Язык алгоритмических суперпозиций ASDIEL

© К. В. Воронцов

17 января 2002 г.

Содержание

1 Введение

Пакет программ ASDIEL предназначен для построения и исследования сложных процедур обработки данных. Аббревиатура ASDIEL означает «Algorithmic Superpositions Description and Investigation Environment and Language» и переводится как «язык и среда для описания и исследования алгоритмических суперпозиций».

ASDIEL — это инструментальное средство, подходящее для решения широкого класса задач в самых разных прикладных областях. Главным образом оно ориентировано на решение задач распознавания, классификации и прогнозирования, называемых также задачами восстановления зависимостей или задачами обучения по прецедентам.

Язык ASDIEL применим в тех случаях, когда требуемая процедура обработки данных допускает представление в виде суперпозиции стандартных алгоритмов. К числу стандартных в ASDIEL относятся алгоритмы аппроксимации, кластеризации, шкалирования, оценивания близости, проецирования, и другие. Некоторые алгоритмы могут использоваться в качестве корректирующих операций над другими алгоритмами. *Язык* ASDIEL *предназначен для описания суперпозиций, составленных из стандартных алгоритмов анализа и преобразования данных.*

На практике состав и структура искомой суперпозиции, как правило, заранее неизвестны — их приходится подбирать в ходе вычислительных экспериментов на реальных данных. При этом описание суперпозиции следует рассматривать как сценарий очередного эксперимента, и только по завершении исследований — как запись решения поставленной задачи. Таким образом ASDIEL *является одновременно языком описания вычислительных экспериментов для задач обработки данных.*

Использование ASDIEL выходит за рамки обычного программирования и подразумевает проведение исследований. По этой причине пользователем языка должен быть квалифицированный аналитик, разбирающийся как в предметной области, так и в вычислительных методах. Необходимо иметь, как минимум, общее представление об их назначении, границах применимости, преимуществах и недостатках (эта информация представлена в документации по библиотеке методов). Язык ASDIEL сам по себе совсем не сложен, однако он требует от пользователя достаточно высокой математической культуры, интуиции и опыта решения прикладных задач.

Почему приходится строить суперпозиции алгоритмов?

Язык ASDIEL задумывался как инструментальное средство, пригодное для решения широкого класса задач. Поэтому в его основу была положена библиотека методов, предназначенных для решения типовых подзадач с максимально широким спектром приложений. При таком подходе решение прикладной задачи сводится к описанию суперпозиции, составленной из нескольких стандартных методов. Построение суперпозиций удобно также в силу следующих, более конкретных, обстоятельств.

- Может случиться, что ни один из стандартных методов не позволяет получить решение нужной точности. В алгебраическом подходе к проблеме распознавания точность решения повышают за счёт коррекции нескольких «плохих» алгоритмов с помощью корректирующей операции, что приводит к образованию алгоритмической суперпозиции.
- Исходные данные могут оказаться разнотипными, неполными, неточными или косвенными. Это потребует предварительной обработки данных с применением одного или нескольких стандартных алгоритмов, например:
  - в случае разнотипных признаков — приведение их к одной шкале;
  - в случае слишком обширного множества объектов — решение задачи кластеризации;
  - при наличии нетипичных объектов — оценивание нетипичности и устранение выбросов;
  - в случае неполных или разреженых данных — выделение наиболее информативной части данных и/или заполнение пропусков.

  Включение алгоритмов предобработки также приводит к возниконовению суперпозиции.
- Некоторые алгоритмы, такие как АВО или МГУА, сами имеют структуру суперпозиции. Причём далеко не всегда выбор этой структуры может быть полностью возложен на машину. Исследователь должен иметь возможность задавать отдельные её части непосредственно или получать с помощью других алгоритмов. Поэтому сложные многоступенчатые алгоритмы целесообразно разделять на более простые, предоставив исследователю возможность вмешаться в процесс постоения такого алгоритма.
- В конкретной задаче может понадобиться то или иное специальное преобразование данных, которое можно реализовать как отдельный алгоритм, применяемый наряду со стандартными.
- Наконец, при передаче данных от одного алгоритма другому часто приходится выполнять различные вспомогательные преобразования, которые также являются отдельными элементами суперпозиции.

Почему для описания суперпозиции нужен специальный язык?

Построение суперпозиции не ограничивается выбором нескольких стандартных алгоритмов и последовательности их выполнения. В нетривиальных прикладных задачах возникают различные дополнительные вопросы, например:

- как выделить из имеющегося множества объектов обучающую и контрольную выборки?
- как отсеять заведомо непоказательные или исключительные объекты?
- какую функцию расстояния между объектами предпочесть?
- как оптимизировать состав признаков, подаваемых на вход того или иного алгоритма?
- какие функциональные преобразования повысят информативность признаков?
- какие значения придать тем параметрам алгоритмов, которые не определяются автоматически в результате обучения?
- какой способ хранения промежуточных данных выбрать, чтобы прийти к компромиссу между скоростью вычислений и размером требуемой памяти?

Ответы на эти вопросы, и даже сам круг вопросов, уникальны в каждом конкретном случае. В основном они касаются структуры суперпозиции и её компонент. Выбор структуры, как правило, наименее формализован и не может быть осуществлён путём автоматической настройки. Чтобы описать суперпозицию, исследователь должен задать:

- состав суперпозиции и последовательность выполнения алгоритмов;
- структуру входных и выходных матриц каждого алгоритма;
- структуру функциональных выражений, определяющих некоторые признаки как функции от других признаков.

Специализированный язык является наилучшим средством для записи структурной информации такого вида. Он позволяет свести её в одном компактном описании и упростить проведение вычислительных экспериментов.

В то же время язык скрывает от исследователя рутинные операции по формированию данных и организации вычислений. Ему не придётся вникать в тонкости хранения данных, реализации алгоритмов и их взаимодействия в составе суперпозиции. Все эти проблемы решены на уровне ядра ASDIEL.

Таким образом, специализированный язык позволяет состедоточиться на содержательных сторонах решаемой задачи.

Ключевые особенности языка ASDIEL

- Благодаря общности модели данных и универсальности библиотечных методов ASDIEL можно использовать в самых разных прикладных областях.

- Средства языка охватывают все этапы обработки данных в задачах восстановления зависимостей: импорт, предварительную обработку, настройку, расчёт, визуализацию и экспорт данных.
- Все библиотечные алгоритмы могут свободно сочетаться друг с другом и с обычными арифметическими выражениями, образуя суперпозиции.
- Входные и выходные матрицы алгоритмов описываются декларативно, как декартовы произведения упорядоченных множеств.
- Данные размещаются в оперативной памяти, в массивах произвольной размерности. Эффективное распределение памяти достигается за счёт отказа от хранения неиспользуемых или легко вычислимых ячеек массивов.
- Суперпозицию можно запускать в различных режимах, в том числе для настройки (`tune`) и вычисления (`calc`). Результатом настройки является суперпозиция с фиксированными параметрами, которую можно сохранить как рабочую ASDIEL-программу.
- Текст ASDIEL-программы можно править во время отладочного запуска. При этом нет необходимости каждый раз перезапускать программу с самого начала — выполнение автоматически возобновляется с места правки.

Как решать прикладные задачи с помощью языка ASDIEL

Язык ASDIEL предназначен в первую очередь для решения нетривиальных задач обработки данных, то есть таких задач, в которых ход решения изначально не ясен и требуется проведение исследований. Ниже перечислены основные этапы решения подобных задач и то, каким образом язык ASDIEL используется на каждом из этапов. Фактически процесс поиска решения сопровождается написанием ASDIEL-программы.

1. Выяснение структуры входных и выходных данных. Приведение всех данных к матричной форме представления, основанной на *наборах* и *массивах*. Описание наборов и массивов командой языка `USE`.
2. Загрузка входных данных из файлов (командой `IMPORT`) или непосредственно из прикладной программы.
3. Поиск и исследование простых зависимостей (в частности зависимостей между отдельными признаками) с помощью встроенных средств визуализации (команды `Table` и `Chart`). Попытки применения стандартных методов из библиотеки ASDIEL для решения поставленной задачи; выяснение причин, по которым отдельные методы не дают качественных результатов.
4. Предварительная обработка данных в случаях неполных, разнородных, сложно структурированных данных. Для этого используются методы предобработки из библиотеки ASDIEL. Выделение выбросов и описание «особенных» областей

путём формирования *поднаборов*. На этом этапе также активно используются средства визуализации.

5. При необходимости — разработка специфических для данной задачи *методов* и *алгоритмов* и их «пристыковка» к ASDIEL в виде динамической библиотеки. Отметим, что ASDIEL поддерживает только алгоритмическую форму представления знаний о предметной области.
6. Построение алгоритмической суперпозиции, при котором учитывается вся информация, полученная на предыдущих этапах исследования. Формулирование и проверка всевозможных гипотез о наличии тех или иных закономерностей в данных, о целесообразности применения тех или иных методов, и т.д. Перебор гипотез осуществляется путём модификации текущего текста ASDIEL-программы, а их проверка — с помощью визуализации промежуточных данных. В процессе построения суперпозиции возможно итеративное возвращение к предыдущим этапам.
7. Настройка (обучение) и отладка суперпозиции. Оценивание её качества на контрольных выборках. Подбор оптимальных форм представления выходной информации. Сохранение настроенной суперпозиции в виде отдельной программы с жёстко фиксированными параметрами каждого метода. Такая программа уже не требует проведения настройки и представляет собой готовый алгоритм.
8. Использование настроенного алгоритма. Технически это реализуется как обращение к ядру ASDIEL из прикладной программы.

1.1 Модель данных и терминология

Приступая к решению задачи, необходимо выделить несколько упорядоченных множеств, которые в дальнейшем будут играть роль размерностей массивов данных. Например, это могут быть множества:

- исследуемых объектов,
- признаков (функций над объектами),
- моментов времени,
- пар объектов,
- функций над парами объектов,
- свойств признаков,
- функций над парами признаков, и т.д.

Затем некоторым декартовым произведениям этих множеств сопоставляются массивы, предназначенные для хранения и представления данных. Все множества и массивы вводятся командой языка `USE`.

С помощью теоретико-множественных операций можно описывать произвольные подмножества и подмассивы, в частности — входы и выходы алгоритмов. Работа

алгоритма состоит в том, чтобы считать данные из входных подмассивов, преобразовать их и занести результат в выходные подмассивы. Однако само преобразование скрыто от пользователя, он обращается с алгоритмом как с «чёрным ящиком». Суперпозиция образуется в том случае, когда на вход одних алгоритмов подаются данные, генерируемые другими алгоритмами.

