2. Функциональный язык параллельного программирования «Пифагор»

Рассмотрено подмножество языка, реализованное в экспериментальной версии транслятора.

2.1. Используемый метаязык

Для описания синтаксиса используются расширенные формы Бэкуса-Наура (РБНФ). Квадратные скобки **"["** и **"]"** означают, что заключенная в них сентенциальная форма может отсутствовать, фигурные скобки **"{"** и **"}"** означают ее повторение (возможно, 0 раз), а круглые скобки **"("** и **")"** используются для ограничения альтернативных конструкций. Сочетание фигурных скобок и косой черты **"{/"** и **"/}"** используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл, написанными на русском языке и разделенными, при необходимости, знаком подчеркивания **"_"**. Каждое правило оканчивается точкой **"."**. Терминальные символы изображаются словами, написанными строчными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак **"$"** в начале строки. Левая часть правила отделяется от правой знаком **":="**.

2.2. Элементарные конструкции

2.2.1. Разделители

Пробелы, символы табуляции, перевода на новую строку и перевода страницы используются как разделители. Вместо одного из таких символов может использоваться любое их количество. Все другие управляющие символы употреблять в тексте программы запрещено.

2.2.2. Комментарии

Многострочные комментарии начинаются парой символов **"/*"** и заканчиваются парой символов **"*/"**. Разрешены везде, где возможны разделители. Вложенность многострочных комментариев не допускается.

Примеры:

```
/* Многострочный комментарий в одной строке */

/*
* Многострочный комментарий,
* размещенный в нескольких строках
*/
```

В языке также допускаются однострочные комментарии. Они начинаются парой символов **"//"** и заканчиваются признаком конца строки. Однострочные комментарии могут начинаться с самого начала строки или стоять после операторов, написанных в этой строке.

Пример:

```
// Однострочный комментарий
```

$ комментарий := "/*" {знак} "*/" | "//" {знак}.

2.2.3. Идентификаторы

Идентификаторы используются для обозначения имен констант, переменных, функций и типов данных. Допустимые символы: цифры **0-9**, прописные и строчные буквы латинского алфавита **A-Z**, **a-z**, символ подчеркивания **"_"**. Первый символ не является цифрой. Идентификатор может быть произвольной длины. Прописные и строчные буквы различаются.

$ ид := (буква | "_"){буква|цифра|"_"}.

Примеры:

```
NAME1   name1   it_is_ID
```

2.2.4. Зарезервированные слова

Зарезервированные слова используются для ключевых слов встроенных типов данных, предопределенных обозначений и функций. Ниже приведен общий их список:

block	**break**	**bool**	**char**	**const**
dup	**datalist**	**delaylist**	**else**	**error**
false	**float**	**func**	**funcdef**	**int**
nil	**parlist**	**prefunc**	**return**	**signal**
true	**type**	**typedef**		

Зарезервированные слова записываются строчными латинскими буквами. Использовать их в качестве идентификаторов запрещено.

Примечание. Следует отметить, что в текущей версии языка отсутствует деление зарезервированных слов по группам, что обычно связывается с природой их создания и использования. Это объясняется простотой языка и нацеленностью текущей реализации на сценарный вариант. Предполагается, что в последующих версиях произойдет более четкое дробление, явно увязанное с особенностями реализации и использования.

2.3. Обозначения

В языке, построенном на основе принципа единственного присваивания, отсутствует понятие переменной. Вместо него вводится понятие обозначения как идентификатора, поставленного в соответствие с каким-либо программным фрагментом. В пределах некоторой области видимости использование идентификатора в качестве обозначения должно быть уникальным. Обозначение получает тип и величину сопоставленного элемента и может использоваться для дальнейшей передачи этих параметров в любую точку программы, обеспечивая тем самым копирование объекта, полученного в ходе вычислений. В языке определены два способа задания обозначений:

- префиксное, при котором знак идентификатор пишется слева от знака "<<", а определяемый объект справа;
- постфиксное, когда слева от знака ">>" задается определяемый объект, а справа его идентификатор.

$ обозначение := идентификатор "<<" элемент |
элемент ">>" идентификатор.

Под элементом понимается любой из объектов языка, выражение, блок или ранее введенное обозначение. Идентификатор задает имя ранее обозначенного элемента. Понятия объекта, выражения и блока будут даны ниже.

$ элемент := объект | выражение | блок |
обозначение | идентификатор.

Примеры:

```
X << 100;  Pi << 3.1415; 10 >> ten;
(a, b):+ >> sum;  x0 << y0 << 0;
```

2.4. Объекты

К объектам языка относятся конструкции, рассматриваемые при выполнении

операций интерпретации как единое целое. Каждый объект характеризуется двойкой:

<тип, значение>.

Объекты могут формироваться как до вычислений, так и в ходе их. Объект, сформированный до вычислений, является константой.

$ объект := атом | список | функция.

Примечание. В дальнейшем предполагается реализация в языке частично сформированных объектов, для которых определен только тип, а окончательное значение еще не вычислено.

Существует неупорядоченное множество типов предопределенных объектов, задаваемых соответствующими именами. Предопределенные объекты делятся на атомарные и составные. Типы атомарных объектов (атомов) и области их допустимых значений определяются аксиоматически. Составные объекты являются комбинацией атомарных и уже существующих составных объектов. Они конструируются по заданным правилам. К составным объектам относятся описания функций и списки. Обозначения предопределенных типов языка, используемые в данной версии, приведены в таблице 2.1.

Таблица 2.1.

Предопределенные типы.

Название	Обозначение типа	Организация
сигнал	signal *(nil)*	атом
целый	int	атом
действительный	float	атом
символьный	char	атом
логический	bool	атом
спецзнаковый	spec *(nil)*	атом
ошибочный	error *(nil)*	атом
список данных	datalist	составной
параллельный список	parlist	составной
задержанный список	delaylist	составной
предопределенная именованная функция	func	атом
функция	func *(nil)*	определяемый

Примечание. В настоящее время реализован небольшой набор типов, позволяющий использовать язык для запланированных экспериментов. Расширение номенклатуры базовых типов планируется в дальнейших реализациях языка после отработки методов параллельной интерпретации и методов преобразования в параллельные программы для других архитектур.

2.5. Сигналы

Сигнальные объекты (или просто **сигналы**) отличаются от других атомарных объектов тем, что у них отсутствует значение. Вместо этого готовность к интерпретации определяется самим фактом появления атома. Учитывая то, что обязательной составляющей всех объектов является тип, появление в качестве результата сигнального типа определяет сам факт срабатывания соответствующего оператора интерпретации. Использование сигналов позволяет, при необходимости, моделировать в функциональных программах

явное управление вычислениями. Они также могут сигнализировать о завершении работы функции, не возвращающей параметры.

