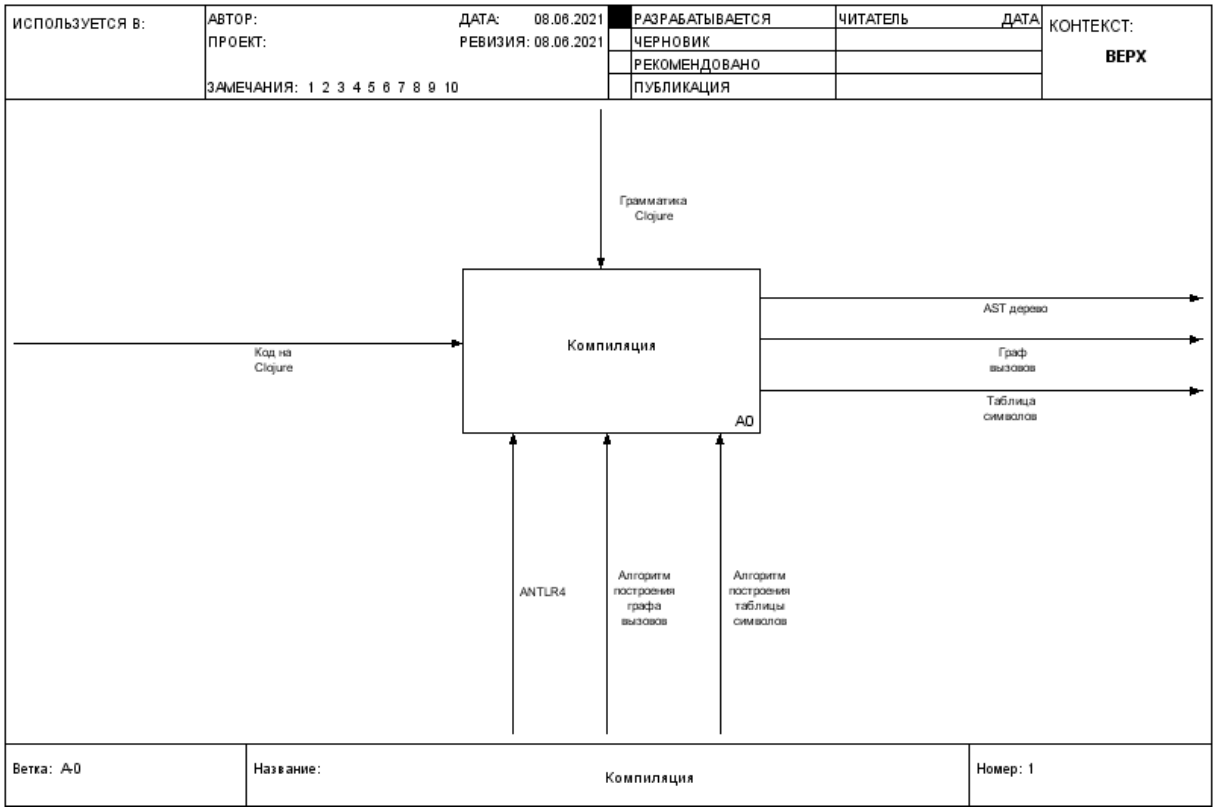

Рисунок 1. Структура компилятора в нотации IDEF0.

На рисунке 2 можно более подробно увидеть этапы анализа программы.

2.2. Алгоритм построения графа вызовов

В языке Clojure терминал '(' является индикатором вызова функции (название которой следует за терминалом '('). Данное правило не