Терминология ASDIEL

Простой атом — элемент, имеющий имя.

Признак — простой атом, имеющий генератор (см. ниже). Генератором признака может быть алгоритм, выражение или внешняя таблица данных.

Объект — простой атом, не имеющий генератора.

Простой набор — линейно упорядоченное множество простых атомов, либо только объектов, либо только признаков.

Составной набор размерности k — декартово произведение вида $S = S_1 \times \ldots \times S_k$, где S_i — простые наборы, $k \geqslant 2$. Элементы составного набора упорядочены лексикографически.

Составной атом размерности k — элемент составного набора размерности k.

Атом — простой или составной атом.

Набор — простой или составной набор.

Поднабор — последовательность элементов набора (называемого *базовым* для данного поднабора). Один и тот же элемент может входить в поднабор несколько раз. Элементы поднабора не обязаны располагаться в том же порядке, что и в базовом наборе.

Массив $[\![F \times S]\!]$ размерности k — отображение, которое каждому элементу (f, s) декартова произведения $F \times S$ ставит в соответствие значение $\hat{f}(s)$, где:
F — набор признаков;
S — составной набор размерности $k-1$, либо простой набор при $k = 2$;
$\hat{f}$ — отображение вида $S \to T_f$, называемое *генератором* признака f;
T_f — множество допустимых значений признака f, определяемое его типом (булевский, числовой, строковой, поднабор или подмассив).
Каждый признак входит в один и только один набор признаков. Каждый набор признаков определяет один и только один массив.

Подмассив — отображение, определённое на декартовом произведении $F' \times S'_1 \times \ldots \times S'_{k-1}$ и совпадающее на нём с массивом $[\![F \times S_1 \times \ldots \times S_{k-1}]\!]$, где $F' \subseteq F$ — поднабор признаков, $S'_i \subseteq S_i$ — поднаборы атомов.

Понижение размерности — группирование нескольких простых наборов массива в один составной. Например, если имеется массив $[\![F \times S_1 \times \ldots \times S_{k-1}]\!]$ и составной набор $S = S_1 \times \ldots \times S_j$, $j < k$, то массив можно представить в виде $[\![F \times S \times S_{j+1} \times \ldots \times S_{k-1}]\!]$. При этом его размерность понижается с k до $k - j + 1$.

Алгоритм — вычислимое отображение вида $(X_1, \ldots, X_m) \mapsto (Y_1, \ldots, Y_n)$, где
$X_1, \ldots, X_m$ — *входные* подмассивы;
$Y_1, \ldots, Y_n$ — *выходные* подмассивы.
Входные и выходные подмассивы алгоритма называются его *аргументами*.

Метод — совокупность конечного множества алгоритмов и конечного множества параметров. Любой алгоритм метода может свободно считывать и/или изменять значение любого параметра метода.

Метод распознавания — метод, имеющий алгоритмы вычисления и настройки. *Алгоритм настройки*, называемый также алгоритмом обучения, устанавливает значения параметров метода. *Алгоритм вычисления* использует значения параметров, установленные алгоритмом настройки, при вычислении своих выходных подмассивов.

Внутренние параметры — параметры метода распознавания, вычисляемые алгоритмом настройки.

Внешние параметры — параметры метода распознавания, не вычисляемые алгоритмом настройки. Их значения явным образом задаются в программе и могут влиять на процедуру настройки.

1.2 Массивы

Массивы используются для хранения данных и результатов вычислений. Приведём некоторые наиболее типичные примеры массивов.

$[\![F \times S]\!]$ — массив «признаки–объекты», обычно используемый для представления исходных данных в задачах распознавания и статистическом анализе.

$[\![F \times S \times T]\!]$ — массив «признаки–объекты–моменты времени», являющийся обобщением предыдущего на случай динамических задач.

$[\![M \times S \times S]\!]$ — массив для представления функций близости объектов, бинарных отношений на множестве S и других функций над парами объектов.

$[\![M \times S \times S \times T]\!]$ — обобщение предыдущего на случай динамических задач.

$[\![P \times F]\!]$ — массив для хранения различных свойств признаков: диапазонов допустимых значений, весов или информативностей, характеристических векторов подмножеств признаков.

$[\![D \times F \times F]\!]$ — массив для представления функций на парах признаков, например корреляций или близостей признаков.

В каждом массиве имеется одна выделенная размерность — набор признаков, отвечающий за запись и считывание значений из массива. То, каким образом значения вычисляются и хранятся, зависит от способа генерации и способа хранения каждого конкретного признака.

Возможны три способа генерации значений признака.

- Значения признака считываются из текстового файла или таблицы базы данных. Данный способ генерации характерен для признаков, используемых в качестве исходных данных.
- Значения признака вычисляются некоторым алгоритмом — для всех признаков, фигурирующих в выходных подмассивах алгоритмов.
- Значения признака вычисляются по заданному выражению — для всех признаков, правило вычисления которых явным образом задано в программе в виде выражения.

Возможны три способа хранения значений признака.

- Значения признака хранятся в оперативной памяти с прямым доступом — для одномерных или бинарных упакованных признаков.
- Значения признака хранятся в оперативной памяти с индексированным доступом. В текущей реализации этот способ предпочтителен для многомерных разреженных массивов с числом пропусков не менее 80%.
- Значения признака вычисляются заново при каждом обращении, для их хранения память не отводится.

Способ хранения указывается явным образом при создании признака. Пользователь имеет возможность существенно влиять на эффективность вычислений, выбирая способ хранения признаков, используемых для представления промежуточных результатов.

1.3 Алгоритмы

В простейшем случае алгоритм распознавания преобразует входную информацию о заданном подмножестве объектов в выходную информацию о тех же объектах. Это можно представить как вычисление признака $f \in F$ по признакам поднабора F_0:

$$C : [\![F_0 \times S_0]\!] \mapsto [\![\{f\} \times S_0]\!], \quad \text{где} \quad S_0 \subseteq S.$$

Построение суперпозиций возможно благодаря тому, что результаты алгоритмов отождествляются с новыми признаками, которые затем указываются на входе других алгоритмов.

Алгоритм настройки в типичном случае требует указать на входе два подмассива: значения того же поднабора признаков F_0 и значения целевого признака $g \in F$

на фиксированной обучающей выборке S_1. Как правило, алгоритм настройки не имеет выходов, и вся его работа состоит исключительно в настройке внутренних параметров метода:

$$T : [\![F_0 \times S_1]\!],\ [\![\{g\} \times S_1]\!] \mapsto \varnothing, \quad \text{где} \quad S_1 \subseteq S.$$

Некоторые методы имеют алгоритм *дополняющей настройки*, действие которого состоит в расширении обучающей выборки S_1 ещё некоторым количеством объектов. Для многих методов данная операция выполняется гораздо эффективнее, чем полная перенастройка на расширенной выборке.

Кроме упомянутых типов методы могут располагать алгоритмами инициализации параметров, записи и чтения параметров через массивы.

Операция понижения размерности применяется в тех случаях, когда размерность массива, из которого необходимо взять данные, превышает размерность матрицы, требуемой на входе или выходе алгоритма. Например при совмещении второй и третьей размерностей трёхмерный подмассив $M_0 \times S_0 \times S_0$ преобразуется в двумерную матрицу «пары объектов–признаки» $[\![(S_0 \times S_0) \times M_0]\!]$. Такую матрицу, в отличие от трёхмерной, можно задавать на входе методов, применяемых к двумерным массивам типа «объекты–признаки».

2 Команды

Программа на языке ASDIEL представляет собой описание суперпозиции, составленной из стандартных библиотечных методов.

Как правило, программа начинается командами `USE` и `MODES`, задающих массивы и режимы выполнения, а также команд `IMPORT`, загружающих данные из внешних источников.

Затем следуют описания методов в том порядке, в котором они образуют суперпозицию. Описание каждого метода начинается командой `METHOD`, которая делает доступными алгоритмы и параметры данного метода. Обычно выполнение программы сопровождается выдачей промежуточных данных в потоки вывода — файлы, окна или консоли.

Выполнение программы в режиме обучения `tune` приводит к перерасчёту внутренних параметров всех методов и их настройке на текущие данные. Настроенную суперпозицию можно сохранить как новую ASDIEL-программу, при этом команды настройки автоматически заменяются командами явной инициализации параметров расчитанными значениями.

2.1 Лексика языка

Программа состоит из команд, разделяемых символом «`;`». Разбиение текста программы на строки может быть произвольным за исключением команды `SECTION`

(см. 2.10), которая обязана начинаться с новой строки.

Алфавит языка содержит все символы ASCII. Подчёркивание _ и символы с кодами от 128 до 255, в том числе русский алфавит, считаются буквами. Разделителями служат пробел, табуляция и конец строки.

В тексте программы допускаются комментарии двух типов. Комментарий первого типа, заключённый между скобками `!*` и `*!`, может охватывать произвольное число строк. Комментарий второго типа заключается между `!` и концом строки.

Идентификатор — это последовательность букв и цифр, начинающаяся с буквы, либо последовательность любых символов, заключённая в апострофы. В ключевых словах и именах функций соответственные символы верхнего и нижнего регистров считаются тождественными. Во всех остальных случаях символы верхнего и нижнего регистров различны.

Для формального описания грамматики языка будет применяться форма Бэкуса-Наура со следующими условными обозначениями. Названия синтаксических конструкций выделяются. Знак ::= означает «является по определению». Конструкции в фигурных скобках { }, могут быть повторены любое число раз или отсутствовать. Конструкции в квадратных скобках [ ] могут быть опущены. Разделитель | означает выбор одного из вариантов.

2.2 Команда `USE`

Команда `USE` объявляет все используемые в программе наборы и массивы. Она состоит из ключевого слова `USE` и списка описателей. Допускаются описатели трёх типов: простой набор объектов, составной набор и массив. Например:

```
USE S, Pairs=S:S1|S:S2, Main=Func[S], Aux=Metr[Pairs];
```

Описатель `S` объявляет простой набор объектов.

Второй описатель объявляет составной набор $\texttt{Pairs} = \texttt{S} \times \texttt{S}$. Символ вертикальной черты обозначает знак декартова произведения. Если один и тот же набор входит в произведение несколько раз, то соответствующим компонентам произведения приписываются *псевдонимы* — идентификаторы, которые в дальнейшем позволят их различать. Здесь для набора `S` заданы псевдонимы `S1` и `S2`.

