**МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

В. Т. Калайда

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Методическое пособие для студентов специальности 220400
«Программное обеспечение вычислительной техники
и автоматизированных систем»

2007

Методическое Пособие рассмотрено и рекомендовано к изданию методическим семинаром кафедры автоматизированных систем управления ТУСУР « » 2007 г.

Зав. кафедрой АСУ
проф. д-р техн. наук _____А.М. Кориков

СОДЕРЖАНИЕ

Введение 6
1 Схемы программ 9
1.1 Предварительные математические сведения 9
1.1.1 Функции и графы 9
1.1.2 Вычислимость и разрешимость 11
1.1.3 Программы и схемы программ 15
1.2 Стандартные схемы программ 16
1.2.1 Базис класса стандартных схем программ 16
1.2.2 Графовая форма стандартной схемы 18
1.2.3 Линейная форма стандартной схемы 19
1.2.4 Интерпретация стандартных схем программ 20
1.3 Свойства и виды стандартных схем программ 23
1.3.1 Эквивалентность, тотальность, пустота, свобода 23
1.3.2 Свободные интерпретации 25
1.3.3 Согласованные свободные интерпретации 26
1.3.4 Логико-термальная эквивалентность 27
1.4 Моделирование стандартных схем программ 29
1.4.1 Одноленточные автоматы 29
1.4.2 Многоленточные автоматы 31
1.4.3 Двухголовочные автоматы 32
1.4.4 Двоичный двухголовочный автомат 33
1.5 Рекурсивные схемы 36
1.5.1 Рекурсивное программирование 36
1.5.2 Определение рекурсивной схемы 37
1.6 Трансляция схем программ 39
1.6.1 О сравнении класс сов схем. 39
1.6.2 Схемы с процедурами 40
1.7 Обогащенные и структурированные схемы 41
1.7.1 Классы обогащенных схем 41
1.7.2 Трансляция обогащенных схем 43
1.7.3 Структурированные схемы 43
Контрольные вопросы 45
2 Семантическая теория программ 45
2.1 Описание смысла программ 45
2.2 Операционная семантика 46
2.3 Аксиоматическая семантика 49
2.3.1 Преобразователь предикатов 50
2.3.2 Аксиоматическое определение операторов языка программирования 52
2.4 Денотационная семантика 55

2.5 Декларативная семантика 59
2.6 Языки формальной спецификации 59
2.7 Верификация программ 62
2.7.1 Методы доказательства правильности программ 62
2.7.2 Использование утверждений в программах 65
2.7.3 Правила верификации К. Хоара 67
Контрольные вопросы 70
3 Теоретические модели вычислительных процессов 70
3.1 Взаимодействующие последовательные процессы 70
3.1.1 Базовые определения 70
3.1.2 Законы взаимодействия последовательных процессов 74
3.1.3 Реализация процессов 75
3.1.4 Протоколы поведения процесса 76
3.1.5 Операции над протоколами 76
3.1.6 Протоколы процесса 79
3.1.7 Спецификации 80
3.2 Параллельные процессы 82
3.2.1 Взаимодействие процессов 83
3.2.2 Параллелизм 83
3.2.3 Задача об обедающих философах 85
3.2.4 Помеченные процессы 87
3.2.5 Множественная пометка 88
3.3 Взаимодействие – обмен сообщениями 88
3.3.1 Ввод и вывод 89
3.3.2 Взаимодействия 90
3.3.3 Подчинение 90
3.4 Разделяемые ресурсы 92
3.4.1 Поочередное использование 93
3.4.2 Общая память 93
3.4.3 Кратные ресурсы 95
3.4.4 Планирование ресурсов 98
3.5 Программирование параллельных вычислений 100
3.5.1 Основные понятия 100
3.5.2 Многопоточная обработка 102
3.5.3 Условные критические участки 103
3.5.4. Мониторы 103
3.6 Модели параллельных вычислений 105
3.6.1 Процесс/канал 106
3.6.2 Обмен сообщениями 106
3.6.3 Параллелизм данных 107
3.6.4 Общей памяти 108

Контрольные вопросы 108
4 Моделирование взаимодействия процессов. Сети Петри 109
4.1 Введение в сети Петри 109
4.2 Основные определения 109
4.2.1 Теоретико-множественное определение сетей Петри 109
4.2.2 Графы сетей Петри 110
4.2.3 Маркировка сетей Петри 111
4.2.4 Правила выполнения сетей Петри 112
4.3 Моделирование систем на основе сетей Петри 113
4.3.1 События и условия 113
4.3.2 Одновременность и конфликт 114
4.4 Моделирование параллельных систем взаимодействующих процессов 115
4.4.1. Моделирование последовательных процессов 115
4.4.2 Моделирование взаимодействия процессов 117
4.4.3 Задача о взаимном исключении 118
4.4.4 Задача о производителе/потребителе 119
4.4.5 Задача об обедающих философах 120
4.5 Анализ сетей Петри 121
4.5.1 Свойства сетей Петри 121
4.5.2 Методы анализа. 123
4.5.3 Анализ свойств сетей Петри на основе дерева достижимости 127
4.5.3 Матричные уравнения 128

Введение

Термин «теоретическое программирование» обозначает математическую дисциплину, изучающую синтаксические и семантические свойства программ, их структуру, преобразования, процесс их составления и исполнения. Это словосочетание построено по аналогии с названиями таких наук, как теоретическая физика, теоретическая механика и т. д. Теоретическая научная дисциплина изучает фундаментальные понятия и законы основной науки и на основании обнаруженных закономерностей строит математические модели исследуемых объектов, на которых ставит и решает прикладные задачи. В нашем случае ситуация усложняется тем, что объект моделирования – программа – уже представляет собой абстрактный объект.

В настоящее время сложились следующие основные направления исследований теоретического программирования.

1. *Математические основы программирования.* Основная цель исследований – развитие математического аппарата, ориентированного на теоретическое программирование, разработка общей теории машинных вычислений. Эта теория тесно соприкасается с теорией алгоритмов и вычислимых функций, теорией автоматов и формальных языков, логикой, алгеброй, с теорией сложности вычислений.
2. *Теория схем программ*. В этих работах внимание концентрируется на изучении структурных свойств и преобразований программ, а именно тех, которые отличают программы от других способов задания алгоритмов. Главным объектом исследования становится схема программы – математическая модель программы, в которой с той или иной степенью детализации отражено строение программы, взаимодействие составляющих ее компонентов.
3. *Семантическая теория программ*. Семантика программы или отдельных конструкций языков программирования – это их смысл, математический смысл для программиста и описание функционирования для машины. Этот раздел теоретического программирования изучает методы формального описания семантики программ, семантические методы преобразования и доказательства утверждений о программах. В частности, работы по методам проверки семантической правильности программ нацелены на автоматизацию их отладки и автоматический синтез программ.
4. *Теория вычислительных процессов и структур (теория параллельных вычислений).* Исследования в этой области направлены на разработку и обоснование новых методов программирования, прежде

всего методов программирования параллельных процессов. Изучаются модели, структуры и функционирование операционных систем, методы распараллеливания алгоритмов и программ. Исследуются новые архитектурные принципы конструирования вычислительных машин и систем на основе результатов и рекомендаций теоретического программирования и вычислительной математики.

5. *Прикладные задачи теоретического программирования*. Сюда в первую очередь относятся разработка и обоснование алгоритмов трансляции и алгоритмов автоматической оптимизации программ.

Две дисциплины государственного стандарта специальности 220400 – Программное обеспечение вычислительной техники и автоматизированных систем – «Теория языков программирования и методы трансляции» и «Теория вычислительных процессов» рассматривают основы теоретического программирования. Первая дисциплина охватывает первый и последний пункты нашей, не претендующей на классификационную строгость и полноту, рубрикации. Вторая дисциплина, составляющая предмет настоящего курса, раскрывает пункты 2 – 4.

Целью программирования является описание процессов обработки данных (*процессов*). *Данные* - это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а *информация* - это смысл, который придается данным при их представлении. *Обработка данных* - это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на *носителях* данных. Совокупность носителей данных, используемых при какой-либо обработке данных, будем называть *информационной средой*. Набор данных, содержащихся в какой-либо момент в информационной среде, будем называть *состоянием* этой информационной среды. *Процесс* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс - значит определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался *автоматически* на компьютере, необходимо, чтобы это описание было *формализованным*. Такое описание называется *программой*. С другой стороны, программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на понятном человеку формализованном *языке программирования*, с которого она автоматически переводится на язык соответствующего ком-

пьютера с помощью другой программы, называемой *транслятором*. *Программисту* приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы и многое другое.

Под «программой» часто понимают правильную программу, т. е. программу, не содержащую ошибок, соответствующую спецификации и дающую возможность формального вывода программы из формального набора предпосылок. Однако понятие ошибки в программе трактуется программистами неоднозначно. Будем считать, что в программе имеется *ошибка*, если она не выполняет того, что разумно ожидать от нее на основании документации по применению программы. Следовательно, правильнее говорить о несогласованности между программами и документацией по их применению.

В связи с тем, что задание на программу обычно формулируется не формально, а также из-за неформализованности понятия ошибки в программе, нельзя доказать формальными методами (математически) правильность программы. Нельзя доказать правильность программы и тестированием: как указал Дейкстра, тестирование может лишь продемонстрировать наличие в программе ошибки.

Альтернативой правильной программы является *надежная* программа. *Надежность* программы - это ее способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под *отказом* в программе понимают проявление в нем ошибки. Таким образом, надежная программа не исключает наличия в ней ошибок — важно лишь, чтобы эти ошибки при практическом применении этой программы в заданных условиях проявлялись достаточно редко. Убедиться, что программа обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении. Таким образом, фактически мы можем разрабатывать лишь надежные, а не правильные программы.

Разрабатываемая программа может обладать различной степенью надежности. Как измерять эту степень? Так же как в технике, степень надежности можно характеризовать вероятностью работы программы без отказа в течение определенного периода времени. Однако в силу специфических особенностей программ определение этой вероятности наталкивается на ряд трудностей.

При оценке степени надежности программ следует также учитывать последствия каждого отказа. Некоторые ошибки в программах могут вызывать лишь некоторые неудобства при его применении, тогда как

другие ошибки могут иметь катастрофические последствия, например, угрожать человеческой жизни. Поэтому для оценки надежности программных средств иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

1 Схемы программ

1.1 Предварительные математические сведения

1.1.1 Функции и графы

Ведем некоторые соглашения об обозначениях элементов теории множеств и логики.

Множество – есть набор несовпадающих объектов, которые будем задавать явным перечислением, и заключать в фигурные скобки. Например: D - множество дней недели:

$D = \{Пн,\ Вт,\ Ср,\ Чт,\ Пт,\ Сб, Вс\ \}, D_1 = \{Вт,\ Ср,\ Чт,\ Пн,\ Сб,\ Пт,\ Вс\}$

D и D_1, т. е. порядок элементов не важен.

Будем использовать обозначения:

$x \in A$ - x есть элемент, и принадлежит множеству A;

$x \notin A$ - x не является элементом множества A.

Для бесконечных множеств метод перечисления элементов множества не применим, для этого используется характеристическое свойство $A = \{x | p(x)\}$, где x – переменная, значениями которой являются некоторые объекты, а p – свойство тех и только тех значений x, которые являются элементами задаваемого множества.

Пустое множество - множество, которое не содержит ни одного элемента, обозначается $\varnothing$.

Если каждый элемент множества A является элементом множества B, то множество A является подмножеством множества B, будем писать $A \subset B$.

Если хотя бы один элемент множества A не является элементом множества B, то множество A не является подмножеством множества B и это записывается $A \not\subset B$.

Декартовым произведением $A_1 \times A_2 \times \ldots \times A_n$ *множеств* $A_1, A_2, \ldots, A_n$ называется множество $\{(a_1, a_2, \ldots, a_n) | a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n\}$, а A^n обозначает $A \times A \times \ldots \times A$ (n раз).

Функцией, отображающей множество X *во множество* Y ($F : X \to Y$) называется множество $F \subseteq X \times Y$ такое, что для любых пар $(x, y) \in F$ и $(x', y') \in F$ из $x = x'$ следует, что $y = y'$.

Множество $\{x|(x,y)\in F\}$ – *область определения* функции F (множество значений ее аргумента); множество $\{y|(x,y)\in F\}$ – *область значений* функции F.

Функцию $F: X \to Y$ называют *n-местной функцией* над множеством A, если $Y = A$ и $X = A^n$.

Предикатом называют функцию, областью значений которой является множество символов-цифр $\{0,1\}$. Предикат $P: X \to \{0,1\}$ *истинен* для $x \in X$ если $P(x) = 1$, и *ложен,* если $P(x) = 0$. Отношение на множестве X – это двухместный предикат $P: X^2 \to \{0,1\}$.

Алфавитом называют непустое конечное множество символов. Например, $V_1 = \{a,b\}$, $V_2 = \{0,1\}$, $V_3 = \{a,+,1,=\}$ – алфавиты. *Словом в алфавите V* называется конечный объект, получаемый выписыванием одного за другим символов V, например, $a+1=1+a$ – слово в алфавите V_3, 101011 – слово в алфавите V_2, *abaab* – слово в алфавите V_1. *Длина слова* – число символов в нем, *пустое слово* не содержит ни одного символа.

Множество всех слов в алфавите V обозначается V^*.

n-местной словарной функцией над алфавитом V называют *n* - местную функцию над V^*, т. е. функцию из $V^* \times V^* \times \ldots \times V^*$ (*n* раз) в V^*.

Направленным графом называется тройка $G = (V, E, \Phi)$, где V - множество вершин, E – множество дуг, а Φ – функция из E в $\left(V \cup \{\omega\}\right)^2$, $\omega \notin V$. Дуга e называется *входом* графа, если $\Phi(e) = (\omega, v)$, для $v \in V \cup \{\omega\}$; *внутренней*, если $\Phi(e) = (v_1, v_1)$ для $v_1, v_2 \in V$; *выходом*, если $\Phi(e) = (v, \omega)$), для $v \in V \cup \{\omega\}$. Дуга e, являющаяся одновременно и входом и выходом графа, называется *висячей;* для нее $\Phi(e) = (\omega, \omega)$. Дуги, не являющиеся внутренними, называются также свободными.

Дуга e *инцидентна* вершине v, если e выходит из v или ведет в v. Две дуги *смежные*, если существует хотя бы одна инцидентная им обеим вершина. Вершина v называется *наследником* вершины v', если в графе имеется хотя бы одна такая дуга, что $\Phi(e) = (v', v)$.

Изображенный на рисунке 1.1 граф G_1 содержит 4 вершины, 8 дуг. Дуга e_1 – входная, дуга e_6 – выходная, дуга e_8 – висячая, остальные дуги внутренние; вершины v_2 и v_3 наследники вершины v_1.

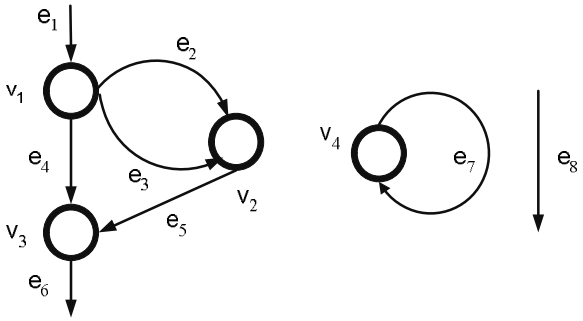


Рис. 1.1. Граф G_1

Путем в графе G называется последовательность $v_i e_i v_{i+1} \ldots$ дуг и вершин, такая, что для всех i $\Phi(e_i) = (v_i, v_{i+1})$. Образом пути называется слово, составленное из пометок проходимых дуг и вершин.

Две вершины v_1, v_2 графа G называются *связанными*, если $v_1 = v_2$ или существует маршрут $e_1, \ldots, e_n$ графа G такой, что дуга e_1 инцидентна вершине v_1, а дуга e_n – вершине v_2.

1.1.2 Вычислимость и разрешимость

Приведенное выше определение функции не содержит указаний о том, как для заданных значений аргументов получить соответствующие значения функции. Целесообразно переформулировать это определение таким образом, чтобы оно содержало конструктивную процедуру, или алгоритм, нахождения значений функции.

Тьюринг формализовал способ получения значений вычислимой функции с помощью абстрактной «математической машины», вычисляющей значения функции по значениям ее аргументов. Конкретная машина Тьюринга задает конкретную вычислимую функцию и его гипотеза (тезис) состояла в том, что каждая функция, для которой существует алгоритм нахождения ее значений, представима некоторой машиной Тьюринга, т. е. является вычислимой. Тезис Тьюринга не может быть доказан, так как наряду с формальным понятием вычислимой функции он содержит эмпирическое понятие алгоритма.

Машина Тьюринга T задает словарную функцию над некоторым алфавитом V и представляет собой описание машины - набор

$(F, Q, q_0, \#, I)$ - и правило функционирования, общее для всех машин, где

V - алфавит машины;

Q - конечное непустое множество символов, называемых состояниями машины $(Q \cap V = \varnothing)$;

q_0 - выделенный элемент множества Q, называемый начальным состоянием;

- специальный «пустой» символ, не принадлежащий ни V, ни Q;

I - программа машины.

Программа машины - это конечное множество слов вида $qa \to q'a'd$, называемых *командами,* где $q, q \in Q$, $a, a' \in V \cup \{\varnothing\}$; $\to$ - вспомогательный символ-разделитель; d - элемент множества $\{l, r, p\}$, содержащего три специальных символа, которых нет ни в V, ни в Q. В программе I никакие две команды не могут иметь одинаковую пару первых двух символов.

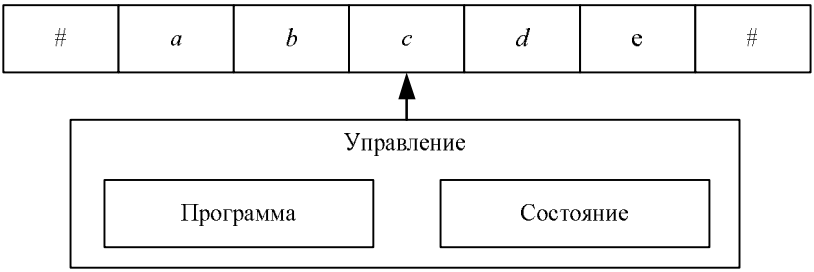


Рис. 1.2 Машина Тьюринга

Машина состоит из потенциально бесконечной ленты, управления и головки, перемещаемой вдоль ленты (см. рисунок 1.2). Лента разбита на клетки, которые могут содержать символы из алфавита V или быть пустыми, т. е. содержать символ #. Управление на каждом шаге работы машины находится в одном из состояний Q, расшифровывает программу, которая однозначно определяет поведение машины и управляет головкой. Головка в каждый момент расположена против некоторой клетки ленты и может считывать символы с ленты, записывать их на ленту и перемещаться в обе стороны вдоль ленты. Машина функционирует следующим образом. В начальный момент на ленте записано некоторое слово из V, а управление находится в начальном состоянии q_0. Начальное слово, равно как и слова, появляющиеся в процессе работы

машины, ограничено с двух сторон пустыми символами #. Головка обозревает крайний слева символ заданного слова.

Работа машины состоит в повторении следующего цикла элементарных действий:

1) считывание символа, находящегося против головки;
2) поиск применимой команды, а именно той команды $qa \to q'a'd$, в которой q - текущее состояние управления, a - считанный символ;
3) выполнение найденной команды, состоящее в переводе управления в новое состояние q', записи в обозреваемую головкой клетку символа a' (вместо стираемого символа a) и последующем перемещении головки вправо, если $d = r$, влево, если $d = l$, или сохранении ее в том же положении, если $d = p$.

Машина останавливается в том и только в том случае, если на очередном шаге ни одна из команд не применима. Результат работы остановившейся машины - заключительное слово на ленте.

Машина Тьюринга, перерабатывая начальные слова на ленте в заключительные, задает словарную функцию, для которой начальные слова - значения аргумента, заключительные - значения функции. (Для представления n-местной функции начальное слово на ленте имеет вид $\#a_1\#a_2\#\ldots\#a_n\#$, где подслова $a_1, a_2, \ldots, a_n$ не содержат символа #). Если машина не останавливается, начав работу с некоторым словом на ленте, то функция, задаваемая машиной, считается неопределенной для этого слова. Таким образом, машина Тьюринга T задает частичную функцию F_T и способ вычисления ее значений. Хотя машины Тьюринга оперируют со словами, они могут задавать и числовые функции в силу установленной выше связи между словами и числами.

По определению, функция F является (частично) вычислимой, если существует машина Тьюринга T такая, что $F_T = F$. Для машины Тьюринга, как и для всех других формальных способов задания алгоритмов, включая программы для ЭВМ, характерны следующие свойства:

1) конструктивность - машина Тьюринга представляет собой конечный объект, построенный по определенным правилам из базовых объектов;
2) конечность - процесс нахождения значений функции для тех значений аргументов, для которых она определена, состоит из конечного числа шагов;

3) однозначность - результат работы машины единственным образом определяется начальным словом;
4) массовость - машина работает с любым начальным словом на ленте, составленным из символов ее алфавита.

Машина Тьюринга однозначно задается своей программой. Если упорядочить ее команды и закодировать последовательность команд словом в алфавите машины Тьюринга, то можно получить ее описание в собственном алфавите.

При изучении свойства программ и их математических абстракций – схем программ, целью является автоматизация программирования, в том числе автоматический анализ свойств программ и их преобразования, осуществляемые с помощью других, специальных программ. Поэтому интересуют алгоритмы, которые могли бы по любой предъявленной программе установить, завершит ли она работу или будет «циклить», дают ли две программы, исходная и оптимизированная, один и тот же результат, является ли программа синтаксически правильной и т. д.

Массовые алгоритмические проблемы формулируются следующим образом. Необходимо указать алгоритм, который бы определял, обладает ли предъявленный объект из некоторого класса объектов интересующим нас свойством, принадлежит ли он множеству M всех объектов, обладающих этим свойством. Если существует такой частичный алгоритм, то говорят, что множество *перечислимо*, а поставленная массовая алгоритмическая проблема *частично разрешима*. Если этот алгоритм к тому же всюду определен, то множество M *разрешимо* и поставленная *проблема* также *разрешима*. Существуют неразрешимые проблемы и даже проблемы, которые не являются частично разрешимыми, что свидетельствует о существовании невычислимых функций.

Пусть V – алфавит, $M \subseteq V$ – множество слов в V.

Характеристической функцией множества M называется предикат $F_M : V^* \to \{0,1\}$, всюду определенный на $V^* : F_M(a) = 1$ если $a \in M$, и $F_M = (a) = 0$, если $a \notin M$.

Частичная характеристическая функция множества M – это функция $H_M : V^* \to \{1\}$, определенная только для слов из M и имеющая вид $H_M(a) = 1$ для всех $a \in M$.

Множество M называется *разрешимым*, если его характеристическая функция вычислима. Множество M *перечислимо,* если его частичная характеристическая функция вычислима. Разрешимость множества M означает, что существует всегда останавливающаяся (окан-

чивающая работу за конечное время) машина Тьюринга, позволяющая для любого слова в алфавите V через конечное число шагов установить, принадлежит ли это слово множеству M или нет. Перечислимость множества M означает, что существует машина Тьюринга, которая останавливается в том и только том случае, если предъявленное слово принадлежит множеству M.

Приведем без доказательства несколько важных теорем.

Теорема (*Пост*). Множество $M \subseteq V^*$ разрешимо тогда и только тогда, когда M и его дополнение $M' = V^* \setminus M$ перечислимы.

Машина Тьюринга, начав работу, или останавливается, или работает бесконечно. Проблема остановки формулируется следующим образом. Пусть M – множество всех пар слов в алфавите V, в каждой паре первое слово – словарное представление некоторой машины Тьюринга, второе – такое слово, что эта машина останавливается, начав работу над ним. Является ли множество M неразрешимым?

Теорема (*Тьюринг*). Проблема остановки машины Тьюринга неразрешима.

Последняя теорема демонстрирует существование невычислимых функций. Из тезиса Тьюринга следует, что для неразрешимых проблем нельзя построить алгоритм, который решал бы их, например, с помощью ЭВМ.

Проблема зацикливания состоит в следующем: существует ли алгоритм, хотя бы частный, который выясняет заранее для произвольной машины Тьюринга, будет ли она работать бесконечно.

Теорема. Проблема зацикливания машины Тьюринга не является частично разрешимой.

1.1.3 Программы и схемы программ

Схемы программ - это математические модели программ, описывающие строение программы, или точнее строение множества программ, где конкретные операции и функции заменены абстрактными функциональными и предикатными символами. Следующий пример программ вычисления факториала $n!$ и переворачивания слов поясняет различие между программами и их схемой S_1.

```
  begin integer x,y;
  ввод(x);
  y:=1;
L:if x=0 then
  goto L1;
  y:=x*y;
  x:=x-1;
  goto L;
L1:вывод(y);
  end
```

```
  begin integer x,y;
  ввод(x);
  y:=ε;
L:if x=0 then
  goto L1;
  y:=CONSCAR(x,y);
  x:=COR(x);
  goto L;
L1:вывод(y);
  end
```

```
  begin
  ввод(x);
  y:=a;
L:if p(x) then
  goto L1;
  y:=g(x,y);
  x:=h(x);
  goto L;
L1:вывод(y);
  end
```

Функция CONSCAR (суперпозиция функций CONS и CAR из языка Лисп) приписывает первую букву первого слова ко второму слову (т. е. CONSCAR (аб, в)=ав), а функция CAR стирает первую букву слова (т. е. CAR (аб)=б).

1.2 Стандартные схемы программ

1.2.1 Базис класса стандартных схем программ

Стандартные схемы программ (**ССП**) характеризуются базисом и структурой схемы. Базис класса фиксирует символы, из которых строятся схемы, указывает их роль (переменные, функциональные символы и др.), задает вид выражений и операторов схем.

Полный *базис B класса стандартных схем* состоит из 4-х непересекающихся, счетных множеств символов и множества операторов - слов, построенных из этих символов.

Множества символов полного базиса:

1) $X = \{x, x_1, x_2, \ldots, y, y_1 y_2, \ldots, z, z_1, z_2, \ldots\}$ - множество символов, называемых *переменными*;
2) $F = \{f^{(0)}, f^{(1)}, f^{(2)}, \ldots, g^{(0)}, g^{(1)}, g^{(2)}, \ldots, h^{(0)}, h^{(1)}, h^{(2)} \ldots\}$ - множество *функциональных символов*; верхний символ задает *местность символа*; нульместные символы называют константами и обозначают начальными буквами латинского алфавита $a, b, c, \ldots$;
3) $P = \{p^{(0)}, p^{(1)}, p^{(2)}, \ldots, q^{(0)}, q^{(1)}, q^{(2)}, \ldots\}$ - множество *предикатных символов*; $p^{(0)}, q^{(0)}$ - нульместные символы называют логическими константами;
4) **{start, stop,...,:=** и т. д.**}** - множество специальных символов.

Термами (*функциональными выражениями*) называются слова, построенные из переменных, функциональных и специальных символов по следующим правилам:

1) односимвольные слова, состоящие из переменных или констант, являются термами;
2) слово τ вида $f^{(n)}(\tau_1,\tau_2,\ldots,\tau_n)$, где $\tau_1,\tau_2,\ldots,\tau_n$ - термы, является термом;
3) те и только те слова, о которых говорится в п. п. 1,2, являются термами.

Примеры термов: $x, f^{(0)}, a, f^{(1)}(x), g^{(2)}(x,\ h^{(2)}(y,\ a))$.

Тестами (логическими выражениями) называются логические константы и слова вида $p^{(n)}(\tau_1,\ \tau_2,...,\tau_n)$.

Примеры: $p^{(0)}$, $p^{(0)}(x)$, $g^{(3)}(x,\ y,z)$, $p^{(2)}(f^{(2}(x,y))$.

Допускается в функциональных и логических выражениях опускать индексы местности, если это не приводит к двусмысленности или противоречию.

Множество операторов включает пять типов:

1) *начальный оператор* - слово вида **start**$(x_1,x_2,...,x_к)$, где $k\geq 0$, а $x_1,x_2,...,x_к$ - переменные, называемые результатом этого оператора;
2) *заключительный оператор* - слово вида **stop**$(\tau_1,\tau_2...\tau_n)$, где $n\geq 0$, а $\tau_1,\tau_2,\ldots,\tau_n$ - термы; вхождения переменных в термы τ называются *аргументами* этого *оператора*;
3) *оператор присваивания* - слово вида $x:=\tau$, где x – переменная (*результат оператора*), а τ - терм; вхождения переменных в термы называются *аргументами* этого *оператора*;
4) *условный оператор* (тест) - логическое выражение; вхождения переменных в логическое выражение называются *аргументами* этого *оператора*;
5) *оператор петли* - односимвольное слово **loop**.

Среди операторов присваивания выделим случаи: когда τ - переменная, то оператор называется *пересылкой* $(x:=y)$ и когда τ - константа, то оператор называется *засылкой* $(x:=a)$.

Подклассы используют ограниченные базисы. Так, например, подкласс Y_1 имеет базис:

$\{x_1,x_2\}$, $\{a,f^{(1)}\}$, $\{p^{(1)}\}$, {**start**, **stop**, `(,),:=,,`}и множество операторов {**start**(`x`$_1$`,x`$_2$`);` `x`$_1$`:=f(x`$_1$`),` `x`$_2$`:=f(x`$_2$`),` `x`$_1$`:=a,`

x_2:=a, p(x_1), p(x_2), **stop**(x_1, x_2)}, т. е. схемы из этого подкласса используют две переменные, константу a, один одноместный функциональный символ, один предикатный символ и операторы указанного вида.

1.2.2 Графовая форма стандартной схемы

Представим стандартную схему программ как размеченный граф, вершинам которого приписаны операторы из некоторого базиса B.

Стандартной схемой в базисе B называется конечный (размеченный ориентированный) граф без свободных дуг и с вершинами следующих пяти видов:

1) *Начальная вершина* (ровно одна) помечена начальным оператором. Из нее выходит ровно одна дуга. Нет дуг, ведущих к начальной вершине.
2) *Заключительная вершина* (может быть несколько). Помечена заключительным оператором. Из нее не выходит ни одной дуги.
3) *Вершина-преобразователь.* Помечена оператором присваивания. Из нее выходит ровно одна дуга.
4) *Вершина-распознаватель.* Помечена условным оператором (называемым условием данной вершины). Из нее выходит ровно две дуги, помеченные 1 (левая) и 0 (правая).
5) *Вершина-петля.* Помечена оператором петли. Из нее не выходит ни одной дуги.

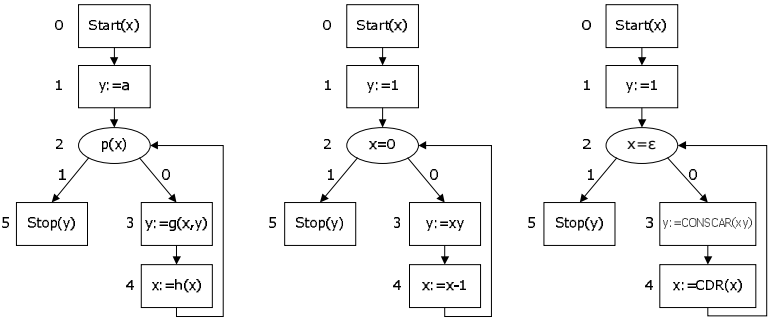
Конечное множество переменных схемы S составляют ее память X_S.

Из определения следует, что один и тот же оператор может помечать несколько вершин схемы. Вершины именуются (метки вершины) целым неотрицательным числом $(0,1,2,\ldots)$. Начальная вершина всегда помечается меткой 0.

Схема S называется правильной, если на каждой дуге заданы все переменные.

Пример правильной **ССП** S_1 в графовой форме приведен на рисунке 1.3, а).

Вершины изображены прямоугольниками, а вершина-распознаватель - овалом. Операторы записаны внутри вершины (рис. 1.3).