Понятие сигнала является синонимом пустого значения или «пусто». Это значение в языке обозначено константой `".”`. Таким образом любая функция, не имеющая список параметров все равно может быть запущена только при наличии сигнала в качестве аргумента операции интерпретации. Постоянно присутствие сигнала, определяющее «моментальный» запуск задается следующим выражением:

```
.:F
```

Формат сигнала, определяющий его внутреннее строение:

```
<signal, .>.
```

2.6. Значащие величины (константы)

Атомы данного вида обеспечивают задание различных величин. Величина принадлежит области ее допустимых значений, которая, в зависимости от типа, может задаваться одним из следующих способов: диапазоном, диапазоном и точностью, перечислением элементов упорядоченного множества, перечислением элементов неупорядоченного множества (если нет необходимости устанавливать между элементами отношение порядка), функцией. В настоящее время в языке реализованы следующие виды констант: целые, действительные, булевские, символы, константы ошибок, специальные знаки. Тип константы в программе определяется ее внешним видом, задаваемым синтаксическими правилами.

$ константа := целая | действительная | символ |
логическая | строка | спецзнак .

Целые константы

Целые константы используются для представления данных в формате стандартного машинного слова, длина которого зависит от архитектуры ВС *(в рассматриваемой реализации используется 32-х разрядное представление)*. В текущей версии языка реализовано представление целых чисел только в десятичной системе счисления.

$ целая := ["-"] {/цифра/}.
$ цифра := "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

Примеры:

```
127   0127  -356
```

Внутренний формат целочисленной величины:

```
<int, Один из множества: {MinInt, …, MaxInt}>.
```

Действительное число

Действительное число задается диапазоном (от минимального отрицательного "MinFloat" до максимального положительного "MaxFloat" с указанием точности перечисления "PrecFloat"). Однако эти параметры можно по умолчанию охарактеризовать и форматом машинного слова, используемого в каждом конкретном случае. В данном случае действительные числа реализованы с использованием 64-разрядного представления.

$ действительная := ["-"] {/цифра/} порядок |
["-"] {/цифра/} "." {/цифра/} [порядок].
$ порядок := ("e"|"E") ["+"|"-"] {/цифра/}.

Примеры:

```
1.0e5   -5E-3   0.5   5.0e2   -2.0   3.14
```

Десятичная точка, отделяющая целую часть от дробной должна обязательно стоять между цифрами.

Внутренний формат действительной величины:

<float, Один из множества: {MinFloat, …, MaxFloat}>.

Символьные константы

Символьные константы состоят из одного видимого знака или управляющего символа используемой кодовой таблицы, например ASCII, ограниченного одинарными кавычками.

$ символ := " ‘ " (видимый_знак | управляющий_символ) " ‘ ".

Управляющие символы задаются с префиксом в виде обратной косой черты. С этой же чертой записываются видимые символы: пробел, обратная косая черта, апостроф и кавычка. Кавычка может записываться в символьных константах и без обратной косой черты. Пробел допускается представлять значением самого символа. Ниже представлены символы, записываемые специальным образом:

- перевод строки (LF) – **‘\n’**;
- горизонтальная табуляция (HT) – **‘\t’**;
- вертикальная табуляция (VT) – **‘\v’**;
- возврат на шаг (BS) – **‘\b’**;
- возврат каретки (CR) – **‘\r’**;
- перевод формата (FF) – **‘\f’**;
- обратная косая (\) - **‘\\’**;
- нулевой символ (nul) – **‘\0’**;
- пробел ( ) – **‘\s’** или **‘ ’**;
- апостроф (‘) – **‘\’’**;
- кавычка (“) – **‘\"’** или **‘"’**.

Примеры:

‘A’ ‘b’ ‘1’ ‘\\’ ‘\n’ ‘ ‘ ≡ ‘\s’

Внутренний формат символьной величины:

(char, Один из множества допустимых символов).

Логическая константа

Логическая константа имеет значения истина ("**true**") и ложь ("**false**"). Она задается соответствующими ключевыми словами.

$ логическая := true | false.

Для логических констант сохраняется отношение порядка:

false < true.

Внутренний формат символьной величины:

<bool, {true | false}>.

Специальные знаки

Специальные знаки используются для задания предопределенных данных и операций языка в зависимости от их местоположения относительно операции интерпретации. Их смысл будет раскрыт при описании правил интерпретации. Эти константы образуют неупорядоченное множество и используются в тексте программы как разделители других конструкций.

$ перечислимая := "+"|"-"|"/"|"*"|"<"|">"|"="|">="|"<="|"<>"|"=>"|
"->"|"<-"|"()"|"{}"|"[]"|"|"|"#"| "%" |".."|"?".

Следует отметить, что ряд специальных знаков в настоящее время не используется и

зарезервирован для дальнейшего расширения языка.

Внутренний формат специальных знаков:

<spec, Одна из перечислимых величин>.

Константы ошибок

Константы ошибок используются для отображения некорректных ситуаций, возникающих в ходе вычислений. Величины этого типа могут обрабатываться наряду с другими данными или как исключительные ситуации.

Область допустимых значений для констант ошибки задается неупорядоченным множеством, которое в дальнейшем предполагается пополнять. В настоящий момент выделяются следующие ошибки:

- **ERROR** - неидентифицируемая ошибка;
- **REALERROR** - некорректное преобразование действительного числа;
- **INTERROR** - некорректное преобразование целого числа;
- **ZERODIVIDE** - деление на ноль;
- **INTERPRERROR** - ошибка операции интерпретации;
- **BOUNDERROR** - ошибка выхода за границы диапазона;
- **BASEFUNCERROR** – неправильное использование предопределенной функции.

Эти имена запрещается использовать в программе в другом контексте.

$ константа_ошибки := ERROR | REALERROR | INTERROR |
ZERODIVIDE | INTERPRERROR | BOUNDERROR | BASEFUNCERROR.

Внутренний формат для величин, задающих ошибки:

(error, Одна из констант, задающих ошибку).

*Примечание. В настоящий момент в языке фигурируют **неопределенные типы**, используемые для объединения значений, распознавание которых в виде отдельных типов пока четко не просматривается. Хотя, эти величины могут принадлежать к какой-либо группе. Для определения принадлежности величины к конкретной группе в этом случае необходимо явно проверить ее значение. Внутренний формат для величин неопределенного типа:*

(nil, Одна из величин, соотнесенная с этим типом).

Пока с неопределенным типом сопоставлены специальные знаки, функции. Однако в дальнейшем предполагается внимательно пересмотреть их состав. Функции, скорее всего, будут выделены в свой тип (описан ниже). Возможно, что надобность в таком типе может отпасть по мере уточнения семантики языка.

2.7. Составные объекты

К составным объектам относятся списки данных, параллельные списки, задержанные списки и строки (являющиеся подмножеством списка данных). Каждый из таких списков формируется путем охвата одного или нескольких элементов соответствующей операцией включения в список. Семантика списка данных, параллельного и задержанного списков рассмотрена в разделе 1.

