Федеральное государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *«ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»*
КАФЕДРА *«ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ»*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ

Разработка нескольких фаз генератора лексических анализаторов для Рефала на основе регулярных выражений.

Руководитель курсового проекта _____ (А. В. Коновалов)
(подпись, дата)

Исполнитель курсового проекта,
студент группы ИУ9-72 _____ (Е. С. Бурлова)
(подпись, дата)

Москва, 2017

Содержание

ВВЕДЕНИЕ

На текущий момент система программирования «Простой Рефал» содержит примитивный генератор лексических анализаторов, который на входе принимает таблицу переходов для конечного автомата с семантическими действиями (выдать прочитанную лексему, отбросить прочитанную лексему).

Недостатком таблицы переходов является слишком многословное описание, которое довольно сложно составлять. В данной курсовой работе предлагается написать для системы программирования на Простом Рефале генератор лексических анализаторов, принимающий на входе описания лексических доменов в виде регулярных выражений.

В первом разделе будет описана мотивация, послужившая основой для данной курсовой работы. Будет описан уже имеющийся генератор лексических анализаторов, достоинства и недостатки его работы, а также его пользовательский интерфейс. Далее будет сформулирована расширенная постановка задачи.

Во втором разделе будут описаны синтаксис и семантика разработанного входного языка. Кроме того, будет приведён пример его использования и проведено сравнение с описанным в предыдущем разделе примером.

В третьем разделе будет описана реализация стадий синтеза: лексический, синтаксический и семантический анализ разрабатываемого языка.

В четвёртом, заключительном, разделе будет проведено тестирование и анализ проведённой работы, а также намечены последующие этапы разработки генератора лексических анализаторов на основе регулярных выражений.

1 МОТИВАЦИЯ

1.1. Обзор существующего генератора лексических анализаторов

Прежде чем формулировать расширенную постановку задачи, опишем интерфейс уже имеющегося генератора лексических анализаторов, который послужит основой данной курсовой работы:

* LexGen -- генератор лексического анализатора. На входе программа принимает описание конечного преобразователя (автомата Мили, так как действия выполняются при переходе в новое состояние), на выходе создаётся сам конечный автомат. Синтаксис лексического анализатора приводится на Листинге 1 далее:

```
   Description = Element* .
   Element = SetDescr | Sentence .

   SetDescr = SETNAME '=' Set* '.' .
   Set = LITERAL | SETNAME .

   Sentence = NAME '=' Alternative { '|' Alternative } '.' .
   Alternative = [Set] [Flush] [NAME] .
   Flush = '!-' | NAMEDFLUSH | ERRORFLUSH .

   SETNAME = ':' строчная-или-прописная-буква-или-цифра* ':' .
   LITERAL = "'" последовательность-символов-с-escape-последо-
вательностями '"' .
   NAME = идентификатор-Рефада .
   NAMEDFLUSH = '!' идентификатор-Рефала .
   ERRORFLUSH = '!"' строка-сообщение '"' .
```

Листинг 1. Синтаксис лексического анализатора

Модель вычислений конечного преобразователя следующая. Конечный преобразователь может находиться в одном из нескольких состояний. Кроме того, преобразователь содержит буфер символов, в конец которого он может добавить символ, сбросить его содержимое в выходной поток или опустошить его. Находясь в некотором состоянии конечный преобразователь считывает или не считывает очередной символ из входного потока, если считывает, то обязательно добавляет в конец буфера, производит

действие над буфером (буфер может быть опустошён с выдачей или невыдачей содержимого в выходной поток) и переходит в другое состояние.

В некоторых случаях преобразователь может ожидать конец ввода.

Программа для конечного автомата состоит из набора предложений и описаний множеств. Предложение содержит слева от знака равенства имя текущего состояния, слева -- набор альтернатив, что делает его внешне похожим на БНФ. Альтернатива состоит из множества входных символов, которые можно считать, действия с буфером и нового состояния. Каждый из компонентов альтернативы может отсутствовать.