Следующий описатель объявляет двумерный массив $\texttt{Main} = [\![\texttt{Func} \times \texttt{S}]\!]$. В квадратных скобках указывается простой или составной набор, перед скобками — набор признаков. Никакие два массива не могут иметь один и тот же набор признаков.

Последний описатель объявляет трёхмерный массив $\texttt{Aux} = [\![\texttt{Metr} \times \texttt{S} \times \texttt{S}]\!]$. Вторая и третья компоненты декартова произведения автоматически получают псевдонимы `S1` и `S2` в наследство от составного набора `Pairs`.

Описание наборов и массивов можно упростить, если воспользоваться умолчаниями команды `USE`.

1. Описатель простого набора можно опустить. Каждый новый идентификатор, появившийся в качестве компоненты составного набора или массива, считается описателем очередного простого набора объектов.
2. Если имя составного набора или массива опущено (вместе со знаком равенства), то оно будет составлено из имён компонент.
3. Если псевдоним опущен, то он будет составлен из имени простого набора и порядкового номера его вхождения в составной набор.
4. Описатель составного набора может быть указан в качестве компоненты другого составного набора или массива.

Приведём описание массивов, рассмотренных в 1.2:

```
USE S,                  ! набор объектов
    FS  = F[S],         ! массив объекты-признаки
    SS  = S:S1 | S:S2,  ! составной набор пар объектов
    MSS = M[SS],        ! массив функций от пар объектов
    PF  = P[F],         ! массив свойств признаков
    FF  = F:F1 | F:F2,  ! составной набор пар признаков
    DFF = D[FF];        ! массив функций от пар признаков
```

Приведём сокращённый, но полностью эквивалентный вариант того же описания:

```
USE F[S], M[S|S], P[F], D[F|F];
```

Поскольку имена наборов будут постоянно встречаться в программе, рекомендуется называть простые наборы одной прописной буквой, а имена составных наборов образовывать из имён компонент. Умолчания команды `USE` придерживаются именно этого стиля.

В заключение приведём формальное определение команды.

USE ::= **USE** Описатель { , Описатель } ;

Описатель ::= Простой | Составной | Массив

Простой ::= НовоеИмя

Составной ::= [НовоеИмя =] Компонента | Компонента { | Компонента }

Компонента ::= Имя : Псевдоним | Имя | Описатель

Массив ::= [НовоеИмя =] НовоеИмя [Имя | Описатель]

2.3 Команда `METHOD`

Описание метода в программе начинается командой `METHOD` и продолжается вплоть до следующего описания.

Основная форма команды `METHOD` имеет вид

METHOD [Библиотека .] Метод ;

где Библиотека — идентификатор динамической библиотеки методов, Метод — идентификатор метода. Если библиотека не указана, метод ищется во всех доступных библиотеках. Указывать имя библиотеки не обязательно, но желательно, так как разные библиотеки могут определять методы с одинаковыми именами.

Например команда

```
METHOD Market.MACD;
```

говорит, что далее следует описание метода с именем MACD, реализация которого находится в динамической библиотеке Market.

Способ реализации библиотек зависит от операционной системы. В настоящее время динамические библиотеки методов доступны только для версии ASDIEL под Windows 95/NT. Динамическая библиотека методов представляет собой файл с именем Библиотека и расширением dll, находящийся в директории %SDLDIR%\bin.

В описании метода, и только в нём, можно обращаться к параметрам и алгоритмам данного метода. Параметры метода доступны точно так же, как обычные переменные, за тем исключением, что их имена содержат символ @. Допустимые имена параметров перечислены для каждого метода в документации по библиотеке методов. Алгоритмы метода описываются и/или вызываются командой описания алгоритма (см. 2.4).

Кроме основной существуют ещё две формы команды METHOD. Первая:

```
METHOD ТипМетода NAMED ИмяМетода ;
```

позволяет присвоить созданному методу имя, заданное идентификатором ИмяМетода. По этому имени метод можно будет снова активизировать в дальнейшем:

```
METHOD NAMED ИмяМетода ;
```

Данная форма команды METHOD используется для того, чтобы повторно обратиться к алгоритмам и/или параметрам метода после того, как были описаны другие методы.

2.4 Команда описания алгоритма

Команда описания алгоритма состоит из имени алгоритма и списка аргументов — входных и выходных подмассивов. Описание относится к последнему методу, объявленному командой METHOD. Запись команды напоминает формальное определение алгоритма (стр. 8), например:

```
calc S|F{f10..f29} -> S|F{result};
```

Имя алгоритма должно совпадать с одним из имён, поддерживаемых данным методом. В документации для каждого метода приводится перечень поддерживаемых алгоритмов с их именами и требованиями к количеству и структуре аргументов. Для методов распознавания это, как правило, алгоритмы:

`tune` — настройка метода;

`calc` — основная функция метода, вычисление выходной информации;

`save` — запись параметров метода в подмассивы;

`load` — загрузка параметров метода из подмассивов;

`more` — дополнительная настройка, расширение обучающей выборки;

В общем случае команда описания алгоритма имеет вид:

Алгоритм ::= ИмяАлгоритма [Аргументы] [-> Аргументы] ;
Аргументы ::= [Подмассив] { , [Подмассив] }

Один из двух списков Аргументы может быть пустым, но не оба сразу. Иногда подмассивы в списке аргументов могут опускаться, однако разделяющие их запятые опускать нельзя. Алгоритм метода не может быть описан более одного раза.

Подмассивы задаются декартовыми произведениями поднаборов (см. 3.4), например:

```
METHOD GenMetrics;
calc S{1..50} | F{x,y,z} -> S{1..50} | S{1..50} | M{rx,ry,rz};
```

Здесь алгоритм `calc` получает на входе двумерный подмассив размером 50×3 и записывает результат в трёхмерный подмассив размером $50 \times 50 \times 3$. Реализация алгоритма `calc` в методе `GenMetrics` такова, что после этого описания признаки `rx`, `ry` и `rz` из `M` будут вычисляться по формулам

$$\left.\begin{aligned} \mathtt{rx}(s_1, s_2) &= |\mathtt{x}(s_1) - \mathtt{x}(s_2)| \\ \mathtt{ry}(s_1, s_2) &= |\mathtt{y}(s_1) - \mathtt{y}(s_2)| \\ \mathtt{rz}(s_1, s_2) &= |\mathtt{z}(s_1) - \mathtt{z}(s_2)| \end{aligned}\right\} \quad \text{для всех } s_1,\ s_2 \text{ из } \mathtt{S\{1..50\}}.$$

После выполнения алгоритма `calc` эти соотношения будут справедливы только для пар объектов из поднабора `S{1..50}`, для остальных пар значения признаков `rx`, `ry` и `rz` будут неопределены. Чтобы алгоритм работал над всеми объектами независимо от того, сколько их содержится в наборе `S`, необходимо использовать свободные размерности.

Размерность подмассива, заданная именем или псевдонимом базового набора, называется *свободной*. В данном случае алгоритм `calc` следовало бы описать так:

```
calc S|F{x,y,z} -> S|S|M{rx,ry,rz};
```

Запуск алгоритма, аргументы которого содержат свободные размерности, откладывается до тех пор, пока не станет ясно, какие конкретные поднаборы должны быть подставлены на место свободных размерностей. Например, при выполнении команды

```
print « S{1..10} | S{51..100} | M{rx,ry,dist};
```

алгоритм `calc` будет запускаться в момент вычисления значений признаков `rx` и `ry`.

Общее правило таково. Если аргументы не содержат свободных размерностей, и в выходных подмассивах нет вычислимых (объявленных как RECALC, см. 3.3) признаков, то алгоритм выполняется немедленно, при этом все значения в выходных подмассивах вычисляются за один раз. В противном случае запуск алгоритма откладывается и выходные значения будут вычисляться по одному по мере необходимости.

Если признак на выходе алгоритма объявлен как хранимый (MEMORY или SPARSE, см. 3.3), то каждое его значение будет вычислено только один раз, и в дальнейшем будет браться из хранилища. По-разному выбирая способы хранения выходных признаков, можно получать различные соотношения между временем выполнения суперпозиции и расходуемой ею памятью.

Команда описания алгоритма назначает его генератором всех признаков, фигурирующих в выходных подмассивах. Если один и тот же признак указан на выходе нескольких алгоритмов, генератором становится тот, который был описан последним.

2.5 Команда MODES и режимы выполнения

Методы имеют, как правило, несколько алгоритмов, выполняющих разные функции. Для того, чтобы вся суперпозиция в целом могла выполнять аналогичные функции, в программу вводятся *режимы выполнения*.

Список всех допустимых режимов задаётся командой MODES в самом начале программы, например

```
MODES +calculate, -tune, +Отладка, -Test;
```

Имена режимов могут быть произвольными идентификаторами, в том числе они могут совпадать с именами алгоритмов. Знак «+» перед именем режима включает его, знак «-» отключает.

Включение и отключение режимов влияет на выполнение блоков MODE–END, например:

```
METHOD LeastSquares;
Базис=F{1,x,y,x*y,x*x,y*y};
MODE tune;
  tune Stune|Базис, Stune|F{goal};
END MODE;
calc S|Базис -> S|F{recalc lsm};
```

Команды внутри блока MODE–END выполняются в том и только том случае, когда в списке режимов команды MODE есть хотя бы один включённый режим.

По умолчанию команда, не находящаяся внутри блока MODE–END, выполняется в режиме Plain. Это позволяет выполнить программу таким образом, чтобы были пройдены только блоки, соответствующие определённому режиму. Например, после команды

```
MODES -Plain, +Load;
```

будут выполняться только блоки `MODE Load` – `END`, а основной текст программы будет проигнорирован.

Формально синтаксис команды `MODES` и блоков определяется по правилам

```
MODES          ::= MODES СписокРежимов ;
СписокРежимов  ::= +|- ИмяРежима { , +|- ИмяРежима }
Блок MODE      ::= MODE ИмяРежима { , ИмяРежима }; Команды END | MODE |;
```

2.6 Команда присваивания

Команда присваивания вычисляет выражение и заносит результат в переменную или элемент массива. Например:

```
z=x*y; avr=(x+y+z)/3;
```

Переменные не нужно описывать заранее. Первое появление переменной слева от знака равенства является одновременно её описанием. Чтобы присвоить значение глобальной переменной, перед её именем необходимо указать ключевое слово `GLOBAL` (см. 2.11).