$ составной := список_данных | параллельный_список |
задержанный_список | строка.
$ список_данных := "(" элемент {"," элемент} ")".
$ паралельный_список := "["элемент{","элемент} "]".
$ задержанный_список := "{"элемент{","элемент} "}".

Объекты, включаемые в список, могут быть произвольной структуры, что позволяет

создавать достаточно сложные иерархические конструкции. Кроме того, список может быть пустым. В этом случае внутри него ставится только **"."**. Семантика и особенности использования пустых списков были рассмотрены при описании модели вычислений.

Строка - это совокупность видимых и управляющих символов, заключенная в кавычки (**"**). Для представления кавычки внутри строки используется ее комбинация с обратной косой чертой (**"**). Апостроф может быть задан как один символ (‘) или в комбинации с обратной косой чертой (\’). Длинная строка может быть представлена последовательностью более коротких строк, разделителем между которыми могут являться только пробельные символы. Пробелы внутри строк могут задаваться с использованием как управляющего символа, так и своим обычным знаком.

$ строка := { /""" { символ_но_не_кавычка | "\"" | управляющий_символ }""" /}.

Строка является частным случаем списка данных и определяет список символьных атомов, задающих некоторый текст. Написание текста в виде строки приводит к более компактному и понятному представлению. Пробелы внутри строк могут задаваться как двойным управляющим символом, так и своим обычным знаком.

Примеры строк:

```
"" ≡ (.)   - пустая строка
"Строка"  =  ('С', 'т', 'р', 'о', 'к', 'а')
" Крикнул: \"Ура!\"\s" = ('\s','К','р','и','к','н','у','л',
                         ':',' ','"','У','р','а','!','\"','\s')
"Это одна" "длинная, но" "разделенная строка!"
```

2.8. Функция

Организация обычной функции

Функция – составной объект, конструируемый специальным образом. Она задается определением, начинающимся с ключевого слова **funcdef**. Состоит из заголовка и тела. В заголовке указывается идентификатор аргумента, обеспечивающего передачу в тело функции необходимых данных. В теле описывается алгоритм обработки аргумента. Доступ к исходным данным осуществляется только через аргумент, тип которого и значение в данной версии языка могут быть произвольными. Тело функции состоит из элементов, заключенных в фигурные скобки и разделяемых между собой символом **";"**.

В ходе выполнения функции обычно формируется результат, который возвращается после обозначения его зарезервированным идентификатором "**return**":

`результат >> return` или **`return << результат`**

Возвращаемый результат может быть любым допустимым значением, полученным в ходе вычислений. Возврат результата может осуществляться до завершения выполнения всех операций в теле функции, которая продолжает существования до завершения всех внутренних операций. Однако повторного возврата, в соответствии с принципом единственного присваивания, произойти не может. В случае параллельного списка возможен асинхронный (не одновременный) возврат его независимых элементов.

Функция может и не возвращать явно результат, а также вообще не иметь элементов в теле. В этом случае в качестве результата возвращается объект сигнального типа, посылаемый в точку возврата по завершении всех вычислений. Возврат осуществляется после того, как завершено выполнение всех операций в теле функции.

*Примечание. Текущая реализация требует явного возврата с использованием **return**. Хотя трансляция проходит без ошибок. Поэтому, для имитации возврата пустого значения можно задать следующее тело функции:*

`{. >> return}`

В дальнейшем этот недостаток предполагается исправить.

$ функция := "funcdef" [аргумент] "{" [элемент {";" элемент }] "}".

Наряду с определением функции допускается ее **предварительное объявление**. Оно полезно при использовании рекурсивных методов, когда одна из функций может вызвать другую, еще не определенную.

Предварительное объявление задается следующим образом:

$ предобъявление := prefunc.

Оно используется в паре с обозначением, позволяя тем самым задавать имена функций. В дальнейшем данное имя может встретиться в обозначении еще раз при определении функции.

Пример:

```
f1 << prefunc  или  prefunc >> f2
```

Перегрузка функций с одинаковой сигнатурой

В языке реализован механизм перегрузки функций, позволяющий гибко и безболезненно расширять уже разработанную программу. Так как в языке отсутствует строгая типизация, все функции с одинаковыми именами становятся неразличимы (имеют одинаковую сигнатуру). Поэтому, вместо выбора одной из перегруженных функций осуществляется их одновременное выполнение. Результат возвращается в виде параллельного списка. В отличие от обычных функций перегруженные функции задаются с помощью простого синтаксического приема: к обозначению добавляются квадратные скобки, в которых может содержаться любое действительное число, задающее ранг. Ранг используется для упорядочения функций в параллельном списке по возрастанию. При отсутствии числа ранг считается равным нулю. Функции с одинаковым рангом могут располагаться в списке в произвольном порядке. Обычно они размещаются в порядке их обработки транслятором. Пример использования рангов:

```
OverFunc[2.5] << funcdef Param { // Тело функции }
OverFunc[]    << funcdef Param { // Тело функции }
OverFunc[-10] << funcdef Param { // Тело функции }
```

Вызов параллельной функции синтаксически ничем не отличается от обычного вызова:

```
X:OverFunc;
```

Перегрузка функций позволяет гибко добавлять новые возможности, обуславливаемые появлением новых данных.Для избавления от пустых значений используется список данных, обладающий свойствами фильтрации.

Примечание. Предопределенная функция должна возвращать параллельный список, который, в окружении пустых значений, должен «ужиматься». На самом деле самосинхронизация списка данных не происходит. Например:

```
OverFunc[2.5] << funcdef x{ (x,2.5):* >>return }
OverFunc[]    << funcdef x{ (x,0):+ >>return }
OverFunc[-10] << funcdef x{ (x,-10):- >>return }

test3 << funcdef {
  (., 3:OverFunc, .) >>return
  // (., 3:OverFunc, .):. >>return // и здесь то же!
  // [., 3:OverFunc, .]:(.) >>return // пока приходится так!
}
=> (.,13,3,7.500000e+000,.)
```

Должно быть: `(13,3,7.500000e+000)`

Необходимо исправить.

Предопределенные именованные функции

Предопределенные именованные функции задаются их именами и используются наравне со спецсимволами. Специфика проявляется только в том, что их имена зарезервированы.

2.9. Блок

Блок - это объединение элементов внутри тела функции, служит для логического соединения группы операторов выполняющих законченное действие, а также для локализации обозначений. Он начинается с ключевого слова **block**, за которым следует тело блока, аналогичное телу функции. Отличие тела блока заключается в том, что выход из него осуществляется по обозначению результата зарезервированным идентификатором **break**, с которым связывается значение, возвращаемое из блока:

`результат >> break` или **`break << результат`**

При отсутствии **break** блок возвращает значение сигнального типа по завершении выполнения всех его операторов. Поведение блока в этом случае полностью совпадает с поведением тела функции.