Если множество символов указано и текущий символ (т.е. не конец ввода) попадает в это множество, то выполняется данная альтернатива и символ считывается. Если же множество отсутствует, но присутствует следующее состояние, то данная альтернатива также активизируется, при этом входной символ не считывается – входной поток не изменяется. При активации альтернативы если символ считывается, то он добавляется в конец буфера. Действие может отсутствовать (в этом случае с буфером ничего не происходит), может быть операцией безымянного сброса (в этом случае буфер опустошается), может быть именованным сбросом (в этом случае содержимое выбрасывается с данным именем – формат выражения (s.Name e.Content), где s.Name – идентификатор – имя сброса (в случае Простого Рефала -- имя функции), e.Content – содержимое буфера в данный момент,включая считанный символ. Если действие – сброс-сообщение об ошибке, то в выходной поток выводится зарезервированная лексема TokenError.

Множество в начале альтернативы можно задать двумя способами -- как множество явно перечисленных символов, так и как именованное множество. Объявление именованного множества состоит из имени множества, знака равенства и перечисления входящих в него множеств. Объявление множества, как и предложение автомата, завершается точкой. Содержимое множества представляет собой объединение множеств, перечисленных справа от знака равно. Поэтому допустима рекурсивная зависимость множеств. Именованные множества позволяют не только сократить описания предложений, но и повысить ясность программы.

Также есть особое именованное множество :Any:, которое включает в себя все возможные символы.

По соглашению, автомат начинает свою работу с состояния Root с пустым буфером.

1.2. Пример использования

Рассмотрим использование существующего генератора лексических анализаторов на примере работы с идентификаторами, числами, арифметическими операциями и строками.

Сначала задаются описания лексических доменов: отступов, цифр, букв, арифметических операций и escape-символов. Далее описываются состояния автомата:

```
/* GEN: TOKENS
 :Space: = ' \t\n'.
 :Digit: = '0123456789'.
 :Letter: = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.
 :Op: = '+-*/=()'.
 :ESCAPED: = 'nrt\'\\'.

 Root =
      :Space:     !-                              Root        |
      :Op:        ! #Operation                    Root        |
      :Digit:                                     Number      |
      :Letter:                                    VarName     |
      :'"'        !-                              String      |
      :Any:       ! "Unknown symbol"              Root        |
                  ! #EOF                                      .
 Number =
      :Digit:                                     Number      |
                  ! #Number                       Root        .
 VarName =
      :Digit:                                     VarName     |
      :Letter:                                    VarName     |
                  ! #VarName                                  .
 String =

      '"'         !-                              Root        |

      '\\'                                        Escaped     |

      '\n'        ! "Unclosed quote"              Root        |

      :Any:       ! #TkChar                       String      |

                  ! "Unexpected EOF in string"                .

Escaped =

      :ESCAPED:   ! #TkChar-Escaped       String              |

                  ! "Bad escaped seq"     String              .

GEN:END*/
```

корневое, распознавание числа, имени переменной, строки и escape-последовательности, а также задаются правила переходов.

Листинг 2. Пример использования генератора для переменных, чисел, арифметических операций и строк.

Описание автомата обрамляется строками `/*GEN:TOKENS` и `GEN:END*/`, где `/**/` – многострочный комментарий языка Си, и сохраняется в отдельном файле (например, __test.sref). Для вызова генератора лексических анализаторов выполняется команда:

```
../../bin/lexgen __test.sref
```

Код автомата генерируется сразу же после данного комментария и до конца файла. Если же между последней строчкой комментария и концом файла находился какой-либо текст, то этот текст удаляется.

1.3. Достоинства и недостатки имеющегося генератора

Главным достоинством текущего генератора является эффективность при компиляции простого Рефала с включёнными оптимизациями. Кроме того, очевидным достоинством является тот факт, что скорость генерации лексера выше, чем реализация лексического анализатора вручную.

К недостаткам же следует отнести то, что необходимо явно составлять автомат. Для описания автомата было бы куда удобнее использовать регулярные выражения.

1.4. Постановка задачи на развитие

Для решения задачи разработки генератора лексических анализаторов для Рефала на основе регулярных выражений предстоит:

1) разработать входной язык, синтаксически схожий с flex.

