МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Математическая логика»

Лабораторная работа №4 «Доработка компилятора CMiLan» Вариант **Цикл for. Тип №2.**

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Востров Алексей Владимирович

«____»_____ 20___ г.

Санкт-Петербург, 2025

Содержание

Введение

Язык *Милан* — учебный процедурный язык программирования. В исходной версии компилятор **CMilan** поддерживал циклическую конструкцию `while`, но отсутствовал удобный счётный цикл. В данной лабораторной работе компилятор расширен за счёт реализации оператора цикла

```
for <идент> := <нач>, <кон> [, <итерация>] do <список_операторов> od
```

Шаг не обязателен; если он опущен, используется значение `1`. Задача потребовала изменить исходный C++-код компилятора **CMilan**. Были решены следующие подзадачи:

- расширить лексический анализатор (добавить ключевое слово `for` и символ ‘`,`‘);
- дополнить грамматику языка;
- реализовать разбор цикла в синтаксическом анализаторе с одновременной генерацией p-кода;
- протестировать компилятор и виртуальную машину.

1 Математическое описание

1.1 Лексический и синтаксический анализ

Лексический анализатор – первый из "слоев"компилятора, отвечающий за выделение лексем для последующей обработки. Лексема – минимальная единица некоего словаря, представляющего наш язык. В роли лексемы могут служить служебные слова, операторы, идентификаторы и так далее. На Рис.1 представлена схема лексического анализатора.

Рис. 1: Схема лексического анализатора

Синтаксический анализ — процесс сопоставления линейной последовательности лексем естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора. Обычно применяется совместно с лексическим анализом. Синтаксический анализатор выражений (парсер) — часть программы, выполняющая чтение и анализ выражения. Существует два типа алгоритмов синтаксического анализа: нисходящий и восходящий:

- Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
- Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

Грамматика языка Милан использует нисходящий парсер. При восстановлении синтаксического дерева при нисходящем разборе слева направо последовательно анализирует все поддеревья, принадлежащие самому левому нетерминалу. Когда самым левым становится другой нетерминал, анализируется уже он. Компилятор языка Милан представляет собой транслятор, использующий метод рекурсивного спуска, и генератор p-кода. На Рис. 2 представлена полная схема компилятора Милан.

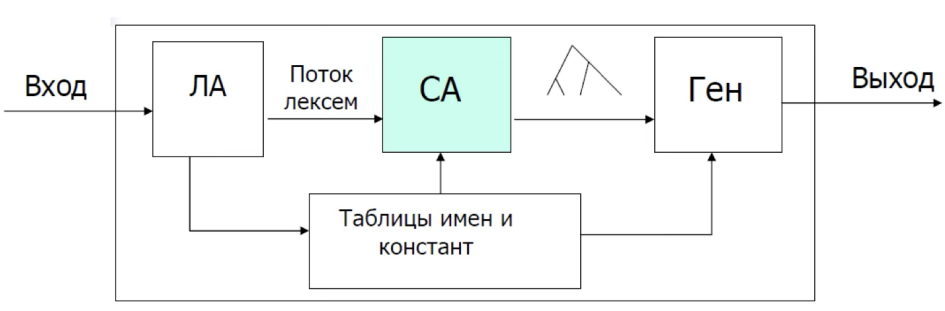

Рис. 2: Схема компилятора Милан

Грамматика языка Милан является LL(1) грамматикой, так как необходимо просмотреть поток всего на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить. Грамматика языка Милан является контекстно-свободной, так как удовлетворяет определению КС-грамматики, т.е. продукции грамматики имеют вид $A \to \alpha$, где A — одиночный нетерминал, а α — произвольная цепочка из терминалов и нетерминалов. Ниже представлена грамматика языка Милан.

