НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

А.А.НАБЕБИН, А.С.ТАРАСИКОВ

АЛГЕБРАИЧЕСКАЯ СПЕЦИФИКАЦИЯ ПРОГРАММНЫХ СИСТЕМ

Москва
ИНЭК
2012

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

А.А.НАБЕБИН, А.С.ТАРАСИКОВ

В тяжком 1941 году под Москвой
на пути немецких танковых колонн
встали Подольские курсанты.
Им, павшим за Родину, посвящается

АЛГЕБРАИЧЕСКАЯ СПЕЦИФИКАЦИЯ ПРОГРАММНЫХ СИСТЕМ

Москва
ООО НВП «ИНЭК»
2012

УДК 512.6 + 510.6
ББК 22.12 + 22.18
Н 12

Алгебраическая спецификация программных систем.
А.А.Набебин, А.С.Тарасиков
М.: Изд-во ИНЭК, 2012. 83с.

РЕЦЕНЗЕНТ: профессор С.М.Авдошин

ISBN 978-5-905675-04-1

Учебное пособие содержит необходимые сведения из однородных и неоднородных универсальных алгебр, системы аксиом для основных алгебраических структур (арифметика, моноид, полугруппы, группы, частичное упорядочение, кольца, поля). Описывается аксиоматический язык программирования OBJ3 с примерами программ на этом языке.

Предназначено студентам высших учебных заведений специальностей: прикладная математика, программирование, информатика.

ISBN 978-5-905675-04-1

© Набебин Алексей Александрович, НИУ ВШЭ

© Тарасиков Александр Сергеевич, НИУ ВШЭ

© ООО НВП «ИНЭК»

ВВЕДЕНИЕ

1. Множество

Понятие множества неопределимо. Это простейшее исходное понятие человечество сформировало из опыта всего своего исторического развития. То же можно сказать о смысле простейшего отношения принадлежности: элемент a принадлежит множеству A (обозначение $a \in A$) и о смысле отношения тождества (совпадения, равенства) двух элементов a и b из некоторого множества (обозначение $a = b$). Другими словами, предполагается, что читатель умеет распознавать совпадение или несовпадение двух элементов и устанавливать факт принадлежности или непринадлежности элемента множеству. Понятия конечного множества, натурального числа тоже неопределимы.

Пусть A, B, C есть произвольные множества; a, b, c есть элементы множеств. Итак, основными неопределяемыми отношениями в теории множеств являются следующие отношения:

$a = b$, элементы a и b равны (совпадают);

$a \in A$, элемент a принадлежит множеству A.

Пусть знак $\leftrightarrow$ означает "тогда и только тогда"; а знаки $\&$, $\vee$, $\rceil$, $\rightarrow$, $\forall$, $\exists$ есть логические знаки конъюнкции, дизъюнкции, отрицания, импликации, квантора общности и квантора существования. Используем их в общепринятом содержательном смысле.

Введем далее следующие отношения.

$A \subseteq B \leftrightarrow \forall a\ (a \in A \rightarrow a \in B)$;

$A = B \leftrightarrow A \subseteq B \ \& \ B \subseteq A$;

$A \subset B \leftrightarrow A \subseteq B \ \& \ A \neq B$;

$A \supseteq B \leftrightarrow B \subseteq A$;

$A \supset B \leftrightarrow B \subset A$.

Обозначим через $P(A)$ множество всех подмножеств множества A и пусть $\varnothing$ есть символ пустого множества. Введем операции над множествами.

$A \cup B = \{x : x \in A \vee x \in B\}$, объединение множеств A и B;

$A \cap B = \{x : x \in A \ \& \ x \in B\}$, пересечение множеств A и B;

$A - B = \{x : x \in A \ \& \ x \notin B\}$, разность множеств A и B;

$A \times B = \{(a, b) : a \in A \ \& \ b \in B\}$, декартово произведение множеств A и B.

Декартово произведение можно распространить на несколько сомножителей, именно, $A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) :$

$a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n\}$. Определим натуральную степень

$A^n = A \times A \times \ldots \times A$ (n раз).

Множества $\varnothing$ и A называются несобственными (тривиальными) подмножествами множества A. Если $A \subset B$ & $A \neq \varnothing$, то A есть собственное подмножество множества B.

Иногда пишут $A \cdot B$ или AB вместо $A \cap B$.

Примем следующие обозначения.

Множество натуральных чисел $\mathbb{N} = \{0,1,2,\dots\}$.

Множество положительных натуральных чисел $\mathbb{N}_+ = \{1,2,...\}$.

Множество целых чисел $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$.

Множество $\mathbb{Z}_n = E_n = \{0,1,2,\dots, n-1\}$.

Множество рациональных чисел $\mathbb{Q} = \{m/n : m, n \in \mathbb{Z}, n \neq 0\}$.

Множество вещественных чисел $\mathbb{R} = (-\infty, +\infty)$.

Множество комплексных чисел $\mathbb{C} = \{x + i\, y : x, y \in \mathbb{R}\}$.

2. *Функция*

Определение. Пусть A и B есть два множества. *Функция* f: $A \to B$ есть отображение, которое каждому элементу x из A ставит в соответствие некоторый элемент y из B. Это обстоятельство записывается как $y = f(x)$.

Замечание. В этом определении функция f всюду определена. Частично определенная функция $f : A \to B$ есть отображение, которое каждому элементу из множества A сопоставляет не более одного элемента из множества B.

Если $f(a)=b$, то элемент b есть образ элемента a, элемент a есть прообраз элемента b. Множество A есть область определения $D(f)$ функции f. Множество B есть область значений $R(f)$ функции f.

Образ $\mathrm{Im}\, f = \{f(x) : x \in A\}$ отображения $f : A \to B$ есть множество $f(A)$ всех значений функции f. Полный прообраз элемента $y \in B$ есть множество $f^{-1}(y) = \{x \in A : f(x) = y\}$. Полный прообраз множества $C \subseteq B$ есть множество $f^{-1}(C) = \{x \in A : f(x) \in C\}$.

Функцию с конечной областью определения удобно задавать таблицей. Например, пусть множества $A = \{1,2,3,4\}$, $B = \{1,2,3,4,5\}$,

функция $f = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 \end{pmatrix}$.. Здесь $f(1)=3$, $f(2)=2$, $f(3)=1$, $f(4)=2$.

Порядок столбцов несущественен. Область определения $D(f) = A = \{1,2,3,4\}$, область значений $R(f) = B = \{1,2,3,4,5\}$, $Im\, f = f(A) = \{1,2,3\}$.

Функции f: $A \to B$ и g: $C \to D$ равны, если $A=C$, $B=D$, $f(x) = g(x)$ $\forall x \in A$.

Функция I_A: $A \to A$, для которой $I(x)=x$ $\forall x \in A$, называется *тождественной* функцией.

Определение. Функция $\varphi: A \to B$ есть *взаимно однозначное соответствие* между множествами A и B, если

1) $\forall\, b \in B \;\; \exists a \in A \;\; f(a)=b$,

2) $\forall a_1, a_2 \in A \;\; (a_1 \neq a_2 \to f(a_1) \neq f(a_2))$.

Замечание. Последнее условие можно заменить на условие

2') $\forall a_1, a_2 \in A \;\; (f(a_1) = f(a_2) \to a_1 = a_2)$.

Определение. *Композиция* $g \circ f$ функций f: $A \to B$ и g: $B \to C$ есть функция $g \circ f : A \to C$, для которой $(g \circ f)(x) = g(f(x)) \; \forall x \in A$.

Замечание. Символ композиции $\circ$ иногда опускается.

Определение. Функция f^{-1}: $B \to A$ называется *обратной* к функции f: $A \to B$, если $f \circ f^{-1} = I_B$ и $f^{-1} \circ f = I_A$.

Замечание. 1. g обратна к $f \leftrightarrow f$ обратна g.

2. Функция f: $A \to B$ имеет обратную функцию если и только если функция f есть взаимно однозначное отображение.

3 . Отношение

Пусть $A_1, A_2, \ldots, A_n$ есть произвольные множества, вообще говоря, разнородные.

Определение. *n-арное отношение* p^n на множествах $A_г, A_г, .., A_n$ есть подмножество декартова произведения $A_1 \times A_2 \times \ldots \times A_n$.

Замечание. n-арное отношение p^n на множестве A есть подмножество декартова произведения $A \times A \times \ldots \times A$ (n раз). Индекс n арности (местности) отношения иногда опускается.

Иногда отношение определяют на множестве $A_1 \times A_2 \times \ldots \times A_n$.

Возможна предикатная $\rho(x_1, \ldots, x_n)$ и множественная $(x_1, \ldots, x_n) \in \rho$ формы записи отношений. Отношение ρ называют также предикатом. Для бинарного отношения используются записи $x\rho y$ и $\rho(x,y)$. Унарное отношение $\rho \subseteq E$ есть подмножество из E.

Набор $a=(a_1,a_2, \ldots, a_n) \in \rho$ (допустима запись $\rho(a_1,a_2, \ldots, a_n)$) называется элементом отношения.

Определение. Отношение *конечно*, если оно состоит из конечного числа элементов.

4. Отношение эквивалентности

Пусть A есть произвольное множество.

Определение. Бинарное отношение $\sigma \subseteq A \times A$ есть *отношение эквивалентности* (обозначение $a \sim b$), если оно удовлетворяет следующим аксиомам: $\forall a,b,c \in A$

1) $a \sim a$, рефлексивность,

2) $a \sim b \to b \sim a$, коммутативность,

3) $a \sim b \;\&\; b \sim c \to a \sim c$, транзитивность.

Обозначение. $a \sim b$, $\sigma(a,b)$, $(a,b) \in \sigma$, $a\sigma b$.

Определение. *Разбиение множества A* есть семейство попарно непересекающихся подмножеств из A, в объединении (в сумме)

дающих все A: $A = \bigcup_{i \in I} A_i$, $A_i \bigcap A_j = \varnothing \ \forall i \neq j$ Подмножества A_i называются смежными классами разбиения.

Пример. $A = \{0,1,2,3,4,5\} = \{0,1,5\} \cup \{2\} \cup \{3,4\}$.

Теорема. 1. Каждому отношению эквивалентности, определенному на множестве *A,* соответствует некоторое разбиение множества *A*.

2. Каждому разбиению множества *A* соответствую некоторое отношение эквивалентности.

Коротко: между классом всех определенных на множестве *A* эквивалентностей и классом всех разбиений множества *A* существует взаимно однозначное соответствие.

Доказательство. 1. Пусть σ есть отношение эквивалентности, определенное на множестве *A*. Пусть $a \in A$. Построим множество $K_a = \{x \in A : x \sim a\}$ всех элементов *x,* эквивалентных *a*. Оно обозначается также через $[a]_\sigma$. Множества K_a называются *смежными классами A по* σ, или классами эквивалентности.

Заметим, что если $b \in K_a$, то $b \sim a$. Покажем, что $a \sim b \leftrightarrow K_a = K_b$. В самом деле, пусть $a \sim b$. Пусть произвольный элемент $c \in K_a$. Тогда $c \sim a$, $a \sim b$, $c \sim b$, $c \in K_b$ и потому $K_a \subseteq K_b$. Аналогично показываем, что $K_b \subseteq K_a$. Тогда $K_a = K_b$. Пусть теперь $K_a = K_b$. Тогда $a \in K_a = K_b$,, $a \in K_b$, $a \sim b$. Утверждение доказано.

Если два класса K_a и K_b имеют общий элемент *c*, то они совпадают. В самом деле, если $c \in K_a$, $c \in K_b$, то $b \sim c$, $c \sim a$ и $b \sim a$, откуда $K_a = K_b$.. Поэтому всякие два класса эквивалентности либо не пересекаются, либо (в случае непустого пересечения) совпадают. Всякий элемент *c* попадает в класс эквивалентности K_c. Поэтому система смежных классов есть разбиение множества *A*.

2. Пусть имеем некоторое разбиение множества *A*. Определим на *A* отношение ~, положив $a \sim b \leftrightarrow$ элементы *a*,*b* принадлежат одному и тому же классу разбиения. Отношение ~ удовлетворяет аксиомам 1) $a \sim a$, 2) $a \sim b \rightarrow b \sim a$, 3) $a \sim b$ & $b \sim c$ и потому оно есть отношение эквивалентности.

Замечание. 1. Разбиение множества *A* на одноэлементные подмножества $A = \bigcup_{a \in A} \{a\}$ и разбиение *A*, состоящее из одного только множества *A*, называются тривиальными (несобственными) разбиениями.

2. Разбиение *A* на одноэлементные подмножества соответствует отношению эквивалентности, которое есть равенство.

3. Разбиение множества *A* состоящее из одного только множества *A*, соответствует отношению эквивалентности, содержащему все множество $A \times A$.

4. $a \sigma b \leftrightarrow [a]_\sigma = [b]_\sigma$.

Определение. Совокупность классов эквивалентности множества *A* называется *фактор-множеством* A/σ множества *A* по эквивалентности σ.

Определение. Отображение $p : A \to A/\sigma$, при котором $p(a) = [a]_\sigma$ называется *каноническим* (естественным).

5. Каноническое разложение функции

Пусть $f : A \to B$ есть некоторая функция. Определим на A отношение $\sigma \in A\times A$, положив $\forall\, a,a'\in A$ $(a \sim a' \leftrightarrow f(a)=f(a'))$. Отношение σ есть отношение эквивалентности, ибо $\forall\, a,b,c \in A$

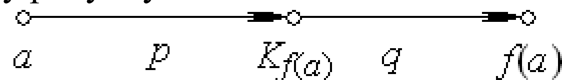
1) $a\sim a$, ибо $f(a)=f(a)$,

2) $a\sim b \to b\sim a$, ибо $f(a)=f(b) \to f(b)=f(a)$,

3) $a\sim b$ & $b\sim c \to a\sim c$, ибо $f(a)=f(b)\&f(b)=f(c) \to f(a)=f(c)$.

Введенное отношение σ называется ядерной эквивалентностью для отображения f. Классы эквивалентности A/σ есть полные прообразы элементов множества B при отображении f, то есть $A_b = f^{-1}(b) = \{a\in A : f(a)=b\}$.

Отображение f можно разложить в композицию двух отображений согласно следующему рисунку:



Имеет место равенство $f = q*p$, то есть $f(a) = q(p(a))$.

Представление $f = q*p$ называется каноническим разложением (представлением) функции f.

Пример. Получить каноническое разложение функции

$f : E_{10} \to E_{10}, f = 0112105533 = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7\,8\,9 \\ 01\;1\,2\,1\,0\,5\,5\,3\,3 \end{pmatrix}$.

Область определения $D(f) = E_{10}$. Область значений $Im(f)=\{0,1,2,3,5\}$. Классы эквивалентности:

$K_0 = [0]_\sigma = f^{-1}(0) = \{0,5\}$, $q(K_0) = 0$,

$K_1 = [1]_\sigma = f^{-1}(1) = \{1,2,4\}$, $q(K_1) = 1$,

$K_2 = [2]_\sigma = f^{-1}(2) = \{3\}$, $q(K_2) = 2$,

$K_3 = [3]_\sigma = f^{-1}(3) = \{8,9\}$, $q(K_3) = 3$,

$K_5 = [5]_\sigma = f^{-1}(5) = \{6,7\}$, $q(K_5) = 5$.

Функции p,q задаются следующим образом.

$p(a) = K_{f(a)} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ K_0 & K_1 & K_1 & K_2 & K_1 & K_0 & K_5 & K_5 & K_3 & K_3 \end{pmatrix}$,

$D(p) = E_{10}, Im(p) = \{K_0, K_1, K_2, K_3, K_5\};\ q(K_a) = a = \begin{pmatrix} K_0 K_1 K_2 K_3 K_5 \\ 0\ \ 1\ \ 2\ \ 3\ \ 5 \end{pmatrix},$

$D(q) = \{K_0, K_1, K_2, K_3, K_5\}, Im(q) = \{0,1,2,3,5\}; f(a) = q(p(a)).$

1. УНИВЕРСАЛЬНЫЕ АЛГЕБРЫ И КОМПЬЮТЕРНЫЕ ПРОГРАММЫ

1.1. Алгебры и спецификации программ

Программа есть последовательность машинных команд, предназначенная для достижения конкретного результата.

Программа согласно ГОСТ 19781-90 есть данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма.

Спецификация программной системы есть описание системы, которое полностью определяет ее цель и функциональные возможности. Различают:

1) словесные спецификации на естественном языке;
2) модельные спецификации;
3) формальные спецификации.

Модельные спецификации есть спецификации, предполагающие построение схем, диаграмм и других информационных структур.

Формальные спецификации есть спецификации, полученные формальным способом с использованием математических формализмов, которые обеспечивают полное определение специфицируемой семантики.

Язык спецификации есть рационально оформленный и синтаксически организованный набор средств спецификации программных систем.

Базисные компоненты языка спецификаций составляют конструкты (конструктивные средства) для специфицирования (описания) индивидуальных программных компонент, таких как:

1) типы и функции,
2) структурные механизмы для построения больших спецификаций модульным способом;
3) описание семантик языка;
4) механизмы для выполнения доказательств свойств спецификаций;
5) детализация (refinement) спецификаций;
6) способ соотнесения спецификаций с программами в языках программирования.

Язык спецификаций есть совместная комбинация этих конструктов.

Алгебраическая спецификация основана на логике равенства. Лежащая в основе семантика выводится из алгебры, в которой изучаются различные математические структуры как группы, кольца поля и т.д. Познакомимся, как строятся алгебры для спецификации элементов компьютерных программ.

Абстрактная алгебра, или просто *алгебра* $\mathcal{A} = (S, \Omega)$ есть совокупность S множеств и совокупность Ω функций (операций), определенных на множествах из S.

Тип данных (*сорт*) есть некоторое множество.

Тип (сорт) *Natural* есть множество $\mathbb{N} = \{0,1,2,\ldots\}$.

Тип (сорт) *Integer* есть множество $\mathbb{Z} = \{\ldots,-2,-1,0,1,2,\ldots\}$.

Тип (сорт) *Rational* есть множество рациональных чисел $\mathbb{Q} = \{m/n : m,n \in \mathbb{Z}, n \neq 0\}$.

Тип (сорт) *Real* есть множество вещественных чисел $\mathbb{R} = (-\infty, \infty)$.

Тип (сорт) *Complex* есть множество комплексных чисел $\mathbb{C} = \{x+ iy : x,y \in \mathbb{R}, i = \sqrt{-1}\}$.

Тип (сорт) *Array* есть множество конечных последовательностей, составленных из элементов выше перечисленных типов.

Алгоритм есть оператор (операция, функция, например, частично рекурсивная функция), определенный и принимающий значения на некоторой совокупности типов данных.

Программа $P = (D, Alg)$ есть совокупность D типов данных и совокупность Alg алгоритмов, то есть функций, определенных и принимающих значения на совокупности D типов данных.

Абстрактно, совокупность D типов данных есть совокупность некоторых множеств, а совокупность Alg алгоритмов есть совокупность функций, определенных на множествах из D. Поэтому абстрактно, программа $P = (D, Alg)$ может быть описана (специфицирована) как совокупность S некоторых множеств и совокупность Ω функций, определенных на этих множествах, то есть как некоторая алгебра $\mathcal{A} = (S, \Omega)$.

Алгебраическая спецификация программы $P = (D, Alg)$ есть алгебра $\mathcal{A} = (S, \Omega)$, описывающая программу P. Алгебра $\mathcal{A}$ есть абстракция программы P.

Для теории и практики программирования интересны формально аксиоматические представления алгебр. Приведем некоторые их примеры.

Определение. *Аксиоматическая арифметика натуральных чисел* есть алгебра $(\mathbb{N}, \{x+y, x\cdot y, x', 0\})$, где $\mathbb{N}$ есть множество натуральных чисел, на котором определены операции: сложение $x+y$, умножение $x\cdot y$, следование $x' = x+1$, отмеченный элемент 0 (ноль), удовлетворяющие следующим аксиомам (Пеано). $\forall\, x,y,z \in \mathbb{N}$:

1. $x = y \to (y = z \to x = z)$.

2. $\neg(x' = 0)$.

3. $x = y \to x' = y'$.

4. $x' = y' \to x = y$.

5. $x + 0 = x$.

6. $x + y' = (x + y)'$.

7. $x \cdot 0 = 0$.

8. $x \cdot y' = x \cdot y + x$.

Определение. *Моноид* есть алгебра $(M, \{f(x,y) = x * y, e\})$ где M есть множество, на котором определены

функция $x * y : M \times M \to M$,

нейтральный элемент e (единица) в M,

удовлетворяющие следующей аксиоме: $\forall x \in M$:

1. $x * e = x$. 2. $e * x = x$.

Определение. *Полугруппа* есть алгебра $(G, \{x*y\})$, где G есть некоторое множество, на котором определена бинарная операция $x*y$ (умножение), удовлетворяющая следующей аксиоме: $\forall x,y,z \in G$

1. $(x*y)*z = x*(y*z)$, ассоциативность.

Определение. *Группа* G есть алгебра $(G, \{f(x,y)=x*y, g(x)= x^{-1}, e\})$, где G есть некоторое множество, на котором определена бинарная операция $x*y$ (умножение), унарная операция x^{-1} (обратный элемент), отмеченный элемент e (единица) в G, удовлетворяющие следующим аксиомам: $\forall x,y,z \in G$

1. $(x * y) * z = x * (y * z)$, ассоциативность.

2. $x^{-1} * x = x * x^{-1} = e$.

3. $x * e = e * x = x$.

Замечание. 1. Всякая переменная есть терм.

2. Единица e есть терм.

3. Если t, t_1, t_2 есть термы, то выражения $f(t_1, t_2)$, $g(t)$ есть термы.

4. Аксиома группы есть равенство некоторых термов.