Если значение выражения не столь интересно, как его побочный результат, имя переменной слева от знака равенства можно опустить:

```
= new memory F{x1, y1, z1};
```

Смысл этой записи в том, чтобы создать в наборе `F` три новых признака и выделить память для их хранения. Результатом выражения является поднабор из трёх элементов, который не присваивается никакой переменной.

Значение выражения можно присвоить элементу массива. Для этого необходимо, чтобы соответствующий признак имел тип хранения `MEMORY` или `SPARSE`. Например:

```
USE F[S], M[S|S];
= new memory F {x,y,z};
= new sparse M {d};
= new S {a::20};
FS[x,a1] = rand();      ! FS - имя массива
F[x,a1] = rand();       ! F - имя набора признаков
x[a1] = rand();         ! x - имя признака
MSS[d,a1,a2] = rand(); ! MSS - имя массива
M[d,a1,a2] = rand();    ! M - имя набора признаков
d[a1,a2] = rand();      ! d - имя признака
```

Элемент массива можно указать любым из трёх способов: используя имя массива, имя набора признаков, или имя признака. В последнем случае признак с таким именем должен быть уникальным.

В общем случае команда присваивания имеет вид

[[GLOBAL] Переменная|ЭлементМассива] = Выражение ;

2.7 Команда вывода в поток

Вывод данных осуществляется через потоки вывода. *Потоком вывода* может служить: текстовый файл, строковой буфер, консоль, окно пользовательского интерфейса (например отладчика), другое приложение (например MS Excel). Команда вывода не зависит от типа потока и имеет вид

Выражение-Поток « Выражение { ,|« Выражение } [,];

Запятая после списка выражений подавляет выводимый по умолчанию символ перевода строки. Значения Выражений могут быть любых типов, включая поднаборы и подмассивы.

Вместо Выражения можно указать одно из ключевых слов:
`FLUSH` — сбросить буфер потока;
`CLOSE` — закрыть поток;
`KERNEL` — вывести в поток отчёт о текущем состоянии ядра.

Обычно все используемые потоки вывода определяются в начале программы или в файле `init.sdl`. Например

```
cout   = ConOutput  (OUT_DEFAULT);
wout   = WinOutput  ("Output", OUT_DEFAULT);
LogOut = FileOutput ("sdl.log", OUT_DEFAULT);
print  = SyncOutput (LogOut, cout, OUT_DEFAULT);
```

Здесь определены четыре потока:
`cout` — стандартный поток вывода, обычно на консоль;
`wout` — поток вывода в окно отладчика;
`LogOut` — поток вывода в файл `sdl.log` и
`print` — поток синхронного вывода на консоль и в тот же файл `sdl.log`.

Для всех потоков установлены стандартные флаги вывода `OUT_DEFAULT` (допустимые значения флагов определены в файле `init.sdl`, который выполняется при запуске ядра ASDIEL и инициализирует необходимые глобальные переменные).

При использовании команды вывода Выражение-Поток обычно является именем потоковой переменной вроде `LogOut` или `print`:

```
Богатыри = S:F{Рост>200 and Вес>120};
LogOut « "Множество богатырей= ", Богатыри,
          ", всего ", card(Богатыри), " человек.";
```

Имеются два потока с предопределёнными именами:
`NulOut` — пустой поток, выводимая информация игнорируется;

`LogOut` — стандартный поток вывода отладочной информации.

Поток `LogOut` можно переопределить в любом месте программы. Команда

```
LogOut = NulOut;
```

подавляет вывод отладочной информации.

Многие методы имеют специальный параметр `@log` потокового типа. Этот поток предназначен для вывода отладочной информации и по умолчанию определён как пустой. Его также можно переопределять с помощью команды присваивания.

2.8 Команды вывода таблиц и графиков

Таблицы и графики выводятся командами `TABLE` и `CHART`. Синтаксис этих команд похож на вывод данных в поток:

TABLE ::= **TABLE** Имя « Выражение { , Выражение } ;
CHART ::= **CHART** Имя « Выражение { , Выражение } ;

Здесь Имя — строковое выражение, Выражение — выражение любого типа, как правило строка, подмассив или параметр метода (возможно, векторный или матричный). В командах `TABLE` и `CHART`, в отличие от вывода в поток, недопустимы ключевые слова `FLUSH`, `CLOSE` и `KERNEL`.

В текущей реализации просмотр таблиц и графиков осуществляется отдельными программами. Описание таблицы или графика сначала выводится в текстовый файл Имя, затем для его просмотра запускается программа-вьюер.

Имя программы-вьюера должно быть прописано в файле `init.sdl`, например:

```
global TABLEVIEW = ValidPath (SDLDIR & "bin/tabView.exe");
global CHARTVIEW = ValidPath (SDLDIR & "bin/chdView.exe");
```

При формировании графиков можно пользоваться полным набором команд формата CHD (CHart Description). При этом полезно помнить, что строковые константы могут переходить с одной строки на другую, и каждый переход записывается в строковую константу в виде символа разрыва строки. Поэтому допустима запись:

```
CHART "test_lsm.chd" « "
  #Format = X Y C=1 / Y C
  #Title = Test of LeastSquares
  #XName = x
  #YName = y
",
  Points|F{x,y,yLsm,IsTrain+5};
```

2.9 Команда IMPORT

Команда IMPORT считывает данные из текстового файла и загружает их в подмассив. Например, следующая команда загружает матрицу размера 100×3 из файла SF.dat:

```
IMPORT "SF.dat" » S{1..100}|memory F{x1,y1,z1};
```

В общем случае команда имеет вид

IMPORT [WIDE|NARROW] [NONEW|SPARSE] ИмяФайла » Подмассив ;

где ИмяФайла — выражение строкового типа, значение которого должно быть именем файла. Опции NARROW и WIDE несовместимы. Опция SPARSE несовместима с NONEW и WIDE.

Опция NONEW подавляет создание новых атомов в простых наборах; при этом данные, соответствующие этим атомам в файле, игнорируются.

Опция SPARSE задаёт способ хранения sparse для всех признаков, создаваемых в процессе чтения «узкой» таблицы.

Для успешной загрузки необходимо выполнение следующих условий:

- каждая строка файла разбивается на поля с помощью символов-разделителей — табуляций или запятых;
- все строки содержат одинаковое число полей, формируя тем самым разбиение файла на колонки;
- все признаки в подмассиве имеют тип хранения memory или sparse.
- подмассив и файл специальным образом согласованы (допустимые типы согласования рассмотрены ниже);

Допускается импортирование четырёх типов таблиц.

1. Самый простой случай импорта — когда опции NARROW и WIDE опущены. Тогда загружается двумерный подмассив фиксированного размера.

Подмассив и файл должны быть согласованы следующим образом:

- подмассив должен быть двумерным;
- число строк файла должно совпадать с первой размерностью подмассива;
- число колонок файла должно совпадать со второй размерностью подмассива;

2. Опция NARROW задаёт импорт подмассива как «узкой» таблицы. Каждая строка такой таблицы содержит одно и только одно значение, которое находится в последнем поле. Все остальные поля содержат имена атомов — координаты данного значения в массиве.

Подмассив и файл должны быть согласованы следующим образом:

- в подмассиве все размерности либо свободные, либо состоят из одного атома;
- все колонки файла кроме последней соответствуют свободным размерностям подмассива и содержат имена атомов;

- последняя колонка файла соответствует значениям признаков.

Таким образом, подмассиву с k свободными размерностями должен соответствовать файл из $k+1$ колонок:

S_1	S_2	...	S_k	Значение

3. Опция `WIDE` задаёт импорт подмассива как «широкой» таблицы. Этот тип таблицы отличается от предыдущего тем, что каждая строка содержит значения нескольких признаков.

Подмассив и файл должны быть согласованы следующим образом:

- в подмассиве все размерности кроме последней либо свободные, либо состоят из одного атома;
- в файле одна или несколько первых колонок соответствуют свободным размерностям подмассива и содержат имена атомов;
- остальные колонки файла соответствуют признакам;
- в подмассиве последняя размерность должна быть поднабором признаков.

Иными словами, если подмассив имеет k свободных размерностей и поднабор из n признаков, то ему должен соответствовать файл из $k+n$ колонок:

S_1	S_2	...	S_k	f_1	f_2	...	f_n

4. Имеется также специальный вариант команды `IMPORT`, предназначенный для импорта атомов простого набора:

`IMPORT ATOMS` ИмяФайла » ИмяПростогоНабора ;

Файл должен содержать имена атомов, разделённые символами табуляции, конца строки или запятыми.

2.10 Команда `SECTION`

С помощью команды `SECTION` программу можно разбить на разделы. Такое разбиение не является обязательным, однако оно упрощает работу с программой на стадии разработки и отладки.

Оболочка интерпретатора ASDIEL использует список разделов в качестве одного из способов отображения программы. При удачном разбиении на разделы этот список отражает «укрупнённую» структуру суперпозиции алгоритмов.

Второе и главное назначение команды `SECTION` состоит в том, чтобы ускорить проведение серий вычислительных экспериментов в тех случаях, когда необходимо пересчитать только часть промежуточных данных. Если во время отладки пользователь приостановил выполнение программы и внёс исправления в её текст, то возобновлять её выполнение с самого начала не обязательно. Отладчик позволяет сделать

откат к началу того раздела, в котором была произведена правка, восстановив состояние программы на момент входа в данный раздел. Таким образом можно избежать повторного выполнения начальной части программы, не затронутой исправлениями. Вообще, командой SECTION рекомендуется завершать участки программы, выполняющие громоздкие вычисления, связанные с ощутимыми затратами времени.

Команда SECTION имеет вид

SECTION | ИмяРаздела | ;

Строковая константа ИмяРаздела даёт разделу имя, которым он будет обозначаться в пользовательском интерфейсе интерпретатора ASDIEL:

```
SECTION "Формирование плоского представления для визуализации"
```

Если имя опущено, в качестве названия раздела будет фигурировать первый комментарий, встретившийся внутри раздела.

На использование команды SECTION накладываются два ограничения: Во-первых, она обязана начинаться в первой позиции строки. Во-вторых, она не может разбивать блоки команд MODE–END, WHILE–END, FOR–END, IF–END.

2.11 Области видимости

Ядро ASDIEL позволяет иметь несколько программ (модулей), работающих с одной и той же суперпозицией алгоритмов и разделяющих общие массивы и переменные. Однако в каждый момент времени активным может быть только один модуль.

