Язык программирования AWL:
предварительное описание

Версия: ***14.02.2008***

Оглавление

Введение

Язык программирования AWL — это интерпретируемый язык (или *язык сценариев*), предназначенный для решения разнообразных задач обработки числовых и символьных данных. AWL во многом основан на идеях других языков (таких, как C, Java, LISP, Perl, JavaScript и других) — но заметно отличается от всех перечисленных, имея ряд собственных интересных особенностей. Одна из перспективных целей AWL — быть основой новой технологии создания гипертекстовых документов, предлагающей единую интегрированную среду для описания структурного документа и средств динамического управления его контентом.

Здесь описано ядро языка — та функциональная основа, на которой могут быть реализованы прикладные AWL-среды.

Рабочая среда AWL и организация диалога

Язык AWL реализован как *интерпретатор*. С точки зрения системы, AWL-интерпретатор — прикладная программа (обычно, **awl.exe**), которая может работать либо в пакетном, либо в интерактивном режиме. Простейший синтаксис ее вызова таков:

```
awl.exe File1.awl File2.awl ... FileN.awl
```

где file1.awl—fileN.awl: имена файлов с исходным кодом на awl (модули). Работа интерпретатора состоит в том, что он последовательно выполняет все перечисленные модули, после чего завершается.
Если ни одного модуля в командной строке не задано, интерпретатор будет работать в *интерактивном* режиме: последовательно читать вводимые пользователем AWL-выражения, распознавать и выполнять их. Далее диалог пользователя с интерпретатором будет показываться в таком виде:

```
ввод пользователя ->
  -> вывод интерпретатора
```

По умолчанию, при интерактивной работе включен режим *трассировки*: это значит, что после каждого шага интерпретатор выводит выражение и результат его вычисления (в виде: `выражение => результат`). Простейший диалог пользователя с AWL-интерпретатором может выглядеть, например, так:

```
2*2 + 3*3;
  add:(mul:(2, 2), mul:(3, 3)) => 13;

sqr(5) + sqr(7);
  add:(sqr:5, sqr:7) => 4.8818193;
```

Заметьте, что все вводимые пользователем выражения должны разделяться точками с запятой. (При вводе выражение может занимать произвольное количество строк.) Как выражения, так и результаты выводятся во *внутреннем* формате, который (как правило) немного отличен от формата ввода (он подробно описан в дальнейших разделах документа).
Диалог с интерпретатором может продолжаться сколь угодно долго. Способ его завершения в общем случае системно-зависим (в Windows необходимо ввести строку, состоящую из одного символа ^Z (Ctrl + Z)).

Элементы данных

Язык AWL поддерживает разнообразные типы и форматы данных. Принципиальной особенностью AWL (роднящей его с такими языками, как LISP и Scheme) является то, что AWL-выражения также являются элементами данных. В отличие от многих языков, AWL не проводит четкого разграничения между кодом и данными: любое выражение языка может иметь как значение (результат вычисления), так и побочный эффект вычисления. Первый аспект характеризует его как элемент данных, второй — как элемент исполняемого программного кода. Очень часто эти аспекты существуют во взаимодействии: так, вызов определенного пользователем *функтора* обычно имеет как некий побочный эффект, так и результат.
Примитивными элементами данных в языке AWL являются *скаляры*, *списки, термы* и *блоки*.

Скаляры

Скаляры — это простейшие элементы данных, не обладающие внутренней структурой и нетривиальной семантикой выполнения. Вычисление скаляров всегда дает определенное значение. В зависимости от типа, скаляры подразделяются на *числовые* и *строковые*.
Числовые скаляры представляют числа (целые или с плавающей точкой). Литеральная запись числового скаляра имеет традиционный для большинства языков программирования (например, C) вид:

```
1
2
3026
3.14159
0.0243
145.3497e-12
```

Здесь первые три значения относятся к целому типу, последние три — к вещественному. Строго говоря, эти два подтипа числового типа различаются: внутреннее представление для целых и вещественных чисел различно. В основном язык обеспечивает неявное преобразование числовых данных (когда это необходимо), но в ряде случаев различие в семантике целых и вещественных типов играет свою роль.
Формат, в котором интерпретатор выводит числа, практически не отличается от формата ввода (вещественные данные могут выводиться в экспоненциальном формате, если интерпретатору это кажется удобным).
Строковые скаляры — это символьные строки неограниченной длины. В отличие от C, строка может содержать произвольные ASCII-символы (включая нулевой). Строковый литерал при вводе может ограничиваться либо одинарными, либо двойными кавычками (что позволяет включать в него кавычки альтернативного типа). Символ '\' в строке дает возможность помещать в нее ряд специальных символов (в основном тех же, что доступны и в C). Вот примеры строковых литералов:

```
''          `(пустая строка)`
"A"
'ABCD'
"1x, 2y, 3z"
'< --- >'
"Hello, world!"
"Это строка #1,
 это строка #2
 и это строка #3."
```

В последнем примере видно, что строковый литерал может занимать несколько строк исходного кода. Формат, в котором интерпретатор выводит строки, также не таит особых сюрпризов (в качестве ограничителей всегда используются двойные кавычки, а некоторые управляющие символы могут кодироваться специальным образом).

Списки

Одной из самых важных структур данных языка является *список*. Список — это последовательность из более чем одного элемента данных. Элементы данных могут быть произвольными: скалярами, другими списками, термами, блоками и т. п. Единственное принципиальное ограничение — последним элементом списка не может быть другой список или неопределенность. (Строго говоря, если добавить список в качестве последнего элемента он просто перестанет быть самостоятельным списком, т. к. будет сцеплен (конкатенирован) с предыдущими).
При вводе список в общем случае задается *конструктором списка.* Простейшая форма

конструктора списка – перечисление элементов, разделенных запятыми, и заключенных в круглые скобки. Вот несколько примеров задания списков:

```
(567, "aa bb cc dd", 432, '9 8 7', 123.321)
((10, 'Aa'), (25, 'Bb'), (31, 'Cc'), 0)
('1', ('2.1', '2.2'), (('3.1.1', '3.1.2', '3.1.3'), ('3.2.1',
'3.2.2', '3.2.3')), 0)
```

Приведенные списки имеют (соответственно) 5, 4 и 4 элемента. В последнем случае, если б отсутствовал завершающий элемент 0, список неявно принял бы такой вид (6 элементов):

```
('1', ('2.1', '2.2'), ('3.1.1', '3.1.2', '3.1.3'), '3.2.1',
'3.2.2', '3.2.3')
```

т. к. последний элемент потерял бы свою структуру.
Синтаксически конструктор списка имеет самый низкий приоритет — т. е. элементами списка могут быть произвольные выражения языка. В тех случаях, когда все элементы списка являются *синтаксически замкнутыми* выражениями допускается более компактная запись списков, при которой в качестве ограничителей используются квадратные скобки, а явные разделители не нужны:

```
[567 "aa bb cc dd" 432 '9 8 7' 123.321]
```

Замкнутыми выражениями (имеющими наивысший синтаксический приоритет) являются литералы, переменные, блоки и другие списки в любой из двух форм. (Это чисто синтаксическое понятие — любое выражение можно сделать замкнутым, просто заключив его в круглые скобки).
Список имеет простую и однозначную семантику выполнения: последовательно вычисляются все элементы списка, и результатом выполнения становится список результатов. Выводится интерпретатором список всегда в первой форме (в круглых скобках и через запятые).

Неопределенность

В языке присутствует пустое, или неопределенное значение (*неопределенность*). С точки зрения семантики оно равносильно пустому списку (и ведет себя во многом подобно атому **nil** в LISP/Scheme).
Неопределенность (иногда также называемая **undef**) чаще всего возникает в результате неких вычислений. Когда (не часто) ее необходимо задать явно, можно сделать это в виде () или []. Интерпретатор всегда выводит **undef** в виде (). Выполнение **undef** не имеет никакого эффекта (поэтому **undef** не только пустое значение, но и аналог пустого оператора во многих языках программирования).

Термы

Не менее важными, чем списки, структурами данных, являются *термы*. Терм — это произвольное выражение языка (называемое *аргументом* терма), снабженное *префиксом* (идентификатором), ссылающимся на определенный ранее *функтор* или *класс*. Вот несколько примеров термов:

```
add(A, B)
mul(sub(A, 2), add(C, 3))
date(2006, 1, 15)
title("Месячный отчет: ", bold(Year), "-", bold(Month))
```

Термы имеют нетривиальную семантику вычисления, которая полностью определяется его

префиксом. В зависимости от того, ссылается ли он на *встроенный функтор*, *пользовательский функтор* или *класс*, терм вычисляется и возвращает значение. В сущности, выполнение AWL-программы в основном состоит из вычисления разнообразных термов. По этой причине для наиболее "популярных" термов предусмотрен альтернативный синтаксис (подробнее об этом в дальнейших разделах).
Аргументом терма в большинстве случаев является список. Он задается только в круглых скобках (чтобы избежать конфликта с обращением к элементам списка, см. далее), но в случае, когда аргумент является скалярным литералом, скобки можно опустить:

```
sin 1;
```

```
sin:1 => 0.84147098;
```

Независимо от того, каким образом терм был задан, интерпретатор выводит его в следующем виде: `префикс:аргумент`.