Определение. *Кольцо* R есть алгебра $(R, \{+, \cdot\})$ где R есть множество, на котором определены две операции (функции):

сложение $f(x, y) = x + y : R \times R \to R$,

умножение $g(x, y) = x \cdot y : R \times R \to R$,

удовлетворяющие следующим аксиомам. $\forall x,y,z \in R$:

1. $(x + y) + z = x + (y + z)$.
2. $x + y = y + x$.
3. $\exists\ 0 \in R\ \ x + 0 = x$.
4. $\forall x \in R\ \exists (-x) \in R\ \ x + (-x) = 0$.
5. $(x \cdot y) \cdot z = x(yz)$.
6. $(x + y) \cdot z = x \cdot z + y \cdot z,\ x \cdot (y + z) = x \cdot y + x \cdot z$.

Кольцо *коммутативно*, если умножение коммутативно:

7. $x \cdot y = y \cdot x$.

Замечание. 1. Всякая переменная есть терм.

2. Аддитивная единица 0 (ноль) есть терм.

3. Если t, t_1, t_2 есть термы, то выражения $f(t_1, t_2)$, $g(t_1, t_2)$ есть термы. Заметим, что термы есть основной материал, из которых строятся аксиомы программы как равенство двух термов.

Замечание. 1. Кольцо конечно, если множество R содержит конечное число элементов. Кольцо есть коммутативная группа по сложению.

2. Аксиомы кольца есть равенства некоторых термов.

Пример. 1. Множество целых чисел $\mathbb{Z}$ со сложением и умножением есть коммутативное кольцо.

2. Множество $\mathbb{Z}_n$ со сложением и умножением по модулю n есть коммутативное кольцо.

Определение. *Поле* F есть алгебра $(F, \{+, \cdot\})$, где F есть множество, на котором определены две операции (функции):

сложение $f(x,y) = x + y : F \times F \to F$,

умножение $g(x,y) = x \cdot y : F \times F \to F$,

удовлетворяющие следующим аксиомам. $\forall\ x,y,z \in F$:

1. $(x + y) + z = x + (y + z)$.
2. $x + y = y + x$.
3. $\exists\ 0 \in F\ \ x + 0 = x$.
4. $\forall x \in F\ \exists (-x) \in F\ \ x + (-x) = 0$.
5. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
6. $x \cdot y = y \cdot x$.
7. $\exists e \in F\ \ \forall x \in F\ \ x \cdot e = x$.
8. $\forall x \in F - \{0\}\ \ \exists x^{-1} \in F\ \ x \cdot x^{-1} = e$.
9. $(x + y) \cdot z = x \cdot z + y \cdot z$.

Замечание. 1. Всякая переменная есть терм.

2. Ноль 0 и единица e есть термы.

3. Если t, t_1, t_2 есть термы, то выражения $f(t_1, t_2)$, $g(t_1, t_2)$ есть термы.

4. Аксиомы поля есть равенства некоторых термов.

Замечание. Поле есть коммутативная группа по сложению. Поле без нуля есть коммутативная циклическая группа по умножению.

Определение. Характеристика поля есть наименьшее положительное число m, для которого $\sum_{i=1}^{m} 1 = 0$, если такое число m существует. Характеристика поля есть 0, если $\sum_{i=1}^{m} 1 \neq 0$ для всякого $m \geq 1$.

Утверждение. Множество $\mathbb{Z}_n$ со сложением и умножением по модулю n есть поле, если и только если n есть простое число. Если число n просто, то $\mathbb{Z}_n$ имеет характеристику n.

Определение. *Линейное векторное пространство* L над полем F есть множество L элементов произвольной природы (векторов) x, y, z,…, на котором определены две (линейные) операции:

сложение векторов $f(x,y) = x+y$: $L \times L \to L$,

умножение $g(a,x) = a \cdot x : F \times L \to L$ элемент a из F на вектор x (при этом $ax=xa$), удовлетворяют следующим аксиомам. $\forall x,y,z \in L$, $\forall a,b,c \in F$:

1. $x + (y + z) = (x + y) + z$.
2. $x + y = y + x$.
3. $\exists 0 \in L \;\; x + 0 = x$.
4. $\exists (-x) \in L \quad x + (-x) = 0$.
5. $a \cdot (x + y) = a \cdot x + a \cdot y$.
6. $(a \oplus b) \cdot x = a \cdot x + a \cdot y$.
7. $(a \odot b) \cdot x = a \cdot (b \cdot x)$.
8. $1 \cdot x = x$, $1 \in F$.

Вектор 0 из L есть нулевой вектор в L. Вектор $–x$ из L называется вектором, противоположным к x.

Замечание. 1. В линейном пространстве существуют объекты двух типов (сортов): векторы и элементы поля.

Определение. *Булева алгебра* A есть множество A, на котором определены:

два выделенных элемента 0 и 1 (из A),

функция $f(x,y) = x \,\&\, y : \; A \times A \to A$,

функция $g(x,y) = x \vee y : \; A \times A \to A$,

функция $n(x) : \; A \to A$ (часто обозначаемая как $\neg x$ или $\bar{x}$),

удовлетворяющие следующим аксиомам. $\forall \, x,y,z \in A$:

1. $x \,\&\, x = x$, $x \vee x = x$.
2. $x \,\&\, y = y \,\&\, x$, $x \vee y = y \vee x$.
3. $x \,\&\, (y \,\&\, z) = (x \,\&\, y) \,\&\, z$, $x \vee (y \vee z) = (x \vee y) \vee z$.
4. $x \,\&\, (x \vee y) = x$, $x \vee (x \,\&\, y) = x$.
5. $x \,\&\, (y \vee z) = x\&y \vee x\&z$, $x \vee (y \,\&\, z) = (x \vee y) \,\&\, (x \vee z)$.

6. $\overline{\overline{x}} = x$.

7. $x \,\&\, 1 = x,\ x \,\&\, 0 = 0,\ x \vee 1 = 1,\ x \vee 0 = x$.

8. $x \vee \overline{x} = 1,\ x \,\&\, \overline{x} = 0$.

9. $\overline{x \,\&\, y} = \overline{x} \vee \overline{y},\ \overline{x \vee y} = \overline{x} \,\&\, \overline{y}$.

Иногда к булевым операциям присоединяют импликацию

$h(x,y) = x \to y : A \times A \to A$ с аксиомой

10. $x \to y = \overline{x} \vee y$.

Пример. 1. Множество $A = \{0, 1\}$ с операциями конъюнкция, дизъюнкция, отрицание, импликация и двумя выделенными элементами 0 и 1 образуют булеву алгебру.

2. Множество $P(A)$ всех подмножеств некоторого множества A с операциями объединения, пересечения, дополнения (до множества A) и двумя выделенными элементами 0 (пустое множество) и 1 (множество A) образует булеву алгебру.

Определение. *Решетка* L есть множество A, на котором определены:

функция $f(x,y) = x \,\&\, y : A \times A \to A$,

функция $g(x,y) = x \vee y : A \times A \to A$,

удовлетворяющие следующим аксиомам. $\forall\, x,y,z \in A$:

1. $x \,\&\, x = x,\ x \vee x = x$.

2. $x \,\&\, y = y \,\&\, x,\ x \vee y = y \vee x$.

3. $x \,\&\, (y \,\&\, z) = (x \,\&\, y) \,\&\, z,\ x \vee (y \vee z) = (x \vee y) \vee z$.

4. $x \,\&\, (x \vee y) = x,\ x \vee (x \,\&\, y) = x$.

Дистрибутивная решетка есть решетка с аксиомой дистрибутивности:

5. $x \,\&\, (y \vee z) = x\&y \vee x\&z,\ x \vee (y \,\&\, z) = (x \vee y) \,\&\, (x \vee z)$.

Определение. *Частично упорядоченное множество* (ЧУМ) есть некоторое множество A с определенным на нем отношением порядка (или просто порядком) $x \preccurlyeq y$ (меньше или равно), удовлетворяющее следующим аксиомам. $\forall\, x,y,z \in A$

1. $x \preccurlyeq x$.

2. $x \preccurlyeq y \,\&\, y \preccurlyeq x \to x = y$.

3. $x \preccurlyeq y \,\&\, y \preccurlyeq z \to x \preccurlyeq z$.

Пример. Множество $P(A)$ всех подмножеств некоторого множества A с отношением включения множеств $x \subseteq y$ есть ЧУМ.

Замечание. Иногда выше определенное ЧУМ называют нестрого частично упорядоченным множеством. *Строго частично упорядоченное множество A* со строгим частичным порядком $x \prec y$ определяется следующими аксиомами. $\forall\, x,y,z \in A$

1. $x \not\prec x$.

2. $x \prec y \;\&\; y \prec z \;\rightarrow\; x \prec z$.

Пример. Множество $P(A)$ всех подмножеств некоторого множества A с отношением строгого включения множеств $x \subset y$ есть строго частично упорядоченное множество.

1.2. Алгебры, подалгебры, гомоморфизм алгебр

1.2.1. Односортные (однородные, homogeneous) алгебры

Пусть A есть непустое множество. n-арная (n-местная) операция на A есть функция (отображение) $f : A^n \rightarrow A$.

0-арная операция на A есть элемент из A.

Обозначим через $n(f)$ арность операции f.

Определение. *Универсальная односортная алгебра* есть система $U = (A, \Omega)$, где A есть некоторое множество и $\Omega = \{\, f_i^{n_i}(x_1,...,x_{n_i}) : i=1,2,...\}$ есть совокупность операций, определенных на A. *Тип* универсальной алгебры U есть последовательность $(n_1, n_2,...)$ арностей функций f_i. *Сигнатура* есть множество $\Omega = \{\, f_i^{n_i} : i=1,2,...\}$ символов операций.

Определение. Алгебра $U = (A, \Omega)$, (слово универсальная часто опускается) называется *конечной алгеброй*, если множество A конечно, и *конечного типа*, если конечно множество Ω.

Пример. 1. Система $(\mathbb{N}, \{+\})$ с операцией сложения, определенной на множестве $\mathbb{N}$ натуральных чисел, есть универсальная алгебра типа 2 сигнатуры $\{+\}$.

2. Система $(\mathbb{Z}, \{+, \cdot\})$ с операциями сложения и умножения, определенными на множестве $\mathbb{Z}$ целых чисел, есть универсальная алгебра типа (2,2) сигнатуры $\{+,\cdot\}$.

3. Система $(\mathbb{Q}, \{+, \cdot, -, /\})$ с операциями сложения, умножения, вычитания, деления, определенными на множестве $\mathbb{Q}$ рациональных чисел, есть универсальная алгебра типа (2,2,2,2) сигнатуры $\{+, \cdot, -, /\}$. Операция деления частично (не всюду) определена.

4. Система $(\mathbb{R},\{+, \cdot, -, /, \uparrow 1, \uparrow 12, \ldots, \uparrow n, \ldots\})$ с операциями сложения, умножения, вычитания, деления, возведения в степень $\uparrow n$, определенными на множестве $\mathbb{R}$ вещественных чисел, есть универсальная алгебра типа (2,2,2,2,1,1,1...) сигнатуры $\{+, \cdot, -, /, \uparrow 1, \uparrow 12, \ldots, \uparrow n, \ldots\}$. Операция деления частично определена.

5. Решетка $L = (A, \{\vee, \wedge\})$ есть универсальная алгебра типа (2,2) сигнатуры $\{\vee, \wedge\}$.

6. Булева алгебра $\mathcal{A} = (A, \{\vee, \wedge, \neg, 0, 1\})$ есть универсальная алгебра типа (2,2,1,0,0) сигнатуры $\{\vee, \wedge, \neg, 0, 1\}$.

Определение. Пусть $U = (A, \Omega)$ есть алгебра. *Суперпозиция* (*терм*) над Ω определяется по индукции следующим образом.

1. Всякая функция из Ω есть суперпозиция над Ω.

2. Если функция $f(x_1,..., x_n) \in \Omega$ и каждое из $g_1,..., g_n$ есть либо суперпозиция над Ω, либо переменная, то $f(g_1,..., g_n)$ есть суперпозиция над Ω.

Замечание. 1. Суперпозиция (терм) над Ω есть обычная подстановка, построенная из функций множества Ω. Суперпозиция над Ω допускает переименование переменных.

2. *Основной терм* над Ω есть терм без переменных.

Пусть $S(x_1,..., x_n)$ есть некоторая суперпозиция над Ω в алгебре $U = (A, \Omega)$ и элементы $a_1,..., a_n$ лежат в A. Тогда $S(a_1,..., a_n)$ есть *значение суперпозиции S* над Ω на наборе $a_1,..., a_n$.

Пусть $U = (A, \Omega)$ есть алгебра и $A_1 \subseteq A$ есть непустое подмножество в A. Множество $[A_1]_f$ есть *замыкание множества* A_1 по n-арной операции $f \in \Omega$, если 1) $A_1 \subseteq [A_1]_f$,

2) $\forall a \in [A_1]_f$ $(a \in A_1 \vee \exists$ суперпозиция $g(y_1,..., y_m)$ над $\{f\}$, $\exists\, b_1,...,b_m \in A_1$ $g(b_1,..., b_m) = a)$.

Другими словами, множество $[A_1]_f$ есть замыкание множества A_1 по n-арной операции $f \in \Omega$, если 1) A_1 есть подмножество в $[A_1]_f$, 2) любой элемент из $[A_1]_f$ либо лежит в A_1, либо есть значение некоторой суперпозиции над $\{f\}$.

Замечание.

1. $A_1 \subseteq [A_1]_f$.

2. $[[A_1]_f] = [A_1]_f$.

3. $A_1 \subseteq A_2 \rightarrow [A_1]_f \subseteq [A_2]_f$.

Определение. Множество A_1 из A *замкнуто по n-арной операции f*, если замыкание $[A_1]_f = A_1$.

Определение. Пусть $U = (A, \Omega)$ есть алгебра и $A_1 \subseteq A$ есть непустое подмножество в A. Множество $[A_1]$ есть *замыкание множества* A_1 в алгебре U, если

1) $A_1 \subseteq [A_1]$,

2) $\forall\, a \in [A_1]$ ($a \in A_1 \vee \exists$ суперпозиция $g(y_1,...,y_m)$ над Ω, $\exists\, b_1,...,b_m \in A_1$ $g(b_1,..., b_m) = a$).

Замечание. 1. $A_1 \subseteq [A_1]$.

2. $[[A_1]] = [A_1]$.

3. $A_1 \subseteq A_2 \rightarrow [A_1] \subseteq [A_2]$.

Определение. Множество A_1 из A замкнуто в алгебре U, если замыкание $[A_1] = A_1$.

Определение. Система $U_1 = (A_1, \Omega)$ с A_1 из A называется *подалгеброй алгебры* $U = (A, \Omega)$, или замкнутым классом, если множество A_1 замкнуто в U (то есть если $[A_1] = A_1$). Подалгебра $U_1 = ([A_1], \Omega)$ называется *подалгеброй, порожденной системой генераторов* A_1.

Пример. 1. Система $(\mathbb{N}2,\{+\})$ четных натуральных чисел со сложением есть подалгебра алгебры $(\mathbb{N},\{+\})$.

2. Система $(\mathbb{N}(p),\{\cdot\})$, где $\mathbb{N}(p)$ есть множество всех натуральных чисел, делящихся на простое число p, есть подалгебра алгебры $(\mathbb{N}, \{\cdot\})$ натуральных чисел с умножением.

Определение. Пусть A, B есть два множества и

$U_A = (A, \Omega_A=\{ f_i^{n_i}(x_1,...,x_{n_i}) : i=1,2,...\}$,

$U_B = (B, \Omega_B=\{ g_i^{n_i}(x_1,...,x_{n_i}) : i=1,2,...\}$

есть две универсальные алгебры одного и того же типа $(n_1,n_2,...)$. Отображение $\varphi : A \rightarrow B$ есть *гомоморфизм* (или гомоморфное отображение) алгебры U_A в алгебру U_B, если функция φ сохраняет операции в U_A :

$\forall\ x_1,...,x_{n_i} \in A,\ \forall\, i=1,2,...$

$\varphi(f_i^{n_i}(x_1,...,x_{n_i})) = g_i^{n_i}(\varphi(x_1),...,\varphi(x_{n_i}))$.

Говорят также, что алгебра U_A *гомоморфна* алгебре U_B . Взаимно однозначный гомоморфизм U_A в U_B называется *изоморфизмом* U_A на U_B.

Говорят тогда, что алгебра U_A *изоморфна* алгебре U_B (обозначение: $U_A \cong U_B$). Изоморфизм U_A на себя называется *автоморфизмом*.

Теория универсальных алгебр преимущественно изучает абстрактные свойства алгебр, то есть свойства, сохраняющиеся при гомоморфизме, изоморфизме, автоморфизме.

Пример. Пусть $U_1 = (\mathbb{R}_+, \{\cdot\})$ и $U_2 = (\mathbb{R}, \{+\})$ есть две алгебры с операциями умножения на множестве положительных вещественных чисел $\mathbb{R}_+$ для U_1 и сложения на множестве вещественных чисел $\mathbb{R}$ для U_2. Взаимно однозначное отображение $\varphi(x) = \ln(x) : \mathbb{R}_+ \to \mathbb{R}$ есть изоморфизм U_1 на U_2, ибо $\varphi(x\cdot y) = \ln(x\cdot y) = \ln(x) + \ln(y) = \varphi(x) + \varphi(y)$.

Теорема. При гомоморфном отображении одной алгебры в другую образы подалгебр и полные прообразы подалгебр являются подалгебрами.

Доказательство. Пусть $\varphi : A \to B$ есть гомоморфизм алгебры

$U_A = (A, \Omega_A = \{ f_i^{n_i}(x_1,...,x_{n_i}) : i=1,2,...\})$ в однотипную алгебру

$U_B = (B, \Omega_B = \{ g_i^{n_i}(x_1,...,x_{n_i}) : i=1,2,...\})$,

Покажем, что всякий образ подалгебры алгебры U_A при гомоморфизме φ есть подалгебра алгебры U_B. Пусть $U_C = (C, \Omega_A)$ есть подалгебра алгебры U_A с $C \subseteq A$ и пусть $U_D = (D=\varphi(C), \Omega_B)$ есть гомоморфный образ U_C при отображении φ. Пусть g^n есть произвольная операция в U_B , для которой в U_A соответствует операция f^n. Пусть $d_1,...,d_n$ есть произвольные элементы в образе $\varphi(C) = D$; $c_1,...,c_n$ есть некоторые прообразы в C для $d_1,...,d_n$ соответственно. Тогда $\varphi(c_i)= d_i$, $i=1,2,...,n$ (рис.2.1).

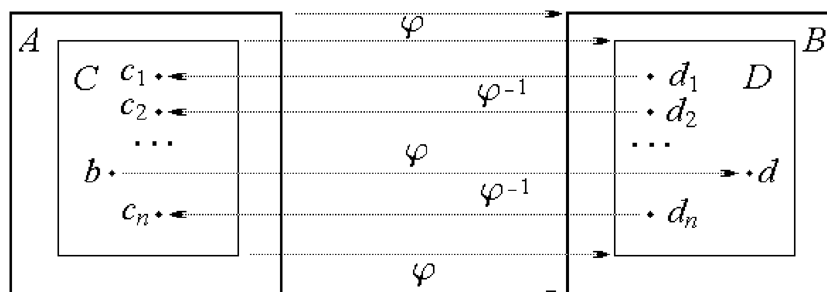


Рис.1.1

Пусть $f(c_1,...,c_n) = b$. Тогда в силу гомоморфизма $\varphi(b) = \varphi(f(c_1,...,c_n)) = g(\varphi(c_1),...,\varphi(c_n)) = g(d_1,...,d_n) = d$. В силу замкнутости C элемент $b \in C$. В силу гомоморфизма $\varphi(b) = g(d_1,...,d_n) = d \in D$. Итак, $\forall\, d_1,...,d_n \in D$ $\forall\, g \in \Omega_B$

$g(d_1,...,d_n) \in D$. Следовательно, множество D замкнуто в U_D и система U_D есть подалгебра алгебры U_B.

Покажем, что полный прообраз U_C некоторой подалгебры U_D алгебры U_B при гомоморфизме φ есть подалгебра алгебры U_A. Пусть $C = \varphi^{-1}(D)$. Покажем, что множество C замкнуто в U_A. Пусть произвольная операция $f^n \in \Omega_A$; операции f^n соответствует $g^n \in \Omega_B$; произвольные элементы $c_1,...,c_n \in C$. Они являются прообразами некоторых элементов $d_1,...,d_n$ в D. Тогда $\varphi(c_i) = d_i$, $i=1,2,...,n$. Так как D замкнуто, то $g(d_1,...,d_n) \in D$. По определению гомоморфизма $\varphi(f(c_1,...,c_n)) = g(\varphi(c_1),..., \varphi(c_n)) = g(d_1,...,d_n) \in D$. Следовательно, множество C замкнуто и потому U_C есть подалгебра алгебры U_A. Теорема доказана.

Следствие. При гомоморфизме φ алгебры U_A в алгебру U_B образ алгебры U_A есть подалгебра алгебры U_B.

1.2.2. Конгруенции

Пусть $\varphi : A \to B$ есть функция из A в B и $\varphi(A)=B$. Отображение $\varphi : A \to B$ порождает разбиение множества A на классы $A_b = \{\varphi^{-1}(b) : b \in B\}$ (ядерной) эвивалентности σ_φ, причем $a_1 \, \sigma_\varphi \, a_2 \leftrightarrow \varphi(a_1)=\varphi(a_2)$ $\forall \, a_1, a_2 \in A$. Пусть A/σ есть множество классов эквивалентности A_b. Тогда отображение $h : A/\sigma \to B$, при котором $h(A_b) = b$, взаимно однозначно. Пусть $p : A \to A/\sigma$, при котором $p(a) = A_{\varphi(a)}$ есть каноническое отображение. Тогда $\varphi = h \cdot p = h(p)$ есть *каноническое разложение* функции.