а *б* *в*

Рис. 1.3 Стандартные схемы программ

1.2.3 Линейная форма стандартной схемы

Для использования линейной формы **СПП** множество специальных символов расширим дополнительными символами {**:, goto, if, then, else**}. **СПП** в линейной форме представляет собой последовательность *инструкций*, которая строится следующим образом:

1) если выходная дуга начальной вершины с оператором $\mathbf{start}(x_1, x_2, ..., x_k)$ ведет к вершине с меткой L, то начальной вершине соответствует инструкция:
   0: **start(**x_1**, . . . ,** x_n**) goto** L;
2) если вершина схемы S с меткой L - преобразователь с оператором присваивания x:=τ, выходная дуга которого ведет к вершине с меткой L1, то этому преобразователю соответствует инструкция:
   L: x:=τ **goto** L1;
3) если вершина с меткой L - заключительная вершина с оператором **stop** (τ$_1$, . . . τ$_m$), то ей соответствует инструкция
   L: **stop**(τ$_1$, . . . , τ$_m$);
4) если вершина с меткой L - распознаватель с условием p(τ$_1$, . . . τ$_k$), причем 1-дуга ведет к вершине с меткой L1, а 0-дуга — к вершине с меткой L0, то этому распознавателю соответствует инструкция
   L: **if** p(τ$_1$, . . . τ$_k$) **then** L1 **else** L0;
5) если вершина с меткой L - петля, то ей соответствует инструкция
   L: **loop**.

Обычно используется сокращенная запись (опускание меток). Полная и сокращенная линейные формы **ССП** приведены ниже

```
0: start(x) goto 1;
1: y:=a goto 2;
2: If p(x) then 5 else 3;
3: y:=g (x, y) goto 4;
4: x:=h(x) goto 2;
5: stop(y);
```

а)

```
   start(x);
   y:=a;
2: If p(x) then 5 else 3;
3: y:=g(x, y);
   x:=h(x) goto 2;
5: stop(y);
```

б)

1.2.4 Интерпретация стандартных схем программ

ССП не является записью алгоритма, поэтому позволяет исследовать только структурные свойства программ, но не семантику вычислений. При построении «семантической» теории схем программ вводится понятие интерпретация **ССП**. Определим это понятие.

Пусть в некотором базисе B определен класс **ССП**. *Интерпретацией базиса* B *в области интерпретации* D называется функция I, которая сопоставляет:

1) каждой переменной x из базиса B - некоторый элемент $d = I(x)$ из области интерпретации D;
2) каждой константе a из базиса B - некоторый элемент $d = I(a)$ из области интерпретации D;
3) каждому функциональному символу $f^{(n)}$ - всюду определенную функцию $F = I\left(f^{(n)}\right)$;
4) каждой логической константе p^0 - один символ множества $\{0,1\}$;
5) каждому предикатному символу $p^{(n)}$ - всюду определенный предикат $P^{(n)} = I\left(p^{(n)}\right)$.

Пара (S, I) называется *интерпретированной стандартной схемой* (**ИСС**), или *стандартной программой* (**СП**).

Определим понятие *выполнения программы*.

Состоянием памяти программы (S, I) называют функцию $W : X_S \to D$, которая каждой переменной x из памяти схемы S сопоставляет элемент $W(x)$ из области интерпретации D.

Значение терма τ при интерпретации I и состоянии памяти W (обозначим $\tau_I(W)$) определяется следующим образом:

1) если $\tau = x$, x – переменная, то $\tau_I(W) = I(x)$;

2) если $\tau = a$, a – константа, то $\tau_I(W) = I(a)$;

3) если $\tau = f^{(n)}(\tau_1, \tau_2, \ldots, \tau_n)$, то $\tau_I(W) = I\left(f^{(n)}\left(\tau_{1I}(W), \tau_{2I}(W), \ldots, \tau_{nI}(W)\right)\right)$.

Аналогично определяется значение теста π при интерпретации I и состоянии памяти W или $\pi_I(W)$:

если $\pi = p^{(n)}(\tau_1, \tau_2, \ldots, \tau_n)$, то $\pi_I(W) = I\left(p^{(n)}\left(\tau_{1I}(W), \tau_{2I}(W), \ldots, \tau_{nI}(W)\right)\right)$, $n \geq 0$.

Конфигурацией программы называют пару $U = (L, W)$, где L - метка вершины схемы S, а W - состояние ее памяти. Выполнение программы описывается конечной или бесконечной последовательностей конфигураций, которую называют протоколом выполнения программы (**ПВП**).

Протокол $(U_0, U_1, \ldots, U_i, U_{i+1}, \ldots)$ выполнения программы (S, I) определяем следующим образом (ниже k_i означает метку вершины, а W_i - состояние памяти в i — й конфигурации протокола, $U_i = (k_i, W_i)$):

$U_0 = (0, W_0)$ – начальное состояние памяти схемы S при интерпретации I.

Пусть $U_i = (k_i, W_i)$ - i — я конфигурация ПВП, а O - оператор схемы S в вершине с меткой k_i. Если O - заключительный оператор **stop** (τ1, . . . , τm), то U_i - последняя конфигурация, так что протокол конечен. В этом случае считают, что, программа (S, I) *останавливается,* а последовательность значений $\tau_{1I}(W), \tau_{2I}(W), \ldots, \tau_{nI}(W)$ объявляют *результатом* $val(S, I)$ *выполнения программы* (S, I). В противном случае, т. е. когда O - не заключительный оператор, в протоколе имеется следующая, $(i+1)$-я конфигурация $U_{i+1} = (k_{i+1}, W_{i+1})$, причем

a) если O - начальный оператор, а выходящая из него дуга ведет к вершине с меткой L, то $k_{i+1} = L$ и $W_{i+1} = W_i$;

b) если O - оператор присваивания $x := \tau$, а выходящая из него дуга ведет к вершине с меткой L, то $k_{i+1} = L$, $W_{i+1} = W_i$, $W_{i+1}(x) = \tau_I(W_i)$;

c) если O - условный оператор π и $\pi_I(W_i) = \Delta$, где $\Delta \in \{0,1\}$, а выходящая из него дуга ведет к вершине с меткой L, то $k_{i+1} = L$ и $W_{i+1} = W_i$;

d) если O - оператор петли, то $k_{i+1} = L$ и $W_{i+1} = W_i$, так что протокол бесконечен.

Таким образом, программа останавливается тогда и только тогда, когда протокол ее выполнения конечен. В противном случае программа *зацикливается* и результат ее выполнения не определен.

Рассмотрим две интерпретации **СПП** S_1 (рисунок 1.3, б). Интерпретация (S_1, I_1) задана так:

- область интерпретации $D_1 \subseteq Nat$ - подмножество множества Nat целых неотрицательных чисел;
- $I_1(x) = 4$; $I_1(y) = 0$; $I_1(a) = 1$;
- $I_1(g) = G$, где G - функция умножения чисел, т. е. $G(d_1, d_2) = d_1 \cdot d_2$;
- $I_1(h) = H$, где H - функция вычитания единицы, т. е. $H(d) = d - 1$;
- $I_1(p) = P_1$, где P_1 - предикат «равно 0», т. е. $P_1(d) = 1$, если $d = 0$.

Программа (S_1, I_1) вычисляет 4! (рисунок 1.3, б).

Интерпретация (S_1, I_2) задана следующим образом:

- область интерпретации $D_2 = V^*$, где $V = \{a, b, c\}$, V^* - множество всех возможных слов в алфавите V.
- $I_2(x) = abc$;
- $I_2(y) = \varepsilon$, где ε - пустое слово;
- $I_2(a) = \varepsilon$;
- $I_2(g) = \mathtt{CONSTAR}$;
- $I_2(h) = \mathtt{CDR}$;
- $I_2(p) = P_2$, где P_2 - предикат «равное ε», т. е. $P_2(\alpha) = 1$, если $\alpha = \varepsilon$.

Программа (S_1, I_2) преобразует слово *abc* в слово *cba* (рисунок 1.3, в)

ПВП (S_1, I_1) и (S_1, I_2) конечен, результат - 24 и - *cba* (таблица 1.1 и 1.2).

Таблица 1.1 Протокол выполнения программы (S_1, I_1)

Конфигурация		U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7	U_8	U_9	U_{10}	U_{11}	U_{12}	U_{13}
Метка		0	1	2	3	4	2	3	4	2	3	4	2	3	
Значения	x	4	4	4	4	3	3	3	2	2	2	1	1	1	0
	y	0	1	1	4	4	4	12	12	12	24	24	24	24	24

Таблица 1.2 Протокол выполнения программы (S_1, I_2)

Конфигурация		U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7	U_8	U_9	U_{10}	U_{11}	U_{12}
Метка		0	1	2	3	4	2	3	4	2	3	4	2	5
Значения	x	*abc*	*abc*	*abc*	*abc*	*bc*	*bc*	*bc*	*c*	*c*	*c*	ε	ε	ε
	y	ε	ε	ε	*a*	*a*	*a*	*ba*	*ba*	*ba*	*cba*	*cba*	*cba*	*cba*

1.3 Свойства и виды стандартных схем программ

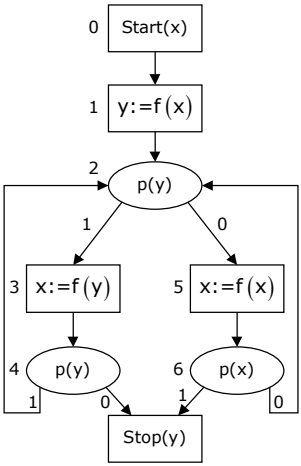
1.3.1 Эквивалентность, тотальность, пустота, свобода

ССП S в базисе B *тотальна*, если для любой интерпретации I базиса B программа (S, I) останавливается.

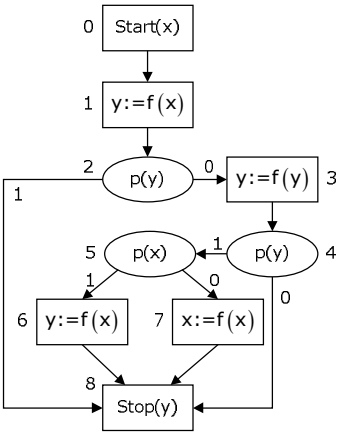
ССП S в базисе B *пуста*, если для любой интерпретации I базиса B программа (S, I) зацикливается

Стандартные схемы S_1 и S_2 в базисе B *функционально эквивалентны* $(S_1 \square S_2)$, если либо обе зацикливаются, либо обе останавливаются с одинаковым результатом, т. е. $val(S_1, I) = val(S_2, I)$.

Примеры тотальных (*а*), пустых (*б*) и эквивалентных (*в, г*) схем приведены на рисунке 1.4.



а *б*



в *г*

Рис. 1.4 Примеры схем программ

Цепочкой стандартной схемы (**ЦСС**) называют:

1) конечный путь по вершинам схемы, ведущий от начальной вершины к заключительной;
2) бесконечный путь по вершинам, начинающийся начальной вершиной схемы.

В случае, когда вершина-распознаватель, то дополнительно указывается верхний индекс (1 или 0), определяющий 1-дугу или 0-дугу, исходящую из вершины.

Примеры цепочек схемы S_1 (рисунок 1.3, *а*):
(0, 1, 2^1, 5); (0, 1, 2^0, 3, 4, 2^0,3,4,2^1,5) и т. д.

Цепочкой операторов (**ЦО**) называется последовательность операторов, метящих вершины некоторой цепочки схемы.

Например для S_1: (**start**(x), y:=a, p^1(x), **stop**(y)) или (**start**(x), y:=a, p^0(x), y:=g(x, y), x:=h(x), p^0(x), y:=g(x, y), x:=h(x), p^0(x), y:=g(x, y), x:=h (x), …)) и т. д.

Предикатные символы **ЦО** обозначаются так же, как вершины распознавателей в **ЦСС**.

Пусть S - **ССП** в базисе B, I - некоторая его интерпретация, $(0,1,2,\ldots,k_1,k_2,k_3,\ldots)$ - последовательность меток инструкций S, выписанных в том порядке, в котором эти метки входят в конфигурации протокола выполнения программы (S,I). Ясно, что эта последовательность – цепочка схемы S. Считают, что интерпретация I *подтверждает* (*порождает*) эту цепочку.

ЦСС в базисе B называют *допустимой*, если она подтверждается хотя бы одной интерпретацией этого базиса.

Не всякая **ЦСС** является допустимой. В схеме S_1 (рисунок 1.4, *а*) цепочки (0, 1, 2^0, 5, 6^1, 7), (0, 1, 2^1, 3, 4^0, 7) и все другие конечные цепочки не подтверждаются ни одной интерпретацией.

Свойство допустимости цепочек играет чрезвычайно важную роль в анализе **ССП.** В частности оно определяет те случаи, когда стандартная схема свободна.

ССП *свободна*, если все ее цепочки допустимы.

Допустимая цепочка операторов - это цепочка операторов, соответствующая допустимой цепочке схемы. В тотальной схеме все допустимые цепочки (и допустимые цепочки операторов) конечны. В пустой схеме - бесконечны.

1.3.2 Свободные интерпретации

Множество всех интерпретаций очень велико и поэтому вводится класс *свободных интерпретаций* (**СИ**), который образует ядро класса всех интерпретаций в том смысле, что справедливость высказываний о семантических свойствах **ССП** достаточно продемонстрировать для программ, получаемых только с помощью **СИ**.

Все **СИ** базиса B имеют одну и ту же область интерпретации, которая совпадает с множеством T всех термов базиса B. Все **СИ** одинаково интерпретируют переменные и функциональные символы, а именно:

a) для любой переменной x из базиса B и для любой **СИ** I_h этого базиса $I_h(x) = x$;

b) для любой константы a из базиса B $I_h(a) = a$;

c) для любого функционального символа $f^{(n)}$ из базиса B, где $n \geq 1$, $I_h\left(f^{(n)}\right) = F^{(n)} : T^{(n)} \to T$, где $F^{(n)}$ - словарная функция такая, что

$$F^{(n)}(\tau_1, \tau_2, \ldots, \tau_n) = f^{(n)}(\tau_1, \tau_2, \ldots, \tau_n),$$

т. е. функция $F^{(n)}$ по термам $\tau_1, \tau_2, \ldots, \tau_n$ из T строит новый терм, используя функциональный смысл символа $f^{(n)}$.

Интерпретация предикатных символов полностью *свободна,* т. е. разные **СИ** различаются лишь интерпретаций предикатных символов.

Таким образом, после введения **СИ** термы используются в двух разных качествах, как функциональные выражения в схемах и как значения переменных и выражений

1.3.3 Согласованные свободные интерпретации

Полагают, что интерпретация I и **СИ** I_h (того же базиса B) согласованы, если для любого логического выражения π справедливо $I_h(\pi) = I(\pi)$. Справедливы прямое и обратное утверждения, что для каждой интерпретации I базиса B существует согласованная с ней **СИ** этого базиса.

Введем понятие *подстановки термов.* Если $x_1, x_2, \ldots, x_n$ $(n \geq 0)$ – попарно различные переменные, $\tau_1, \tau_2, \ldots, \tau_n$ – термы из T, а π - функциональное или логическое выражение, то через $\pi[\tau_1/x_1, \tau_2/x_2, \ldots, \tau_n/x_n,]$ будем обозначать выражение, получающееся из выражения π одновременной заменой каждого из вхождений переменной x_i на терм τ_i $(i = \overline{1,n})$.

Например:

$a[y/x] = a$, $f(x,y)[y/x, x/y] = f(y,x)$, $g(x)[g(x)/x] = g(g(x))$,

$p(x)[a/x] = p(a)$.

Приведем без доказательства несколько важных утверждений:

Если интерпретация I и свободная интерпретация I_h согласованы, то программы (S, I) и (S, I_h) либо зацикливаются, либо обе останавливаются и $val(S, I) = val(S, I_h)$, т. е. каждой конкретной программе можно поставить во взаимно-однозначное соответствие свободно интерпретированную (стандартную) согласованную программу.

Если интерпретация и свободная интерпретация согласованы, они порождают одну и ту же цепочку операторов схемы.

<u>*Теорема Лакхэма – Парка – Паттерсона.*</u> Стандартные схемы S_1 и S_2 в базисе B функционально эквивалентны тогда и только тогда, когда они функционально эквивалентны на множестве всех свободных интерпретаций базиса B, т. е. когда для любой свободной интерпретации I_h программы (S_1, I_h) и (S_2, I_h) либо обе зацикливаются, либо обе останавливаются и

$$val(S_1, I_h) = \{I(S_1, I_h) = I(S_2, I_h)\} = val(S_2, I_h)$$

Стандартная схема S в базисе B пуста (тотальна, свободна) тогда и только тогда, когда она пуста (тотальна, свободна) на множестве всех свободных интерпретаций этого базиса, т. е. если для любой свободной интерпретации I_h программа (S, I_h) зацикливается (останавливается).

Стандартная схема S в базисе B свободна тогда и только тогда, когда она свободна на множестве всех свободных интерпретаций этого базиса, т. е. когда каждая цепочка схемы подтверждается хотя бы одной свободной интерпретацией.

1.3.4 Логико-термальная эквивалентность

Отношение эквивалентности E, заданное на парах стандартных схем, называют *корректным,* если для любой пары схем S_1 и S_2 из $S_1 \square E \square S_2$ следует, что $S_1 \square S_2$, т. е. S_1 и S_2 функционально эквивалентны.

Построение таких (корректных и разрешимых) отношений связанно с введением понятия истории цепочки схем. В истории с той или иной степенью детальности фиксируются промежуточные результаты выполнения операторов рассматриваемой цепочки. Эквивалентными объявляются схемы, у которых совпадают множества историй всех конечных цепочек.

Одним из отношений эквивалентности является *логико-термальная эквивалентность* (**ЛТЭ**).

Определим термальное значение переменной x для конечного пути ω схемы S как терм $t(\omega, x)$, который строится следующим образом.

1) Если путь ω содержит только один оператор A, то: $t(\omega, x) = \tau$, если A – оператор присваивания, $t(\omega, x) = x$, в остальных случаях.
2) Если $\omega = \omega' Ae$, где A – оператор, e – выходящая из него дуга, ω' – непустой путь, ведущий к A, а $x_1, x_2, \ldots, x_n$ – все переменные $t(Ae, x)$, то

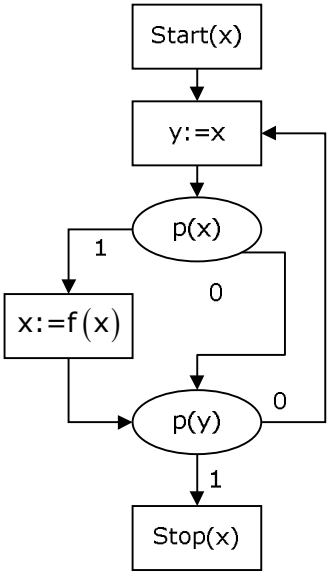
$$t(\omega, x) = t(Ae, x)\left[t(\omega', x_1)/x_1, t(\omega', x_2)/x_2, \ldots, t(\omega', x_n)/x_n\right].$$

Понятие термального значения распространим на произвольные термы τ:

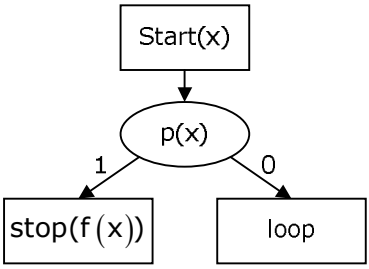
$$t(\omega, x) = \tau\left[t(\omega', x_1)/x_1, t(\omega', x_2)/x_2, \ldots, t(\omega', x_n)/x_n\right]$$

Например, пути

start(x); y:=x; p^1(x); x:=f(x); p^0(y); y:=x; p^1(x); x:=f(x) в схеме на рисунке 1.5 а соответствует термальное значение f(f(x)) переменной x.



а



б

Рис. 1.5

Для пути ω в стандартной схеме S определим ее *логико-термальную историю* (**ЛТИ**) $lt(S,\omega)$ как слово, которое строится следующим образом.

1) Если путь ω не содержит распознавателей и заключительной вершины, то $lt(S,\omega)$ – пустое слово.
2) Если $\omega = \omega' ve$, где v – распознаватель с тестом $p(\tau_1,\tau_2,\ldots,\tau_n)$, а e – выходящая из него Δ-дуга, $\Delta \in \{0,1\}$, то
$$lt(S,\omega) = lt(S,\omega')\, p^{\Delta}\left(t(\omega',\tau_1), t(\omega',\tau_2),\ldots, t(\omega',\tau_n)\right).$$
3) Если $\omega = \omega' v$, где v – заключительная вершина с оператором **stop**(τ_1, . . . , τ_n), то
$$lt(S,\omega) = lt(S,\omega')t(\omega',\tau_1), t(\omega',\tau_2),\ldots, t(\omega',\tau_n)$$

Например, логико-термальной историей пути, упомянутого в приведенном выше примере, будет p^1(x) p^0(x) p^1(f(x)).

Детерминантом ($\det(S)$) стандартной схемы S называют множество **ЛТИ** всех ее цепочек, завершающихся заключительным оператором.

Схемы S_1 и S_2 называют **ЛТЭ** (лт - эквивалентными) $S_1 \text{ } \square \text{ ЛТ } \square \text{ } S_2$, если их детерминанты совпадают.

Логико-терминальная эквивалентность корректна, т. е. из **ЛТЭ** следует функциональная эквивалентность ($S_1 \text{ } \square \text{ ЛТ } \square \text{ } S_2 \Rightarrow S_1 \text{ } \square \text{ } S_2$). Обратное утверждение как видно из рисунка 1.5 б не верно.

1.4 Моделирование стандартных схем программ

1.4.1 Одноленточные автоматы

Конечный одноленточный (детерминированный, односторонний) автомат обнаруживают ряд полезных качеств, используемых в теории схем программ для установления разрешимости свойств **ССП**.

Одноленточный конечный автомат (**ОКА**) над алфавитом V задается набором

$A = (V, Q, R, q_0, \#, I)$ и правилом функционирования, общим для всех таких автоматов. В наборе A

V - алфавит;

Q - конечное непустое *множество состояний* $(Q \cap \not= \varnothing)$;

R - множество выделенных *заключительных состояний* $(R \subseteq Q)$;

q_0 - выделенное *начальное* состояние;

I - *программа автомата*;

\# - «пустой» символ.

Программа автомата I представляет собой множество *команд* вида $qa \to q'$, в которой $q, q' \in Q$, $a \in V$ и для любой пары (q, a) существует единственная команда, начинающаяся этими символами.

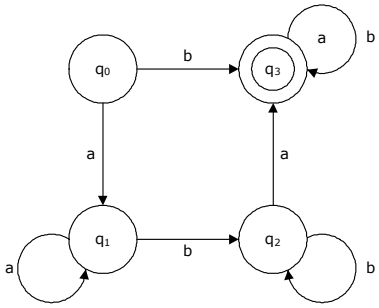
Неформально **ОКА** можно представить как абстрактную машину, похожую на машину Тьюринга, но имеющую следующие особенности:

1) выделены заключительные состояния;
2) машина считывает символы с ленты, ничего не записывая на нее;
3) на каждом шаге головка автомата, считав символ с ленты и перейдя согласно программе в новое состояние, обязательно передвигается вправо на одну клетку;
4) автомат останавливается в том и только в том случае, когда головка достигнет конца слова, т. е. символа #.

Автомат допускает слово α в алфавите V, если, начав работать с лентой, содержащей это слово, он останавливается в заключительном состоянии. Автомат A задает *характеристическую функцию* множества M_A *допускаемых им слов* в алфавите V, т. е. он распознает, принадлежит ли заданное слово множеству M_A если связать с остановкой в заключительном состоянии символ 1, а с остановкой в незаключительном состоянии – 0.

Наглядным способом задания **ОКА** служат графы автоматов. Автомат A представляется графом, множество вершин которого – множество состояний Q, и из вершины q в вершину q' ведет дуга, помеченная символом a, тогда и только тогда, когда программа автомата содержит команду $qa \to q'$. Работе автомата над заданным словом соответствует путь из начальной вершины q_0. Последовательность проходимых вершин этого пути – это последовательность принимаемых автоматом состояний, образ пути по дугам – читаемое слово. Любой путь в графе автомата, начинающийся в вершине q_0 и заканчивающийся в вершине $q \in R$, порождает слово, допустимое автоматом.

Пример **ОКА** $A = \left(\{a,b\},\{q_0,q_1,q_2,q_3\},\{q_2\},q_0,\#,I\right)$, допускающего слова $\left\{a^n b^m \,\middle|\, n=1,2,\ldots; m=1,2\ldots\right\}$, задается графом (рисунок 1.6).



Программа I содержит команды:

$q_0 a \to q_1$;
$q_0 b \to q_3$;
$q_1 a \to q_1$;
$q_1 b \to q_2$;
$q_2 a \to q_3$;
$q_2 a \to q_2$;
$q_3 a \to q_3$;
$q_3 b \to q_3$.

Рис. 1.6 Пример **ОКА**

Автомат называется *пустым*, если $M_A = \varnothing$. Автоматы A_1 и A_2 *эквивалентны*, если $M_{A_1} = M_{A_2}$.

Для ОКА доказано:

1) *Проблема пустоты* **ОКА** *разрешима*. Доказательство основано на проверке допустимости конечного множества всех слов, длина ко-

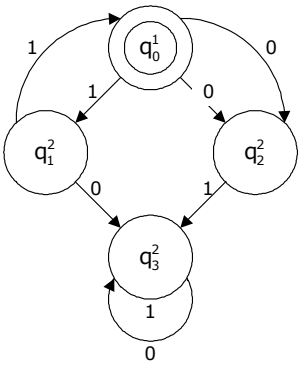
торых не превышает числа состояний **ОКА** - n. Если ни одно слово из этого множества не допускается, то **ОКА** «пуст».

2) *Проблема эквивалентности* **ОКА** *разрешима*. Доказательство основано на использовании отношения эквивалентности двух состояний q и q'': если состояния q и q' эквивалентны, то для всех $a \in A$ состояния $\delta(q,a)$ и $\delta'(q',a)$ также эквивалентны.

1.4.2 Многоленточные автоматы

Многоленточный (детерминированный, односторонний) конечный автомат (**МКА**) задается так же, как **ОКА**. Отличие состоит в том, что множество состояний Q разбивается на $n \geq 2$ непересекающихся подмножеств $Q_1, Q_2, \ldots, Q_n$. «Физическая» интерпретация n-ленточного автомата отличается тем, что он имеет n лент и n головок, по головке на ленту. Если автомат находится в состоянии $q \in Q_i$, то i-я головка читает i-ю ленту так же, как это делает **ОКА**. При переходе **МКА** в состояние $q' \in Q_j$ $(i \neq j)$ i-я головка останавливается, а j-я начинает читать свою ленту. **МКА** останавливается, когда головка на одной из лент прочитает символ #.

Пример. Рассмотрим функционирование **МКА** с $n = 2$ (рисунок 1.7) на примере сравнения пар слов в алфавите $\{0,1\}$ и допуске только совпадающих слов.



Здесь $Q = Q_1 \cup Q_2$, где $Q_1 = \{q_0^1\}$; $Q_2 = \{q_1^2, q_2^2, q_3^2\}$; $R = \{q_0^1\}$; $V = \{0,1\}$, начальное состояние - q_0^1. **МКА** обрабатывает наборы слов (U_1, U_2), где слово U_1 записано на первой ленте, а U_2 - на второй. Допустимое множество наборов M_A - это все возможные пары одинаковых слов, т. е. наборы, где $U_1 = U_2$. *Например*, набор может быть $(111001,111001)$ и т. п.

Рис. 1.7 Пример **МКА**

В том случае, когда слова совпадают, **МКА** остановится в заключительном состоянии q_0^1 (именно в этом состоянии поступит символ #) и

набор будет допущен. Если слова не совпадут хотя бы в одном символе, **МКА** перейдет в состояние q_3^2, из которого не выйдет до появления символа #, который и зафиксирует отсутствие совпадения слов, т. е. не допустит искаженный набор.

1.4.3 Двухголовочные автоматы

Двухголовочный конечный автомат (**ДКА**) имеет одну ленту и *две головки*, которые могут независимо перемещаться вдоль ленты в одном направлении. Множество состояний Q разбито на два непересекающихся множества. В состояниях Q_1 активна первая головка, а в состояниях Q_2 - вторая.

Двухголовочный автомат можно рассматривать как такой двухленточный автомат, который работает с идентичными словами на обеих лентах.

Пример. **ДКА** проверяет равенство двух последовательно записанных слов в алфавите $V = \{0,1\}$. Признаком окончания каждого из слов является вспомогательный символ $*$, не входящий в V. Автомат должен допускать только слова вида $a^* a^*$, где $a \in V^*$. $A = \left(V \cup \{*\}, Q_1 \cup Q_2, q_0^1, \#, I\right)$, где $Q_1 = \left\{q_0^1, q_1^1, q_2^1, q_7^1\right\}$ - множество состояний, в которых активна первая головка; $Q_2 = \left\{q_3^2, q_4^2, q_5^2, q_6^2\right\}$ - множество состояний, в которых активна вторая головка; $R = \left\{q_7^1\right\}$ - множество, содержащее единственное заключительное состояние. Граф автомата показан на рисунке 1.8, на котором вместо многих «параллельных» дуг с разными пометками нарисована одна дуга со всеми этими пометками.

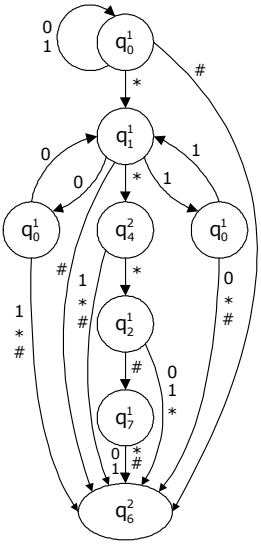


Рис. 1.8 Пример **ДКА**

Находясь в состоянии q_0^1, автомат передвигает первую головку к началу второго слова и, обнаружив его, переходит в состояние q_1^1. Если конец ленты # встречается раньше символа *, автомат переходит в

незаключительное состояние q_6^2. Если же автомат приходит к состоянию q_1^1, он считывает поочередно символы второго слова первой головкой (состояние q_1^1), а символы первого слова - второй головкой (состояния q_3^2 и q_5^2), сравнивая эти символы. Автомат возвращается каждый раз в состояние q_1^1, если символы одинаковы. Если же обнаружится несовпадение символов или первая головка встречает конец слова раньше символа *, автомат уходит в состояние q_6^2. Попав в это состояние, автомат не может выйти из него; перемещая вторую головку к концу слова на ленте, он достигает #, находясь в незаключительном состоянии, так что слово на ленте отвергается. Если первая головка достигнет конца второго слова, а вторая головка обнаружит, что первое слово тоже просмотрено до конца, то автомат перейдет в заключительное состояние q_7^1. В противном случае автомат перейдет в состояние q_6^2, отвергая слово.

Этот пример легко обобщить на случай произвольного алфавита V, увеличивая количество состояний множества Q.

Говорят, что **ДКА** *моделирует работу машины Тьюринга над некоторым начальным словом,* если автомат допускает единственное слово - конечный протокол работы машины над ним.

Лемма (*Розенберг*). Существует алгоритм, который для любой машины Тьюринга и для любого начального слова строит двухголовочный автомат, моделирующий ее работу над этим словом.

Теорема. Проблема пустоты **ДКА** не является частично разрешимой.

Теорема. Проблема эквивалентности **ДКА** не является частично разрешимой.

Из неразрешимости проблемы пустоты следует неразрешимость проблемы эквивалентности, так как пустоту можно рассматривать как частный случай эквивалентности.

1.4.4 Двоичный двухголовочный автомат

Стандартные схемы могут моделировать двухголовочные автоматы, что позволяет свести проблему пустоты этих автоматов к проблеме пустоты схем. Такое моделирование можно осуществить более простым способом, если использовать специальный класс двухголовочных автоматов, а именно класс двоичных автоматов, работающих со словами над алфавитом $\{0,1\}$.

Лемма. Существует алгоритм преобразования двухголовочных автоматов в двоичные двухголовочные автоматы (**ДДКА**), сохраняющий

пустоту автоматов (построенный двоичный автомат A_b пуст тогда и только тогда, когда пуст исходный автомат A).

Доказательство. Пусть **ДКА** A над алфавитом $V=\{a_1,a_2,\ldots,a_n\}$ имеет множество состояний $Q_A=\{q_1^k,q_2^k,\ldots,q_k^k\}$, где верхний индекс $k=(1,2)$ определяет номер активной головки. Преобразование этого автомата в двоичный начнем с кодировки символов и слов из V^* словами в алфавите $\{0,1\}$ по следующему правилу:

– код $(\#)=0$;

– код $(a_i)=11\ldots10 \quad (i=1,2,\ldots n)$;

– код $(\alpha a_i)=\text{код}(\alpha)\text{код}(a_i)$.

Так как символ # кодируется нулем, то любому непустому слову на ленте автомата A соответствует двоичное слово на ленте автомата A_b, оканчивающееся двумя нулями..