Блоки

Когда выражение представляет собой определенную последовательность вычислений или действий, есть смысл группировать их в блоки. Синтаксически *блок* – это произвольная последовательность выражений, заключенная в фигурные скобки, и разделенная точками с запятой. Блок является замкнутым выражением.
Вычисление (или выполнение) блока состоит в последовательном вычислении всех его элементов, причем значение последнего элемента возвращается в качестве значения блока.

```
{ a = b + c; b =*: 2; c =/: 2 }    ` пример блока `
```

В принципе, все, что можно сделать с помощью блоков, можно и с помощью списков. Существенная разница заключается лишь в том, что результатом вычисления списка является список из содержащихся в нем значений, а результатом вычисления блока – только одно (последнее) вычисленное значение. Однако, при выполнении программы это может существенную сэкономить ресурсы – поэтому в тех случаях, когда "промежуточные" значения не нужны, использование блоков предпочтительно.
Если в блоке между двумя разделителями (;) ничего не задано, предполагается (). Формат вывода интерпретатором блоков идентичен синтаксису ввода, но для лучшей читаемости каждый элемент блока выводится с новой строки (и, возможно, с отступом, пропорциональным глубине вложенности).

Идентификаторы

Идентификаторы — это символические имена. Все идентификаторы AWL обозначают *переменные* или *префиксы*.
Синтаксис идентификаторов в AWL традиционен: идентификатор состоит из букв латинского алфавита, десятичных цифр и знаков подчеркивания (но при этом не должен начинаться с цифры). Важно, что при этом заглавные и строчные буквы *различаются*. Вот примеры допустимых идентификаторов:

```
A
xyz
average_value
link122
ParagraphCounter
```

Идентификаторы всегда интерпретируются в контексте некого *пространства имен*. При этом имена переменных и имена префиксов всегда относятся к двум *разным* пространствам имен. По умолчанию, всегда существуют *глобальные* пространства имен для переменных и

префиксов (относящиеся к текущему модулю или текущей сессии интерпретатора).

Пробелы и комментарии

В промежутках между литералами и/или идентификаторами допускаются произвольные пробельные символы (в т. ч. собственно пробелы, табуляции или концы строк). В некоторых случаях пробелы необходимы — например, между двумя соседними выражениями, не разделенными явно.
В этих же местах допустимы *комментарии*. Комментарий — это произвольная непустая последовательность символов, заключенная в *обратные кавычки* (и не содержащая таких кавычек сама).

```
` Это – простой комментарий `
```

Комментарий также может занимать несколько строк программы подряд.

Программа и ее выполнение

Обычно AWL-программа представляет собой единственный файл (*модуль*) исходного кода. Он, так же как и блок, представляет собой последовательность исполняемых инструкций (разделенных точками с запятыми), роль которых играют произвольные выражения, а также определения (функторов или классов).
По завершении выполнения модуля, интерпретатор выводит краткую статистику. Если у нас есть файл с исходным кодом (скажем, **test.awl**):

```
a = 12; b = 23;
a*2 + b*3;
(a - b) * (a + b)
```

результат его обработки интерпретатором будет примерно таким:

```
<test> :: {
set:(a, 12) => 12;
set:(b, 23) => 23;
add:(mul:(a, 2), mul:(b, 3)) => 93;
mul:(sub:(a, b), add:(a, b)) => -385;
} :: #4 / [4]
```

Здесь <test> – имя модуля, [4] – число инструкций “верхнего уровня” и #4 – число строк в исходном коде.

Скалярные операции

В AWL имеется большое количество *скалярных операций,* для которых характерно то, что и их операндами, и результатами обычно являются значения скалярного типа. Как и большинство операций языка, каждой соответствует встроенный в язык (предопределенный) функтор, а обращение к этой операции – это терм с данных функтором-префиксом. Этот терм может быть записан в явном виде (эта запись называется нормальной формой), когда же используется альтернативная (операционная) форма записи, интерпретатор все равно неявно преобразует ее в терм. Таким образом, с точки зрения семантики обе формы записи совершенно равносильны, какую из них предпочесть — дело вкуса. Далее в таблицах приведены обе формы для каждой операции.
При разборе выражений в операционной форме, учитываются приоритеты операций. Все операции разбиты на следующие приоритетные группы (в порядке уменьшения их приоритета):

- унарные (префиксные / постфиксные);

- мультипликативные (группа умножения);
- аддитивные (группа сложения);
- максимум и минимум;
- компаративные (сравнения);
- битовые/логические;
- управляющие (условные и циклические);
- присваивания и ввод/вывод.

Арифметические скалярные операции

Для всех этих операций операнд(ы) и возвращаемое значение являются числами (целыми или вещественными). Здесь (и далее) для компактности будут использоваться следующие обозначения:
- `A, B, C`: предполагают произвольные числовые значения;
- `I, J, K`: целые значения;
- `X, Y, Z`: вещественные значения.

Если реальное значение не совпадает с требуемым, выполняются *приведения* (о них подробно далее).

Унарные операции:

Синтаксис	*Нормальная форма*	*Семантика*
`- A`	**neg**(A)	*Арифметическое отрицание* `A`
`+ A`	**abs**(A)	*Абсолютная величина* `A`
`~ I`	**not**(I)	*Битовая инверсия I*

Бинарные мультипликативные операции:

Синтаксис	*Нормальная форма*	*Семантика*
`A * B`	**mul**(A,B)	*Произведение A и B*
`X / Y`	**div**(X,Y)	*Частное X и Y*
`I % J`	**idiv**(I,J)	*Целочисленное частное I и J*
`I %% J`	**irem**(I,J)	*Целочисленный остаток I и J*
`I << J`	**shl**(I,J)	*Арифметический сдвиг I влево на J битов*
`I >> J`	**shr**(I,J)	*Арифметический сдвиг I вправо на J битов*

При выполнении целочисленного деления, происходит округление к нулю (как в C), и всегда справедливо тождество: `I == (I%J) * J + (I%%J)`.

Бинарные аддитивные операции:

Синтаксис	*Нормальная форма*	*Семантика*
`A + B`	**add**(A,B)	*Сумма A и B*
`A - B`	**sub**(A,B)	*Разность A и B*

Бинарные операции максимума и минимума:

Синтаксис	*Нормальная форма*	*Семантика*
`A ?< B`	**`min`**`(A,B)`	*Минимум (меньшее из A и B)*
`A ?> B`	**`max`**`(A,B)`	*Максимум (большее из A и B)*

Бинарные операции арифметического сравнения (компаративные):

Синтаксис	*Нормальная форма*	*Семантика*
`A < B`	**`lt`**`(A,B)`	*A меньше B ?*
`A > B`	**`gt`**`(A,B)`	*A больше B ?*
`A <= B` `A ~> B`	**`le`**`(A,B)`	*A меньше или равно B ?*
`A >= B` `A ~< B`	**`ge`**`(A, B)`	*A больше или равно B ?*
`A == B`	**`eq`**`(A, B)`	*A равно B ?*
`A ~= B` `A <> B`	**`ne`**`(A, B)`	*A не равно B ?*
`A <?> B`	**`cmp`**`(A, B)`	*Результат сравнения A и B*

В AWL нет специального логического типа: значение 0 считается ложью, 1 (в общем случае – не ноль) – истиной. Все операции сравнения (кроме <?>) возвращают 1, если условие истинно, и 0 в противном случае.
Последняя операция (**cmp**) возвращает значение -1 (если `A < B`); 1 (если `A > B`) или 0 (если `A == B`).

Бинарные битовые/логические операции:

Синтаксис	*Нормальная форма*	*Семантика*
`I & J`	**`and`**`(I,J)`	*Битовое "и" для I и J*
`I | J`	**`or`**`(I,J)`	*Битовое "или" для I и J*
`I ~ J`	**`xor`**`(I,J)`	*Битовое "исключающее или" для I и J*

(Внимание: в отличие от C и Java, символом операции XOR является '~', а не '^'!)
Любое выражение легко привести к нормальной форме: например, выражение `A*B + C/D` равносильно **`add`**(**`mul`**`(A,B)`, **`div`**`(C,D)`). Однако, помимо перечисленных операций, имеется еще ряд встроенных (как с одним, так и с двумя операндами), не имеющих специальной операционной формы, и доступных только в виде термов:

Арифметические операции:

Нормальная форма	*Семантика*
`floor``(X)`	*Округление X вниз до ближайшего целого*
`ceil``(X)`	*Округление X вверх до ближайшего целого*
`sqr``(X)`	*Квадратный корень из X*
`exp``(X)`	*Экспонента от X*
`log``(X)`	*Натуральный логарифм от X*
`exp_by``(X,Y)`	*X в степени Y*
`log_by``(X,Y)`	*Логарифм X по основанию Y*

Нормальная форма	*Семантика*
sin(X)	*Синус X*
cos(X)	*Косинус X*
tan(X)	*Тангенс X*
asin(X)	*Арксинус X*
acos(X)	*Арккосинус X*
atan(X)	*Арктангенс X*
sinh(X)	*Гиперболический синус X*
cosh(X)	*Гиперболический косинус X*
tanh(X)	*Гиперболический тангенс X*
rad(X,Y)	*Радиус вектора (X, Y)* *(sqr (X*X + Y*Y))*
ang(X,Y)	*Угол вектора (X, Y)*
pi(X)	*Число π, умноженное на X*

Аргумент для всех тригонометрических операций (и результат для обратных тригонометрических и ang) выражен в радианах.

Строковые скалярные операции

Строковые операции легко отличить от арифметических: большинство из них имеет суффикс '$'. На них распространяется та же система приоритетов, что и на арифметические операции. Далее S, T, U — значения строкового типа.