2) дополнить существующий генератор новым режимом «GEN:FLEX», реализовав стадии синтеза языка.

Также работа должна удовлетворять следующим требованиям:

1) Языком реализации должен быть Простой Рефал, причём лексика должна быть описана на себе (на последней итерации).

2) Сгенерированный код должен быть совместим с Простым Рефалом и РЕФАЛом-5.

3) Разработанное ПО должно быть кроссплатформенным.

2 СИНТАКСИС И СЕМАНТИКА ВХОДНОГО ЯЗЫКА

Описание лексики пишется внутри комментария языка Си:

```
/*GEN:FLEX
 ...
 GEN:END*/
```

и представляет собой последовательность описаний лексических доменов и именованных подвыражений. Описание лексических доменов имеет следующий синтаксис:

```
[<State1, State2, ...>]/regexp/ → Domain[, NextState]
```

Здесь квадратные скобки обозначают необязательные элементы.

Описание лексических доменов начинается с необязательного списка состояний, перечисленных в угловых скобках. Если последние отсутствуют, используется состояние DEFAULT. Далее указывается регулярное выражение, после которого записывается семантическое действие.

Действие может быть:

1) Формированием лексемы из прочитанной строки

2) Отбрасыванием прочитанной строки

3) Отбрасыванием прочитанной строки с выводом сообщения об ошибке.

После действия может быть указано состояние, в которое следует перейти, по умолчанию – остаётся то же состояние, что и было раньше.

Именованное регулярное выражение будет иметь следующий синтаксис:

```
Expr_name =  /regexp/
```

Здесь `Expr_name` – имя регулярного выражения, начинающееся с заглавных или строчных латинских букв, может содержать цифры. Неформальное описание регулярных выражений приведено в Таблице 1.

Регулярное выражение	Описание
x	Соответствует литере ’x’.
.	Любая литера, кроме перевода строки.
[xyz]	Класс литер, распознающий литеры ’x’, ’y’ или ’z’.
[abc-mZ]	Класс литер с интервалом, распознаёт литеры ’a’, ’b’, любые литеры в интервале от ’c’ до ’m’, а также литеру ’Z’.

[^A-Z]	Инвертированный класс литер, содержащий все литеры, кроме перечисленных.
r*	Ноль или более вхождений r, где r – произвольное регулярное выражение.
r+	Один или более вхождений r.
r?	Ноль или одно вхождений r.
{Name}	Подстановка именованного регулярного выражения, определённого ранее.
\X	Escape-последовательность; в качестве X допустимы ’n’, ’r’, ’t’, ’\\’ (их смысл аналогичен Escape последовательностям языка C) или любой символ, используемый в качестве метасимвола расширенного языка регулярных выражений (’(’, ’)’, ’[’, ’]’, ’{’, ’}’, ’.’, ’*’, ’+’, ’?’, ’\’ и т.д.).
\123	Литера с восьмиричным кодом 123.
\x2A	Литера с шестнадцатеричным кодом 2A.
(r)	Регулярное выражение r (круглые скобки служат для задания приоритета).
rs	Конкатенация регулярных выражений r и s.
r|s	Либо r, либо s.

Таблица 1. Описание регулярных выражений, заданное на расширенном языке регулярных выражений

Далее опишем пример генерации лексического анализатора для случая описанного в разделе 1.2 с использованием разработанного языка регулярных выражений.

```
/*GEN:FLEX
      OP                  = /[+-*]=()/
      Digit               = /[0-9]/
      Number              = /{Digit}+/
      Letter              = /[a-zA-Z]/
      VarName             = /{Letter}({Letter}|{Digit}*)/
      Space               = /\t\n/
      /{Space}/           -> -
      /{Number}/          -> Number
      /{VarName}/         -> VarName
      /./                 -> "Unknown symbol"
      /“/                 -> -, STRING
      <STRING>/[^\\“\n]/  -> TkChar
      <STRING>/\\[nt“\\]/ -> TkEscaped
      <STRING>/“/         -> -, DEFAULT
      <STRING>/\n/        -> "Unclosed quote", <>
      <STRING>/\\/        -> "Bad escaped seq"
      <<EOF>>             -> EOF
GEN:END*/
```

Листинг 3. Пример использования генератора для переменных, чисел, арифметических операций и строк с применением регулярных выражений.