$ блок := "block" "{" [элемент {";" элемент }] "}".

2.10. Выражение

Выражение - это терм или цепочка термов, связанных между собой операциями интерпретации и их альтернативными частями. Под термом понимается объект, блок или имя ранее обозначенного элемента. Наличие операции интерпретации позволяет трактовать два ее операнда как функцию и аргумент. Существуют префиксная и постфиксная формы записи операции интерпретации, отличающиеся друг от друга только порядком следования аргумента и функции. Префиксная операция интерпретации задается стрелкой вверх **“^”**, слева от которой стоит терм, выступающий в роли функции, а справа - аргумент: **`F^X`**. При постфиксной записи это же выражение будет выглядеть следующим образом: **`X:F`**.

Наряду с непосредственной обработкой данных, операция интерпретации имеет также необязательную альтернативную часть, выполнение которой осуществляется при возникновении ошибки в основной ветви вычислений. Альтернативная ветвь отделяется ключевым словом **else**. Она не запускается, если выполнение основной части прошло без ошибок. Данная конструкция позволяет оперативно осуществлять нестандартную обработку исключений или создавать ветвления путем имитации ошибки. В альтернативную часть передается аргумент следующего формата:

`(Возникшая ошибка, Исходный аргумент)`

Это позволяет достаточно гибко восстановить причину ошибки и продолжить вычисления. Альтернативная часть (**else**-часть) содержит выражение, определяющее функцию, выделяющую из аргумента составные части и, на основе их обработки, обеспечивающую коррекцию дальнейших вычислений. Если исходный аргумент является сигналом, то операнд, поступающий в альтернативную ветвь, вырождается до одноэлементного списка:

`(Возникшая ошибка, .)≡ (Возникшая ошибка)`

$ выражение := терм {("^" выражение | ":" терм) [else терм]}.
$ терм := объект | блок | идентификатор.

Приведенный синтаксис выражения показывает, что альтернативная часть принадлежит ближайшей слева операции интерпретации. Кроме этого следует отметить, что префиксная операция интерпретации выполняется справа налево, а постфиксная и обработка альтернатив слева направо. Изменение приоритетов можно осуществить использованием

квадратных или круглых скобок, являющихся функциями группировки в список, и, следовательно, формирователями новых промежуточных объектов.

2.11. Структура программы

Программа состоит из последовательности описаний. Описание является обозначением константного выражения, определением или предварительным объявлением функции. Описания разделяются точкой с запятой.

$ программа := обозначенное_описание { ";" обозначенное_описание }.
$ обозначенное_описание := {/ идентификатор "<<" /}
описание {">>"идентификатор}
| [описание">>"] идентификатор {/ ">>" идентификатор /}.
$ описание := функция | prefunc | const константное_выражение.

Константное выражение - это любой объект языка, вычисляемый на этапе компиляции, и используемый в последующих выражениях как атомарная константа, список данных или параллельный список, атомами которого на самом нижнем уровне вложенности являются константы.

Пример:

```
pi << const 3.14
```

2.12. Предопределенные функции и данные

Предопределенные функции и данные формируются на основе атомов, каждый из которых может быть в роли аргумента или функции операции интерпретации. При этом ряд атомарных объектов могут выступать только в роли данных, другие - в роли функций, третьи - в той и другой.

Большинство специальных знаков используются как предопределенные функции. Обычно за ними закрепляются вычислительные операции, традиционные для этих значков в большинстве существующих языков программирования. Задаются допустимые аргументы и значения этих функций. При этом тип операции не связан только со знаком. Он также зависит от типа аргумента. Поэтому нельзя, например, говорить о знаке "+" как об арифметической операции, так как при булевских аргументах он используется для обозначения дизъюнкции.

Примечание. Это утверждение является спорным, так как использование одних и тех же знаков с различной семантической окраской затрудняет понимание исходного текста программы. Возможно, более приемлемым, все-таки, является введение дополнительных обозначений для широко известных операций, например, для булевских данных. Поэтому в дальнейшем еще возможен пересмотр использования предопределенных знаков. Решение о сильной перегрузке спецсимволов принималось на первоначальных этапах и казалось привлекательным из-за модного тогда использования перегрузок операций. В дальнейшем акцент на механизм управления вычислениями отодвинул пересмотр принятого решения.

В рассматриваемой версии не все спецзнаки имеют определенную семантику. Предполагается, что в дальнейшем она будет разрабатываться и уточняться. Наряду со специальными знаками в качестве предопределенных функций могут выступать и идентификаторы, которые, как и ключевые слова, запрещены для других применений.

Ниже приводится описание семантики предопределенных функций, используемых в текущей версии. Следует отметить, что для аргументов, тип которых при описании не задан результатом будет ошибка: «неправильное использование предопределенных функций».

Использование специальных знаков

Использование знака “.”

При интерпретации в качестве функции данный знак обозначает сигнал и может интерпретироваться как пустая операция. Если аргумент является списком данных, параллельным списком или атомом, то происходит его выдача в качестве результата. Если же аргумент – задержанный список, то происходит его раскрытие с последующим вычислением и передачей вычисленных значений в качестве результата.

Пример:

```
{(2,3):+}:. ⇒ 5
```

Использование этой операции позволяет получать из параллельного списка последовательный список данных (в соответствии с правилами эквивалентных преобразований):

```
[2,3]:(.) ⇒ (2,3)
```

Использование знака **“.”** *в качестве данных* интерпретируется как отсутствие аргумента. Он может использоваться для формирования операции интерпретации тех функций, которые не обрабатывают входных параметров. Например:

```
.:sin_pi_div_4
```

Использование знака “+”

Интерпретация знака **“+”** *в качестве функции* зависит от типа аргумента. Если аргумент является двухэлементным списком числовых атомов (целых или действительных), то выполняется **арифметическое сложение**. При сложении двух целых чисел результат всегда является целым числом. В этом случае автоматическое преобразование к действительному числу не происходит даже при переполнении, а выдается соответствующая ошибка. Во всех остальных случаях осуществляется сложение действительных чисел с преобразованием, в случае необходимости, целочисленного операнда к действительному. Результатом в этом случае является действительное число.

Если аргумент является числовым атомом, то он выдается в качестве результата без каких-либо преобразований.

Аргумент функции **“+”** может также быть булевским списком длиной, большей или равной 2. Результатом интерпретации в этом случае является дизъюнкция (логическое «или») всех элементов списка. Допускается одноэлементный булевский список или булевский атом, порождающий в качестве результата значение этого элемента.

Во всех остальных случаях результатом является ошибка операции интерпретации **BASEFUNCERROR**.

Знак **“+”** *в качестве аргумента* имеет тип **spec**.

