ИНСТИТУТ ТРАНСПОРТА И СВЯЗИ



ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК И ТЕЛЕКОММУНИКАЦИЙ

Лабораторная работа №1

По курсу «Проектирование компиляторов»

Тема «Программирование лексического анализатора»

Выполнили:

Михаил Красильников

Группа 4701BV

Проверила Ирина Птицина

Рига, 2020

**Содержание**

[Краткое теоретическое обоснование 3](#_Toc54631334)

[Текст заданного программного фрагмента индивидуального задания 4](#_Toc54631335)

[Грамматические правила для имеющихся в заданном программном фрагменте лексем. 5](#_Toc54631336)

[Диаграмма состояний соответствующего конечного автомата. 6](#_Toc54631337)

[Исходные таблицы лексем (ключевые слова и специальные символы). 6](#_Toc54631338)

[Текст программы сканера. 8](#_Toc54631339)

[Результаты работы сканера. 11](#_Toc54631340)

[Выводы 13](#_Toc54631341)

# Краткое теоретическое обоснование

Лексический анализатор – это компонент, который читает исходный текст программы и распознает типы лексем, всего существуют несколько типов лексем, а именно:

* Индетификаторы
* Ключевые слова
* Специальные символы
* Литералы

Для построения лексического анализатора требуется выделить все лексемы из программного фрагмента и построить грамматику языка, грамматику языка можно изобразит при помощи формы Бэкуса наура.

BNF форма состоит из терминалов и нетерминалов.

* Терминал - состоит из конечного множества символов.
* Нетерминал – описывает сущность языка(к примеру слово или формул)

Следующий шаг заключается в том, что требуется построить детерминированный конечный автомат по построенным грамматическим правилам.

# Текст заданного программного фрагмента индивидуального задания

|  |
| --- |
| while Tmp<=nil do  Begin  if Tmp.Num = StrToInt(S) then i := i+1;  Begin  i := i+1;  Summ := Summ + Tmp^.Mark  tmp:=Tmp^.Next  End;  j:= Summ/i;  Edit5.Text := FloatToStr(j)  End; |

# Грамматические правила для имеющихся в заданном программном фрагменте лексем.

|  |
| --- |
| <identifier> ::= <letter> | < identifier> <letter> | < identifier > <number>  <integer-literal> ::= <number>  <two\_delim> :: = <less> **>** | <colon> **=**  <one\_delim> :: = **=** | **+** | **;** | **.** | **^** | **/** | **(** | **)**  <less>::= <  <colon> ::= :  <letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z  <keywords> ::= while | nil |do | if | then |end  <number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# 

# 

# Диаграмма состояний соответствующего конечного автомата.

**Рисунок 1 Диаграмма состояний**

# 

# Исходные таблицы лексем (ключевые слова и специальные символы).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Идентификаторы | Cлужебные слова | однопозиционные разделители | двухпозиционные разделители | Константа |
| Tmp | While | = | <> | 1 |
| Num | Nil | + | := |  |
| StrToInt | Do | ; |  |  |
| S | If | . |  |  |
| I | Then | ^ |  |  |
| Summ | End | / |  |  |
| Mark |  | ( |  |  |
| Next |  | ) |  |  |
| j |  |  |  |  |
| Edit5 |  |  |  |  |
| Text |  |  |  |  |
| FloatToStr |  |  |  |  |

