Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

В.Т. Калайда, В.В. Романенко

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Учебное методическое пособие

2007

Корректор: Осипова Е.А.

Калайда В.Т., Романенко В.В.
Теория вычислительных процессов и структур: Учебное методическое пособие. — Томск: Томский межвузовский центр дистанционного образования, 2007. — 24 с.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Учебной программной специальности 230105 в рамках изучения дисциплины «Теория вычислительных процессов и структур» (ТВПиС) для студентов дистанционной формы обучения предусмотрено выполнение трех лабораторных работ:

1. Анализ описания переменных.
2. Разбор математического выражения.
3. Решение системы регулярных уравнений.

В данном методическом пособии изложены задания для лабораторных работ, в приложении — пример оформления титульного листа отчета по лабораторным работам.

Предполагается, что учащиеся хорошо владеют хотя бы одним высокоуровневым структурным (или иным) языком программирования и имеют хорошую математическую подготовку.

1 АНАЛИЗ ОПИСАНИЯ ПЕРЕМЕННЫХ

Создание программы для анализа описания переменных — цель выполнения лабораторной работы № 1.

1.1 Входные данные

На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий **только** описания переменных на выбранном языке (Pascal или C). Например, для языка Pascal содержание текстового файла может быть следующим:

```
var a, b, c: real;
d: array [1..6, 6..9] of integer;
s1: string;
s2: string[10];
```

То есть описание переменных начинается с ключевого слова «**var**», далее следуют списки имен переменных с указанием типа. В список типов необходимо включить наиболее часто используемые базовые типы. Тип может быть также массивом (в т.ч. многомерным) или строкой. У массива нижняя граница индекса не должна быть меньше верхней. Строка не может быть длиннее 255 символов.

Имя переменной — это последовательность букв и цифр, начинающаяся с буквы. Под буквами понимаются большие и малые буквы латинского алфавита (*a…z, A…Z*) и подчеркивание (_). Большинство современных компиляторов не имеют ограничений по длине имен переменных, однако значащими считаются только первые *N* символов (число *N* зависит от конкретного компилятора). Например, если $N = 8$, то переменные `a1234567` и `a12345678` рассматриваются компиляторами как идентичные, хотя обе записи являются синтаксически верными. Значение числа *N* можно выбрать любое ($N \geq 1$), но обычно это 8, 16, 32 и т.д.

В качестве разделителей, отделяющих друг от друга ключевые слова, имена переменных, знаки пунктуации и т.п., могут выступать:

– пробелы (код ASCII — 32 или $20 в шестнадцатеричном виде);

– переводы строк и возвраты кареток (коды ASCII — 10 ($0A) и 13 ($0D) соответственно);
– табуляции (код ASCII — 9 или $09).

Для указания символа по его коду в языке Pascal используется знак «#». Либо для этих целей можно использовать функцию CHR (см. ее описание в файле справки компилятора языка Pascal). При указании букв A…F в шестнадцатеричных числах можно использовать как большие, так и малые буквы. Так, для проверки символа ch на возврат каретки можно написать

```
if ch = #13 …
```

или

```
if ch = #$0D …
```

или же

```
if ch = chr(13) …
```

Для языка C файл может быть таким:

```
double a[10], b, c;
int x_q[5][5];
```

То есть описание состоит из указаний типов данных со следующими за ними списками имен переменных. Язык C поддерживает различные модификаторы типов — модификаторы размера (**long**/**short**), знака (**signed**/**unsigned**) и прочие (**auto**, **register**, **volatile**, **const**, **static**). Для упрощения задачи будем рассматривать только модификаторы размера. Обратите внимание, что модификатор может использоваться без указания базового типа, в этом случае в качестве базового типа подразумевается тип **int**. Например, запись

```
long
```

эквивалентна записи

```
long int
```

Модификатор **long** может использоваться с типами **int** и **double**, модификатор **short** — только с типом **int**.

Т.к. язык C не делает различий между символом и целым числом (байтом), то для проверки кода числа можно просто сравнить символ с кодом:

```
if(ch == 13) …
```

Если компилятор выдает в этом случае предупреждение о возможном несоответствии данных, можно использовать явное преобразование типа:

```
if(ch == char(13)) …
```

Либо можно использовать запись `'\OOO'` или `'\xHH'`, где `OOO` — восьмеричная запись кода символа (максимальный код — 377_8), `HH` — шестнадцатеричная (максимальный код — FF_{16}). Так, проверку на символ перевода строки можно также записать в таком виде:

```
if(ch == '\x0d') …
```

или

```
if(ch == '\15') …
```

В языке C существуют также специальные символы для обозначения некоторых непечатных элементов таблицы ASCII:

- `'\n'` — перевод строки (**n**ext line);
- `'\r'` — возврат каретки (carriage **r**eturn);
- `'\t'` — табуляция (**t**abulate);

Пробелы (как в Pascal, так и в C) можно обозначать просто в виде пробела, заключенного в кавычки (`' '`).