Примеры сложения:

```
(3,5):+ ⇒ 8
(3,5.0):+ ⇒ 8.0
(5):+ ⇒ (BASEFUNCERROR, (5))
5:+ ⇒ 5
(max_integer,1):+ ⇒ (INTERROR, (max_integer,1))
(true, false, true):+ ⇒ true
true:+ ⇒ true
false:+ ⇒ false
(true):+ ⇒ true
```

Примечание. Возможно, имеет смысл использовать для булевских операций другой символ или идентификатор. Например, ***or*** *или «!».*

Использование знака “-”

*Интерпретация знака **“-”** в качестве функции* зависит от типа аргумента. Если аргумент является двухэлементным списком числовых атомов (целых или действительных), то выполняется **арифметическое вычитание**. При вычитании двух целых чисел результат всегда является целым числом. В этом случае автоматическое преобразование к действительному числу не происходит даже при переполнении, а выдается соответствующая ошибка. Во всех остальных случаях осуществляется вычитание действительных чисел с преобразованием, в случае необходимости, целочисленного операнда к действительному. Результатом в этом случае является действительное число.

Если аргумент является числовым атомом, то выполняется операция «унарный минус», изменяющая знак числа.

Аргумент функции **“-”** может также быть непустым булевским списком длиной большей или равной 2. Результатом интерпретации в этом случае является «исключающее или» (сложение по модулю два) всех элементов списка. Допускается одноэлементный булевский список, возвращающий в качестве результата отрицание элемента. Если аргумент – булевский атом, то результат данной операции равен отрицанию значения аргумента.

Во всех остальных случаях результатом является ошибка операции интерпретации **BASEFUNCERROR**.

*Знак **“-”** в качестве аргумента* имеет тип **spec**.

```
(3,5):- ⇒ -2
(3,5.0):- ⇒ -2.0
(5):- ⇒ (BASEFUNCERROR, (5))
5:- ⇒ -5
(max_integer,-1):- ⇒ (INTERROR, (max_integer,-1))
(true,false,true):- ⇒ false
true:- ⇒ false
false:- ⇒ true
(true):- ⇒ false
```

*Примечание. Возможно, имеет смысл использовать для булевских операций другой символ или идентификатор. Например, **xor** или «~».*

Использование знака “*”

Аргумент функции “”* может быть двухэлементным списком числовых атомов (целых или действительных), а знак **“*”** интерпретируется как **арифметическое умножение**. При умножении двух целых чисел результат всегда является целым числом. В этом случае автоматическое преобразование к действительному числу не происходит даже при переполнении, а выдается соответствующая ошибка. Во всех остальных случаях осуществляется умножение действительных чисел с преобразованием, в случае необходимости, целочисленного операнда к действительному. Результатом в этом случае является действительное число.

Аргумент функции **“*”** может также быть булевским списком длиной, большей или равной 2. Результатом интерпретации в этом случае является конъюнкция (логическое «и») всех элементов списка. Допускается одноэлементный список, возвращающий в качестве результата значение этого элемента. Если аргумент – булевский атом, то результат данной операции равен значению аргумента.

Во всех остальных случаях результатом является ошибка операции интерпретации **BASEFUNCERROR**.

*Знак **“*”** в качестве аргумента* имеет спецзнаковый тип.

Примеры умножения:

```
(3,5):* ⇒ 15
```

```
(3,5.0):* ⇒ 15.0
(5):* ⇒ (BASEFUNCERROR, (5))
5:* ⇒ (BASEFUNCERROR, 5)
(max_integer, 2):* ⇒ INTERROR
(true, false, true):* ⇒ false
(true, true, true):* ⇒ true
true:* ⇒ true
(true):* ⇒ true
```

Примечание. Возможно, имеет смысл использовать для булевских операций другой символ или идентификатор. Например, ***and*** *или «****&****».*

Использование знака “/”

Аргумент функции должен быть двухэлементным списком числовых атомов (целых или действительных), а знак **“/”** интерпретируется как **арифметическое деление**. При этом результат всегда является действительным числом. При делении на ноль выдается ошибка **ZERODIVIDE**.

Во всех остальных случаях результатом является ошибка операции интерпретации **(BASEFUNCERROR, (5))**.

Знак **“/”** в качестве аргумента имеет тип **spec**.

Примеры деления:

```
(3,5):/ ⇒ 0.66667
(3,5.0):/ ⇒ 0.66667
(5):/ ⇒ (BASEFUNCERROR, (5))
5:/ ⇒ (BASEFUNCERROR, 5)
```

Использование знака “%”

Аргумент должен быть двухэлементным списком целочисленных атомов, а знак **“%”** интерпретируется как **целочисленное деление с формированием частного и остатка**. Результатом данной операции явится двухэлементный список целых констант, первая из которых является целым частным элементов аргумента, а вторая – остатком от деления. Используется «компьютерная» интерпретация результатов, при которой частное округляется к нулю, а знак остатка равен знаку делимого. Операции связаны между собой следующим выражением:

```
x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y, если x > 0 или
0 >= (x MOD y) > y, если x < 0
```

При делении на ноль выдается ошибка **ZERODIVIDE**. Во всех остальных случаях результатом является ошибка операции интерпретации **BASEFUNCERROR**.

Знак **“%”** в качестве аргумента имеет тип **spec**.

Примеры целочисленного деления:

```
(13,5):% ⇒ (2, 3)
(13,-5):% ⇒ (-2, 3)
(-13,5):% ⇒ (-2, -3)
(-13,-5):% ⇒ (2, -3)
```

Подобная трактовка отличается от математической, рассматриваемой, например в первом томе книги Кнута [Кнут]:

```
x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y, если y > 0 или
```

```
0 >= (x MOD y) > y, если y < 0
```

Кстати, Кнут и не называет операцию **MOD** остатком. Получить математические версии функции, использую предопределенную операцию можно следующим образом:

```
// функция целочисленного деления:
div << funcdef x {
  dm<<x:%:1;
  [((dm,0):[>=,<]):?]^
  (
      dm,
      {(dm,1):-}
  ):. >>return
}

// функция выделения остатка от целочисленного деления:
mod << funcdef x {
  dm<<x:%;
  [((dm:1,0):[>=,<]):?]^
  (
    dm:2,
    [((dm:2,0):[>=,<]):?]^
    (
      {(dm:2,1):-:-},
      {(dm:2,1):+:-}
    )
  ):. >>return
}
```

Использование знаков: “=”, “!=”, “<”, “<=”, “>”, “>=”

Представленные знаки используются как функции сравнения аргументов, двухэлементного списка данных. Подобная интерпретация используется во многих языках программирования. Элементы списка должны быть сравнимы между собой. Если на множестве сравниваемых элементов определено отношение порядка, то могут использоваться любые функции. В противном случае допускается сравнение только на равенство (**“=”**) и неравенство (**“!=”**). Допускается сравнение между собой:

- Всех числовых данных (все операции);
- Символов (все операции);
- Булевских данных (все операции);
- Спецсимволов (на равенство и неравенство);
- Типов (на равенство и неравенство);
- Функций (на равенство и неравенство).

