

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Московский технологический университет»**

**МГУПИ**

Институт комплексной безопасности и специального приборостроения

**Кафедра КБ-3 Управления и моделирования систем**

**Разработка проблемно-ориентированных транслирующих средств**

**Отчет**

Студент 1 курса

заочного отделения

кафедра КБ-3

направление 09.04.04

Матросов Кирилл Дмитриевич

Преподаватель каф. КБ-3

«Управления и моделирования систем»:

Корягин С.В.

2017

Содержание

Введение

Целью работы является реализовать транслятор по БНФ форме.

Трансляция – преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдает для печати тексты программы и т.д.

Этапы трансляции:

1. Лексический анализ – процесс аналитического разбора входной последовательности символов (например, код на одном из языков программирования) с целью получения на выходе последовательности символов, называемых «токенами» (подобно группировке букв в словах). Группа символов входной последовательности, идентифицируемая на выходе процесса как токен называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.  
   Токен можно представить в виде структуры, содержащий идентификатор токена и если нужно, последовательность символов лексемы, выделенной из входного потока (строка, число и т.д.).
2. Синтаксический анализ (парсинг) – это процесс сопоставления последовательности лексем (слов, токенов) языка с его формальной грамматикой результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.
3. Преобразование дерева разбора в другой язык программирования, либо интерпретирование дерева разбора.

1. Цель работы

По заданной БНФ разработать синтаксически-управляемый транслятор, который будет выполнять следующие функции:

1. Анализ входного текста и определение его принадлежности к данной БНФ.
2. Определение места и типа первой ошибки во входном тексте.
3. Расчет выражений и вывод значений переменных.

2. Форма Бэкуса-Наура

Lan = "Program Defaults Interval Step Method Operations"

Defaults = "parametrs" var "=" num | "values" var "=" num

Interval = "interval" [real, real]

Step = "step" num

Method = "method" "eiler"|"runge2"|"runge4"

Operation = label "=" expression

expression = part <maths part>

part = expression | func"("var|num|part")

maths = a&s|m&d|exp

exp = "^"

m&d = "\*", "/"

a&s = "+", "-"

funcs = "cos"|"sin"

num = int|real

var = l<symbol\*>

label = symbol+"'"

symbol = l|n

real = int"."int

int = n...n

n = ["0"-"9"]

l = ["a"-"z"]

3. Список ошибок

1. Программа должна начинаться с оператора “Program”.
2. Отсутствуют дифференциальные уравнения.
3. Отсутствуют значения по умолчанию.
4. Оператор «parametrs» должен быть указан до оператора «values».
5. Оператор «values» должен быть указан до оператора «interval».
6. Оператор «interval» должен быть указан до оператора «step».
7. Оператор «step» должен быть указан до оператора «method».
8. Отсутствует значение по интервалу/шагу.
9. Правая часть в блоке «parametrs» должна быть числом.
10. За оператором «method» должен быть указан из методов: eiler, runge2, runge4
11. Пропущен знак «=».
12. Дублируется знак «=».
13. Не число
14. Количество открывающих скобок не соответствует количеству закрывающих скобок
15. Нет левой части
16. Нет правой части
17. Знак после открывающей скобки
18. Знак до закрывающей скобки
19. Повторяющий знак подряд n раз
20. Знак в конце строки

4. Блок-схемы

**Lan = "Program Defaults Interval Step Method Operations"**

|  |  |
| --- | --- |
|  | this.text = this.text.split('\n').filter(line => line !== '').join('\n');  this.text = this.text.split('\n');  for (let i = 0, len = this.text.length; i < len; i++) {  this.text[i] = this.text[i].trim();  }  this.text = this.text.join('\n');  const index = this.text.search('Program');  if (index === -1) {  throw new this.exceptions.programBlock()  }  let last\_block = {};  ['parametrs', 'values', 'interval', 'step', 'method'].forEach((block, i) => {  const index = this.text.search(block);  if (index === -1) {  throw new this.exceptions.missBlock(block)  }  if (i && index < last\_block.index) {  throw new this.exceptions.orderBlock(block, last\_block.block)  }  last\_block = {block, index};  });  this.getFunctions(this.text.split(/method.\*\n/gi)[1]);  this.getStep(this.text.match(/step\s\*(\d+\.?\d\*)\n/));  this.getInterval(this.text.match(/interval.\*\[(.+),(.+)\]\n/));  this.getMethod(this.text.match(/method(.\*)/));  this.build(); |