В отличие от языка Pascal в языке C размерность массива указывается в виде одной цифры (обязательно положительной), и каждая размерность заключается в квадратные скобки.

1.2 Выходные данные

Ваша программа должна проанализировать имеющиеся в текстовом файле описания переменных и выдать (в текстовый файл OUTPUT.TXT или на экран) результат проверки. Это может быть:

1. Сообщение о том, что описание корректное.
2. Сообщение о синтаксической ошибке (неправильные имена переменных, ошибки при использовании ключевых слов, неверные индексы массивов, отсутствие знаков пунктуации и т.д.). Указывать тип ошибки не обязательно, требуется только указать строку и позицию в строке входного файла, где наблюдается ошибка. Достаточно находить только первую ошибку в описании.

3. Сообщение о дублировании имен переменных. В этом случае на выходе программы необходимо указать имя дублируемой переменной, а также строку и позицию в строке, где встретился дубликат.

2 РАЗБОР МАТЕМАТИЧЕСКОГО ВЫРАЖЕНИЯ

Создание программы для разбора математического выражения — цель выполнения лабораторной работы № 2.

2.1 Входные данные

На вход программы подается текстовый файл (с именем INPUT.TXT), содержащий единственную строку символов. Данная строка задает присваивание переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = *ВЫРАЖЕНИЕ*.

Выражение может включать:

- Знаки сложения и умножения («+» и «*»);
- Круглые скобки («(» и «)»);
- Константы (например, 5; 3.8; 1e+18, 8.41E–10);
- Имена переменных.

Имя переменной — это последовательность букв и цифр, начинающаяся с буквы. Входное выражение считать правильным.

2.2 Краткая теория

Рассмотрим краткую теорию преобразования математического выражения в псевдокод, а также оптимизации кода. Более подробные данные можно получить в [1] (разделы 2.4-2.8).

В качестве примера возьмем выражение

COST = (*PRICE*+*TAX*)*0.98.

2.2.1 Лексический анализ

Проанализируем выражение:

- *COST*, *PRICE* и *TAX* — лексемы-идентификаторы;
- 0.98 — лексема-константа;
- =, +, * — просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем

<ИД$_1$>=(<ИД$_2$>+<ИД$_3$>)*<ИД$_4$>.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) — показана в виде индексов. Символы «=», «+» и «*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

2.2.2 Работа с таблицей имен

После того как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имен.

Для нашего примера *COST*, *PRICE* и *TAX* — переменные с плавающей точкой. Рассмотрим вариант такой таблицы. В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 2.1).

Таблица 2.1 — Таблица имен

Номер элемента	Идентификатор	Информация
1	*COST*	Переменная с плавающей точкой
2	*PRICE*	Переменная с плавающей точкой
3	*TAX*	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

2.2.3 Синтаксический анализ

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

Пример:

<ИД$_1$>=(<ИД$_2$>+<ИД$_3$>)*<ИД$_4$>.

По этой цепочке необходимо выполнить следующие действия:

1) <ИД$_3$> прибавить к <ИД$_2$>;
2) результат (1) умножить на <ИД$_4$>;
3) результат (2) поместить в ячейку, резервированную для <ИД$_1$>.

Этой последовательности соответствует дерево, изображенное на рис. 2.1.

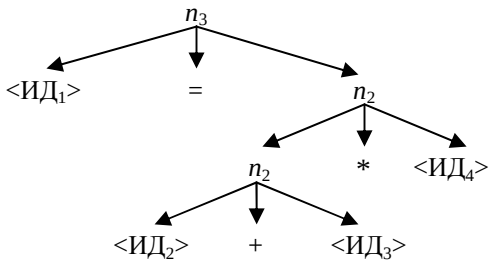

Рис. 2.1 — Последовательность действий
при вычислении выражения

То есть мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности знаки «+», «*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

2.2.4 Генерация кода

Дерево, построенное синтаксическим анализатором, используется для того, чтобы получить перевод входной программы. Рассмотрим машину с одним регистром и команды языка типа «ассемблер» (табл. 2.2).