Примечание. Дальнейшие варианты предполагается уточнять в ходе последующих работ.

Знаки **“=”**, **“!=”**, **“<”**, **“<=”**, **“>”**, **“>=”** в качестве аргумента имеют тип **spec**.

Использование знака “|”

Задает функцию нахождения длины списка. Аргумент – список данных любой размерности и любого типа элементов. Результат – целое число, задающее количество элементов в списке первого уровня вложенности. Если аргумент не является списком, то результатом является ошибка операции интерпретации **BASEFUNCERROR**. Использование функции позволяет проверить размер аргумента перед обработкой, а лишь затем начать выделение его элементов.

Примеры:

```
(a, n, (q, w), s):| ⇒ 4
(1, 2, 3, 4, 5):| ⇒  5
((1, 2, (f, d), x)):| ⇒ 1
```

Знак **"|"** в качестве аргумента имеет тип **spec**.

Использование знака "?"

Функция, вычисляющая номера позиций истинных булевских констант в булевском списке. В качестве результата формируется параллельный целочисленный список с номерами тех элементов списка аргументов, чьи значения были равны **true**. Функция полезна для организации выборочного продолжения дальнейших вычислений.

Полученные целочисленные значения используются для выбора элементов из списков данных. Например:

```
(true, false, true, false, false, true):? ⇒ [1, 3, 6]
```

Если список состоит только из ложных значений, на выходе формируется целочисленный ноль:

```
(false, false, false):? ⇒ [] ⇒ .
```

Это позволяет непосредственно использовать результат проверки для выбора элементов списка. При нулевом значении выбор из списка данных не происходит, а возвращается пустое значение:

```
(false, (7), 5):0 ⇒ .
```

Во всех остальных случаях выдается ошибка **BASEFUNCERROR**.

Знак **"?"** в качестве аргумента имеет тип **spec**.

Использование знака "#"

Используется для задания функции транспонирования элементов списка подсписков. Он аналогичен матрице, но отличается от последней тем, что количество элементов в разных строках может отличаться. Результатом является транспонированный список подсписков, в котором элементы первой строки будут состоять из первых элементов подстрок обрабатываемой строки, вторая строка будет состоять из вторых элементов и т.д. В результате транспонирования списка, состоящего из подсписков разной длины, происходит перераспределение длины строк. Последние строки будут более короткими. Данная операция в результате оказывается необратимой.

Пример:

```
((1,2,3),(4,5,6,7),(8),(9,0)):# ⇒
   ((1,4,8,9),(2,5,0),(3,6),(7))

((1,4,8,9),(2,5,0),(3,6),(7)):# ⇒
   ((1,2,3,7),(4,5,6),(8,0),(9))
```

Знак **"#"** в качестве аргумента имеет тип **spec**.

Использование знака "()"

Задает функцию охвата круглыми скобками. Аргумент – любой. Результат – одноэлементный список, содержащий аргумент. При использовании в качестве аргумента атома или списка данных она создает одноэлементные списки:

```
атом:() ⇒ (атом)

(элемент, ... элемент):() ⇒ ((элемент, ... элемент))
```

Если в качестве аргумента вступает параллельный список, то операция группировки в список выполняется над каждым из его элементов:

```
[элемент,  ...  элемент]:() ⇒
⇒ [элемент:(),  ...  элемент:()]
```

Задержанный список перед выполнением данной функции как обычно раскрывается в параллельный, а затем интерпретируется:

```
{элемент, ... элемент}:()  ⇒
⇒ [элемент, ... элемент]:() ⇒
⇒ [элемент:( ), ... элемент:()]
```

Знак **“()”** в качестве аргумента имеет тип **spec**.

Использование знака “[]”

Задает функцию преобразования в параллельный список. Если аргумент является списком данных, то он преобразуется в параллельный список путем преобразования внешних круглых скобок в квадратные. Над атомами она выполняется как пустая операция:

```
атом:[] ⇒ [атом] ⇒ атом
```

Если аргумент является списком данных, то он заменяется на параллельный список:

```
(элемент, ... элемент):[ ] ⇒
⇒ [элемент, ... элемент]
```

При аргументе, имеющим тип "параллельный список", функция выполняется над каждым из его элементов:

```
[элемент, ... элемент]:[ ] ⇒
⇒ [элемент:[ ], ... элемент:[ ] ]
```

Такое же выполнение будет и при задержанном списке в качестве аргумента. Однако, перед этим происходит раскрытие задержанного списка и вычисление каждого из его элементов:

```
{элемент, ... элемент}:[ ]  ⇒
⇒ [элемент, ... элемент]:[ ] ⇒
⇒ [элемент:[ ], ... элемент:[ ] ]
```

Знак **“[]”** в качестве аргумента имеет тип **spec**.

Использование знака “{}”

Использование функции формирования задержанного списка, задаваемой знаком "{}", позволяет создавать из других объектов задержанные списки:

```
атом:{} ⇒ {атом}

(элемент, ... элемент):{} ⇒
⇒{элемент, ... элемент}

[элемент, ... элемент]:{} ⇒
⇒ [элемент:{}, ... элемент:{}]

{элемент, ... элемент}:{}  ⇒
⇒ [элемент, ... элемент]:{} ⇒
⇒ [элемент:{}, ... элемент:{}]
```

Знак **“{}”** в качестве аргумента имеет тип **spec**.

Примечание. В текущей версии интерпретатора генерируется «ошибка интерпретации».

Использование знака “..”

Знак **".."** используется в качестве функции, формирующей список данных из числовых атомов. В качестве аргумента может выступать трехэлементный числовой список, в котором первое число задает начало интервала, второе - его конец, а третье - шаг. Числа могут быть как целые, так и действительные, а шаг принимать как положительные, так и отрицательные

значения. Необходимо отсутствие расхождений между значением шага и границами интервала.

Кроме этого аргумент функции может быть двухэлементным целочисленным списком. В этом случае первый элемент определяет нижнюю целочисленную границу интервала, а второй верхнюю. При этом нижняя граница должна быть меньше или равной верхней границе, а шаг по умолчания принимается равным единице. При некорректном задании границ интервала данная функция возвращает ошибку **BOUNDERROR**. Если некорректно задана структура списка или тип его элементов, то возвращается ошибка предопределенной функции **BASEFUNCERROR**.

Примеры:

```
(-3.5, 2.0, 1.5):.. ⇒   (-3.5, -2.0, -0.5, 1.0)
        (1, 5):.. ⇒   (1, 2, 3, 4, 5)
      (2, 1):.. ⇒ (BOUNDERROR, (2,1))
```

Знак **“..”** в качестве аргумента имеет тип **spec**.

Примечание. Пока, при неправильных границах возвращается ошибка интерпретации и выполнение завершается.

Использование данных

Многие данные тоже могут допускать различное толкование в зависимости от того, в какой части операции интерпретации они встретились.