Обозначим через $[a]_\sigma$ класс эквивалентности, содержащий элемент a из A. Тогда $p(a) = [a]_\sigma = A_{\varphi(a)}$, $h([a]_\sigma) = \varphi(a)$.

Пусть записи $(a, a') \in \sigma$, $a \, \sigma \, a'$, $a \sim a'(\sigma)$, $a \sim a'$ означают: элементы a и a' эквивалентны по эквивалентности σ.

Определение. Функция $f : A \to A$ сохраняет m-арное отношение $\rho \in A$, определенное на множестве A, если для всяких n наборов

$(a_{11}, \dots, a_{1m}) \in \rho$,

$\dots$ набор $(f(\boldsymbol{a}^1), ..., f(\boldsymbol{a}^m)) \in \rho$.

$(a_{n1}, \dots, a_{nm}) \in \rho$,

$\boldsymbol{a}^1$, ..., $\boldsymbol{a}^m$))

Замечание. Функция $f : A^n \to A$ сохраняет отношение эквивалентности σ, если для всяких n эквивалентностей

$(a_1 \sim a_1'),$

$\ldots$ будет $f(a_1, \ldots, a_n) \sim f(a_1', \ldots, a_n')$ и потому

$(a_n \sim a_n')$

$[f(a_1, \ldots, a_n)]_\sigma = [f(a_1', \ldots, a_n')]_\sigma$.

Определение. Эквивалентность σ, определенная на множестве A универсальной алгебры U_A, есть *конгруенция* на U_A, если всякая функция из Ω_A сохраняет отношение σ.

Пусть на алгебре $U_A = (A, \Omega_A = \{ f_i^{n_i} : i=1,2,...\})$ задана конгруенция σ. Пусть функция $f^n \in \Omega_A$. Определим на множестве A/σ однотипную с U_A алгебру $U_{A/\sigma} =(A/\sigma, \Omega_{A/\sigma} = \{ F_i^{n_i} : i=1,2,...\})$, положив $F([a_1]_\sigma ,..., [a_n]_\sigma) = [f(a_1, \ldots, a_n)]_\sigma$ для каждой операции F^n из $U_{A/\sigma}$, соответствующей операции f^n из Ω_A.

Операция F на A/σ определена корректно. В самом деле, так как всякая функция f^n из Ω_A сохраняет отношение σ, то при $a_1 \sim a_1', ..., a_n \sim a_n'$ будет $f(a_1, \ldots, a_n) \sim f(a_1', \ldots, a_n')$, откуда $[f(a_1, \ldots, a_n)]_\sigma = [f(a_1', \ldots, a_n')]_\sigma$ и потому

$F([a_1]_\sigma ,..., ([a_n]_\sigma) = [f(a_1, \ldots, a_n)]_\sigma = [f(a_1', \ldots, a_n')]_\sigma = F([a_1']_\sigma ,...,[a_n']_\sigma)$.

Алгебра $U_{A/\sigma}$ называется *фактор-алгеброй* алгебры U_A по конгруенции σ.

Теорема. Каноническое отображение $p : A \to A/\sigma$ алгебры U_A на фактор-алгебру $U_{A/\sigma}$ есть гомоморфизм, для которого конгруенция σ служит ядерной эквивалентностью.

Доказательство. Очевидно, что отображение p порождает ядерную эквивалентность σ. Отображение p сохраняет операции алгебры U_A, ибо для всякой операции f^n из Ω_A, для всяких элементов $a_1, \ldots, a_n$ из A по построению функций F^n будет:

$p(f(a_1, \ldots, a_n)) = [f(a_1, \ldots, a_n)]_\sigma = F([a_1]_\sigma ,..., [a_n]_\sigma) = F(p(a_1), ..., p(a_n))$.

Теорема. Пусть $\varphi : A \to B$ есть гомоморфизм алгебры $U_A = (A, \Omega_A = \{ f_i^{n_i} : i=1,2,...\})$ на однотипную алгебру $U_B = (B, \Omega_B = \{ g_i^{n_i} : i=1,2,...\})$. Ядерная эквивалентность σ гомоморфизма φ есть конгруенция на U_A, и отображение

h: $A/\sigma \to B$, при котором $h(A_b) = b$, есть изоморфизм алгебры $U_{A/\sigma}$ на алгебру U_B .

Доказательство. Покажем, что ядерная эквивалентность σ есть конгруенция на U_A, то есть что всякая функция из U_A сохраняет отношение σ. Пусть операции f^n из Ω_A соответствует операция g^n из Ω_B. Пусть $a_1 \sim a_1'$, ..., $a_n \sim a_n'$ по эквивалентности σ. Тогда $\varphi(a_1) = \varphi(a_1')$, ..., $\varphi(a_n) = \varphi(a_n')$,

$g(\varphi(a_1), ..., \varphi(a_n)) = g(\varphi(a_1'),..., \varphi(a_n'))$, откуда $\varphi(f(a_1, \ldots, a_n)) =$

$g(\varphi(a_1),..., \varphi(a_n)) = g(\varphi(a_1'),..., \varphi(a_n')) = \varphi(f(a_1', ..., a_n'))$ и потому

$f(a_1, \ldots, a_n) \sim f(a_1', ..., a_n')$,

то есть функция f сохраняет отношение σ. Следовательно, ядерная эквивалентность σ есть конгруенция на алгебре U_A. Конгруенция σ порождает фактор-алгебру $U_{A/\sigma}$, однотипную с алгебрами U_A и U_B.

Отображение $h : A/\sigma \to B$, при котором $h([a]_\sigma) = \varphi(a)$, взаимно однозначно. Отображение h есть гомоморфизм, ибо

$h(F([a_1]_\sigma ,..., [a_n]_\sigma) =$ [по построению функции F] $=$

$h([f(a_1, \ldots, a_n)_\sigma]) =$ [в силу $h([a]_\sigma) = \varphi(a)$] $=$

$\varphi(f(a_1, \ldots, a_n)) =$ [так как φ есть гомоморфизм] $=$

$g(\varphi(a_1),..., \varphi(a_n)) =$ [в силу $h([a]_\sigma) = \varphi(a)$] $=$

$g(h([a_1]_\sigma), ..., h([a_n]_\sigma))$,

и потому функция h сохраняет операции из $U_{A/\sigma}$. Взаимно однозначный гомоморфизм h есть изоморфизм алгебр $U_{A/\sigma}$ и U_B. Теорема доказана.

Замечание. На рис.1.2 показано разложение гомоморфизма φ алгебры U_A

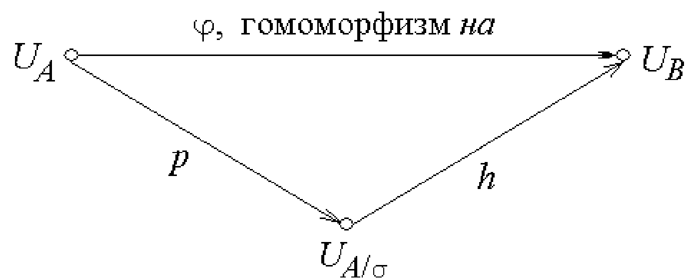


Рис.1.2

на алгебру U_B в произведение гомоморфизма *p* алгебры U_A на фактор-алгебру $U_{A/\sigma}$ и изоморфизма *h* фактор-алгебры $U_{A/\sigma}$ на алгебру U_B по ядерной эквивалентности σ (являющейся конгруенцией на U_A), порожденной отображением φ. С другой стороны, любая конгруенция на U_A, являясь отношением эквивалентности на *A*, порождает гомоморфизм φ алгебры U_A на фактор-алгебру $U_{A/\sigma}$. Поэтому множество всех гомоморфных образов алгебры U_A равномощно множеству всех конгруенций σ на U_A, и потому совокупность гомоморфных образов алгебры U_A на алгебру U_B с точностью до изоморфизма исчерпывается совокупностью всех фактор-алгебр алгебры U_A по различным ее конгруенциям.

1.2.3. Многосортные (неоднородные, heterogeneous) алгебры

Определение. *Универсальная неоднородная алгебра* есть система $\Sigma = (S, \Omega)$, где $S = S_1 \cup \ldots \cup S_k$ есть система из *k* сортов попарно непересекающихся множеств, и $\Omega = \{ f_i^{n_i}(x_{j_1},...,x_{j_{n_i}}) : i=1,2,...\}$ есть совокупность операций, определенных на подмножествах из *S*. При этом $x_{j_1} \in S_{j_1},\ldots,\ x_{j_{n_i}} \in S_{j_{n_i}}$.

Определение. Пусть $\Sigma_A = (S_A, \Omega_A)$, $\Sigma_B = (S_B, \Omega_B)$ есть две неоднородных алгебры, где

$S_A = S_1^A \cup \ldots \cup S_{k_A}^A$;

$\Omega_A = \{ f_i^{n_i}(x_{j_1},...,x_{j_{n_i}}) : i=1,2,...\};\ x_{j_1} \in S_{j_1}^A,\ \ldots,\ x_{j_{n_i}} \in S_{j_{n_i}}^A$;

$S_B = S_1^B \cup \ldots \cup S_{k_B}^B$,

$\Omega_B = \{ g_i^{n_i}(x_{j_1},...,x_{j_{n_i}})\ i=1,2,...\};\ x_{j_1} \in S_{j_1}^B,\ \ldots,\ x_{j_{n_i}} \in S_{j_{n_i}}^B$.

Гомоморфизм из Σ_A в Σ_B есть отображение *F*, которое каждому сорту s_A из S_A ставит в соответствие сорт s_B из S_B, причем если f^n из Ω_A соответствует g^n из Ω_B, то $f^n(x_{j_1},...,x_{j_n}) = g^n(F(x_{j_1}),...,F(x_{j_n}))$.

Замечание. Алгебраическая спецификация программ часто требует неоднородных (многосортных) алгебр.

Пример 1. Пусть алгебра $\mathcal{U} = (S, \Omega)$, где *S* есть узлы плоской прямоугольной решетки с неравномерным шагом по четырем направлениям;

$\Omega = \{right(x), left(x), below(x), above(x)\}$ есть множество из четырех унарных операций (функций), определенных на *S*. Ниже следуют аксиомы, определяющие эти операции.

1. *left(right(x)) = x = right(left(x)).*
2. *above(below(x)) = x = below(above(x)).*
3. *right(above(x)) = above(right(x)).*
4. *left(above(x)) = above(left(x)).*
5. *right(below(x)) = below(right(x)).*
6. *left(below(x)) = below(left(x)).*

Пример 2. Пусть алгебра $\mathcal{L} = (S, \Omega)$, где S есть узлы $p=(x,y)$ плоской прямоугольной решетки с целочисленными координатами и с равномерным шагом 1 по четырем направлениям;

$\Omega = \{\ rightnext(p), leftnext(p),\ abovenext(p), belownext(p)\}$ есть множество из четырех операций (функций), определенных на S. Ниже следуют аксиомы, определяющие эти операции.

$rightnext(p) = (x+1, y)$

$leftnext(p) = (x-1, y)$

$abovenext(p) = (x, y+1)$

$belownext(p) = (x, y-1)$

Аксиомы алгебры $\mathcal{U}$ выполняются (истинны, верны) для интерпретации $\mathcal{L}$. Алгебра $\mathcal{L}$ есть конкретная модель алгебры $\mathcal{U}$.

Пример 3. *Алгебра $\mathcal{T}$ линейных трансформаций*. Рассмотрим целочисленную матрицу $A = \begin{pmatrix} 1 & 2 & 4 \\ 3 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix}$. Пусть $\mathrm{CRS}(A) = \begin{pmatrix} 4 & 1 & 2 \\ 7 & 3 & 5 \\ 9 & 6 & 8 \end{pmatrix}$ есть операция циклического сдвига столбцов вправо (элементы столбца 1 ставятся на место столбца 2; элементы столбца 2 ставятся на место столбца 3; элементы столбца 3 ставятся на место столбца 1.

Пусть CBS есть операция циклического сдвига строк против часовой стрелки (элементы строки 1 ставятся на место строки 2 и т.д.). Пусть CLS есть операция циклического сдвига столбцов влево, и CAS есть операция циклического сдвига строк по часовой стрелке.

$$\mathrm{CLS}(A) = \begin{pmatrix} 2 & 4 & 1 \\ 5 & 7 & 3 \\ 8 & 9 & 6 \end{pmatrix},\ \mathrm{CAS}(A) = \begin{pmatrix} 3 & 5 & 7 \\ 6 & 8 & 9 \\ 1 & 2 & 4 \end{pmatrix},\ \mathrm{CBS}(A) = \begin{pmatrix} 6 & 8 & 9 \\ 1 & 2 & 4 \\ 3 & 5 & 7 \end{pmatrix}.$$

Аксиомы алгебры $\mathcal{U}$ справедливы для этих операций, примененных к матрице A. Аксиомы верны, если операции применяются к любой $n\times n$ матрице. Алгебра $\mathcal{T}$ для $n\times n$ матриц с этими четырьмя операциями есть конкретная модель алгебры $\mathcal{U}$.

Пример 4. *Алгебра $\mathcal{S}$ стрингов*. Рассмотрим множество S стрингов над некоторым алфавитом. Определим операции CROT и CLOT как циклический сдвиг вправо и циклический сдвиг влево знаков стринга. Например,

CROT(*abac*)=(*caba*) и CLOT(*abac*)=(*baca*). Две другие операции на множестве S есть тождества LID и RID. Непосредственно проверяется истинность аксиом алгебры $\mathcal{U}$ относительно алгебры $\mathcal{S}$ в качестве интерпретации.

Пример 5. Рассмотрим файл записей, содержащий некоторую информацию. Три понятия для абстрагирования есть *file*, *record*, *information*. Выбираем для них типы (сорта) *file*, *record*, *infor*. Рассмотрим над ними следующие операции.

insert: добавить запись в файл,
delete: удалить запись из файла,
trash: чистить файл,
update: обновить информацию в записи.

Трехсортная алгебра есть (S, Ω), где

$S = \{file, record, infor\}$,

$\Omega = \{ \ \{insert, delete\}\colon file \times record \to file;$
$\{update\}\colon file \times record \times infor \to file;$
$\{trash\}\colon file \to file \ \}$

Четыре операции из Ω сгруппированы по местности функций (схем). Алгебра записей есть интерпретация алгебры $\mathcal{U}$.

Пример 6. Рассмотрим множество $\mathbb{R}^2$ двумерных векторов на множестве вещественных чисел $\mathbb{R}$ с операцией + (векторное сложение), определяемое как $(x,y)+(a,b) = (x+a,\ y+b)$. Операция + коммутативна и ассоциативна. Вектор (0,0) есть единичный элемент и $(-x,-y)$ есть обратный для вектора (x,y). Множество $\mathbb{R}^2$ векторов со сложением есть аддитивная группа. Определим функцию $f:\mathbb{R}^2 \to \mathbb{R}^2$ как $f(x,y)=(Ax+By,\ Cx+Dy)$, где A,B,C,D есть константы. Отображение f есть гомоморфизм. В самом деле, $f((x,y)+(x',y'))$ $= f(x,y)+f(x',y')$. Так как $f(0,0)=(0,0)$, то единица переходит в себя. Так как $f(-x,-y) = (-Ax-By,\ -Cx-Dy) = -(Ax+By, Cx+Dy) = -f(x,y)$, то f отображает обратный для (x,y) в обратный для $f(x,y)$. Гомоморфизм сохраняет коммутативность и ассоциативность. Заметим, что f есть линейное преобразование $f(x,y) = \begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} Ax+By \\ Cx+Dy \end{pmatrix}$.

Пример 7. Рассмотрим множество $X=\{0,1,\ldots,2^k-1\}$ со сложением по модулю 2^k и множество Y k-битовых бинарных чисел со сложением по модулю 2. Определим отображение $f\colon Y \to X$, где f отображает бинарное число в им представляемое натуральное число. Для двух бинарных чисел $b_1 = b_2$, $f(b_1) = f(b_2)$. Для каждого числа из X существует только одно бинарное

число. Поэтому *f* есть изоморфизм.

Пример 8. Рассмотрим алгебру $\mathcal{U}$ с операторами *right, left, above, below* и алгебру $\mathcal{T}$ с операторами CRS, CLS, CAS, CBS, определенными в примере 1. Определим гомоморфизм $f: \mathcal{U} \rightarrow \mathcal{T}$, который отображает элемент алгебры $\mathcal{U}$ в $n \times n$ матрицу алгебры $\mathcal{T}$ и отображает операторы следующим образом:

$f(right)$ = CRS; $f(left)$ = CLS; $f(above)$ = CAS; $f(below)$ = CBS.

Терм вида $above(x)$ в $\mathcal{U}$ отображается в терм $f(above)(f(x))$ = CAS(m) в алгебре $\mathcal{T}$. Аксиомы отображаются соответственно. Например, аксиома $above(below(x)) = x = below(above(x))$ отображается в CAS(CBS(m)) = m = CBS(CAS(m)). Построенное отображение есть гомоморфизм (и даже изоморфизм). Аналогично, мы можем определить гомоморфизм из $\mathcal{U}$ в $\mathcal{L}$.

Пример 9. Рассмотрим алгебру $\mathcal{S} = (S_1, \Omega_1)$ и алгебру $\mathcal{C} = (S_2, \Omega_2)$, где

$S_1 = \{Stack, Elm\}$

$\Omega_1 = \{\{push\} : Stack \times Elm \rightarrow Stack$

$\{pop\} : Stack \rightarrow Stack$

$\{top\} : Stack \rightarrow Elm$

$\{newstack\} : \rightarrow Stack\}$

$S_2 = \{Container, Item\}$

$\Omega_2 = \{\{put\} : Container \times Item \rightarrow Container$

$\{get\} : Container \rightarrow Item$

$\{newc\} : \rightarrow Container\}$

Определим отображение $F : S_2 \rightarrow S_1$, так что $F(Container) = Stack$, $F(Item) = Elm$, $F(put) = push$, $F(newc) = newstack$ и $F(get)=top$. Отображение F есть гомоморфизм из $\mathcal{C}$ в $\mathcal{S}$. Например, F отображает терм *put*(*newc, item*) в *push*(*newstack, elm*).

Полагая, что *Elm* соответствует *Nat*, сорту натуральных чисел, примем, что

1) *newstack* есть пустой стек;

2) *push*(*push*(*newstack,zero*), *succ*(*zero*)) есть стек с двумя элементами, с элементом *succ(zero)* наверху стека;

3) *top*(*pop*(*push*(*push*(*newstack,zero*), *succ*(*succ*(*zero*))))) есть стек с верхним элементом *zero*.

1.3. Спецификация

Сигнатура абстрактного типа данных есть алгебра (S, Ω), включающая сорта, операторы (операции, функции), переменные для сортов.

Спецификация абстрактного типа данных есть сигнатура вместе с множеством *аксиом*. Аксиомы характеризуют свойства типа данных с помощью многосортного логического формализма. Обычно этот формализм ограничен логикой первого порядка с равенством.

Имя спецификации должно быть уникальным. Имена сортов, ассоциированных с формальными параметрами, если они есть, должны быть

специфицированы. Сорт, определенный в спецификации может иметь то же самое имя, что и спецификация. Имя сорта может быть и другим. Операторы могут быть частично или всюду определенные. Переменные в аксиомах связаны кванторами общности. Алгебра, указанная в спецификации, *удовлетворяет* аксиомам в спецификации, если для всякой аксиомы $t_1 = t_2$ два терма t_1 и t_2 обозначают один и тот же элемент алгебры для каждых возможных значений переменных в этих термах. Например, два терма

push(newstack, succ(zero)) и *push(pop(push(newstack, zero)), succ(zero))* равны.

Проиллюстрируем редукцию алгебраических равенств к некоторой канонической форме, используя аксиомы как правила вывода.

Пример 9. Выражение с переменными сводится к выражению, свободному от переменных, используя аксиомы как правила вывода (as rewrite rules). В выводе подвыражение f в выражении e сравнивается (is matched) с левой частью t_1 аксиомы $t_1 = t_2$, и если сравнение верно, то f замещается на правую часть t_2. Рассмотрим редукцию терма в спецификации абстрактного типа данных, определяющих сложение натуральных чисел.

1. Для упрощения выражения *add(succ(succ(zero)), succ(succ(a)))*, выполняются следующие шаги.

Аксиомы.

A1. *add(zero,v) = v*;

A2. *add(succ(u),v) = add(u,succ(v))*;

Запросы.

C1. *add(succ(succ(zero)), succ(succ(a)))*.

C2. *add(succ(zero), succ(succ(succ(a))))*.

C3. *add(zero, succ(succ(succ(succ(a)))))*.

C4. *succ(succ(succ(succ(a))))*.

Протокол вычислений.

1. Согл(C1, A1). Согласование (matching) безуспешно (противоречиво) по первому аргументу левой части A1. Проводится попытка согласования C1 со второй аксиомой A2.

2. Согл(C1, A2). Присваивание

u := *succ(zero)*, *v* := *succ(succ(a))*

успешно (непротиворечиво). Формируется новый запрос C2, который есть правая часть A2 при найденных значениях переменных. Проводится попытка согласования C2 с аксиомой A1.

3. Согл(C2, A1). Согласование безуспешно (противоречиво) по первому аргументу левой части A1. Проводится попытка согласования C2 с аксиомой A2.

4. Согл(C2, A2). Присваивание

u := *zero*, *v* := *succ(succ(succ(a)))*

успешно (непротиворечиво). Формируется новый запрос C3, который есть правая часть A2 при найденных значениях переменных. Проводится попытка согласования C3 с аксиомой A1.

5. Согл(C3, A1). Присваивание

v := *succ(succ(succ(succ(a))))*
успешно (непротиворечиво). Формируется новый запрос С4, который есть правая часть А1 при найденных значениях переменных. Проводится попытка согласования С4 с аксиомой А1.