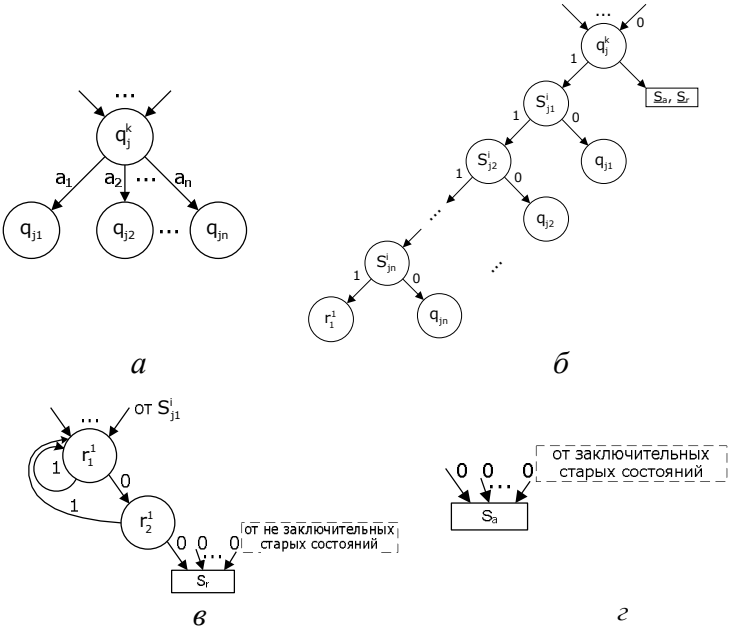


Рис. 1.9 Преобразование двухголовочнго автомата

Автомат A преобразуется в двоичный автомат A_b так, как показано на графах рисунка 1.9. Каждый фрагмент графа A (рисунок 1.9, а) заменяется фрагментом (рисунок 1.9, б) для A_b.

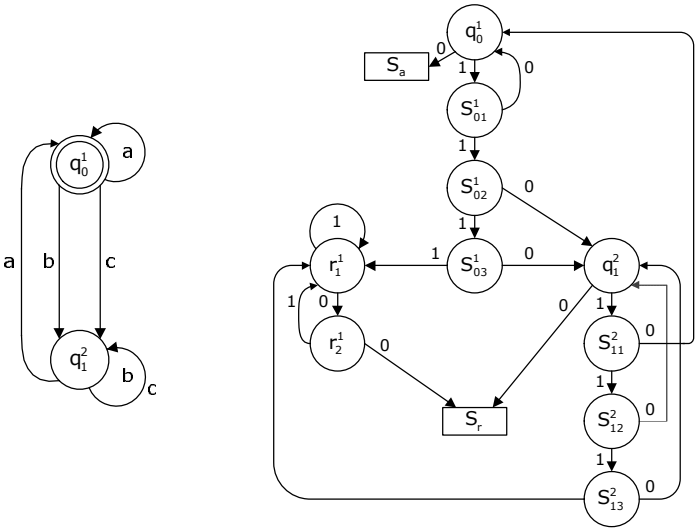
После замены добавляются фрагменты (рисунок 1.9 в, г).

Множество состояний автомата A_b включает:

- все старые состояния из Q_A;
- для каждого старого состояния q_j^k n новых состояний, n - число символов алфавита V;
- два новых состояния r_1^1 и r_2^1.

Заключительными состояниями автомата A являются заключительные состояния автомата A_b.

Вершины S_a (останов допускающий) и S_r (останов отвергающий) носят на графе вспомогательный характер в графе A_b. Они отмечают тот факт, что автомат прочитал два нуля подряд и остановился в заключительном состоянии (случай S_a) или в незаключительном состоянии (случай S_r).



а *б*

Рис. 1.10 Пример преобразования

Легко убедиться, что автомат A_b допускает двоичное слово p тогда и только тогда, когда оно является двоичным кодом слова из V^*,

допускаемого автоматом A. Таким образом, из пустоты автомата A следует пустота автомата A_b, и наоборот.

На рисунке 1.9 а приведен граф **ДКА** A допускающий только те слова в алфавите $V = \{a, b, c\}$, в которых символ a встречается не меньшее число раз, чем символы b и c, вместе взятые. Заключительное состояние $R = \{q_0^1\}$. На рисунке 1.9 б приведен граф **ДДКА**, построенный по автомату A (10 – код символа a, 110 – код b, 1110 – код c).

1.5 Рекурсивные схемы

1.5.1 Рекурсивное программирование

Формализации понятия вычислимой функции метод Тьюринга - Поста основан на уточнении понятия процесса вычислений, для чего используются абстрактные «машины», описанные в точных математических терминах. Другой подход (метод Черча - Клини) основан на понятии рекурсивной функции. Рекурсивная функция задается с помощью рекурсивных определений. Рекурсивное определение позволяет связать искомое значение функции для заданных аргументов с известными значениями той же функции при некоторых других аргументах. Эта связь устанавливается с помощью универсального механизма рекурсии, дающего механическую процедуру поиска значений функции. Двум подходам к определению вычислимых функций соответствуют два метода программирования этих функций - операторное и рекурсивное программирование. При операторном методе программа представляет собой явно выписанную последовательность, описаний действий гипотетической вычислительной машины (последовательность операторов, команд и т. п.).

Язык Фортран - типичный представитель операторных языков. С другой стороны, рекурсивная программа - это совокупность рекурсивных определений, задающих рекурсивную функцию, для которой аргументами служат начальные данные программы, а значением - результат выполнения программы. Известный язык рекурсивного программирования - язык Лисп - предназначен для обработки символьной информации. В других зыках комбинируют оба метода программирования. Так, Паскаль - операторный язык с возможностью рекурсивного программирования, предоставляемой механизмом рекурсивных процедур и функций.

Примером рекурсивно определяемой функции является факториальная функция $FACT : Nat \to Nat$:

FACT(x)=1,если x=0, FACT(x)=x×FACT(x-1), если x>0.

Эту функцию можно запрограммировать в некотором рекурсивном языке, базирующемся на механизме рекурсивных функций языка Паскаль:

```
FACT(a),
FACT(x)=if x=0 then 1 else x×FACT(x-1),
```

где a - некоторое целое неотрицательное число.

Вычисление функционального выражения сводится к его упрощению, т. е. выполнению всех возможных вычислений. Если в упрощенном выражения остается вхождение символа определяемой функции FACT, то осуществляется переход к новому шагу выполнения программы. На этом шаге вхождение FACT(m), где m - значение внутри скобок после упрощения, заменяется левым (m=0) или правым (m>0) функциональным выражением, в котором все вхождения x заменены на m. Упрощения продолжаются до тех пор, пока не будет получено выражение, не содержащее FACT.

Вычисление рекурсивной программы может завершиться за конечное число шагов с результатом, равным значению запрограммированной функции для заданных аргументов (начальных значений переменных), но может и продолжаться бесконечно. В последнем случае значение функции не определено.

1.5.2 Определение рекурсивной схемы

Рекурсивная схема (**РС**) так же, как **СПП** определяется в некотором базисе. *Полный базис* **РС**, как и базис **ССП**, включает четыре счетных множества символов: *переменные, функциональные символы, предикатные символы, специальные символы.*

Множества переменных и предикатных символов ничем не отличаются от ССП. Множество специальных символов — другое, а именно: {**if, then, else, (,),,**}. Отличие множества функциональных символов состоит в том, что оно разбито на два непересекающиеся подмножества*: множество базовых функциональных символов и множество определяемых функциональных символов* (обозначаются для отличия прописными буквами, например, $F^{(1)}$, $G^{(2)}$, и т. д.).

В базисе **РС** нет множества операторов, вместо него – множество логических выражений и множество термов.

Простые термы определяются так же, как термы–выражения в **СПП.** Среди простых термов выделим *базовые термы*, которые не содержат определяемых функциональных символов, а также *вызовы-*

термы вида $F^{(n)}(\tau_1,\tau_2,\ldots,\tau_n)$, где $\tau_1,\tau_2,\ldots,\tau_n$ - простые термы, $F^{(n)}$ - определяемый функциональный символ.

Логическое выражение — слово вида

$$p^{(n)}(\tau_1,\tau_2,\ldots,\tau_n),$$

где $p^{(n)}$ - предикатный символ, а $\tau_1,\tau_2,\ldots,\tau_n$ - базовые термы.

Терм — это *простой терм*, или *условный терм*, т. е. слово вида **if** π **then** τ_1 **else** τ_2,

где π - логическое выражение, τ_1,τ_2 - простые термы, называемые *левой* и соответственно *правой альтернативой.*

Примеры термов:

- $f(x,g(x,y)); h(h(a))$ - базовые термы;
- $f(F(x),g(x,F(y)))$; $H(H(a))$ - простые термы;
- $F(x)$; $H(H(a))$ - вызовы;
- **if** $p(x,y)$ **then** $h(h(a))$ **else** $F(x)$ - условный терм.

Расширим в базисе B множество специальных символов символом «=».

Рекурсивным уравнением, или *определением функции* F назовем слово вида

$$F^{(n)}(x_1,x_2,\ldots,x_n)=\tau(x_1,x_2,\ldots,x_n),$$

где $\tau(x_1,x_2,\ldots,x_n)$ - терм, содержащий переменные, называемые *формальными параметрами* функции F.

Рекурсивной схемой называется пара (τ,M), где τ - терм, называемый *главным термом схемы* (или ее *входом*). M - такое множество рекурсивных уравнений, что все определяемые функциональные символы в левых частях уравнений различны и всякий определяемый символ, встречающийся в правой части некоторого уравнения или в главном терме схемы, входит в левую часть некоторого уравнения. Другими словами, в **РС** имеется определение всякой вызываемой в ней функции, причем ровно одно.

Примеры РС:

- R_{S1} : `F(x); F(x)=`**`if`** `p(x)` **`then`** `a` **`else`** `g(x,F(h(x))).`
- R_{S2} : `A(b,c); A(x,y)=`**`if`** `p(x)` **`then`** `f(x)` **`else`**
  `B(x,y);`
  `B(x,y)=`**`if`** `p(y)` **`then`** `A(g(x),a)` **`else`** `C(x,y);`

C(x,y)=A(g(x), A(x,g(y))).

- R_{S3} : F(x); F(x)=**if** p(x) **then** x **else** f(F(g(x)),F(h(x))).

Пара (R_s, I), где R_S - **PC** в базисе B, а I - интерпретация этого базиса, называется *рекурсивной программой.* При этом заметим, что определяемые функциональные символы не интерпретируются.

1.6 Трансляция схем программ

1.6.1 О сравнении класс сов схем.

Любую вычислимую функцию можно запрограммировать и найти ее значения для заданных значений аргументов. При этом не требуется богатого набора программных примитивов и базовых операций: достаточно тех средств, которые моделируются стандартными схемами. Однако одни программные примитивы являются «более выразительными», чем другие. Запись алгоритмов с привлечением рекурсии короче, чем итерационное представление, но вычисления по такой программе могут потребовать больше времени, и т. д. При переходе к схемам программ возникает проблема выражения одних наборов примитивов через другие. Задачи такого типа образуют сравнительную схематологию, основу которой составляют теоремы о возможности или невозможности преобразования схем из одного класса в схемы другого.

Будем сравнивать классы схем, у которых базисы *согласованны* в том смысле, что множества *переменных*, базовых *функциональных* символов и предикатных символов *одинаковы* в данных базисах. Это дает возможность превращать в программы схемы из разных классов с помощью одной и той же интерпретации базисов. Например, полные базисы стандартных и рекурсивных схем согласованны, т. е. определение функциональной эквивалентности может быть обобщено на схемы из разных классов.

Схема S_1 из класса W и схема S_2 из класса W' *функционально эквивалентны,* если для любой интерпретации I согласованных базисов классов W и W' программы (S_1, I), (S_2, I) или обе зацикливаются, или обе останавливаются с одним и тем же результатом.

Говорят, что *класс схем* W *мощнее класса схем* W', или *класс* W' *транслируем в класс* W, если для любой схемы из W' существует эквивалентная ей схема в классе W. *Классы* W *и* W' *равномощны,* если W' мощнее W и W мощнее W'.

Доказано, что класс **ССП** транслируем в класс **РС** и класс **РС** не транслируем в класс **ССП**.

Существуют некоторые классы **РС**, транслируемые в **ССП**. К ним относится класс *линейных унарных* **РС**, имеющих базис с единственной переменной x и одноместными функциональными и предикатными символами. Например, линейная унарная **РС**

R_{S4} : `F(x); F(x)=`**`if`** `p(x)` **`then`** `x` **`else`** `f(x,F(g(x)))`

транслируема в **ССП**.

1.6.2 Схемы с процедурами

Схемы с процедурами строятся в объединенном базисе классов стандартных и рекурсивных схем. Она состоит из двух частей - *главной схемы* и множества *схем процедур.* Главная схема - это стандартная схема, в которой имеются операторы присваивания специального вида $x = F^{(n)}\left(y_1, y_2, \ldots, y_n\right)$, называемые *операторами вызова процедур.* Схема процедуры состоит из *заголовка и тела процедуры,* разделенных символом равенства. Заголовок имеет тот же вид, что и левая часть рекурсивных уравнений. Тело процедуры — это стандартная схема того же вида, что и главная схема. Заключительный оператор тела процедуры всегда одноместен (**`stop`**`(x)`). Ниже приведен пример схемы с процедурами.

Главная схема	Множество схем процедур

`(`**`start`**`(x),`
`1:z:=x,`
`2:u:=a,`
`3:x:=F(x,z,u),`
`4:u:=b,`
`5:z:=F(z,x,u)`
`6:`**`stop`**`(z))`
	`F(y,v,w)=`**`start`**`,`
	`1:`**`if`** `p(y)` **`then`** `2` **`else`** `4,`
	`2:y:=h(y),`
	`3:v:=G(v,w)` **`goto`** `1,`
	`4:`**`if`** `q(w)` **`then`** `5` **`else`** `6,`
	`5:y:=v,`
	`6:`**`stop`**`(y))`
	`G(t,r)=(`**`start`**`,`
	`1:`**`if`** `q(r)` **`then`** `2` **`else`** `3,`
	`2:t:=f(t),`
	`3:`**`stop`**`(t);`

Доказано, что класс **РС** транслируем в класс схем с процедурами и наоборот.

1.7 Обогащенные и структурированные схемы

1.7.1 Классы обогащенных схем

Выделяют следующие классы обогащенных схем: класс с*четчиковых* схем, класс *магазинных* схем, класс схем с *массивами*.

Классы счетчиковых и магазинных схем образован добавлением в базис **ССП** счетного множества счетчиков и магазинов с их интерпретированными операторами.

Счетчик - интерпретированная переменная, у которой областью значений является множество Nat; начальное значение счетчика равно 0.

Интерпретированные операторы имеют следующий вид:

$c := c + 1$ - оператор прибавления единицы;

$c := c - 1$ - оператор вычитания единицы;

$c = 0$ - условный оператор проверки равенства счетчика нулю.

При значении счетчика равном 0 оператор вычитания единицы не изменяет его, оно остается равным 0.

К интерпретированным операторам может быть добавлен оператор пересылки значения счетчика $c_1 = c_2$, который может быть получен при помощи стандартных операторов.

Магазин - неинтерпретированная переменная сложной структуры. В процессе выполнения интерпретированной схемы *состояние магазина* - это конечный набор элементов $(d_1, d_2, \ldots, d_n)$ из области интерпретации, где d_n - *верхушка магазина*.

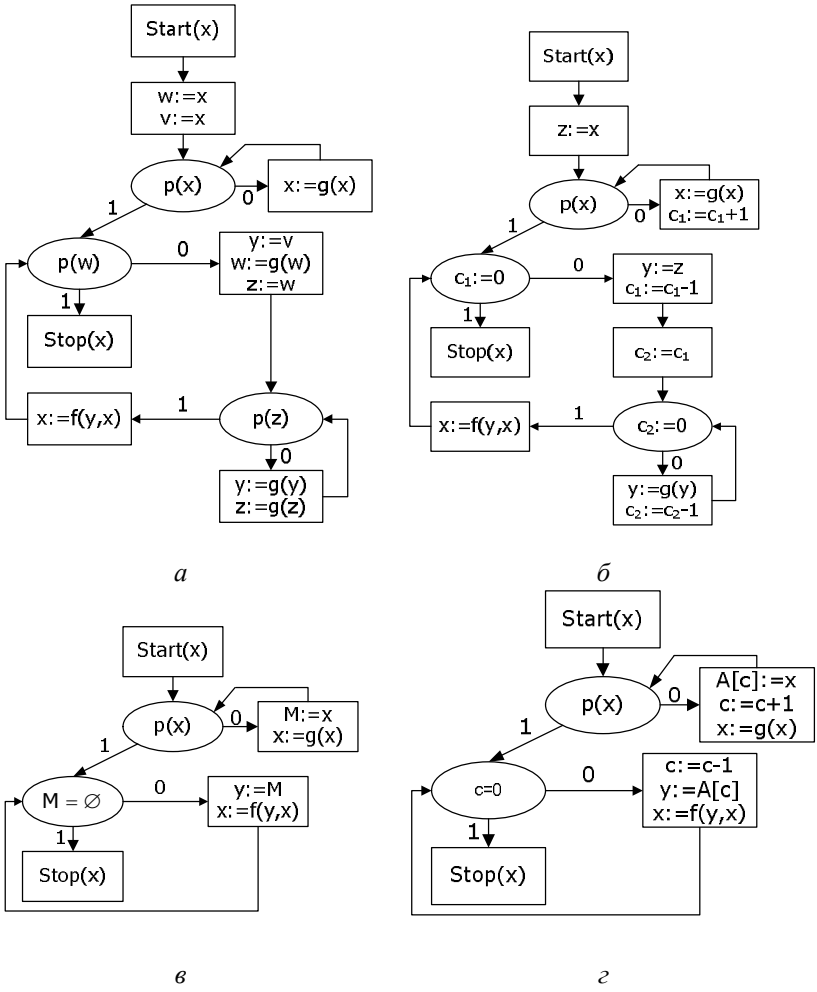
Интерпретированные операторы имеют следующий вид:

$M := x$ - запись в магазин;

$x := M$ - выборка из магазина;

$M = \varnothing$ - условный оператор проверки пустоты магазина,

где M – магазин, x - обычная переменная. Первый оператор меняет состояние $(d_1, d_2, \ldots, d_n)$ магазина M на состояние $(d_1, d_2, \ldots, d_{n+1})$, где d_{n+1} - значение переменной x. После выполнения этого оператора элемент d_{n+1} становится новой верхушкой магазина. Второй оператор присваивает переменной x значение, равное верхушке магазина, состояние которого меняется с $(d_1, d_2, \ldots, d_n)$ на $(d_1, d_2, \ldots, d_{n-1})$, при этом d_{n-1} становится новой верхушкой магазина. Если магазин M пуст, то применение второго оператора оставляет его пустым, а переменная x не меняет своего значения. Третий оператор - предикат проверки магазина на пустоту; если магазин пуст, то значение предиката $M = \varnothing$ равно 1, в противном случае - 0.



а *б*

в *г*

Рис. 1.11 Примеры обогащенных схем

Класс схем с массивами - это расширение класса счетчиковых схем за счет добавления счетного множества массивов и операторов над ними.

Массив - неинтерпретированная переменная сложной структуры. При выполнении интерпретированной схемы *состояние массива* - бесконечная последовательность $(d_1, d_2, \ldots, d_i, \ldots)$ элементов из области интерпретациии.

Интерпретированные операторы имеют следующий вид:

$A[c] := x$ - запись в магазин;

$x := A[c]$ - выборка из магазина,

где A - массив, $[c]$ - целое число, равное текущему значению счетчика c.

На рисунке 1.11. приведены четыре схемы: стандартная (а), счетчиковая (б), магазинная (в) и схема с массивами (г). Все они эквивалентны друг другу и рекурсивной схеме:

```
F(x), F(x)=if p(x) then x else f(x,F(g(x))).
```

1.7.2 Трансляция обогащенных схем

Диаграмма на рисунке 1.12 дает полную информацию о возможности трансляции одного класса схем в другой, классы имеют следующие обозначения:

Y - стандартные схемы; $Y(M)$ - магазинные схемы;

$Y(R)$ - рекурсивные схемы; $Y(A)$ - схемы с массивами;

$Y(c)$ - счетчиковые схемы; $Y(P)$ - схемы с процедурами.

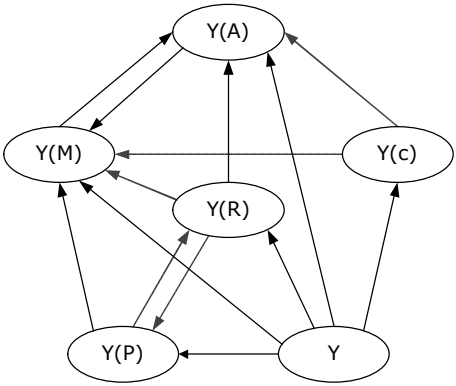


Рис. 1.12 Диаграмма взаимной трансляции схем

Диаграмма показывает, что классы $Y(M)$ и $Y(A)$ являются универсальными в том смысле, что схемы всех других классов транслируемы в них. В то же время, в класс Y не транслируются схемы ни одного другого класса.

1.7.3 Структурированные схемы

Возрастающая сложность программ заставляет пересмотреть принципы организации программ, их структуру. Дейкстра первым выска-

зался против неупорядоченного использования переходов на метки, которое может привести к переусложнению структуры программы, затрудняющему ее понимание. Реализуя концепцию так называемого структурированного программирования, он предложил отказаться от использования переходов и ограничиться более дисциплинирующими средствами управления вычислениями, такими, как условные операторы и операторы цикла.

Класс с*труктурированных схем* $Y(S)$ определяется в том базисе B, который был введен для **ССП** Y.

Различие между Y и $Y(S)$ проявляется на уровне структур схем. Вместо специальных символов Y вводятся специальные символы: **if, then, else, while, do, end**. Вместо инструкций с метками вводятся три типа *схемных оператора* в базисе B: простой оператор, условный оператор и оператор цикла.

Простой оператор — это начальный (заключительный) оператор и оператор присваивания.

Условный оператор — это оператор вида

if π **then** σ_1 **else** σ_0 **end,**

где π - логическое выражение, σ_0, σ_1 - последовательности (может быть, пустые) схемных операторов, среди которых нет ни начального, ни заключительного.

Операторы цикла имеют вид

while π **do** σ **end** или **until** π **do** σ **end,**

где π, σ имеют тот же смысл, что и выше.

Ниже приведен пример эквивалентных схем Y и $Y(S)$.

Стандартная схема программ	Структурированная схема программ

start(x),
1:y :=f(x),
2:**if** p(y) **then** 7 **else** 3,
3:y:=f(y),
4:**if** p(y) **then** 5 **else** 7,
5:**if** p(x) **then** 6 **else** 7,
6:x:=h(x) **goto** 5,
7:**stop**(x,y).
	start(x),
	y:=f(x),
	if p(y) **then else**
	  y:=f(y),
	  if p(x) **then**
	  while p(x)
	  do x:=h(x) **end**
	  else
	  end
	end,
	stop(x,y).

Доказано, что класс Y мощнее класса $Y(S)$, т. е. схемы $Y(S)$ транслируемы в Y, но не наоборот.

Усилить класс $Y(S)$ можно за счет усложнения логических выражений в условных операторах и операторах цикла $Y(SL)$, введением символов логических операций **NOT, OR, AND** и атомарных формул, которыми являются логические выражения в старом смысле, например:

```
NOT p(x) AND q(y,x);
p(g(x,t)) OR q(h(x),x).
```

В этом случае справедлива *теорема* (*Ашкрофт - Манн*)

Класс стандартных схем Y транслируем в класс схем с логическими операциями $Y(SL)$

Контрольные вопросы

2 Семантическая теория программ

2.1 Описание смысла программ

Семантическая теория программ заниматься описанием *семантики программ*, или смысла выражений, операторов и программных единиц.

В языке имеется множество различных конструкций, точное определение которых необходимо как программисту, использующему язык, так и разработчику реализации этого языка. Программисту эти сведения нужны для того, чтобы писать правильные программы и заранее знать результат выполнения любых операторов программы. Разработчику компилятора корректные определения конструкций необходимы для создания правильной реализации языка.

Как правило, определение семантики дается в виде обычного текста. Сначала при помощи какой-либо формальной грамматики дается определение синтаксиса конструкции, а затем для пояснения семантики приводятся несколько примеров и небольшой пояснительный текст. Смысл этого текста часто неоднозначен. Программист может получить ошибочное представление о том, что именно будет делать написанная им программа при выполнении, а разработчик может реализовать какую-либо языковую конструкцию иначе, чем разработчики других реализаций того же языка. Как и для синтаксиса, нужен какой-то метод, позволяющий дать удобочитаемое, точное и лаконичное определение семантики языка.

Задача определения семантики языка программирования рассматривается теоретиками давно, но до сих пор не найдено удовлетворительного универсального решения..

2.2 Операционная семантика

Операционная семантика, сводится к описанию смысла программы посредством выполнения ее операторов на реальной или виртуальной машине. Смысл оператора определяется изменениями, произошедшими в состоянии машины после выполнения данного оператора.

Пусть состояние компьютера - это значения всех его регистров и ячеек памяти, в том числе коды условий и регистры состояний. Если просто записать состояние компьютера, выполнить команду, смысл которой нужно определить, а затем изучить новое состояние машины, то семантика этой команды станет понятной: она представляется изменением в состоянии компьютера, вызванным выполнением команды.

Описание операционной семантики операторов языков программирования высокого уровня требует создания реального или виртуального компьютера. Аппаратное обеспечение компьютера является чистым интерпретатором его машинного языка. Чистый интерпретатор любого языка программирования может быть создан с помощью программных средств, которые становятся виртуальным компьютером для данного языка. Семантику языка высокого уровня можно описать, используя чистый интерпретатор данного языка. При таком подходе существуют две проблемы. *Во-первых*, сложность и индивидуальные особенности аппаратного обеспечения компьютера и операционной системы, используемых для запуска чистого интерпретатора, затрудняют понимание происходящих действий. *Во-вторых*, выполненное таким образом семантическое определение будет доступно только для людей с абсолютно идентичной конфигурацией компьютера.

Этой проблемы можно избежать, заменив реальный компьютер виртуальным компьютером низкого уровня. Регистры, память, информация о состоянии и процесс выполнения операторов можно смоделировать, соответствующими программами. Набор команд можно создать так, чтобы семантику каждой отдельной команды было легко понять и описать. Таким образом, машина была бы идеализирована и значительно упрощена, что облегчило бы понимание изменений ее состояния.

Использование операционного метода для полного описания семантики языка программирования L требует создания двух компонентов. *Во-первых*, для преобразования языка L в операторы выбранного языка низкого уровня нужен транслятор. *Во-вторых*, для этого

языка низкого уровня необходима виртуальная машина, состояние которой изменяется с помощью команд, полученных при трансляции операторов высокого уровня. Именно изменения состояния этой виртуальной машины определяет смысл данного оператора.

Семантику конструкции **for** языка *C* можно описать в терминах следующих простых команд.

Оператор языка *C*	Операционная семантика

`for (expr1;expr2;expr3)`
`{`
 	
`}`
	`exrp1`
	`loop:if expr2=0 goto out`
	
	`expr3;`
	`goto loop`
	`out:`

Формальное описание операционной семантики можно представить в виде системы равенств:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= E_1 \\ f_2(x_1, x_2, \ldots, x_n) &= E_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ f_m(x_1, x_2, \ldots, x_n) &= E_m \end{aligned},$$

где в левых частях равенств явно указаны определяемые функции с формальными параметрами, включающими обозначения всех входных данных $x_1, x_2, \ldots, x_n$, а правые части представляют собой выражения, содержащие, вообще говоря, вхождения этих функций с аргументами, задаваемыми некоторыми выражениями, зависящими от входных данных $x_1, x_2, \ldots, x_n$.

Операционная семантика интерпретирует эти равенства как систему подстановок. Под подстановкой $\langle s; E; \tau \rangle$ терма τ в выражение E вместо символа s будем понимать переписывание выражения E с заменой каждого вхождения в него символа s на выражение τ. Каждое равенство $f_i(x_1, x_2, \ldots, x_n) = E_i$ задает в параметрической форме множество правил подстановок:

$$\langle x_1, x_2, \ldots, x_n; f_i(\tau_1, \tau_2, \ldots, \tau_n) \to E_i; \tau_1, \tau_2, \cdots, \tau_n \rangle$$

где $\tau_1, \tau_2, \cdots, \tau_n$ - конкретные аргументы данной функции. Это правило допускает замену вхождения левой его части в какое-либо выражение на его правую часть.

Интерпретация системы равенств для получения значений определяемых функций в рамках операционной семантики производится сле-

дующим образом. Пусть задан набор входных данных $d_1, d_2, \ldots, d_n$. На первом шаге осуществляется подстановка этих данных в левые и правые части равенств с выполнением там, где это возможно, предопределенных операций и с выписыванием получаемых в результате этого равенств. На каждом следующем шаге просматриваются правые части полученных равенств. Если правая часть является каким-либо значением, то оно и является значением функции, указанной в левой части этого равенства. В противном случае правая часть является выражением, содержащим вхождения каких-либо определяемых функций с теми или иными наборами аргументов. Если для такого вхождения соответствующая функция с данным набором аргументов имеется в левой части какого-либо из полученных равенств, то либо вместо этого вхождения подставляется вычисленное значение правой части этого равенства, либо не производится никаких изменений. В том же случае, если эта функция с данным набором аргументов не является левой частью никакого из полученных равенств, то формируется (и дописывается к имеющимся) новое равенство. Оно получается из исходного равенства для данной функций подстановкой в него вместо параметров указанных аргументов этой функции. Такие шаги осуществляются до тех пор, пока все определяемые функции не будут иметь вычисленные значения.

Пример. Рассмотрим систему равенств, определяющую функцию $FACT(n) = n!$

$$FACT(0) = 1$$
$$FACT(n) = n \times FACT(n-1).$$

Для вычисления значения $FACT(3)$ осуществляются 6 шагов.

шаг 1

$$FACT(0) = 1$$
$$FACT(3) = 3 \times FACT(2)$$

шаг 2

$$FACT(0) = 1$$
$$FACT(3) = 3 \times FACT(2)$$
$$FACT(2) = 2 \times FACT(1)$$

шаг 3

$$FACT(0) = 1$$
$$FACT(3) = 3 \times FACT(2)$$
$$FACT(2) = 2 \times FACT(1)$$
$$FACT(1) = 1 \times FACT(0)$$

шаг 4

$$FACT(0) = 1$$
$$FACT(3) = 3 \times FACT(2)$$
$$FACT(2) = 2 \times FACT(1)$$
$$FACT(1) = 1$$

шаг 5

$$FACT(0) = 1$$
$$FACT(3) = 3 \times FACT(2)$$
$$FACT(2) = 2$$
$$FACT(1) = 1$$

шаг 6

$$FACT(0) = 1$$
$$FACT(3) = 6$$
$$FACT(2) = 2$$
$$FACT(1) = 1$$

Операционная семантика является эффективной до тех пор, пока описание языка остается простым и неформальным. Операционная семантика зависит от алгоритмов, а не от математики. Операторы одного языка программирования описываются в терминах операторов

другого языка программирования, имеющего более низкий уровень. Этот подход может привести к порочному кругу, когда концепции неявно выражаются через самих себя.

2.3 Аксиоматическая семантика

Аксиоматическая семантика была создана в процессе разработки метода доказательства правильности программ. Семантику каждой синтаксической конструкции языка можно определить как некий набор аксиом или правил вывода, который можно использовать для вывода результатов выполнения этой конструкции. Чтобы понять смысл всей программы, эти аксиомы и правила вывода следует использовать так же, как при доказательстве обычных математических теорем. В предположении, что значения входных переменных удовлетворяют некоторым ограничениям, аксиомы и правила вывода могут быть использованы для получения ограничений на значения других переменных после выполнения каждого оператора программы. Когда программа выполнена, получаем доказательство того, что вычисленные результаты удовлетворяют необходимым ограничениям на их значения относительно входных значений. То есть, доказано, что выходные данные представляют значения соответствующей функции, вычисленной по значениям входных данных.

Такие доказательства показывают, что программа выполняет вычисления, описанные ее спецификацией. В доказательстве каждый оператор программы сопровождается предшествующим и последующим логическими выражениями, устанавливающими ограничения на переменные в программе. Эти выражения используются для определения смысла оператора вместо полного описания состояния абстрактной машины.