Как видно, подобная запись является более компактной по сравнению с предыдущим примером. Кроме того, упрощается процесс составления автомата – у пользователя нет необходимости держать в уме диаграмму переходов и учитывать все возможные альтернативы, достаточно описать автомат непосредственно набором правил.

3 РЕАЛИЗАЦИЯ СТАДИИ СИНТЕЗА

3.1. Лексический анализ

На данном этапе предстоит составить автомат, обрабатывающий входную последовательность символов, используя существующий генератор лексических анализаторов *LexGen.

Для обработки синтаксиса языка регулярных выражений, описанного в предыдущем разделе, понадобятся следующие лексические домены (Листинг 4.1): строчные и заглавные буквы, числа в десятичной, восьмеричной и шестнадцатеричной системах счисления, имена переменных, отступы, escape-последовательности, квантификаторы и специальные символы.

```
/* GEN : TOKENS
  :UpLetters: = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
  :DownLetters: = 'abcdefghijklmnopqrstuvwxyz'.
  :Digit: = '0123456789'.
  :ODigit: = '01234567' .
  :XDigit: = '0123456789ABCDEFabcdef' .
  :NameChars: = :UpLetters: :DownLetters: :Digit:.
  :Spaces: = ' \t\r'.
  :GenericEsc: = '\\ntr'.
  :Quantifier: = '?*+'.
  :SpecChar: = '.()|^$?*+'.
```

Листинг 4.1 Описание синтаксиса языка регулярных выражений, заданное с помощью существующего генератора

Далее опишем состояния автомата. Находясь в корневом состоянии, мы можем (Листинг 3, Листинг 4.2):

1) прочитать число или имя состояния

2) прочитать символ «стрелки», обозначающей переход к действию при обработке прочитанного правила

3) начать обрабатывать встретившееся регулярное выражение

4) выдать сообщение об ошибке

```
Root =
    :Spaces:        !-                      Root                |
    '\n'            !#TNewLine              Root                |
    '/'             !#TStartRegexp          ReadRegexp          |
    '-'             !-                      ReadArrow           |
    '='             !#TEquals               Root                |
    :Digit:                                 Number              |
    :UpLetters:                             Name                |
    '<'             !#TStateStart           Root                |
    ','             !#TComma                Root                |
    '>'             !#TStateEnd             Root                |
    '"'             !-                      ErrorMessage        |
    :Any:           !#TUnexpecteChar        Root                |
                    !#TEOF                                      .
  ReadArrow =
    '>'             !#TArrow                Root                |
                    !#TSkip                 Root                .
```

Листинг 4.2 Описание синтаксиса языка регулярных выражений, заданное с помощью существующего генератора.

При чтении регулярного выражения, мы можем либо встретить инвертированную или обычную группу символов, либо выражение в фигурных скобках, либо продолжить чтение (Листинг 4.3).

```
  ErrorMessage =
    '"'             !#TErrorMessage         Root                |
    :Any:                                   ErrorMessage        |
                    !"Unclosed error message"                   .
  ReadRegexp =
    '/'             !#TEndRegexp            Root                |
    '\n'            !#TNewLine              Root                |
    '\\'            !-                      EscRegexp           |
    '['             !#TStartGroup           StartGroup          |
    :Quantifier:    !#TQuantifier           ReadRegexp          |
    :SpecChar:      !#TSpecChar             ReadRegexp          |
    '{'             !-                      ReadInclude         |
    :Any:           !#TChar                 ReadRegexp          |
                    !"Unexpected EOF in Regexp"                 .
  EscRegexp =
    :GenericEsc:    !#TGenericEscChar       ReadRegexp          |
    '/'             !#TChar                 ReadRegexp          |
    'x'             !-                      EscRegexp-XCode     |
    'X'             !-                      EscRegexp-XCode     |
    :ODigit:                                EscRegexp-OCode     |
    :SpecChar:      !#TChar                 ReadRegexp          |
    :Any:           !"Bad escaped symbol"   ReadRegexp          |
                    !"Unexpected EOF in Regexp"                 .
```

Листинг 4.3 Описание синтаксиса языка регулярных выражений, заданное с помощью существующего генератора.