6. Согл(С4, А1). Согласование безуспешно (противоречиво) по несовпадению первых имен функций в С4 и левой части А1. Проводится попытка согласования С4 с аксиомой А2.

7. Согл(С4, А2). Согласование безуспешно (противоречиво) по несовпадению первых имен функций в С4 и левой части А2.

Других аксиом для согласования нет. С4 = *succ(succ(succ(succ(a))))* есть результат работы транслятора OBJ3 над выражением С1. Транслятор заканчивает работу и останавливается.

2. Можно проверить, что в арифиетике со сложением и умножением выражения *mult(succ(succ(0)),a)* и *add(add(0,a),a)* эквивалентны.

Аксиомы .

А1. *add(zero,v) = v*;

А2. *add(succ(u),v) = add(u,succ(v))*;

А3. *mult(zero,v) = zero*;

А4. *mult(succ(u),v) = add(mult(u,v),v)*.

Запросы для *mult(succ(succ(0)),a)*.

С1. *mult(succ(succ(0)),a)*.

С2. *add(mult(succ(0),a),a)*.

С3. *add*(*add*(*mult(0,a),a*),*a*).

С4. *add*(*add(0,a)*,*a*).

С5. *add*(*a*,*a*).

Протокол вычислений для *add(succ(succ(zero)*.

1. Согл(С1, А1). Согласование безуспешно (противоречиво) по первому имени функций в С1 и А1. Проводится попытка согласования С1 со второй аксиомой А2.

2. Согл(С1, А2). Согласование безуспешно (противоречиво) по первому имени функций в С1 и А2. Проводится попытка согласования С1 с аксиомой А3.

3. Согл(С1, А3). Согласование безуспешно (противоречиво) по первому аргументу в С1 и А3. Проводится попытка согласования С1 с аксиомой А4.

4. Согл(С1, А4). Присваивание
u := *succ(0)*, *v* := *a*
успешно (непротиворечиво). Формируется новый запрос С2, который есть правая часть А4 при найденных значениях переменных. Проводится попытка согласования С2 с аксиомой А1.

5. Согласования С2 с аксиомами А1, А2, А3 невозможны. Проводится попытка согласования С2 с аксиомой А4.

6. Согл(С2, А4) по терму *mult(succ(0),a)* в С2. Присваивание
u := 0, *v* := *a*
успешно (непротиворечиво). Формируется новый запрос С3, который есть правая часть А4 при найденных значениях переменных, причем эта часть

add(mult(0,a),a)) вставляется в C2 вместо *mult(succ(0),a)*. Проводится попытка согласования C3 с аксиомой A1.

7. Согласования C3 с аксиомами A1,A2,A4 невозможны. Проводится попытка согласования C3 с аксиомой A3.

8. Согл(C3, A3) по терму *mult(0,a)* в C3. Присваивание

$u := 0,\ v := a$

успешно (непротиворечиво). Формируется новый запрос C4, который есть правая часть A3 при найденных значениях переменных, причем эта часть 0 вставляется в C3 вместо *mult(0,a)*. Проводится попытка согласования C4 с аксиомой A1.

9. Согл(C4, A1) по терму *add(0,a)* в C4. Присваивание

$v := a$

успешно (непротиворечиво). Формируется новый запрос C5, который есть правая часть A1 при найденных значениях переменных, причем эта часть *a* вставляется в C4 вместо *add(0,a)*. Проводится попытка согласования C5 с аксиомой A1.

10. Согласования C5 с аксиомами A1, A2, A3, A4 невозможны. Транслятор выдает C5 = *add(a,a)* и прекращает работу.

Протокол вычислений для терма *add(add(0,a),a)* выдаст *add(a,a)*.

В результате получили, что термы *mult(succ(succ(0)),a)* и *add(add(0,a),a)* равны одному и тому же терму *add(a,a)* и потому эквивалентны.

2. АЛГОРИТМИЧЕСКИЙ ЯЗЫК ПРОГРАММИРОВАНИЯ OBJ3 (аксиоматический язык алгебраических спецификаций)

OBJ3 есть функциональный язык программирования широкого спектра, который базируется на порядковой сортовой (order-sorted) логике равенства. Система OBJ3 состоит из языка спецификаций и интерпретатора.

Дизайн OBJ3 позволяет расширительное и модульное развитие (incremental and modular development) спецификаций, которые исполнимы, повторно используемы и компонуемы.Эта цель достигается наличием трех программных единиц (entity) языка: *объект* (*object*), *теория* (*theory*), *вью* (*view, проекция*). Объект инкапсулирует исполняемый код. Теория определяет свойства, которые могут удовлетворяться другой теорией или объектом. *Модуль* есть объект или теория. Программная единица вью обеспечивает связь (соответствие, binding) между теорией и модулем. Работа интерпретатора системы OBJ3 основана на выводе термов (*term rewriting*). Правила вывода есть аксиомы программы. Каждая аксиома есть равенство термов.

2.1. Базисный синтаксис OBJ3

Основная программная единица в OBJ3 есть *объект* (*object*), который инкапсулирует исполняемый код (executable code). Синтаксически, определение объекта начинается с ключевого слова `obj` и заканчивается

ключевым словом endo. Идентификатор объекта появляется сразу после ключевого слова obj. Ключевое слово is следует за именем. Затем следует тело объекта.

Все следующие примеры программ написаны для версии BOBJ (behaviour OBJ) языка программирования OBJ3. Программу и запросы к ней следует написать в блокноте, а потом перенести в BOBJ. Если система BOBJ отключится, загрузить ее снова и загрузить программу по частям. Затем загрузить запросы и получить ответы.

2.1.1. Спецификация объекта

Спецификация объекта включает пять следующих определений.

1. *Оъявление сортов.* Объект (*module*) с именем FLAVORS и сорт (тип) с именем Flavor вводятся в первой строке. Имя объекта пишется большими буквами. Имя сорта начинается с большой буквы. Объявление объекта

```
obj NUMBER is sorts Nat Rat .
```

вводит о объект NUMBER и два сорта с именами Nat и Rat. Пробелы (перед последней точкой тоже) обязательны.

2. *Оъявление операторов (функций)*. Операторы могут определяться тремя следующими способами (styles).

а) *функция* (стандартная префиксная форма записи функции):

```
op first : Flavor Flavor -> Flavor .
```

б) *mixfix операция* (инфиксная форма записи функции).

```
op _second_ : Flavor Flavor -> Flavor .
```

Символ подчеркивания есть место для аргумента, как, например, в выражении X second Y. Определение

```
op __ : Bit Bits -> Bits .
```

используется для создания бит стринга. Результат применения этой операции к паре стрингов 1 011 есть стринг 1011.

в) *константы*. Константа есть операция местности (arity) 0. Например,

```
ops Chocolate Vanilla Strawberry : -> Flavor .
```

Ключевое слово ops используется для введения более чем одной операции. Для разделения операций в сложных случаях удобно использовать скобки. Например,

```
ops (_+_) (_-_) : IntExp IntExp -> IntExp .
```

3. *Оъявление переменных*. Переменная объявляется ключевым словом var. Более одной переменной одного и того же сорта можно ввести ключевым словом vars:

```
vars X Y : Flavor .
```

Имена переменных начинаются с большой буквы.

4. *Оъявление аксиом*. Аксиома есть равенство двух термов. Например,

```
eq first (X, Y) = X .
eq X second Y = Y .
```

5. *Конец спецификации*. Ключевое слово endo маркирует конец спецификации объекта. После слова endo точки нет.

Пример объекта FLAVORS.

```
obj FLAVORS is sort  Flavor .
  op first : Flavor Flavor -> Flavor .
  op _second_ : Flavor Flavor -> Flavor .
  ops Chocolate Vanilla Strawberry : -> Flavor .
  vars X Y : Flavor .
  eq first(X,Y) = X .
  eq X second Y = Y .
endo
```

Ниже следуют запросы к программе.

```
red first(X,Y) .
red X second Y .
red first(Chocolate,Y) .
red first(Vanilla,Y) .
red Strawberry second Strawberry .
red Chocolate second Strawberry .
```

2.1.2. Сорта и подсорта

OBJ3 базируется на строгом сортировании. Каждый символ имеет сорт или ассоциируется с сортом. Сорта вводятся в OBJ3 синтаксисом

sorts <*SortIds*>, например, как в

```
sorts Nat Int .
```

В OBJ3 действует система подсортов, которая поддерживает обработку частичных операторов, кратную наследственность (multiple inheritance) и обработку ошибок (error handling). OBJ3 можно использовать для формальной спецификации иерархии объектно ориентированных программных компонент.

Сорт (множество) s′ есть подсорт (подмножество) сорта s, запись s′< s верна, если домен для s включает домен для s′ и операторы для s доступны для s′. Подсорт в OBJ3 объявляется командой

```
subsort s′ < s .
```

Подмножество частичного упорядочения может быть установлено между локально определенными и импортированными сортами. Например,

```
subsort MyInt < MyRat < MyReal .
```

где MyInt есть подмножество в MyRat, а MyRat есть подмножество в MyReal. Всякая операция, определенная для сорта MyReal доступна для переменных из сортов Myint и MyRat. Следующие примеры иллюстрируют отношение быть подсортом некоторого сорта.

1. Сорт NzNat положительных целых чисел есть подсорт сорта Nat натуральных (неотрицательных целых) чисел.

```
subsort NzNat < Nat .
```

2. Непустой список есть подсорт списка.

```
subsort NeList < List .
```

3. Ограниченный стек есть подсорт сорта стек.

```
subsort BStack < Stack .
```

4. Непустое дерево натуральных чисел есть подсорт сорта дерева

натуральных чисел.

```
subsort NeNatTree < NatTree .
```

Подсортность (Subsorting) гарантирует корректное применение функций к переменным соответствующих подсортов и индуцирует процесс работы с исключительными операциями должным образом. Примеры следуют ниже.

1. Оператор деления определен лишь для `NzNat`; таким образом, мы избежим деления на 0.

2. Операция `head`, в списке может быть ограничена непустым списком.

3. Операция `size` для стека имеет смысл лишь для ограниченного стека `Bstack`, подсорта сорта `Stack`.

4. Операции *left, right, content*, *isfound*, определенные для сорта `NatTree` могут быть переопределены на ограниченный сорт `NeNatTree`, характеризующий непустоту деревьев натуральных чисел.

Упорядоченная сортовая алгебра поддерживает также кратную наследственность, так что подсорт может иметь более чем один надсорт.

2.1.3. Расширение многосортных спецификаций (импорт модулей)

OBJ3 имеет четыре способа импорта модулей: `protecting` (защита), `extending` (расширение), `using` (использование), including (включение). Вместо ключевых слов `protecting, extending, using, including` можно использовать их аббревиатуры `pr, ex, us, inc`. Если модуль *X* импортирует модуль *Y*, который в свою очередь импортирует модуль *Z*, то модуль *X* тоже импортирует модуль *Z*, то есть отношение *импортирует* (*imports*) транзитивно.

1. Способ импорта `protecting` осуществляется следующим образом.

```
obj NATTREE is sorts Nebtree Btree .
   protecting NAT .
   protecting BOOL .
   subsorts Nat < Nebtree < Btree .
        ...
  endo
```

Если модуль *X* импортирует модуль *Y*, то модуль *Y* защищен. Никакие новые элементы сортов из модуля *Y* не могут быть определены в этом модуле. Сигнатура модуля *Y* не может быть изменена, не могут быть введены никакие новые операции с сортами модуля *Y* . Уже определенная в модуле *Y* функция не может быть переопределена. Однако сигнатура модуля *Y* может быть использована в определении операций в *X*.

2. Способ импорта `extending` осуществляется, например, в модуле `ORDLIST` следующим образом.

```
obj ORDLIST is sort List .
      extending LIST .
      op insert : List Nat -> List .
      vars I J : Nat. var L : List.
      eq insert(null, I) = (I null) .
      cq insert(I L, J) = if I > J then (J I L)
                                 else (I insert(L, J)) .
```

```
endo
```

Если модуль *X* импортирует модуль *Y* и модуль *Y* расширяется, то новые пункты данных сорта из *Y* могут быть определены в модуле *X*. Однако операции в модуле *X* не переопределяют никакую функцию, уже определенную в модуле *Y*. Из этого следует, что новые операции могут быть добавлены, чтобы расширить поведение модуля *Y* в модуле *X*.

3. Способ импорта `using` осуществляется следующим образом.

```
obj X is using Y .
   ...
endo
```

Если модуль *X* импортирует модуль *Y* способом `using`, то нет гарантии в том смысле, что новые (data) элементы данных сортов из модуля *Y* могут быть созданы, как и старые (data) элементы данных сортов из модуля *Y* могут быть переопределены. Этот способ импорта аналогичен повторному использованию (to code reuse) в объектно ориентированной парадигме. OBJ3 имплементирует использование копированием импортируемых модульных верхнее уровневых структур, делящих (sharing) все из модулей (all of the modules), которые это импортирует. Поэтому от сортов, определяемых данным модулем требуется иметь различные имена, и все копируемые операции должны быть уникально (uniquely) идентифицированы их именем и рангом.

4. Способ импорта `including` осуществляется следующим образом.

```
obj X is including Y .
  ...
endo
```

Если модуль *X* включает модуль *Y*, то модуль *Y* инкорпорируется в модуль *X* без копирования. В этом единственная разница между способами `using` и `including`.

Важно заметить, что OBJ3 не проверяет, корректен ли импорт в программе, написанной пользователем. Последствия некорректного объявления импорта может вести к неполным редукциям в одних случаях и недостаточные редукции в других.

2.1.3.1. Программная спецификация `NATURAL` *сложения и умножения унарных натуральных чисел*

```
obj NATURAL is sort Natural .
  op zero : -> Natural .
  op succ : Natural -> Natural .
  op add : Natural Natural -> Natural .
  op mult : Natural Natural -> Natural .
  vars A X : Natural .
  eq add(zero, A) = A .
  eq add(A, zero) = A .
  eq add(succ(A), X) = succ(add(A, X)) .
  eq mult(zero, A) = zero .
  eq mult(A, zero) = zero .
  eq mult(succ(X), A) = add(mult(X, A), A) .
endo
```

Ниже следуют запросы к программе.

```
red mult(succ(succ(succ(zero))),succ(succ(zero))) .
red add(succ(succ(succ(zero))), succ(succ(zero))) .
let n1 = succ(succ(succ(zero))) .
let n2 = succ(succ(zero)) .
red mult(n1,n2) .
red add(n1,n2) .
```

2.1.3.2. Программная спецификация BOOLEAN

```
obj BOOLEAN is sort Boolean .
  op TRUE : -> Boolean .
  op FALSE : -> Boolean .
  op NOT : Boolean -> Boolean .
  op _AND_ : Boolean Boolean -> Boolean .
  op _OR_ : Boolean Boolean -> Boolean .
  op _IMPL_ : Boolean Boolean -> Boolean .
  op _EQ_ : Boolean Boolean -> Boolean .
  vars X Y : Boolean .
  eq NOT(TRUE) = FALSE .
  eq NOT(FALSE) = TRUE .
  eq FALSE AND X = FALSE .
  eq TRUE AND X = X .
  eq TRUE OR X = TRUE .
  eq FALSE OR X = X .
  eq X OR X = X .
  eq X OR Y = Y OR X .
  eq X AND X = X .
  eq X AND Y = NOT(NOT(X) OR NOT(Y)) .
  eq X IMPL Y = NOT(X) OR Y .
  eq X EQ Y = (X IMPL Y) AND (Y IMPL X) .
endo
```

Ниже следуют запросы к программе.

```
red TRUE AND X .
red FALSE OR X .
red X AND X .
red X OR X .
red X OR Y .
```

2.1.3.3. Программная спецификация ORDEREDNATURAL *упорядоченных унарных натуральных чисел*

```
obj ORDEREDNATURAL is sort Orderednatural .
  protecting NATURAL .
  protecting BOOLEAN .
  op EQ : Natural Natural -> Boolean .
  op LT : Natural Natural -> Boolean .
  op LE : Natural Natural -> Boolean .
  vars X Y Z : Natural .
  eq EQ(zero, zero) = TRUE .
  eq EQ(zero, succ(X)) = FALSE .
  eq EQ(succ(X), zero) = FALSE .
  eq EQ(succ(X), succ(Y)) = EQ(X, Y) .
  eq LT(zero, zero) = FALSE .
  eq LT(zero, succ(X)) = TRUE .
```

```
  eq LT(succ(X), succ(Y)) = LT(X, Y) .
  eq LE(X, Y) = EQ(X, Y) OR LT(X, Y) .
endo

red EQ(zero, zero) .
red EQ(zero, succ(X)) .
red EQ(succ(zero), succ(zero)) .
red EQ(succ(zero), succ(succ(succ(Y)))) .
red LT(zero, zero) .
red LT(zero, succ(X)) .
red LT(succ(zero), succ(zero)) .
red LE(succ(zero), succ(zero)) .
red LE(zero, succ(zero)) .
```

2.1.3.4. Программная спецификация NAT1 *сложения унарных натуральных чисел*

```
obj NAT1 is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat . vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
endo
red s(s(s(0))) + s(s(0)) .
```

2.1.3.5. Программная спецификация NAT2 *сложения и умножения унарных натуральных чисел (импорт* NAT1 *в* NAT2 *методом* protecting)

```
obj NAT2 is
  protecting NAT1 .
  vars M N : Nat .
  eq N * 0 = 0 .
  eq 0 * N = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
endo
red s(s(s(0))) + s(s(0)) .
red s(s(s(0))) * s(s(0)) .
```

2.1.3.6. Программная спецификация NAT3 *сложения, умножения, порядка натуральных чисел*

(импорт NAT2 *в* NAT3 *методом* protecting)

```
obj NAT3 is
  protecting NAT2 .
  vars M N : Nat .
  var N' : NzNat .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
endo
red s(s(s(0))) + s(s(0)) .
red s(s(s(0))) * s(s(0)) .
red s(s(0)) > s(s(s(0))) .
red s(s(s(0))) > s(s(0)) .
```

2.1.3.7. Программная спецификация NAT4 *сложения, умножения, порядка, нахождения частного и НОД унарных натуральных чисел (импорт* NAT3 *в* NAT4 *методом* protecting)

```
obj NAT4 is
  protecting NAT3 .
  vars M N : Nat .
  var M' N' : NzNat .
  eq d(0,N) = N .
  eq d(s N, s M) = d(N,M) .
  eq quot(N,M') = if ((N > M') or (N == M'))
                    then s quot(d(N,M'),M')
                    else 0
                  fi .
  eq gcd(N',M') = if N' == M'
                     then N'
                     else (if N' > M'
                              then  gcd(d(N',M'),M')
                              else gcd(N',d(N',M'))
                           fi)
                   fi .
endo

red s(s(s(0))) + s(s(0)) .
red s(s(s(0))) * s(s(0)) .
red s(s(0)) > s(s(s(0))) .
red s(s(s(0))) > s(s(0)) .
red quot(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))),
         s(s(s(0)))) .
red gcd( s(s(s(s(s(s(0)))))), s(s(s(s(0)))) ) .
```

2.1.3.8. Программная спецификация NAT *сложения, умножения, порядка, нахождения частного и НОД унарных натуральных чисел*

```
obj NAT is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
```

```
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat . vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq 0 * N = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0,N) = N .
  eq d(s N, s M) = d(N,M) .
  eq quot(N,M') = if ((N > M') or (N == M'))
                    then s quot(d(N,M'),M')
                    else 0
                  fi .
  eq gcd(N',M') = if N' == M'
                     then N'
                     else (if N' > M'
                              then  gcd(d(N',M'),M')
                              else gcd(N',d(N',M'))
                           fi)
                  fi .
endo

red quot( s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))),
          s(s(s(0))) ) .
red gcd( s(s(s(s(s(0))))), s(s(s(0))) ) .
```

2.1.4. Встроенные сорта

Система OBJ3 включает библиотеку, содержащую несколько встроенных сортов следующих часто используемых абстрактных типов данных.