Аксиоматическая семантика основана на математической логике. Будем называть предикат, помещенный в программу *утверждением*. Утверждение, непосредственно предшествующее оператору программы, описывает ограничения, наложенные на переменные в данном месте программы. Утверждение, следующее непосредственно за оператором программы, описывает новые ограничения на те же (а возможно, и другие) переменные после выполнения оператора.

Введем обозначение (*триада* Хоара)

$$\{Q\}\, S\, \{R\}$$

где Q, R - предикаты, S - программа (оператор или последовательность операторов). Обозначение определяет следующий смысл: «Если выполнение S началось в состоянии, удовлетворяющем Q, то имеет-

ся гарантия, что оно завершится через конечное время в состоянии, удовлетворяющем R».

Предикат Q называется *предусловием* или входным утверждением S, предикат R - *постусловием* или выходным утверждением, R определяет то, что нужно установить (R определяет спецификацию задачи). В предусловии Q нужно отражать тот факт, что входные переменные получили начальные значения.

Пример. Рассмотрим оператор присваивания для целочисленных переменных и постусловие:

```
sum:=2*x+1 {sum>1}
```

Одним из возможных предусловий данного оператора может быть `{x>10}`.

Слабейшими предусловиями называются наименьшие предусловия, обеспечивающие выполнение соответствующего постусловия. Например, для приведенного выше оператора и его постусловия предусловия `{x>10}`, `{x>50}` и `{x>1000}` являются правильными. Слабейшим из всех предусловий в данном случае будет `{x>0}`.

2.3.1 Преобразователь предикатов.

Э. Дейкстра рассматривает слабейшие предусловия, т. е. предусловия, необходимые и достаточные для гарантии желаемого результата.

«Условие, характеризующее множество всех начальных состояний, при которых запуск обязательно приведет к событию правильного завершения, причем система останется в конечном состоянии, удовлетворяющем заданному постусловию, называется слабейшим предусловием, соответствующим этому постусловию».

Если S - некоторый оператор (последовательность операторов), R - желаемое постусловие, то соответствующее слабейшее предусловие будем обозначать $wp(S,R)$. Аббревиатура wp определяется начальными буквами английских слов *weakest* (слабейший) и *precondition* (предусловие).

Определение семантики оператора дается в виде правила, описывающего, как для любого заданного постусловия R можно вывести соответствующее слабейшее предусловие $wp(S,R)$.

Для фиксированного оператора S такое правило, которое по заданному предикату R вырабатывает предикат $wp(S,R)$, называется *«преобразователем предикатов»:* $\{wp(S,R)S\{R\}\}$.

Это значит, что описание семантики оператора S представимо с помощью преобразователя предикатов. Применительно к конкретным S и R часто бывает неважным точный вид $wp(S,R)$, бывает достаточно более сильного условия Q, т. е. условия, для которого можно доказать, что утверждение $Q \Rightarrow wp(S,R)$ справедливо для всех состояний. Значит, множество состояний, для которых Q - истина, является подмножеством того множества состояний, для которых $wp(S,R)$ - истина. Возможность работать с предусловиями Q, не являющимися слабейшими, полезна, поскольку выводить $wp(S,R)$ явно не всегда практично.

Сформулируем свойства $wp(S,R)$.

1) $wp(S,\mathbf{F}) = \mathbf{F}$ для любого S. (Закон исключенного чуда).
2) Закон монотонности. Для любого S и предикатов P и R таких, что $P \Rightarrow R$ для всех состояний, справедливо для всех состояний $wp(S,P) \Rightarrow wp(S,R)$.
3) Дистрибутивность конъюнкции. Для любых S,R,P справедливо $wp(S,P)$ **AND** $wp(S,R) \Rightarrow wp(S,P$ **AND** $R)$.
4) Дистрибутивность дизъюнкции. Для любых S,R,P справедливо $wp(S,P)$**OR** $wp(S,R) \Rightarrow wp(S,P$ **OR** $R)$.

Если для каждого оператора языка по заданным постусловиям можно вычислить слабейшее предусловие, то для программ на данном языке может быть построено корректное доказательство. Доказательство начинается с использования результатов, которые надо получить при выполнении программы, в качестве постусловия последнего оператора программы, и выполняется с помощью отслеживания программы от конца к началу с последовательным вычислением слабейших предусловий для каждого оператора. При достижении начала программы первое ее предусловие отражает условия, при которых программа вычислит требуемые результаты.

Для некоторых операторов программы вычисление слабейшего предусловия на основе оператора и его постусловия является достаточно простым и может быть задано с помощью аксиомы. Однако, как правило, слабейшее предусловие вычисляется только с помощью *правила логического вывода,* т. е. метода выведения истинности одного утверждения на основе значений остальных утверждений.

2.3.2 Аксиоматическое определение операторов языка программирования

Определим слабейшее предусловие для основных операторов: оператора присваивания, составного оператора, оператора выбора и оператора цикла.

Оператор присваивания имеет вид: $x := E$, где x - простая переменная, E – выражение.

Слабейшее предусловие оператора присваивания $Q = wp(x := E, R)$, получается из R заменой каждого вхождения x на E $\left(Q = R_E^x\right)$.

Предполагается, что значение E определено, и вычисление выражения E не может изменить значения ни одной переменной. Следовательно, можно использовать обычные свойства выражений такие, как ассоциативность, коммутативность и логические законы.

Составной оператор имеет вид: **begin** S_1**;** S_2**;...;** S_n **end.**

Слабейшее предусловие для последовательности двух операторов:

$$wp(S_1; S_2, R) = wp(S_1, wp(S_2, R)).$$

Аналогично слабейшее предусловие определяется для последовательности из n операторов.

Оператор выбора определяется:

$$\textbf{if } B_1 \to S_1 \ \Pi\ B_2 \to S_2 \ldots \Pi\ B_n \to S_n \textbf{ fi}.$$

Здесь $n \geq 0$, $B_1, B_2, \ldots, B_n$ - логические выражения, называемые охранами, $S_1, S_2, \ldots, S_n$ - операторы, пара $B_i \to S_i$ называется охраняемой командой, Π - разделитель, **if** и **fi** играют роль операторных скобок.

Выполняется оператор следующим образом.

Проверяются все B_i. Если одна из охран не определена, то происходит аварийное завершение. Далее, по крайней мере, одна из охран должна иметь значение истина, иначе выполнение завершается аварийно. Выбирается одна из охраняемых команд $B_i \to S_i$ у которой значение B_i истина, и выполняется S_i.

Слабейшее предусловие для этого оператора:

$$wp(\textbf{if}, R) = BB \textbf{ AND } \left(B_1 \Rightarrow wp(S_1, R)\right) \textbf{AND} \ldots \textbf{AND } \left(B_n \Rightarrow wp(S_n, R)\right),$$

где $BB = B_1$ **OR** B_2 **OR…OR** B_n.

Естественно определить $wp(\textbf{if}, R)$ с помощью кванторов:

$$wp(\textbf{if}, R) = \left(\exists i : 1 \leq i \leq n : B_i\right) \textbf{ AND } \left(\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R)\right)$$

Пример. Определить $z = |x|$.

С использованием оператора выбора:

$$\textbf{if } x \geq 0 \to z := x \,\Pi\, x < 0 \to z := -x \textbf{ fi}.$$

К особенностям оператора выбора следует отнести, *во-первых*, тот факт, что он включает условный оператор (**`if...then`**..), альтернативный оператор (**`if`**... **`then... else...`**) и оператор выбора (**`case`**).

Во-вторых, оператор выбора не допускает умолчаний, что помогает при разработке сложных программ, так как каждая альтернатива представлена подробно, и возможность что-либо упустить уменьшается.

В-третьих, благодаря отсутствию умолчаний, запись оператора выбора представлена в симметричном виде.

Оператор цикла имеет вид: **`do`** `B → S` **`do`**.

Обозначим это соотношение через DO и представим его в следующем виде:

$$\text{DO}: \textbf{do } B_1 \to S_1 \,\Pi\, B_2 \to S_2 \,\Pi \ldots \Pi\, B_n \to S_n \textbf{ od},$$

где $n \geq 0$, $B_i \to S_i$ - охраняемые команды.

Пока возможно выбирается охрана B_i со значением истина, и выполняется соответствующий оператор S_i. Как только все охраны будут иметь значение ложь, выполнение DO завершится.

Выбор охраны со значением истина и выполнение соответствующего оператора называется выполнением шага цикла. Если истинными являются несколько охран, то выбирается любая из них. Следовательно, оператор DO эквивалентен оператору

$$\textbf{do } BB \to \textbf{if } B_1 \to S_1 \,\Pi\, B_2 \to S_2 \ldots \Pi\, B_n \to S_n \textbf{fi od} \text{ или } \textbf{do } BB \to \text{IF} \textbf{od},$$

где BB - дизъюнкция охран, IF - оператор выбора.

Пример. Алгоритм Евклида.

Вариант 1.

задать (N, M);

if $N > 0$ **AND** $M > 0 \to n, m := N,\ M$;

do $n \neq m \to$ **if** $n > m \to n := n - m \,\Pi\, m > n \to m := m - n$ **fi od**;

выдать (n)

fi

Вариант 2.

задать (N, M);

if $N > 0$ **AND** $M > 0 \to n, m := N,\ M$;

do $n > m \to n := n - m \,\Pi\, m > n \to m := m - n$ **od**;

выдать (n)

fi

Пусть предикат $H_0(R)$ определяет множество состояний, в которых выполнение DO завершается за 0 шагов (в этом случае все охраны с самого начала ложны, после завершения R имеет значение истина):

$$H_0(R) = \mathbf{NOT}\, BB\, \mathbf{AND}\, R\,.$$

Чтобы оператор цикла DO завершил работу, не производя выборки охраняемой команды, необходимо, чтобы $\mathbf{NOT}\, BB = \mathbf{T}$. При этом истинность R до выполнения DO является необходимым условием для истинности R после выполнения DO.

Определим предикат $H_k(R)$ как множество состояний, в которых выполнение DO заканчивается за k шагов при значении R истина ($H_k(R)$ будет определяться через $H_{k-1}(R)$):

$$H_k(R) = H_0(R)\,\mathbf{OR}\, wp\left(\mathrm{IF}, H_{k-1}(R)\right), k > 0 \to wp(\text{DO}, R) \quad (\exists k : k \geq 0 : H_k(R)).$$

Это значит, что должно существовать такое значение k, что потребуется не более чем k шагов, для обеспечения завершения работы в конечном состоянии, удовлетворяющем постусловию R.

Теорема инвариантности для оператора цикла. Пусть оператор выбора IF и предикат P таковы, что для всех состояний справедливо

$$(P\ \mathbf{AND}\ BB) \Rightarrow wp(\mathrm{IF}, R).$$

Тогда для оператора цикла справедливо:

$$\left(P\,\mathbf{AND}\, wp(\mathrm{DO}, \mathbf{T})\right) \Rightarrow wp(\mathrm{DO}, P\,\mathbf{AND}\,\mathbf{NOT}\, BB).$$

Предикат P, истинный перед выполнением и после выполнения каждого шага цикла, называется *инвариантным отношением* или просто *инвариантом цикла.*

Это условие означает, что если предикат P первоначально истинен и одна из охраняемых команд выбирается для выполнения, то после ее выполнения P сохранит значение истинности. После завершения оператора, когда ни одна из охран не является истиной, будем иметь:

$$P\,\mathbf{AND}\,\mathbf{NOT}\, BB\,.$$

Работа завершится правильно, если условие $wp(\mathrm{DO}, \mathbf{T})$ справедливо и до выполнения DO. Так как любое состояние удовлетворяет **T**, то $wp(\mathrm{DO}, \mathbf{T})$ является слабейшим предусловием для начального состоя-

ния такого, что запуск оператора цикла DO приведет к правильно завершаемой работе.

При определении семантики полного языка программирования с использованием аксиоматического метода для каждого вида операторов языка должны быть сформулированы аксиома или правило логического вывода. Но определение аксиом и правил логического вывода для некоторых операторов языков программирования - очень сложная задача. Трудно построить «множество основных аксиом, достаточно ограниченное для того, чтобы избежать противоречий, но достаточно богатое для того, чтобы служить отправной точкой для доказательства утверждений о программах» (Э. Дейкстра).

Решением такой проблемы является разработка языка, использующего аксиоматического метода, т. е. содержащей только те операторы, для которых могут быть написаны аксиомы или правила логического вывода. К сожалению, подобный язык оказался бы слишком маленьким и простым что отражает нынешнее состояние аксиоматической семантики как науки.

Аксиоматическая семантика является мощным инструментом для исследований в области доказательств правильности программ, она также создает великолепную основу для анализа программ, как во время их создания, так и позднее. Однако ее полезность при описании содержания языков программирования весьма ограничена как для пользователей языка, так и для разработчиков компиляторов.

2.4 Денотационная семантика

Денотационная семантика — самый строгий широко известный метод описания значения программ. Она опирается на теорию рекурсивных функций.

Основной концепцией денотационной семантики является определение для каждой сущности языка некоего математического объекта и некоей функции, отображающей экземпляры этой сущности в экземпляры этого математического объекта. Поскольку объекты определены строго, то они представляют собой точный смысл соответствующих сущностей. Сама идея основана на факте существования строгих методов оперирования математическими объектами, а не конструкциями языков программирования. Сложность использования этого метода заключается в создании объектов и функций отображения. Название метода «денотационная семантика» происходит от английского слова denote (обозначать), поскольку математический объект обозначает смысл соответствующей синтаксической сущности.

Для введения в денотационный метод мы используем очень простую языковую конструкцию — двоичные числа. Синтаксис этих чисел можно описать следующими грамматическими правилами:

$$\begin{aligned}\langle\text{двоичное\_число}\rangle \to\ & 0 \\ & \mid 1 \\ & \mid \langle\text{двоичное\_число}\rangle\ 0 \\ & \mid \langle\text{двоичное\_число}\rangle\ 1\end{aligned}$$

Для описания двоичных чисел с использованием денотационной семантики и грамматических правил, указанных выше, их фактическое значение связывается с каждым правилом, имеющим в своей правой части один терминальный символ. Объектами в данном случае являются десятичные числа.

В примере значащие объекты должны связываться с первыми двумя правилами. Остальные два правила являются правилами вычислений, поскольку они объединяют терминальный символ, с которым может ассоциироваться объект, с нетерминальным, который может представлять собой некоторую конструкцию.

Пусть область определения семантических значений объектов представляет собой множество неотрицательных десятичных целых чисел Nat. Это именно те объекты, которые мы хотим связать с двоичными числами. Семантическая функция M_b отображает синтаксические объекты в объекты множества N согласно указанным выше правилам. Сама функция M_b определяется следующим образом:

$$\begin{aligned}& M_b('0') = 0, M_b('1') = 1 \\ & M_b(\langle\text{двоичное\_число}\rangle'0') = 2 \times M_b(\langle\text{двоичное\_число}\rangle) \\ & M_b(\langle\text{двоичное\_число}\rangle'1') = 2 \times M_b(\langle\text{двоичное\_число}\rangle) + 1\end{aligned}$$

Пример. Описание значения десятичных синтаксических литеральных констант.

$$\begin{aligned}\langle\text{десятичное\_число}\rangle \to\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ & \mid \langle\text{десятичное\_число}\rangle 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

Денотационные отображения для этих синтаксических правил имеют следующий вид:

$$M_d('0') = 0, M_d('1') \quad 1, \ldots, M_d('9') \quad 9=$$
$$M_d(\langle \text{десятичное\_число} \rangle '0') = 10 \times M_d(\langle \text{десятичное\_число} \rangle)$$
$$M_d(\langle \text{десятичное\_число} \rangle '1') = 10 \times M_b(\langle \text{десятичное\_число} \rangle) + 1$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$M_d(\langle \text{десятичное\_число} \rangle '1') = 10 \times M_b(\langle \text{десятичное\_число} \rangle) + 9$$

Денотационную семантику программы можно определить в терминах изменений состояний идеального компьютера. Состояния определяются в терминах значений всех переменных, объявленных в программе. В денотационной семантике они определяются строгими математическими функциями. Пусть состояние s программы определяется следующим набором упорядоченных пар: $\{\langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \ldots, \langle i_n, v_n \rangle\}$.

Каждый параметр i является именем переменной, а соответствующие параметры v являются текущими значениями данных переменных. Любой из параметров v может иметь специальное значение **undef**, указывающее, что связанная с ним величина в данный момент не определена.

Пусть VARMAP - функция двух параметров, имени переменной и состояния программы. Значение функции $\texttt{VARMAP}(i_k, s)$ равно v_k (значение, соответствующее параметру i_k в состоянии s).

Большинство семантических функций отображения для программ и программных конструкций отображают состояния в состояния. Эти изменения состояний используются для определения смысла программ и программных конструкций.

Выражения являются основой большинства языков программирования. Пусть имеем дело только с очень простыми выражениями. Единственными операторами являются операторы + и ×; выражения могут содержать не более одного оператора; единственными операндами являются скалярные переменные и целочисленные литеральные константы; круглые скобки не используются; значение выражения является целым числом. Ниже следует описание этих выражений в форме БНФ:

$$\langle \text{выражение} \rangle \to \langle \text{деятичное\_число} \rangle \mid \langle \text{переменная} \rangle \mid \langle \text{двоичное\_выражение} \rangle$$
$$\langle \text{двоичное\_выражение} \rangle \to \langle \text{выражение\_слева} \rangle \langle \text{оператор} \rangle \langle \text{выражение\_справа} \rangle$$
$$\langle \text{оператор} \rangle \to + \mid \times$$

Единственной рассматриваемой ошибкой в выражениях является неопределенное значение переменной. Разумеется, могут появляться и другие ошибки, но большинство из них зависят от машины. Пусть Z - набор целых чисел, а *error* - ошибочное значение. Тогда множество $Z \cup \{error\}$ является множеством значений, для которых выражение может быть вычислено.

Функция отображения для данного выражения E и состояния s приведена ниже. Символ ≡ обозначает равенство по определению функции.

$M_E(\langle\text{выражение}\rangle, s) \equiv$
case $\langle\text{выражение}\rangle$ **of**
$\langle\text{десятичное\_число}\rangle \Rightarrow M_в(\langle\text{десятичное\_число}\rangle, s)$
$\langle\text{переменная}\rangle \Rightarrow$ **if** VARMAP$(\langle\text{переменная}\rangle, s) =$ **undef**
then *error*
else VARMAP$(\langle\text{переменная}\rangle, s)$
$\langle\text{двоичное\_выражение}\rangle \Rightarrow$
if $(M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_слева}\rangle, s) =$ **undef OR**
$M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_справа}\rangle, s) =$ **undef** $)$
then *error*
else if $(M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{оператор}\rangle, s) =$ '+' **then**
$M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_слева}\rangle, s) +$
$M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_справа}\rangle, s)$
else $M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_слева}\rangle, s) \times$
$M_E(\langle\text{двоичное\_выражение}\rangle.\langle\text{выражение\_справа}\rangle, s)$

Оператор присваивания - это вычисление выражения плюс присваивание его значения переменной, находящейся в левой части. Его можно описать следующей функцией:

$M_A(x = E) \equiv$ **if** $M_E(E, s) = error$
then *error*
esle $\{\langle i'_1, v'_1\rangle, \langle i'_2, v'_2\rangle, \ldots, \langle i'_n, v'_n\rangle\}$ **where**
for $j = 1, 2, \ldots, n, v'_j$ VARMAP(i'_j, s) *if* $i_j \neq x$
$M_E(E, s)$ **if** $i_j = x$

Сравнения, выполняющиеся в строках $(i_j \neq x$ и $i_j = x)$ относятся к именам, а не значениям.

После определения полной системы для заданного языка ее можно использовать для определения смысла полных программ этого языка.

Денотационная семантика может использоваться для разработки языка. Операторы, описать которые с помощью денотационной семантики трудно, могут оказаться сложными и для понимания пользователями языка, и тогда разработчику следует подумать об альтернативной конструкции.

С одной стороны, денотационные описания очень сложны, с другой - они дают великолепный метод краткого описания языка.

2.5 Декларативная семантика

Декларативная семантика является существенной характеристикой языков логического программирования, в которых программы состоят из объявлений (деклараций), а не из операторов присваивания и управляющих операторов. Эти объявления в действительности являются операторами, или высказываниями, в символьной логике.

Основная концепция декларативной семантики заключается в том, что существует простой способ определения смысла каждого оператора, и он не зависит от того, как именно этот оператор используется для решения задачи. Декларативная семантика значительно проще, чем рассмотренные выше семантики. Она не требует для проверки отдельного оператора рассмотрения его контекста, локальных объявлений или последовательности выполнения программы.

Формальное определение семантики становится общепринятой частью определения нового языка. Тем не менее, изучение формальных определений семантики не оказало такого сильного влияния на практическое определение языков, как изучение формальных грамматик - на определение синтаксиса. Ни один из методов определения семантики не оказался по настоящему полезным ни для пользователя, ни для разработчиков компиляторов языков программирования.

2.6 Языки формальной спецификации

Языки и методы формальной спецификации, как средство проектирования и анализа программного обеспечения появились более сорока лет назад. За это время было немало попыток разработать как универсальные, так и специализированные языки формальных спецификаций (**ЯФС**), которые могли бы стать практическим инструментом разработки программ, таким же, как, например, языки программирования. Хотя сейчас даже термин «языки формальных спецификаций» известен далеко не всем программистам в этой отрасли программирования получены значительные результаты. Они выражаются, *во-первых*, в

том, что есть несколько **ЯФС**, которые уже нельзя назвать экспериментальными, более того, некоторые **ЯФС** подкреплены соответствующими стандартами. *Во-вторых*, более четким стало представление о способах использования **ЯФС**, о месте формальных методов в жизненном цикле разработки программного обеспечения и в процессах его разработки и использования.

Второй из перечисленных факторов важен, поскольку способ использования языка серьезно влияет на эволюцию языка. Если рассматривать историю языков спецификации общего назначения (универсальных, не специализированных), то видно, что они развивались в соответствии со следующим представлением о «правильном порядке» разработки программного обеспечения:

- описать эскизную модель (функциональности, поведения);
- доказать, что модель корректна (не противоречива);
- детализировать (уточнить) модель;
- доказать, что детализация проведена корректно;
- повторять два предыдущих шага до тех пор, пока не будет получена готовая программа.

Установка на данную схему процесса разработки приводила к акцентированному вниманию на средства обобщенного описания функциональности, средства уточнения, средства аналитического доказательства корректности в стиле традиционных математических доказательств. Как следствие, появились языки, которые не содержали в себе конструкций, затрудняющих аналитические доказательства, при этом они лишались и тех конструкций, без которых трудно описать реализацию сколько-нибудь сложной программной системы.

У специализированных языков другая история. Некоторые из них, например ***SDL*** (*Specification and Design Language*), родились из практики проектирования систем релейного управления, где проект традиционно был больше похож на чертеж, чем на текстовое (языковое) описание. Здесь эволюция заключалась во взаимном сближении графической и текстовой нотации на основе взаимных компромиссов и ограничений.

ЯФС традиционно рассматривались как средство проектирования. Новый взгляд на **ЯФС** появился, когда стала актуальной задача анализа уже существующего программного обеспечения. Существенное продвижение на этом фронте было связано с направлением *Объектно-Ориентированного Анализа.* Его идеи во многом созвучны с *Объектно-Ориентированным Проектированием*. Оба эти направления предлагают близкие изобразительные средства для описания архитектуры и поведения систем. Наиболее известным средством такого рода являет-

ся графический язык ***UML*** (*Unified Modelling Language*). Вместе с тем ***UML*** и подобные ему языки спецификации являсь неплохими средствами проектирования, непригодны для доказательства правильности, на что делался акцент в классических языках спецификации.

Новые требования к языкам спецификации появились с идеей использования их как источников для генерации тестов. Оказалось, что разные виды приложений требуют различных подходов к спецификации и имеют непохожие друг на друга возможности для генерации тестов. В частности, одни виды спецификаций в большей степени пригодны для генерации последовательностей тестовых воздействий, тогда как другие предоставляют удобные возможности для генерации тестовых оракулов – программ, оценивающих результат, полученный в ответ на тестовое воздействие.

Имеется несколько способов классификации подходов к спецификации. Различают моделе-ориентированные и свойство-ориентированные спецификации или спецификации, основанные на описании состояний и действий. Единой классификации не существует, мы рассмотрим следующие четыре класса подходов к спецификации: исполняемые, алгебраические, сценарные и ограничения.

Исполнимые спецификации, исполнимые модели. Этот подход предполагает разработку прототипов (моделей) систем для демонстрации возможности достижения поставленной цели и проведения экспериментов при частичной реализации функциональности. Примерами таких методологий и языков являются ***SDL***, ***RSL*** (*RAISE Specification Language*).

Алгебраические спецификации предполагают описание свойств композиций операций. Композиции могут быть последовательными, параллельными, с временными ограничениями и без. Преимуществом этого подхода является то, что в идеале можно полностью абстрагироваться от структур данных, которые используются в качестве входных и выходных значений и, возможно, используются для сохранения внутреннего состояния моделей. Основной недостаток – это нетрадиционность приемов спецификации, что затрудняет их внедрение в промышленных разработках. В качестве примера языка алгебраических спецификаций можно назвать ***ASN1*** (*Abstract Syntax Notation One*), стандарт которого входит в группу стандартов, описывающих ***SDL***, ***RSL***.

Сценарные спецификации описывают не непосредственно целевую систему, а способы ее использования или взаимодействия с ней. Такие косвенные описания, с одной стороны, позволяют судить о некоторых свойствах системы и, с другой стороны, такие спецификации могут

служить хорошим руководством по использованию системы, что не всегда можно сказать о других видах спецификаций. Наибольшее распространение получили работы ***OMG*** группы и продукты компании *Rational Corporation*.

Ограничения состоят из пред- и постусловий функций, процедур и других операций и инвариантов данных. Имеются расширения этого подхода для объектно-ориентированных спецификаций. В этом случае к спецификациям операций добавляются спецификации методов классов, а к инвариантам – инварианты объектов и классов. Языком, поддерживающим спецификацию ограничений, является ***RSL***.

2.7 Верификация программ

2.7.1 Методы доказательства правильности программ

Универсальные вычислительные машины могут быть запрограммированы для решения самых разнородных задач - в этом заключается одна из основных их особенностей, имеющая огромную практическую ценность. Один и тот же компьютер, в зависимости от того, какая программа находится у него в памяти, способен осуществлять арифметические вычисления, доказывать теоремы и редактировать тексты, управлять ходом эксперимента и создавать проект автомобиля будущего, играть в шахматы и обучать иностранному языку. Однако успешное решение всех этих и многих других задач возможно лишь при том условии, что компьютерные программы не содержат ошибок, которые способны привести к неверным результатам.

Можно сказать, что требование отсутствия ошибок в программном обеспечении совершенно естественно и не нуждается в обосновании. Но как убедиться в том, что ошибки, в самом деле, отсутствуют?

К неформальным методам доказательства правильности программ относят *отладку* и *тестировани*е, которые являются необходимой составляющей на всех этапах процесса программирования, хотя и не решают полностью проблемы правильности. Существенные ошибки легко найти, если использовать соответствующие приемы отладки (контрольные распечатки, трассировки).

Тестирование – процесс выполнения программы с намерением найти ошибку, а не подтвердить правильность программы. Суть его сводится к следующему. Подлежащую проверке программу неоднократно запускают с теми входными данными, относительно которых результат известен заранее. Затем сравнивают полученный машиной результат с ожидаемым. Если во всех случаях тестирования налицо совпадение этих результатов, появляется некоторая уверенность в том, что и

последующие вычисления не приведут к ошибочному итогу, т. е. что исходная программа работает правильно.

С интуитивной точки зрения программа будет правильной, если в результате ее выполнения будет достигнут результат, с целью получения которого и была написана программа. Сам по себе факт безаварийного завершения программы еще ни о чем не говорит: вполне возможно, что программа в действительности делает совсем не то, что было задумано. Ошибки такого рода могут возникать по различным причинам.

В дальнейшем будем предполагать, что обсуждаемые программы не содержат синтаксических ошибок, поэтому при обосновании их правильности внимание будет обращаться только на содержательную сторону дела, связанную с вопросом о том, достигается ли при помощи данной программы данная конкретная цель. Целью можно считать поиск решения поставленной задачи, а программу рассматривать как способ ее решения. Программа будет правильной, если она решит сформулированную задачу.

Метод установления правильности программ при помощи строгих средств известен как *верификация программ*.

В отличие от тестирования программ, где анализируются свойства отдельных процессов выполнения программы, верификация имеет дело со свойствами программ.

В основе метода верификации лежит предположение о том, что существует программная документация, соответствие которой требуется доказать. Документация должна содержать:

- спецификацию ввода-вывода (описание данных, не зависящих от процесса обработки);
- свойства отношений между элементами векторов состояний в выбранных точках программы;
- спецификации и свойства структурных подкомпонентов программы;
- спецификацию структур данных, зависящих от процесса обработки.

К такому методу доказательства правильности программ относится метод индуктивных утверждений, независимо сформулированный К. Флойдом и П. Науром.

Суть этого метода состоит в следующем:

1) формулируются входное и выходное утверждения: входное утверждение описывает все необходимые входные условия для программы (или программного фрагмента), выходное утверждение описывает ожидаемый результат;
2) предполагая истинным входное утверждение, строится промежуточное утверждение, которое выводится на основании семантики

операторов, расположенных между входом и выходом (входным и выходным утверждениями); такое утверждение называется выведенным утверждением;

3) формулируется теорема (условия верификации):
   из выведенного утверждения следует выходное утверждение;
4) доказывается теорема; доказательство свидетельствует о правильности программы (программного фрагмента).

Доказательство проводится при помощи хорошо разработанных математических методов, использующих исчисление предикатов первого порядка.

Условия верификации можно построить и в обратном направлении, т. е., считая истинным выходное утверждение, получить входное утверждение и доказывать теорему:

Из входного утверждения следует выведенное утверждение.

Такой метод построения условий верификации моделирует выполнение программы в обратном направлении. Другими словами, условия верификации должны отвечать на такой вопрос: если некоторое утверждение истинно после выполнения оператора программы, то, какое утверждение должно быть истинным перед оператором?

Построение индуктивных утверждений помогает формализовать интуитивные представления о логике программы. Оно и является самым сложным в процессе доказательства правильности программы. Это объясняется, *во-первых*, тем, что необходимо описать все содержательные условия, и, *во-вторых*, тем, что необходимо аксиоматическое описание семантики языка программирования.

Важным шагом в процессе доказательства является доказательство завершения выполнения программы, для чего бывает достаточно неформальных рассуждений.

Таким образом, алгоритм доказательства правильности программы методом индуктивных утверждений представляется в следующем виде:

1) Построить структуру программы.
2) Выписать входное и выходное утверждения.
3) Сформулировать для всех циклов индуктивные утверждения.
4) Составить список выделенных путей.
5) Построить условия верификации.
6) Доказать условие верификации.
7) Доказать, что выполнение программы закончится.

Этот метод сравним с обычным процессом чтения текста программы (метод сквозного контроля). Различие заключается в степени формализации.

Преимущество верификации состоит в том, что процесс доказательства настолько формализуем, что он может выполняться на вычислительной машине. В этом направлении в восьмидесятые годы проводились исследования, даже создавались автоматизированные диалоговые системы, но они не нашли практического применения.

Для автоматизированной диалоговой системы программист должен задать индуктивные утверждения на языке исчисления предикатов. Синтаксис и семантика языка программирования должны храниться в системе в виде аксиом на языке исчисления предикатов. Система должна определять пути в программе и строить условия верификации.

Основной компонент доказывающей системы - это *построитель* условий верификации, содержащий операции манипулирования предикатами, алгоритмы интерпретации операторов программы. Вторым компонентом системы является *подсистема доказательства* теорем.

Отметим трудности, связанные с методом индуктивных утверждений. Трудно построить «множество основных акс*иом*, достаточно ограниченное для того, чтобы избежать противоречий, но достаточно богатое для того, чтобы служить отправной точкой для доказательства утверждений о программах» (Э. Дейкстра). Вторая трудность - семантическая, заключающаяся в формировании самих утверждений, подлежащих доказательству. Если задача, для которой пишется программа, не имеет строгого математического описания, то для нее сложнее сформулировать условия верификации.

Перечисленные методы имеют одно общее свойство: они рассматривают программу как уже существующий объект и затем доказывают ее правильность.

Метод, который сформулировали К. Хоар и Э. Дейкстра основан на формальном выводе программ из математической постановки задачи.