Строковые операции:

Синтаксис	*Норм. форма*	*Приоритет*	*Семантика*
#$ S	**s_len**(S)	унарная	*Длина строки S (целое число)*
S +$ T	**s_cat**(S,T)	аддитивная	*Конкатенация (сцепление) строк S и T*
S *$ N	**s_rep**(S,N)	мультипликативная	*Репликация (повторение) строки S N раз подряд*

Например.

```
"Black" +$ " and" +$ " white";
```

```
s_cat:(s_cat:("Black", " and"), " white") => "Black and
white";
```

```
"No! " *$ 3;
```

```
s_rep:("No! ", 3) => "No! No! No! ";
```

Для строковых скаляров также определены свои операции сравнения. Сравнение имеет *словарную* семантику (т. е. строки считаются упорядоченными в соответствии с кодами символов, как в словаре), и возвращают значение, подобное их арифметическим аналогам (-1, 0 или 1 для **s_cmp**; 0 (ложь) или 1 (истина) для всех остальных).

Бинарные операции строкового сравнения (компаративные):

Синтаксис	*Нормальная форма*	*Семантика*
`S <$ T`	**`s_lt`**`(S,T)`	*S до T в словаре?*
`S >$ T`	**`s_gt`**`(S,T)`	*S после T в словаре?*
`S <=$ T` `S ~>$ T`	**`s_le`**`(S,T)`	*S до (или совпадает с) T в словаре?*
`S >=$ T` `S ~< T`	**`s_ge`**`(S,T)`	*S после (или совпадает с) T в словаре?*
`S ==$ T`	**`s_eq`**`(S,T)`	*S совпадает с T ?*
`S <>$ T` `S ~= T`	**`s_ne`**`(S,T)`	*S не совпадает с T ?*
`S <?>$ T`	**`s_cmp`**`(S,T)`	*Результат словарного сравнения строк S и T*

Наконец, для строк также имеются операции *словарного* максимума и минимума:

Бинарные операции строкового максимума и минимума:

Синтаксис	*Нормальная форма*	*Семантика*
`S ?<$ T`	**`s_min`**`(S,T)`	*Меньшая (словарно) из строк S и T*
`S ?>$ T`	**`s_max`**`(S,T)`	*Большая (словарно) из строк S и T*

Отрезок строки

Очень часто операция должна быть выполнена не над всей строкой, а над каким-нибудь ее непрерывным фрагментом. Для этого тоже есть своя операция:

Синтаксис	*Нормальная форма*	*Семантика*
`S $[R]`	**`s_slice`**`(R,S)`	*Отрезок строки S (заданный диапазоном R)*

Диапазон (range) – это понятие, которое будет встречаться еще нередко. Обычно диапазон – это список из двух целых чисел, первое из которых задает начальное значение, а второе – конечное (при этом конечное, в отличие от начального, *не включается* в диапазон). Поскольку такого рода структуры требуются часто, для их задания есть специальный синтаксис:

Синтаксис	*Нормальная форма*
`I .. J`	`((I), (J))`

Например, в диапазон 10 .. 15 входят числа (10, 11, 12, 13, 14). Диапазон всегда содержит J - I значений, если же I >= J, то диапазон является пустым, т. е. не содержит ни одного значения. Наконец, в качестве диапазона может быть задано одно единственное число N – в этом случае диапазон считается эквивалентным 0 .. N.

Операция **s_slice** возвращает фрагмент строки S, ограниченный диапазоном R (при этом отсчет символов строки начинается с 0). Например:

```
"Hello, world" $[5];
```

```
s_slice:(5, "Hello, world") => "Hello";
```

```
“Hello, world” $[7..12];
```

```
s_slice:((7, 12), “Hello, world”) => “world”;
```

Очевидно, что диапазон 0..#$S (или просто #$S) содержит все символы в строке S. Если же начало, конец или обе границы диапазона выходят за эти пределы, это не считается ошибкой: в этом случае результат дополняется (в начале и/или конце) необходимым количеством пробелов. Таким образом, строка-результат всегда имеет ровно столько символов, сколько предполагается диапазоном.

Поиск в строке

Конечно, в AWL имеются операции, позволяющие отыскать подстроку в строке.

Строковые операции поиска:

Синтаксис	*Нормальная форма*	*Семантика*
`S <<$ T`	**`s_findlast`**`(S,T)`	*Найти последнее вхождение T в S*
`S >>$ T`	**`s_findfirst`**`(S,T)`	*Найти первое вхождение T в S*

Приоритет этих операций тот же, что и у мультипликативных. Операция s_findfirst ищет первое вхождение подстроки T в строке S, начиная с начала; s_findlast ищет последнее, начиная с конца. Если подстрока найдена, возвращается позиция ее первого символа в строке (отсчет ведется с 0), в противном случае возвращается -1. Пустая строка всегда “находится” в начале/конце строки.

```
"underground" >>$ 'nd';
```

```
s_findfirst:("underground", "nd") => 1;
```

```
"underground" <<$ 'nd';
```

```
s_findlast:("underground", "nd") => 9;
```

Типизация, приведения и проверки типов

Большинство встроенных операций имеет жестко определенную семантику, включая и определенные типы своих операндов. Поэтому в AWL невозможны ситуации (столь нередкие, например, в JavaScript), когда семантика операции — например, числовая она или строковая — зависит от типов операндов периода выполнения.
Однако, ситуация, когда тип операнда не совпадает с ожидаемым, не считается ошибкой. В этом случае происходит неявное *приведение* значения к требуемому операцией типу. Любое скалярное значение может быть преобразовано к другому скалярному типу в соответствии со следующими соглашениями:

- целые числа могут преобразовываться в действительные, а действительные в целые (в последнем случае происходит отбрасывание дробной части, т. е. округление к нулю);
- числовые значения могут преобразовываться в строки (таким же образом, как это происходит при выводе числа интерпретатором);
- строковые значения преобразуются в числа — строго говоря, преобразуется та начальная часть строки, которая может быть интерпретирована как целое или действительное число. Если таковой нет, возвращается значение 0).

Здесь, хотя оба операнда — строки, операция является числовой, и результат — число:

```
'222' + '113';
```

```
add:("222", "113") => 335;
```

Вот противоположный случай (числовые операнды для строковой операции):

```
156 +$ 208;
```

```
s_cat:(156, 208) => "156208";
```

Но бывают случаи, когда необходимы *явные преобразования типов*. Их обеспечивают следующие унарные функторы:

Явные преобразования типов:

Нормальная форма	***Семантика***
int(Q)	*Преобразовать скаляр Q в целое число*
float(Q)	*Преобразовать скаляр Q в действительное число*
num(Q)	*Преобразовать скаляр Q в число (целое или действительное)*
string(Q)	*Преобразовать скаляр Q в строку*

Явные преобразования осуществляются в соответствии с теми же правилами, что и неявные. Наконец, имеется ряд функторов-предикатов (возвращающих логическое значение), которые позволяют непосредственно проверить, к какому типу принадлежит их операнд:

Проверки скалярных типов:

Нормальная форма	***Семантика***
is_int(Q)	*Скаляр Q — целое число?*
is_float(Q)	*Скаляр Q — действительное число?*
is_num(Q)	*Скаляр Q — число (целое или действительное)?*
is_string(Q)	*Скаляр Q — строка?*

Списки как операнд скалярных операций

Хотя синтаксическая форма операций обычно компактнее и яснее, знать их нормальную форму полезно по ряду причин. Например, все скалярные операции способны работать со списками напрямую.

Применение любой скалярной бинарной операции к списку, содержащему более двух элементов, возвращает результатом список, в котором *первые два элемента* заменены на *результат выполнения* соответствующей операции над ними (а “хвост” списка остается неизменным). Например:

```
a = (11, 12, 13, 14);
add(a);
```

```
add:a => (23, 13, 14);
```

```
add(add(a));
```

```
add:add:a => (36, 14);
```

```
add(add(add(a)));
```

```
add:add:add:a => 50;
```

```
mul(a);
  mul:a => (132, 13, 14);

mul(mul(a));
  mul:mul:a => (1716, 14);
```

Аналогично, применение любой унарной операции к списку, содержащему более одного элемента, возвращает в качестве результата список, в котором *первый элемент* заменен на *результат* этой операции над ним (а "хвост" списка остается неизменным). Например:

```
neg(a);
  neg:a => (-11, 12, 13, 14);
```

Операция редукции

Иногда нужен результат применения той или иной операции ко всем элементам списка — например, чтобы просуммировать все элементы списка, нужно последовательно применять бинарную операцию + (add) ко всем его элементам. Для таких операций имеется специальная операция (*редукция*):

```
[=] + a;
  reduce:add:a => 50;
```

По сути, редукцию можно назвать *мета-операцией* (т. е. операцией над операцией). За символом [=] следует знак соответствующей бинарной операции, а затем операнд. Эта конструкция преобразуется в обращение к встроенному функтору **reduce**, выполняющему необходимую задачу: начиная с первого элемента, заданная операция последовательно применяется к текущему результату и следующему элементу, до тех пор, пока не достигнут конец списка. Так, `[=] + L` возвращает сумму всех элементов в списке L, `[=] * L` — произведение элементов, `[=] ?> L` и `[=] ?< L` — соответственно, большее и меньшее значение в списке, и т. п. Если редукция просто применяется к скаляру, возвращается его значение, если к **undef** – возвращается **undef**.