Встроенная теория `TRUTH-VALUE`

`TRUTH-VALUE` выдает константу `truth` как значения `true` и `false`.

```
dth TRUTH-VALUE is
   sort Bool .
   op true : -> Bool  .
   op false : -> Bool .
end
```

Встроенная теория `TRUTH`

`TRUTH` есть `TRUTH-VALUE` с операторами `==`, `=/=`, `if _ then _ else _`

fi. Инфиксный полиморфный оператор `_==_` применим к любому сорту. Сведением термов к нормальной форме и сравнением нормальных форм на синтаксическое (посимвольное) равенство оператор `_==_` проверяет, равны ли синтаксически два основных (без переменных) терма. Если оператор `_==_` применяется к двум термам несравнимых сортов, то парсинг выдает ошибку. Для термов сорта `Bool` оператор `_==_` превращается в `_iff_`. Операторы `=/=`, `if _ then _ else _ fi` обеспечиваются модулем `BOOL` и тоже полиморфны.

```
  dth TRUTH is
     protecting TRUTH-VALUE .
     sort Universal .
     subsorts Bool < Universal
     vars X Y : Universal .
     var B : Bool .
     ops true false : -> Bool .
     op _ == _ : Universal Universal -> Bool .
     op _ =/= _ : Universal Universal -> Bool .
     op if _ then _ else _ fi : Bool Universal Universal -> 
Universal .
     eq X == X = true .
     eq if true then X else Y fi = X .
     eq if false then X else Y fi = Y .
     eq if B then X else X fi = X .
  end
```

Встроенная теория `BOOL`

`BOOL` выдает булевы инфиксные операторы `and`, `or`, `xor`, префиксный `not`, инфиксный `implies`. Поддерживаются операции из `TRUTH`. Квотированные идентификаторы не поддерживаются. Операции `==`, `=/=`, `if_then_else_fi` поддерживаются. Они полиморфны, то есть применимы к любым объявленным сортам.

```
  dth BOOL is
     protecting TRUTH .
     op _ and _ : Bool Bool -> Bool [assoc comm prec 20] .
     op _ or _ : Bool Bool -> Bool [assoc comm prec 30] .
     op _ xor _ : Bool Bool -> Bool [assoc comm ] .
     op _ implies _ : Bool Bool -> Bool   .
     op not _ : Bool -> Bool [prec 10] .
     eq not true = false .
     eq not false = true .
     eq true and B = B .
     eq false and B = false .
     eq true or B = true .
     eq false or B = B .
     eq B1 implies B2 = (not B1) or B2 .
  end
```

Встроенная теория `IDENTICAL`

Теорию `IDENTICAL` можно использовать вместо `BOOL`. Операции `===`, `=/==` проверяют посимвольное (графическое) равенство термов без их

вычисления.

```
th IDENTICAL is
   protecting TRUTH .
   op _===_ : Universal Universal -> Bool .
   op _=/==_ : Universal Universal -> Bool .
end
```

Встроенная теория `NAT`

`NAT` выдает натуральные числа.

```
dth NAT is
   protecting BOOL .
   sorts Nat Zero NzNat .
   subsorts NzNat Zero < Nat
   vars N M : Nat .
   op _ + _ : NzNat NzNat -> NzNat [assoc comm ] .
   op s _ : NzNat -> NzNat [prec 15] .
   op 0 : -> Zero [prec 0] .
   op s _ : Nat -> NzNat [prec 15] .
   op p _ : NzNat -> Nat [prec 15] .
   op _ > _ : Nat Nat -> Bool [prec 41] .
   op _ < _ : Nat Nat -> Bool [prec 41] .
   op _ <= _ : Nat Nat -> Bool [prec 41] .
   op _ >= _ : Nat Nat -> Bool [prec 41] .
   op _ + _ : Nat Nat -> Nat [assoc comm ] .
   op _ * _ : Nat Nat -> Nat [assoc comm prec 30] .
   op _ div _ : Nat Nat -> Bool [prec 30] .
   op eq : Nat Nat -> Bool [prec 0] .
   eq 0 > N = false .
   eq (s N) > 0 = true .
   eq (s N) > (s M) = N > M .
   eq eq(N , N) = true .
   eq eq(0 , s N) = false .
   eq eq(s N , 0) = false .
   eq eq(s N , s M) = eq(N , M) .
   cq eq(N , M) = false if (N < M) or (M < N) .
   eq M < 0 = false .
   eq 0 < (s N) = true .
   eq (s N) < (s M) = N < M .
   eq (s M) <= N = M < N .
   eq N <= M = eq(N , M) or (N < M) .
   eq (s M) > 0 = true .
   eq N >= 0 = true .
   eq (s N) >= (s M) = N >= M .
   eq 0 >= (s N) = false .
   eq N >= N = true .
end
```

Встроенная теория `obj NZNAT`

`NZNAT` выдает ненулевые натуральные числа. Встроенная теория `NZNAT` в BOBJ отсутствует.

Встроенная теория `INT`

`INT` выдает целые числа.

```
dth INT is
```

```
   protecting NAT .
   sort Int NzInt Nat Zero NzNat .
   subsorts Nat NzNat Zero NzInt < Int
   subsorts NzNat Zero < Nat
   subsorts NzNat < NzInt
   vars I J K : Int .
   op - _ : Int -> Int [prec 20] .
   op s _ : Int -> Int [prec 15] .
   op p _ : Int -> Int [prec 20] .
   op - _ : NzInt -> NzInt [prec 20] .
   op _ + _ : Int Int -> Int [assoc comm prec 40] .
   op _ - _ : Int Int -> Int [assoc prec 20] .
   op _ * _ : Int Int -> Int [assoc prec 30] .
   op _ < _ : Int Int -> Bool .
   op _ <= _ : Int Int -> Bool .
   op _ > _ : Int Int -> Bool .
   op _ >= _ : Int Int -> Bool .
   op _ quo _ : Int Int -> Int .
   op _ rem _ : Int Int -> Int .
   op _ divides _ : Int Int -> Int .
   eq s (p I) = I .
   eq p (s I) = I .
   eq I + 0 = I .
   eq I + (s J) = s (I + J) .
   eq I + (p J) = p (I + J) .
   eq I * 0 = 0 .
   eq I * (s J) = (I * J) + I .
   eq I * (p J) = (I * J) - I .
   eq I * (J + K) = (I * J) + (I * K) .
   eq - (- I) = I .
   eq - (s I) = p (- I) .
   eq - (p I) = s (- I) .
   eq I - J = I + (- J) .
   eq I + (- I) = 0 .
   eq - (I + J) = (- I) - J .
   eq I * (- J) = - (I * J) .
end
```

Встроенная теория `RAT`

`RAT` выдает рациональные числа. Встроенная теория `RAT` в BOBJ отсутствует.

Встроенная теория `FLOAT`

Теория `FLOAT` выдает числа с плавающей точкой.

```
dth FLOAT is
   sort Float .
   op not _ : Bool -> Bool  [ prec 10 ]  .
   op _ + _ : Float Float -> Float  [ assoc comm prec 40 ]  .
   op _ - _ : Float Float -> Float  [ assoc prec 40 ]  .
   op _ * _ : Float Float -> Float  [ assoc prec 30 ]  .
   op _ / _ : Float Float -> Float  [ assoc prec 30 ]  .
   op _ < _ : Float Float -> Bool .
   op _ <= _ : Float Float -> Bool .
   op _ > _ : Float Float -> Bool .
```

```
    op _ >= _ : Float Float -> Bool .
    op exp : Float -> Float .
    op log : Float -> Float .
    op sqrt : Float -> Float .
    op abs : Float -> Float .
    op sin : Float -> Float .
    op cos : Float -> Float .
    op atan : Float -> Float .
    op pi : -> Float .
    op - _ : Float -> Float  [ prec 20 ]  .
  end
```

Встроенная теория `ID`

`ID` выдает идентификаторы (identifiers, слова), включает лексикографическое упорядочение и все операции, доступные в `BOOL`. Встроенная теория `ID` в BOBJ отсутствует.

Встроенная теория `QID`

`QID` работает так же как `ID`. Различие с `ID` в том, что в `QID` идентификаторы (identifiers) начинаются с символа апостроф. Например, `'a`, `'b`, `'1300`, `'anyidentifier`. `QID` не имеет встроенных операций.

```
  dth QID is
    protecting BOOL .
    sort Id .
  end
```

Встроенная теория `QIDL`

`QIDL` выдает идентификаторы (identifiers, слова) с апострофами и включает все операции, доступные в `BOOL`. `QIDL` дополнительно включает лексикографическое упорядочение.

```
  dth QIDL is
  end
```

Встроенная теория `QIDL` в BOBJ отсутствует.

2.1.4.1. Программная спецификация `SIMPLESET-NAT` *простого множества натуральных чисел*

Модуль `SIMPLESET-NAT` защищает импортированные модули `NAT` и `BOOL`.

```
  obj SIMPLESET-NAT is sort Simpleset .
    protecting BOOL .
    protecting NAT .
    op empty : -> Simpleset .
    op insert : Nat Simpleset -> Simpleset .
    op member : Nat Simpleset -> Bool .
    vars S : Simpleset .
    vars N M : Nat .
    eq member(N, empty) = false .
    eq member(N, insert(M, S)) = (M == N) or member(N, S) .
    eq insert(N, insert(M, S)) = insert(M, insert(N, S)) .
  endo
  red insert(10, insert(11,empty)) .
```

```
red member(11, insert(10, insert(11,empty))) .
red member(15, insert(10, insert(11,empty))) .
let S = insert(10, insert(11,empty)) . red member(11,S) .
let S = insert(10, insert(11,empty)) . red member(15,S) .
let S = insert(10, insert(11,empty)) . red member(11,S) .
let S = insert(10, insert(11,empty)) . red member(15,S) .
```

2.1.4.2. Программная спецификация SIMPLESET-WORD *простого множества слов*

Модуль SIMPLESET-WORD защищает импортированные модули BOOL и QID.

```
obj SIMPLESET-WORD is sort Simpleset .
  protecting BOOL .
  protecting QID .
  op empty : -> Simpleset .
  op insert : Id Simpleset -> Simpleset .
  op member : Id Simpleset -> Bool .
  vars S : Simpleset .
  vars N M : Id .
  eq member(N, empty) = false .
  eq member(N, insert(M, S)) = (M == N) or member(N, S) .
  eq insert(N, insert(M, S)) = insert(M, insert(N, S)) .
endo

red insert('10, insert('11,empty)) .
red member('11, insert('10, insert('11,empty))) .
red member('15, insert('10, insert('11,empty))) .
let S = insert('10, insert('11,empty)) . red member('11,S) .
let S = insert('10, insert('11,empty)) . red member('15,S) .
let S = insert('10, insert('11,empty)) . red member('11,S) .
let S = insert('10, insert('11,empty)) . red member('15,S) .
```

2.1.4.3. Программная спецификация obj NATTREE *бинарного дерева натуральных чисел*

Модуль NATTREE защищает импортированные модули NAT и BOOL.

```
obj NATTREE is sort Tree .
  protecting NAT .
  protecting BOOL .
  op empty : -> Tree .
  op node : Tree Nat Tree -> Tree .
  op left : Tree -> Tree .
  op right : Tree -> Tree .
  op content : Tree -> Nat .
  op isempty : Tree -> Bool .
  op isfound : Tree Nat -> Bool .
  vars X Y : Tree .
  vars N M : Nat .
  eq isempty(empty) = true .
  eq isempty(node(X, N, Y)) = false .
  eq left(node(X, N, Y)) = X .
  eq right(node(X, N, Y)) = Y .
  eq content(node(X, N, Y)) = N .
```

```
  eq isfound(empty, M) = false .
  eq isfound(node(X, N, Y), M) =
            (M == N) or (isfound(X, M) or isfound(Y, M)) .
endo

let n5 = node(empty, 3,  empty) .
let n6 = node(empty, 7, empty) .
let n3 = node(n5, 10, n6) .
let n4 = node(empty, 15, empty) .
let n1 = node(empty, 1, empty) .
let n2 = node(n3, 1, n4) .
let n0 = node(n1, 4, n2) .
red n0 .
red content(n0) .
red left(n0) .
red right(n0) .
red isempty(n2) .
red isfound(n0, 3) .
red isfound(n3, 10) .
```

2.1.4.4. Программная спецификация `obj WORDTREE` *бинарного дерева слов*

```
obj WORDTREE is sort Tree .
  protecting BOOL .
  protecting QID .
  op empty : -> Tree .
  op node : Tree Id Tree -> Tree .
  op left : Tree -> Tree .
  op right : Tree -> Tree .
  op content : Tree -> Id .
  op isempty : Tree -> Bool .
  op isfound : Tree Id -> Bool .
  vars X Y : Tree .
  vars N M : Id .
  eq isempty(empty) = true .
  eq isempty(node(X, N, Y)) = false .
  eq left(node(X, N, Y)) = X .
  eq right(node(X, N, Y)) = Y .
  eq content(node(X, N, Y)) = N .
  eq isfound(empty, M) = false .
  eq isfound(node(X, N, Y), M) =
            (M == N) or (isfound(X, M) or isfound(Y, M)) .
endo

let n5 = node(empty, '3,  empty) .
let n6 = node(empty, '7, empty) .
let n3 = node(n5, '10, n6) .
let n4 = node(empty, '15, empty) .
let n1 = node(empty, '1, empty) .
let n2 = node(n3, '1, n4) .
let n0 = node(n1, '4, n2) .
```

```
red n0 .
red content(n0) .
red left(n0) .
red right(n0) .
red isempty(n2) .
red isfound(n0, '3) .
red isfound(n3, '10) .
```

2.1.4.5. Программная спецификация STACK-OF-NAT-1 *стека натуральных чисел с обработкой ошибок*

```
obj STACK-OF-NAT-1 is
   sorts Stack NeStack .
   subsort NeStack < Stack .
   protecting NAT .
   op empty : -> Stack .
   op push : Nat Stack -> NeStack .
   op top_ : NeStack -> Nat .
   op pop_ : NeStack -> Stack .
   var X : Nat . var S : Stack .
   eq top push(X,S) = X .
   eq pop push(X,S) = S .
endo

red top push(1,empty) .
red pop push(1,empty) .
red top empty .
red top pop empty .
```

2.1.4.6. Другая программная спецификация STACK-OF-NAT-2 *стека натуральных чисел с обработкой ошибок*

```
obj STACK-OF-NAT-2 is
   sorts Stack Stack? Nat? .
   subsort Stack < Stack? .
   protecting NAT .
   subsort Nat < Nat? .
   op empty : -> Stack .
   op push : Nat Stack -> Stack .
   op push : Nat Stack? -> Stack? .
   op top_ : Stack -> Nat? .
   op pop_ : Stack -> Stack? .
   op topless : -> Nat? .
   op underflow : -> Stack? .
   var X : Nat . var S : Stack .
   eq top push(X,S) = X .
   eq pop push(X,S) = S .
   eq top empty = topless .
   eq pop empty = underflow .
endo

red top push(1,empty) . ***> should be: 1
red pop push(1,empty) . ***> should be: empty
red top empty .         ***> should be: topless
```

```
red pop empty .          ***> should be: underflow
red top pop empty .
```

2.1.4.7. Программная спецификация WORDSTACK *словарого стека*

```
obj WORDSTACK is sort Stack .
  protecting QID .
  op empty : -> Id .
  op newstack : -> Stack .
  op push : Stack Id -> Stack .
  op pop : Stack -> Stack .
  op top : Stack -> Id .
  vars S : Stack .
  vars N : Id .
  eq pop(newstack) = newstack .
  eq top(newstack) = empty .
  eq pop(push(S, N)) = S .
  eq top(push(S, N)) = N .
endo

red push(push(push(newstack,'5),'7),'10) .
red pop(push(push(push(newstack,'5),'7),'10)) .
red pop(push(push(newstack,'5),'7)) .
red pop(push(newstack,'5)) .
red top(newstack) .
let st1 = push(push(push(newstack,'5),'7),'10) .
red st1 .
```

2.1.5. Декларация атрибутов

Некоторые свойства операций удобно рассматривать как атрибуты и объявлять их совместно с объявлением операций. Это ассоциативность, коммутативность, равенство. Объявление атрибутов операций влияет на порядок вычислений (evaluation) и синтаксический разбор (parsing).

Ассоциативность и коммутативность. Следующие примеры иллюстрируют объявление ассоциативных операций.

```
op _or_ : Bool Bool -> Bool [assoc] .
op __ : NeList List -> NeList [assoc] .
```

Выражения, включающие ассоциативный оператор не требует скобок. Например, мы можем записать `(x or y or z)` вместо `(x or (y or z))`

Бинарные инфиксные операции могут быть объявлены как *коммутативные* атрибутом `comm`, который семантически есть аксиома коммутативности, имплементированная как коммутативное правило вывода. Аксиомы, такие как

```
eq x + y = y + x
```

могут вести к не заканчивающимся выводам. Таких аксиом следует избегать, а свойство коммутативности включать как атрибут для операции `eq`. Бинарная операция может иметь коммутативные и ассоциативные атрибуты одновременно.

Атрибут тождества (одинаковости) может быть объявлен для бинарной

операции. Например, команда

```
op _or_ : Bool Bool -> Bool [assoc id : false] .
```

содержит атрибут `assoc` (ассоциативность) и атрибут `id : false`, который логически эквивалентен команде

```
(B or false = B) and (false or B = B) .
```

Атрибуты тождества могут быть константы, такие как `0` для + и `1` for ∗, равно как и `nil` при объединении множества и пустого множества.

Приоритет (precedence) оператора есть натуральное число в пределах от ноля до 127. При этом меньшее число имеет больший приоритет и связывает сильнее большего числа в качестве приоритета. Например, встроенный объект `INT` имеет команды:

```
op _+_ : Int Int -> Int [assoc comm id: 0 prec 33] .
op _*_ : Int Int -> Int [assoc comm id: 1 prec 31] .
```

Приоритет по умолчанию для оператора стандартного вида (то есть префикс со скобками) есть 0. Если оператор не начинается и не заканчивается символом подчеркивания, то по умолчанию его приоритет есть тоже 0. Унарный префиксный оператор имеет по умолчанию приоритет 15. Во всех других случаях приоритет по умолчанию есть 41.

Поведение «по умолчанию» может быть изменено.

Группировка скобок (**gathering pattern**) оператора есть последовательность элементов `e, E, &` (один элемент на каждое аргументное место), которая ограничивает приоритеты термов, разрешенных в качестве аргументов. Элемент `e` означает, что соответствующий аргумент должен иметь строго меньший приоритет. Элемент `E` разрешает равный или меньший приоритет. Элемент `&` разрешает любой приоритет. Например, `gathering pattern (E e)` для бинарного оператора в команде

```
op _+_ : T T -> T [gather (E e)] .
```

требует левую группировку скобок.

2.1.5.1. Программная спецификация `PROPC` *пропозиционального исчисления*

Программа преобразует булеву формулу в полином Жегалкина. Слагаемые полинома располагаются в лексикографическом порядке.

```
obj PROPC is
  sort Prop .
  extending TRUTH .
  protecting QID .
  subsorts Id Bool < Prop .
  op _and_ : Prop Prop -> Prop [assoc comm
                              idem idr: true prec 2] .
  op _xor_ : Prop Prop -> Prop [assoc comm
                              idr: false prec 3] .
  vars p q r : Prop .
  eq p and false = false .
  eq p xor p = false .
  eq p and (q xor r) = (p and q) xor (p and r) .
  op _or_ : Prop Prop -> Prop [assoc prec 7] .
```

```
    op not_ : Prop -> Prop [prec 1] .
    op _implies_ : Prop Prop -> Prop [prec 9] .
    op _iff_ : Prop Prop -> Prop [assoc prec 11] .
    eq p or q = (p and q) xor p xor q .
    eq not p = p xor true .
    eq p implies q = (p and q) xor p xor true .
    eq p iff q = p xor q xor true .
  endo

  red 'a implies 'b iff not 'b implies not 'a .
    ***> should be: true
  red not('a or 'b) iff not 'a and not 'b .
    ***> should be: true
  red 'c or 'c and 'd iff 'c .  ***> should be: true
  red 'a iff not 'b .  ***> should be: 'a xor 'b
  red 'a and 'b xor 'c xor 'b and 'a . ***> should be: 'c
  red 'a iff 'a iff 'a iff 'a .        ***> should be: true
  red 'a implies 'b and 'c iff ('a implies 'b) and
    ('a implies 'c) .  ***> should be: true
  red (not 'p and not 'q and not 'r) or
      (not 'p and 'q and not 'r) or
      (not 'p and not 'q and 'r) or
      ('p and not 'q and 'r) or
      ('p and 'q and 'r) .
  red (not 'p and not 'q and not 'r) or
      (not 'p and 'q and not 'r) or
      ('p and not 'q and 'r) or
      ('p and not 'q and 'r) or
      ('p and 'q and 'r) .
  red (not 'p and not 'q and not 'r) or
      (not 'p and 'q and not 'r) or
      ('p and not 'q and not 'r) or
      ('p and not 'q and 'r) or
      ('p and 'q and 'r) .
  *** Эквивалентность
  red ((not 'p and not 'q and not 'r) or
      (not 'p and 'q and not 'r) or
      ('p and not 'q and not 'r) or
      ('p and not 'q and 'r) or
      ('p and 'q and 'r))
      iff
      ((not 'p and not 'q and not 'r) or
      (not 'p and 'q and not 'r) or
      ('p and not 'q and not 'r) or
      ('p and not 'q and 'r) or
      ('p and 'q and 'r)) or false .
```

2.2. Параметризованное программирование

Параметризованное программирование осуществляется в OBJ3 с помощью следующих программных единиц: `object` (объект), `theory` (теория), `view` (проекция, присваивание), модульное выражение. Цель параметризации есть

максимизация возможностей повторного использования программных единиц, которые, оставаясь замкнутыми в себе, недоступными для редактирования, готовы для повторного использования в других единицах программы.

Для абстрактных типов данных, например таких, как множество, могут быть написаны спецификации для операций и аксиом, которые затем могут быть использованы в качестве составной части таких спецификаций как, например, конкретное множество целых чисел, множество вещественных чисел, множество последовательностей. Независимые от типа элементов множественные операции (объединение, пересечение, дополнение) могут быть специфицированы отдельно и вызываться потом в другие программные единицы, описывающие конкретные множества. Удобно отдельно специфицировать множество элементов сорта `E`, а затем использовать параметризованный модуль спецификации `SET[T]`, где `T` есть формальный параметр, который может быть отображен на сорт `E`. OBJ3 обеспечивает механизм для такой параметрической спецификации. Параметризованное программирования делит код на параметризованные компоненты. На уровне спецификации программные единицы `object, theory, view`, модульное выражение обеспечивают поддержку для написания параметрических спецификаций. Программная единица `Theory` используется для определения интерфейса и свойств параметризованного модуля. Программная единица `view` привязывает фактические параметры некоторого модуля к некоторой теории. Инстанциация (instantiating) параметризованного модуля с использованием некоторого конкретного `view` в качестве фактического параметра, дает новый модуль. Модульные выражения, содержащие интерактивные (взаимодействующие) модули, порождают новый модуль который составляется из интерактивных единиц в соответствии с модульным выражением.

2.2.1. Теории

Теории в OBJ3 имеют ту же структуру, что и объекты. Они описывают сорта, операции, переменные, равенства. Теории могут импортировать другие теории и объекты, они также могут быть параметризованы. Разница между объектами и теориями есть то, что объекты исполнимы (executable), в то время как теории лишь определяют свойства. Ниже следуют примеры теорий.