Язык не накладывает никаких ограничений на использование в модуле массивов, объектов, признаков и переменных, созданных другим модулем. В этом смысле все модули равноправны. Единственное требование заключается в том, чтобы все атомы и переменные были созданы до первого обращения к ним. Пользователь, применяющий несколько модулей для описания одной суперпозиции, должен сам следить за тем, чтобы они выполнялись в правильной последовательности, не нарушающей этого условия.

Все массивы и базовые наборы, определяемые командой USE, а также методы, определяемые командой METHOD, являются глобальными и доступны всем модулям.

Переменные, определяемые командой присваивания, по умолчанию являются локальными в данном модуле. Чтобы присвоить значение глобальной переменной, перед её именем необходимо поставить ключевое слово GLOBAL. Локальная переменная не может иметь то же имя, что и глобальная.

Режимы выполнения, определяемые командой MODES, являются локальными и не влияют на выполнение других модулей.

2.12 Запись настроенной программы. Команда PARAMETERS

После того, как программа полностью выполнена и все описанные в ней методы настроены, её можно сохранить в отдельном файле, записав в него (вместе с текстом самой программы) рассчитанные значения параметров методов. Этот способ записи программы используется для того, чтобы получить окончательный «рабочий» вариант алгоритмической суперпозии, которую можно было бы запускать без предварительной настройки. Настроенная программа получается из текста исходной программы следующим образом.

Во время выполнения программы, встречая команду

```
PARAMETERS;
```

интерпретатор запоминает текущие значения параметров всех методов, настроенных к данному моменту.

Во время сохранения все команды переносятся из исходной программы без изменений за исключением команды PARAMETERS, вместо которой вставляется блок

```
MODE parameters;
  METHOD NAMED ...;
      @param = value;
      ...
  METHOD NAMED ...;
      @param = value;
      ...
END MODE;
```

Внутри блока находятся команды двух типов: активизация методов METHOD NAMED и присваивание значений параметрам метода. Параметры каждого метода записываются ровно один раз независимо от того, сколько раз встретилась команда PARAMETERS.

Значения параметров соответветствуют тому моменту времени, когда выполнялась команда PARAMETERS. Если они были изменены позже, то на сохраняемую программу это уже не повлияет.

Все методы, параметры которых предполагается записывать при сохранении настроенной программы, должны быть созданы командой METHOD с опцией NAMED.

Команда PARAMETERS не может находиться внутри блоков MODE–END, WHILE–END, FOR–END, IF–END. В противном случае интерпретатор выдаст сообщение об ошибке и выполнение программы будет прервано. Команду PARAMETERS рекомендуется вставлять в текст программы сразу после блока настройки каждого метода, имеющего внутренние параметры:

```
MODE tune;
  ...
```

```
END MODE;
PARAMETERS;
```

2.13 Команды FOR, WHILE, IF

Команды FOR, WHILE, IF и END позволяют менять естественный порядок выполнения программы. *Используйте циклы* FOR *и* WHILE *только при крайней необходимости. В языке имеется достаточное количество декларативных средств, позволяющих избежать их явного применения.*

Цикл WHILE выполняется до тех пор, пока ЛогическоеВыражение истинно. Если его значение ложно при первом входе в цикл, Команды не выполняются ни разу.

WHILE ЛогическоеВыражение ; Команды ; END [WHILE];

При выполнении цикла FOR Переменная цикла пробегает все атомы из заданного Поднабора, и для каждого атома выполняются заданные Команды.

FOR Переменная IN Поднабор ; Команды ; END [FOR];

Если поднабор пуст, тело цикла не выполняется ни разу.

Переменная цикла принимает целочисленные значения — номера атомов поднабора в базовом наборе. Нумерация атомов в наборах всегда начинается с нуля. Например, результатом программы

```
USE S;
= new S{x,y,s::100};   !создали 102 атома: x, y, s1..s100
count = 1;
FOR i IN S{s38,s67,y,x};
print « count, ") ", i, " - ", S{#i};
count = count + 1;
END FOR;
```

будет распечатка

```
1) 39 - S{s38}
2) 68 - S{s67}
3) 1 - S{y}
4) 0 - S{x}
```

Команда IF выполняет набор команд Команды-И, если Выражение истинно, либо набор команд Команды-Л, если оно ложно.

IF Выражение ; Команды-И ; [ELSE; Команды-Л ;] END [IF];

2.14 Команда EXEC

Команда **EXEC** запускает внешнее приложение.

EXEC [**WAIT**] [КомандняяСтрока] [**DIR** ИмяДиректории] ;

Здесь КомандняяСтрока и ИмяДиректории — произвольные строковые выражения.

Если указана опция **WAIT**, выполнение программы будет продолжено только после того, как отработает запущенное приложение. Опция **DIR** задаёт текущую директорию для запускаемого приложения.

2.15 Команда RUN

Команда **RUN** вычисляет строковое выражение, и, рассматривая его как отдельную ASDIEL-программу, выполняет его.

RUN СтроковоеВыражение ;

Команда **RUN** позволяет выполнить другую программу, записанную в отдельном файле:

```
RUN file("my_init.sdl");
```

Функция **file** загружает содержимое файла и возвращает его как строковую переменную.

2.16 Команда STOP

Команда **STOP** прерывает выполнения программы. После прерывания управление передаётся оболочке. Работая с отладчиком, пользователь имеет возможность продолжить выполнение программы с команды, непосредственно следующей за **STOP**.

3 Выражения

3.1 Типы данных

Переменные, признаки, аргументы функций и параметры алгоритмов могут иметь любой тип из числа перечисленных ниже.

- Целые типы: логический **BOOL**, 2 бита **BIT2**, полбайта **BIT4**, байт **BYTE**, короткое целое **SHORT**, длинное целое **LONG**.
- Вещественные типы: одинарной **SINGLE** и двойной **DOUBLE** точности.
- Хронологические: дата **DATE**, время **TIME**, дата и время **DATETIME**.
- Строки произвольной длины **STRING**.
- Поднабор **SUBSET**.
- Подмассив **ARRAY**, образуется декартовым произведением поднаборов.

При хранении признаков типа BOOL, BIT2 или BIT4 используется битовая упаковка. При вычислениях все целые типы преобразуются к INT, который является синонимом LONG, а все вещественные — к REAL, который является синонимом DOUBLE.

Правила совместимости типов.

Все числовые типы полностью совместимы друг с другом и со строковым типом. При приведении вещественных типов к целому происходит округление. При приведении целых к более коротким целым старшие биты отбрасываются. При приведении числовых типов к BOOL положительные значения интерпретируются как 1, отрицательные и нуль — как 0.

Значение любого типа можно преобразовать в STRING. Если строковое значение оказывается операндом числовой операции, оно наиболее естественно приводится к типу REAL. Например, 2+"5" равно 7.

Типы DATE, TIME и DATETIME эквивалентны за исключением формы вывода: в первом случае выводится только дата, во втором — только время, в третьем — дата и время. Значения этих типов полностью совместимы с типом LONG, и к ним применимы операции сложения и вычитания. Разность двух дат выражается в секундах. Констант хронологического типа в языке не существует, однако можно пользоваться строковой записью в формате "гг/мм/дд чч:мм:сс". Символьные названия месяцев не допустимы.

Тип ARRAY является частным случаем SUBSET. При приведении строкового типа к SUBSET строка интерпретируется как ASDIEL-выражение, которое необходимо вычислить.

Для явного приведения типа в выражениях используется функциональная запись. Каждому типу данных соответствует одноимённая функция, преобразующая свой единственный аргумент к этому типу. Например выражение int(X/10)*10 вычисляет ближайшее к заданному вещественному X целое, кратное 10.

Строковые константы заключаются в кавычки. Внутри строковой константы допустимы escape-последовательности:

\a — beep, звуковой сигнал;
\b — перевод курсора назад;
\n — перевод строки;
\t — табуляция;
\v — вертикальная табуляция;
\xHH — символ с 16-чным кодом HH
\ddd — символ с 10-чным кодом ddd, начальные нули обязательны;
\[— символ [;
\] — символ].

Внутри строки допустимы вставки — произвольные выражения, заключённые в скобки [и]. Например, строковая константа "aaa[2*2]bbb" равна "aaa4bbb".

Двоичные константы имеют вид `0b1100.1001.1001`. Они могут содержать произвольное число точек, которые используются для отделения разрядов и при оценивании константы игнорируются.

Шестнадцатиричные константы имеют вид `0xFF`, `0xAF71`, `0xFFFFFFFF`.

3.2 Приоритет операций

Выражения строятся с помощью скобок и знаков операций из констант, переменных, имён признаков и вызовов функций. В таблице 1 перечислены в порядке старшинства все операции, допустимые в выражениях. Для каждой операции указан тип аргументов и результата. Операции, находящиеся в одной графе, имеют одинаковый приоритет и выполняются слева направо.

В отличие от большинства языков программирования цепочки равенств и/или неравенств имеют обычный математический смысл. Например, следующие два выражения эквивалентны:

```
0 <= x == z < 1
0<=x and x==z and z<1
```

Операция	Результат	Назначение
real `^` *real*	*real*	возведение в степень
real `*` *real*, *real* `/` *real*	*real*	умножение и деление
int `%` *int*	*int*	остаток от деления целых
subset `*` *subset*	*subset*	пересечение поднаборов
real `+` *real*, *real* `-` *real*	*real*	сложение и вычитание
subset `+` *subset*, *subset* `-` *subset*	*subset*	объединение и разность поднаборов
subset `&` *subset*	*subset*	конкатенация поднаборов
string `&` *string*	*string*	конкатенация строк
subset `:` *subset*	*subset*	выборка элементов поднабора
subset `$` *subset*	*subset*	сортировка поднабора
subset `|` *subset*	*subset*	декартово произведение поднаборов
real `==` *real*	*bool*	арифметическое сравнение
subset `==` *subset*		сравнение поднаборов
аналогично: `>`, `<`, `>=`, `<=`, `<>`		
`not` *bool*	*bool*	логическое НЕ
bool `and` *bool*	*bool*	логическое И
bool `or` *bool*	*bool*	логическое ИЛИ

Таблица 1: Сводка операций

3.3 Поднаборы-перечисления

Перечисление — это базовая конструкция, с помощью которой создаются поднаборы. Оно представляет собой список элементов, заключённый в фигурные скобки. Перед открывающей фигурной скобкой указывается имя или псевдоним базового набора. Элементом списка может быть:

- имя атома;
- символ `#`, за которым следует порядковый номер атома от 0 до $(n-1)$, где n — число атомов в базовом наборе;
- символ `%`, за которым следует относительный номер атома, выраженный вещественным числом от 0 до 100;
- диапазон атомов;
- серия атомов.