Переменные и присваивания

Как и во всех языках программирования, в AWL присутствуют *переменные*.
Простейший вид переменных AWL — *глобальные* для текущего модуля. Они не требуют предварительного описания или объявления, и создаются автоматически: любое упоминание переменной с заданным именем вызывает ее к жизни. AWL — бестиповой язык, и переменным также не свойственны строго заданные типы: любая переменная может иметь произвольное значение, и менять его в процессе выполнения программы.
Пока переменной не присвоено значение, она содержит **undef**. Основным способом изменения значения переменных является *присваивание*, которое в AWL (как и в C-подобных языках) является *операцией* (другими словами, встроенным в язык функтором).
Менее обычно то, что в языке имеется две разные операции присваивания: *немедленное* (**set**) и *латентное* (**let**).

Операции присваивания:

Синтаксис	***Нормальная форма***	***Семантика***
`V = W`	**`set`**`(V, W)`	*Немедленное присваивание:* *присвоить V вычисленное значение W)*

Синтаксис	*Нормальная форма*	*Семантика*
`V := W`	**`let`**`(V, W)`	*Латентное присваивание: присвоить V выражение W (без вычисления последнего)*

Немедленное присваивание

Операция немедленного присваивания эквивалентна функтору **set**. Его выполнение состоит в том, что происходит вычисление операнда W, и его результат присваивается операнду V. Операция требует, чтобы операнд V был *мутабельным* (т. е. доступным для изменения). Простейший (но не единственный) вид мутабельного выражения — переменная. Вот несколько примеров присваивания значений переменным:

```
a = 12*3;
b = 'Hello, ' +$ 'world!';
c = (2*2, 3*3, 4*4);
d = (('x' *$ 3, 'y' *$ 2), 'z' *$ 5)
```

В результате четыре переменные получат следующие значения:

```
a; b; c; d;
```

```
a => 36;
b => "Hello, world!";
c => (4, 9, 16);
d => (("xxx", "yy"), "zzzzz");
```

Более интересно то, что мутабельным выражением также является *список*, элементами которого являются мутабельные выражения. Таким образом, мутабельному списку может быть присвоен любой список (при этом списки присваиваются *поэлементно*):

```
[a b c] = (10.5,12.9,14.3);
[d e] = ('Yes', 'No')

a; b; c; d; e;
```

```
a => 10.5;
b => 12.9;
c => 14.3;
d => "Yes";
e => "No";
```

При этом, если какой-нибудь из элементов списка W имеет внутреннюю структуру, она сохраняется при присваивании. Несовпадение длин списков-операндов также не считается ошибкой. Именно, если левый список короче, его последний элемент получает в качестве значения оставшуюся неприсвоенной часть правого списка. Таким образом, значения в присваиваемом списке никогда не “пропадают” бесследно. (Perl-программисты найдут такую семантику присваивания списков знакомой.)

```
[a b c] = [10 22 35 67];

a; b; c;
```

```
a => 10;
b => 22;
c => (35, 67);
```

Если же имеет место обратная ситуация (т. е. левый список длиннее правого), всем переменным, на которые “не хватит значений”, будет присвоено **undef**.

```
[a b c] = ['ABC' 'DEF'];

a; b; c;
```

```
a => "ABC";
b => "DEF";
c => ();
```

(Заметим, что, как следствие этого, чтобы стереть переменную V (т. е. присвоить ей **undef**), достаточно написать **set** (V).
Отметим также то, что при присваивании списков все присваивания элементов выполняются *синхронно*. Другими словами, это значит, что

```
[x y] = [y x];
```

— это допустимый способ поменять местами значения переменных x и y, который легко обобщить и на большее количество переменных. (Однако, для обмена местами двух значений удобнее использовать **swap**, описанный ниже).
Наконец, поскольку присваивание является операцией, оно возвращает значение (то самое, которое было присвоено). Его можно использовать в любом выражении (при этом обычно требуются скобки, т. к. присваивания имеют самый низкий приоритет):

```
X = (Y = 15) + (Z = 25);

(X, Y, Z);
```

```
(X, Y, Z) => (40, 15, 25);
```

Латентное присваивание

Операция *латентного присваивания* (которую реализует встроенный функтор **let**) интерпретирует свои операнды так же, как и **set** (правый операнд может быть произвольным выражением, левый должен быть мутабельным). Принципиальная разница состоит в том, что правый операнд *не вычисляется*, а непосредственно присваивается левому. Например:

```
S := (a + b)*2;
T := (a*b, b*c, c*a);

S; T;
```

```
S => mul:(add:(a, b), 2);
T => (mul:(a, b), mul:(b, c), mul:(c, a));
```

Здесь значениями переменных S и T станут целые *выражения* (как говорилось выше, в AWL они являются такими же структурами данных, как и все прочие), вместо результатов их вычислений. В частности, совершенно неважно, какие имеют значения (и имеют ли вообще) переменные a, b, c в момент присваивания. Это существенно лишь тогда, когда произойдет вычисление значений S и T. (Для вычисления подобного рода выражений нужна специальная операция ^.) Например:

```
[a b c] = [2 3 5];

^S; ^T;
```

```
reval:S => 10;
reval:T => (6, 15, 10);
```

```
[a b c] = [6 10 13];

^S; ^T;
```

```
reval:S => 32;
reval:T => (60, 130, 78);
```

Латентное присваивание является мощным механизмом, аналоги которого в процедурных языках программирования встречаются редко. Отчасти, можно рассматривать его как разновидность макроподстановки (однако, в отличие, например, от препроцессора в C, подстановка здесь не просто текстуальная, а семантическая). Также можно рассматривать "латентные" выражения как простейший вид определяемых пользователем функций (без параметров и локальных переменных).
Но, как и всякий мощный механизм, непрямое присваивание следует применять к осторожностью. В частности, нельзя допускать *зацикливания ссылок* (т. е. опасно, когда правый операнд содержит прямые или косвенные ссылки на левый). Если написать что-нибудь вроде:

```
a := ^a + 1
```

то, попытка вычислить значение ^a приведет к рекурсивному циклу с непредсказуемыми последствиями.

Операции отложенного и немедленного вычисления

Более общей является ситуация, когда надо обратиться к выражению, не вычисляя его (т. е. просто взять ссылку на это выражение). Для этого есть отдельная унарная операция @ (ей соответствует функтор **deval**). Так, например, сама операция латентного присваивания (`A := B`) равносильна сочетанию просто присваивания с отложенных вычислением (`A = @B`). (Использовать ее в этом качестве не стоит, т. к. одна операция предпочтительнее двух.) Другой случай применения операции @ — возможность взглянуть на значение переменной или выражения, не вычисляя его — мы уже видели выше. Вот другой (тривиальный) пример:

```
@(a/2 + b/3 + c/4);
```

```
deval:add:(add:(div:(a, 2), div:(b, 3)), div:(c, 4)) =>
add:(add:(div:(a, 2), div:(b, 3)), div:(c, 4));
```

Операция явного вычисления ^ (функтор **reval**) имеет противоположный эффект: она предписывает вычислить операнд и вернуть результат вычисления. Несколько примеров ее применения мы уже видели в предыдущей главе.

Совмещенное присваивание

Во многих языках программирования (прежде всего, C-подобных) есть возможность совместить присваивание с выполнением некоторых бинарных операций. Есть такая возможность и в AWL (хотя используемый синтаксис несколько другой).
Вычисление выражения

```
MUTABLE = op : VALUE
```

(где **op** — символ бинарной операции) эквивалентно

```
MUTABLE = MUTABLE op VALUE
```

с той разницей, что выражение MUTABLE (оно, очевидно, должно быть мутабельным) вычисляется только один раз. В качестве операции **op** могут использоваться следующие операции: `+, -, *, /, %, %%, ?<, ?>, <<, >>, &, |, ~, +$, *$, ?<$, ?>$`. Например:

```
`(прибавить elem к sum_all)`
sum_all =+: elem;
`(умножить elem на prod_all)`
product_all =*: elem;
`(добавить elem в конец join_all с разделителем)`
join_all =+$: elem +$ ", ";
```

Помимо этого, присваивание также может быть совмещено с унарными операциями (`+, -, ~`). Выражение:

```
MUTABLE = : op
```

равносильно более развернутому

```
MUTABLE = op MUTABLE
```

при том, что MUTABLE вычисляется только один раз. Например:

```
val =:-   ` (инвертировать знак переменной val) `
```

Как и редукция, совмещенное присваивание является *мета-операцией,* и реализуется через обращение к встроенному функтору **comb**, выполняющему необходимые действия над своими операндами:

```
@( Mut =+: Value );
@( Mut =:~ );
```

```
deval:comb:add:(Mut, Value) => comb:add:(Mut, Value);
deval:comb:not:Mut => comb:not:Mut;
```

Инкремент и декремент

Операции *увеличения* и *уменьшения* числа на единицу требуются настолько часто, что для них (как и в C-подобных языках) предусмотрены отдельные унарные операции. Операции инкремента и декремента существуют как в префиксной, так и в постфиксной форме.

Синтаксис	*Нормальная форма*	*Семантика*
`++ A`	**inc**(A)	*Инкремент A (префиксный вариант)*
`-- A`	**dec**(A)	*Декремент A (префиксный вариант)*
`A ++`	**inc_p**(A)	*Инкремент A (постфиксный вариант)*
`A --`	**dec_p**(A)	*Декремент A (постфиксный вариант)*

Все эти операции требуют, чтобы операнд A был мутабелен, и имел значение числового типа. (Однако, в отличие от C, значение может быть и нецелым.) Операции инкремента

увеличивают значение операнда на единицу, декремента – уменьшают его на единицу. В качестве результата *префиксные* операции возвращают значение операнда после его модификации; *постфиксные* – до нее. Например:

```
a = 10;
```

```
set:(a, 10) => 10;
```

```
b0 = ++ a; b1 = -- a;
```

```
set:(b0, inc:a) => 11;
set:(b1, dec:a) => 10;
```

```
c0 = a ++; c1 = a --;
```

```
set:(c0, inc_p:a) => 10;
set:(c1, dec_p:a) => 11;
```

Операция обмена

Операция “обмена значений местами” требуется настолько часто, что для нее предусмотрен специальный функтор.