$$
\begin{aligned}
&\text{Prog} \to \textbf{begin } L \textbf{ end} \\
&L \to S \mid L \ ; \ S \\
&S \to i_k := E \\
&\quad \mid \textbf{while } B \textbf{ do } L \textbf{ od} \\
&\quad \mid \textbf{if } B \textbf{ then } L \textbf{ fi} \\
&\quad \mid \textbf{if } B \textbf{ then } L \textbf{ else } L \textbf{ fi} \\
&\quad \mid \textbf{write } (E) \\
&B \to E \ q_k \ E \\
&E \to E \ aop_k \ T \mid T \\
&T \to T \ mop_k \ P \mid P \\
&P \to i_k \mid c_k \mid (E) \mid \textbf{read}
\end{aligned}
$$

2 Синтаксические диаграммы

Ниже представлены синтаксические диаграммы для модифицированной грамматики языка Милан. Синтаксические диаграммы соответствуют расширенной БНФ-нотации грамматики языка Милан, каждому нетерминалу соответствует своя синтаксическая диаграмма.

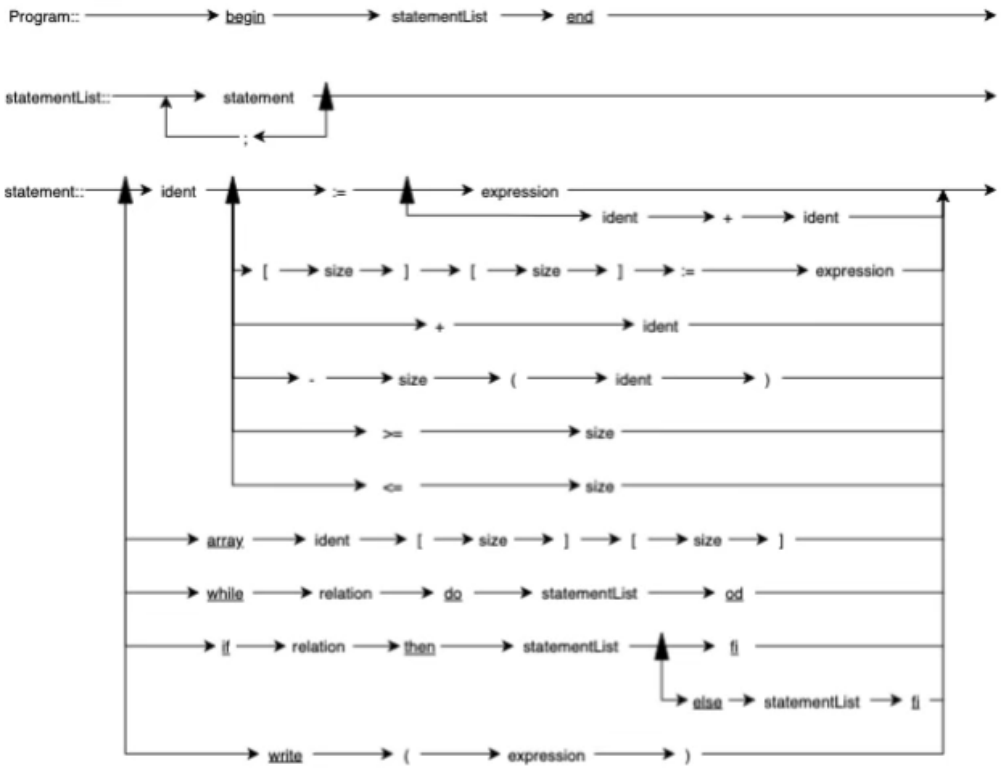

Рис. 3: Синтаксическая диаграмма

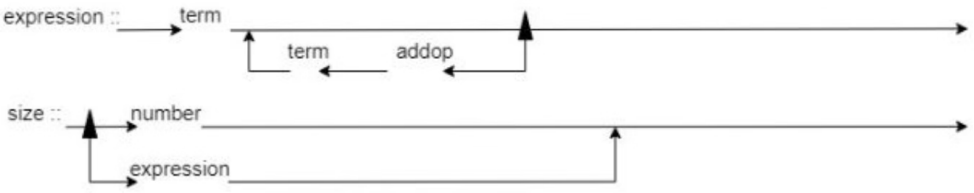

Рис. 4: Синтаксическая диаграмма для Term

Первая синтаксическая диаграмма соответствует продукции начального нетерминала (ключевые слова 'begin' и 'end' отвечают за начало и конец программы соотвественно). Между границами программы находится тело программы (продукция statementList, вторая диаграмма), которая может включать в себя одну или более строк, разделенных знаком ';'.