Использование целочисленной константы

Целочисленная константа может интерпретироваться как функция выбора элемента из списка. Аргумент – список любой размерности, содержащий элементы любого типа. Результат зависит от значения константы.

Если константа является положительном числом в диапазоне от 1 до величины, равной длине списка, то результат равен элементу из этого списка, порядковый номер которого соответствует значению константы. Если значение константы превышает длину списка, выдается ошибка **BOUNDERROR**, сигнализирующая о выходе за границу диапазона.

Целочисленная отрицательная константа интерпретируется как функция исключения элемента из списка. Аргумент – список любой размерности и любого типа элементов. Результат – список, полученный из аргумента путём удаления из него элемента, чей порядковый номер соответствует абсолютному значению аргумента. Если абсолютное значение константы превышает длину списка, выдается ошибка **BOUNDERROR**, сигнализирующая о выходе за границу диапазона.

Нулевое значение константы интерпретируется как функция, осуществляющая возврат в качестве результата пустого значения, обозначаемого **“.”**.

Примеры:

```
(234, 56.75, F, 3.14):2 ⇒ 56.75
(35, 23, 45, 76):[1, 3] ⇒ [35, 45]
(10, 9, 23, 43, 22):-4 ⇒ (10, 9, 23, 22)
(234, 56.75, F, 3.14):0 ⇒ .
```

Целочисленная константа в качестве аргумента имеет тип **int**.

Использование булевской константы

Булевские величины, при использовании в качестве функций, играют роль клапана. Если значение селектора равно **true**, то аргумент выдается в качестве результата. При значении равном **false** результатом является пустое значение. Подобная интерпретация булевское величины позволяет в дальнейшем фильтровать результаты селекции с использованием списка данных.

Примеры:

```
(x, y):true   ⇒ (x, y)
   (x, y):false  ⇒ .
     t: true   ⇒ t
      t:false  ⇒ .
(1:true, 2: false) ⇒ (1, .) ⇒ (1)
      (.):true  ⇒  (.)
        .:true  ⇒ .
        .:false  ⇒ .
```

Булевская константа в качестве аргумента имеет тип **bool**.

Следует отметить, что предопределенное использование булевской константы в качестве селектора не позволяет непосредственно реализовать селекцию, аналогичную условному оператору. Однако существует несколько приемов, позволяющих решить эту задачу. Например, можно использовать дополнительные математические преобразования в целое с вычитанием из двойки:

```
(expr1,expr2):[(2,(x,0):=:int):-]
```

Другим возможным вариантов является использование альтернативных условий, которые после синхронизации в списке данных порождают необходимое выражение, раскрываемое преобразованием в параллельный список:

```
(expr1:=,expr2:!=):[]
```

Использование специальных функций

Использование функции “dup”

Функция обеспечивает создания списка из одинаковых элементов путем дублирования. Аргумент – двухэлементный список, первый элемент которого – любая допустимая в языке конструкция, а второй – целочисленная константа. Результат – список, элементами которого являются копии первого элемента аргумента, а количество элементов результирующего списка равно значению второго элемента аргумента.

Пример:

```
(10, 5):dup ⇒ (10, 10, 10, 10, 10)
```

Функция **dup** в качестве аргумента имеет тип **func**.

2.13. Использование предопределенных типов

Механизм работы с типами, используемый в настоящее время, является традиционным для языков с динамической типизацией. Все предопределенные данные имеют признак (тег), задающий их тип. Значение размещается либо непосредственно за тегом или расположено в памяти и доступно через указатель на некоторую область памяти. Любая операция перед выполнением анализирует теги аргументов и в соответствии с этим интерпретирует значение. Формально объект данных можно представить в виде двойки:

Структура элемента = (тип, величина).

Наряду с обработкой данных, осуществляемой неявно, допускается выделять тип любого элемента данных. Для этого используется предопределенная операция **type**. Формируемая при этом величина принадлежит к «типовым» и имеет точно такую же организацию, как и любой другой аргумент. Ее специфика проявляется лишь в том, что типом аргумента является **type**.

Структура типового элемента = (type, значение типа).

Имена предопределенных типов также могут использоваться интерпретироваться в качестве функций и данных. Если имена типов используются в качестве данных, то в роли функций могут выступать сравнения, что позволяет сравнивать типы различных объектов и проверять принадлежность некоторого объекта заданному типу. Для выделения типа

объекта используется предопределенная функция **type**, аргументом которой является объект, а результатом – значение его типа.

Например:

```
10:type  ⇒  int
3.14:type  ⇒  float
(1, 2, (4, 7)) :type ⇒ datalist
[1, 2, 3, (3, 4)] :type ⇒ [int, int, int, datalist]
{x, y, z}:type ⇒ [int,int,int]
```

Функция **type** в качестве аргумента имеет тип **func**.

Следует отметить, что данная функция не определяет тип для параллельных и задержанных списков. Применение функции **type** к «типовому» элементу невозможно и ведет к ошибке интерпретации **TYPEERROR**, например:

int : type ⇒ TYPEERROR

Использование предопределенных типов в качестве функций позволяет осуществлять преобразование объектов.

Функция **int** осуществляет преобразование к целочисленной величине действительных символьных и булевых значений. Если аргумент является символом, то в качестве результата преобразования берется значения кода символа в соответствии с используемой таблицей кодировки. Если же аргумент - булева величина, то значение **false** преобразуется в 0, а **true** - в 1. Действительные числа преобразуются с округлением в соответствии с общепринятыми математическими правилами. При невозможности преобразования действительных чисел к целым возвращается ошибка целочисленного переполнения.

Примечание. В существующей версии интерпретатора действительные числа не округляются, а просто берется целая часть. Надо исправить! Или привести в соответствие с C++ (может так и есть).

Вместо ошибки для слишком больших чисел возвращается нулевое значение. Надо исправить!

Преобразование русских букв осуществляется некорректно! Они становятся отрицательными числами. Необходимо исправить!

Функция **float** осуществляет аналогичные преобразования булевских, целых и символьных величин к действительному значению.

Функция **char** обеспечивает перевод целых чисел в символы. Если значение целого числа выходит за диапазон таблицы, то возвращается ошибка выхода за границы диапазона.

Примечание. Преобразование чисел осуществляется некорректно! Выход за границы диапазона не отлавливается. Допускаются отрицательные значения. В преобразовании участвуют и действительные числа. Необходимо исправить!

Функция **bool** преобразует целые и действительные числа в булевское значение. Значение **false** формируется при аргументе, равном нулю, а значение **true** - при любом отличном от нуля входном значении.

Примечание. Осуществляется преобразование символов, что вряд ли имеет смысл. Необходимо исправить!

Функция **datalist** является аналогом предопределенной функции **"( )"**.

Функция **parlist** является аналогом предопределенной функции **"[ ]"**.

Функция **delaylist** является аналогом предопределенной функции **"{}"**.