2.7.2 Использование утверждений в программах

Утверждения используются для доказательства правильности программ. Тогда утверждения необходимо формулировать в некоторой формальной логической системе. Обычно используется *исчисление предикатов первого порядка*.

Исчисление - это метод или процесс рассуждений посредством вычислений над символами. В исчислении предикатов утверждения являются логическими переменными или выражениями, имеющими значение **T** - истина или **F** - ложь. При написании программы ставиться задача доказать истинность утверждения $\{Q\}\,S\,\{R\}$. Для этого нужно уметь записывать его в исчислении предикатов и формально доказывать его истинность.

Предикат, помещенный в программу, называется *утверждением*. Утверждается, что он истинен в соответствующий момент выполнения программы. В предусловии Q нужно отражать тот факт, что входные переменные получили начальные значения. Для обозначения начальных значений будем использовать большие буквы.

Пример. Пусть надо определить приближенное значение квадратного корня: $s = sqrt(n)$, где $n, s \in Nat$. Определим постусловие в виде:

$$R : s \times s \le n \le (s+1) \times (s+1).$$

Пример. Даны целочисленные значения $n > 0$ и массив $a[1,\ldots n]$. Отсортировать массив, т. е. установить

$$R : (\forall i : 1 \le i \le n : a[i] \le a[i+1]).$$

Пример. Определить x как максимальное значение массива $a[1,\ldots n]$. Определим постусловие:

$$R : \{x = \max(y \mid y \subseteq a)\}.$$

Для построения программы следует определить математическое понятие max. Тогда

$$R : \{(\exists i : 1 \le i \le n : x \quad a[i]) \ \mathbf{AND} \ (\forall i : 1 \le i \le n : a[n] = x)\}.$$

Пример. Пусть имеем программу S обмена значениями двух целых переменных a и b. Сформулируем входное и выходное утверждения программы и представим программу S в виде предиката:

$$\{a = A \ \mathbf{AND} \ b = B\} \, S \, \{a \quad B \ \mathbf{AND} \ b \quad A\}.$$

где A, B - конкретные значения переменных a, b.

Программа вместе с утверждениями между каждой парой соседних операторов называется *наброском доказательства.* Последовательно, для каждого оператора программы формулируя предикат, можно доказать, что программа удовлетворяет своим спецификациям. Представим набросок доказательства для программы S:

$$\{a = A \ \mathbf{AND} \ b = B\};$$
$$r = a; \{r \quad a \ \mathbf{AND} \ a \quad a \ \mathbf{AND} \ b \quad B\};$$
$$a = b; \{r \quad a \ \mathbf{AND} \ a \quad B \ \mathbf{AND} \ b \quad B\};$$
$$b = r; \{a \quad B \ \mathbf{AND} \ b \quad A\}.$$

Не обязательно набросок доказательства должен быть настолько полным. Для документирования программы нужно вставить достаточно утверждений, чтобы программа стала понимаемой.

Программа, содержащая утверждения для ее документирования, называется *аннотированной программой*. Чтобы использовать утверждения для доказательства правильности программы, необходимы соответствующие правила верификации.

2.7.3 Правила верификации К. Хоара

Сформулируем правила (аксиомы) К.Хоара, которые определяют предусловия как достаточные предусловия, гарантирующие, что исполнение соответствующего оператора при успешном завершении приведет к желаемым постусловиям.

*A*1. *Аксиома присваивания*: $\{R_0\}\, x := E\{R\}$.

Неформальное объяснение аксиомы: так как x после выполнения будет содержать значение E, то R будет истинно после выполнения, если результат подстановки E вместо x в R истинен перед выполнением. Таким образом, $R_0 = R(x)$ при $x = E$. Для R_0 вводится обозначение: $R_0 = RxE$ (у Вирта) или $Rx \to E$ (у Дейкстры), что означает, что x заменяется на E.

Аксиома присваивания будет иметь вид: $\{RxE\}\, x := E\{R\}$.

Сформулируем два очевидных правила монотонности.

*A*2. Если известно: $\{Q\}\, S\{P\}$ и $\{P\} \Rightarrow \{R\}$, то $\{Q\}\, S\{R\}$.

*A*3. Если известно: $\{Q\}\, S\{P\}$ и $\{R\} \Rightarrow \{Q\}$, то $\{R\}\, S\{P\}$.

Пусть S - это последовательность из двух операторов $S1; S2$ (составной оператор).

*A*4. Если известно: $\{Q\}\, S1\{P1\}$ и $\{P1\}\, S2\{R\}$, то $\{Q\}\, S\{R\}$.

Это правило можно сформулировать для последовательности, состоящей из n операторов.

Сформулируем правило для условного оператора (краткая форма).

*A*5. Если известно:

$\{Q$ **AND** $B\}\, S1\{R\}$ и $\{Q$ **NOT** $B\} \Rightarrow \{R\}$, то $\{Q\}$ **if** B **then** $S1\{R\}$.

Правило *A*5 соответствует интерпретации условного оператора в языке программирования.

Сформулируем правило для альтернативного оператора (полная форма условного оператора).

*A*6. Если известно: $\{Q$ **AND** $B\}\, S1\{R\}$ и $\{Q$ **NOT** $B\}\, S2\{R\}$, то

$$\{Q\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\{R\}.$$

Сформулируем правила для операторов цикла.

Предусловия и постусловия цикла **until** удовлетворяют правилу:

*A*7. Если известно: $\{Q \textbf{ AND NOT } B\}\, S1\, \{Q\}$, то

$$\{Q\} \textbf{ repeat } S1 \textbf{ until } B \{Q \textbf{ AND NOT } B\}.$$

Правило отражает инвариантность цикла. В данном случае единственная операция - это выполнение шага цикла при условии истинности Q вначале.

Предусловия и постусловия цикла **while** удовлетворяют правилу:

*A*8. Если известно: $\{Q \textbf{ AND NOT } B\}\, S1\, \{Q\}$, то

$$\{Q\} \textbf{ while } B \textbf{ do } S1 \{Q \textbf{ AND NOT } B\}.$$

Правила *A*1 - *A*8 можно использовать для проверки согласованности передачи данных от оператора к оператору, для анализа структурных свойств текстов программ, для установления условий окончания цикла и для анализа результатов выполнения программы.

Пример. Пусть надо определить частное q и остаток r от деления x на y.

Входные данные x, y и выходные данные $q, r \in Nat$, причем $y > 0$.

Задать $(x, y); / \times x, y$ получают конкретные значения $X, Y \times /$.

```
r:=x; q:=0;
while y≤r do
 begin
  r:=r-y; q:=q+1
 end;
выдать(q,r);
```

Сформулируем постусловие

$R: (r < y) \textbf{ AND } (x \quad y \times q + r)$.

Нужно доказать, что

$\{y > 0 \textbf{ AND } x/y\}\, S\, \{(r < y) \textbf{ AND } (x \quad y \times q + r)\}$.

Доказательство.

1) Очевидно, что $Q \Rightarrow x = x + y \times 0$.
2) Применим аксиому *A*1 к оператору $r := x$, тогда получим
   $\{x = x + y \times 0\}\, r := x\, \{x \quad r + y \times 0\}$.
3) Аналогично, применяя *A*1 к оператору $q := 0$, получим:
   $\{x = x + y \times 0\}\, q := 0\, \{x \quad r + y \times q\}$.
4) Применяя правило *A*3 к результатам пунктов 1) и 2), получим
   $\{Q\}\, r := x\, \{x \quad r + y * 0\}$.

5) Применяя правило $A4$ к результатам пунктов 4) и 3), получим $\{Q\}\, r := x; q := 0\, \{x \quad x + y \times q\}$.

6) Выполним равносильное преобразование $x = r + y \times q$ **AND** $y \le r \Rightarrow x = (r - y) + y \times (q + 1)$.

7) Применяя правило $A1$ к оператору $r := r - y$, получим $\{x = (r - y) + y \times (q + 1)\}\, r := r - y\, \{x \quad r + y \times (q + 1)\}$.

8) Для оператора $q := q + 1$ аналогично получим $\{x = r + y \times (q + 1)\}\, q := q + 1\, \{x \quad r + y \times q\}$.

9) Применяя правило $A4$ к результатам пунктов 7 и 8, получим $\{x = (r - y) + y \times (q + 1)\}\, r := r - y; q := q + 1\, \{x \quad r + y \times q\}$.

10) Применяя правило $A2$ к результатам пунктов 6) и 9), получим $\{x = r + y \times q$ **AND** $y \le r\}\, r := r - y; q := q + 1\, \{x \quad r + y \times q\}$.

11) Применяя правило $A8$ к результату пункта 10), получим
$\{x = r + y \times q\}$ **while** $y \le r$ **do begin** $r := r - y; q := q + 1$ **end** $\{$**NOT** $(y \le r$ **AND** $x \quad r + y \times q)\}$.

Утверждение $x = r + y \times q$ является инвариантом цикла, так как значение его остается истинным до цикла и после выполнения каждого шага цикла.

12) Применяя правило $A4$ к результатам пунктов 5) и 11), получаем то, что требовалось доказать,
$\{Q\}\, S\, \{$**NOT** $(y \le r)$ **AND** $(x \quad r + y \times q)\}$.

Осталось доказать, что выполнение программы заканчивается.

Доказывать будем от противного, т.е. предположим, что программа не заканчивается. Тогда должна существовать бесконечная последовательность значений r и q, удовлетворяющая условиям

1) $y \le r$;
2) $r, q \in Nat$.

Но значение r на каждом шаге цикла уменьшается на положительную величину: $r := r - y\,(y > 0)$. Значит, последовательность значений r и q является конечной, т.е. найдется такое значение r, для которого не будет выполняться условие $y \le r$ и циклический процесс завершится.

Контрольные вопросы

3 Теоретические модели вычислительных процессов

3.1 Взаимодействующие последовательные процессы

Наиболее эффективной сферой применения результатов и рекомендаций теоретического программирования и вычислительной математики, служит спецификация, разработка и реализация вычислительных систем, которые непрерывно действуют и взаимодействуют со своим окружением. На основе модели взаимодействующих последовательных процессов (**ВПП**) эти системы можно разложить на параллельно работающие подсистемы, взаимодействующие как друг с другом, так и со своим общим окружением.

Такой подход обладает целым рядом преимуществ. *Во-первых*, он позволяет избежать многих традиционных для параллельного программирования проблем, таких, как взаимное влияние и взаимное исключение, прерывания, семафоры, многопоточная обработка и т. д.

Во-вторых, он включает в себя в виде частных случаев модели структурного программирования: мониторы, классы, модули, пакеты, критические участки, конверты, формы и даже подпрограммы.

В-третьих, он позволяет избежать такие ошибки как расходимость, тупики, зацикливание.

3.1.1 Базовые определения

Неформально, *процесс* можно представить себе как группу ячеек памяти, содержимое которых меняется по определенным правилам. В ЭВМ эти правила описываются программой, которую интерпретирует процессор. Синоним термина «Процесс», – «задача», «программа».

«Задача – основная единица, подчиняющаяся управляющей программе в мультипрограммном режиме»; «Процесс – это программа, выполняемая псевдопроцессором»; «Процесс – это то, что происходит при выполнении программы на ЭВМ».

Хорнинг и Ренделл построили формальное определение понятие *процесса*. Основными понятиями модели являются:

- набор переменных состояния;
- состояние;
- пространство состояний;
- действия;
- работа;
- функция действия;

– процесс;
– начальное состояние.

В модели **ВПП** понятие *процесс* используется для обозначения поведения объекта. Для формального описания поведения объекта в **ВПП** необходимо сначала выделить в таком поведении наиболее важные *события* или *действия*, и выбрать для каждого из них подходящее название, или имя.

В случае простого автомата, торгующего шоколадками, существуют два вида событий;

мон - опускание монеты в щель автомата,

шок - появление шоколадки из выдающего устройства.

Имя каждого события обозначает целый *класс* событий; отдельные вхождения события внутри одного класса разделены во времени. Множество имен событий, выбранных для конкретного описания объекта, называется его *алфавитом.*

Считается, что конкретное событие в жизни объекта происходит мгновенно, т. е. является элементарным действием, не имеющим протяженности во времени. Протяженное действие следует рассматривать как пару событий, первое из которых отмечает начало действия, а второе - его завершение. Два протяженных действия перекрываются по времени, если начало каждого из них предшествует завершению другого. Когда совместность событий существенна (например, при синхронизации), такие события сводятся в одно событие, или же совместные события происходят в любом относительно друг друга порядке.

Введем следующие соглашения:

1) Имена событий будем обозначать словами, составленными из строчных букв, например, *шок* , а также буквами $a, b, c, \ldots$
2) Имена процессов будем обозначать словами, составленными из прописных букв, например, $ТАП$ - простой торговый автомат, а буквами $P, Q, R, \ldots$ будем обозначать произвольные процессы.
3) Буквы $x, y, z, \ldots$ используются для переменных, обозначающих события.
4) Буквы A, B, C используются для обозначения множества событий.
5) Буквы X, Y используются для переменных, обозначающих процессы.
6) Алфавит процесса P обознается αP $\left(\alpha ТАП = \{мон, шок\}\right)$.
7) Процесс с алфавитом V , такой, что в нем не происходит ни одно событие из V , назовем $СТОП_A$. Этот процесс описывает поведение сломанного объекта. Далее определим систему обозначений,

которая также предназначена для описания поведения объектов.

Префиксы

Пусть x - событие, а P - процесс. Тогда $(x \to P)$ «P за x» описывает объект, который вначале участвует в событии x, а затем ведет себя в точности как P, где $\alpha(x \to P) \quad \alpha P,\ x \in \alpha P$.

Пример. Простой торговый автомат, который благополучно обслуживает двух покупателей и затем ломается:

$$\left(мон \to \left(шок \to \left(мон \to \left(шок \to СТОП_{\alpha ТАП}\right)\right)\right)\right).$$

Рекурсия

Префиксную запись можно использовать для полного описания поведения процесса, который рано или поздно останавливается. Было бы желательно, чтобы этот способ был компактным и не требовал знать заранее срок жизни объекта. Рассмотрим простой долговечный объект - часы, функционирование которых состоит в том, чтобы тикать.

$$\alpha ЧАСЫ = \{тик\}.$$

Теперь рассмотрим объект, который вначале издает единственный «*тик*», а затем ведет себя в точности как *ЧАСЫ*

$$(тик \to ЧАСЫ).$$

Поведение этого объекта неотличимо от поведения исходных часов. Следовательно, один и тот же процесс описывает поведение обоих объектов. Эти рассуждения позволяют сформулировать равенство

$$ЧАСЫ = (тик \to ЧАСЫ).$$

Это уравнение можно развертывать простой заменой в правой части уравнения члена *ЧАСЫ* на равное ему выражение $(тик \to ЧАСЫ)$ столько раз, сколько нужно, при этом возможность для дальнейшего развертывания сохраняется.

Рекурсивный метод определения процесса, будет правильно работать, только если в правой части уравнения рекурсивному вхождению имени процесса предшествует хотя бы одно событие. *Например*, рекурсивное «определение» $X = X$ не определяет ничего, так как решением этого уравнения может служить все что угодно. Описание процесса, начинающееся с префикса, называется *предваренным.*

Определение. Если $F(X)$ - предваренное выражение, содержащее имя процесса X, а V - алфавит X, то уравнение $X = F(X)$ имеет единственное решение в алфавите V. Это решение обозначают выражением

$$\mu X : V.F(X).$$

Пример. Простой торговый автомат, полностью удовлетворяющий спрос на шоколадки:

$$ТАП = (мон \to (шок \to ТАП)).$$

Решение этого уравнения может быть записано в виде

$$ТАП = \mu X : \{мон, шок\}.(мон \to (шок \to X)).$$

Выбор

Используя префиксы и рекурсию, можно описывать объекты, обладающие только одной возможной линией поведения. Однако поведение многих объектов зависит от окружающей их обстановки.

Если x и y - различные события, то $(x \to P | y \to Q)$ описывает объект, который сначала участвует в одном из событий x, y, где

$$\alpha(x \to P | y \to Q) \quad \alpha P, \not{x}, y \in \alpha P \text{ и } \alpha P = \alpha Q.$$

Последующее же поведение объекта описывается процессом P, если первым произошло событие x, или Q, если первым произошло событие y.

Пример. Процесс копирования состоит из следующих событий:

вв.0 - считывание нуля из входного канала,

вв.1 - считывание единицы из входного канала,

выв.0 - запись нуля в выходной канал,

выв.1 - запись единицы в выходной канал.

Поведение процесса состоит из повторяющихся пар событий. На каждом такте он считывает, а затем записывает один бит.

$$КОПИБИТ = \mu X : (вв.0 \to выв.0 | вв.1 \to выв.1 \to X).$$

Определение выбора легко обобщить на случай более чем двух альтернатив. В общем случае если B - некоторое множество событий, а $P(x)$ - выражение, определяющее процесс для всех различных x из B, то запись

$$(x : B \to P(x))$$

определяет процесс, который сначала предлагает на выбор любое событие y из B, а затем ведет себя как $P(y)$.

Взаимная рекурсия

Рекурсия позволяет определить единственный процесс как решение некоторого единственного уравнения. Эта техника легко обобщается на случай решения систем уравнений с более чем одним неизвестным.

Для достижения желаемого результата необходимо, чтобы правые части всех уравнений были предваренными, а каждый неизвестный процесс входил ровно один раз в правую часть одного из уравнений.

Пример. Автомат с газированной водой имеет рукоятку с двумя возможными положениями - *ЛИМОН* и *АПЕЛЬСИН*. Действия по установке рукоятки в соответствующее положение назовем *устлимон* и *устапельсин*, а действия автомата по наливанию напитка - *лимон* и *апельсин*. Вначале рукоятка занимает некоторое нейтральное положение, к которому затем уже не возвращается. Поведение трех процессов описывается уравнением:

$\alpha\, АГАЗ = \alpha G = \alpha W = \{устлимон, устапельсин, мон, лимон, апельсин\}$.

$АГАЗ = (устлимон \to G \mid устапельсин \to W)$,

$G = (мон \to \ лимон \to G \mid устапельсин \to W)$,

$W = (мон \to апельсин \to W \mid устлимон \to G)$.

3.1.2 Законы взаимодействия последовательных процессов

Тождественность процессов с одинаковыми алфавитами можно устанавливать с помощью алгебраических законов, очень похожих на законы арифметики.

<u>*Закон*</u> 1. Два процесса, определенные с помощью оператора выбора, различны, если на первом шаге они предлагают различные альтернативы или после одинакового первого шага ведут себя по-разному. Если же множества начального выбора оказываются равными и для каждой начальной альтернативы дальнейшее поведение процессов совпадает, то, очевидно, что процессы тождественны:

$(x : A \to P(x)) = (y : B \to Q(y)) \equiv (A \quad B$ **AND** $\forall x \in A.P(x) = Q(x))$.

Следствие 1:

$СТОП = (a \to P)$

Следствие 2:

$(c \to P) \neq (d \to Q)$, если $c \neq d$.

Следствие 3:

$(c \to P \mid d \to Q) = (d \to Q \mid c \to P)$.

Следствие 4:

$(c \to P) = (c \to Q) \equiv P = Q$.

Пример.

$$(мон \to шок \to мон \to шок \to СТОП) \neq (мон \to СТОП).$$

Доказательство: Следует из следствий 2 и 1.

Закон 2. Всякое должным образом предваренное рекурсивное уравнение имеет единственное решение. Если $F(X)$ - предваренное выражение, то $(Y = F(Y)) \equiv (Y = \mu X.F(X))$.

Следствие.

$\mu X.F(X)$ является решением соответствующего уравнения $\mu X.F(X) = F(\mu X.F(X))$

Пример. Пусть $TA1 = (мон \to TA2)$, а $TA2 = (шок \to TA1)$. Требуется доказать, что $TA1 = TAП$.

Доказательство:

$TA1 = (мон \to TA2) =$ по определению $TA1 = (мон \to (шок \to TA1))$.

Таким образом, $TA1$ является решением того же рекурсивного уравнения, что и $TAП$. Так как это уравнение предварённое, оно имеет единственное решение. Значит, $TA1 = TAП$.

3.1.3 Реализация процессов

Любой процесс P, записанный с помощью введенных обозначений, можно представить в виде

$$(x : B \to F(x)),$$

где F - функция, ставящая в соответствие множеству символов множество процессов. Множество B может быть пустым (в случае $P = СТОП$), может содержать только один элемент (в случае префикса) или - более одного элемента (в случае выбора).

Таким образом, каждый процесс можно рассматривать как *функцию* F с областью определения B (множество начальных событий), и областью значения $\{F(x) | x \in B\}$.

Каждое событие из алфавита процесса представлено атомом $("мон")$. При этом если символ не может быть начальным событием процесса, то результатом функции будет специальный символ $"BLEEP"$. *Например*, для процесса $(x : \varnothing \to СТОП(x))$ значением функции будет $"BLEEP"$, что обозначим

$$СТОП = \lambda x."DLEEP"$$

Если же аргумент является событием, возможным для процесса, результатом функции будет другая функция, определяющая последую-

щее поведение процесса.

Пример. Функция, реализующая процесс $(c \to P)$ может иметь вид:

$$\textit{префикс}(c, P) = \lambda x.\textbf{if } x = c \textbf{ then } P \textbf{ else } "BLEEP".$$

Пример. Функция, реализующая двуместный выбор $(c \to P | d \to Q)$ может иметь вид:

$$\textit{выбор}(c, d, P, Q) = \lambda x.\textbf{if } x = c \textbf{ then } P \textbf{ else if } x = y \textbf{ then } Q \textbf{ else } "BLEEP".$$

3.1.4 Протоколы поведения процесса

Протоколом поведения процесса называется конечная последовательность символов, фиксирующая события, в которых процесс участвовал до некоторого момента времени. Можно представить себе наблюдателя с блокнотом, который следит за процессом и записывает имя каждого происходящего события.

Будем обозначать протокол последовательностью символов, разделенной запятыми и заключенной в угловые скобки, например, протокол $\langle x, y \rangle$ состоит из двух событий - x и следующего за ним y, $\langle x \rangle$ - состоит из одного события x, а протокол $\langle\ \rangle$ - пустой протокол.

Пример. Протокол простого торгового автомата *ТАП* в момент завершения обслуживания первых двух покупателей:

$$\langle \textit{мон}, \textit{шок}, \textit{мон}, \textit{шок} \rangle$$

Протокол того же автомата перед тем, как второй покупатель вынул свою шоколадку:

$$\langle \textit{мон}, \textit{шок}, \textit{мон} \rangle.$$

3.1.5 Операции над протоколами

Протоколам принадлежит основная роль в фиксировании, описании и понимании поведения процессов. Введем следующие обозначения:

s, t, u - протоколы,

S, T, U - множества протоколов,

f, g, h – функции.

Конкатенация

Наиболее важной операцией над протоколами является конкатенация $s \hat{} t$, которая строит новый протокол из пары протоколов s и t, просто соединяя их в указанном порядке. *Например*,

$$\langle \textit{мон}, \textit{шок} \rangle \hat{} \langle \textit{мон} \rangle = \langle \textit{мон}, \textit{шок}, \textit{мон} \rangle.$$

Самые важные свойства конкатенации - это ее ассоциативность и то, что пустой протокол $\langle\ \rangle$ служит для нее единицей:

$$s \wedge \langle\ \rangle = \langle\ \rangle \wedge s$$
$$s \wedge (t \wedge u) = (s \wedge t) \wedge u$$

Пусть f - функция, отображающая протоколы в протоколы. Будем говорить, что функция *строгая,* если она отображает пустой протокол в пустой протокол: $f(\langle\ \rangle) = \langle\ \rangle$.

Будем говорить, что функция f *дистрибутивна,* если $f(s \wedge t) = f(s) \wedge f(t)$.

Все дистрибутивные функции являются строгими.

Если n - натуральное число, то t^n будет обозначать конкатенацию n копий протокола t. Отсюда следует:

$$t^{n+1} = t \wedge t^n;$$
$$(s \wedge t)^{n+1} = s \wedge (s \wedge t)^n \wedge t.$$

Сужение

Выражение $(t \uparrow A)$ обозначает протокол t, суженный на множество символов A; он строится из t отбрасыванием всех символов, не принадлежащих A.

Сужение дистрибутивно и поэтому строго.

$$\langle\ \rangle \uparrow A = \langle\ \rangle;$$
$$(s \wedge t) \uparrow A \quad (s \uparrow A) \wedge (t \uparrow A).$$

Эффект сужения на одноэлементных последовательностях очевиден:

$$\langle x \rangle \uparrow A \quad \langle x \rangle, \text{ если } x \in A;$$
$$\langle y \rangle \uparrow A \quad \langle\ \rangle, \text{ если } y \notin A.$$

Приведенные ниже законы раскрывают взаимосвязь сужения $\uparrow$ и операций над множествами:

$$s \uparrow \varnothing = \langle\ \rangle;$$
$$(s \uparrow A) \uparrow B = s \uparrow (A \cap B).$$

Голова и хвост

Если s - непустая последовательность, обозначим ее первый элемент s_0, а результат, полученный после его удаления - s'. *Например*, $\langle x, y, x \rangle_0 = x$, $\langle x, y, x \rangle' = y$. Обе эти операции не определены для пустой последовательности.

$$\left(\langle x\rangle \hat{} s\right)_0 = x;$$
$$\left(\langle x\rangle \hat{} s\right)' = s;$$
$$s = \left(\langle s_0\rangle \hat{} s'\right), \text{ если } s \neq \langle\ \rangle.$$

Закон равенства или неравенства двух протоколов:

$$s = t \equiv \left(s \quad t \quad \langle\ \rangle = \textbf{OR}\ \left(s_0 \quad t_0 = \textbf{AND}\ s' \quad t'\right)\right). \qquad =$$

Звёздочка

Множество A^* - это набор всех конечных протоколов (включая $\langle\ \rangle$), составленных из элементов множества A. После сужения на A такие протоколы остаются неизменными, Отсюда следует простое определение:

$$A^* = \left\{s \middle| \left(s \uparrow A\right) = s\right\}.$$

Следствиями этого определения являются законы:

$$\langle\ \rangle \in A^*;$$
$$\langle x\rangle \in A^* \equiv x \in A;$$
$$\left(s \hat{} t\right) \in A^* \equiv s \in A^* \ \textbf{AND}\ t \in A^*.$$

Они обладают достаточной мощностью, чтобы определить, принадлежит ли протокол множеству A^*.

Например, если $x \in A$, а $y \notin A$, то

$$\begin{aligned}\langle x, y\rangle \in A^* &\equiv \left(\langle x\rangle \hat{} \langle y\rangle\right) \in A^* \\ &\equiv \left(\langle x\rangle \in A^*\right) \textbf{ AND } \left(\langle y\rangle \in A^*\right) \\ &\equiv \textbf{T AND F} = \textbf{F}.\end{aligned}$$

Порядок

Если s - копия некоторого начального отрезка t, то можно найти такое продолжение u последовательности s, что $s \hat{} u = t$. Определим отношение порядка

$$s \leq t \quad \left(\exists u : s \hat{} u = t\right).$$

и будем говорить, что s является *префиксом* t. Например: $\langle x, y\rangle \leq \langle x, y, z\rangle$. Отношение $\leq$ является частичным упорядочением и имеет своим наименьшим элементом $\langle\ \rangle$. Об этом говорят законы

$\langle\ \rangle \leq s$	наименьший элемент.
$s \leq s$	рефлексивность.

$s \le t$ **AND** $t \le s \Rightarrow t = s$ антисимметричность.

$s \le t$ **AND** $t \le u \Rightarrow s \le u$ транзитивность.

Следующий закон позволяет определить, является ли справедливым отношение $s \le t$:

$$\left(\langle x \rangle \hat{} s\right) \le t \equiv t \ne \langle \Rightarrow \rangle \textbf{ AND } x \quad t_0 \textbf{ AND } s \le t'$$

Будем говорить, что функция f из множества протоколов во множество протоколов *монотонна,* если она сохраняет отношение порядка $\le$, т. е. $f(s) \le f(t)$ всякий раз, когда $s \le t$.

Длина

Длину протокола t будем обозначать $\#t$. Например, $\#\langle x, y, z \rangle = 3$.

Следующие законы определяют операцию *#:*

$$\#\langle\ \rangle = 0;$$
$$\#\langle x \rangle = 1;$$
$$\#(s \hat{} t) = \#s + \#t.$$

Число вхождений символа x в протокол s определяется как $s \downarrow x \quad \#(s \uparrow \{x\})$.

3.1.6 Протоколы процесса

Протокол это последовательная запись поведения процесса вплоть до некоторого момента времени. До начала процесса неизвестно, какой именно из возможных протоколов будет реализован: его выбор зависит от внешних по отношению к процессу факторов. Однако полный набор всех возможных протоколов процесса P *может* быть известен заранее. Введем функцию *протоколы*(P) для обозначения этого множества.

Пример. Единственным протоколом процесса *СТОП* является $\langle\ \rangle$:

протоколы$(СТОП) = \{\langle\ \rangle\}$.

Пример.

протоколы$(ЧАСЫ) = \{\langle\ \rangle, \langle тик \rangle, \langle тик,\ тик \rangle, \ldots\} = \{тик\}^*$.

Здесь множество протоколов бесконечно.

Законы

Закон 1

протоколы$(СТОП) = \{t \mid t = \langle\ \rangle\} = \{\langle\ \rangle\}$.

Протокол процесса $(c \to P)$ может быть пустым, поскольку $\langle\ \rangle$

является протоколом поведения любого процесса до момента наступления его первого события.

Закон 2

$$протоколы(c \to P) \quad \{t | t = \langle\ \rangle \textbf{ OR } (t_0 = c \textbf{ AND } t' \in протоколы(P))\}.$$

Эти два закона можно объединить в один общий закон, которому подчиняется конструкция выбора:

Закон 3

$$протоколы(x : B \to P(x)) \quad \{t | t = \langle\ \rangle \textbf{ OR } (t_0 \in B \ AND \ t' \in протоколы(P(t_0)))\}$$

Протоколы рекурсивно определенного процесса описываются *законом* 4:

$$протоколы(\mu X : A.F(X)) = \bigcup_{n \geq 0} протоколы(F^n(СТОП_A))$$

Пример.

$$протоколы(ТАП) = \bigcup_{n \geq 0} \{s | s \leq \langle мон, шок \rangle^n\}$$

$\langle\ \rangle$ является протоколом любого процесса до момента наступления его первого события. Кроме того, если $(s \hat{}\ t)$ – протокол процесса до некоторого момента, то s должен быть протоколом того же процесса до некоторого более раннего момента времени. Наконец, каждое происходящее событие должно содержаться в алфавите процесса. Три этих факта находят свое формальное выражение в следующих *законах*:

$$\langle\ \rangle \in протоколы(P)$$
$$s \hat{}\ t \in протоколы(P) \Rightarrow s \in протоколы(P)$$
$$протоколы(P) \subseteq \{\alpha P\}^*$$

После

Если $s \in протоколы(P)$, то P/s (P после s) – это процесс, ведущий себя так, как ведет себя P с момента завершения всех действий, записанных в протоколе s. Если s не является протоколом P, то P/s не определено.

Пример.

$$(ТАП/\langle мон \rangle) = (шок \to ТАП)$$

3.1.7 Спецификации

Спецификация изделия – это описание его предполагаемого поведения. Это описание представляет собой предикат, содержащий сво-

бодные переменные, каждая из которых соответствует некоторому обозримому аспекту поведения изделия.

Например, спецификация электронного усилителя с входным диапазоном в один вольт и с усилением входного напряжения приблизительно в 10 раз задается предикатом

$$УСИЛ10 = \left(0 \le u \le 1 \Rightarrow \left|u' - 10 \times u\right| \le 1\right).$$

В этой спецификации u обозначает входное, а u' - выходное напряжения.

В случае процесса в качестве результата наблюдения за его поведением рассматривается протокол событий, произошедших вплоть до данного момента времени. Для обозначения произвольного протокола процесса будем использовать специальную переменную *пр* .