Таблица 2.2 — Команды языка типа «ассемблер»

Команда	Действие
`LOAD M`	$C(m) \rightarrow$ сумматор
`ADD M`	C(сумматор) + $C(m) \rightarrow$ сумматор
`MPY M`	C(сумматор) * $C(m) \rightarrow$ сумматор
`STORE M`	C(сумматор) $\rightarrow m$
`LOAD =M`	$m \rightarrow$ сумматор
`ADD =M`	C(сумматор) + $m \rightarrow$ сумматор
`MPY =M`	C(сумматор) * $m \rightarrow$ сумматор

Запись «$C(m) \rightarrow$ сумматор» означает, что содержимое ячейки памяти m надо поместить в сумматор. Запись $=m$ означает численное значение m.

С помощью дерева, полученного синтаксическим анализатором, и информации, хранящейся в таблице имен, можно построить объектный код.

С каждой вершиной n связывается цепочка $C(n)$ промежуточного кода. Код для вершины n строится сцеплением в фиксированном порядке кодовых цепочек, связанных с прямыми потомками вершины n, и некоторых фиксированных цепочек. Процесс перевода идет, таким образом, снизу вверх (от листьев к корню). Фиксированные цепочки и фиксированный порядок задаются используемым алгоритмом перевода.

Здесь возникает важная проблема: для каждой вершины n необходимо выбрать код $C(n)$ так, чтобы код, приписываемый корню, оказывался искомым кодом всего оператора. Вообще говоря, нужна какая-то интерпретация кода $C(n)$, которой можно было бы единообразно пользоваться во всех ситуациях, где встретится вершина n.

Вернемся к исходному дереву (рис. 2.1). Есть три типа внутренних вершин, зависящих от того, каким из знаков помечен средний потомок: «=», «+» или «*» (рис. 2.2). Здесь треугольники — произвольные поддеревья (в том числе, состоящие из единственной вершины).

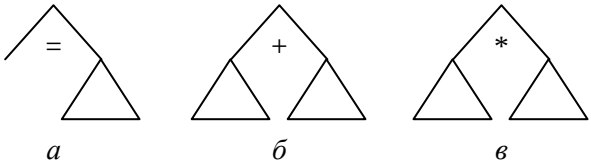

Рис. 2.2 — Типы вершин

Для любого арифметического оператора присвоения, включающего только арифметические операции «+» и «*», можно построить дерево с одной вершиной типа «а» и остальными вершинами только типов «б» и «в».

Код соответствующей вершины будет иметь следующую интерпретацию:

1) если n — вершина типа «а», то $C(n)$ будет кодом, который вычисляет значение выражения, соответствующее правому поддереву, и помещает его в ячейку, зарезервированную для идентификатора, которым помечен левый поток;
2) если n — вершина типа «б» или «в», то цепочка LOAD $C(n)$ будет кодом, засылающим в сумматор значение выражения, соответствующего поддереву, над которым доминирует вершина n.

Так, для нашего дерева код LOAD $C(n_1)$ засылает в сумматор значение выражения <ИД$_2$>+<ИД$_3$>, код LOAD $C(n_2)$ засылает в сумматор значение выражения (<ИД$_2$>+<ИД$_3$>)*<ИД$_4$>, а код $C(n_3)$ засылает в сумматор значение последнего выражения и помещает его в ячейку, предназначенную для <ИД$_1$>.

Теперь надо показать, как код $C(n)$ строится из кодов потомков вершины n. В дальнейшем мы будем предполагать, что операторы языка ассемблера записываются в виде одной цепочки и отделяются друг от друга точкой с запятой или началом новой строки. Кроме того, мы будем предполагать, что каждой вершине n дерева приписывается число $l(n)$, называемое уровнем, которое означает максимальную длину пути от этой вершины до листа, т.е. $l(n) = 0$, если n — лист, а если n имеет потомков $n_1, n_2, \ldots, n_k$, то

$$l(n) = \max_{1 \le i \le k} l(n_i) + 1.$$

Уровни $l(n)$ можно вычислить снизу вверх одновременно с вычислением кодов $C(n)$ (рис. 2.3).