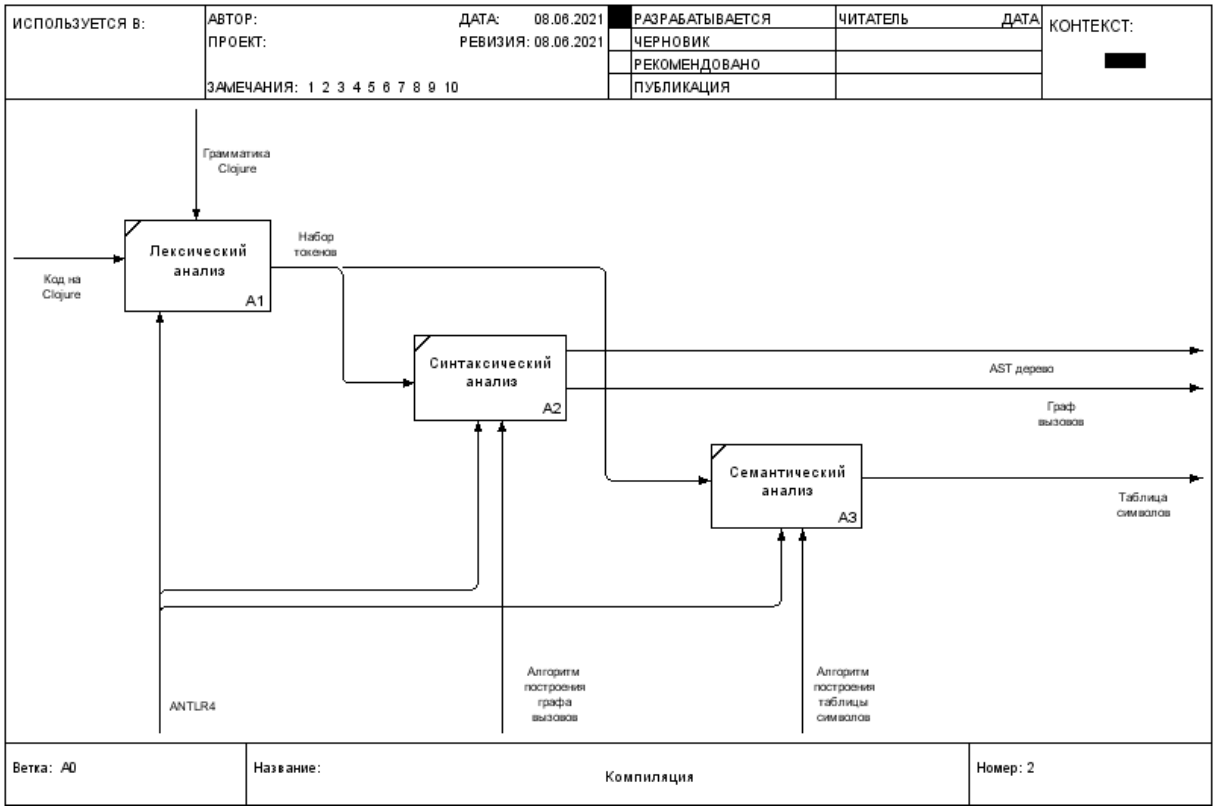

Рисунок 2. Этапы анализа текста программы.

выполняется в случае вызова функции quote, код внутри круглых скобок следует интерпретировать как список. Таким образом, при обходе AST дерева с помощью слушателя, можно построить граф вызовов.

Для этого был разработан следующий алгоритм:

1) Начало обхода
2) Встречен символ quote
   а) Установить флаг встречи quote
   б) Записать новую вершину в список
   в) Записать новую дугу в список, если есть текущая расматриваемая вершина
3) Встречена ’(’ и флаг встречи quote не установлен
   а) Установить флаг вызова функции
4) Установлен флаг вызова функции и нет текущей вершины
   а) Записать новую вершину в список

б) Записать новую дугу в список

в) Сбросить флаг вызова функции

5) Встречена ’)’

а) Обновить текущую рассматриваемую вершину

б) Сбросить флаг встречи quote

2.3. Алгоритм построения таблицы символов

Построение таблицы символов происходит посредством обхода синтаксического дерева от корня к листьям. Разработанный алгоритм можно описать следующим образом:

1) Начало обхода

2) Встречен список, который является вызовом функции, которая порождает новую область видимости

а) Создать новую область видимости

б) Поместить ее в стек областей

3) Обнаружен выход из списка

а) Извлечь из стека список областей

Стоит отметить несколько особенностей выделения областей видимости. В языке Clojure, в формах let и loop допускается переопределение переменных, соотвественно для каждой пары присваиваний из этих форм создается своя область видимости. Также важным аспектом является специфика работы форм def и defn: вне зависимости от своего положения в дереве, объяленные ими символы имеют глобальную область видимости.