Пример. Владелец торгового автомата не желает терпеть убытков. Поэтому он оговаривает, что число выданных шоколадок не должно превышать числа опущенных монет:

$$НЕУБЫТ = \left(\#\left(пр \uparrow \{шок\}\right) \le \#\left(пр \uparrow \{мон\}\right)\right) \quad пр \downarrow шок \le пр \downarrow мон.$$

Пользователь автомата хочет быть уверенным в том, что машина не будет поглощать монеты, пока не выдаст уже оплаченный шоколад:

$$ЧЕСТН = \left(пр \downarrow мон \le \left(пр \downarrow шок + 1\right)\right).$$

Изготовитель торгового автомата должен учесть требования, как владельца, так и клиента:

$$ТАПВЗАИМ = НЕУБЫТ \ \textbf{AND} \ ЧЕСТН \quad \left(0 \le \left(пр \downarrow мон - пр \downarrow шок\right) \le 1\right)$$

Если P - объект, отвечающий спецификации S, то говорят, что P *удовлетворяет* $\left(P \ \textbf{уд} \ S\right)$.

Это означает, S истинно всякий раз, когда его переменные принимают значения, полученные в результате наблюдения за объектом P :

$$\forall пр.пр \in протоколы\left(P\right) \Rightarrow S.$$

Законы, описывающие наиболее общие свойства отношения *удовлетворяет,* приведены ниже.

Спецификации *истина,* не накладывающей никаких ограничений на поведение, будут удовлетворять все объекты.

$$P \ \textbf{уд} \ \textbf{F} .$$

Если объект удовлетворяет двум различным спецификациям, он удовлетворяет также и их конъюнкции:

$$\textbf{if} \ \left(P \ \textbf{уд} \ S\right) \ \textbf{AND} \ \left(P \ \textbf{уд} \ T\right) \ \textbf{then} \ P \ \textbf{уд} \left(S \ \textbf{AND} \ T\right).$$

Пусть $S(n)$ - предикат, содержащий переменную n и P не зависит от n.

$$\textbf{if } \forall n.\left(P \textbf{ уд } S(n)\right) \textbf{ then } P \textbf{ уд } \forall n.S(n).$$

Если из спецификации S логически следует другая спецификация T, то всякое наблюдение, описываемое S, описывается также и T.

$$\textbf{if } \left(P \textbf{ уд } S\right) \textbf{ AND } \left(S \Rightarrow T\right) \textbf{ then } P \textbf{ уд } T$$

При проектировании изделия разработчик несёт ответственность за то, чтобы оно соответствовало своей спецификации. Ниже приводится набор законов, позволяющих с помощью математических рассуждений убедиться в том, что процесс P соответствует своей спецификации S.

Результатом наблюдения за процессом *СТОП* всегда будет пустой протокол:

$$СТОП \textbf{ уд } \left(np = \langle\ \rangle\right).$$

Протокол процесса $(c \to P)$ вначале пуст. Каждый последующий протокол начинается с c, а его хвост является протоколом P.

$$\textbf{if } P \textbf{ уд } S(np) \textbf{ then } \left(c \to P\right) \textbf{ уд } \left(np \quad \not= \rangle\ OR\ \left(np_0 \quad \in \textbf{ AND } S(np')\right)\right)$$

Все приведенные выше законы являются частными случаями закона для обобщенного оператора выбора:

$$\textbf{if } \forall x \in B.\left(P(x) \textbf{ уд } S(np, x)\right) \textbf{ then}$$

$$\left(x : B \to P(x)\right) \textbf{ уд } \left(np = \langle\ \rangle \textbf{ OR } \left(np_0 \in B \textbf{ AND } S(np', np_0)\right)\right)$$

Закон, устанавливающий корректность рекурсивно определенного процесса.

Пусть $F(X)$ - предваренное выражение. $СТОП \textbf{ уд } S$, а $\left(X \textbf{ уд } S\right) \Rightarrow \left(F(X) \textbf{ уд } S\right)$, тогда

$$\left(\mu X.F(X) \textbf{ уд } S\right)$$

3.2 Параллельные процессы

Процесс определяется полным описанием его потенциального поведения. При этом часто имеется выбор между несколькими различными действиями. В каждом таком случае выбор того, какое из событий произойдет в действительности, может зависеть от окружения, в котором работает процесс. Само окружение процесса может быть описано как процесс, поведение которого определяется в введенных терминах. Это позволяет исследовать поведение целой системы, состоя-

щей из процесса и его окружения, взаимодействующих по мере их параллельного исполнения. Всю систему также следует рассматривать как процесс, поведение которого определяется в терминах поведения составляющих его процессов. Эта система в свою очередь может быть помещена в еще более широкое окружение, и т.д.

3.2.1 Взаимодействие процессов

Объединяя два процесса для совместного исполнения, как правило, хотят, чтобы они взаимодействовали друг с другом. Эти взаимодействия можно рассматривать, как события, требующие одновременного участия обоих процессов.

Законы

Законы, управляющие поведением $(P \| Q)$, параллельно развивающихся процессов P и Q, достаточно просты. *Первый закон* выражает логическую симметрию между процессом и его окружением:

$$P \| Q = Q \| P .$$

Закон 2. При совместной работе трех процессов неважно, в каком порядке они объединены оператором параллельной композиции ||:

$$P \| (Q \| R) = (P \| Q) \| R .$$

Закон 3. Процесс, находящийся в тупиковой ситуации, приводит к тупику всей системы.

$$P \| СТОП_{\alpha P} = СТОП_{\alpha P} .$$

Закон 4. Пара процессов с одинаковыми алфавитами либо одновременно выполняет одно и то же действие, либо попадает в состояние тупика, если начальные события процессов не совпадают:

$$(c \to P) \| (c \to Q) = c \to (P \| Q).$$
$$(c \to P) \| (d \to Q) = СТОП.$$

3.2.2 Параллелизм

Рассмотрим случай, когда операнды P и Q оператора || имеют неодинаковые алфавиты $\alpha P \neq \alpha Q$.

Когда такие процессы объединяются для совместного исполнения, события, содержащиеся в обоих алфавитах, требуют одновременного участия P и Q. События же из алфавита P, не содержащиеся в алфавите Q, не имеют никакого отношения к Q, который физически неспособен ни контролировать, ни замечать такие события. Такие события процесса

P могут происходить независимо от Q. Аналогично Q может самостоятельно участвовать в событиях, содержащихся в его алфавите, но отсутствующих в алфавите P. Таким образом, множество всех событий, логически возможных для данной системы, есть просто объединение алфавитов составляющих ее процессов $\alpha\left(P\|Q\right)=\alpha P\cup\alpha Q$.

Законы

Пусть $a\in\left(\alpha P-\alpha Q\right), b\in\left(\alpha Q-\alpha P\right), \{c,d\}\in\left(\alpha P\cap\alpha Q\right)$. Следующие *законы* показывают, каким образом процесс P один участвует в событии a, Q один участвует в b, а c и d требуют одновременного участия P и Q.

1. $\left(c\to P\right)\|\left(c\to Q\right)=c\to\left(P\|Q\right)$.
2. $\left(c\to P\right)\|\left(d\to Q\right)=СТОП$.
3. $\left(a\to P\right)\|\left(c\to Q\right)\quad a\to\left(P\|\left(c\to Q\right)\right)$.
4. $\left(c\to P\right)\|\left(b\to Q\right)=b\to\left(\left(c\to P\right)\|Q\right)$.
5. $\left(a\to P\right)\|\left(b\to Q\right)\quad\left(a\to\left(P\|\left(b\to Q\right)\right)\middle|\, b\left(a\to P\|Q\right)\right)$.

Эти законы можно обобщить для общего случая оператора выбора:

Пусть $P=\left(x:A\to P\left(x\right)\right)\square Q\quad\left(y:B\to Q\left(y\right)\right)$. Тогда

$$P\|Q=\left(z:C\to\left(P'\|Q'\right)\right),$$

где $C=\left(A\cap B\right)\cup\left(A-\alpha Q\right)\cup\left(B-\alpha P\right)$,

$P'=P\left(z\right)$ если $z\in A$, иначе $P'=P$,

$Q'=Q\left(z\right)$ если $z\in B$, иначе $Q'=Q$.

С помощью этих законов можно переопределить процесс, удалив из его описания, параллельный оператор, как это показано в следующем примере.

Пример.

Пусть $\alpha P=\{a,c\}\quad\alpha Q\quad\{b,c\}, P\quad\left(a\to c\to P\right), Q\quad\left(c\to b\to Q\right)$.

Тогда $P\|Q=\left(a\to c\to P\right)\|\left(c\to b\to Q\right)\quad a\to c\to\left(P\|\left(b\to Q\right)\right)$, а

$$\begin{aligned}\left(P\|\left(b\to Q\right)\right)\quad & a\to\left(\left(c\to P\right)\|\left(b\to Q\right)\right)\middle|\, b\to\left(P\|Q\right)\\ &=a\to b\to\left(\left(c\to P\right)\|Q\right)\middle|\, b\to\left(P\|Q\right)\\ &=a\to b\to c\to\left(P\|\left(b\to Q\right)\right)\middle|\, b\to a\to c\to\left(P\|\left(b\to Q\right)\right)\\ &=\mu X.\left(a\to b\to c\to X\middle|\, b\to a\to c\to X\right).\end{aligned}$$

Отсюда следует

$$P \| Q = a \to c \to \mu X.\left(a \to b \to c \to X \middle| b \to a \to c \to X\right).$$

Протоколы параллельных процессов

Пусть t - протокол $\left(P \| Q\right)$. Тогда все события из t, принадлежащие алфавиту αP, являлись событиями в жизни P, а все события из t, не принадлежащие αP, происходили без участия P. Таким образом, $\left(t \uparrow \alpha P\right)$ - это протокол всех событий, в которых участвовал процесс P, и поэтому он является протоколом P. По тем же соображениям $\left(t \uparrow \alpha Q\right)$ является протоколом Q. Более того, каждое событие из t должно содержаться либо в αP, либо в αQ. Эти рассуждения позволяют сформулировать <u>закон</u>

$$\begin{aligned} &\textit{протоколы}\left(P \| Q\right) = \left\{t \middle| \left(t \uparrow \alpha P\right) \in \textit{протоколы}\left(P\right) \textbf{ AND} \right. \\ &\left. \left(t \uparrow \alpha Q\right) \in \textit{протоколы}\left(Q\right) \textbf{ AND } t \in \left(\alpha P \cup \alpha Q\right)^* \right\} \end{aligned}$$

3.2.3 Задача об обедающих философах

Рассмотрим трактовку задачи предложенной Дейкстрой об обедающих философах в терминах параллельных процессов.

«*В некоем пансионе нашли пристанище пять философов. У каждого философа была своя комната. Была у них и общая столовая с круглым столом, вокруг которого стояли пять стульев. В центре стола стояла большая миска спагетти, содержимое которой постоянно пополнялось. На столе также лежало пять вилок, по одной между соседними посадочными местами. Звали философов* $ФИЛ_0, ФИЛ_1, ФИЛ_2, ФИЛ_3, ФИЛ_4$ *и за столом они располагались в этом же порядке, против часовой стрелки. Почувствовав голод, философ шёл в столовую, садился на свой стул, брал сначала слева от себя вилку, затем справа и приступал к еде. Закончив трапезу, он клал на место обе вилки, выходил из-за стола, и уходили размышлять*».

Алфавиты

Теперь построим математическую модель этой системы. Сначала надо выбрать множества событий. Для $ФИЛ_i$ определим это множество как

$$\begin{aligned} &\alpha ФИЛ_i = \left\{i.садится, i.встает, i.берет\ вил.i, \right. \\ &\left. i.берет\ вил.\left(i +_5 1\right), i.кладет\ вил.i, i.кладет\ вил.\left(i +_5 1\right)\right\}, \end{aligned}$$

где $+_5$ - сложение по модулю 5.

Другие «действующие лица» - это пять вилок, каждая из которых имеет тот же номер, что и философ, которому она принадлежит. Вил берет и кладет на место или сам этот философ, или его сосед слева. Алфавит *i*-й вилки определим как

$$\alpha ВИЛКА_i = \{i.берет\ вил.i, (i -_5 1).берет\ вил.i,$$
$$i.кладет\ вил.i, (i -_5 1).кладет\ вил.i\}$$

где $-_5$ обозначает вычитание по модулю 5.

Таким образом, каждое событие, кроме *садится* и *встает,* требует участия двух соседних действующих лиц - философа и вилки.

Поведение

Жизнь каждого философа представляет собой повторение цикла из шести событий:

$$ФИЛ_i = (i.садится \to i.берет\ вил.i \to i.берет\ вил.(i +_5 1) \to$$
$$i.кладет\ вил.i \to i.кладет\ вил.(i +_5 1) \to i.встает \to ФИЛ_i).$$

Вилку циклически берёт и кладёт на место кто-нибудь из соседних с ней философов:

$$ВИЛКА_i = (i.берет\ вил.i \to i.кладет\ вил.i \to ВИЛКА_i |$$
$$(i -_5 1).берет\ вил.i \to (i -_5 1).кладет\ вил.i \to ВИЛКА_i)$$

Поведение всего пансиона можно описать как параллельную комбинацию поведения компонент:

$$ФИЛОСОФЫ = (ФИЛ_0 \| ФИЛ_1 \| ФИЛ_2 \| ФИЛ_3 \| ФИЛ_4)$$
$$ВИЛКИ = (ВИЛКА_0 \| ВИЛКА_1 \| ВИЛКА_2 \| ВИЛКА_3 \| ВИЛКА_4)$$
$$ПАНСИОН = (ФИЛОСОФЫ \| ВИЛКИ).$$

Тупик

Построенная математическая модель системы позволяет обнаружить опасность возникновения тупиковой ситуации, когда каждый из философов возьмёт вилку в левую руку. Одним из решений этой проблемы явилось введение в систему слуги. Слуге даётся указание следить за тем, чтобы за столом никогда не оказывалось больше четырех философов одновременно. Алфавит его определяется как $C \cup B$, где

$$C = \{0.садится, \ldots, 4.садится\},\ B = \{0.встает, \ldots, 4.встает\}.$$

Поведение слуги проще всего описать с помощью взаимной рекурсии. Пусть $СЛУГА_j$ определяет поведение слуги, когда за столом сидят j философов:

$$СЛУГА_0 = \left(x : C \to СЛУГА_1\right),$$
$$СЛУГА_j = \left(x : C \to СЛУГА_{j+1}\right) \| \left(y : B \to СЛУГА_{j-1}\right),\ j \in \{1,2,3\},$$
$$СЛУГА_4 = \left(y : B \to СЛУГА_3\right)$$

Пансион, свободный от тупика, определяется как

$$НОВПАНСИОН = \left(ПАНСИОН \| СЛУГА_0\right).$$

Бесконечный перехват

Помимо тупика обедающего философа подстерегает опасность бесконечного перехвата инициативы. Предположим, что левая рука у сидящего философа очень медлительна, в то время как его левый сосед очень проворен.

Всякий раз, когда философ пытается дотянуться до своей левой вилки, его левый сосед мгновенно перехватывает вил. Таким образом, может оказаться, что сидящий философ никогда не приступит к еде.

Таким образом, при реализации системы необходимо добиваться, чтобы любое желаемое и возможное событие обязательно наступило в пределах разумного интервала времени.

3.2.4 Помеченные процессы

Помечать процессы особенно полезно при создании групп сходных процессов, которые в режиме параллельной работы представляют некоторые идентичные услуги их общему окружению, но никак не взаимодействуют друг с другом. Это означает, что все они должны иметь взаимно непересекающиеся алфавиты. Чтобы этого достичь, снабдим каждый процесс отдельным именем; каждое событие помеченного процесса помечено тем же именем. Помеченное событие $l.x$, где x – его имя, а l – метка.

Процесс P с меткой l обозначают $l : P$. Он участвует в событии $l.x$, когда по определению P участвует в x.

Пример. Два работающих рядом торговых автомата

$$\left(лев : ТАП\right) \| \left(прав : ТАП\right).$$

Алфавиты этих процессов не пресекаются, и каждое происходящее событие помечено именем того устройства, на котором оно происходит. Если перед их параллельной установкой устройства не получили имен, каждое событие будет требовать участия их обоих, и тогда пара машин будет неотличима от одной. Это является следствием того, что $\left(ТАП \| ТАП\right) = ТАП$.

3.2.5 Множественная пометка

Определение пометки можно расширить, позволив помечать каждое событие любой меткой l из некоторого множества L. Если P – процесс, определим $(L:P)$ как процесс, ведущий себя в точности как P с той разницей, что он участвует в событии $l.x$ (где $l \in L$, $x \in \alpha P$), если по определению P участвует в x.

Пример. Лакей – это младший слуга, который имеет одного хозяина, провожает его к столу и из-за стола и прислуживает ему, пока тот ест:

$$\alpha ЛАКЕЙ = \{садится, встает\}$$
$$ЛАКЕЙ = (садится \rightarrow встает \rightarrow ЛАКЕЙ).$$

Лакея обслуживающего всех пятерых философов по очереди определим:

$$L = \{0,1,2,3,4\}$$
$$ОБЩИЙ\ ЛАКЕЙ = (L : ЛАКЕЙ).$$

Общего лакея можно нанимать на период отпуска слуги для предохранения обедающих философов от тупиковой ситуации. Конечно, в течение этого времени философам придется поголодать, ибо находиться за столом они смогут только по очереди.

3.3 Взаимодействие – обмен сообщениями

Выше события рассматривались как действия, не имеющего протяженности во времени, наступление которого может требовать одновременного участия более чем одного независимо описанного процесса. Важным для практического применения является специальный класс событий, называемых *взаимодействиями.* Взаимодействие состоит в передаче сообщений и является событием, описываемым парой $c.v$, где c – имя канала по которому происходит взаимодействие, а v – значение передаваемого сообщения.

Различают следующие виды каналов:

- *Синхронные.* Отправив сообщение, передающий процесс ожидает от принимающего подтверждение о приеме сообщения прежде, чем послать следующее сообщение, т. е. принимающий процесс не выполняется, пока не получит данные, а передающий - пока не получит подтверждение о приеме данных.
- *Асинхронно/синхронные*. Операция передачи сообщения асинхронная - она завершается сразу (сообщение копируется в некоторый буфер, а затем пересылается одновременно с работой процесса-отправителя), не ожидая того, когда данные будут получены

приемником. Операция приема сообщения синхронная: она блокирует процесс до момента поступления сообщения.

– *Асинхронные.* Обе операция асинхронные, то есть они завершаются сразу. Операция передачи сообщения работает, как и в предыдущем случае. Операция приема сообщения, обычно, возвращает некоторые значения, указывающие на то, как завершилась операция - было или нет, принято сообщение. В некоторых реализациях операции обмена сообщениями активируют сопроцессы, которые принимают/отправляют сообщения, используя временные буфера и соответствующие синхронные операции. В этом случае имеется еще синхронизирующая операции, которая блокирует процесс до тех пор, пока не завершатся все инициированные операции канала.

Множество всех сообщений, с помощью которых P может осуществлять взаимодействие по каналу c, определяется как

$$\alpha c(P) = \{v \mid c.v \in \alpha P\}.$$

Кроме того, определены функции, выбирающие имя канала $канал(c.v) = c$ и значение сообщения $сообщ(c.v) = v$.

Каналы используются для передачи сообщений только в одном направлении и только между двумя процессами.

3.3.1 Ввод и вывод

Пусть v – элемент $\alpha c(P)$. Процесс, который сначала выводит v по каналу c, а затем ведет себя как P, обозначим

$$(c!v \to P) = (c.v \to P).$$

Единственное событие, к которому этот процесс готов в начальном состоянии – это взаимодействие $c.v$.

Процесс, который сначала готов ввести любое значение x, предаваемое по каналу c, а затем ведет себя как $P(x)$, обозначим

$$(c?v \to P(x)) = (y : \{y \mid канал(y) \quad c\} \to P(сообщ(y))).$$

Пусть P, Q – процессы, c – выходной канал P и входной канал Q. Если P, Q объединены в параллельную систему $(P \| Q)$, то взаимодействие по каналу c будет происходить всякий раз, когда P выводит сообщение, а Q в тот же самый момент вводит его. Выводящий процесс однозначно определяет значение передаваемого сообщения, тогда как вводящий процесс готов принять любое поступающее значение. Поэтому в действительности происходящим событием будет взаимо-

действие $c.v$, где v – значение, определяемое выводящим процессом. Отсюда вытекает очевидное ограничение, что алфавиты на обоих концах канала должны совпадать, т.е. $\alpha c(P) = \alpha c(Q)$.

Пример. Процесс, копирующий каждое сообщение, поступающее слева, выводя его направо:

$$\alpha лев(КОПИР) = \alpha прав(КОПИР)$$
$$КОПИР = \mu X.(лев?x \to прав!x \to X).$$

Пример. Процесс, похожий на $КОПИР$, с той разницей, что каждое вводимое число перед выводом удваивается:

$$\alpha лев = \alpha прав = Nat$$
$$УДВ = \mu X.(лев?x \to прав!(x+x) \to X).$$

Пример. Процесс, принимающий сообщение по одному из двух каналов $лев1$ и $лев2$ и немедленно передающий его направо:

$$\alpha лев1 = \alpha лев = \alpha прав$$
$$СЛИЯНИЕ = (лев1?x \to прав!x \to СЛИЯНИЕ \mid лев2?x \to прав!x \to СЛИЯНИЕ)$$

3.3.2 Взаимодействия

Пусть P, Q – процессы, c – выходной канал P и входной канал Q. Тогда множество, состоящее из событий-взаимодействий $c.v$, содержится в пересечении алфавита P с алфавитом Q. Если процессы объединены в параллельную систему $(P \| Q)$, то взаимодействие $c.v$ может происходить только когда в этом событии одновременно участвуют оба процесса, т. е. в тот момент, когда P выводит значение v по каналу c, а Q одновременно вводит это значение. Вводящий процесс готов принять любое возможное поступающее сообщение, и поэтому то, какое именно значение передается, определяет выводящий процесс.

Таким образом, вывод можно рассматривать как специальный случай операции префиксации, а ввод – как специальный случай выбора.

$$(c!v \to P) \| (c?x \to Q(x)) \quad c!v \to (P \| Q(v)).$$

3.3.3 Подчинение

Пусть P, Q – процессы, такие, что $\alpha P \subseteq \alpha Q$. В комбинации $(P \| Q)$ каждое действие P может произойти, только если это позволяет Q, тогда, как Q может независимо осуществлять действия из $(\alpha P - \alpha Q)$ без разрешения партнера P. Таким образом, P служит по отношению к

Q подчиненным процессом, тогда как Q выступает как главный процесс. Это записывается так: $P//Q$. Отсюда: $\alpha(P//Q) = (\alpha P - \alpha Q)$.

Обычно давать подчиненному процессу дается имя (например m), которое используется главным процессом для всех взаимодействий с подчиненным. Каждое взаимодействие по каналу c обозначается тройкой $m.c.v$, где $\alpha m.c(m:P) = \alpha c(P)$, а $v \in \alpha c(P)$.

В конструкции $(m:P//Q)$ процесс Q взаимодействует с P по каналам с составными именами вида $m.c$, тогда как P для этих же взаимодействий использует соответствующий простой канал c. Так как все эти взаимодействия скрыты от обстановки, имя m недоступно снаружи; следовательно, оно служит локальным именем подчиненного процесса.

Подчинение может быть вложенным, *например* $(n:(m:P//Q)//R)$. Процесс R не имеет возможности ни непосредственно взаимодействовать с P, ни знать о существовании P и его имени m.

Пример. $удв : УДВ//Q$.

Подчиненный процесс ведет себя как обыкновенная подпрограмма, вызываемая внутри главного процесса Q. Внутри Q значение $2 \times e$ может быть получено поочередным выводом аргумента e по левому каналу процесса $удв$ и вводом результата по правому каналу:

$$удв.лев!e \to (удв.прав?x \to \ldots).$$

Пример. $ст : СТЕК//Q$.

Внутри главного процесса $Q.ст.лев!v$ используется для проталкивания в верхушку стека, а $ст.прав?x$ выталкивает верхнее значение. Использование конструкции выбора позволяет рассматривать ситуацию, когда стек пуст:

$$(ст.прав?x \to Q1(x) \,|\, ст.пуст \to Q2).$$

Если стек не пуст, то выбирается первый путь; если пуст, то выбор второго пути позволяет избежать тупика.

Оператор подчинения может использоваться для рекурсивного определения подпрограмм. Каждый уровень рекурсии (кроме последнего) задает новую локальную подпрограмму, работающую с рекурсивным вызовом.

Пример.

$$ФАКТ = \mu X.лев?n \to \left(\textbf{if } n \quad 0 \textbf{ then } (прав! \to X)\right)$$
$$\textbf{else}\left(f : X // \left(f.лев(n-1) \to f.прав?y \to прав!(n \times y) \to X\right)\right).$$

Программа *ФАКТ* использует каналы *лев* и *прав* для обмена результатами и параметрами с вызывающим ее процессом, а каналы $f.лев$ и $f.прав$ – для взаимодействия со своим подчиненным процессом f.

3.4 Разделяемые ресурсы

Обозначение $(m : //S)$ использовалось для именованного подчиненного процесса $(m : R)$, единственной обязанностью которого является удовлетворение потребностей главного процесса S. Предположим теперь, что S состоит из двух параллельных процессов $(P \| Q)$ и они оба нуждаются в услугах одного и того же подчиненного процесса $(m : R)$. P и Q не могут взаимодействовать с R по одним и тем же каналам, потому что тогда эти каналы должны содержаться в алфавитах обоих процессов, и, значит, согласно определению оператора ||, взаимодействия с $(m : R)$ могут происходить, только когда P и Q одновременно посылают одно и то же сообщение. Если же требуется чередование взаимодействий между P и $(m : R)$ с взаимодействиями между Q и $(m : R)$, то в этом случае $(m : R)$ служит как ресурс, разделяемый P и Q. Каждый из них использует его независимо, и их взаимодействия с ним чередуются.

Когда все процессы-пользователи известны заранее, можно организовать работу так, чтобы каждый процесс-пользователь имел свой набор каналов для взаимодействий с совместно используемым ресурсом. Эта техника применялась в задаче об обедающих философах: каждая вилка совместно использовалась всеми пятью. Общий метод разделения ресурсов дает множественная пометка, которая является эффективным средством создания достаточного числа каналов для независимого взаимодействия с каждым процессом-пользователем. Отдельные взаимодействия по этим каналам произвольно чередуются. Однако при таком методе необходимо знать заранее имена всех процессов-пользователей, и поэтому он не подходит для подчиненного процесса, предназначенного для обслуживания главного процесса, который разбивается на произвольное число параллельных подпроцессов.

3.4.1 Поочередное использование

Проблему, вызванную использованием комбинирующего оператора ||, можно избежать, используя параллелизм в форме чередования $(P|||Q)$. Здесь P и Q имеют одинаковые алфавиты, а их взаимодействия с совместно используемыми внешними процессами произвольно чередуются.

Пример. Совместно используемая подпрограмма: $удв : УДВ // (P|||Q)$.

Здесь и P и Q могут содержать вызов подчиненного процесса

$$(удв.лев!v \to удв.прав?x \to ПРОПУСК),$$

где *ПРОПУСК* – процесс, который ничего не делает, но благополучно завершается.

Пример. Совместно используемое алфавитно-цифровое печатающее устройство:

$$АЦПУ = занят \to \mu X.(лев?s \to h!s \to X | свободен \to АЦПУ).$$

Здесь h – канал, соединяющий *АЦПУ* с аппаратной частью устройства. После наступления события *занят АЦПУ* копирует в аппаратную часть последовательно поступающие по левому каналу строки, пока сигнал *свободен* не приведет его в исходное состояние, в котором он доступен для использования другими процессами. Этот процесс используется как разделяемый ресурс:

$$ацпу : АЦПУ // \ldots (P|||Q) \ldots$$

Внутри P или Q вывод последовательности строк, образующей файл, заключен между событиями *ацпу.занят* и *ацпу.свободен* :

$$ацпу.занят \to \ldots ацпу.лев!"Вася" \to \ldots ацпу.лев!следстр \to \ldots ацпу.свободен$$

3.4.2 Общая память

Поведение систем параллельных процессов реализуется на обыкновенной ЭВМ с хранимой программой с помощью режима разделения времени, при котором единственный процессор поочередно выполняет каждый из процессов, причем смена выполняемого процесса происходит по прерыванию, исходящему от некоторого внешнего или синхронизирующего устройства. При такой реализации легко позволить параллельным процессам совместно использовать общую память, выборка и загрузка которой осуществляется каждым процессом.

Ячейка общей памяти – это разделяемая переменная:

$$(счет : ПЕРЕМ // (счет.лев!0 \to (P|||Q))).$$

Вместе с тем, произвольное чередование присваиваний в ячейку общей памяти различными процессами является следствием многочисленных опасностей.

Пример. Взаимное влияние.

Разделяемая переменная *счет* используется для подсчета числа исполнений некоторого важного события. При каждом наступлении этого события соответствующий процесс P или Q пытается изменить значение счетчика парой взаимодействий

$$счет.прав?x;\cdots счет.лев!(x+1)$$

Эти два взаимодействия могут перемежаться аналогичной парой взаимодействий от другого процесса, в результате чего мы получим последовательность

$$счет.прав?x \to счет.прав?y \to счет.лев!(y+1) \to счет.лев!(x+1) \to ...$$

В итоге значение счетчика увеличится лишь на единицу, а не на два. Такого рода ошибки известны как *взаимное влияние* и часто допускаются при проектировании процессов, совместно использующих общую память. Кроме того, проявление такой ошибки недетерминировано; ее воспроизводимость очень ненадежна, и поэтому ее практически невозможно диагностировать обычными методами тестирования.

Возможным решением этой проблемы может быть контроль над тем, чтобы смена процесса не происходила при совершении последовательности действий, нуждающихся в защите от чередования. Такая последовательность называется *критическим участком.* При реализации с одним процессором требуемое исключение обычно достигается запрещением всех прерываний на протяжении критического участка. Нежелательным эффектом такого решения является задержка ответа на прерывание.

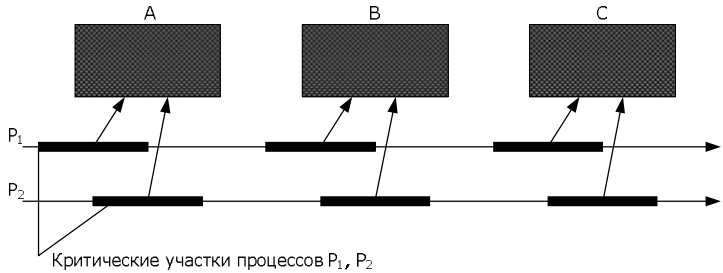


Рис. 3.1 Критическте участки

A, B, C – разделяемый ресурс, $P1, P2$ – процессы.

Для решения этой проблемы Э. Дейкстрой принадложил идею использования двоичных семафоров. Семафор можно описать как про-

цесс, поочередно выполняющий действия с именами P и V:

$$CEM = \left(P \to V \to CEM\right).$$

Он описывается как совместно используемый ресурс

$$\left(взаискл : CEM // \ldots\right).$$

При условии, что все процессы подчиняются этой дисциплине, каждый из двух процессов не сможет влиять на изменение счетчика - произвести действие *взаискл.V*. Таким образом, критический участок, на котором происходит увеличение счетчика, должен иметь вид

$$взаискл.\ P \to счет.прав\,?x \to счет\_лев\,!\left(x+1\right) \to взаискл.V \to \ \ldots$$

Если все процессы подчиняются этой дисциплине, каждый из двух процессов не сможет влиять на изменение счетчика своим партнером. Но если какой-нибудь процесс пропустит *P* или *V* или выполнит их в обратном порядке, результат будет непредсказуемым и может привести к катастрофической (неуловимой) ошибке.

Избежать взаимного влияния гораздо более надежным способом можно, встроив необходимую защиту в саму конструкцию общей памяти, воспользовавшись знанием о предполагаемом способе ее использования. Если, например, переменная будет использоваться только как счетчик, то ее увеличение можно задать одной элементарной операцией *счет.вверх*, а соответствующий разделяемый ресурс определить как CT_0:

$$счет : CT_0 // \left(\ldots P ||| Q \ldots\right).$$

В общем случае неоходимо, чтобы каждый совместно используемый ресурс заранее проектировался для своих целей и, чтобы в разработке системы с элементами параллелизма универсальная память не использовалась совместно. Этот метод не только предупреждает серьезную опасность случайного взаимного влияния, но и позволяет получать конструкции, поддающиеся эффективной реализации на сетях распределенных процессорных элементов, а также на одно- и многопроцессорных ЭВМ с физически общей памятью.