Обмен:

Синтаксис	*Нормальная форма*	*Семантика*
`V1 :=: V2`	**`swap`**`(V1, V2)`	*Поменять местами значения V1 и V2*

Выражения V1 и V2 должны быть мутабельными, при этом они оба вычисляются только один раз.

```
[s1 s2] = ['alpha' 'beta'];
```

```
set:((s1, s2), "alpha", "beta") => ("alpha", "beta");
```

```
s1 :=: s2;
```

```
swap:(s1, s2) => ();
```

```
[s1 s2];
```

```
(s1, s2) => ("beta", "alpha");
```

Управление выполнением программы

Конечно, операции, позволяющие организовывать условное или циклическое выполнение, присутствуют во всех современных языках программирования. Но специфика AWL заключается в том, что такие конструкции также являются встроенными функторами. В отличие от, скажем, скалярных операций, вычисление одного или нескольких операндов для этих функторов может быть и необязательным, и многократным.

Условные операции

В большинстве программ приходится принимать какие-то решения – и, в зависимости от этого, выполнять те или иные действия. В AWL условное выполнение реализуется через следующую группу операций (как всегда, записываемых или в нормальной, или в синтаксической форме).

Условные операции (тернарные):

Синтаксис	*Нормальная форма*	*Семантика*
`P ? V1 : V2`	**`if`**`(P,V1,V2)`	*Условное выполнение (прямое):* *если P то V1, иначе V2*
`P ~? V1 : V2`	**`unless`**`(P,V1,V2)`	*Условное выполнение (обратное):* *если не P то V1, иначе V2*

Обычно эти операции имеют три операнда. Первый операнд (P) вычисляется всегда. Его значение рассматривается как *условие* (вспомним, что специального логического типа данных в AWL нет): числовое значение 0, пустая строка или **undef** рассматриваются как *ложь*, все остальное – как *истина*. (Perl-программистам следует иметь в виду, что строка "0" в AWL является "истинной".)
Все дальнейшие действия определяются значением P и полярностью условной операции. В *прямой* форме (**if**), если условие P истинно, то вычисляется/выполняется операнд V1, иначе вычисляется/выполняется V2. *Обратная* форма (**unless**) зеркально симметрична: если P истинно, то вычисляется V2, иначе V1. В любом случае, вычисляется или выполняется *только один* из операндов V1 и V2, и его значение возвращается в качестве результата операции. Приведем примеры:

```
`Если v четно, то строка "чет", иначе "нечет"`
v & 1 ? "нечет" : "чет";

`Если x принадлежит отрезку, ограниченному a и b,
то строка "<>"; если x меньше отрезка, то "<", иначе ">"`
a < b ?
    (x < a ? "<" : x > b ? ">" : "<>")
    (x < b ? "<" : x > a ? ">" : "<>")
```

Третий операнд может быть опущен. В этом случае, если P ложно (в **if**-форме) или истинно (в **unless**-форме), ничего не делается (и возвращается значение **undef**).

Условно-логические операции

К группе условных операций очень близки операции условного "и" / условного "или".

Условно-логические операции (бинарные):

Синтаксис	*Нормальная форма*	*Семантика*
`P && Q`	**`c_and`**`(P,Q)`	*Условное логическое "и":* *если P то Q, иначе 0*
`P || Q`	**`c_or`**`(P,Q)`	*Условное логическое "или":* *если P то 1, иначе Q*

Эти операции отличаются от своих скалярных аналогов (**and** и **or**) тем, что безусловно вычисляется только *первый* операнд. Второй операнд вычисляется (и возвращается как значение) только если P истинно (для **c_and**) или ложно (для **c_or**). В противном случае, **c_and** возвращает 0 (*ложь*), **c_or** – 1 (*истину*). Иными словами, второй операнд вычисляется, только если без его вычисления окончательный результат неочевиден.
Заметим, что эти операции можно выразить через **if** и **unless** таким образом:

```
! c_and(P,Q) = if(P,Q,0);

! c_or(P,Q) = unless(P,Q,1);
```

Циклы с предусловием/постусловием

Конечно, не менее важную роль, чем условия, в программах играют *циклы*.
Чаще всего нужны циклы с проверкой условия (до или после выполнения тела цикла). Вот полный список встроенных циклических операций-функторов AWL:

Циклические операции с условием (бинарные):

Синтаксис	*Нормальная форма*	*Семантика*
`P ?? U`	**`while`**`(P,U)`	*Цикл с предусловием (прямым):* *пока P, выполнять U*
`P ~?? U`	**`until`**`(P,U)`	*Цикл с предусловием (обратным):* *пока не P, выполнять U*
`?? U P`	**`do_while`**`(P,U)`	*Цикл с постусловием (прямым):* *выполнять U пока P*
`~?? U P`	**`do_until`**`(P,U)`	*Цикл с постусловием (обратным):* *выполнять U пока не P*

Как можно видеть, система циклических операций тоже имеет симметрию (двойную): операции различаются как *полярностью* условия (*пока истинно*/*пока ложно*), так и моментом его проверки (*перед* каждой итерацией, или *после* нее). Выражение P является *условием цикла* (истинность или ложность которого определена аналогично условным операциям), выражение U – *телом цикла*. Пока условие P истинно (для **while**-форм) или ложно (для **until**-форм), будет выполняться тело цикла U.
Все циклические выражения также возвращают значение: результат вычисления U на последней итерации, или **undef**, если U не было вычислено ни разу (что возможно только для формы с предусловием).
В заключение отметим, что при использовании нормальной формы можно опустить операнд U. В этом случае **while**/**do_while** просто вычисляют P до тех пор, пока оно не станет ложным (а **until**/**do_until** – пока оно не станет истинным). В этом случае никакого различия между формами с предусловием и постусловием не существует; и возвращается значение **undef**.

Циклы с параметром

Не намного реже возникает необходимость в другой разновидности циклов – с *параметром*, изменяющимся с постоянным шагом на каждой итерации (аналогичные циклам FOR в Паскале, Модуле или Обероне). В AWL есть встроенные итераторы для реализации таких циклов (однако, они требуют, чтобы параметр цикла был целым, и менялся с шагом 1 или -1). Эти итераторы не имеют специального синтаксиса.

Циклические операции с параметром (тернарные):

Нормальная форма	*Семантика*
`for_inc``(V,R,W)`	*Инкрементный цикл с параметром:* *для всех V в диапазоне R (по возрастанию)* *выполнить W.*
`for_dec``(V,R,W)`	*Декрементный цикл с параметром:* *для всех V в диапазоне R (по убыванию)* *выполнить W.*

Итак, циклы с параметром имеют две формы: *инкрементную* и *декрементную*. Независимо от формы, у них имеется три параметра:
- переменная (в общем случае любое мутабельное выражение) V, задающее параметр цикла;
- диапазон R, задающий значения, который пробегает параметр V;
- тело цикла W, выполняющееся на каждой итерации.

Семантика диапазона полностью совпадает с той, которая уже рассматривалась в разделе об отрезках строк. Напомним: диапазон обычно является списком из двух чисел, задающих начальное значение (*включенное* в диапазон) и конечное значение (*не включенное* в него). Обычно диапазон задается в виде From .. To. Если в качестве диапазона задано единственное число N, то диапазон считается эквивалентным 0 .. N. В зависимости от формы, параметр цикла меняет свое значение от первого элемента к последнему с шагом 1 (**for_inc**), или от последнего к первому с шагом -1 (**for_dec**). Для каждого значения параметра цикла выполняется его тело. Например, для

```
for_inc (I, 10..15, W);
```

выражение W будет выполнено для значений `I = 10, 11, 12, 13, 14`. Для выражения:

```
for_dec (J, 50..100, W);
```

W будет выполняться для значений `J = 99, 98, 97 ... 52, 51, 50`.
По завершении выполнения цикла с параметром, возвращается последнее вычисленное значение W, а параметр цикла сохраняет последнее значение, при котором тело цикла было выполнено (в приведенном случае `I == 14`, `J == 50`). Естественно, если диапазон является пустым (`From >= To`), то тело цикла не будет выполнено ни разу, а в качестве значения возвращается **undef**.
Заметим, что параметру цикла можно присваивать любые значения в его теле, но это никак не повлияет на его выполнение (при начале каждой итерации цикла, ему все равно будет присвоено предыдущее/следующее значение диапазона). Также никакого влияния не оказывают изменения выражений, определяющих диапазон (т. к. он вычисляется только один раз, перед началом выполнения цикла).
В завершение заметим, что иногда нужно выполнить определенный код заданное количество раз, и при этом номер итерации знать не требуется. Это удобно описывать с помощью отдельного итератора **times**:

Нормальная форма	***Семантика***
`times``(N,W)`	*Выполнить W ровно N раз.*

Тело цикла (выражение W) выполняется ровно N раз (или ни разу, если `N <= 0`). Как и для всех итераторов, в качестве значения возвращается последнее вычисленное значение W (**undef**, если W не было вычислено ни разу).