При обработке группы символов, мы можем встретить восьмеричные и шестнадцатеричные числа, либо продолжить чтение (Листинг 4.4-4.5).

```
  StartGroup =
    '^'             !#TInvertGroup          ReadGroup           |
                                            ReadGroup           .
  ReadGroup =
    ']'             !#TEndGroup             ReadRegexp          |
    '-'             !#TGroupRange           ReadGroup           |
    '\\'            !-                      EscGroup            |
    :Any:           !#TChar                 ReadGroup           |
                    !"Unexpected EOF in Regexp"                 .
  EscGroup =
    :GenericEsc:    !#TGenericEscChar       ReadGroup           |
    ']'             !#TChar                 ReadGroup           |
    '-'             !#TChar                 ReadGroup           |
    'x'             !-                      EscGroup-XCode      |
    'X'             !-                      EscGroup-XCode      |
    :ODigit:                                EscGroup-OCode      |
    :Any:           !"Bad escaped symbol"   ReadGroup           |
                    !"Unexpected EOF in Regexp"                 .
  ReadInclude =
    :UpLetters:                             ReadBody            |
    :Any:           !"Expected Name in Include"
                                            ReadRegexp          |
                    !"Unexpected EOF in Include"                .
  ReadBody =
    :NameChars:                             ReadBody            |
    '}'             !#TInclude              ReadRegexp          |
                    !"Unclused include"     ReadRegexp          .
```

Листинг 4.4 Описание синтаксиса языка регулярных выражений, заданное с помощью существующего генератора.

```
  Name =
    :NameChars:                             Name                |
                    !#TName                 Root                .
  Number =
    :Digit:                                 Number              |
                    !#TkNumber              Root                .
  EscRegexp-XCode =
    :XDigit:                                EscRegexp-XCode     |
                    !#TkHexNumber           ReadRegexp          .
  EscRegexp-OCode =
    :ODigit:                                EscRegexp-OCode-C1  |
                    !#TkOctNumber           ReadRegexp          .
  EscRegexp-OCode-C1 =
    :ODigit:        !#TkOctNumber           ReadRegexp          |
                    !#TkOctNumber           ReadRegexp          .
  EscGroup-XCode =
    :XDigit:                                EscGroup-XCode      |
                    !#TkHexNumber           ReadGroup           .
  EscGroup-OCode =
    :ODigit:                                EscGroup-OCode-C1   |
                    !#TkOctNumber           ReadGroup           .
  EscGroup-OCode-C1 =
    :ODigit:        !#TkOctNumber           ReadGroup           |
                    !#TkOctNumber           ReadGroup           .
GEN:END*/
```

Листинг 4.5 Описание синтаксиса языка регулярных выражений, заданное с помощью существующего генератора.

3.2. Синтаксический анализ (грамматика и описание дерева)

На этапе синтаксического анализа составляется грамматика регулярных выражений в РБНФ.

```
Description = (Subexpr|Domain)+.
Subexpr     = NAME "=" Regexp.
Domain      = States?Regexp "->" Flush(","State)?.
State       = "<" NAME? ">".
Regexp      = "/" Alt "/" | "<<EOF>>".
Alt         = Term*("|"Term*)*.
Term        = SimpleTerm Repeater?.
Repeater    = "*"|"+"|"?".
SimpleTerm  = "."|Set|"{"NAME"}"|SYMBOL|"("Alt")".
Set         = "[" "^"? (SYMBOLSET("-"SYMBOLSET)?)+"]".
States      = "<"(NAME(","NAME)*)?">".
Flush       = NAME | "-" | ERROR.
```

Листинг 5. Грамматика регулярных выражений в РБНФ.