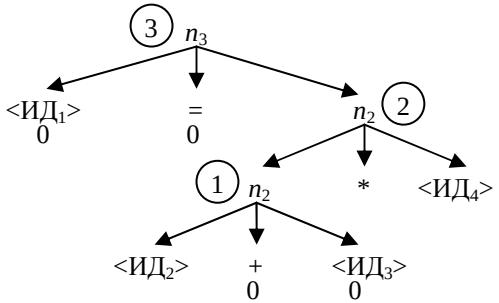

Рис. 2.3 — Дерево с уровнями

Уровни записываются для того, чтобы контролировать использование временных ячеек памяти. Две нужные нам величины нельзя заслать в одну и ту же ячейку памяти.

Теперь определим синтаксически управляемый алгоритм генерации кода, предназначенный для вычисления кода $C(n)$ всех вершин дерева, состоящих из листьев корня типа «а» и внутренних вершин типа «б» и «в».

Алгоритм.

Вход. Помеченное упорядоченное дерево, представляющее собой оператор присвоения, включающий только арифметические операции «*» и «+». Предполагается, что уровни всех вершин уже вычислены.

Выход. Код в ячейке ассемблера, вычисляющий этот оператор присвоения.

Метод. Делать шаги 1) и 2) для всех вершин уровня 0, затем для вершин уровня 1 и т.д., пока не будут отработаны все вершины.

1) Пусть n — лист с меткой <ИД$_i$>.
   1.1. Допустим, что элемент i таблицы идентификаторов является переменной. Тогда $C(n)$ — имя этой переменной.
   1.2. Допустим, что элемент j таблицы идентификаторов является константой k, тогда $C(n)$ — цепочка $=k$.
2) Если n — лист с меткой «=», «+» или «*», то $C(n)$ — пустая цепочка.
3) Если n — вершина типа «а» и ее прямые потомки — это вершины n_1, n_2 и n_3, то $C(n)$ — цепочка LOAD $C(n_3)$; STORE $C(n_1)$.

4) Если n — вершина типа «б» и ее прямые потомки — это вершины n_1, n_2 и n_3, то $C(n)$ — цепочка $C(n_3)$; STORE \$$l(n)$; LOAD $C(n_1)$; ADD \$$l(n)$. Эта последовательность занимает временную ячейку, именем которой служит знак «\$» вместе со следующим за ним уровнем вершины n. Непосредственно видно, что если перед этой последовательностью поставить LOAD, то значение, которое она поместит в сумматор, будет суммой значений выражений поддеревьев, над которыми доминируют вершины n_1 и n_3. Выбор имен временных ячеек гарантирует, что два нужных значения одновременно не появятся в одной ячейке.
5) Если n — вершина типа «в», а все остальное — как и в 4), то $C(n)$ — цепочка $C(n_3)$; STORE \$$l(n)$; LOAD $C(n_1)$; MPY \$$l(n)$.

Применим этот алгоритм к нашему примеру (рис. 2.4).

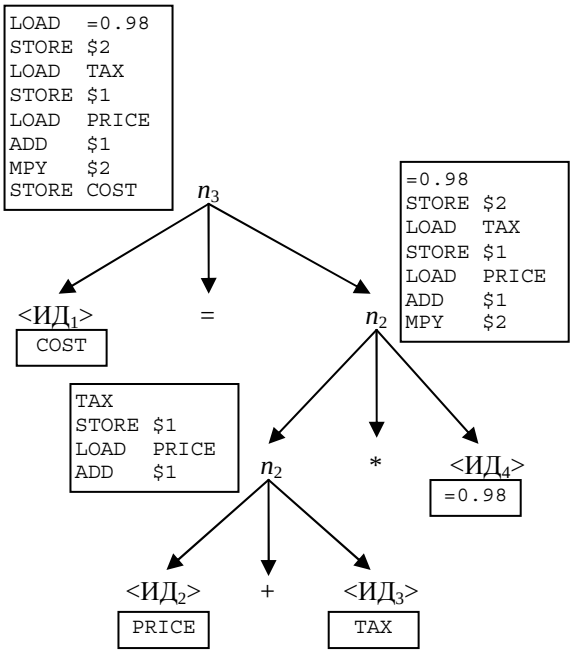

Рис. 2.4 — Дерево с генерированными кодами

Таким образом, в корне мы получили программу на языке типа «ассемблер», эквивалентную исходному выражению.

Естественно, эта программа далека от оптимальной, но это можно исправить на этапе оптимизации.