3.4.3 Кратные ресурсы

Выше анализировались параллельные процессы, которые различным образом могут совместно использовать один подчиненный процесс. Каждый процесс-пользователь соблюдает дисциплину чередования ввода и вывода или чередования сигналов *занят/свободен* с тем, чтобы в каждый момент времени разделяемым ресурсом пользовался не более чем один процесс. Такие ресурсы называют *последовательно переиспользуемыми.* На практике применяются массивы процессов,

представляющие кратные ресурсы с одинаковым поведением. Индексы в массиве означают тот факт, что каждый элемент достоверно взаимодействует только с использующим его процессом.

Будем использовать индексы и операторы с индексами, смысл которых очевиден. Например:

$$\|\|\|_{i<12} P_i = \left(P_0 \,|||\, P_1 \,|||\, \ldots \,|||\, P_{11}\right)$$

$$\|_{i\geq 0} \left(f(i) \to P_i\right) \quad \left(f(0) - P_0 \,|\, f(1) \to P_1 \,|\ldots\right)$$

В последнем примере f является взаимно однозначной для того, чтобы выбор между альтернативами осуществлялся обстановкой.

Пример. Повторно входимая подпрограмма

Последовательно переиспользуемая общая подпрограмма может обслуживать вызывающие ее процессы только по одному. Если выполнение подпрограммы требует значительных вычислений, соответствующие задержки могут возникнуть и в вызывающем процессе. Если же для вычислений доступны несколько процессоров, нам ничто не мешает позволить нескольким экземплярам подпрограммы параллельно исполняться на различных процессорах. Подпрограмма, имеющая несколько параллельно работающих экземпляров, называется *повторно входимой* и определяется как массив параллельных процессов

$$\textit{удв}: \left(\|_{i<27} \left(i : \textit{УДВ}\right) \right) // \ldots$$

Типичным вызовом этой подпрограммы будет

$$\left(\textit{удв}.3.\textit{лев}!30 \to \textit{удв}.3.\textit{прав}?\, y \to \textit{ПРОПУСК}\right).$$

Присутствие индекса 3 гарантирует, что результат вызова получен от того же самого экземпляра *удв*, которому были посланы аргументы, даже несмотря на то, что в это же время некоторые другие параллельные процессы могут вызывать другие элементы массива, что приводит к чередованию сообщений типа

$$\textit{удв}.3.\textit{лев}.30, \ldots \textit{удв}.2.\textit{лев}.20, \ldots \textit{удв}.3.\textit{прав}.60, \ldots \textit{удв}.2.\textit{прав}.40, \ldots$$

Когда процесс вызывает повторно входимую подпрограмму, на самом деле не имеет значения, какой из элементов массива ответит на этот вызов; годится любой в данный момент свободный процесс. Поэтому вместо того, чтобы указывать конкретный индекс 2 или 3, вызывающий процесс должен делать произвольный выбор, используя конструкцию

$$\|_{i\geq 0} \left(\textit{удв}.i.\textit{лев}!30 \to \textit{удв}.i.\textit{прав}?\, y \to \textit{ПРОПУСК}\right)$$

При этом по-прежнему существенно требование, чтобы для передачи аргументов и (сразу после этого) получения результата использо-

вался один и тот же индекс.

Различают *локальные* вызовы процедуры, поскольку предполагается, что выполнение процедуры происходит на том же процессоре, что и выполнение вызывающего процесса, и *дистанционные* вызовы общей процедуры, когда предполагает исполнение на отдельном, возможно, удаленном процессоре. Так как эффект дистанционного и локального вызова должен быть одинаковым, причины для использования именно дистанционного вызова могут быть только организационными или экономическими. Например, для хранения кода процедуры в секрете или для исполнения его на машине, имеющей какие-то специальные средства, слишком дорогие для установки их на той машине, на которой исполняются процессы-пользователи.

Типичным примером таких дорогостоящих устройств могут служить внешние запоминающие устройства большой емкости - такие, как дисковая или доменная память.

Пример. Общая внешняя память.

Запоминающая среда разбита на B секторов, запись и чтение с которых могут производиться независимо. В каждом секторе может храниться один блок информации, которая поступает слева и выводится направо. Запоминающая среда реализована по технологии разрушающего считывания, так что каждый блок может быть считан только один раз. Дополнительная память в целом представляет собой массив таких секторов с индексами, не превосходящими B :

$$ДОППАМ = \|_{i<B}\, i : КОПИР .$$

Предполагается, что эта память используется как подчиненный процесс

$$(доп : ДОППАМ\,//...)$$

Внутри основного процесса доступ к этой памяти осуществляется взаимодействиями

$$доп.i.лев!блок \to ...доп.i.прав?y \to,\ldots$$

Дополнительная память может также совместно использоваться параллельными процессами. В этом случае действие

$$\|_{i<B} (доп.i.лев!блок \to \ldots)$$

одновременно займет произвольный свободный сектор с номером i и запишет в него значение *блок*. Аналогично, $доп.1.прав?x$ за одно действие считает содержимое сектора i в x и освободит этот сектор для дальнейшего использования, вполне вероятно, уже другим процессом. Именно это упрощение и послужило истинной причиной использования *КОПИР* для моделирования каждого сектора.

Успешное совместное использование этой дополнительной памяти требует от процессов-пользователей строжайшего соблюдения дисциплины. Процесс может совершить ввод информации с некоторого сектора, только если именно этот процесс последним выводил туда информацию, причем за каждым выводом рано или поздно должен последовать такой ввод. Несоблюдение этого порядка приводит к тупиковой ситуации или к еще худшим последствиям.

3.4.4 Планирование ресурсов

Когда ограниченное число ресурсов разделено между большим числом потенциальных пользователей, всегда существует возможность того, что некоторым пользователям, стремящимся занять ресурс, приходится ждать, пока его освободит другой процесс. Если к моменту освобождения ресурса его хотят занять два или более процесса, выбор того, который из ожидающих процессов получит ресурс, во всех приводившихся примерах был недетерминированным. Предположим, что к тому моменту, когда ресурс снова освободится, к множеству ожидающих присоединится еще один процесс. Поскольку выбор между ожидающими процессами по-прежнему недетерминирован, может случиться, что повезет именно вновь присоединившемуся процессу. Если ресурс сильно загружен, так может случиться снова и снова. В результате может оказаться, что некоторые процессы будут откладываться бесконечно или, по крайней мере, в течение полностью неопределенного времени.

Одним из решений проблемы является обеспечение того, чтобы все ресурсы были несильно загружены. Этого можно достигнуть либо введением дополнительных ресурсов, либо установлением высокой платы за предоставляемые услуги. Фактически это единственно приемлемые решения в случае постоянно загруженного ресурса. К сожалению, даже в среднем несильно загруженный ресурс достаточно часто оказывается сильно загруженным в течение длительных периодов (в часы пик). Иногда проблему удается сгладить введением дифференцированного тарифа в попытке регулирования спроса, но это не всегда помогает и даже не всегда удается. В течение таких пиков задержки процесса-пользователя в среднем неизбежны. Важно лишь следить за соразмерностью и предсказуемостью таких задержек.

Задача распределения ресурса между ожидающими пользователями известна как *планирование ресурсов.* Для успешного планирования необходимо знать, какие процессы в текущий момент ожидают получения ресурса. По этой причине получение ресурса нельзя рассматривать как одно элементарное событие. Его необходимо разбить на два

события:

пожалуйста, осуществляющее запрос ресурса,

спасибо, сопровождающее реальное получение ресурса.

Период между, *пожалуйста* и *спасибо*, для каждого процесса является временем, в течение которого он ждет ресурс. Чтобы различать ожидающие процессы, каждое вхождение события *пожалуйста, спасибо* и *свободен* помечено отличным от других натуральным индексом. При каждом запросе ресурса процесс получает номер с помощью конструкции:

$$\|_{i\geq 0} \left(рес.i.\ пожалуйста;\ рес.i.спасибо;...;\ рес.i.свободен \rightarrow ПРОПУСК \right)$$

Простым и эффективным способом планирования ресурса является назначение его процессу, ожидавшему дольше всех. Такая политика называется «*первым пришел - первым, обслужен*» (FCFS) или «*первым пришел - первым ушел*» (FIFO) и представляет собой принцип очереди, соблюдаемый, к примеру, пассажирами на автобусной остановке.

В заведении же типа поликлиники, где посетители не *могут;* или не хотят выстраиваться в очередь, для достижения того же результата действует другой механизм. Регистратура выдает талоны со строго возрастающими последовательными номерами. При входе в поликлинику посетитель берет талон. Когда врач освободился, он вызывает посетителя, имеющего талон с наименьшим номером, но еще не принятого. Этот алгоритм, называемый алгоритмом поликлиники. Будем предполагать, что одновременно могут обслуживаться до R посетителей,

Пример. Алгоритм поликлиники.

Введем три счетчика:

p - посетители, сказавшие *пожалуйста,*

t - посетители, сказавшие *спасибо,*

r - посетители, освободившие свои ресурсы.

Очевидно, что в любой момент времени $r \leq t \leq p$. Кроме того, p всегда будет номером, который получает очередной посетитель, приходящий в поликлинику, а t - номером очередного обслуживаемого посетителя; далее, $p-t$ будет числом ожидающих посетителей, а $R+r-t$ - числом ожидающих врачей. Вначале значения всех счетчиков равны нулю и могут быть вновь положены равными нулю в любой момент, когда их значения совпадают - например, вечером, после ухода последнего посетителя.

Одной из основных задач алгоритма является обеспечение того, чтобы никогда не было одновременно свободного ресурса и ждущего посетителя; как только возникает такая ситуация, следующим событием должно стать *спасибо* посетителя, получающего ресурс.

$$ПОЛИКЛИНИКА = B_{0,0,0}$$

$$B_{p,t,r} = \textbf{if } 0 < r = t = p \textbf{ then } ПОЛИКЛИНИКА$$
$$\textbf{else if } R + r - t > 0 \textbf{ AND } p - t > 0$$
$$\textbf{then } t.спасибо \rightarrow B_{p,t+1,r}$$
$$\textbf{else} \left(p.пожалуйста \rightarrow B_{p+1,t,r} \Big|\Big|_{i<t} \left(i.свободен \rightarrow B_{p,t,r+1} \right) \right)$$

3.5 Программирование параллельных вычислений

3.5.1 Основные понятия

Исполнение процессов параллельной программы прерывается значительно чаще, чем процессов, работающих в последовательной среде, так как процессы параллельной программы выполняют еще действия, связанные с обменом данными между процессорами. Манипулирование полновесными процессами в мультипрограммной среде является дорогостоящим действием, поскольку это тесно связанно с управлением и защитой памяти. Вследствие этого большинство параллельных компьютеров использует легковесные процессы, называемые *нитями* или *потоками* управления, а не полновесные процессы. Легковесные процессы не имеют собственных защищенных областей памяти (хотя могут обладать собственными локальными данными), а в результате очень сильно упрощается манипулирование ими. Более того, их использование более безопасно.

В соответствии с возможностями параллельного компьютера процессы взаимодействуют между собой обычно одним из следующих способов:

- *Обмен сообщениями*. Посылающий процесс формирует сообщение с заголовком, в котором указывает, какой процессор должен принять сообщение, и передает сообщение в сеть, соединяющую процессоры. Если, как только сообщение было передано в сеть, посылающий процесс продолжает работу, то такой вид отправки сообщения, называется неблокирующим. Если же посылающий процесс ждет, пока принимающий процесс не примет сообщение, то такой вид отправки сообщения, называется блокирующим. Принимающий процесс должен знать, что ему необходимы данные, и должен указать, что готов получить сообщение, выполнив соответствующую команду приема сообщения. Если ожидаемое сообщение еще не поступило, то принимающий процесс приостанавливается до тех пор, пока сообщение не поступит.
- *Обмен через общую память*. В архитектурах с общедоступной памятью процессы связываются между собой через общую память

- посылающий процесс помещает данные в известные ячейки памяти, из которых принимающий процесс может считывать их. При таком обмене сложность представляет процесс обнаружения того, когда безопасно помещать данные, а когда удалять их. Чаще всего для этого используются стандартные методы операционной системы, такие как *семафоры* или *блокировки* процессов. Однако это дорого и сильно усложняет программирование. Некоторые архитектуры предоставляют биты «*занято/свободно*», связанные с каждым словом общей памяти, что обеспечивает легким и высокоэффективный способ синхронизации отправителей и приемников.

– *Прямой доступ к удаленной памяти*. В первых архитектурах с распределенной памятью работа процессоров прерывалась каждый раз, когда поступал какой-нибудь запрос от сети, соединяющей процессоры. В результате процессор плохо использовался. Затем в таких архитектурах в каждом процессорном элемент стали использовать пары процессоров - один процессор (вычислительный), исполняет программу, а другой (процессор обработки сообщений) обслуживает сообщения, поступающие из сети или в сеть. Такая организация обмена сообщениями позволяет рассматривать обмен сообщениями как прямой доступ к удаленной памяти, к памяти других процессоров. Эта гибридная форма связи, применяется в архитектурах с распределенной памятью, обладает многими свойствами архитектурах с общей памятью.

Рассмотренные механизмы связи необязательно используются только непосредственно на соответствующих архитектурах. Так легко промоделировать обмен сообщениями, используя общую память, с другой стороны можно смоделировать общую память, используя обмен сообщениями. Последний подход известен как *виртуальная общая память*.

Обязательными признаками параллельных алгоритмов и программ являются:

– параллелизм,
– маштабируемость,
– локальность,
– модульность.

Параллелизм указывает на способность выполнения множества действий одновременно, что существенно для программ, выполняющихся на нескольких процессорах.

Маштабируемость - другой важнейший признак параллельной программы, который требует гибкости программы по отношению к изменению числа процессоров, поскольку наиболее вероятно, что их число будет постоянно увеличиваться в большинстве параллельных

сред и систем.

Локальность характеризует необходимость того, чтобы доступ к локальным данным был более частым, чем доступ к удаленным данным. Важность этого свойства определяется отношением стоимостей удаленного и локального обращений к памяти. Оно является ключом к повышению эффективности программ на архитектурах с распределенной памятью.

Модульность отражает степень разложения сложных объектов на более простые компоненты. В параллельных вычислениях это такой же важный аспект разработки программ, как и в последовательных вычислениях.

Код, исполняющийся в одиночном процессоре параллельного компьютера, находится в некоторой программной среде такой же, что и среда однопроцессорного компьютера с мультипрограммной операционной системой, поэтому и в контексте параллельного компьютера так же говорят о *процессах*, ссылаясь на код, выполняющийся внутри защищенного региона памяти операционной системы. Многие из действий параллельной программы включают обращения к удаленным процессорам или ячейкам общей памяти. Выполнение этих действий может потребовать время, существенное, особенно, по отношению к времени исполнения обычных команд процессора. Поэтому большинство процессоров исполняет более одного процесса одновременно, и, следовательно, в программной среде отдельно взятого процессора параллельного компьютера применимы обычные методы мультипрограммирования.

3.5.2 Многопоточная обработка

Если L - метка некоторого места в программе, то команда

fork L

передает управление на метку L, а также и на следующую команду в тексте программы. В результате создается эффект, что с этого момента два процессора одновременно исполняют одну и ту же программу; каждый из них независимо обрабатывает свою последовательность команд. Поскольку каждая такая последовательность обработки может снова разветвиться, эта техника получила название *многопоточной обработки.*

Введя способ разбиения одного процесса на два, мы нуждаемся и в способе слияния двух процессов в один. Проще всего ввести команду **join,** которая может выполниться только при одновременном исполнении ее *двумя* процессами. Первый достигший этой команды процесс должен ждать, когда ее достигнет другой. После этого уже только

один процесс продолжает исполнение последующих команд.

Разновидность команды ветвления до сих пор используется в операционной системе UNIX™. При этом ветвление не подразумевает переход по метке. Его эффект заключается во взятии совершенно новой копии всей памяти программы и передачи этой копии новому процессу. Как исходный, так и новый процессы продолжают исполнение с команды, следующей за командой ветвления. У каждого процесса есть средство определить, является ли он *порождающим* (отец) или *порождаемым* (сын). Выделение процессам непересекающихся участков памяти снимает основные трудности и опасности многопоточной обработки, но может быть неэффективным как по времени, так и по объему памяти. Это означает, что параллелизм допустим только на самом внешнем (самом глобальном) уровне задания, а использование его в мелком масштабе затруднительно.

3.5.3 Условные критические участки

Предположим, например, что один процесс изменяет некоторую переменную с целью, чтобы другой процесс считывал ее новое значение. Второй процесс не должен считывать значения переменной до тех пор, пока оно не будет изменено. Аналогично, первый процесс не должен изменять значение переменной до тех пор, пока все остальные процессы не считают ее предыдущие значения.

Для решения этой проблемы предложено удобное средство, называемое *условным критическим участком.* Он имеет вид

`with` *общперем* **`when`** *условие* **`do`** *критический участок*

При входе в критический участок проверяется значение условия. Если оно истинно, критический участок исполняется как обычно, но если условие ложно, данный вход в критический участок задерживается, чтобы позволить другим процессам войти в свои критические участки и изменить общую переменную. По завершении каждого такого изменения происходит перепроверка условия. Если оно стало истинным, отложенному процессу позволяют продолжать исполнение своего критического участка; в противном случае процесс вновь откладывается. Если можно запустить более чем один из приостановленных процессов, выбор между ними произвольный.

3.5.4. Мониторы

Своим возникновением и развитием мониторы обязаны понятию класса. Основной идеей является то, что все осмысленные операции над данными (включая их инициализацию) должны быть собраны вместе с описанием структуры и типа самих данных; активизация этих

операций должна происходить при вызове процедуры всякий раз, когда этого требуют процессы, совместно использующие данные. Важной характеристикой монитора является то, что одновременно может быть активным только одно из его процедурных тел; даже когда два процесса одновременно делают вызов процедуры (одной и той же или двух различных), один из вызовов («ждет») откладывается до завершения другого. Таким образом, тела процедур ведут себя как критические участки, защищенные одним и тем же семафором.

Приведем пример очень простого монитора, ведущего себя как счетчиковая переменная.

```
1 monitor счет;
2 var n: integer
3 procedure* вверх; begin n := n + 1 end;
4 procedure* вниз; when > 0 do begin n := n - 1 end;
5 function* приземл. Boolean; begin приземл := (n = 0) end;
6 begin n := 0;
7  ...;
8  if n ≠ 1 then print(n)
9 end
```

Строка 1 описывает монитор с именем *счет.*

2 описывает локальную для монитора общую переменную *n.* Она доступна только внутри самого монитора.

3 - 5 описывают три процедуры и их тела. Звездочки обеспечивают вызов этих процедур из программы, использующей монитор.

6 Здесь начинается исполнение монитора.

7 Три жирные точки обозначают *внутреннее* предложение, соответствующее блоку, который будет использовать монитор.

8 При выходе из использующего блока печатается конечное значение *n* (если оно ненулевое).

Новый экземпляр этого монитора может быть описан локальным для блока P :

instance *ракета: счет; P*

Внутри блока P можно вызывать помеченные звездочками процедуры, используя команды

ракета.вверх;... ракета.вниз;. ..;, **if** *ракета.приземл* **then...**

Непомеченная же процедура или такая переменная, как n, недостижимы из P. Свойственное мониторам взаимное исключение позволяет вызывать процедуру монитора любому числу процессов внутри P без взаимного влияния при изменении n. Заметим, что попытка вызвать *ракета.вниз* при $n = 0$ будет отложена до тех пор, пока неко-

торый другой процесс внутри *P* не вызовет *ракета.вверх*. Это гарантирует, что значение *n* никогда не станет отрицательным.

Неэффективность повторяющейся проверки входных условий привела к появлению мониторов с более сложной схемой явного ожидания и явной подачей сигнала о возобновлении ожидающего процесса. Эти схемы даже позволяют приостанавливаться процедурному вызову в процессе его исполнения под воздействием автоматически возникающего исключения до того момента, когда некоторый последующий вызов процедуры другим процессом подаст сигнал о возобновлении приостановленного процесса. Таким путем можно эффективно реализовать множество оригинальных способов планирования, которые реализуются программой-планировщиком.

Так как в мониторе ожидающих процессов может быть несколько, простой порядок обслуживания очереди, состоящий в том, что освобождается процесс, ожидающий дольше всех, гарантирует, что ни один из обратившихся процессов не будет задержан на неопределенное время.

3.6 Модели параллельных вычислений

Параллельное программирование представляет дополнительные источники сложности - необходимо явно управлять работой тысяч процессоров, координировать миллионы межпроцессорных взаимодействий. Для того решить задачу на параллельном компьютере, необходимо распределить вычисления между процессорами системы, так чтобы каждый процессор был занят решением части задачи. Кроме того, желательно, чтобы как можно меньший объем данных пересылался между процессорами, поскольку коммуникации значительно больше медленные операции, чем вычисления. Часто, возникают конфликты между степенью распараллеливания и объемом коммуникаций, то есть чем между большим числом процессоров распределена задача, тем больший объем данных необходимо пересылать между ними. Среда параллельного программирования должна обеспечивать адекватное управление распределением и коммуникациями данных.

Из-за сложности параллельных компьютеров и их существенного отличия от традиционных однопроцессорных компьютеров нельзя просто воспользоваться традиционными языками программирования и ожидать получения хорошей производительности. Рассмотрим основные модели параллельного программирования, их абстракции адекватные и полезные в параллельном программировании:

3.6.1 Процесс/канал

В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять. Модель характеризуется следующими свойствами:

1) Параллельное вычисление состоит из одного или более одновременно исполняющихся процессов, число которых может изменяться в течение времени выполнения программы.
2) Процесс - это последовательная программа с локальными данными. Процесс имеет входные и выходные порты, которые служит интерфейсом к среде процесса.
3) В дополнение к обычным операциям процесс может выполнять следующие действия: послать сообщение через выходной порт, получить сообщение из входного порта, создать новый процесс и завершить процесс.
4) Посылающаяся операция асинхронная - она завершается сразу, не ожидая того, когда данные будут получены. Получающаяся операция синхронная: она блокирует процесс до момента поступления сообщения.
5) Пары из входного и выходного портов соединяются очередями сообщений, называемыми каналами. Каналы можно создавать и удалять. Ссылки на каналы (порты) можно включать в сообщения, так что связность может измениться динамически.
6) Процессы можно распределять по физическим процессорам произвольным способами, причем используемое отображение (распределение) не воздействует на семантику программы. В частности, множество процессов можно отобразить на одиночный процессор.

Понятие процесса позволяет говорить о местоположении данных: данные, содержащихся в локальной памяти процесса - расположены «близко», другие данные «удалены». Понятие канала обеспечивает механизм для указания того, что для того, чтобы продолжить вычисление одному процессу требуются данные другого процесса (зависимость по данным).

3.6.2 Обмен сообщениями

В этой модели программы, возможно различные, написанные на традиционном последовательном языке исполняются процессорами

компьютера. Каждая программа имеют доступ к памяти исполняющего ее процессора. Программы обмениваются между собой данными, используя подпрограммы приема/передачи данных некоторой коммуникационной системы.

На сегодняшний день модель обмен сообщениями является наиболее широко используемой моделью параллельного программирования. Программы этой модели, подобно программам модели процесс/канал, создают множество процессов, с каждым из которых ассоциированы локальные данные. Каждый процесс идентифицируется уникальным именем. Процессы взаимодействуют, посылая и получая сообщения. В этом отношение модель обмен сообщениями является разновидностью модели процесс/канал и отличается только механизмом, используемым при передаче данных.

Модель не накладывает ограничений ни на динамическое создание процессов, ни на выполнение нескольких процессов одним процессором, ни на использование разных программ для разных процессов. Просто, формальные описания систем обмена сообщениями не рассматривают вопросы, связанные с манипулированием процессами, Однако, при реализации таких систем приходится принимать какое-либо решение в этом отношении. На практике сложилось так, что большинство систем обмена сообщениями при запуске параллельной программы создает фиксированное число идентичных процессов и не позволяет создавать и разрушать процессы в течение работы программы.

В таких системах каждый процесс выполняет одну и ту же программу (параметризованную относительно идентификатора либо процесса, либо процессора), но работает с разными данными.

3.6.3 Параллелизм данных

В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Распараллеливание операций над массивами, циклов обработки массивов позволяет увеличить производительность программы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти. Программы с параллелизмом данных могут быть оттранслированы и исполнены как на MIMD, так и на SIMD компьютерах.

Модель также является часто используемой моделью параллельного программирования. Название модели происходит оттого, что она эксплуатирует параллелизм, который заключается в применении одной и той же операции к множеству элементов структур данных. Например, «умножить все элементы массива M на значение x», или «снизить цену автомобилей со сроком эксплуатации более 5-ти лет». Программа с параллелизмом данных состоит из последовательностей подобных операций. Поскольку операции над каждым элементом данных можно рассматривать как независимые процессы, то степень детализации таких вычислений очень велика, а понятие «локальности» (распределения по процессам) данных отсутствует. Следовательно, компиляторы языков с параллелизмом данных часто требуют, чтобы программист предоставил информацию относительно того, как данные должны быть распределены между процессорами, другими словами, как программа должны быть разбита на процессы.

3.6.4 Общей памяти

В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти как с запросами на чтение, так и с запросами на запись, что создает проблемы при выборе момента, когда можно будет поместить данные в память, когда можно будет удалить их. Для управления доступом к общей памяти используются стандартные механизмы синхронизации - семафоры и блокировки процессов.

В модели все процессы совместно используют общее адресное пространство, к которому они асинхронно обращаются с запросами на чтение и запись. В таких моделях для управления доступом к общей памяти используются всевозможные механизмы синхронизации типа семафоров и блокировок процессов. Преимущество этой модели с точки зрения программирования состоит в том, что понятие собственности данных (монопольного владения данными) отсутствует, следовательно, не нужно явно задавать обмен данными между производителями и потребителями. Эта модель, с одной стороны, упрощает разработку программы, но, с другой стороны, затрудняет понимание и управление локальностью данных, написание детерминированных программ. В основном эта модель используется при программировании для архитектур с общедоступной памятью.

Контрольные вопросы

4 Моделирование взаимодействия процессов. Сети Петри

4.1 Введение в сети Петри

Сети Петри это инструмент для математического моделирования и исследования сложных систем. Цель представления системы в виде сети Петри и последующего анализа этой сети состоит в получении важной информации о структуре и динамическом поведении моделируемой системы. Эта информация может использоваться для оценки моделируемой системы и выработки предложений по ее усовершенствованию. Впервые сети Петри предложил немецкий математик Карл Адам Петри.

Сети Петри предназначены для моделирования систем, которые состоят из множества взаимодействующих друг с другом *компонент.* При этом компонента сама может быть системой. Действиям различных компонент системы присущ параллелизм. Примерами таких систем могут служить вычислительные системы, в том числе и параллельные, компьютерные сети, программные системы, обеспечивающие их функционирование, а также экономические системы, системы управления дорожным движением, химические системы, и т. д.

В одном из подходов к проектированию и анализу систем сети Петри используются, как вспомогательный инструмент анализа. Здесь для построения системы используются общепринятые методы проектирования. Затем построенная система моделируется сетью Петри, и модель анализируется. Если в ходе анализа в проекте найдены изъяны, то с целью их устранения проект модифицируется. Модифицированный проект затем снова моделируется и анализируется. Этот цикл повторяется до тех пор, пока проводимый анализ не приведет к успеху.

Другой подход предполагает построение проекта сразу в виде сети Петри. Методы анализа применяются только для создания проекта, не содержащего ошибок. Затем сеть Петри преобразуется в реальную рабочую систему.

В *первом* случае необходима разработка методов моделирования систем сетями Петри, а во *втором* случае должны быть разработаны методы реализации сетей Петри системами.

4.2 Основные определения

4.2.1 Теоретико-множественное определение сетей Петри

Пусть *мультимножество* это множество, допускающее вхождение нескольких экземпляров одного и того же элемента.

Сеть Петри N является четверкой $N = (P, T, I, O)$,

где $P = \{p_1, p_2, \ldots, p_n\}$ - конечное множество *позиций*, $n \geq 0$;

$T = \{t_1, t_2, \ldots, t_m\}$ - конечное множество *переходов*, $m \geq 0$;

$I : T \to P^*$ - входная функция, сопоставляющая переходу мультимножество его входных позиций;

$O : T \to P^*$ - выходная функция, сопоставляющая переходу мультимножество его выходных позиций.

Позиция $p \in P$ называется *входом* для перехода $t \in T$, если $p \in I(t)$. Позиция $p \in P$ называется *выходом* для перехода $t \in T$, если $p \in O(t)$. *Структура* сети Петри определяется ее позициями, переходами, входной и выходной функциями.

Пример. Сеть Петри $N = (P, T, I, O)$,

$$P = \{p_1, p_2, p_3\},$$
$$T = \{t_1, t_2\},$$
$$I(t_1) = \{p_1, p_1, p_2\}, O(t_1) = \{p_3\},$$
$$I(t_2) = \{p_1, p_2, p_2\}, O(t_2) = \{p_3\}.$$

Использование мультимножеств входных и выходных позиций перехода, а не множеств, позволяет позиции быть *кратным входом* и *кратным выходом* перехода соответственно. При этом *кратность* определяется числом экземпляров позиции в соответствующем мультимножестве.

4.2.2 Графы сетей Петри

Наглядным представлением сети Петри является её графическое представление, которое представляет собой двудольный, ориентированный мультиграф.

Граф сети Петри обладает двумя типами узлов: *кружок* μ, представляющий позицию сети Петри; и *планка***,** представляющая переход сети Петри. Ориентированные дуги этого графа (стрелки) соединяют переход с его входными и выходными позициями. При этом дуги направлены от входных позиций к переходу и от перехода к выходным позициям. Кратным входным и выходным позициям перехода соответствуют кратные входные и выходные дуги. Граф сети Петри рассмотренного примераприведен на рисунке 4.1.

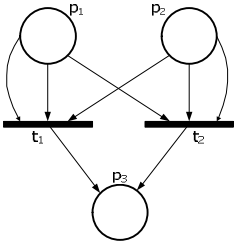


Рис. 4.1. Граф сети Петри.

В графе сети Петри не возможны дуги между двумя позициями и между двумя переходами.

4.2.3 Маркировка сетей Петри

Маркировка - это размещение по позициям сети Петри *фишек*, изображаемых на графе сети Петри точками. Фишки используются для определения выполнения сети Петри. Количество фишек в позиции при выполнении сети Петри может изменяться от 0 до бесконечности.

Маркировка μ сети Петри $N = (P, T, I, O)$ есть функция, отображающая множество позиций P во множество Nat неотрицательных целых чисел. Маркировка μ, может быть также определена как n — вектор $\mu = \langle \mu(p_1), \mu(p_2), \ldots, \mu(p_n) \rangle$, где n – число позиций в сети Петри и для каждого $1 \le i \le n \quad \mu(p_i) \in Nat$ – количество фишек в позиции p_i.

Маркированная сеть Петри $N = (P, T, I, O, \mu)$ определяется совокупностью структуры сети Петри $N = (P, T, I, O)$ и маркировки μ. На рисунке 4.2 представлена маркированная сеть Петри $\mu = \langle 1, 01 \rangle$.

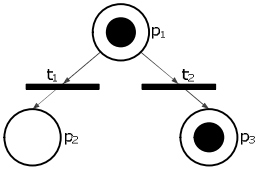


Рис. 4.2. Маркированная сеть Петри.

Множество всех маркировок сети Петри бесконечно. Если фишек, помещаемых в позицию слишком много, то удобнее не рисовать фишки в кружке этой позиции, а указывать их количество.

4.2.4 Правила выполнения сетей Петри

Сеть Петри *выполняется* посредством *запусков* переходов. *Запуск* перехода управляется фишками в его входных позициях и сопровождается удалением фишек из этих позиций и добавлением новых фишек в его выходные позиции.

Переход может запускаться только в том случае, когда он разрешен. Переход называется *разрешенным*, если каждая из его входных позиций содержит число фишек, не меньшее, чем число дуг, ведущих из этой позиции в переход (или кратности входной дуги).

Пусть функция $^\#: P \times T \to Nat$ для произвольных позиции $p \in P$ и перехода $t \in T$ задает значение $^\#(p,t)$, которое совпадает с кратностью дуги, ведущей из p в t, если такая дуга существует, и с нулем, в противном случае.

Пусть функция $\#^: T \times P \to Nat$ для произвольных и перехода $t \in T$ позиции $p \in P$ задает значение $\#^(t,p)$, которое совпадает с кратностью дуги, ведущей из t в p, если такая дуга существует, и с нулем, в противном случае.

Переход $t \in T$ в маркированной сети Петри $N = (P,T,I,O,\mu)$ разрешен, если для всех $p \in I(t)$ справедливо $\mu(p) \geq ^\#(p,t)$.