Ввод и вывод

До сих пор, единственным вводом и выводом, с которыми мы имели дело, был ввод AWL-выражений в интерпретатор, и вывод (трассировочный) результатов их вычисления.
Конечно, в AWL предусмотрены функторы, обеспечивающие явный обмен информацией с внешней средой в процессе выполнения программы. Они оперируют понятием *потоков*, которые являются еще одним встроенным в AWL стандартным типом данных. Три стандартных потока, всегда доступных в AWL-среде – это ввод, вывод и вывод ошибок.

Далее приведена сводка операций ввода-вывода.

Операции ввода/вывода:

Синтаксис	*Нормальная форма*	*Семантика*
`OUT <: V_LIST`	**`f_put`**`(OUT,V_LIST)`	*Вывести значения из списка V_LIST в выходной поток OUT*
`<: V_LIST`	**`f_put`**`((),V_LIST)`	*Вывести значения из списка V_LIST в стандартный вывод*
`IN :> M_LIST`	**`f_get`**`(IN,M_LIST)`	*Ввести значения из входного потока IN в список (мутабельный) M_LIST*
`:> M_LIST`	**`f_get`**`((),M_LIST)`	*Ввести значения из стандартного ввода в список (мутабельный) M_LIST*

Вывод данных

Основным средством вывода данных является функтор **f_put**. Выполнение выражения **`f_put`** `(Output, Expr)` (в операторной форме `Output <: Expr`) заключается в том, что выражение Expr (обычно это список, но может быть и скаляр) вычисляется, и его значение (или все значения последовательно) выводятся в выходной поток, заданный Output. Если выводится набор значений, они выводятся подряд в заданном порядке без каких-либо разделителей или ограничителей. Если значением Output является **undef**, для вывода используется стандартный вывод. Приведем простой пример:

```
a = b = 2;
<: (a, '*', b, '=', a*b, '\n');
```

```
2*2=4
```

Значением, возвращаемым **f_put**, является количество успешно выведенных операндов.

Ввод данных

Ввод данных является обратной операцией по отношению к выводу. Выполнение функтора **`f_get`** `(Input, Expr)` (в операторной форме `Input :> Expr`) последовательно читает из входного потока Input строку (N строк, если Expr является списком из N выражений), и присваивает прочитанное Expr (которое, очевидно, должно быть мутабельным). Таким образом, f_get всегда возвращает результат(ы) в виде строк (если вам нужны, например, числовые скаляры, преобразуйте их самостоятельно). Если значением Input является **undef**, для ввода используется стандартный ввод.

Строки, прочитанные с помощью **f_get**, *не содержат* символа конца строки (в отличие, например, от аналогичных операций Perl). В качестве значения **f_get** возвращает количество успешно прочитанных аргументов. Причиной неуспеха обычно является достижение конца файла (т. е. если вызов **f_get** возвращает 0, это обычно означает, что конец файла достигнут).

Приведем несколько примеров совместной работы операций ввода и вывода. Вот тривиальный код, копирующий стандартный ввод на стандартный вывод (аналогично UNIX утилите cat). (Этот пример не 100% корректен: если последняя строка файла не завершена, при выводе она будет дополнена символом '\n').

```
(>: inline) ?? (<: (inline, '\n'));
```

Вот несколько более интересный пример: он читает строки из стандартного ввода, накапливая их в массиве lines_list, и затем выводит их в обратном порядке (в этом примере

используются списковые операции, описанные подробно в следующей главе).

```
(>: inline)?? (lines_list [<-] inline);
n = 0;
(n ~= #lines_list)?? (<: (lines_list[n++], '\n'));
```

Стандартные потоки ввода/вывода

Теоретически, потоки ввода/вывода могут быть связаны с любыми файлами, или другими системными ресурсами (хотя сейчас эти средства не разработаны). Однако, три потока, предопределенных в AWL-системе, можно задать в явном виде:

```
f_in ();        `стандартный ввод`
f_out ();       `стандартный вывод`
f_err ();       `стандартный вывод ошибок`
```

Заметьте, что эти функторы не принимают никаких аргументов, и единственной их задачей является обеспечение явного доступа к предопределенным системным потокам. Вот так, например, можно вывести сообщение в стандартный поток ошибок:

```
f_err() <: ('Ошибка: значение i (', i, ') вне диапазона!');
```

Операции над списками

Как уже говорилось выше, списки AWL являются встроенной в язык структурой данных. В частности, язык предоставляет множество встроенных операций-функторов, позволяющих работать со списками – как поэлементно, так и как с единым целым.
Ниже приведена сводная таблица списковых операций, детальное описание которых дано в следующих главах. В таблице L и M обозначают произвольные списки (однако, частным случаем списка может быть скаляр или даже **undef** – об этом подробно далее).

Списковые операции:

Синтаксис	*Норм. форма*	*Приоритет*	*Семантика*
`# L`	**l_len**(L)	унарная	*Длина списка L*
`[~] L`	**l_rev**(L)	унарная	*Инверсия списка L*
	l_ref(L)		*Ссылка на список L*
`L [+] M`	**l_cat**(L, M)	аддитивная	*Конкатенация списков L и M*
`L [*] N`	**l_rep**(N, L)	мультипликативная	*Репликация (повторение) N раз подряд списка L*
`[<] L`	**l_head**(L)	унарная	*“Голова” списка L*
`L [>]`	**l_tail**(L)	унарная	*“Хвост” списка L*
	l_head_by(N, L)		*Повторить l_head N раз*
	l_tail_by(N, L)		*Повторить l_tail N раз*
`L [N]`	**l_item**(N, L)	унарная	*Элемент списка L с индексом N*
`L [<-] Q`	**l_push**(L, Q)	присваивания	*Добавить элементы из Q в начало списка L*
`L [->] Q`	**l_pop**(L, Q)	присваивания	*Извлечь элементы из начала L в Q*

В синтаксическая форме большинство списковых операций содержат квадратные скобки, что не дает спутать их с числовыми операциями.

Длина списка

Унарный функтор **l_len** всегда возвращает целое число: количество элементов в своем операнде-списке. Для любого нетривиального операнда-списка результат всегда больше 1. При применении к скаляру эта операция возвращает 1, при применении к **undef** – 0.

```
L_a = [10 20 30 40 50];
L_b = ['север' 'запад' 'юг' 'восток'];
```

```
set:(L_a, 10, 20, 30, 40, 50) => (10, 20, 30, 40, 50);
set:(L_b, "север", "запад", "юг", "восток") => ("север",
"запад", "юг", "восток");
```

```
#L_a;
#L_b;
```

```
l_len:L_a => 5;
l_len:L_b => 4;
```

Как и все прочие списковые операции языка, эта операция демонстрирует некую общность свойств списков и скаляров (скаляр эквивалентен списку длины 1, а **undef** - списку длины 0. Можно сказать, что **l_len** возвращает общее количество “значений”, “содержащихся” в элементе данных AWL (невзирая на то, список это, скаляр или неопределенность).

Списки как последовательности

Бинарный функтор **l_item** позволяет получить доступ к произвольному элементу списка. Особенность этого функтора состоит к том, что он является *мутабельным*, т. е. обычно возвращает мутабельный результат. Иными словами, его можно использовать в контексте, где требуется мутабельное выражение (например, слева от любого из операторов присваивания).
Первый операнд функтора является *индексом элемента*, второй – собственно списком (заметим, что в операторной форме порядок операндов является обратным). Индекс вычисляется как целое число (с обычными для скалярных типов приведениями). Интерпретируется он так же, как и в C-подобных языках (т. е. первый элемент списка L – это L[0], а последний элемент – L[#L-1]). Если же значение индекса некорректно (< 0 или >= #L), возвращается **undef**. Так, для списков из предыдущей главы справедливо следующее:

```
L_a[2];
```

```
l_item:(2, L_a) => 30;
```

```
L_b[3];
```

```
l_item:(3, L_b) => "восток";
```

```
++ L_a[1];
```

```
inc:l_item:(1, L_a) => 21;
```

```
L_a;
```

```
L_a => (10, 21, 30, 40, 50);
```

```
L_b[0] = 'north';
```

```
set:(l_item:(0, L_b), "north") => "north";
```

```
L_b;
```

```
L_b => ("north", "запад", "юг", "восток");
```

В отличие от других языков (например Perl), эта операция *не может* неявно расширять список – присваивание элементу, которого не существует, в общем случае является ошибкой. Ради общности, о которой шла речь выше, операция применима и к скалярам: для любого скаляра S выражение S[0] возвращает его (мутабельное) значение (а любые другие значения индекса – **undef**).

Списки как бинарные деревья

Как и в LISP-подобных языках, каждый список фактически является рекурсивной структурой данных – *бинарным деревом*. Другими словами, список состоит из двух компонент (“головы” и “хвоста”), причем каждой из этих компонент, в свою очередь, может быть список произвольной структуры. Так, при конструировании списка в простейшей форме – (A, B) – создается список, головой которого является A, а хвостом – B. Если в конструкторе списка более двух элементов, список строится рекурсивно: так, список (A, B, C, D) равносилен списку (A, (B, (C, D))). Такой подход к спискам альтернативен рассмотрению списка как линейного массива, и иногда открывает новые возможности для работы с ними.