Синтаксическая диаграмма для реализованного цикла for представлена ниже.

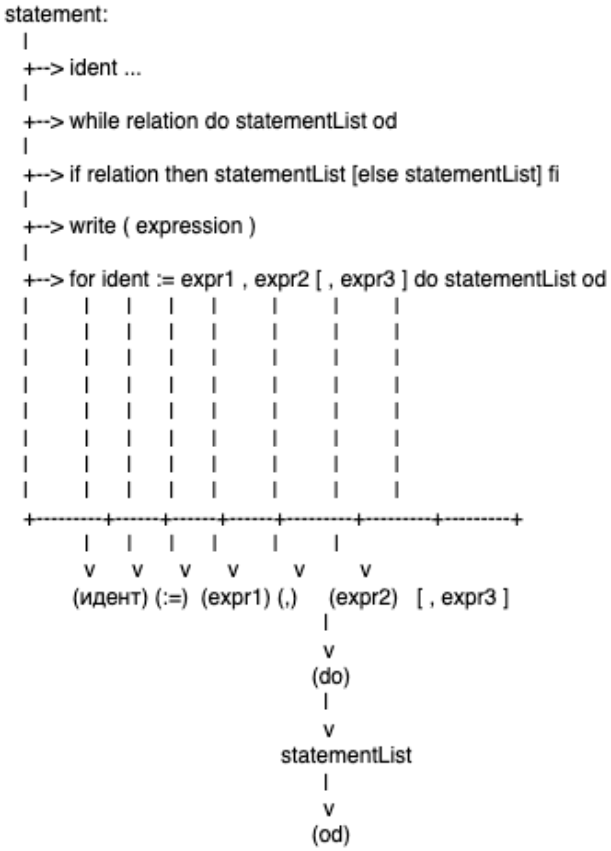

Рис. 5: Синтаксическая диаграмма после добавления for

3 Компилятор CMilan

Компилятор состоит из трёх основных компонент:

1. *лексический анализатор* (файл `scanner.*`);
2. *синтаксический анализатор* — рекурсивный нисходящий LL(1)-парсер (`parser.*`);
3. *генератор кода* для стековой виртуальной машины (`codegen.*`).

Лексер разбивает исходный текст на лексемы (`Token`), удаляя пробелы и комментарии. Парсер, получая поток лексем, проверяет его соответствие грамматике и одновременно формирует последовательность команд виртуальной машины (p-код). Так как грамматика Милана — LL(1), достаточно просмотра одного символа вперёд.

4 Формальная грамматика языка после расширения

4.1 Определение КС-грамматики

Расширенный язык задаётся КС-грамматикой $G = \langle N, \Sigma, P, S \rangle$.

- $N = \{\langle program \rangle, \langle statementList \rangle, \langle statement \rangle, \langle expression \rangle, \langle term \rangle, \langle factor \rangle, \langle relation \rangle\}$.
- Σ содержит все терминалы: ключевые слова {`begin`,`end`,`if`,`then`,`else`,`fi`,`while`,`do`,`od`,`for`,`write`,`read`}, служебные символы `':='` `','` `'('` `')'` `';'` и множество идентификаторов/чисел.
- $S = \langle program \rangle$.
- P приведено в БНФ ниже (дополнения выделены **жирным**).

4.2 БНФ (расширенная)

```
<program>       ::= 'begin' <statementList> 'end'

<statementList> ::= <statement> ';' <statementList>
                  | <statement>

<statement>     ::= <ident> ':=' <expression>
                  | 'if' <relation> 'then' <statementList>
                        ['else' <statementList>] 'fi'
                  | 'while' <relation> 'do' <statementList> 'od'
                  | **'for' <ident> ':=' <expression> ',' <expression>
                        [',' <expression>] 'do' <statementList> 'od'**
                  | 'write' '(' <expression> ')'

<expression>    ::= <term> { <addop> <term> }
<term>          ::= <factor> { <mulop> <factor> }
<factor>        ::= <number>
                  | <ident>
                  | '(' <expression> ')'
                  | '-' <factor>
                  | 'read'

<relation>      ::= <expression> <cmp> <expression>
```

5 Грамматика LL(1)

1) У нетерминала `<statement>` множества FIRST альтернатив пополняются лексемой `for`. Новая лексема не пересекается с уже существующими (`if, while, write, id`), поэтому свойство LL(1) не нарушается.