Встроенная теория `TRIV`

Теория `TRIV` вводит новый сорт `Elt`.

```
th TRIV is sort Elt .
endth
```

Теория `PREORDERED` *частичного пред-порядка*

```
th PREORDERED is sort Elt .
   op _<=_ : Elt Elt -> Bool .
   vars E1 E2 E3 : Elt .
   eq E1 <= E1 = true .
   cq E1 <= E3 = true if E1 <= E2 and E2 <= E3 .
endth
```

Теория POSET *строгого частичного порядка*

```
th POSET is
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth
```

Теория EQV *эквивалентности*

```
th EQV is
  sort Elt .
  op _eq_ : Elt Elt -> Bool [comm] .
  vars E1 E2 E3 : Elt .
  eq (E1 eq E1) = true .
  cq (E1 eq E3) = true if (E1 eq E2) and (E2 eq E3) .
endth
```

Теория MONOID *моноида*

```
th MONOID is
  sort M .
  op e : -> M .
  op _*_ : M M -> M [assoc id: e] .
endth
```

Теория FIELD *поля*

```
th FIELD is
  sorts Field NzField .
  subsorts NzField < Field .
  protecting BOOL .
  op 0 : -> Field .
  op 1 : -> NzField .
  op _+_ : Field Field -> Field [assoc comm id: 0].
  op _*_ : Field Field -> Field [assoc comm id: 1].
  op _*_ : NzField NzField -> NzField [assoc comm id: 1].
  op -_ : Field -> Field .
  op _^-1 : NzField -> NzField .
  op nz : Field -> Bool .
  vars X Y Z : Field .
  vars X' Y' : NzField .
  *** as NzField : X if nz(X) .
  eq X + (- X) = 0 .
  eq X' * (X' ^-1) = 1 .
  eq X * (Y + Z) = (X * Y) + (X * Z) .
  cq X = 0 if not nz(X) .
endth
```

2.2.2. View (проекция)

View (*отображение*, *проекция*) описывает связь между *теорией* и *объектом* таким образом, что сорта теории отображаются на сорта объекта с сохранением отношения подсортности. Операции теории отображаются на операции объекта. View есть гомоморфизм из алгебры, описанной теорией, в алгебру, описанную объектом. Ниже следуют примеры view .

View TRIV-TO-FLAVORS *из теории* TRIV *в объект* FLAVORS

TRIV-TO-FLAVORS есть имя для view из теории TRIV в объект FLAVORS.

```
view TRIV-TO-FLAVORS from TRIV to FLAVORS is
  sort Elt to Flavor .
  op newop to first .
endv
```

View NATORD *из теории* PREORDERED *в объект* NAT

NATORD есть имя для view (отображения) из теории PREORDERED в объект NAT. View NATORD описывает упорядочение less-than or equal-to (меньше или равно) на NAT.

```
view NATORD from PREORDERED to NAT is
   sort Elt to Nat .
   vars X Y : Elt .
   op X <= Y to X < Y or X == Y.
endv
```

View NATG *из теории* POSET *в объект* NAT

View NATG есть проекция из строгого частичного порядка меньше в строгий частичный порядок больше натуральных чисел

```
view NATG from POSET to NAT is
  sort Elt to Nat .
  op _<_ to _>_ .
endv
```

View NATLEQ *из предпорядка* PREORD *в предпорядок натуральных чисел*

```
view NATLEQ from PREORD to NAT is
  vars L1 L2 : Elt .
  op L1 <= L2 to L1 < L2 or L1 == L2 .
endv
```

View NATD *из отношения строгого частичного порядка в отношение делимости натуральных чисел*

(2 div 6 *есть* true)

```
view NATD from POSET to NAT is
  vars L1 L2 : Elt .
  op L1 < L2 to L1 div L2 and L1 =/= L2 .
endv
```

View NATV *из строгого частичного порядка меньше в строгий частичный порядок меньше натуральных чисел*

```
view NATV from POSET to NAT is
  sort Elt to Nat .
  op  _<_ to  _<_  .
endv
```

В следующем сокращенном view выражение Elt to Nat опущено. Сорт может быть объявлен в модуле командой principal sort <*Sort*>.

View NATG *из строгого частичного порядка меньше в строгий частичный порядок меньше натуральных чисел без объявления сорта*

```
view NATG from POSET to NAT is
  op _<_ to _<_ .
endv
```

View NAT* *из моноида* MONOID *в моноид натуральных*

чисел с умножением

```
view NAT* from MONOID to NAT is
  sort M to Nat .
  op _*_ to _*_ .
  op e to 1 .
endv
```

View `NAT+` *из моноида* `MONOID` *в моноид натуральных чисел с сложением*

```
view NAT+ from MONOID to NAT is
  op _*_ to _+_ .
  op e to 0 .
endv
```

View `LISTM` *из моноида* `MONOID` *в моноид списков с конкатенацией*

```
view LISTM from MONOID to LIST is
  op _*_ to __ .
  op e to nil .
endv
```

2.2.3. Параметризованные модули

Теории параметризованных модулей должны быть определены раньше в последовательности модулей, которые используют эти теории. Параметризованные модули (объекты или теории) представляются в системе OBJ3 следующим образом.

`obj NAME[X :: THEORY1]` для объекта,

`th NAME[X :: THEORY1]` для теории.

Таким объявлением все сорта, операции, равенства из `THEORY1` становятся видимы в модуле `NAME`.

2.2.4. Инстанциация

Инстанциация (*instantiation*, замещение, присваивание, назначение формальным переменным) параметризованного модуля замещает его формальные параметры фактическими параметрами. Каждая теория замещается соответствующим фактическим модулем с использованием `views` для связывания (to bind) фактических параметров с формальными параметрами. Инстанциация не допускает кратных копий импортируемых модулей. Инстанциация модуля `BAR`

```
obj BAR[X :: TRIV] is sort Flavor .
```

с формальным параметром `X`, отображаемого на объект `FLAVORS`, может быть выполнена по одной из следующих команд.

1. `view` используется как фактический параметр:
   `BAR[TRIV-TO-FLAVORS ] .`
2. Неназванный `view` используется как фактический параметр:
   `BAR[view from TRIV to FLAVORS is endv] .`
3. По умолчанию `view` из `TRIV` to `FLAVORS` используется как фактический параметр:
   `BAR[FLAVORS] .`

Когда алгебра инстанциации (instantiated algebra) используется в

нескольких контекстах, удобно именовать алгебру, используя команду make. Например,

```
make BAR-FLAVORS is BAR[TRIV-TO-FLAVORS] endm .
```

где BAR-FLAVORS есть имя, данное замещенному объекту

```
BAR[TRIV-TO-FLAVORS] .
```

Команда make позволяет заместить модуль только один раз. Команда make упрощает модульные выражения (*module expressions*).

Используя по умолчанию view from TRIV to NAT, параметризованный модуль ORD-PAIR может быть инстанциирован модулем NAT как фактическим параметром, чтобы получить выражение для объекта POINT:

```
make POINT is ORD-PAIR[NAT,NAT] endm .
```

Используя различные по умолчанию views from TRIV to NAT и from TRIV to BOOL, параметризованный модуль ORD-PAIR может быть инстантиирован фактическими параметрами NAT и BOOL, чтобы получить модульное выражение:

```
make PAIR-NATBOOL is ORD-PAIR[NAT,BOOL] endm .
```

Используя по умолчанию view from TRIV to POINT, мы можем получить модульное выражение:

```
make LINE-SEGMENT is ORD-PAIR[POINT,POINT] endm .
```

Например,

1) последовательность натуральных чисел определяется модулем

```
make SEQUENCE-OF-NAT is SEQUENCE[NAT] endm .
```

2) элемент *word* (слово) может быть создан с помощью встроенного сорта QID для идентификаторов. Тогда мы можем определить тип данных *note* (запись) как последовательность слов с помощью модуля:

```
make SEQUENCE-OF-WORDS is SEQUENCE[QID] endm .
```

3) упорядоченные пары могут быть охарактеризованы модулем:

```
make ORD-PAIR[QID, SEQUENCE[QID]] endm .
```

2.2.5. Модульное выражение (*Module expression*)

Модульные выражения в OBJ3 образуют формальный базис для повторного использования компьютерных программ. При построении модульных выражений возможны лишь три операции над модулями. Это *инстанциация* (*instantiation*), *переименование* (*renaming*), *сумма* (*sum*).

1. *Инстанциация* обсуждалась в предыдущем параграфе.

2. *Переименование* используется для получения нового модуля переименованием сортов и операций существующего модуля. Ниже сорт Element переименован в Newelement, сорт Flavor переименован в Vegetable, операция first переименована в newfirst, операция second в newsecond. Переименование применяется к модульному выражению постфиксно (postfix) после символа “*”, чем создается новый модуль со спецификациями предыдущего модуля.

```
th NEWELEMENT is
  using ELEMENT * (sort Element to Newelement) .
endth
```

```
FLAVORS * (sort Flavor to Vegetable .
   op first to newfirst .
   op _second_ to _newsecond_ .)
```

Операция суммы (*Sum*) строит объединение объектов. Она создает новый модуль, составляя спецификации из всех компонент суммы. Модульное выражение `A + B` создает модуль, который инкорпорирует объединение аксиом, переменных, операций, сортов из модулей `A` и `B`. Можно суммировать несколько объектов. Пусть, например, имеем два модуля типа `object` с именами `A` и `B`. Тогда выражение `A+B` создает новый модуль с именем `AB` следующего вида.

```
obj AB is
  protecting A .
  protecting B .
endo
```

2.3. Примеры параметризации

2.3.1. Параметризованная теория линейного векторного пространства с параметром

```
th VECTOR-SP[F :: FIELD] is
  sort Vector .
  op 0 : -> Vector .
  op _+_ : Vector Vector -> Vector [assoc comm id: 0] .
  op _*_ : Field Vector -> Vector .
  vars F F1 F2 : Field .
  vars V V1 V2 : Vector .
  eq (F1 + F2) * V = (F1 * V) + (F2 * V) .
  eq (F1 * F2) * V = F1 * (F2 * V) .
  eq F * (V1 + V2) = (F * V1) + (F * V2) .
endth
```

2.3.2. Параметризованный модуль **ORD-PAIR** *пар вида (натуральное число, слово)*

Модули могут иметь более одного параметра. Двупараметризованный модуль имеет следующую сигнатуру.

```
obj NAME[X :: THEORY1, Y :: THEORY2]
```

Если две теории одинаковы, то можно писать:

```
obj NAME[X Y :: THEORY1]
```

Модуль `ORD-PAIR` имеет два параметра `S` и `T`, являющихся двумя различными именами для теории `TRIV`.

```
obj ORD-PAIR[S :: TRIV, T :: TRIV] is sort OrdPair .
  protecting NAT .
  protecting BOOL .
  protecting IDENTICAL .
  protecting QID .
  op pair : Elt.S Elt.T -> OrdPair .
  op first : OrdPair -> Elt.S .
  op second : OrdPair -> Elt.T .
  op eqp : OrdPair OrdPair -> Bool [comm ] .
  var Et : Elt.T . var Es : Elt.S . vars P Q : OrdPair .
```

```
  eq first(pair(Es, Et)) = Es .
  eq second(pair(Es, Et)) = Et .
  eq eqp(P, Q) = (first(P) === first(Q)) and
   (second(P) === second(Q)) .
endo

view TRIV-TO-NAT from TRIV to NAT is
  sort Elt to Nat .
endv

view TRIV-TO-QID from TRIV to QID is
  sort Elt to Id .
endv

make NATWORD-ORD-PAIR is
   ORD-PAIR[TRIV-TO-NAT, TRIV-TO-QID]
endm

red pair (1, '2) .
red first(pair(1, '2)) .
red second(pair(1, '2)) .
red eqp(pair(1, '2), pair(1, '2)) .
red eqp(pair(1, '2), pair(1, '3)) .
```

2.3.3. Программная спецификация **LIST-OF-INT** *списков целых чисел*

```
obj LIST-OF-INT is
  sort List .
  protecting INT .
  subsort Int < List .
  op __ : Int List -> List .
  op length_ : List -> Int .
  var I : Int .  var L : List .
  eq length I = 1 .
  eq length (I L) = 1 + length(L) .
endo

red length(12) .
red length 23 .
red length(12 56 7) .  *** Скобки обязательны
```

2.3.4. Программная спецификация **LIST-OF-INT1** *списков целых чисел*

```
obj LIST-OF-INT1 is
  sorts List NeList .
  protecting INT .
  subsort Int < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
*** id: nil есть (nil List = List) and (List nil = nil)
  op __ : NeList List -> NeList [assoc id: nil] .
  op head_ : NeList -> Int .
```

```
  op tail_ : NeList -> List .
  var I : Int .  var L : List .
  eq head(I L) = I .
  eq tail(I L) = L .
endo

red 0 nil 1 nil 3 .
red nil 0 .
red nil nil 0 .
red 0 nil 0 .
red 0 1 3 .
red (6 7) (nil) .
red (6 7) nil .
red (nil) (6 7) .
red nil (6 7) .
red nil 6 7 .
red (-1 2 7) (5 5 -6) .
red (1 1 1) (4 4) (5) .
red head(9 -4 87) .
red tail(9 -4 87) .
red tail nil .
let l1 = (1 2) . let l2 = (3 4 5) .
red l1 l2 .
```

2.3.5. Программная спецификация **LIST-OF-WORDS** *списков слов*

```
obj LIST-OF-WORDS is
  sorts List NeList .
  protecting QID .
  subsort Id < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
*** id: nil есть (nil List = List) and (List nil = nil)
  op __ : NeList List -> NeList [assoc id: nil] .
  op head_ : NeList -> Id .
  op tail_ : NeList -> List .
  var I : Id .  var L : List .
  eq head(I L) = I .
  eq tail(I L) = L .
endo

red '0 nil '1 nil '3 .
red nil '0 .
red nil nil '0 .
red '0 nil '0 .
red '0 '1 '3 .
red '0 ('1 '3) .
red (6 7) (nil) .
red (6 7) nil .
red (nil) ('6 '7) .
red nil ('6 '7) .
red nil '6 '7 .
red ('-1 '2 '7) ('5 '5 '-6) .
```

```
red ('1 '1 '1) ('4 '4) ('5) .
red head('9 '-4 '87) .
red tail('9 '-4 '87) .
red tail nil .
let l1 = ('1 '2) . let l2 = ('3 '4 '5) .
red l1 l2 .

red 'wjf nil 'lkj nil '3 .
red nil 'qwerty .
red nil nil '0hjk .
red 'rtryru nil 'HgH8 .
red '0po9 '1 'k399 .
red '8rtyu ('1yui '3) .
red ('6rt 'jh7) (nil) .
red ('mnb 'h76Y) nil .
red (nil) ('6 '7) .
red nil ('NyTy '7UyT) .
red nil 'tr6 'kg7 .
red ('-1poi '2hj '7) ('5 '5 '-6) .
red ('1 '1 '1) ('4 '4v) ('595) .
red head('9 '-4 '87) .
red tail('9 '-4 '87) .
red tail nil .
let l3 = ('ui 'Rtg) . let l4 = ('3 '45 '5) .
red l3 l4 .
```

2.3.6. Параметризованный модуль **LIST[X :: TRIV]**

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList NeList -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt .
  var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = (L == nil) .
endo

red head(X L) .
red tail(X L) .
red empty? L .
red empty?(L) .
red empty?(nil) .
```

2.3.7. Программная спецификация **nTUPLE**

```
*** nTUPLE, 2 <= n <= 4.
obj 2TUPLE[C1 :: TRIV, C2 :: TRIV] is
```

```
  sort 2Tuple .
  op <<_;_>> : Elt.C1 Elt.C2 -> 2Tuple .
  op 1*_  : 2Tuple -> Elt.C1 .
  op 2*_  : 2Tuple -> Elt.C2 .
  var E1 : Elt.C1 .
  var E2 : Elt.C2 .
  eq 1* <<E1 ; E2>> = E1 .
  eq 2* <<E1 ; E2>> = E2 .
endo

red 1*(<<E1 ; E2>>) .
red 2*(<<E1 ; E2>>) .

obj 3TUPLE[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV] is
  sort 3Tuple .
  op <<_;_;_>> : Elt.C1 Elt.C2 Elt.C3 -> 3Tuple .
  op 1*_  : 3Tuple -> Elt.C1 .
  op 2*_  : 3Tuple -> Elt.C2 .
  op 3*_  : 3Tuple -> Elt.C3 .
  var E1 : Elt.C1 .
  var E2 : Elt.C2 .
  var E3 : Elt.C3 .
  eq 1* <<E1 ; E2 ; E3>> = E1 .
  eq 2* <<E1 ; E2 ; E3>> = E2 .
  eq 3* <<E1 ; E2 ; E3>> = E3 .
endo

red 1*(<<E1 ; E2 ; E3>>) .
red 2*(<<E1 ; E2 ; E3>>) .
red 3*(<<E1 ; E2 ; E3>>) .


obj 4TUPLE[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV, C4 :: TRIV] is
  sort 4Tuple .
  op <<_;_;_;_>> : Elt.C1 Elt.C2 Elt.C3 Elt.C4 -> 4Tuple .
  op 1*_  : 4Tuple -> Elt.C1 .
  op 2*_  : 4Tuple -> Elt.C2 .
  op 3*_  : 4Tuple -> Elt.C3 .
  op 4*_  : 4Tuple -> Elt.C4 .
  var E1 : Elt.C1 .
  var E2 : Elt.C2 .
  var E3 : Elt.C3 .
  var E4 : Elt.C4 .
  eq 1* <<E1 ; E2 ; E3 ; E4>> = E1 .
  eq 2* <<E1 ; E2 ; E3 ; E4>> = E2 .
  eq 3* <<E1 ; E2 ; E3 ; E4>> = E3 .
  eq 4* <<E1 ; E2 ; E3 ; E4>> = E4 .
endo

red 1*(<<E1 ; E2 ; E3 ; E4>>) .
red 2*(<<E1 ; E2 ; E3 ; E4>>) .
red 3*(<<E1 ; E2 ; E3 ; E4>>) .
red 4*(<<E1 ; E2 ; E3 ; E4>>) .
```

2.3.8. Параметризованный модуль **PREPOSET[X :: TRIV]** *натуральных чисел*

```
th PREPOSET[X :: TRIV] is
  protecting BOOL .
  op _=LE=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X =LE= X = true .
  eq (X =LE= Y) and (Y =LE= Z) = X =LE= Z .
endth

view TRIV-TO-NAT from TRIV to NAT is
  sort Elt to Nat .
endv

make NATPREPOSET is
   PREPOSET[TRIV-TO-NAT]
endm

red (1 =LE= 1) .
red (1 =LE= 2 and 2 =LE= 3) .
```

2.3.9. Параметризованный модуль **SEQUENCE[X :: ELEMS]** *списков натуральных чисел и списков слов*

```
th ELEMS is
  sort Elems .
endth

obj SEQ[X :: TRIV] is
  protecting BOOL .
endo

obj SEQUENCE[X :: ELEMS] is sort Seq .
  protecting SEQ[X] .
  protecting NAT .
  protecting BOOL .
  protecting QID .
  subsort Elems < Seq .
  op empty : -> Seq .
  op __ : Seq Seq -> Seq [assoc id: empty] .
  op tail : Seq -> Seq .
  op head : Seq -> Elems .
  op length : Seq -> Nat .
  op equ : Seq Seq -> Bool [comm] .
  vars U V : Seq . vars X Y : Elems .
  eq length(empty) = 0 .
  eq length(X U) = 1 + (length(U)) .
  eq head(X U) = X .
  eq tail(empty) = empty .
  eq tail(X U) = U .
  eq equ(empty, empty) = true .
```

```
  cq equ(U, empty) = false if U =/= empty .
  cq equ(U, V) = equ(tail(U), tail(V))
   if U =/= empty and V =/= empty and head(U) == head(V) .
  cq equ(U, V) = false
   if U =/= empty and V =/= empty and head(U) =/= head(V) .
endo

view ELEMS-TO-NAT from ELEMS to NAT is
  sort Elems to Nat .
endv

make NAT-SEQUENCE is
  SEQUENCE[ELEMS-TO-NAT]
endm

red (1 2 3) .
red length(1 2 3) .
red head(1 2 3) .
red tail(1 2 3) .
red equ((1 2 3), (1 2 3)) .
red equ((1 2 3), (2 3 4)) .
red equ((1 1 2 2 3), (1 1 2 2 3)) .
red equ((1 2 3), (1 1 2 3)) .
red (1 2 3) (1 2 3) .
red (1 2 3) (1 2 3) (1 2 3) (1 2 3) .
red (1 1) (2 2 2) (3 3 3) (4 4) .

view ELEMS-TO-QID from ELEMS to QID is
  sort Elems to Id .
endv

make WORD-SEQUENCE is
  SEQUENCE[ELEMS-TO-QID]
endm

red ('1 '2 '3) .
red length('1 '2 '3) .
red head('1 '2 '3) .
red tail('1 '2 '3) .
red equ(('1 '2 '3), ('1 '2 '3)) .
red equ(('1 '2 '3), ('2 '3 '4)) .
red equ(('1 '1 '2 '2 '3), ('1 '1 '2 '2 '3)) .
red equ(('1 '2 '3), ('1 '1 '2 '3)) .
red ('1 '2 '3) ('1 '2 '3) .
red ('1 '2 '3) ('1 '2 '3) ('1 '2 '3) ('1 '2 '3) .
red ('1 '1) ('2 '2 '2) ('3 '3 '3) ('4 '4) .
```

2.3.10. Параметризованный модуль **`PREORDER[X :: TRIV]`** *предпорядка для чисел натуральных, целых, вещественных*

```
th PREORDER[X :: TRIV] is
  protecting BOOL .
  op _=LE=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
```