Есть операции, явно позволяющие работать со списками как с бинарными деревьями. Операция **l_head**(L) предоставляет доступ к “голове” списка L (фактически, она эквивалентна L[0], и введена только для симметрии). Операция **l_tail**(L) предоставляет доступ к “хвосту” списка L (т. е. всем его элементам, кроме первого). Эти операции аналогичны car/cdr в Lisp/Scheme, или hd/tl в ML. Обе операции также могут быть записаны в синтаксической форме: **l_head**(L) как [<] L, а **l_tail**(L) – как L [>]. (Если вы представите список в виде бинарного дерева - корень сверху, а потомки – снизу/слева и снизу/справа - то смысл этой мнемоники становится прозрачен.)

Существуют также модификации этих операций, первым операндом которых является *повторитель* (вычисляемый как целое). Операция **l_head_by**(N,L) равносильна **l_head** (**l_head**(... **l_head**(L))), где **l_head** повторяется N раз. Совершенно аналогично, **l_tail_by**(N,L) равносильно **l_tail**(**l_tail**(... **l_tail**(L))), где **l_tail** повторяется N раз. Очевидно, что если N == 1, эти операции эквивалентны **l_head**(L) и **l_tail**(L); если же N <= 0, обе операции просто возвращают L. Эти операции позволяют эффективно работать с глубоко вложенными списками.

Как всегда, приведем несколько примеров:

```
[<] L_a;
L_a [>];
```

```
l_head:L_a => 10;
l_tail:L_a => (21, 30, 40, 50);
```

```
[<] L_b;
L_b [>];
```

```
l_head:L_b => "north";
l_tail:L_b => ("запад", "юг", "восток");
```

```
l_tail_by(3, L_a);
l_tail_by(2, L_b);
```

```
l_tail_by:(3, L_a) => (40, 50);
l_tail_by:(2, L_b) => ("юг", "восток");
```

Немаловажно то, что все эти операции также *мутабельны*, т. е. обращение к ним выдает мутабельный результат. В частности, присвоив вызову **l_tail** любой список, можно сделать его новым "хвостом" списка-операнда (старый хвост при этом может быть потерян!)

```
L_b [>] = ['west' 'south' 'east'];
```

```
set:(l_tail:L_b, "west", "south", "east") => ("west",
"south", "east");
```

```
L_b;
```

```
L_b => ("north", "west", "south", "east");
```

Как и следует ожидать, применение **l_head** к скаляру возвращает его самого, а применение к скаляру **l_tail** всегда возвращает **undef** (т. к. у скаляров, в отличие от списков, нет "хвоста").

Вставка и удаление элементов списка

Операции **l_push** и **l_pop** позволяют работать со списком как со *стеком*: вставлять элементы в его начало или удалять его начальные элементы. В принципе, эти операции избыточны (т. к. их можно заменить разными комбинациями списковых присваиваний), но их использование часто намного эффективнее и нагляднее.
Операция **l_push** (L, a, b, c, ...) (в синтаксической форме: L [<-] (a, b, c, ...)) добавляет в начало списка L результаты вычисления выражений a, b, c и т. д. Заметим, что порядок, в котором они окажутся в L, *обратен* порядку, в котором они идут в списке операндов. Естественно, выражение L должно быть мутабельно. Если прежним значением L был скаляр, он окажется в конце списка; если L имело значение **undef**, новым значением L просто станет (..., c, b, a). Например (если aa ранее не было определено):

```
aa [<-] [1 2 3];
aa;
```

```
l_push:(aa, 1, 2, 3) => (3, 2, 1);
aa => (3, 2, 1);
```

```
aa [<-] ['xx' 'yy'];
aa;
```

```
l_push:(aa, "xx", "yy") => ("yy", "xx", 3, 2, 1);
aa => ("yy", "xx", 3, 2, 1);
```

Операция **l_pop** (L, ra, rb, rc, ...) (в синтаксической форме: L [->] (ra, rb, rc, ...)) выполняет обратную функцию: она извлекает из списка L элементы, последовательно присваивая их своим операндам ra, rb, rc, ... Естественно, эта операция требует, чтобы они, как и L, были мутабельны. Если после **l_pop** в L остается один элемент, L становится скаляром; если же из L извлекаются все элементы, L становится равным **undef**. Вот и примеры:

```
aa [->] I;
I; aa;
```

```
l_pop:(aa, I) => ("xx", 3, 2, 1);
```

```
I => "yy";
aa => ("xx", 3, 2, 1);
```

```
aa [->] [J K];
[J K]; aa;
```

```
l_pop:(aa, J, K) => (2, 1);
(J, K) => ("xx", 3);
aa => (2, 1);
```

```
aa [->] [L M];
[L M]; aa;
```

```
l_pop:(aa, L, M) => ();
(L, M) => (2, 1);
aa => ();
```

Нетрудно использовать эти операции для вставки/удаления элементов не в начале, а в произвольных позициях списка, если использовать в качестве первого операнда обращение к **l_tail/l_tail_by**. Например (для списка L_b из предыдущей главы):

```
L_b [<-] "northeast";
l_tail_by (2, L_b) [<-] "northwest";
l_tail_by (4, L_b) [<-] "southwest";
l_tail_by (6, L_b) [<-] "southeast";

L_b;
```

```
L_b => ("northeast", "north", "northwest", "west",
"southwest", "south", "southeast", "east");
```

Конкатенация списков

Бинарная операция **l_cat**(L, M) (в синтаксической форме: `L [+] M`) *конкатенирует* (последовательно сцепляет) два своих списка-операнда и возвращает список-результат. Например:

```
L1 = [11 22 33 44];
L2 = ['aaa' 'bbb' 'ccc'];
```

```
set:(L1, 11, 22, 33, 44) => (11, 22, 33, 44);
set:(L2, "aaa", "bbb", "ccc") => ("aaa", "bbb", "ccc");
```

```
L1 [+] L2;
L2 [+] L1;
```

```
l_cat:(L1, L2) => (11, 22, 33, 44, "aaa", "bbb", "ccc");
l_cat:(L2, L1) => ("aaa", "bbb", "ccc", 11, 22, 33, 44);
```

Любой из операндов может быть скаляром, и даже **undef** (очевидно, что конкатенация любого операнда с **undef** возвращает его самого). В общем случае, для конкатенации списков характерно простое тождество: `#(L [+] M) == #L + #M`.
Наверное, не надо подчеркивать различие между конкатенацией списков и применением конструктора списка (только когда L является скаляром, выражения L [+] M и (L, M) выдают одинаковый результат).

Репликация списка

Бинарная операция **l_rep**(N, L) (в синтаксической форме: `L [*] N` - учтите обратный порядок операндов) *реплицирует* (последовательно повторяет) ровно N раз свой список-операнд L, и возвращает список-результат. (Очевидно, что операнд N предварительно вычисляется как целое число.) Например:

```
L1 [*] 2;
L2 [*] 3;
```

```
l_rep:(2, L1) => (11, 22, 33, 44, 11, 22, 33, 44);
l_rep:(3, L2) => ("aaa", "bbb", "ccc", "aaa", "bbb", "ccc",
"aaa", "bbb", "ccc");
```

Операнд L может быть скаляром (в этом случае результат будет содержать N копий этого скаляра) или **undef** (результат, очевидно, тоже будет **undef**). Если N == 1, возвращается копия L; если N <= 0, также возвращается **undef**. Для списковой репликации также верно тождество: `#(L [*] N) == #L * N`.

Инверсия списка

Унарная операция **l_rev**(L) (в синтаксической форме: `[~] L`) возвращает результатом *инверсию* своего операнда, т. е. список, содержащий элементы операнда в противоположном порядке. Например:

```
[~] L1;
[~] L2;
[~] 10;
```

```
l_rev:L1 => (44, 33, 22, 11);
l_rev:L2 => ("ccc", "bbb", "aaa");
l_rev:10 => 10;
```

Заметим, что инвертирование скалярного значения всегда возвращает его самого (а инвертирование **undef**, очевидно, возвращает **undef**). Для инверсии списков всегда справедливо тождество: `#[~]L == #L`. Вполне очевидные тождества также связывают инверсию с конкатенацией (`[~](L [+] M) [==] ([~]M) [+] ([~]L)`) и репликацией (`[~](L [*] N) [==] ([~]L) [*] N`).

Списковый итератор

Очень часто необходимо выполнить некоторую операцию последовательно над каждым из элементов списка. Для этой цели есть специальные встроенные функторы – итераторы **l_loop** и **l_loop_r**. Специальная синтаксическая запись для них не предусмотрена.
Операция **l_loop** тернарна, т. е. имеет три ожидаемых аргумента. Вызов **l_loop** (V, L, X) выполняется следующим образом: для каждого элемента списка L его значение присваивается мутабельному V (переменной цикла), после чего вычисляется/выполняется выражение X (тело цикла). Очевидно, что X всегда будет выполнено #L раз. Результатом вычисления **l_loop** является результат вычисления X на последней итерации. Например:

```
L1;
```

```
L1 => (11, 22, 33, 44);
```

```
l_loop (l, L1, <: ['<' l '>\n']);
```

```
<11>
```

```
<22>
<33>
<44>

l_loop:(l, L1, f_put:((), "<", l, ">
")) => 3;
```

Операция **l_loop_r** отличается от **l_loop** только тем, что проход по списку происходит в обратном направлении: от последнего элемента к первому. Например:

```
L2;
```

```
L2 => ("aaa", "bbb", "ccc");
```

```
l_loop_r (l, L2, <: ['<' l '>\n']);
```

```
<ccc>
<bbb>
<aaa>

l_loop_r:(l, L2, f_put:((), "<", l, ">
")) => 3;
```

Ссылка на список

При присваивании списков обычно происходит копирование операндов – т. е. создается *новый* список, содержащий копию присваиваемого. Иногда, однако, необходимо присвоить не копию списка, а ссылку на него – возможность для этого дает функтор **l_ref**. Например:

```
Lx = [100 200 300];
Ly = l_ref(Lx);
```

```
set:(Lx, 100, 200, 300) => (100, 200, 300);
set:(Ly, l_ref:Lx) => (100, 200, 300);
```

Теперь переменные Lx и Ly физически ссылаются на один и тот же список. Любые внесенные в него изменения будут видны через любую из ссылок – например:

```
++ Ly[1];
Lx;
Ly;
```

```
inc:l_item:(1, Ly) => 201;
Lx => (100, 201, 300);
Ly => (100, 201, 300);
```

Применение **l_ref** позволяет создавать сложные структуры данных, имеющие общие элементы-подсписки. Конечно, **l_ref** тоже следует применять с осторожностью. С его помощью очень легко создать, например, список, замкнутый в кольцо. Результат выполнения большинства встроенных операций над таким списком непредсказуем.