2) Внутри продукции цикла ключевое слово `do` однозначно разделяет заголовок и тело, а `od` помечает конец; это исключает конфликты при выборе правил для `<statementList>`.

3) Добавление терминалов `T_FOR` и « `,` » не создаёт пересечений $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta)$ для альтернатив одного нетерминала.

Следовательно, G остаётся контекстно-свободной LL(1) грамматикой; алгоритм рекурсивного нисходящего разбора не требует изменений.

6 Дерево разбора

На рисунке 6, показано дерево разбора.

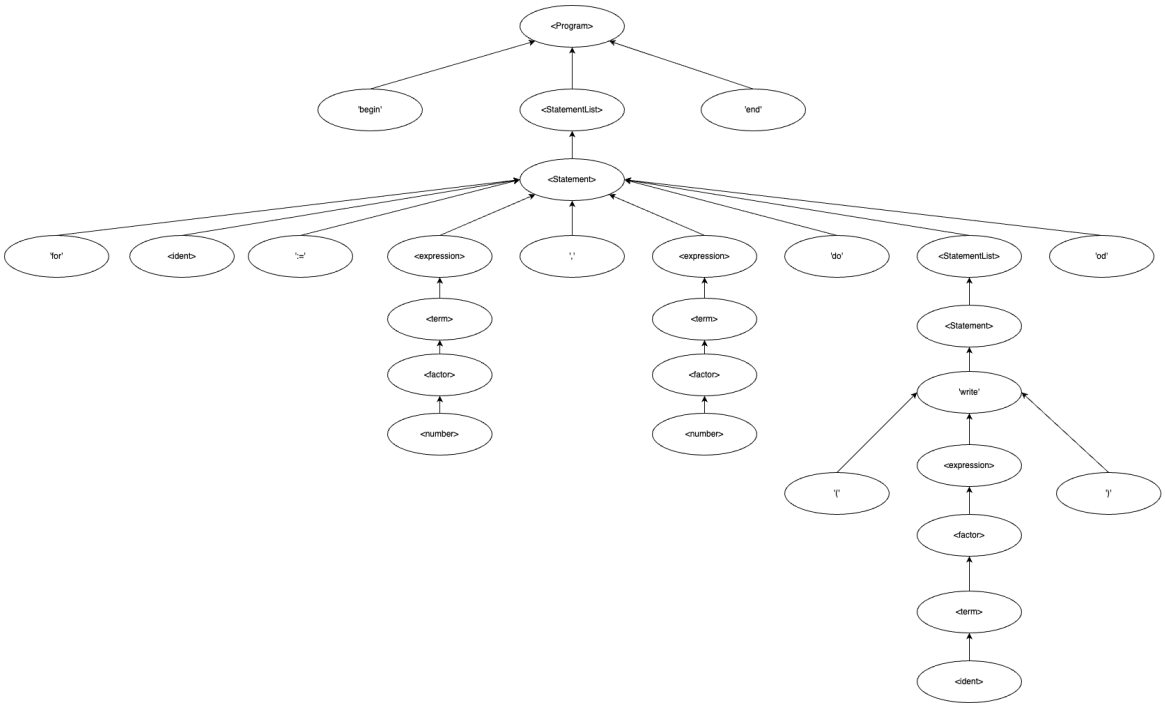

Рис. 6: Дерево разбора

7 Трансляционные правила

Обозначим: $addr(X)$ — адрес переменной цикла, ℓ — временная ячейка для предела, σ — ячейка шага, $gen(\cdot)$ — операция «сгенерировать команду VM».