На приведённом выше листинге 5:

NAME — множество слов, начинающихся с большой буквы

SYMBOL – множество некоторых символов, а также \n, \., \\, \/

Однострочные комментарии начинаются со знака # и продолжаются до конца строки.

SYMBOLSET – множество { \], \-, \^}.

ERROR – текст в двойных кавычках.

SYMBOL – любой символ, кроме /, \, ., [, {, (, *, +, ?, |, либо переход на новую строку, либо \\, \/, \., \[…

Далее мы переходим от РБНФ к БНФ с целью избавиться от итераторов и ветвлений в целевой грамматике.

На вход синтаксическому анализатору лексером подается поток лексем. На выходе мы получаем абстрактное синтаксическое дерево.

БНФ грамматики и структура дерева приведены в Листинге в приложениях. Реализация парсера будет производиться методом рекурсивного спуска.

Как правило, синтаксический анализатор использует глобальную переменную, хранящую текущий символ и функция вычисления следующего символа (который и

присваивался глобальной переменной). В данном случае проход лексического анализатора отдельный — сразу читается список лексем. Поэтому в глобальной переменной G_Tokens хранится ещё не прочитанная последовательность лексем.

Перечислим несколько служебных функций:

```
Current {
  =
    <Fetch
     <G_Tokens>
     {
       (s.Type t.Position e.Info) e.OtherTokens =
             (s.Type t.Position e.Info)
             <G_Tokens (s.Type t.Position e.Info) e.OtherTokens>;
      }
     >;
}
MoveNext {
  =
    <Fetch
       <G_Tokens>
       { t.First e.OtherTokens = <G_Tokens e.OtherTokens>; }
    >;
}
```

Листинг 6. Служебные функции парсера `Current` и `MoveNext`.

Функция `Current` возвращает текущий токен (первый в последовательности), функция `MoveNext` отбрасывает первый токен.

```
CurrentTag {
  =
    <Fetch
       <Current>
       { (s.Type t.Position e.Info) = s.Type; }
    >;
}

CurrentAttr {
  =
    <Fetch
       <Current>
       { (s.Type t.Position e.Info) = e.Info; }
    >;
}

Expect {
  s.Expected e.ExpectedDescription =
    <Fetch
       <CurrentTag>
       {
         s.Expected = <MoveNext>;
         s.Unexpected =
            <Error 'Unexpected ' <Flex-TextFromToken <Current>> ',
            but expected ' e.ExpectedDescription>;
       }
    >;
}
```

Листинг 7. Служебные функции парсера `CurrentTag`, `CurrentAttr` *и* `Expect`.

Функции CurrentTag и CurrentAttr возвращают тэг и атрибут текущей лексемы. Функция Expect используется, если необходимо проверить, присутствует ли терминальный символ в некоторой точке по середине правила. Expect анализирует текущий символ и, если последний не совпадает с ожидаемым символом, возвращает ошибку.

Также рассмотрим следующие два случая на Листинге 8.

```
//<InvertOpt> = "^" | e

//<Repeater> = "*" | "+" | "?" | e
```

Листинг 8. Одна и несколько альтернатив во множестве FIRST.

В первом случае во множестве FIRST альтернативы присутствует только один символ, во втором – три. При их анализе, чтобы избежать дублирования кода, удобно обобщить группы

тегов токенов в один с помощью функции Generalize, приведенной на Листинге 9.

```
Generalize {
  s.Tag e.Generics-B (s.GenericTag e.Tags-B s.Tag e.Tags-E) e.Generics-E
      = s.GenericTag;
  s.Tag e.Generics
      = s.Tag;
}

<Generalize #C (#XX #A #B #C) (#YY #P #Q #R)> → #XX
<Generalize #Q (#XX #A #B #C) (#YY #P #Q #R)> → #YY
<Generalize #K (#XX #A #B #C) (#YY #P #Q #R)> → #K
```

Листинг 9. Обобщение групп токенов.