```
  eq X =LE= X = true .
  eq (X =LE= Y) and (Y =LE= Z) = X =LE= Z .
endth

view TRIV-TO-NAT from TRIV to NAT is
  sort Elt to Nat .
endv
make NATPREORDER is
   PREORDER[TRIV-TO-NAT]
endm
red (1 =LE= 1) .
red (1 =LE= 3) and (3 =LE= 7) .

view TRIV-TO-INT from TRIV to INT is
  sort Elt to Int .
endv
make INTPREORDER is PREORDER[TRIV-TO-INT]
endm
red (1 =LE= 1) .
red (1 =LE= 3) and (3 =LE= 7) .
red (-1 =LE= -1) .
red (-51 =LE= -30) and (-30 =LE= 7) .

view TRIV-TO-FLOAT from TRIV to FLOAT is
  sort Elt to Float .
endv
make FLOATPREORDER is PREORDER[TRIV-TO-FLOAT]
endm
red (-1.7 =LE= -1.7) .
red (-10.5 =LE= -3.1) and (-3.1 =LE= 7.0) .
red (-1.0 =LE= -1.0) .
red (-51.1 =LE= -30.4) and (-30.4 =LE= 7.15) .
```

2.3.11. Параметризованный модуль **MONOID[X :: TRIV]** *моноида для натуральных чисел: умножение, сложение*

```
th MONOID[X :: TRIV] is
  op e : -> Elt .
  op _*_ : Elt Elt -> Elt [assoc id: e] .
endth

view NAT* from MONOID to NAT is
  sort Elt to Nat .
  op _*_ to _*_ .
  op e to 1 .
endv
make NAT*MONOID is MONOID[NAT] endm
red (3 * 1) .
red (1 * 3) .
red (3 * 7) .

view NAT+ from MONOID to NAT is
  sort Elt to Nat .
```

```
  op _*_ to _+_ .
  op e to 0 .
endv
make NAT+MONOID is MONOID[NAT] endm
red (0 + 3) .
red (3 + 0) .
red (3 + 7) .
```

Можно как ниже(хотя с `TRIV` лучше, ибо `TRIV` встроен

```
th TRIV* is
  sort M .
endth

th MONOID[X :: TRIV*] is
   op e : -> M .
   op _*_ : M M -> M [assoc id: e] .
endth

view NAT* from MONOID to NAT is
  sort M to Nat .
  op _*_ to _*_ .
  op e to 1 .
endv
make NAT*MONOID is MONOID[NAT]
endm
red (3 * 1) .
red (1 * 3) .
red (3 * 7) .

view NAT+ from MONOID to NAT is
  sort M to Nat .
  op _*_ to _+_ .
  op e to 1 .
endv
make NAT+MONOID is MONOID[NAT]
endm

red (3 + 0) .
red (0 + 3) .
red (3 + 7) .
```

2.3.12. Параметризованный модуль **`LIST[X :: TRIV]`** *моноида списков натуральных чисел и моноида списков слов*

```
th MONOID is
   sort M .
   sort Elt .
   op e : -> M .
   op _*_ : M M -> M [assoc id: e] .
endth
```

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt . var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo

view LISTM from MONOID to LIST is
  op _*_ to __ .
  op e to nil .
endv
make LISTM is LIST[NAT] endm
red (1 2 3) (1 2 3 4) .
red (1 2) nil .
red head (1 2 3) .
red tail (1 2 3) .

view LISTM from MONOID to LIST is
  op _*_ to __ .
  op e to nil .
endv
make LISTM is LIST[QID] endm
red ('1 '2 '3) ('1 '2 '3 '4) .
red ('1 '2) nil .
red head ('1 '2 '3) .
red tail ('1 '2 '3) .
red empty? ('1 '2) .
red empty? nil .
```

2.3.13. Программная спецификация **FNS, FN-FNS-2[F :: FN]**

```
obj FNS is
  protecting INT .
  ops (sq_) (dbl_) (_*3) : Int -> Int .
  var N : Int .
  eq sq N = N * N .
  eq dbl N = N + N .
  eq N * 3 = N * 3 .
endo

red sq(4) .
red sq(-4) .
red dbl(4) .
red dbl(-4) .
red 4 * -3 .
```

```
th FN is
  sort S .
  op f : S -> S .
endth

obj FNS is
  protecting INT .
  ops (sq_)(dbl_)(_*3) : Int -> Int .
  var N : Int .
  eq sq N = N * N .
  eq dbl N = N + N .
  eq N *3 = N * 3 .
endo

obj 2[F :: FN] is
  op xx : S -> S .
  var X : S .
  eq xx(X) = f(f(X)) .
endo

red in 2[(sq_).FNS] : xx(3) .    ***> should be: 81
red xx(4) .                      ***> should be: 256
red in 2[(dbl_).FNS] : xx(3) .   ***> should be: 12
red in 2[2[(sq_).FNS]*(op xx to f)] : xx(2) .
                                 ***> should be: 65536
red xx(3) .                       ***> should be: 43046721
```

2.3.14. Программная спецификация **PROPC-TIME-WIRE-NOT-F**

```
obj PROPC is
  sort Prop .
  extending TRUTH .
  protecting QID .
  subsorts Id Bool < Prop .
  op _and_ : Prop Prop -> Prop
     [assoc comm idem idr: true prec 2] .
  op _xor_ : Prop Prop -> Prop
     [assoc comm idr: false prec 3] .
  vars p q r : Prop .
  eq p and false = false .
  eq p xor p = false .
  eq p and (q xor r) = (p and q) xor (p and r) .
  op _or_ : Prop Prop -> Prop [assoc prec 7] .
  op not_ : Prop -> Prop [prec 1] .
  op _implies_ : Prop Prop -> Prop [prec 9] .
  op _iff_ : Prop Prop -> Prop [assoc prec 11] .
  eq p or q = (p and q) xor p xor q .
  eq not p = p xor true .
  eq p implies q = (p and q) xor p xor true .
  eq p iff q = p xor q xor true .
endo

obj TIME is
    sort Time .
```

```
    op 0 : -> Time .
    op s_ : Time -> Time .
endo

th WIRE is
    protecting TIME + PROPC .
    op f1 : Time -> Prop .
endth

obj NOT[W :: WIRE] is
    op f2 : Time -> Prop .
    var T : Time .
    eq f2(0) = false .
    eq f2(s T) = not f1(T) .
endo

obj F is
    extending TIME + PROPC .
    op t : -> Time .
    op f0 : Time -> Prop .
endo

make 2NOT is NOT[ NOT[F]*(op f2 to f1) ] endm

red f2(s s t) iff f0(t) . ***> should be: true
```

2.3.15. Программная спецификация булевых свойств множеств (объединение, пересечение, разность)

```
obj BSET[X :: TRIV] is
  sort Set .
  *** pr IDENTICAL .
  ops ({}) (omega) : -> Set .
  op  {_} : Elt -> Set .
  op _+_ : Set Set -> Set [assoc comm id: {}] .
     *** exclusive or
  op _&_ : Set Set -> Set [assoc comm idem id: omega] .
     *** intersection
  vars S S' S'' : Set . vars E E' : Elt .
  eq S + S = {} .
  cq { E } & { E' } = {} if E =/= E' .
  eq S & {} = {} .
  cq S & (S' + S'') = (S & S') + (S & S'')
                 if (S' =/= {}) and (S'' =/= {}) .
     *** made conditional as an example of how
     *** to avoid non-termination
     *** from identity completion (in fact, not required)
endo

obj SET[X :: TRIV] is
  protecting BSET[X] .
  protecting INT .
  op _U_ : Set Set -> Set [assoc comm id: {}] .
  op _-_ : Set Set -> Set .
```

```
  op #_ : Set -> Int [prec 0] .
  op _in_ : Elt Set -> Bool .
  op _in_ : Set Set -> Bool .
  op empty?_ : Set -> Bool .
  var X : Elt .
  vars S S' S'' : Set .
  eq S U S' = (S & S') + S + S' .
  eq S - S' = S + (S & S') .
  eq empty? S = S == {} .
  eq X in S = { X } & S =/= {} .
  eq S in S' = S U S' == S' .
  eq #{} = 0 .                              *** the volume
  cq #({ X } + S) = #S if X in S .          *** the volume
  cq #({ X } + S) = 1 + # S if not X in S . *** the volume
endo

*** test cases

obj SET-OF-INT is
  protecting SET[INT] .
  ops s1 s2 s3 : -> Set [memo] .
  eq s1 = { 1 } .
  eq s2 = s1 U { 2 } .
  eq s3 = s2 U { 3 } .
endo

red s3 .               ***> should be: {1,2,3}
red # s3 .             ***> should be: 3
red (s2 U s1) .        ***> should be: {1,2}
red #(s3 U s1) .       ***> should be: 3
red empty?(s3 + s3) .  ***> should be: true
red empty?(s1 + s3) .  ***> should be: false
red 3 in s2 .          ***> should be: false
red s1 in s3 .         ***> should be: true
red s1 - s3 .          ***> should be: {}
red s3 - s2 .          ***> should be: {3}
red s3 & s1 .          ***> should be: {1}
red s3 & s2 .          ***> should be: {1,2}
red omega U s2 .       ***> should be: omega
```

2.3.16. Программная спецификация **LIST** *(список натуральных чисел и список слов)*

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt . var L : List .
```

```
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo

view TRIV-TO-NAT from TRIV to NAT is
  sort Elt to Nat .
endv
make NAT-LIST is LIST[TRIV-TO-NAT] endm
red empty?(nil) .
red empty?(1 2 3) .
red (1 2 3) (1 2) .
red tail(1 2 3) .
red head(1 2 3) .

view TRIV-TO-QID from TRIV to QID is
  sort Elt to Id .
endv
make WORD-LIST is LIST[TRIV-TO-QID] endm
red empty?(nil) .
red empty?('1 '2 '3) .
red ('1 '2 '3) ('1 '2) .
red tail('1 '2 '3) .
red head('1 '2 '3) .
```

2.3.17. Расширенная программная спецификация **`LIST`** *(список натуральных чисел и список слов)*

```
th LIST[X :: TRIV] is
  sorts List NeList .
  protecting NAT .
  protecting QID .
  op nil : -> List .
  subsorts Elt < NeList < List .
  op __ : List List -> List [assoc id: nil] .
  op __ : NeList List -> NeList .
  op __ : NeList NeList -> NeList .
  op |_| : List -> Nat .
  eq | nil | = 0 .
  var E : Elt . var L : List .
  eq | E L | = 1 + | L | .
  op tail_ : NeList -> List [prec 120] .
  op head_ : NeList -> Elt [prec 120] .
  var E : Elt . var L : List .
  eq tail E L = L .
  eq head E L = E .
endth

view TRIV-TO-NAT from TRIV to NAT is
  sort Elt to Nat .
endv
make NAT-LIST is LIST[TRIV-TO-NAT] endm
red | (1 2 3) | .
```

```
red (1 2 3) (1 2) .
red | nil | .
red tail(1 2 3) .
red head (1 2 3) .

view TRIV-TO-QID from TRIV to QID is
  sort Elt to Id .
endv
make WORD-LIST is LIST[TRIV-TO-QID] endm
red | ('1 '2 '3) | .
red ('1 '2 '3) ('1 '2) .
red | nil | .
red tail('1 '2 '3) .
red head ('1 '2 '3) .
```

2.3.18. Программная спецификация: решето Эратосфена

Решето Эратосфена есть алгоритм «просеивания» последовательности S натуральных чисел $2,3,4,\ldots,N$, состоящий в следующем.

1. Удалить из S все числа, кратные 2, кроме простого p_0 =2. В полученной последовательности удалить все числа, кратные 3, кроме простого p_1 =3. Пока очередное простое $p_k \leqslant \sqrt{N}$, в получающихся последовательностях продолжать последовательное удаление чисел кратных p_k. В результате в S останется последовательность всех простых чисел между 2 и N.

```
obj LAZYLIST[X :: TRIV] is
  sort List .
  subsort Elt < List .
  op nil : -> List .
  op __ : List List ->  List
     [assoc idr: nil prec 80 strat (0)] .
endo

obj SIEVE is
  protecting LAZYLIST[INT] .
  op force : List List -> List [strat (1 2 0)] .
  op show_upto_ : List Int -> List .
  op filter_with_ : List Int -> List .
  op ints-from_ : Int -> List .
  op sieve_ : List -> List .
  op primes : -> List .
  vars P I E : Int .
  var S L : List .
  eq force(L,S) = L S .
  eq show nil upto I = nil .
  eq show E S upto I =
     if I == 0 then nil else force(E,show S upto (I - 1)) fi .
  eq filter nil with P = nil .
  eq filter I S with P = if (I rem P) == 0
     then filter S with P
     else I (filter S with P) fi .
  eq ints-from I = I (ints-from (I + 1)) .
  eq sieve nil = nil .
```

```
  eq sieve (I S) = I (sieve (filter S with I)) .
  eq primes = sieve (ints-from 2) .
endo

red show primes upto 10 .
   ***> should be: 2 3 5 7 11 13 17 19 23 29
red show primes upto 15 .
```

2.3.19. Сумма и произведение элементов списка натуральных чисел

```
th MONOID is
  sort M .
  op e : -> M .
  op _*_ : M M -> M [assoc id: e] .
endth

obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt . var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo

obj ITER[M :: MONOID] is
  protecting LIST[M] .
  op iter : List -> M .
  var X : M . var L : List .
  eq iter(nil) = e .
  eq iter(X L) = X * iter(L) .
endo

view NAT+ from MONOID to NAT is
  op _*_ to _+_ .
  op e to 0 .
endv
make SIGMA is ITER[NAT+] endm
red iter(1 2 3 4) . ***> should be 10

view NAT* from MONOID to NAT is
  op _*_ to _*_ .
  op e to 1 .
endv
make PI is ITER[NAT*] endm
red iter(1 2 3 4) . ***> should be 24.

make SIGMA+PI is
```

```
ITER[NAT+] * (op iter to sigma) + ITER[NAT*] * (op iter to pi)
endm
red sigma(1 2 3 4) + pi(1 2 3 4) .
***> should be 34
```

2.3.20. Сортировка списков слов и пузырьковая сортировка списка натуральных чисел

```
th POSET is
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth

obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt . var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo

view NATD from POSET to NAT is
  vars L1 L2 : Elt .
  op L1 < L2 to L1 div L2 and L1 =/= L2 .
endv

view NATG from POSET to NAT is
  sort Elt to Nat .
  op _<_ to _>_ .
endv

obj LEXL[X :: POSET] is
  protecting LIST[X] .
  op _<<_ : List List -> Bool .
  vars L L' : List . vars E E' : Elt .
  eq L << nil = false .
  eq nil << E L = true .
  eq E L << E L' = L << L' .
  cq E L << E' L' = E < E' if E =/= E' .
endo

make LEXL-LIST is LEXL[NATG] endm
make NAT-LIST is LIST[NAT] endm
make LEXL-NAT is LEXL[NAT] endm
```

```
make LEXL-NATD is LEXL[NATD] endm
make PHRASE is LEXL[QIDL] endm
make PHRASE-LIST is LEXL[LEXL[QIDL]] endm
make PHRASE-LIST is LEXL[LEXL[QIDL] * (op __ to _._) ] endm

obj SORTING[X :: POSET] is
  protecting LIST[X] .
  op sorting_ : List -> List .
  op unsorted_ : List -> Bool .
  vars L L' L'' : List .
  vars E E' : Elt .
  cq sorting L = L if unsorted L =/= true .
  cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
  cq unsorted L E L' E' L'' = true if E' < E .
endo

red in SORTING[INT] : unsorted 1 2 3 4 .
   ***> should not be: true
red unsorted 4 1 2 3 .     ***> should be: true
red sorting 1 4 2 3 .      ***> should be: 1 2 3 4

make SORTING-PH-LIST is SORTING[ LEXL[QIDL] * (op __ to _._) ]
endm

***> good
red sorting (('b . 'a)('a . 'a)('a . 'b)) .
***> should be: ('a . 'a)('a . 'b)('b . 'a)

red in SORTING[NATD] : sorting(18 6 5 3 1) .
***> should contain: 1 3 6 18

th TOSET is
  using POSET .
  vars E1 E2 E3 : Elt .
  cq E1 < E2 or E2 < E1 = true if E1 =/= E2 .
endth

obj BUBBLES[X :: TOSET] is
  protecting LIST[X] .
  op sorting_ : List -> List .
  op sorted_ : List -> Bool .
  vars L L' L'' : List .
  vars E E' : Elt .
  cq sorting L = L if sorted L .
  cq sorting L E E' L'' = sorting L E' E L'' if E' < E .
  eq sorted nil = true .
  eq sorted E = true .
  cq sorted E E' L = sorted E' L if E < E' or E == E' .
endo

red in A is BUBBLES[NAT] : sorting(18 5 6 3) .
   ***> should be: 3 5 6 18
red sorting(8 5 4 2) .          ***> should be: 2 4 5 8
```

```
*** red in B is BUBBLES[NATD] : sorting(18 5 6 3) .
   ***> mightnt contain: 3 6 18
*** red sorting(8 5 4 2) .  ***> mightnt contain: 2 4 8
red in C is SORTING[NATD] : sorting(18 5 6 3) .
   ***> should contain: 3 6 18
red sorting(8 5 4 2) .            ***> should contain: 2 4 8
red in A : sorting(9 6 3 1) .     ***> should be: 1 3 6 9
*** red in B : sorting(9 6 3 1) .  ***> mightnt be: 1 3 6 9
red in C : sorting(9 6 3 1) .     ***> should be: 1 3 6 9
```

2.3.21. Сумма и произведение элементов списка натуральных чисел

```
th MONOID is
  sort M .
  op e : -> M .
  op _*_ : M M -> M [assoc id: e] .
endth

obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt . var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo

obj ITER[M :: MONOID] is
  protecting LIST[M] .
  op iter : List -> M .
  var X : M . var L : List .
  eq iter(nil) = e .
  eq iter(X L) = X * iter(L) .
endo

view NAT+ from MONOID to NAT is
  op _*_ to _+_ .
  op e to 0 .
endv
make SIGMA is ITER[NAT+] endm
red iter(1 2 3 4) . ***> should be 10

view NAT* from MONOID to NAT is
  op _*_ to _*_ .
  op e to 1 .
endv
make PI is ITER[NAT*] endm
```

```
red iter(1 2 3 4) . ***> should be 24.

make SIGMA+PI is
ITER[NAT+] * (op iter to sigma) + ITER[NAT*] * (op iter to pi)
endm
red sigma(1 2 3 4) + pi(1 2 3 4) .
***> should be 34
```

2.4. Многооконное окружение

Приведем пример построения спецификации в OBJ3 для обслуживания дисплея (screen) с многими (multiple) окнами, где каждое окно ассоциируется с множеством некоторых геометрических форм (shapes). Ради простоты ограничимся окнами, ассоциированными с квадратами (рис.2.1).

2.4.1. Требования (Requirements)

Экран (дисплей) есть прямоугольная область, которая включает совокупность окон. Каждое окно (window) на экране есть прямоугольник со сторонами, параллельными осям экрана. Окно ассоциируется с совокупностью квадратных форм, начерченных внутри его. Когда окно перемещается внутри экрана на другое место, квадратные формы, с ним ассоциированные, перемещаются тоже без изменения их относительных позиций (without any change to their relative positions) внутри окна. В процессе разработки нужно выполнить следующее.

1. Создать окно.
2. Определить, находится ли курсор внутри данного окна.
3. Выбрать окно, отмеченное (identified) курсором.
4. Переместить окно to a specified location внутри экрана.
5. Определить, не перекрываются (окнаoverlap) ли два окна.
6. Добавить окно к совокупности окон.
7. Ассоциировать список квадратов с данным окном.
8. Добавить квадрат к списку квадратов, ассоциированных с окном.
9. Переместить квадрат горизонтально внутри окна.
10. Переместить квадрат вертикально внутри окна.

Fig. 13.26 Coordinate axes for window environment

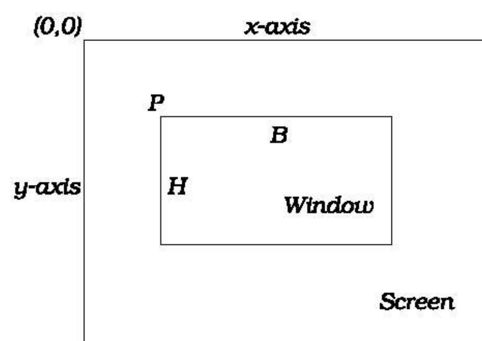


Рис.2.1. Координатные оси для window environment

2.4.2. Моделирование

Определим координаты точки на экране как пару натуральных чисел. Моделируем курсор как точку. Прямоугольник строится из точки с указанием ее верхнего левого угла и двух натуральных чисел, обозначающих denoting широту и высоту прямоугольника. Специфицируем прямоугольный объект, расширяя его новыми операциями и аксиомами, чтобы получить квадратный объект. Подобно специфицируется прямоугольный объект для моделирования окна.

Определим параметрический объект с одним параметром, чтобы моделировать список элементов некоторого сорта. Этот объект может быть инстанциирован (instantiated), чтобы получить списки квадратов и списки окон. Моделируем экран как список окон, где каждое окно ассоциировано со списком квадратов.

Позиция точки на экране задается по отношению к координатным осям, где начало координат (origin) есть левый верхний угол экрана, *x*-ось есть горизонтальная ось, а *y*-ось есть вертикальная ось, как это показано на рис.13.26.

2.4.3. Программные спецификации многооконного окружения

```
*** ********************************************************
***                 SIMPLE WINDOW SYSTEM                 ***
***                   OBJ SPECIFICATION                  ***
*** ********************************************************

*** when multiple objects are defines
*** you can open the needed one by typing
*** open MODULENAME ., for example:
*** BOBJ>open CURSOR, and then close it.