**Operation = label "=" expression (описание дифференциального уравнения)**

|  |  |
| --- | --- |
|  | if (!text)  {  throw new this.exceptions.noLines('functions');  }  let fs = text.trim().split('\n');  let \_\_functions = [];  fs.forEach((func) => {  this.checkSigns(func, '=');  let parts = func.split('=');  const name = parts[0].split(this.exprSign)[0].trim(), f = parts[1].trim();  if (name === '') {  throw new this.exceptions.noPart(func);  } else if (f === '') {  throw new this.exceptions.noPart(func, true);  }  else {  \_\_functions.push({name, f});  }  });  if (\_\_functions.length === 0)  {  throw new this.exceptions.noLines('functions');  }  \_\_functions.forEach(this.checkExpression);  this.functions = \_\_functions; |

**Step = "step" num**

|  |  |
| --- | --- |
|  | this.getStep(this.text.match(/step\s\*(\d+\.?\d\*)\n/));  getStep(text){  if (!text)  {  throw new this.exceptions.missValue('step');  }  this.isNumber(text ? text[1] : NaN);  this.step = parseFloat(text[1]);  } |

**Interval = "interval" [real, real]**

|  |  |
| --- | --- |
|  | this.getInterval(this.text.match(/interval.\*\[(.+),(.+)\]\n/));  getInterval(text){    if (!text)  {  throw new this.exceptions.missValue('interval');  }  this.isNumber(text[1]);  this.isNumber(text[2]);  this.interval = [parseFloat(text[1]), parseFloat(text[2])];  } |

**Method = "method" "eiler"|"runge2"|"runge4"**

|  |  |
| --- | --- |
|  | const METHODS = ['eiler', 'runge2', 'runge4'];  this.method = text[1].trim().toLowerCase();  if (!METHODS.includes  (this.method)) {  throw new this.exceptions.  noMethod(this.method);  } |

**"parametrs" var "=" num**

|  |  |
| --- | --- |
|  | this.getParams(this.text.match(/parametrs(\n.\*)+values/));  getParams(ps){  ps = ps[0].split('parametrs\n')[1];  ps = ps.split('\nvalues')[0];  ps = ps.split('\n');  ps.forEach((\_ps) => {  this.checkSigns(\_ps, '=');  const p = \_ps.split('='), left = p[0].trim(), right = p[1].trim();  if (left === '') {  throw new this.exceptions.noPart(\_ps);  } else if (right === '') {  throw new this.exceptions.noPart(\_ps, true);  }  else {  this.isNumber(right);  this.params[left] = parseFloat(right);  }  })} |

**"values" var "=" num**

|  |  |
| --- | --- |
|  | let values = this.text.split('values\n')[1].split('inter')[0].split('\n').filter(v => v !== '');  this.getDefaults(values);  getDefaults(ds){    if (ds.length === 0)  {  throw new this.exceptions.noLines('values');  }  if (this.functions.length !== ds.length) {  throw new this.exceptions.countDefault();  }  for (let i = 0, len = ds.length; i < len; i++) {  this.checkSigns(ds[i], '=');  let d = ds[i].split('='), left = d[0].trim(), right = d[1].trim();  let func = this.functions.find(f => f.name === left);  if (func) {  if (left === '') {  throw new this.exceptions.noPart(ds);  } else if (right === '') {  throw new this.exceptions.noPart(ds, true);  }  else {  this.isNumber(right);  func.def = parseFloat(right);  console.log(func.def)  }  }  }  } |

**label = symbol+"'"**

|  |  |
| --- | --- |
|  | let parts = func.split('=');  const name = parts[0].split(this.exprSign)[0].trim(), f = parts[1].trim();  if (name === '') {  throw new this.exceptions.noPart(func);  } else if (f === '') {  throw new this.exceptions.noPart(func, true);  } |

**a&s = "+", "-"**

|  |  |
| --- | --- |
|  | function *parseAddSubtract* (res, token, last) { var node, operators, name, fn, params;  node = *parseMultiplyDivide*();  operators = { '+': 'add', '-': 'subtract' }; while (operators.hasOwnProperty(token)) { name = token; fn = operators[name];  *getTokenSkipNewline*(); params = [node, *parseMultiplyDivide*()]; node = new OperatorNode(name, fn, params); }  res.calc(node, last); } |

**m&d = "\*", "/"**

|  |  |
| --- | --- |
|  | function parseMultiplyDivide (node) { var last, operators, name, fn;  last = node;  operators = { '\*': 'multiply',  '/': 'divide' };  while (true) { if (operators.hasOwnProperty(token)) { name = token; fn = operators[name];  getTokenSkipNewline();  last = parseUnary(); node = new OperatorNode(name, fn, [node, last]); } else if ((token\_type == TOKENTYPE.SYMBOL) || (token == 'in' && (node && node.isConstantNode)) || (token\_type == TOKENTYPE.NUMBER && !last.isConstantNode && (!last.isOperatorNode || last.op === '!')) || (token == '(')) {  last = parseUnary(); node = new OperatorNode('\*', 'multiply', [node, last], true ); } else { break; } }  return node; } |