2.2.5 Оптимизация кода

Рассмотрим приемы, которые делают код более коротким:

1) Если «+» — коммутативная операция, то можно заменить последовательность команд LOAD α; ADD β; последовательностью LOAD β; ADD α. Требуется, однако, чтобы в других местах не было перехода к оператору ADD β.
2) Подобным же образом, если «*» — коммутативная операция, то можно заменить LOAD α; MPY β; на LOAD β; MPY α.
3) Последовательность операторов типа STORE α; LOAD α; можно удалить из программы при условии, что или ячейка α не будет использоваться далее, или перед использованием ячейка α будет заполнена заново.
4) Последовательность LOAD α; STORE β; можно удалить, если за ней следует другой оператор LOAD и нет перехода к оператору STORE β, а последующие вхождения β будут заменены на α вплоть до того места, где появится другой оператор STORE γ.

Получим оптимизированную программу для нашего примера (табл. 2.3).

Таблица 2.3 — Оптимизация кода

Этап 1	Этап 2	Этап 3

Применяем правило 1 к последовательности
`LOAD PRICE`
`ADD $1`
Заменяем ее последовательностью
`LOAD $1`
`ADD PRICE`
	Применяем правило 3 и удаляем последовательность
	`STORE $1`
	`LOAD $1`
		К последовательности
		`LOAD =0.98`
		`STORE $2`
		применяем правило 4 и удаляем ее. В команде
		`MPY $2`

Окончание табл. 2.3

Этап 1	Этап 2	Этап 3
		заменяем $2 на =0.98
LOAD =0.98 STORE $2 LOAD TAX STORE $1 LOAD $1 ADD PRICE MPY $2 STORE COST	LOAD =0.98 STORE $2 LOAD TAX ADD PRICE MPY $2 STORE COST	LOAD TAX ADD PRICE MPY =0.98 STORE COST

2.3 Выходные данные

В выходном файле (с именем OUTPUT.TXT) для исходного выражения, заданного во входном файле, необходимо привести:

1) таблицу имен;
2) неоптимизированный код;
3) оптимизированный код.

3 РЕШЕНИЕ СИСТЕМЫ РЕГУЛЯРНЫХ УРАВНЕНИЙ

Решение системы регулярных уравнений — цель выполнения лабораторной работы № 3.

3.1 Входные данные

Во входном файле (с именем INPUT.TXT) задается размерность системы регулярных уравнений n ($1 \le n \le 8$) а затем — ее коэффициенты:

$$\alpha_{10}\ \alpha_{11}\ \alpha_{12}\ \ldots\ \alpha_{1n}$$
$$\alpha_{20}\ \alpha_{21}\ \alpha_{22}\ \ldots\ \alpha_{2n}$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$\alpha_{n0}\ \alpha_{n1}\ \alpha_{n2}\ \ldots\ \alpha_{nn}$$

Максимальная длина регулярного выражения для каждого коэффициента равна 3.

3.2 Краткая теория

Рассмотрим краткую теорию решения уравнений с регулярными коэффициентами. Более подробные данные можно получить в [1] (раздел 3.5).

Определение. Регулярные выражения в алфавите Σ и регулярные множества, которые они обозначают, определяются рекурсивно следующим образом:

1) $\varnothing$ — регулярное выражение, обозначающее регулярное множество $\varnothing$;
2) e — регулярное выражение, обозначающее регулярное множество $\{e\}$;
3) если $a \in \Sigma$, то a — регулярное выражение, обозначающее регулярное множество $\{a\}$;
4) если p и q — регулярные выражения, обозначающие регулярные множества P и Q, то
   а) $(p+q)$ — регулярное выражение, обозначающее $P \cup Q$;
   б) pq — регулярное выражение, обозначающее $P \cap Q$;
   в) p^* — регулярное выражение, обозначающее P^*;
5) ничто другое не является регулярным выражением.

Принято обозначать p^+ для сокращенного обозначения pp^*. Расстановка приоритетов:

– * (итерация) — наивысший приоритет;
– конкатенация;
– + (объединение).

Например, $0 + 10^* = (0 + (1\ (0^*)))$.

Таким образом, для каждого регулярного множества можно найти регулярное выражение, его обозначающее, и наоборот.