7.1 Шаблон генерации p-кода для цикла

Инициализация:	$\text{gen}(E_0),\ \text{gen}(\text{STORE}, addr(X)),$ $\text{gen}(E_f),\ \text{gen}(\text{STORE}, \ell),$ $\text{gen}\big(E_s \text{ или } \text{PUSH}\,1\big),\ \text{gen}(\text{STORE}, \sigma)$
Условие:	$cond \leftarrow$ addr текущей команды, $\text{gen}(\text{LOAD}, addr(X)),\ \text{gen}(\text{LOAD}, \ell),\ \text{gen}(\text{COMPARE}, \le),$ $j \leftarrow \text{gen}\big(\texttt{JUMP\_NO}, Exit\big)$
Тело цикла:	$\ldots\ \text{gen}(p\text{-код списка } S)\ \ldots$
Инкремент:	$\text{gen}(\text{LOAD}, addr(X)),\ \text{gen}(\text{LOAD}, \sigma),\ \text{gen}(\text{ADD}),$ $\text{gen}(\text{STORE}, addr(X)),\ \text{gen}(\text{JUMP}, cond),$ $\text{gen}_{\text{at}}\big(j, \texttt{JUMP\_NO}, Exit\big)$

7.2 FIRST/FOLLOW

Для доказательства единственности выбора продукции рассчитаем множества FIRST и FOLLOW нетерминала `<statement>`.

Множество FIRST

$$\text{FIRST}(\langle statement \rangle) = \{id,\ \texttt{if},\ \texttt{while},\ \texttt{for},\ \texttt{write}\}$$

Каждый элемент отвечает уникальной продукции, поэтому FIRST-множество альтернатив поп pairwise disjoint.

Множество FOLLOW

$$\text{FOLLOW}(\langle statement \rangle) = \{\texttt{;},\ \texttt{else},\ \texttt{fi},\ \texttt{od},\ \texttt{end}\}$$

Получено стандартным алгоритмом, учитывая, что `<statementList>` $\Rightarrow^*$ `<statement>`.

Условие LL(1) Для двух альтернатив $A \to \alpha \mid \beta$ необходимо $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$; эти множества приведены выше и не пересекаются. Комплекты $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A)$ также пусты, так как ε-продукции отсутствуют. Следовательно, конфликтов в парсинговой таблице нет, и грамматика остаётся LL(1).

Таблица 1: Фрагмент таблицы после расширения

	id	`if`	`while`	`for`	`write`
`<statement>`	$X := E$	`if...`	`while...`	`for...`	`write`(E)

8 Новые токены

Для расширения грамматики и поддержки цикла `for` были добавлены новые токены в лексический анализатор и грамматику языка Милан:

- **T_FOR** — ключевое слово `for`. Используется для распознавания начала конструкции цикла for. В грамматике инициирует разбор соответствующей продукции:

```
<statement> ::= ... | 'for' <ident> ':=' <expression> ',' <expression> [','

<expression>] 'do' <statementList> 'od' | ...
```

- **T_COMMA** — символ запятой (`,`). Используется для разделения начального значения, конечного значения и (опционально) шага в заголовке цикла for.
- **T_DO**, **T_OD** — ключевые слова `do` и `od`. Обозначают начало и конец тела цикла (и других управляющих конструкций).
- **T_ASSIGN** — символ присваивания (`:=`). Используется для задания значения переменной цикла.
- **T_IDENTIFIER** — идентификатор переменной (имя переменной цикла).
- **T_NUMBER** — числовая константа (используется в выражениях).
- **T_WRITE**, **T_READ** — ключевые слова для ввода/вывода.
- **T_BEGIN**, **T_END** — ключевые слова для начала и конца программы.
- **T_IF**, **T_THEN**, **T_ELSE**, **T_FI** — ключевые слова для условных операторов.
- **T_WHILE** — ключевое слово для цикла while.
- **T_SEMICOLON**, **T_LPAREN**, **T_RPAREN** — служебные символы (точка с запятой, скобки).