*** an object representing the two coordinates of a point.
obj POINT is sort Point .
  protecting NAT .
  op point : Nat Nat -> Point .
  op x     : Point -> Nat .
  op y     : Point -> Nat .
  vars A B : Nat .
  eq x(point(A,B)) = A .
  eq y(point(A,B)) = B .
endo

*** Example:
*** BOBJ> reduce x(point(42, 0)).
*** ==========================================
*** reduce in POINT : x(point(42, 0))
*** result NzNat: 42
*** rewrite time: 41ms       parse time: 1ms
```

```
*** Example: let-binding
*** (like assigning variables in imperative languages)
*** BOBJ> let mypoint = point(42,99).
*** opening module RECTANGLE.
*** BOBJ> red x(mypoint).
*** ==========================================
*** reduce in RECTANGLE : x(mypoint)
*** result NzNat: 42
*** rewrite time: 20ms       parse time: 1ms

*** ------------------------------------------------------ ***
*** defining an object representing the click at a point.
obj CURSOR is sort Cursor .
  protecting NAT .
  protecting POINT .
  op cursor  : Point -> Cursor .
  op move    : Cursor Nat Nat -> Cursor .
  op x       : Cursor -> Nat .
  op y       : Cursor -> Nat .
  vars X Y : Nat . var P : Point .
  eq x(cursor(P)) = x(P) .
  eq y(cursor(P)) = y(P) .
  eq move(cursor(P),X,Y) = cursor(point(X,Y)) .
endo

*** Example:
*** BOBJ> reduce x(cursor(point(42,1))).
*** ==========================================
*** reduce in CURSOR : x(cursor(point(42, 1)))
*** result NzNat: 42
*** rewrite time: 21ms       parse time: 2ms

*** BOBJ> let zap = cursor(point(2,1)).
*** BOBJ> red x(zap).
*** ==========================================
*** reduce in CURSOR : x(zap)
*** result NzNat: 2
*** rewrite time: 3ms       parse time: 1ms


*** ------------------------------------------------------ ***
*** defining an object representing a rectangle.
obj RECTANGLE is sort Rectangle .
  protecting NAT .
  protecting POINT .
  op rectangle : Point Nat Nat -> Rectangle .
  op locate     : Rectangle -> Point .
  op breadth     : Rectangle -> Nat .
  op height     : Rectangle -> Nat .
  op topleft    : Rectangle -> Point .
  op topright   : Rectangle -> Point .
  op downleft   : Rectangle -> Point .
```

```
    op downright  : Rectangle -> Point .
    var P : Point . vars B H : Nat .
    eq locate(rectangle(P,B,H)) = P .
    eq breadth(rectangle(P,B,H)) = B .
    eq height(rectangle(P,B,H)) = H .
    eq topleft(rectangle(P,B,H)) = P .
    eq topright(rectangle(P,B,H)) = point(x(P) + B, y(P)) .
    eq downleft(rectangle(P,B,H)) = point(x(P), y(P) + H) .
    eq downright(rectangle(P,B,H))= point(x(P) + B, y(P) + H) .
  endo

  *** BOBJ> let myrect = rectangle(point(1,2), 42, 50).
  *** BOBJ> red breadth(myrect).
  *** ==========================================
  *** reduce in RECTANGLE : breadth(myrect)
  *** result NzNat: 42
  *** rewrite time: 3ms       parse time: 2ms

  *** BOBJ> red downright(myrect).
  *** ==========================================
  *** reduce in RECTANGLE : downright(myrect)
  *** result Point: point(43, 52)
  *** rewrite time: 8ms       parse time: 2ms

  *** ------------------------------------------------------ ***
  *** defining an object representing a square.
  obj SQUARE is
    extending RECTANGLE * ( sort Rectangle to Square ,
                            op rectangle to square ) .
    protecting NAT .
    protecting POINT .
    op square : Point Nat -> Square .
    op side   : Square -> Nat .
    var P : Point . vars B H S : Nat .
    cq locate(square(P,B,H)) = P if B == H .
    cq side(square(P,B,H))   = B if B == H .
    eq locate(square(P,S))   = P .
    eq side(square(P,S))     = S .
    cq square(P,B,H) = square(P,B) if B == H .
  endo

  *** BOBJ> let mysquare = square(point(3,4), 90).
  *** opening module SQUARE.

  *** BOBJ> red side(square(point(30, 30),30,30)).
  *** ==========================================
  *** reduce in SQUARE : side(square(point(30, 30), 30, 30))
  *** result NzNat: 30
  *** rewrite time: 2ms       parse time: 5ms

  *** If we specify and invalid rect (i.e., where width !=
height)
  *** Then, the reduction does not yield a final result
```

```
  *** Instead, it yields the last expression that failed to
reduce
  *** BOBJ> red side(square(point(30,30), 40, 50)).
  *** ==========================================
  *** reduce in SQUARE : side(square(point(30, 30), 40, 50))
  *** result Nat: side(square(point(30, 30), 40, 50))
  *** rewrite time: 4ms       parse time: 5ms


  *** ---------------------------------------------------------- ***
  *** defining an object representing a window.
  obj WINDOW is
    extending RECTANGLE * ( sort Rectangle to Window ,
                            op rectangle to window ) .
    protecting NAT .
    protecting BOOL .
    protecting POINT .
    protecting CURSOR .
    op move     : Window Point -> Window .
    op contains : Window Cursor -> Bool .
    op cross    : Window Window -> Bool .
    vars B H : Nat . vars P : Point . var C : Cursor .
    vars W W1 W2 : Window .
    *** move window to a specified location.
    eq move(W,P) = window(P, breadth(W), height(W)) .
    *** true if the window contains the location of the cursor
    eq contains(window(P,B,H),C) = x(C) >= x(P) and
                                   x(C) <= (x(P) + B) and
                                   y(C) >= y(P) and
                                   y(C) <= (y(P) + H) .
    *** two windows cross each other if one of them has a
    *** corner which is contained in the other window.
    eq cross(W1,W2) = contains(W1,cursor(topleft(W2))) or
                      contains(W1,cursor(topright(W2))) or
                      contains(W1,cursor(downleft(W2))) or
                      contains(W1,cursor(downright(W2))) .
  endo

  *** Remember, window is a rectangle!
  *** Therefore, it is constructed in the same way
  *** BOBJ> let mywnd = window(point(4,4), 300, 200).

  *** BOBJ> red move(mywnd, point(5,5)).
  *** ==========================================
  *** reduce in WINDOW : move(mywnd, point(5, 5))
  *** result Window: window(point(5, 5), 300, 200)
  *** rewrite time: 7ms       parse time: 1ms

  *** ---------------------------------------------------------- ***
  *** a parameterized object LIST which takes one parameter.
  obj LIST[X :: TRIV] is sort NeList List .
    protecting NAT .
    protecting BOOL .
```

```
    subsorts Elt < NeList < List .
    op null      : -> List .
    op __        : List List -> List [assoc id: null] .
    op __        : NeList List -> NeList .
    op tail _    : List -> List .
    op head _    : NeList -> Elt .
    op empty? _  : List -> Bool .
    op length _  : List -> Nat .
    op copy      : List List -> List .
    var X : Elt . vars L L1 : List .
    eq empty? null = true .
    eq empty? L = L == null .
    eq length null = 0 .
    eq length(X L) = length(L) + 1 .
    eq head(X L) = X .
    eq tail(null) = null .
    eq tail(X L) = L .
    eq empty?(copy(null,L)) = true .
    eq tail(copy((X L),L1)) = L .
  endo

  *** ----------------------------------------------------- ***
  *** defining a view from the pre-defined theory TRIV
  *** to the object WINDOW
  view WIN from TRIV to WINDOW is
    sort Elt to Window .
  endv
  *** creating an object representing a list of windows.
  *** instantiating the parameterised object LIST.

  *** WARNING: original source had a dot after parentheses
  *** However, we need to remove it for BOBJ
  make WINLIST is
    LIST[WIN] * ( sort List to Winlist )
  endm

  *** BOBJ> let w1 = window(point(4,4), 300, 200).
  *** BOBJ> red w1 .
  *** ==========================================
  *** reduce in WINLIST : w1
  *** result Window: window(point(4, 4), 300, 200)
  *** rewrite time: 1ms       parse time: 0ms

  *** BOBJ> let w2 = window(point(240, 300), 400, 500) .
  *** BOBJ> let wlist = (w1 w2) .
  *** BOBJ> red wlist .
  *** ==========================================
  *** reduce in WINLIST : wlist
  *** result NeList: (window(point(4, 4), 300, 200))
(window(point(240,
  ***      300), 400, 500))
  *** rewrite time: 4ms       parse time: 0ms
```

```
*** ----------------------------------------------------- ***
*** Demonstration of defining and using LISTs
*** ----------------------------------------------------- ***

*** let us make a list of natural numbers
*** first, we need to make a view (i.e., replace instances
*** of pseudo-type Elt with Nat)

*** view NNT from TRIV to NAT is
***     sort Elt to Nat .
*** endv

*** now, let us define a list of Nats
*** we need to specify the list construction operation
*** make NatList is
***     LIST[NNT] * ( sort List to NatList)
*** endm

*** a single natural number is a list
*** BOBJ>red head(2).
*** ==========================================
*** reduce in NatList : head 2
*** result NzNat: 2
*** rewrite time: 0ms       parse time: 1ms

*** BOBJ> let mmlist = (2 3 4).

*** BOBJ> red length(mmlist).
*** ==========================================
*** reduce in NatList : length mmlist
*** result NzNat: 3
*** rewrite time: 5ms       parse time: 0ms

*** BOBJ> red tail(mmlist).
*** ==========================================
*** reduce in NatList : tail mmlist
*** result NeList: 3 4
*** rewrite time: 4ms       parse time: 0ms


*** ----------------------------------------------------- ***
*** defining an object representing a screen.
obj SCREEN is sort Screen .
  protecting BOOL .
  protecting CURSOR .
  protecting WINDOW .
  protecting WINLIST .
  op screen  :  Winlist -> Screen .
  op winlist :  Screen -> Winlist .
  op addwin  :  Screen Window -> Screen .
  op overlap :  Screen Window -> Bool .
  op select  :  Screen Cursor -> Window .
  var B : Cursor . var W : Window .
```

```
  var WL : Winlist . var S : Screen .
  eq winlist(screen(WL)) = WL .
  eq winlist(addwin(S,W)) = W winlist(S) .
  eq overlap(screen(null),W) = false .
  eq overlap(S,W) = cross(head(winlist(S)),W) or
                    overlap(screen(tail(winlist(S))),W) .
  eq select(screen(WL),B) = if contains(head(WL),B)
                            then head(WL)
                            else (select(screen(tail(WL)),B))
                            fi .
endo

*** ----------------------------------------------------- ***
*** defining a view from the pre-defined theory TRIV
*** to the object SQUARE
view SQUAR from TRIV to SQUARE is
  sort Elt to Square .
endv

*** creating an object representing a list of squares.
*** instantiating the parameterized object LIST.

*** WARNING: dot before endm is an error in BOBJ
***
make SQUARELIST is
  LIST[SQUAR] * ( sort List to Squarelist )
endm

*** consturcting a screen from a list of windows
*** BOBJ> let s1 = screen(wlist) .
*** BOBJ> red s1 .
*** ==========================================
*** reduce in SCREEN : s1
*** result Screen: screen((window(point(4, 4), 300, 200))
***    (window(point(240, 300), 400, 500)))
*** rewrite time: 6ms       parse time: 0ms

*** checking for overlap
*** BOBJ> let w3 = window(point(100, 120), 400, 500) .
*** BOBJ> red overlap s1 w3 .
*** ==========================================
*** No parse for overlap s1 w3 at line 1.
*** BOBJ> red overlap(s1, w3) .
*** ==========================================
*** reduce in SCREEN : overlap(s1, w3)
*** result Bool: true
*** rewrite time: 18ms       parse time: 1ms

*** selecting a window
*** BOBJ> let cur = cursor(point(200, 150)) .
*** BOBJ> red select(s1, cur) .
*** ==========================================
*** reduce in SCREEN : select(s1, cur)
```

```
*** result Window: window(point(4, 4), 300, 200)
*** rewrite time: 18ms      parse time: 1ms


*** ----------------------------------------------------- ***
*** defining an object representing the shapes in a window.
*** the object associates a window with a list of squares.

*** WARNING: added dot after 'Windowshapes'
***
obj WINDOWSHAPES is sort Windowshapes .
  protecting NAT .
  protecting BOOL .
  protecting POINT .
  protecting CURSOR .
  protecting WINDOW .
  protecting SQUARELIST .
  op windowshapes      : Window Squarelist -> Windowshapes .
  op win               : Windowshapes -> Window .
  op squarelist        : Windowshapes -> Squarelist .
  op squarewithin      : Square Window -> Bool .
  op addsquare         : Square Windowshapes -> Windowshapes .
  op htranslatesquare  : Square Window Nat -> Square .
  op vtranslatesquare  : Square Window Nat -> Square .
  var W : Window . var SL : Squarelist .
  var WS : Windowshapes . var S : Square .
  var P : Point . vars L X Y : Nat .
  eq win(windowshapes(W,SL)) = W .
  eq squarelist(windowshapes(W,SL)) = SL .
  *** determine if a square fits within a window.
  eq squarewithin(square(P,L),W) =
         contains(W,cursor(topleft(square(P,L)))) and
         contains(W,cursor(topright(square(P,L)))) and
         contains(W,cursor(downleft(square(P,L)))) and
         contains(W,cursor(downright(square(P,L)))) .
  *** add square to list of squares if within the window.
  eq addsquare(S,WS) =
         if squarewithin(S,win(WS))
         then windowshapes(win(WS), (S squarelist(WS)))
         else WS fi .
  *** translate square horizontally if fits in window
  *** at new location.
  eq htranslatesquare(square(P,L),W,X) =
         if squarewithin(square(point(x(P) + X, y(P)), L), W)
         then square(point(x(P) + X, y(P)), L)
         else square(P,L) fi .
  *** translate square vertically if fits in window
  *** at new location.
  eq vtranslatesquare(square(P,L),W,Y) =
        if squarewithin(square(point(x(P), y(P) + Y), L), W)
        then square(point(x(P), y(P) + Y), L)
        else square(P,L) fi .
endo
```

```
*** BOBJ> let sq1 = square(point(3,4), 90).
*** BOBJ> let sq2 = square(point(30, 30), 40) .
*** BOBJ> let w1 = window(point(4,4), 300, 200).
*** BOBJ> let slist = (sq1 sq2) .

*** BOBJ> red win(wshape) .
*** ==========================================
*** reduce in WINDOWSHAPES : win(wshape)
*** result Window: window(point(4, 4), 300, 200)
*** rewrite time: 6ms       parse time: 0ms

*** BOBJ> red squarelist(wshape) .
*** ==========================================
*** reduce in WINDOWSHAPES : squarelist(wshape)
*** result NeList: (square(point(30, 30), 40))
(square(point(3, 4), 90))
*** rewrite time: 7ms       parse time: 1ms


*** ----------------------------------------------------- ***
```

ЛИТЕРАТУРА

1. **Ершов Ю.Л.**, **Лавров И.А.**, **Тайманов А.Д.**, **Тайцлин М.А.** Элементарные теории// Успехи матем. наук, т.20, вып.4 (124), 1965. С.37-108.
2. **Alagar V.S.**, **Periyasamy K**. Specification of software systems. Springer Verlag London Limited. 2011. 644p.
3. **Goguen** Joseph A. Introducing OBJ. // SRI CSL-88-9, SRI International, USA, 1988. www.kindsoftware.com/products/opensource/
4. **Nakagava A.T.**, **Savada T.**, **Futatsugi K**. CafeOBJ user's manual. www.ldl.jaist.ac.jp/cafeobj/
5. **Tatami project**. http://cseweb.ucsd.edu/groups/tatami/
6. The OBJ family. http://cseweb.ucsd.edu//users/goguen/sys/obj.html#BOBJ ftp://ftp.cs.ucsd.edu/pub/fac/goguen/bobj/

ОГЛАВЛЕНИЕ

Введение … 3
1. Множество … 3
2. Функция … 4
3 . Отношение … 5
4. Отношение эквивалентности … 5
5. Каноническое разложение функции … 7

1. Универсальные алгебры и компьютерные программы … 8
1.1. Алгебры и спецификации программ … 8
1.2. Алгебры, подалгебры, гомоморфизм алгебр … 14
1.2.1. Односортные (однородные, homogeneous) алгебры … 14
1.2.2. Конгруенции … 18
1.2.3. Многосортные (неоднородные, heterogeneous) алгебры … 21
1.3. Спецификация … 24

2. Алгоритмический язык программирования OBJ3 … 27
2.1. Базисный синтаксис OBJ3 … 27
2.1.1. Спецификация объекта … 28
2.1.2. Сорта и подсорта … 29
2.1.3. Расширение многосортных спецификаций (импорт модулей) … 30
2.1.3.1. Программная спецификация `NATURAL` *сложения и умножения унарных натуральных чисел* … 31
2.1.3.2. Программная спецификация `BOOLEAN` … 32
2.1.3.3. Программная спецификация `ORDEREDNATURAL` *упорядоченных унарных натуральных чисел* … 32
2.1.3.4. Программная спецификация `NAT1` *сложения унарных натуральных чисел* … 33
2.1.3.5. Программная спецификация `NAT2` *сложения и умножения унарных натуральных чисел (импорт* `NAT1` *в* `NAT2` *методом* `protecting`) … 33
2.1.3.6. Программная спецификация `NAT3` *сложения, умножения, порядка натуральных чисел (импорт* `NAT2` *в* `NAT3` *методом* `protecting`) … 33
2.1.3.7. Программная спецификация `NAT4` *сложения, умножения, порядка, нахождения частного и НОД унарных натуральных чисел (импорт* `NAT3` *в* `NAT4` *методом* `protecting`) … 34
2.1.3.8. Программная спецификация `NAT` *сложения, умножения, порядка, нахождения частного и НОД унарных натуральных чисел* … 34

2.1.4. Встроенные сорта … 35
2.1.4.1. Программная спецификация SIMPLESET-NAT *простого множества натуральных чисел* … 39
2.1.4.2. Программная спецификация SIMPLESET-WORD *простого множества слов* … 40
2.1.4.3. Программная спецификация obj NATTREE *бинарного дерева натуральных чисел* … 40
2.1.4.4. Программная спецификация obj WORDTREE *бинарного дерева слов* … 41
2.1.4.5. Программная спецификация STACK-OF-NAT-1 *стека натуральных чисел с обработкой ошибок* … 42
2.1.4.6. Другая программная спецификация STACK-OF-NAT-2 *стека натуральных чисел с обработкой ошибок* … 42
2.1.4.7. Программная спецификация WORDSTACK *словарого стека* … 43
2.1.5. Декларация атрибутов … 43
2.1.5.1. Программная спецификация PROPC *пропозиционального исчисления* … 44
2.2. Параметризованное программирование … 45
2.2.1. Теории … 46
2.2.2. View (проекция) … 47
2.2.3. Параметризованные модули … 49
2.2.4. Инстанциация … 49
2.2.5. Модульное выражение (Module expression) … 50
2.3. Примеры параметризации … 51
2.3.1. Параметризованная теория линейного векторного пространства с параметром … 51
2.3.2. Параметризованный модуль ORD-PAIR *пар вида (натуральное число, слово)* … 51
2.3.3. Программная спецификация LIST-OF-INT *списков целых чисел* … 52
2.3.4. Программная спецификация LIST-OF-INT1 *списков целых чисел* … 52
2.3.5. Программная спецификация LIST-OF-WORDS *списков слов* … 53
2.3.6. Параметризованный модуль LIST[X :: TRIV]… 54
2.3.7. Программная спецификация nTUPLE … 54
2.3.8. Параметризованный модуль PREPOSET[X :: TRIV] *натуральных чисел* … 56
2.3.9. Параметризованный модуль SEQUENCE[X :: ELEMS] *списков натуральных чисел и списков слов* … 56
2.3.10. Параметризованный модуль PREORDER[X :: TRIV] *предпорядка для чисел натуральных, целых, вещественных* … 57
2.3.11. Параметризованный модуль MONOID[X :: TRIV]

моноида для натуральных чисел: умножение, сложение … 58
2.3.12. Параметризованный модуль `LIST[X :: TRIV]` *моноида списков натуральных чисел и моноида списков слов* … 59
2.3.13. Программная спецификация `FNS, FN-FNS-2[F :: FN]` … 60
2.3.14. Программная спецификация `PROPC-TIME-WIRE-NOT-F` … 61
2.3.15. Программная спецификация булевых свойств множеств (*объединение, пересечение, разность*) … 62
2.3.16. Программная спецификация `LIST` (*список натуральных чисел и список слов*) … 63
2.3.17. Расширенная программная спецификация `LIST` (*список натуральных чисел и список слов*) … 64
2.3.18. Программная спецификация: решето Эратосфена … 65
2.3.19. Сумма и произведение элементов списка натуральных чисел … 66
2.3.20. Сортировка списков слов и пузырьковая сортировка списка натуральных чисел … 67
2.3.21. Сумма и произведение элементов списка натуральных чисел … 69
2.4. Многооконное окружение … 70
2.4.1. Требования (*Requirements*) … 70
2.4.2. Моделирование … 71
2.4.3. Программные спецификации многооконного окружения … 71
Литература … 79

Алексей Александрович НАБЕБИН
Александр Сергеевич ТАРАСИКОВ

АЛГЕБРАИЧЕСКАЯ СПЕЦИФИКАЦИЯ ПРОГРАММНЫХ СИСТЕМ

Издательство ИНЭК

Отпечатано с готовых о/м
в типографии ООО НВП «ИНЭК»
Редактор И.Г.Ерохина
Тираж 500 экз.

117419 Москва, Ленинградское шоссе, д. 18
тел/факс 8-495-786-22-31