Запуск разрешённого перехода $t \in T$ из своей входной позиции $p \in I(t)$ удаляет $^\#(p,t)$ фишек, а в свою выходную позицию $p' \in O(t)$ добавляет $\#^(t,p')$ фишек.

Сеть Петри до запуска перехода t_1 (рис. 4.3, а). Сеть Петри после запуска перехода t_1 (рис. 4.3, б).

а *б*

Рис. 4.3.Запуск сети Петри

Переход t в маркированной сети Петри с маркировкой μ может быть запущен всякий раз, когда он разрешен и в результате этого запуска образуется новая маркировка μ', определяемая для всех $p \in P$ следующим соотношением:

$$\mu'(p) = \mu(p) - ^\#(p,t) + \#^(t,p).$$

Запуски могут осуществляться до тех пор, пока существует хотя бы один разрешенный переход. Когда не останется ни одного разрешенного перехода, выполнение *прекращается.*

Если запуск произвольного перехода t преобразует маркировку μ сети Петри в новую маркировку μ', то будем говорить, что μ' достижима из μ посредством запуска перехода t и обозначать этот факт, как $\mu \to^t \mu'$. Это понятие очевидным образом обобщается для случая последовательности запусков разрешённых переходов. Через $R(N,\mu)$ обозначим множество всех достижимых маркировок из начальной маркировки μ в сети Петри N.

Преобразование маркировки сети Петри изображено на рисунке 4.3. Переход t_1 преобразует маркировку $\mu = \langle 5,1 \rangle$ в маркировку $\mu' = \langle 2,3 \rangle$.

4.3 Моделирование систем на основе сетей Петри

4.3.1 События и условия

Представление системы сетью Петри основано на двух основополагающих понятиях: *событиях* и *условиях.* Возникновением событий управляет состояние системы, которое может быть описано множеством условий. Условие может принимать либо значение «истина», либо значение «ложь».

Возникновение события в системе возможно, если выполняются определённые условия – *предусловия* события. Возникновение события может привести к выполнению других условий – *постусловий* события. В качестве примера рассмотрим следующую ниже задачу моделирования.

Пример. Моделирование последовательной обработки запросов сервером базы данных. Сервер находится в состоянии ожидания до тех пор, пока от пользователя не поступит запрос клиента, который он обрабатывает и отправляет результат такой обработки пользователю.

Условиями для рассматриваемой системы являются:

а) сервер ждет;

б) запрос поступил и ждет;

в) сервер обрабатывает запрос;

г) запрос обработан.

Событиями для этой системы являются:

1) Запрос поступил.
2) Сервер начинает обработку запроса.
3) Сервер заканчивает обработку запроса.
4) Результат обработки отправляется клиенту.

Для перечисленных событий можно составить следующую таблицу их пред- и постусловий

Таблица 4.1. Условия и события системы

Событие	Предусловия	Постусловия
1	нет	б
2	а, б	в
3	в	г, а
4	г	нет

Такое представление системы легко моделировать сетью Петри. В сети Петри условия моделируются позициями, события - переходами. При этом входы перехода являются предусловиями соответствующего события; выходы — постусловиями. Возникновение события моделируется запуском соответствующего перехода. Выполнение условия представляется фишкой в позиции, соответствующей этому условию. Запуск перехода удаляет фишки, представляющие выполнение предусловий и образует новые фишки, которые представляют выполнение постусловий.

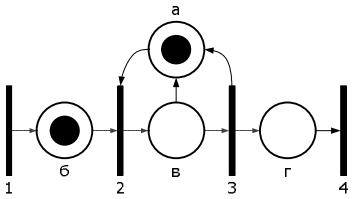


Рис. 4.4. Сеть Петри сервера базы данных

На рисунке 4.4. предусловие выполняется для события 2.

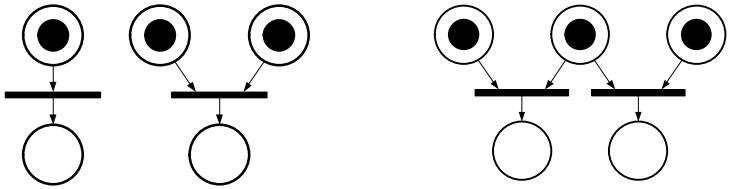
4.3.2 Одновременность и конфликт

Особенность сетей Петри - их *асинхронная* природа. В сетях Петри отсутствует измерение времени. В них учитывается лишь важнейшее свойство времени – частичное упорядочение событий.

Выполнение сети Петри (или поведение моделируемой системы) рассматривается здесь как *последовательность* дискретных событий, которая является одной из возможных. Если в какой-то момент времени разрешено более одного перехода, то любой из них может стать «следующим» запускаемым. Переходы в сети Петри, моделирующей некоторую систему, представляют ее *примитивные* события (длитель-

ность которых считается равной 0), и в один момент времени может быть запущен только один разрешённый переход.

Моделирование одновременного (параллельного) возникновения независимых событий системы в сети Петри демонстрируется на рисунке 4.5 а.



а *б*

Рис. 4.5. Сети Петри для моделирования параллельных процессов

В этой ситуации два перехода являются разрешенными и не влияют друг на друга в том смысле, что могут быть запущены один вслед за другим в любом порядке.

Другая ситуация в приведенной на рисунке 4.5 б сети Петри. Эти два разрешённые перехода находятся в *конфликте*, т. е. запуск одного из них удаляет фишку из общей входной позиции и тем самым запрещает запуск другого. Таким образом, моделируются взаимоисключающие события системы.

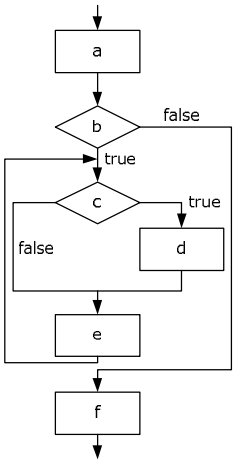
4.4 Моделирование параллельных систем взаимодействующих процессов

4.4.1. Моделирование последовательных процессов

Вырожденным случаем параллельной системы процессов является система с одним процессом. Сначала рассмотрим, как сетью Петри может быть представлен отдельный процесс. Отдельный процесс описывается программой на одном из существующих языков программирования.

Пример. Последовательная программа на абстрактном языке программирования, вычисляющая $Y!$ и произведение всех чётных чисел из отрезка $[1, Y]$ для $Y \in Nat$.

```
begin
 read(Y);
 X1:=1;
 X2:=1;
 while Y>0 do
  begin
    if mod(Y,2)=0
      then begin
           X1:=X1*Y;
           end;
     X2:=X2*Y;
     Y:=Y-1;
  end;
 write(X1);
 write(X2);
end
```



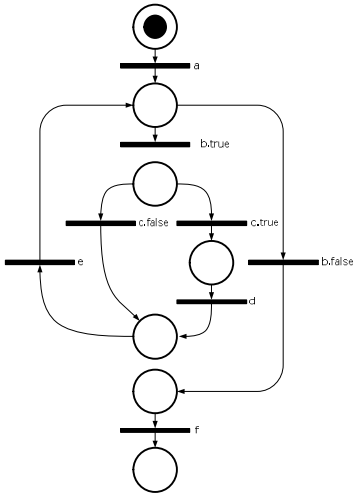
а *б*

Рис. 4.6. Программа и блок схема

Программа представляет два различных аспекта процесса: вычисление и управление. Сети Петри удачно представляют структуру и управление программ. Они предназначены для моделирования упорядочения действий и потока информации, а не для действительного вычисления самих значений.

Стандартный способ представления структуры программы и потока управления в ней - это *блок-схемы*, которые в свою очередь могут быть представлены сетями Петри. Блок-схема программы состоит из узлов двух типов (принятия решения, обозначаемых ромбами, и вычисления, обозначаемых прямоугольниками) и дуг между ними. Блок-схема программы изображена на рисунке 4.6 а, где блоки:

```
a) read(Y);X1:=1; X2:=1;
b) Y>0
c) mod(Y,2)=0
d) X1:=X1*Y;
e) X2:=X2*Y; Y:=Y-1;
f) write(X1); write(X2);
```



а *б*

Рис. 4.7. Сеть Петри программы

В сети Петри (рисунок 4.7 б), моделирующей блок-схему, узлы блок-схемы представляются переходами сети Петри, а дуги блок-схемы - позициями сети Петри. Фишка в сети Петри представляет счетчик команд блок-схемы.

4.4.2 Моделирование взаимодействия процессов

Параллельная система может строиться несколькими способами. Один из способов состоит в простом объединении процессов, без взаимодействия во время их одновременного выполнения. Так, например, если система строится этим способом из двух процессов, каждый из которых может быть представлен сетью Петри, то сеть Петри, моделирующая одновременное выполнение двух процессов, является простым объединением сетей Петри для каждого из двух процессов. Начальная маркировка составной сети Петри имеет две фишки, по одной в каждой сети, представляя первоначальный счетчик команд процесса.

Такой способ введения параллелизма имеет низкое практическое значение. Далее будем рассматривать параллельные системы процессов, допускающие взаимодействие процессов во время их параллельного выполнения.

Существуют различные виды взаимодействия (синхронизации) процессов, в том числе: взаимодействие посредством общей памяти; - посредством передачи сообщения различных видов.

Таким образом, для моделирования сетями Петри параллельных систем процессов, помимо последовательных процессов, необходимо уметь моделировать различные механизмы взаимодействия (синхронизации) процессов.

4.4.3 Задача о взаимном исключении

Пусть несколько процессов разделяют общую переменную, запись, файл или другой элемент данных. Для обновления разделяемого элемента данных процесс должен сначала считать старое значение, затем вычислить новое и, наконец, записать его на то же место. Если два процесса P_1 и P_2 в одно и то же время пытаются выполнить такую последовательность действий, то могут возникнуть искажение данных. Например, возможна следующая последовательность:

1) Процесс P_1 считывает значение x из разделяемого объекта;
2) Процесс P_2 считывает значение x из разделяемого объекта;
3) Процесс P_1 вычисляет новое значение $x' = f(x)$;
4) Процесс P_2 вычисляет новое значение $x'' = g(x)$;
5) Процесс P_1 записывает x' в разделяемый объект;
6) Процесс P_2 записывает x'' в разделяемый объект, уничтожая значение x.

Результат вычисления процесса P_1 потерян.

Для исключения подобных проблем используется метод взаимного исключения, основанный на понятии *критическая секция*. Критическая секция – это участок кода процесса, на котором он осуществляет доступ к разделяемому объекту данных. Прежде, чем выполнить свою критическую секцию, процесс ждёт, пока другой процесс не закончит выполнение собственной критической секции (если такое выполнение имеет место). Затем он входит в критическую секцию и блокирует доступ для любого другого процесса к своей критической секции. После выполнения процессом критической секции деблокируется доступ для других процессов к разделяемому объекту данных.

Сеть Петри (рисунок 4.8) моделирует механизм взаимного исключения для двух процессов, P_1 и P_2. Она легко обобщается на произвольное число процессов.

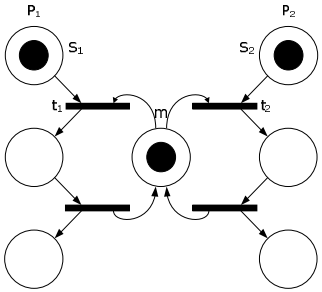


Рис. 4.8. Механизм взаимного исключения для двух процессов

Позиция m представляет условие «критическая секция свободна», разрешающее вход в критическую секцию. Попытка процесса $P_1\,(P_2)$ войти в критическую секцию осуществляется после помещения фишки в его позицию $s_1\,(s_2)$. Такая попытка может увенчаться успехом, если в позиции m содержится фишка. Если оба процесса пытаются войти в критическую секцию одновременно, то переходы t_1 и t_2 вступят в конфликт, и только один из них сможет запуститься. Запуск t_1 запретит запуск перехода t_2, вынуждая процесс P_2 ждать, пока процесс P_1 выйдет из своей критической секции, и возвратит фишку обратно в позицию m.

4.4.4 Задача о производителе/потребителе

В задаче о производителе/потребителе также присутствует разделяемый объект – буфер, посредством которого реализуется взаимодействие через асинхронную передачу сообщений. Процесс-производитель создает сообщения, которые помещаются в буфер. Потребитель ждет, пока сообщение не будет помещено в буфер, извлекает его оттуда и использует. Такое взаимодействие может быть промоделировано сетью Петри, изображенной на рисунке 4.9.

Позиция B представляет буфер, каждая фишка соответствует сообщению, которое произведено, но еще не использовано.

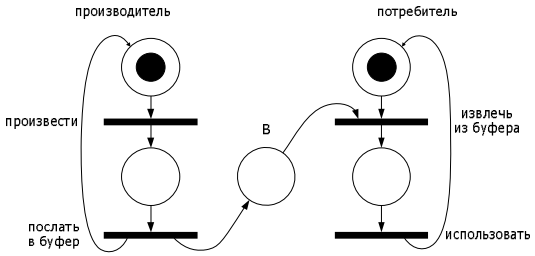


Рис. 4.9. Сеть Петри для задачи «производитель/потребитель»

4.4.5 Задача об обедающих философах

В задаче об обедающих философах возможна ситуация, в которой каждый философ возьмет вилку слева, а затем будет ждать, когда освободится вилка с правой стороны. Так они будут ждать, пока не умрут от голода. Тем самым, это состояние системы «обедающие философы» является *тупиковым*.

Проблема тупика в этой системе может быть решена путем следующей модификации ее правил поведения. Пусть философ при переходе из состояния размышления в состояние приема пищи берет одновременно обе вилки (слева и справа), если они свободны. Сеть Петри (рисунок 4.10) моделирует такую модифицированную систему обедающих философов, свободную от тупиков.

В этой сети Петри позиция п_i, $i \in \{1,2,3,4,5\}$, представляет условие «i — тая вилка свободна». В начальной маркировке каждая из этих позиций имеет фишку. Каждому философу $i \in \{1,2,3,4,5\}$ соответствует две позиции: позиция д_i – представляющая условие «i-тый философ думает»; и позиция е_i – представляющая условие «i-тый философ ест». В начальной маркировке все позиции д_i содержат фишку, а все позиции е_i пусты.

Каждому философу $i \in \{1,2,3,4,5\}$ также соответствует два перехода: переход нач_i – представляющий событие «начало приема пищи i-тым философом»; и переход зав_i – представляющий событие «завершение приема пищи i-тым философом».

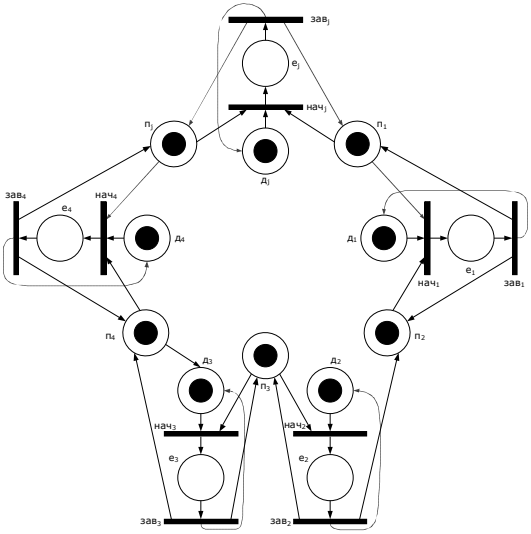


Рис. 4.10. Задача об обедающих философах

4.5 Анализ сетей Петри

Моделирование систем сетями Петри, прежде всего, обусловлено необходимостью проведения глубокого исследования их поведения. Для проведения такого исследования необходимы методы анализа свойств самих сетей Петри. Этот подход предполагает сведения исследования свойств реальной системы к анализу определенных свойств моделирующей сети Петри.

4.5.1 Свойства сетей Петри

Позиция $p \in P$ сети Петри $N = (P, T, I, O)$ с начальной маркировкой μ является k-ограниченной, если $\mu'(p) \le k$ для любой достижимой маркировки $\mu' \in R(N, \mu)$. Позиция называется ограниченной, если она является k-ограниченной для некоторого целого значения k. Сеть Петри ограниченна, если все ее позиции ограниченны.

Позиция $p \in P$ сети Петри $N = (P, T, I, O)$ с начальной маркировкой μ является *безопасной*, если она является 1-ограниченной. Сеть Петри *безопасна,* если безопасны все позиции сети.

Сеть Петри $N=(P,T,I,O)$ с начальной маркировкой μ является *сохраняющей*, если для любой достижимой маркировки $\mu' \in R(N,\mu)$ справедливо следующее равенство.

$$\sum_{p\in P}\mu'(p)=\sum_{p\in P}\mu(p)\mu'(p).$$

Тупик в сети Петри – один или множество переходов, которые не могут быть запущены. Определим для сети Петри N с начальной маркировкой μ следующие *уровни* активности переходов:

Уровень 0*:* Переход t обладает *активностью уровня* 0 и называется *мёртвым*, если он никогда не может быть запущен.

Уровень 1*:* Переход t обладает *активностью уровня* 1 и называется *потенциально живым,* если существует такая $\mu' \in R(N,\mu)$, что t разрешён в μ'.

Уровень 2*:* Переход t, обладает *активностью уровня* 2 и называется *живым,* если для всякой $\mu' \in R(N,\mu)$ переход t является потенциально живым для сети Петри N с начальной маркировкой μ'.

Сеть Петри называется *живой*, если все её переходы являются живыми.

Задача достижимости: Для данной сети Петри с маркировкой μ и маркировки μ' определить: $\mu' \in R(N,\mu)$?

Задача покрываемости: Для данной сети Петри N с начальной маркировкой μ и маркировки μ' определить, существует ли такая достижимая маркировка $\mu'' \in R(N,\mu)$, что $\mu'' \geq \mu'$.

(Отношение $\mu'' \geq \mu'$ истинно, если каждый элемент маркировки μ'' не меньше соответствующего элемента маркировки μ'.)

Сети Петри присуще некоторое поведение, которое определяется множеством ее возможных последовательностей запусков переходов или ее множеством достижимых маркировок. Понятие эквивалентности сетей Петри определяется через равенство множеств достижимых маркировок.

Сеть Петри $N=(P,T,I,O)$ с начальной маркировкой μ и сеть Петри $N'=(P',T',I',O')$ с начальной маркировкой μ' эквивалентны, если справедливо $R(N,\mu)=R(N',\mu')$.

Понятие эквивалентности сетей Петри может быть определено также через равенство множеств возможных последовательностей запусков переходов.

Более слабым, по сравнению с эквивалентностью, является свойство *включения*, определение которого совпадает с определением эквивалентности, с точностью до замены = на $\subseteq$.

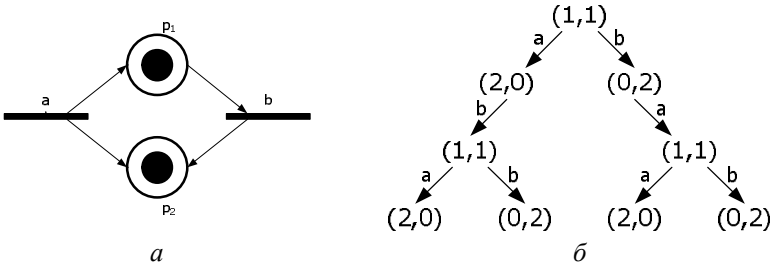
4.5.2 Методы анализа.

Особый интерес вызывают методы анализа свойств сетей Петри, которые обеспечивают автоматический анализ моделируемых систем. Сначала рассмотрим метод анализа сетей Петри, который основан на использовании *дерева достижимости*.

Дерево достижимости

Дерево достижимости представляет все достижимые маркировки сети Петри, а также – все возможные последовательности запусков ее переходов.

Пример. Частичное дерево достижимости маркированной сети Петри изображено на рисунке 4.11, а, а частичное дерево достижимости для трёх шагов построения имеет вид (рисунок 4.11, б).



а *б*

Рис. 4.11. Частное дерево достижимости для маркированной сети Петри

Для сети Петри с бесконечным множеством достижимых маркировок дерево достижимости является бесконечным. Сеть Петри с конечным множеством достижимых маркировок также может иметь бесконечное дерево достижимости. Для превращения бесконечного дерева в полезный инструмент анализа строится его конечное представление. При построении конечного дерева достижимости для обозначения бесконечного множества значений маркировки позиции используется символ ω. Также используются следующие ниже операции над ω, определяемые для любого постоянного a.

$$\omega - a \quad \omega, = \omega + a \quad \omega, = a < \omega, \quad \omega \le \omega .$$

Алгоритм построения конечного дерева достижимости. Каждая вершина дерева достижимости классифицируется алгоритмом или как *граничная* вершина, *терминальная* вершина, *дублирующая* вершина, или как *внутренняя* вершина. Алгоритм начинает работу с определения начальной маркировки корнем дерева, и *граничной* вершиной. Один шаг алгоритма состоит в обработке граничной вершины. Пусть x граничная вершина, тогда её обработка заключается в следующем:

1) Если в дереве имеется другая вершина y, не являющаяся граничной, и с ней связана та же маркировка, $\mu[x] = \mu[y]$, то вершина x становится *дублирующей.*
2) Если для маркировки $\mu[x]$ ни один из переходов не разрешен, то х становится *терминальной.*
3) В противном случае, для всякого перехода $t \in T$, разрешенного в $\mu[x]$, создаётся новая вершина z дерева достижимости. Маркировка $\mu[z]$, связанная с этой вершиной, определяется для каждой позиции $p \in P$ следующим образом:

   3.1) Если $\mu[x](p) = \omega$, то $\mu[z](p) = \omega$.

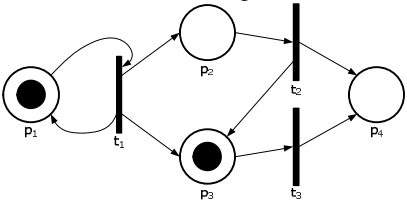
   3.2) Если на пути от корневой вершины к x существует вершина y с $\mu[y] < \mu'$ (где μ' – маркировка, непосредственно достижимая из $\mu[x]$ посредством запуска перехода t) и $\mu[y](p) < \mu'(p)$, то $\mu[z](p) = \omega$. (В этом случае последовательность запусков переходов, ведущая из маркировки $\mu[y]$ в маркировку μ', может неограниченно повторяться и неограниченно увеличивать значение маркировки в позиции p.) В противном случае $\mu[z](p) = \mu'(p)$.
4) Строится дуга с пометкой t, направленная от вершины x к вершине z. Вершина *x* становится *внутренней*, а вершина z – *граничной*.

Такая обработка алгоритмом граничных вершин продолжается до тех пор, пока все вершины дерева не станут терминальными, дублирующими или внутренними. Затем алгоритм останавливается.

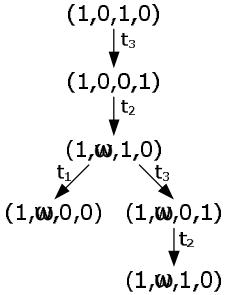
Важнейшим свойством алгоритма построения конечного дерева достижимости является то, что он за конечное число шагов заканчивает работу.

Пример. Конечное дерево достижимости сети Петри.

Сеть Петри



Конечное дерево достижимости



Важнейшим свойством алгоритма построения конечного дерева достижимости является то, что он за конечное число шагов заканчивает работу. Доказательство основано на трёх леммах.

Лемма 1. В любом бесконечном направленном дереве, в котором каждая вершина имеет только конечное число непосредственно последующих вершин, существует бесконечный путь, исходящий из корня.

Доказательство. Пусть x_0 корневая вершина. Поскольку имеется только конечное число непосредственно следующих за x_0 вершин, но общее число вершин в дереве бесконечно, по крайней мере, одна из непосредственно следующих за x_0 вершин должна быть корнем бесконечного поддерева. Выберем вершину x_1, непосредственно следующую за x_0, и являющуюся корнем бесконечного поддерева. Теперь одна из непосредственно следующих за ней вершин также является корнем бесконечного поддерева, выберем в качестве такой вершины x_2. Если продолжать этот процесс бесконечно, то получим бесконечный путь в дереве – $x_0, x_1, x_2, \ldots, x_n \ldots$.

Лемма 2. Всякая бесконечная последовательность неотрицательных целых содержит бесконечную неубывающую последовательность.

Доказательство. Возможны два случая:

1) Если какой-либо элемент последовательности встречается бесконечно часто, то пусть x_0 является таким элементом. Тогда бесконечная подпоследовательность $x_0, x_0, \ldots, x_0 \ldots$ является бесконечной неубывающей подпоследовательностью.
2) Если никакой элемент не встречается бесконечно часто, тогда каждый элемент встречается только конечное число раз. Пусть x_0 - произвольный элемент последовательности. Существует самое большее

x_0 целых, неотрицательных и меньших, чем x_0 $(0,\ldots x_0-1)$, причем каждый из них присутствует в последовательности только конечное число раз. Следовательно, продвигаясь достаточно долго по последовательности, мы должны встретить элемент $x_1, x_1 \geq x_0$. Аналогично должен существовать в последовательности $x_2, x_2 \geq x_1$, и т. д. Это определяет бесконечную неубывающую последовательность $x_0, x_1, x_2, \ldots, x_n, \ldots$.

Таким образом, в обоих случаях бесконечная неубывающая подпоследовательность существует.

Лемма 3. Всякая бесконечная последовательность n-векторов над расширенными символом ω неотрицательными целыми содержит бесконечную неубывающую подпоследовательность.

Доказательство. Доказываем индукцией по n, где n - размерность векторного пространства.

1. *Базовый случай* $(n=1)$. Если в последовательности имеется бесконечное число векторов вида $\langle\omega\rangle$, то они образуют бесконечную неубывающую последовательность (так как справедливо $\omega \leq \omega$). В противном случае бесконечная последовательность, образованная удалением конечного числа экземпляров $\langle\omega\rangle$, имеет бесконечную неубывающую подпоследовательность.

2. *Индуктивное предположение*. (Допустим, что лемма верна для n докажем её справедливость для $n+1$.) Рассмотрим первую координату. Если существует бесконечно много векторов, имеющих в качестве первой координаты ω, тогда выберем эту бесконечную подпоследовательность, которая не убывает (постоянна) по первой координате. Если только конечное число векторов имеют ω в качестве первой координаты, то рассмотрим бесконечную последовательность целых, являющихся значениями первых координат. По лемме 2 эта последовательность имеет бесконечную неубывающую подпоследовательность. Она определяет бесконечную Последовательность векторов, которые не убывают по своей первой координате.

В любом случае мы имеем последовательность векторов, неубывающих по первой координате. Применим индуктивное предположение к последовательности n-векторов, которая получается в результате отбрасывания первой компоненты $n+1$-векторов. Полученная таким образом бесконечная подпоследовательность является неубывающей по каждой координате.

Теорема . Дерево достижимости сети Петри конечно.

Доказательство. Докажем методом от противного. Допустим, что дерево достижимости бесконечно. Тогда по лемме 1 (и так как число вершин, следующих за каждой вершиной в дереве, ограничено числом переходов m) в нём имеется бесконечный путь $x_0, x_1, x_2, \ldots$, исходящий из корня x_0. Тогда $\mu, [x_0], \mu[x_1], \mu[x_2], \ldots, \ldots$ - бесконечная последовательность n-векторов. над $Nat \cup \{\omega\}$, а по лемме 3 она имеет бесконечную неубывающую подпоследовательность $\mu, [x_{k0}] \le \mu[x_{k1}] \le \mu[x_{k2}] \le \ldots$. Но по построению дерева достижимости $\mu[x_i] \ne \mu[x_j]$ (для $i \ne j$), поскольку тогда одна из вершин была бы дублирующей и не имела следующих за собой вершин. Следовательно, это бесконечная строго возрастающая последовательность $\mu, [x_{k0}] < \mu[x_{k1}] < \mu[x_{k2}] < \ldots$. Но по построению, так как $\mu[x_i] < \mu[x_j]$ нам следовало бы заменить по крайней мере одну компоненту $\mu[x_j]$, не являющуюся ω, на ω в $\mu[x_j]$. Таким образом, $\mu[x_{k1}]$ имеет по крайней мере одну компоненту, являющуюся ω, $\mu[x_{k2}]$ имеет по крайней мере две ω-компоненты, а $\mu[x_{kn}]$ имеет по крайней мере n ω-компонент. Поскольку маркировки n-мерные, $\mu[x_{kn}]$ имеет во всех компонентах ω. Но тогда у $\mu[x_{kn+1}]$ не может быть больше $\mu[x_{kn}]$. Пришли к противоречию, что доказывает теорему.

4.5.3 Анализ свойств сетей Петри на основе дерева достижимости

Анализ безопасности и ограниченности. Сеть Петри ограничена тогда и только тогда, когда символ ω отсутствует в ее дереве достижимости.

Присутствие символа ω в дереве достижимости ($\mu[x](p) = \omega$ для некоторой вершины x и позиции p) означает, что для произвольного положительного целого k существует достижимая маркировка со значением в позиции p, большим, чем k (а также бесконечность множества достижимых маркировок). Это, в свою очередь, означает неограниченность позиции p, а следовательно, и самой сети Петри.

Отсутствие символа ω в дереве достижимости означает, что множество достижимых маркировок конечно. Следовательно, простым перебором можно найти верхнюю границу, как для каждой позиции в отдельности, так и общую верхнюю границу для всех позиций. По-

следнее означает ограниченность сети Петри. Если граница для всех позиций равна 1, то сеть Петри безопасна.

Анализ сохранения. Так как дерево достижимости конечно, для каждой маркировки можно вычислить сумму начальной маркировки. Если эта сумма одинакова для каждой достижимой маркировки, то сеть Петри является сохраняющей. Если суммы не равны, сеть не является сохраняющей. Если маркировка некоторой позиции совпадает с ω, то эта позиция должна был исключена из рассмотрения.

Анализ покрываемости. Задача покрываемости требуется для заданной маркировки μ' определить, достижима ли маркировка $\mu'' \geq \mu'$. Такая задача решается путём простого перебора вершин дерева достижимости. При этом ищется такая вершина x, что $\mu[x] \geq \mu'$. Если такой вершины не существует, то маркировка μ' не является покрываемой. Если она найдена, то $\mu[x]$ определяет покрывающую маркировку для μ'. Если компонента маркировки $\mu[x]$, соответствующая некоторой позиции p совпадает с ω, то конкретное её значение может быть вычислено. В этом случае на пути от начальной маркировки к покрывающей маркировке имеется повторяющаяся последовательность переходов, запуск которой увеличивает значение маркировки в позиции p. Число таких повторений должно быть таким, чтобы значение маркировки в позиции p превзошло или сравнялось с $\mu'(p)$.

Анализ живости. Переход t сети Петри является потенциально живым, тогда и только тогда, когда он метит некоторую дугу в дереве достижимости этой сети.

Доказательство очевидно.

Ограниченность метода дерева достижимости. Как видно из предыдущего, дерево достижимости можно использовать для решения задач безопасности, ограниченности, сохранения и покрываемости. К сожалению, в общем случае его нельзя использовать для решения задач достижимости и активности, эквивалентности. Решение этих задач ограничено существованием символа ω. Символ ω означает потерю информации: конкретные количества фишек отбрасываются, учитывается только существование их большого числа.

4.5.3 Матричные уравнения

Другой подход к анализу сетей Петри основан на матричном представлении сетей Петри и решении матричных уравнений. Альтерна-

тивным по отношению к определению сети Петри N в виде (P,T,I,O) является определение сети N в виде двух матриц D^- и D^+, представляющих входную и выходную функции I и O. Пусть каждая из матриц D^- и D^+ имеет $m=|T|$ строк (по одной на переход) и $n=|P||$ столбцов (по одному на позицию).

Матричный вид сети Петри $N=(P,T,I,O)$ задаётся парой (D^-,D^+), где

$D^-[k,i]=\hat{}\#(p_i,t_k)$ – кратность дуги, ведущей из позиции p_i в переход t_k;

$D^+[k,i]=\#\hat{}(p_i,t_k)$ – кратность дуги, ведущей из перехода t_k в позицию p_i,

для произвольных $1\le k\le m,\quad 1\le i\le n$.

Пусть $e[k]$ - m-вектор, k-тый элемент которого равен 1, а остальные равны 0. Переход $t_k, 1\le k\le m$, в маркировке μ разрешен, если $\mu\ge e[k]D^-$. Результатом запуска разрешённого перехода t_k в маркировке μ является следующая ниже маркировка μ':

$$\mu'=\mu-e[k]D^-+e[k]D^+\quad \mu+e[k]D,$$

где $D=(D^+-D^-)$ - *составная матрица изменений.*