Функция Generalize получает символ и одну или несколько групп — скобочных термов, содержащих имя группы и её элементы. Если символ входит в одну из групп, то возвращается её имя (если входит в несколько групп — то имя самой первой из них). Если ни в одну из групп не входит — возвращается как есть.

```
Error {
  e.Message =
       <Fetch
        <Current>
        {
         (s.Type t.Pos e.Info) = <ErrorAt t.Pos e.Message>;
        }
        >;
}
```

Листинг 10. Обработка ошибок.

Функция Error должна брать у текущего токена его координату и распечатывать сообщение об ошибке вместе с координатой, после чего завершать программу с ошибкой.

3.3. Реализация семантического анализа

На стадии семантического анализа будет проведена проверка синтаксического дерева на корректность.

Реализована функция Flex-Checker, обеспечивающая проверку того, что ни один домен не упоминается дважды.

Для полной проверки синтаксического дерева на корректность необходимо обойти AST в поиске конструкций, которые на данный момент не поддерживаются:

1) именованные подвыражения внутри регулярных выражений доменов,

2) символы «^» и «$» в регулярных выражениях доменов,

3) знак инверсии «^» в группах,

4) знак «.» (точка) в регулярных выражениях (т. к. этот знак эквивалентен выражению [^\n], содержащему инверсию),

5) лексические домены для сбросов («-») и сообщений об ошибках (текст в кавычках) — все домены считаем именованными.

Состояния и именованные группы также не поддерживаются, а игнорируются (отбрасываются при анализе).

При обнаружении неподдерживаемого знака необходимо выдавать сообщение об ошибке и ее позицию. Позицией ошибки можно считать либо 0 (нулевую строку), либо позицию лексического домена.

При обнаружении запрещённого элемента, сообщение об ошибке должно прерывать выполнение программы. Таким образом до следующих стадий будут доходить только программы, не содержащие этих конструкций.

4 ТЕСТИРОВАНИЕ

```
/*GEN:FLEX
  /ab|cd/ -> Ident
  /[0-9]+/ -> Digit
GEN:END*/
```

Листинг 11. Описание лексики в тестовом файле.

```
 (#Domain                      //t.Domain
  (2 'fail_test.sref' )        //t.SrcPos
  ()                           //(e.States)
  (#Regexp                     //t.Regexp ::= (#Regexp e.Alt)
   ((#Term                     //e.Alt ::= (t.Term*)+
     (#Char 'a' ))             //t.Term ::= (#Term t.SimpleTerm s.Rep)
    (#Term
     (#Char 'b' )))
   ((#Term
     (#Char 'c' ))
    (#Term
     (#Char 'd' ))))
  (#NamedDomain 'Ident' )#NoNextState )
 (#Domain
  (3 'fail_test.sref' )
  ()
  (#Regexp
   ((#Term
   . (#Set #Direct
   .  ('09' ))#ManyOne )))
  (#NamedDomain 'Digit' )#NoNextState )
```

Листинг 12. Сообщение об ошибке, выводимое при анализе тестового файла.

ЗАКЛЮЧЕНИЕ

В данной работе были полностью реализованы первые три фазы генерации лексического анализатора – разработка синтекса и семантики нового входного языка, а также лексический и синтаксический анализ.

Предстоит доработать стадию семантического анализа – осуществлять проверку синтаксического дерева на корректность и упростить конструкцию дерева.

Завершающим будет этап генерации кода, который включит в себя подстановку всех подстрок, построение конечных автоматов для каждого лексического домена и построение распознавателя, который будет передан генератору кода на Рефале.