Примечание. **delaylist***, как и {}, отрабатывает некорректно. Необходимо разобраться с семантикой!*

Функция **signal** преобразует любой вычисленный объект в сигнал (пустое значение).

Примечание. Должно формироваться пустое значение. Необходимо исправить!

Функция **error** в данной версии не интерпретируется.

2.14. Пользовательские типы

Инструментальная поддержка механизма динамически порождаемых пользовательских типов позволяет создавать аналоги абстрактных типов данных. Для этого используется дополнительных конструкций:

1. Определение пользовательского типа;
2. Сравнения пользовательских типов на равенство и неравенство.
3. Проверка на принадлежность некоторого значения величине, допустимой для заданного пользовательского типа.
4. Преобразование в пользовательский тип.
5. Разыменование пользовательского типа.

Определение пользовательского типа задается соответствующим предикатом, сопоставляющим проверяемый элемент некоторому выражению. Если результат проверки является истиной, то элемент принадлежит проверяемому типу. Предикат оформляется в виде специальной функции **typedef**, возвращающей булевское значение. Ее обозначение регистрируется в таблице пользовательских типов.

В качестве примера можно рассмотреть, как задаются треугольник и круг:

```
// Описание пользовательского типа, задающего треугольник как
// трехэлементный целочисленный список
Triangle << typedef X {
  [(((X:type,datalist):=,(X:|,3):=):*:int,1):+]^
  (
    false,
    {([(X:1:type,int),(X:2:type,int),(X:3:type,int)]:=):*}
  ):.
  >> return
}

// Описание пользовательского типа, задающего круг как
// целочисленный атом
Circle << typedef X
typedef X
{
  (X:type,int):= >> return;
}
```

Сравнение пользовательских типов осуществляется точно также как и сравнение базовых типов языка: выделяется тип элемента функцией **type**, проверяется совпадение имен выделенного и проверяемого типа. Результат сравнения является истиной при совпадении имен типов. Ниже приводится пример использования сравнения пользовательских типов для описания типа обобщенной геометрической фигуры.

```
// Описание фигуры, являющейся треугольником или кругом
Figure << typedef X {
  // Аргумент - треугольник или круг
  X:type >> t;
  ([(t, Triangle), (t, Circle)]:=):+ >> return;
};
```

Проверка на принадлежность позволяет выяснить возможность соответствия между динамически формируемыми данными и **typedef**. Для этого используется функция **in**, которая возвращает значение, полученное в результате выполнения предиката, заданного в описании пользовательского типа. Принадлежность позволяет в дальнейшем осуществить преобразование проверяемого аргумента в элемент пользовательского типа. Ниже представлены примеры использования функции принадлежности:

```
((10,20,15),Triangle):in ⇒ true
((10,20,15),Circle):in ⇒ false
(10,Circle):in ⇒ true
```

Преобразование в пользовательский тип используется для формирования требуемых абстракций по принципу «обертки» преобразуемых данных. Является расширением операции преобразования базовых типов. Суть заключается в получении нового значения элемента, следующей структуры:

Элемент пользовательского типа = (пользовательский тип, преобразуемый элемент).

Само преобразование задается указанием пользовательского типа в качестве функции и осуществляется в зависимости от значения аргумента:

1. Если тип аргумента совпадает с типов в операции преобразования, то возвращается значение исходного аргумента.
2. Преобразование осуществляется только в том случае, если проверка аргумента на принадлежность функцией **in**, осуществляемая неявно, дает «истину».
3. Во всех остальных случаях функция преобразования в пользовательский тип возвращает ошибку **TYPEERROR**.

Использование данной операции позволяет формировать необходимые абстракции при выполнении программы:

```
(10,20,15):Triangle ⇒ Треугольник со сторонами (10,20,15)
```

Описанная операция не обеспечивает автоматического преобразования пользовательских типов друг в друга, даже если их значения принадлежать единому подмножеству. Данное ограничение введено для более строгого контроля. Зачастую подобные преобразования бывают необходимы. В этом случае можно воспользоваться **разыменованием пользовательского типа**, заключающемся в выделении «обернутого» значение функцией **value**. Данная функция «отбрасывает» пользовательский тип, тем самым «обезличивая» преобразуемый элемент:

```
(10,20,15):Triangle:value ⇒ (10,20,15)
(10,20,15):Triangle:value:1:Circle ⇒ Круг радиусом 10
```

Попытка применить операцию разыменования к базовым типам ведет к генерации ошибки **VALUEERROR**:

```
10:value ⇒ VALUEERROR
```

2.15. Правила эквивалентных преобразований

Правила эквивалентных преобразований уже рассматривались при описании модели вычислений. Ниже они сведены воедино с учетом дополнительно введенных конструкций.

1. Слияние параллельных списков в списке данных:

$$([X_1], [X_2], \ldots, [X_n]) \equiv (X_1, X_2, \ldots, X_n) \ .$$

2. Эквивалентность параллельных списков набору их элементов:

$$[x_1, x_2, \ldots x_n] \equiv x_1, x_2, \ldots x_n \ .$$

3. Интерпретация параллельных списков:

$$[x_1, \ldots x_n]:[f_1, \ldots f_k] \equiv$$
$$\equiv x_1:f_1, \ldots x_1:f_k, \ldots x_n:f_1, \ldots x_n:f_k \ .$$

Как частные случаи можно рассмотреть ситуации, когда функция или аргумент являются атомами:

$$[x_1, x_2, \ldots x_n]:f \equiv x_1:f, x_2:f, \ldots x_n:f,$$
$$x:[f_1, f_2, \ldots f_k] \equiv x:f_1, x:f_2, \ldots x:f_k.$$

4. Эквивалентность многократно вложенных задержанных списков:

$$\{\{ X \}\} \equiv \{ X \} \quad .$$

5. Эквивалентность формирования списков данных:

$$X:(F) \equiv (X:F) \quad .$$

При пустом списке данных в качестве функции имеем:

$$X:(.) \equiv (X) \quad .$$

6. Эквивалентность пустого элемента и пустого параллельного списка:

$$. \equiv [.] \equiv \{.\} \quad .$$

7. Данное правило определяет размножение альтернативной части операции интерпретации, если его аргумент и функция являются параллельными списками. В этом случае альтернатива приписывается каждой созданной операции интерпретации:

$$[x_1, \ldots x_n]:[f_1, \ldots f_k] \text{ else } Z \equiv$$
$$\equiv [x_1:f_1 \text{ else } Z, \ldots x_1:f_k \text{ else } Z, \ldots$$
$$\ldots x_n:f_1 \text{ else } Z, \ldots x_n:f_k \text{ else } Z]$$

8. Если список данных содержит пустой элемент ".", то этот элемент исключается из списка. При этом длина данного списка уменьшается на количество содержавшихся пустых элементов.

Примеры:

$$(x1, x2, ., ., x3) \equiv (x1, x2, x3)$$