Введем леммы, обозначающие основные алгебраические свойства регулярных выражений. Пусть α, β и γ регулярные выражения, тогда:

1) $\alpha + \beta = \beta + \alpha$
2) $\varnothing^* = e$
3) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
4) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
5) $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
6) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$
7) $\alpha e = e\alpha = \alpha$
8) $\alpha\varnothing = \varnothing\alpha = \varnothing$
9) $\alpha^* = \alpha + \alpha^*$
10) $(\alpha^*)^* = \alpha^*$
11) $\alpha + \alpha = \alpha$
12) $\alpha + \varnothing = \alpha$

При работе с языками часто удобно пользоваться уравнениями, коэффициентами и неизвестными которых служат множества. Такие уравнения будем называть *уравнениями с регулярными коэффициентами*

$$X = aX + b,$$

где a и b — регулярные выражения. Можно проверить прямой подстановкой, что решением этого уравнения будет a^*b:

$$aa^*b + b = (aa^* + e)b = a^*b,$$

т.е. получаем одно и то же множество. Таким же образом можно установить и решение системы уравнений.

Определение. Систему уравнений с регулярными коэффициентами назовем *стандартной системой* с множеством неизвестных $\Delta = \{X_1, X_2, \ldots, X_n\}$, если она имеет вид:

$$X_1 = \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + \ldots + \alpha_{1n}X_n$$
$$X_2 = \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2 + \ldots + \alpha_{2n}X_n$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$X_n = \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{n2}X_2 + \ldots + \alpha_{nn}X_n$$

где α_{ij} — регулярные выражения в алфавите, не пересекающемся с Δ. Коэффициентами уравнения являются выражения α_{ij}.

Если $\alpha_{ij} = \varnothing$, то в уравнении для X_i нет числа, содержащего X_j. Аналогично, если $\alpha_{ij} = e$, то в уравнении для X_i член, содержащий X_j, — это просто X_j. Иными словами, $\varnothing$ играет роль коэффициента 0, а e — роль коэффициента 1 в обычных системах линейных уравнений.

Алгоритм решения.

Вход. Стандартная система Q уравнений с регулярными коэффициентами в алфавите Σ и множеством неизвестных $\Delta = \{X_1, X_2, \ldots, X_n\}$.

Выход. Решение системы Q.

Метод: Аналог метода решения системы линейных уравнений методом исключения Гаусса.

Шаг 1. Положить $i = 1$.

Шаг 2. Если $i = n$, перейти к шагу 4. В противном случае с помощью тождеств леммы записать уравнения для X_i в виде

$$X_i = \alpha X_i + \beta,$$

где α — регулярное выражение в алфавите Σ, а β — регулярное выражение вида

$$\beta_0 + \beta_{i+1}X_{i+1} + \ldots + \beta_n X_n,$$

причем все β_i — регулярные выражения в алфавите Σ. Затем в правых частях для уравнений $X_{i+1}, \ldots, X_n$ заменим X_i регулярным выражением $\alpha^*\beta$.

Шаг 3. Увеличить i на 1 и вернуться к шагу 2.

Шаг 4. Записать уравнение для X_n в виде $X_n = \alpha X_n + \beta$, где α и β — регулярные выражения в алфавите Σ. Перейти к шагу 5 (при этом $i = n$).

Шаг 5. Уравнение для X_i имеет вид $X_i = \alpha X_i + \beta$, где α и β — регулярные выражения в алфавите Σ. Записать на выходе $X_i = \alpha^*\beta$, в уравнениях для $X_{i-1}, \ldots, X_1$ подставляя $\alpha^*\beta$ вместо X_i.

Шаг 6. Если $i = 1$, остановиться, в противном случае уменьшить i на 1 и вернуться к шагу 5.

3.3 Выходные данные

В выходной файл (с именем OUTPUT.TXT) необходимо вывести:

1) Полное решение системы регулярных уравнений.
2) Упрощенное решение.

Упрощенное решение получается, если применить к полученному решению леммы 1—12.

СПИСОК ЛИТЕРАТУРЫ

1. Калайда В.Т. Теория вычислительных процессов и структур: Учеб. пособие. — Томск: ТМЦДО, 2007. — 269 с.

ПРИЛОЖЕНИЕ А. ТИТУЛЬНЫЙ ЛИСТ ОТЧЕТА

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Томский межвузовский центр дистанционного образования (ТМЦДО)

Кафедра автоматизированных систем управления (АСУ)

НАЗВАНИЕ РАБОТЫ

Отчет по лабораторной работе №X по дисциплине
«Теория вычислительных процессов и структур»

Выполнил: _____

_____

«____» _____ 2007

Проверил: _____

_____

«____» _____ 2007

Томск — 2007

Содержание