ПРИЛОЖЕНИЯ

```
//<Program> = <Description><Descriptions>
/*
  e.Program ::= t.Description+
*/

//<Description> = <Subexpr>|<Domain>
/*
  t.Description ::=
      t.Subexpr
    | t.Domain
*/

//<Descriptions> = <Description> <Descriptions> | e
/*
  e.Descriptions ::= t.Description*
*/

//<Subexpr> = <NAME>"="<Regexp>
/*
  t.Subexpr ::= (#Subexpr t.Pos e.Name t.Regexp)
*/

//<Domain> = <DomainFrom>"->"<DomainTo>
/*
  t.Domain ::= (#Domain t.Pos e.LeftPart e.RightPart)
*/

//<DomainFrom> = <StatesOpt><Regexp>
/*
  e.LeftPart ::= (e.States) t.Regexp
*/

//<StatesOpt> = <States> | e
/*
  e.States ::= (e.StateName)*
*/

//<DomainTo> = <Flush> <NextStateOpt>
/*
  e.RightPart = t.Flush e.NextState
*/

//<Flush> = <NAME> | "-" | <ERROR>
/*
  t.Flush ::=
      (#NamedDomain e.Name)
    | #SkippedDomain
    | (#ErrorDomain e.Message)
*/
```

Листинг 13.1 БНФ-грамматика регулярных выражений и структура полученного дерева.

```
//<NextStateOpt> = "," <State> | e
/*
  t.NextState ::=
      (#NoNextState)
    | (#NextState t.Pos e.Name)
*/

//<State> = "<"<NameOpt>">"
//<NameOpt> = <NAME> | e
//<Regexp> = "/" <Alt> "/"
/*
  t.Regexp ::= (#Regexp e.Alt)
*/

//<Terms> = <Term> <Terms> | e
//<Term> = <SimpleTerm><Repeater>
/*
  t.Term ::= (#Term t.SimpleTerm s.Rep)
*/

//<Repeater> = "*"|"+"|"?"|e
/*
  s.Rep ::= #ManyZero | #ManyOne | #Optional
*/

//<SimpleTerm> = "."|<Set>|"{"<NAME>"}"|<SYMBOL>|"("<Alt>")"
/*
  t.SimpleTerm ::=
      #AnyChar
    | t.Group
    | (#NamedRegexp e.Name)
    | (#Char s.Char)
    | (#Alt e.Alt)
*/

//<Set> = "["<InvertOpt><ComplexSYMBOLSET><ComplexSYMBOLSETS>"]"
/*
  t.Group ::= (#Set s.GroupMode e.Set)
  e.Set :: = (t.ComplexSYMBOLSET)+
*/

//<InvertOpt> = "^"|e
/*
  s.GroupMode ::= #Inverted | #Direct
*/

//<ComplexSYMBOLSETS> = <ComplexSYMBOLSET><ComplexSYMBOLSETS> | e
//<ComplexSYMBOLSET> = <SYMBOL><optSYMBOL>
/*
  t.ComplexSYMBOLSET ::= s.SYMBOLSET+
  s.SYMBOLSET ::= (s.Char) | (s.CharStart s.CharEnd)
  // equivalent: (s.Char e.OptSymbol)
*/

//<optSYMBOL> = "-"<SYMBOL> | e
/*
  e.OptSymbol ::= s.Char | empty
*/
```

Листинг 13.2 БНФ-грамматика регулярных выражений и структура полученного дерева.

```
//<States> = "<"<StateNames>">"
//<StateNames> = <NAME><NextName> | e
/*
  e.StateNames ::= (t.StateName)*
  t.StateName ::= (#Name t.Pos e.Name)
*/

//<NextName> = "," <NAME> <NextNAME> | e
//<Alt> = <Terms> <AltTail>
/*
  e.Alt ::= (t.Term*)+
*/

//<AltTail> = "|" <Terms> <AltTail> | e
```

Листинг 13.3 БНФ-грамматика регулярных выражений и структура полученного дерева.

Список использованной литературы.

1) Турчин В. Ф. Пользовательская документация для языка РЕФАЛ-5 [Электронный ресурс]: Содружество «РЕФАЛ/Суперкомпиляция» .- Режим доступа: http://www.refal.net/rf5_frm.htm .
2) Турчин В. Ф. Алгоритмический язык рекурсивных функций (РЕФАЛ). — М.: ИПМ АН СССР, 1968.
3) Коновалов А.В. Пользовательская документация для языка Простой Рефал [Электронный ресурс]: GitHub .- Режим доступа: https://github.com/bmstu-iu9/simple-refal .