Например, поднабор

```
S {x, y, #10, %100}
```

состоит из четырёх атомов набора `S`: `x`, `y`, 10-ого и последнего атома в наборе.

После знаков `#` и `%` может следовать любое числовое выражение. Выражение после `#` будет автоматически преобразовано к типу `INT`. В поднаборах объектов знак `#` можно опускать. Отрицательные номера указывают на нулевой атом. Номера, превышающие число атомов в базовом наборе, указывают на последний атом.

Диапазон $a_1..a_2{:}k$ добавляет в поднабор все атомы, расположенные в базовом наборе между a_1 и a_2 включительно с шагом k. Диапазон, у которого a_2 предшествует a_1, будет проигнорирован. Если шаг опущен, атомы выбираются подряд. Начало, конец и шаг диапазона можно указывать любым из трёх возможных способов и в любом сочетании:

```
S {x1..x10, y..#84:2, z, %0..%80:%8}
```

Серия образуется атомами, имена которых удовлетворяют заданному образцу. *Образец* — это имя, содержащее ровно один символ `*`, например: `r*`, `*dist`, `metric*x`. Описание серии имеет следующий синтаксис:

Образец `::` Выражение

Значение Выражения должно быть либо целым числом, либо диапазоном чисел, либо поднабором, либо подмасивом.

Если Выражение — это целое число n, то вместо `*` будут по очереди подставлены все числа от 1 до n включительно. Например, следующие выражения эквивалентны:

```
S{obj1, obj2, obj3, obj4, obj5, obj6, obj7, obj8}
S{obj*::8}
S{obj::8}
```

Если символ * пропущен, он автоматически присоединяется в конец образца.

Если Выражение — это диапазон чисел вида $n_1..n_2:k$, то вместо * будут по очереди подставлены числа от n_1 до n_2 включительно с шагом k. Если диапазон имеет более простой вид $n_1..n_2$, предполагается единичный шаг. Например, Sx::0..33:5 — то же самое, что

```
S{x0, x5, x10, x15, x20, x25, x30}
```

Отметим, что при $n_1 = k = 1$ этот случай соответствует предыдущему, так что следующие выражения эквивалентны:

```
S{obj::8}
S{obj::1..8}
S{obj::1..8:1}
```

Если Выражение является поднабором, вместо * будут по очереди подставлены имена всех атомов данного поднабора. Например, если `Five=F{x,y,z,result,amount}`, то следующие две команды эквивалентны:

```
Mfive = M{*Dist::Five};
Mfive = M{xDist, yDist, zDist, resultDist, amountDist};
```

Если Выражение является подмассивом, вместо * будут по очереди подставлены все элементы подмассива, причём повторяющиеся значения учитываются только один раз:

```
set_of_clusters = K{new clust::S|F{cluster}};
```

Если признак `cluster` принимает значения, скажем, 0, 1 и 2, то в наборе `K` будут созданы атомы `clust1`, `clust2` и `clust3`.

Составные поднаборы также можно задавать перечислением элементов. Составной атом записывается как последовательность простых атомов, разделённых символом декартова произведения `|`. Число простых атомов должно совпадать с размерностью базового набора, а сами атомы — принадлежать соответствующим простым наборам. Составной атом не имеет собственного идентификатора, и данная запись фактически является его именем.

Пусть было определено `USE F[S|T]`. Тогда выражение

```
ST{Липецк|май95, Тверь|сен95, Курск|%100}
```

задаёт составной поднабор из трёх атомов: `Липецк|май95`, `Тверь|сен95` и `Курск|%100`. Атомы `Липецк`, `Тверь` и `Курск` должны принадлежать набору объектов S, а атомы `май95` и `сен95` — набору объектов T.

Серия не может быть элементом составного поднабора-перечисления.

Признаки-выражения. Элементом перечисления признаков может быть произвольное выражение. В таком случае будет создан новый признак, и его значение

всегда будет вычисляться по этому выражению. Именем признака по умолчанию станет строковая запись выражения, из которой удалены все пробелы. Можно дать ему другое имя, указав перед выражением идентификатор и знак равенства. Например поднабор

```
F {x, y, x+y, Euclid=sqrt(x^2+y^2+z^2) }
```

состоит из 4 признаков, причём третий `x+y` и четвёртый `Euclid` будут созданы в процессе трансляции поднабора. Любые признаки базового набора могут входить в выражение подобно обычным переменным.

При обнаружении синтаксической ошибки в признаке-выражении выполнение программы прерывается. Если же ошибка произошла при вычислении выражения (например, деление на 0), результат полагается равным `EMPTY`, и программа продолжает выполняться.

Создание новых атомов. Чтобы создать новый объект, достаточно записать его имя в качестве элемента перечисления и поставить перед ним ключевое слово `NEW`. Если объекта с таким именем ещё нет в базовом наборе, он будет создан и добавлен последним элементом.

Ключевое слово `NEW` в выходных подмассивах алгоритмов можно не указывать. В этом случае все новые признаки создаются автоматически.

Если ключевое слово `NEW` указать перед серией атомов, то все недостающие атомы будут созданы и добавлены в конец набора в том порядке, в котором они перечислены в серии. Например, если набор `S` пуст, то

```
S{new s::8}         ! создаёт s1,s2,s3,s4,s5,s6,s7,s8
S{new s4, new s::8} ! создаёт s4,s1,s2,s3,s5,s6,s7,s8
```

Это правило не распространяется на диапазоны атомов.

Ключевое слово `NEW` можно вынести за скобки и поставить перед поднабором-перечислением. Тогда его действие распространяется на все атомы поднабора.

Типы и способы хранения признаков указываются перед именем признака и разделяются пробелами.

Признак может храниться одним из трёх способов:

- `MEMORY` — значения признака сохраняются в оперативной памяти для всех атомов. Этот способ приемлем для одномерных или упакованных бинарных признаков.
- `SPARSE` — значения признака сохраняются в оперативной памяти только для некоторых атомов. Для доступа к данным используется индексирование. В текущей реализации этот способ хранения предпочтительнее для многомерных разреженных массивов с числом пропусков не менее 80%.
- `RECALC` — значения признака не хранятся, а вычисляются при каждом обращении.

Если тип признака опущен, предполагается `SINGLE`. Если опущен способ хранения, предполагается `RECALC` для признаков-выражений, `MEMORY` для результатов алгоритмов в двумерных массивах и `SPARSE` для результатов алгоритмов в массивах большей размерности.

Типы и способы хранения признаков, также как и ключевое слово `NEW`, можно ставить перед серией и выносить перед поднабором-перечислением. Ключевое слово внутри поднабора подавляет действие внешнего.

Приведём в заключение формальное определение синтаксиса поднабора перечислительного типа.

Перечисление ::= [Хранение] ИмяБазовогоНабора { Элемент { , Элемент } }
Элемент ::= Признак | Объект | Серия
Признак ::= [Хранение] [Тип] [[Имя] =] Выражение
Признак ::= [Хранение] [Тип] Имя
Объект ::= NEW Атом
Объект ::= Атом . . Атом [:Число]
Хранение ::= NEW | MEMORY | DATABASE | RECALC

3.4 Операции над поднаборами

Всего имеется девять операций для построения поднаборов: пересечение, объединение, конкатенация, разность, дополнение, декартово произведение, понижение размерности, выборка и сортировка.

Теоретико-множественные операции. Пересечение, объединение, разность и дополнение образуют полный набор теоретико-множественных операций над поднаборами. Эти операции применимы только к поднаборам одного и того же базового набора.

Конкатенация `A&B` — это поднабор, состоящий из элементов `A`, за которыми следуют все элементы `B`. Порядок элементов в `A&B` тот же, что в поднаборах `A` и `B`. Число элементов (мощность) поднабора `A&B` всегда равна сумме мощностей `A` и `B`.

Объединение `A+B` — это поднабор, состоящий из элементов `A`, за которыми следуют все элементы `B`, не принадлежащие `A`. Порядок элементов в `A+B` тот же, что в поднаборах `A` и `B`. Мощность поднабора `A+B` может оказаться меньшей, чем сумма мощностей `A` и `B`.

Пересечение `A*B` — это поднабор, состоящий из элементов `A`, принадлежащих `B`. Порядок элементов в `A*B` тот же, что и в `A`.

Разность `A-B` — это поднабор, состоящий из элементов `A`, не принадлежащих `B`. Порядок элементов в `A-B` тот же, что и в `A`.

Дополнение `-A` — это поднабор, состоящий из всех элементов базового набора, не принадлежащих `A`. Порядок элементов на `-A` совпадает с порядком на базовом наборе.

Декартово произведение поднаборов определяет составной поднабор. Оно также может трактоваться как подмассив, если его структура совпадает со структурой некоторого массива. Например, если задано `USE M[S|S]`, то из трёх произведений

```
M{x,y} | S{1..10}
M{x,y} | S{1..10} | S{1..10}
M{x,y} | S{1..10} | S{1..10} | S{1..10}
```

первое и третье являются составными поднаборами, а второе — подмассивом. Только подмассивы могут задаваться в качестве входов и выходов алгоритмов.

Порядок следования сомножителей влияет на *ориентацию* порождаемого подмассива. Например, подмассивы `F{x,y,z}|S{1..20}` и `S{1..20}|F{x,y,z}` содержат одни и те же данные, но имеют разные размеры: 3×20 и 20×3 соответственно.

Понижение размерности. Размерность подмассива можно понизить, сгруппировав несколько сомножителей декартова произведения в составной поднабор. Делается это с помощью обычных скобок.

Например в выражении

```
S{1..10} | S{10..20} | M{dist}
```

третий поднабор состоит только из одного элемента, но несмотря на это оно порождает трёхмерный подмассив размера $10 \times 11 \times 1$. Чтобы получить двумерную матрицу, следует записать

```
S{1..10} | (S{10..20} | M{dist})
```

Результатом будет двумерный подмассив размера 10×11. Возможна иная расстановка скобок. Например,

```
(S{1..10} | S{10..20}) | M{dist}
```

порождает двумерную матрицу размером 110×1, у которой строки соответствуют парам объектов, а столбец — признаку `dist`.