3. Технологический раздел

В данном разделе приведены примеры использования разработанной программы, а также некоторые технические особенности реализации.

3.1. Классы анализаторов

С помощью ANTLR4 будут сгенерированы классы анализаторов. ANTLR позволяет генерировать лексер, парсер, а также интерфейсы для слушателей и посетителей. Слушатель и посетитель – две возможные реализации обхода синтаксического дерева. Таким образом, будут сгенерирвоаны:

- ClojureLexer – лексический анализатор;
- ClojureParser – синтаксический анализатор;
- IClojureListener – интерфейс слушателя;
- IClojureVisitor – интерфейс посетителя.

Для обхода дерева, создаваемого сгенерированным классом лексического анализатора, необходимо реализовать сгенерированные интерфейсы слушателя и посетителя.

3.2. Обнаружение ошибок

Все ошибки, которые обнаруживаются лексическим и синтаксическим анализаторами ANTLR, по умолчанию выводятся в стандартный поток вывода ошибок. Данные ошибки возможно перехватить стандартным обработчиком ошибок языка на котором ведется разработка компилятора.

3.3. AST дерево

AST дерево строится сгенерированным синтаксическим анализатором. Рассмотрим на примере следующей программы:

```
(defn kek [a b c]
  (let [a (+ 5.0 3) b (- 5 2)]
    (loop [g a]
      (when `(not= g 0)
        (recur (dec g))))))
```

Листинг 1. Пример программы на Clojure

Построенное дерево приведено в Приложении А.

3.4. Граф вызова функций

Граф строится в нотации DOT, и в дальнейшем может быть представлен и виде изображения. Для программы, приведенной в листинге 1, граф вызовов приведен на рисунке 3.

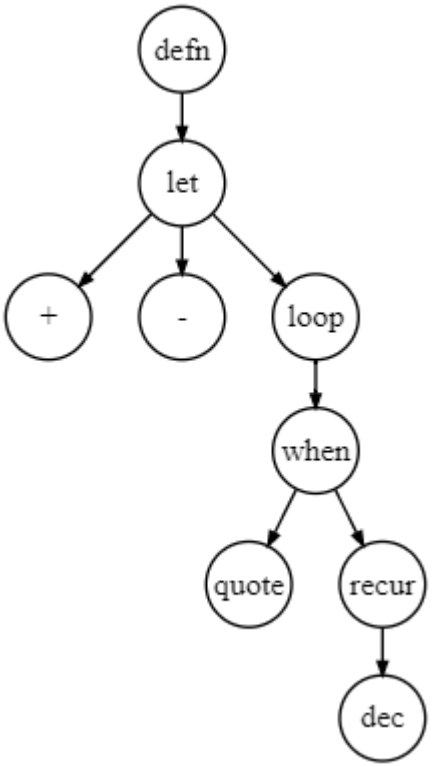

Рисунок 3. Граф вызовов для программы из листинга 1.

3.5. Таблица символов

Таблица символов в виде графа представлено на рисунке 4

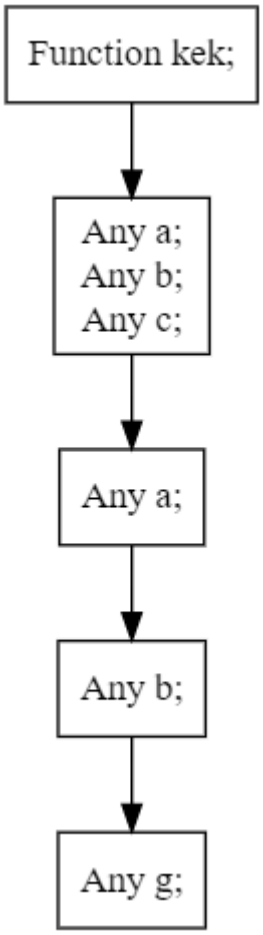

Рисунок 4. Таблица символов в виде графа для листинга 1.