# Текст программы сканера.

|  |
| --- |
| <?php declare(strict\_types=1);  namespace Krasilnikovm\Compiler;  /\*\*  \* Class Scanner  \*  \* @author Mihail Krasilnikov <mihail.krasilnikov.j@gmail.com>  \*/  final class Scanner  {  private const KEYWORDS = [  'while',  'nil',  'do',  'if',  'then',  'end',  'begin',  ];  private const ONE\_DELIMITER = [  '=',  '+',  ';',  '.',  '^',  '/',  '(',  ')',  ];  private const TWO\_DELIMITER = [  '<>',  ':=',  ];  private FileReader $fileReader;  private string $word = '';  private ?string $char = '';  /\*\*  \* Scanner constructor.  \* @param FileReader $fileReader  \*/  public function \_\_construct(FileReader $fileReader)  {  $this->fileReader = $fileReader;  }  public function scan(): \Generator  {  $this->char = $this->fileReader->getChar();  while ($this->char !== null) {  $this->word = '';  if (ctype\_space($this->char) || $this->char === '\n') {  $this->char = $this->fileReader->getChar();  continue;  }  switch (true) {  case ctype\_alpha($this->char):  yield $this->handleIdentifierState();  break;  case ctype\_digit($this->char):  yield $this->handleLiteralState();  break;  case in\_array($this->char, [':', '<'], true):  yield $this->handleTwoDelimiterState();  break;  case in\_array($this->char, self::ONE\_DELIMITER, true):  yield $this->handleOneDelimiterState();  break;  default:  $position = $this->fileReader->getCurrentPosition() - strlen($this->word);  yield [$this->char, Constants::ERROR\_TYPE, "Unexpected lexem found on line {$this->fileReader->getCurrentLine()} and on position $position"];  $this->char = $this->fileReader->getChar();  }  }  }  private function handleIdentifierState(): array  {  $isCorrectIdentifier = true;  do {  if (!ctype\_digit($this->char) && !ctype\_alpha($this->char)) {  $isCorrectIdentifier = false;  }  $this->word .= $this->char;  $this->char = $this->fileReader->getChar();  } while (!ctype\_space($this->char) && !in\_array($this->char, self::ONE\_DELIMITER) && !in\_array($this->char, ['<', ':']));  if (!$isCorrectIdentifier) {  $position = $this->fileReader->getCurrentPosition() - strlen($this->word);  $message = "Unexpected lexem found on line {$this->fileReader->getCurrentLine()} and on position $position";  return [$this->word, 'error', $message];  }  $type = in\_array(strtolower($this->word), self::KEYWORDS) ? Constants::KEYWORD : Constants::IDENTIFIER\_TYPE;  return [$this->word, $type, null];  }  private function handleLiteralState(): array  {  do {  $this->word .= $this->char;  $this->char = $this->fileReader->getChar();  } while (ctype\_digit($this->char) || ctype\_alpha($this->char));  $position = $this->fileReader->getCurrentPosition() - strlen($this->word);  $message = null;  $type = Constants::LITERAL\_TYPE;  if (!ctype\_digit($this->word)) {  $message = "Unexpected lexem found on line {$this->fileReader->getCurrentLine()} and on position $position";  $type = Constants::ERROR\_TYPE;  }  return [$this->word, $type, $message];  }  private function handleTwoDelimiterState(): array  {  $this->word .= $this->char;  $this->char = $this->fileReader->getChar();  $this->word .= $this->char;  $type = in\_array($this->word, self::TWO\_DELIMITER) ? Constants::DELIMITER : Constants::ERROR\_TYPE;  if ($type === 'error') {  $position = $this->fileReader->getCurrentPosition() - strlen($this->word);  $message = "Unexpected lexem found on line {$this->fileReader->getCurrentLine()} and on position $position";  }  $this->char = $this->fileReader->getChar();  return [$this->word, $type, $message ?? null];  }  private function handleOneDelimiterState(): array  {  $this->word .= $this->char;  $this->char = $this->fileReader->getChar();  return [$this->word, Constants::DELIMITER, null];  }  } |

# Результаты работы сканера.

Рисунок 2 Результат работы сканера при условии, если нет ошибок

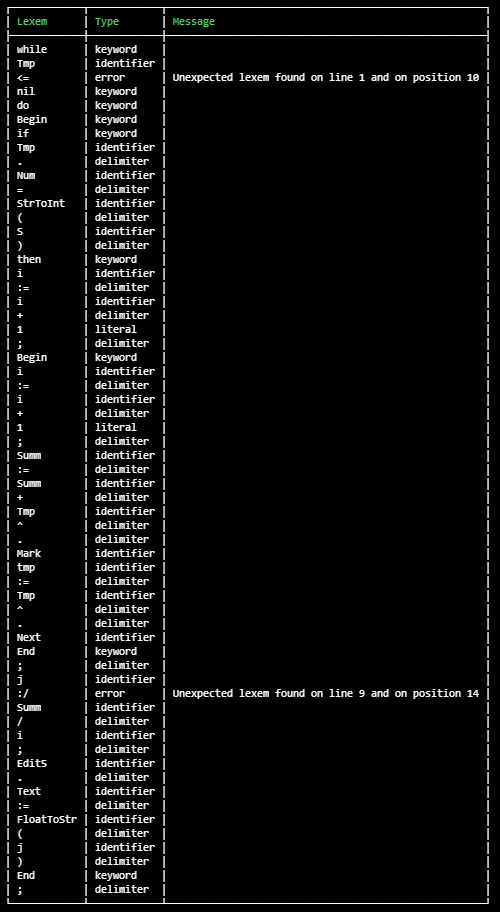


Рисунок 3 Результат работы сканера при наличии ошибок

# Выводы

После выполнения лабораторной работы были получены практические навыки построение сканера.

Данная работа была разделена на четыре этапа, а именно:

* Выделить все типы лексем из фрагмента кода
* Построить грамматику языка
* Построить конечный автомат
* Запрограммировать сканнер

Грамматика языка была построена при помощи BNF нотации, полученные знания из курса “Принципы языков программирования” помогли быстрее построить грамматические правила т. к. требовалось только освежить память, построение конечного автомата так же не вызвал особых проблем.

Полученные знания из данной лабораторной работы я буду использовать дальнейшем, потому что у меня был опыт использования компонента *symfony/expression-language* и после полученных знаний мне стало легче ориентироваться в исходном коде и появилось понимание почему данный компонент сделан так как он сделан.