Операция выборки задаёт поднабор условием вида «все элементы $x \in S$, для которых условие $P(x)$ истинно». Выборка элементов набора `X` возможна только в том случае, если определён массив вида `Y[X]`. Тогда в качестве предиката $P(x)$ допускается любой признак из `Y`, который может быть преобразован к типу `BOOL`.

Операция выборки является бинарной: первым операндом должен быть поднабор, из которого выбираются элементы, вторым — поднабор признаков, задающих условие выборки. Если признаков несколько, условием будет результат их логического И. Пусть задано `USE F[S]`. Тогда выражение

```
S : F{2<x<3}
```

определяет поднабор объектов, для которых `x` находится в интервале $(2, 3)$.

В условный поднабор включаются те и только те атомы, для которых условие истинно. Если условие не определено или ошибочно (имеется в виду ошибка вычисления, вроде деления на 0, но не синтаксическая), атомы не включаются в результирующий поднабор.

Выборка элементов составного набора может потребовать ощутимых затрат времени уже в массивах небольшой размерности. Один из выходов состоит в том, чтобы по возможности сразу ограничить область перебора, указав некоторый вполне обозримый поднабор. Например выражение

```
(S{1..10}|S{10..20}):M{dist>10} | M{dist,r1,r2}
```

потребует перебора только 110 пар объектов.

Отметим, что скобки здесь обязательны. Приоритет операции произведения ниже, чем у операций выборки и сортировки. Поэтому скобки можно не ставить, когда выборка или сортировка применяется к простым поднаборам, но не в данном случае.

Операция сортировки упорядочивает поднабор объектов по возрастанию заданных признаков. Подобно операции выборки, она применима, если существует массив вида `Y[X]`, где `X` — базовый набор упорядочиваемого поднабора.

Пусть имеется массив `F[S]` и набор `F` содержит признаки `Date`, `Security` и `Price`. Тогда выражение

```
S{1..30} $ F{Date, Security, Price}
```

задаёт поднабор из первых 30 объектов набора `S`, расположенных по возрастанию значения признака `Date`. Если несколько объектов имеют равные значения первого признака, то для сортировки будет использован второй (в данном случае `Security`), затем третий, и так далее. Чтобы упорядочить не по возрастанию, а по убыванию, следует задать

```
S{1..30} $ F{-Date, -Security, -Price}
```

3.5 Доступ к элементам массивов

Значение любого элемента массива можно считать с помощью операции `[]`. Значение элемента можно изменить с помощью той же операции и команды присваивания (стр. 16), если соответствующий признак является хранимым.

Допустим, при формировании подмассива необходимо получить значение из другого массива. Проиллюстрируем это на примере определения метрики. Зададим массивы `F[S]` и `M[S|S]`. Требуется определить функцию от пары объектов $r(s_1, s_2) = |x(s_1) - x(s_2)|$, где $(s_1, s_2) \in S \times S$, а x — признак из набора `F`. Делается это одним их трёх способов:

```
M {r=abs(FS[x,S1]-FS[x,S2])}
M {r=abs(F[x,S1]-F[x,S2])}
```

```
M {r=abs(x[S1]-x[S2])}
```

Операция `F[x,S1]` или `x[S1]` выдаёт значение признака `x` из массива `F[S]`. Последняя запись короче, но для её надёжного использования необходимо, чтобы признак с именем `x` имелся только в наборе `F`. Псевдонимы `S1` и `S2` в данном случае воспринимаются интерпретатором не как набор `S`, а как порядковые номера текущих атомов по первой и второй размерности массива `M[S|S]`.

Текущий атом вводится для каждой размерности массива во время формирования подмассива. Если массив имеет размерность k, то с ним связывается ровно столько же текущих атомов. В совокупности они пробегают по всем элементам формируемого подмассива.

Общее правило интерпретации псевдонимов наборов таково. Если псевдоним стоит на месте атома, он интерпретируется как номер текущего атома в соответствующем наборе. Во всех остальных случаях он обозначает базовый набор.

Для того, чтобы гарантированно получить номер текущего атома, а не базовый набор, используется конструкция Массив.Набор, где Массив — имя массива или набора признаков (напомним, что набор признаков однозначно соответствует массиву), Набор — имя или псевдоним простого набора, указывающий размерность в этом массиве. Номер текущего атома всегда имеет тип `INT`.

Отметим, что текущий атом относится не к простому набору в отдельности, а к размерности массива, то есть к паре «простой набор — массив». Если обращение к текущему атому встретилось при описании признака-выражения, то в качестве массива берётся тот, в котором этот признак находится. В случае определения метрики это как раз то, что нужно. Если же возникает неоднозначность, необходимо явно указать, в каком массиве и по какой размерности взять текущий атом, например:

`FS.S` или `F.S` — текущий атом по размерности `S` массива `FS`;

`MSS.S1` или `M.S1` — текущий атом по размерности `S1` массива `M[S:S1|S:S2]`;

`MSS.S2` или `M.S2` — текущий атом по размерности `S2` массива `M[S:S1|S:S2]`. Для этой же цели можно использовать функцию `CurrAtom`.

Операцию `[]` можно применить для доступа к элементу того же массива. Выделим, например, все пары (s_1, s_2) из S, для которых $d(s_1, s_2) \neq d(s_2, s_1)$, где `d` — признак из набора `M`:

```
npairs = S|S : M{d<>d[S2,S1]};
```

Ещё один пример — формирование предыстории признака:

```
F{x, x[S-1], x[S-2], x[S-3]}
```

Конструкция `x[S-1]` выдаёт значение признака `x` для предыдущего объекта базового набора `S`.

4 Функции

Функции языка ASDIEL допускают в общем случае произвольное число аргументов произвольных типов. Вызов функции имеет традиционный синтаксис:

ИмяФункции ([Аргумент { , Аргумент }])

Аргумент функции может быть произвольным выражением. Если функция не имеет аргументов, после имени необходимо оставить пустые скобки. Соответственные символы верхнего и нижнего регистров считаются тождественными в именах функций.

Условные обозначения в описаниях функций:

A — (array) подмассив, тип `ARRAY`;
B — (base set) имя или псевдоним базового набора;
C — (character string) символьная строка, тип `STRING`;
D — (datum) значение любого типа;
I — (integer) целое число, тип `INT`;
L — (logical) логическое `true` или `false`, тип `BOOL`;
O — (output stream) поток вывода;
P — (parameters) вектор параметров;
R — (real) вещественное число, тип `REAL`;
T — (time) дата и/или время, типы `TIME`, `DATE` и `DATETIME`;
S — (subset) поднабор, тип `SUBSET`;
V — (vector) числовой вектор (vector);
SI — аргумент типа `SUBSET` или `INT`, возможны произвольные сочетания;
$R_1, \ldots, R_n$ — произвольное число аргументов типа `REAL`;
$[R]$ — аргумент необязательный;
$[R_1, R_2]$ — аргументы могут быть опущены только вместе.

4.1 Математические функции

$R =$ `abs(`R`)` — абсолютная величина числа R.
$I =$ `sign(`R`)` — сигнум числа R.
$R =$ `plus(`R`)` — положительная срезка: $R_+ = R$ при $R > 0$, $R_+ = 0$ при $R \leqslant 0$.
$R =$ `round(`R`)` — округление числа R до ближайшего целого.
$R =$ `uround(`R`)` — округление числа R до ближайшего большего целого.
$R =$ `dnround(`R`)` — округление числа R до ближайшего меньшего целого.
$R =$ `mod(`R_1, R_2`)` — остаток от деления R_1 на R_2.
$R =$ `sqrt(`R`)` — корень числа R.
$R =$ `sin(`R`)` — синус числа R.
$R =$ `cos(`R`)` — косинус числа R.
$R =$ `tan(`R`)` — тангенс числа R.

$R =$ `exp(`R`)` — экспонента числа R.

$R =$ `log(`$R[, R_0]$`)` — логарифм числа R по основанию R_0; натуральный логарифм, если второй аргумент опущен.

$R =$ `PNorm(`$R_1, [R_2, R_3]$`)` — функция нормального распределения с матожиданием $M = R_2$ и средним квадратическим отклонением $\sigma = R_3$. Если R_2 и R_3 опущены, то $M = 0$, $\sigma = 1$.

$R =$ `rand(`$[R_1, [R_2]]$`)` — случайное число в интервале $[R_1, R_2)$, либо $[0, R_1)$ если второй аргумент опущен, либо $[0, 1)$ если оба аргумента опущены.

$D =$ `if(`$L_1, D_1, \ldots, L_n, D_n, D_0$`)` — условное выражение. Если L_1 истинно, возвращает значение D_1, иначе проверяет значение L_2, и если оно истинно, возвращает D_2, и т.д. Если все $L_1, \ldots, L_n$ ложны, возвращает D_0.

$I =$ `steps(`$R, R_1, \ldots, R_n$`)` — ступенчатая функция, возвращающая

$$\begin{cases} 0 & \text{если } R < R_1, \\ 1 & \text{если } R_1 \leqslant R < R_2, \\ \ldots & \\ n & \text{если } R_n \leqslant R. \end{cases}$$

$L =$ `xor(`$L_1, \ldots, L_n$`)` — Исключающее ИЛИ двоичных чисел.

4.2 Битовые функции

$I =$ `Bit(`I_1, I_2`)` — значение I_2-го бита в целом числе I_1.

$I =$ `SetBit(`$I_1, [I_2], [L]$`)` — установка I_1-го бита в целом числе I_2 (по умолчанию 0) равным L (по умолчанию 1).

$I =$ `OrBit(`$I_1, \ldots, I_n$`)` — побитовое или

$I =$ `AndBit(`$I_1, \ldots, I_n$`)` — побитовое и

$I =$ `XorBit(`$I_1, \ldots, I_n$`)` — побитовое исключающее или

$I =$ `NotBit(`I`)` — побитовое отрицание

4.3 Функции над наборами чисел

Следующие функции могут принимать в качестве аргументов не только числа, но и любые числовые структуры данных: подмассивы; векторы параметров; векторы, состоящие из элементов типа `DATUM`; числовые векторы, формируемые функцией `Distrib`. При поиске числовых значений осуществляется рекурсивный просмотр всех структур данных. Нечисловые значения игнорируются.

$R =$ `min(`$x_1, \ldots, x_n$`)` — минимум.

$R =$ `max(`$x_1, \ldots, x_n$`)` — максимум.