ЗАКЛЮЧЕНИЕ

Рассмотрены основные фазы функционирования приложения, выполняющего лексический, синтаксический и семантический анализ кода языка Clojure.

Приведен обзор основных алгоритмов лексического и синтаксического анализа. Рассмотрены стандартные средства построения синтаксических анализаторов. Подробно описано использование пакета прикладных программ ANTLR4 для генерации исходного кода классов анализаторов по заданной грамматике языка Clojure. Указаны возможные ошибки компиляции, обнаруживаемые в ходе лексического и синтаксического анализа.

СПИСОК ЛИТЕРАТУРЫ

1. Серебряков В. А., Галочкин М. П. «Основы конструирования компиляторов»
2. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.
3. Terence Parr «Definitive ANTLR4 reference». — M.: Pragmatic Bookshelf, 2013.

ПРИЛОЖЕНИЕ А

Листинг AST дерева в нотации DOT.

```
digraph ParseTree {
"58225482
File" -> "54267293
Forms"

"54267293
Forms" -> "18643596
Form"

"18643596
Form" -> "33574638
List"

"33574638
List" -> "33736294
TerminalNodeImpl
Token = \"(\""

"33574638
List" -> "35191196
Forms"

"35191196
Forms" -> "48285313
Form"

"48285313
Form" -> "31914638
Literal"

"31914638
Literal" -> "18796293
Symbol"

"18796293
Symbol" -> "34948909
Simple_sym"

"34948909
Simple_sym" -> "46104728
TerminalNodeImpl
Token = \"defn\""

"35191196
```

```
45 Forms" -> "12289376
46 Form"
47
48 "12289376
49 Form" -> "43495525
50 Literal"
51
52 "43495525
53 Literal" -> "55915408
54 Symbol"
55
56 "55915408
57 Symbol" -> "33476626
58 Simple_sym"
59
60 "33476626
61 Simple_sym" -> "32854180
62 TerminalNodeImpl
63 Token = \"kek\""
64
65 "35191196
66 Forms" -> "27252167
67 Form"
68
69 "27252167
70 Form" -> "43942917
71 Vector"
72
73 "43942917
74 Vector" -> "59941933
75 TerminalNodeImpl
76 Token = \"[\""
77
78 "43942917
79 Vector" -> "2606490
80 Forms"
81
82 "2606490
83 Forms" -> "23458411
84 Form"
85
86 "23458411
87 Form" -> "9799115
88 Literal"
89
90 "9799115
91 Literal" -> "21083178
92 Symbol"
```

```
93
94 "21083178
95 Symbol" -> "55530882
96 Simple_sym"
97
98 "55530882
99 Simple_sym" -> "30015890
100 TerminalNodeImpl
101 Token = \"a\""
102
103 "2606490
104 Forms" -> "1707556
105 Form"
106
107 "1707556
108 Form" -> "15368010
109 Literal"
110
111 "15368010
112 Literal" -> "4094363
113 Symbol"
114
115 "4094363
116 Symbol" -> "36849274
117 Simple_sym"
118
119 "36849274
120 Simple_sym" -> "63208015
121 TerminalNodeImpl
122 Token = \"b\""
123
124 "2606490
125 Forms" -> "32001227
126 Form"
127
128 "32001227
129 Form" -> "19575591
130 Literal"
131
132 "19575591
133 Literal" -> "41962596
134 Symbol"
135
136 "41962596
137 Symbol" -> "42119052
138 Simple_sym"
139
140 "42119052
```

```
141 Simple_sym" -> "43527150
142 TerminalNodeImpl
143 Token = \"c\""
144
145 "43942917
146 Vector" -> "56200037
147 TerminalNodeImpl
148 Token = \"]\""
149 ...
```

Листинг 2. AST дерево для листинга 1.