Пример использования новых токенов в грамматике:

```
begin
    for i := 1, 10, 2 do
        write(i)
    od
end
```

Здесь лексер выделяет следующие токены:

```
T_FOR T_IDENTIFIER T_ASSIGN T_NUMBER T_COMMA T_NUMBER T_COMMA T_NUMBER T_DO

T_WRITE T_LPAREN T_IDENTIFIER T_RPAREN T_OD
```

Каждый из этих токенов участвует в разборе соответствующих продукций грамматики, обеспечивая корректную работу синтаксического анализатора и генерацию p-кода для новых конструкций языка.

9 Генерируемый p-код

Для программы

```
1  begin
2      for i := 1, 5 do
3          write(i)
4      od
5  end
```

компилятор выдаёт:

```
0:  PUSH    1
1:  STORE   0          ; i := 1
2:  PUSH    5
3:  STORE   1          ; limit
4:  PUSH    1
5:  STORE   2          ; step
6:  LOAD    0
7:  LOAD    1
8:  COMPARE 4          ; <=
9:  JUMP_NO 19
10: LOAD    0
11: PRINT
12: LOAD    0
13: LOAD    2
14: ADD
15: STORE   0
16: JUMP    6
19: STOP
```

10 Тестовые примеры

1. **demo2.mil**

```
    begin
        for k := 5, 5 do
            write(k)
        od
    end
```

Вывод: `5`

2. **demo3.mil**

```
    begin
        for x := 2, 8, 3 do
            write(x)
        od
    end
```

Вывод: `2 5 8`

3. **demo4.mil** (Вложенный цикл)

```
      begin
          for i := 1, 3 do
              for j := 1, 2 do
                  write(i)
                  write(j)
              od
          od
      end
```

Вывод: `1 1 1 2 2 1 2 2 2`

4. **demo5.mil** (Отрицательная итерация)

```
      begin
          for k := 5, 1, -2 do
              write(k)
          od
      end
```

Вывод: `5 3 1`

11 Инструкция по запуску

1. **Сборка компилятора**
   `cd cmilan/src && make`
2. **Компиляция исходников**
   `./cmilan ../test/demo2.mil > ../../demo2.asm`
3. **Сборка виртуальной машины**
   `cd ../../vm/vm && make`
4. **Запуск программы**
   `./mvm ../../demo2.asm`

Заключение

В ходе работы грамматика языка Милан была расширена, а компилятор **CMilan** модифицирован для поддержки счётного цикла `for`. Добавлены два токена (`T_FOR`, `T_COMMA`) и одна продукция грамматики; изменены файлы `scanner.h/.cpp` и `parser.cpp`. FIRST/FOLLOW-анализ подтвердил, что грамматика осталась LL(1), а семантические схемы гарантируют корректную генерацию p-кода. Тестовые программы успешно выполняются на виртуальной машине, что демонстрирует отсутствие регрессий в существующем функционале. Реализованы вложенные циклы.

Плюсы

- **Минимальные правки.** Лексер и парсер дополнены точечно; остальной код и интерфейсы остались неизменными.
- **Поддержка отрицательных итераций.** Так же реализованы циклы с обратным шагом.

Минусы

- *Нет проверки знака шага.* Если итерация указан нулём, цикл становится бесконечным — на уровне компиляции это не диагностируется.
- *Только линейный счётчик.* Условие окончания жёстко «$\leq$» или «$\geq$» в зависимости от знака шага; более сложные предикаты (например, `while i*i<n`) требуют обычного `while`.
- *Отсутствие поддержки `break`/`continue`.* Досрочный выход из цикла не предусмотрен.

Дополнение

- Добавить статическую проверку «итерация $\neq 0$» и предупреждение о неверном направлении счётчика.
- Реализовать ключевые слова `break` и `continue` с генерацией соответствующих `JUMP`-инструкций.
- Поддержать диапазонный синтаксис : `for i in 1..10 [step 2] do ... od`.
- Расширить виртуальную машину новым сравнением «`>= / <=` с шагом ± 1» для компактного p-кода.

Список литературы

1. Ю.Г. Карпов, Теория и технология программирования. Основы построения трансляторов. СПб.: БХВ-Петербург, 2005.
2. А. Ахо, М. Лам, Р. Сети, Дж. Ульман, Компиляторы: принципы, технологии и инструментарий, 2-е изд. М.: Вильямс, 2011.