$R =$ `sum(`$x_1, \ldots, x_n$`)` — сумма.

$R =$ `avr(`$x_1, \ldots, x_n$`)` — среднее значение.

$R =$ `dev(`$x_1, \ldots, x_n$`)` — среднее отклонение.

$R =$ `stdev`$(x_1, \ldots, x_n)$ — среднее квадратичное отклонение.

$R =$ `count`$(x_1, \ldots, x_n)$ — количество числовых значений.

$R =$ `countAll`$(x_1, \ldots, x_n)$ — количество всех значений, включая нечисловые.

$R =$ `countNonum`$(x_1, \ldots, x_n)$ — количество нечисловых значений.

$R =$ `countValue`$(R, x_1, \ldots, x_n)$ — количество значений, равных R.

$R =$ `countRange`$(R_1, R_2, x_1, \ldots, x_n)$ — количество значений в диапазоне $[R_1, R_2)$.

$V =$ `Distrib`$(x_1, \ldots, x_n)$ — собирает числовые значения в один вектор и производит его предварительную обработку (сортировку). Последующие функции требуют в качестве аргумента вектор, сформированный функцией `Distrib`.

$R =$ `NthMin`(I, V) — I-ое минимальное значение, при $I = 1$ совпадает с `min`.

$R =$ `NthMax`(I, V) — I-ое максимальное значение, при $I = 1$ совпадает с `max`.

$R =$ `CutOff`(R, V) — пороговое значение x такое, что Rn чисел не превосходят x, где $0 < R < 1$ и n — количество чисел в векторе V.

4.4 Функции для работы с датами и временем

$T =$ `now()` — текущие дата и время.

$I =$ `year`(T) — год, целое, начиная с 1900.

$I =$ `month`(T) — месяц, целое в диапазоне $[1, 12]$.

$I =$ `day`(T) — день, целое в диапазоне $[1, 31]$.

$I =$ `hours`(T) — часы, целое в диапазоне $[0, 23]$.

$I =$ `minutes`(T) — минуты, целое в диапазоне $[0, 59]$.

$I =$ `seconds`(T) — секунды, целое в диапазоне $[0, 59]$.

$I =$ `WeekDay`(T) — день недели, целое в диапазоне $[0, 6]$, 0 — воскресенье.

$I =$ `YearDay`(T) — день в году, целое в диапазоне $[1, 356]$.

4.5 Строковые функции

$I =$ `len`(C) — число символов в строке C.

$I =$ `pos`(C_1, C_2) — позиция строки C_2 в строке C_1; возвращает (-1), если вхождение не найдено.

$C =$ `left`(C, I) — первые I символов строки C.

$C =$ `right`(C, I) — последние I символов строки C.

$C =$ `mid`(C, I_1, I_2) — часть строки C, начиная с I_1-го символа из I_2 символов.

4.6 Функции над наборами, поднаборами и подмассивами

$I =$ `dim`(SAB) — размерность поднабора, подмассива или базового набора.

$I =$ `card`(SAB) — мощность поднабора, подмассива или базового набора. Функция может вычисляться неверно, если число элементов превышает $2 \cdot 10^9$.

$I =$ `len`(SAB) — число элементов в поднаборе S, синоним `card`.

I = `pos(`S_1, S_2`)` — позиция поднабора S_2 в поднаборе S_1; возвращает (-1), если вхождение не найдено.

S = `left(`S, I`)` — первые I элементов поднабора S.

S = `right(`S, I`)` — последние I элементов поднабора S.

S = `mid(`S, I_1, I_2`)` — часть поднабора S, начиная с I_1-го элемента из I_2 элементов.

C = `atom(`$B[, I]$`)` — имя I-го атома в базовом наборе B, либо текущего атома, если аргумент I опущен.

I = `AtomIn(`IB, S`)` — порядковый номер атома в поднаборе S, если его порядковый номер в базовом наборе равен I; функция возвращает (-1), если атом не принадлежит поднабору.

I = `AtomID(`I, S`)` — порядковый номер атома в базовом наборе, если его порядковый номер в поднаборе S равен I; если не выполнено условие $0 \leqslant I <$ `card(S)`, функция возвращает (-1).

S = `DiffLines(`A`)` — множество попарно различных строк двумерного подмассива A. Если подмассив имеет вид $[\![X \times Y]\!]$, то возвращается поднабор $X' \subseteq X$.

S = `NoZeroLines(`$A[, R[, I]]$`)` — множество ненулевых строк двумерного подмассива A. Нулевыми считаются элементы, по модулю не превышающие R. Нулевыми считаются строки, содержащие не более I ненулевых элементов. По умолчанию $R = 0$ и $I = 0$.

S = `DiffValues(`A`)` — множество попарно различных значений двумерного подмассива A. Является составным набором той же размерности, что и массив.

S = `ProjDim(`S, I`)` — Проекция составного набора S на I-ую размерность.

4.7 Функции для работы с потоками вывода

O = `FileOutput(`$C[, I]$`)` — поток вывода в файл с именем C и флагами вывода I. Если файл ещё не открыт, открывает его.

O = `ConOutput(`$[I]$`)` — стандартный поток вывода с флагами вывода I.

O = `WinOutput(`$C[, I]$`)` — поток вывода в окно с именем C и флагами вывода I. Для вывода в окно отладчика требуется задать C = `"Output"`.

O = `StrOutput(`$C[, I]$`)` — поток вывода в строку с флагами вывода I. Аргумент C является именем потока. Для выдачи строки необходимо использовать функцию преобразования типа `Str`.

O = `SyncOutput(`$O_1, \ldots, O_n[, I]$`)` — поток синхронного вывода в потоки $O_1, \ldots, O_n$. Устанавливает флаги вывода I.

4.8 Функции проверки значений

L = `IsEmpty(`D`)` — является ли значение пустым или ошибочным.

I = `ErrCode(`D`)` — если значение является пустым или ошибочным, возвращает код ошибки, иначе возвращает 0.

$L =$ `IsNumber(`D`)` — является ли значение числовым.
$L =$ `IsTime(`D`)` — является ли значение датой или временем.

5 Запуск и отладка

Интерпретаторы языка ASDIEL являются кросс-платформенными приложениями, одинаково работающими в среде `MSDOS`, `Windows` и `UNIX`.

5.1 Диалоговый интерпретатор

Приложение `SDLcmd` предоставляет простейший диалоговый интерпретатор, который выполняет команды ASDIEL и вычисляет выражения, вводимые пользователем из командной строки. Если строка заканчивается символом `;`, то она воспринимается как команда (или несколько команд, разделённых `;`) и немедленно выполняется. В противном случае она рассматривается как выражение, вычисляется и выводится на экран. Таким образом `SDLcmd` можно использовать в качестве простого консольного калькулятора.

Диалоговый интерпретатор позволяет запустить ASDIEL-программу двумя способами. Во-первых, после приглашения `SDL>` можно набрать команду

```
RUN MyProg.sdl;
```

где `MyProg.sdl` — имя файла с программой. Во-вторых, `SDLcmd` можно вызвать, передав имя файла в качестве параметра. В таком случае интерпретатор выполняет программу, не переходя в диалоговый режим.

Несмотря на то, что диалоговый режим позволяет описывать методы и алгоритмы, построение и исследование полноценной алгоритмической суперпозиции таким способом вряд ли удобно.

5.2 Консольный отладчик

Приложение `SDLdbg` требует в качестве параметра имя программы. Это простейший консольный отладчик, позволяющий выполнять программу пошагово, просматривать сообщения об ошибках, выводить состояние ядра ASDIEL и время выполнения отдельных команд. При возникновении ошибки отладчик останавливается, подсвечивая неверную команду.

Экран отладчика поделён на 4 области: список разделов программы (слева), текст текущего раздела программы (в центре), окно сообщений об ошибках (внизу), список всех клавиш-команд отладчика (справа).

Отладчик может работать только с одной программой. Для выполнения другой программы необходимо запустить ещё одну копию отладчика.

Отладчик позволяет просматривать программу по разделам, но собственных средств редактирования не имеет. Программу можно редактировать любым внешним редактором, в том числе во время работы отладчика и даже во время её выполнения. Функция перезагрузки (клавиша `F11`) повторно считывает программу, сличает новый текст со старым и находит место первой модификации. Если перезагрузка произошла во время выполнения программы, и модифицированный участок уже пройден, выполнение будет возвращено к началу того раздела, в котором была обнаружена правка (см. команду `SECTION`). Все переменные, атомы, массивы и методы, созданные предыдущими разделами, будут сохранены, а созданные последующими разделами — удалены[1]. Это позволяет избежать повторного прогона начальной части программы, не затронутой исправлениями.

Отладчик может сохранить настроенную программу в файл `tuned.sdl`. Программа считается настроенной, если она полностью пройдена при включённом режиме `tune`. Текст программы `tuned.sdl` формируется из исходного текста следующим образом.

- Перед каждым блоком команд `MODE tune...END` записывается блок

  ```
  MODE parameters;
      Установка параметров;
  END MODE;
  ```

  где установка параметров — это последовательность команд присваивания, явным образом инициализирующих внутренние параметры метода настроенными значениями.
- Если такой блок уже имеется (данная программа когда-то была сгенерирована функцией сохранения), он будет заменён новым независимо от содержащихся в нём команд.
- В командах `MODES` добавляется режим `parameters` и выключается (знаком «минус») режим `tune`.

Таким образом, программа `tuned.sdl` описывает ту же самую алгоритмическую суперпозицию, но теперь — с жёстко фиксированным набором параметров. Это «рабочая» программа, которую можно использовать для расчёта без предварительной настройки.

Клавиши-команды консольного отладчика

[1] В текущей реализации интерпретатора в действительности удаляются только методы и признаки, сгенерированные в данном разделе или позже. Переменные, атомы и массивы не затрагиваются перезапуском и сохраняют свои прежние значения. В дальнейшем эта схема будет заменена полным восстановлением текущего состояния ядра.

F2	Un/Zoom	Расширение текущего окна
F3	Kernel	Вывод состояния ядра ASDIEL
F5	GoProgrm	Выполнение программы
F6	GoMethod	Выполнение раздела
F7	GoComand	Выполнение команды
F8	StopGo	Приостановить выполнение
F11	Reload	Перезагрузка программы
F12	Save	Сохранение настроенной программы
Tab	Switch	Переключение между окнами
Пробел	Output	Расширение окна сообщений
Alt+X	Exit	Выход из отладчика