Пользовательские функторы

Все функторы, с которыми мы имели дело выше, были *предопределенными* и встроенными в систему. Но пользователь легко может определить свои собственные функторы, работа с которыми в большинстве случаев не сложнее, чем с предопределенными.

Описание пользовательского функтора

Описание (декларация) пользовательского функтора имеет следующий общий вид:

```
! functor (par1 par2 par3 … parN) : [loc1 loc2 loc3 … locN] =
functor_body
```

Здесь:

functor — идентификатор описываемого функтора;
par1 … parN — идентификаторы *параметров* функтора;
loc1 … locN — идентификаторы *локальных переменных* функтора;
functor_body — тело функтора (произвольное AWL-выражение).

Обязательными в этом описании являются: символ '!', имя функтора, символ '=' и тело функтора. Оба списка – параметров и локальных переменных – являются необязательными и могут быть опущены.

Декларация функтора создает свою собственную область видимости (*пространство имен*). В ней локализуются его параметры (*par1 … parN*) и локальные переменные (*loc1 … locN*). По умолчанию они невидимы вне функтора, но доступны в его теле. В своей области видимости локальные переменные временно замещают одноименные внешние (если такие имеются). Любая переменная, не декларированная в этих списках, рассматривается как глобальная.

Приведем примеры описаний функторов:

```
` Диагональ параллелепипеда (со сторонами x,y,z) `
! Diag(x y z) = sqr(x*x + y*y + z*z);

` Факториал N (итеративное вычисление) `
! Fact(N) : [F] = { F = 1; N ?? (F =*: N --); F };

` Сцепление двух строк (в скобках через ':') `
! Join2 (s1 s2) = '(' +$ s1 +$ ':' +$ s2 +$ ')';
```

Как результат обработки перечисленных описаний, интерпретатор выдаст следующее:

```
! Diag:(x y z) [3] = sqr:add:(add:(mul:(x, x), mul:(y, y)),
mul:(z, z)) => Diag:;

! Fact:(N) [2] = {
set:(F, 1);
while:(N, comb:mul:(F, dec_p:N));
F
} => Fact:;

! Join2:(s1 s2) [2] = s_cat:(s_cat:(s_cat:(s_cat:("(", s1),
":"), s2), ")") => Join2:;
```

Заметьте, что при выводе каждый функтор сопровождается числом в квадратных скобках, указывающим размер его *локального контекста* (сумма количества его параметров и локальных переменных). Легко видеть, что каждое описание функтора также возвращает значение - *ссылку* на новый функтор (о которой в подробностях далее).

Обращение к функтору

Использовать определенные пользователем функторы так же просто, как и встроенные:

```
Diag(10, 20, 30);
```

```
Diag:(10, 20, 30) => 37.416574;
```

```
Fact(6);
```

```
Fact:6 => 720;
```

```
Join2 ('Hello', 'World');
```

```
Join2:("Hello", "World") => "(Hello:World)";
```

Если интересны технические детали, то при вызове пользовательского функтора выполняются следующие действия:

- создается новый локальный контекст (и с ним связываются параметры и локальные переменные функтора);
- параметры функтора инициализируются списком аргументов, заданных в его вызове. Инициализация может происходить по-разному (подробнее об этом в следующей главе). Все неинициализированные параметры и локальные переменные получат значение **undef**;
- вычисляется/выполняется тело функтора;
- уничтожается локальный контекст вместе со всеми переменными/параметрами;
- наконец, вычисленное значение тела функтора возвращается в качестве результата.

Вот еще несколько примеров полезных функторов:

```
` Среднее арифметическое чисел в списке L `
! avg_arithm (L):[S V] = {
    S = 0;
    l_loop (V, L, S =+: V);
    S / #L
    };
```

```
! avg_arithm:(L) [3] = {
set:(S, 0);
l_loop:(V, L, comb:add:(S, V));
div:(S, l_len:L)
} => avg_arithm:;
```

```
` Среднее геометрическое чисел в списке L `
! avg_geom (L):[P V] = {
    P = 1;
    l_loop (V, L, P =*: V);
    exp_by (P, 1/#L)
    };
```

```
! avg_geom:(L) [3] = {
set:(P, 1);
l_loop:(V, L, comb:mul:(P, V));
exp_by:(P, div:(1, l_len:L))
} => avg_geom:;
```

```
` Строковое сцепление всех элементов в списке L
  (со вставкой между ними разделителей S) `
! l_join (L S):[R V] = {
    R = string(l_head(L));
    l_loop (V, l_tail(L), R =+$: (S +$ V));
    R
    };
```

```
! l_join:(L S) [4] = {
set:(R, string:l_head:L);
```

```
l_loop:(V, l_tail:L, comb:s_cat:(R, s_cat:(S, V)));
R
} => l_join:;
```

Передача параметров

Вызов функтора является термом, и фактически состоит из префикса (имени вызываемого функтора) и его аргумента (фактически, обычно это *список* аргументов). Между списком параметров функтора и списком аргументов соответствие устанавливается поэлементно.
Возможна ситуация, когда длины этих списков *не совпадают* – в этом случае семантика передачи параметров аналогична той, что имеет место при присваивании списков. А именно: если список аргументов функтора *длиннее* списка параметров, последний параметр получит своим значением весь список оставшихся аргументов; если же он *короче*, все неинициализированные параметры сохранят значение **undef**.
Для каждого из параметров передача осуществляется по значению, исключая списки и более нетривиальные структуры данных, о которых речь пойдет позже (объекты, массивы, хэши), которые передаются по ссылке.
Возможен, однако, еще один вариант передачи параметров – *латентный*. Если в декларации функтора соответствующий параметр предваряет символ '@', то соответствующий ему аргумент передается *без вычисления* - аналогично тому, как это происходит при латентном присваивании с помощью **let**. Собственно вычисление аргумента в этом случае произойдет лишь в тот момент, когда в теле функтора потребуется его значение (причем, оно может необязательным или же многократным). В описании любого функтора допустима любая комбинация обычных и латентных параметров. Во многих встроенных функторах (таких как условные или итераторы) один или несколько аргументов вычисляются латентно – механизм латентных параметров позволяет и пользователю определять функторы, обладающие таким же свойством.
Предположим, что мы хотим описать функтор, семантика которого полностью аналогична встроенному итератору **while**, но порядок аргументов противоположен – сначала идет тело цикла, потом его предусловие. Создать такой функтор несложно:

```
` Цикл "while" наоборот `
! while_rev(@Body @Cond) = while(^Cond, ^Body);
```

```
! while_rev:(@Body @Cond) [2] = while:(reval:Cond,
reval:Body) => while_rev:;
```

и нетрудно убедиться, что он работает так, как ожидается:

```
i = 10;
while_rev (<: [i ' '], -- i);
```

```
set:(i, 10) => 10;
9 8 7 6 5 4 3 2 1 while_rev:(f_put:((), i, " "), dec:i) =>
2;
```

Вот более полезный пример. В AWL отсутствует точный аналог C/Java-подобного оператора for. Однако, если хочется иметь функтор-итератор с очень похожей функциональностью, его легко определить в одну строку:

```
` C-for - подобный итератор `
! c_for (@Init @Cond @Iter @Body) =
{^Init; ^Cond ?? {^Body; ^Iter}};
```

```
! c_for:(@Init @Cond @Iter @Body) [4] = {
```

```
reval:Init;
while:(reval:Cond, {
reval:Body;
reval:Iter
})
} => c_for:;
```

Использовать его так же просто, как for в C - только синтаксис вызова традиционен для функторов:

```
c_for (I = 1, I <= 5, ++ I, <: ['I = ' I '\n']);
```

```
I = 1
I = 2
I = 3
I = 4
I = 5
c_for:(set:(I, 1), le:(I, 5), inc:I, f_put:((), "I = ", I,
"
")) => 6;
```

Рекурсия

Пользовательские функторы могут быть *рекурсивными*, т. е. могут обращаться к самим себе. Для этого не требуется никаких специальных усилий – имя определяемого функтора *уже* доступно в его теле. Например, вот рекурсивное определение факториала:

```
! r_fact (N) = N ? N * r_fact (N-1) : 1;
r_fact (10);
```

```
! r_fact:(N) [1] = if:(N, mul:(N, .r_fact:sub:(N, 1)), 1)
=> r_fact:;
r_fact:10 => 3628800;
```

Понятно, что у каждого вызова рекурсивного функтора есть свой собственный контекст (набор аргументов и локальных переменных), независимый от других обращений к нему. Один рекурсивный вызов может порождать несколько других. Например, вот так можно вычислять биномиальные коэффициенты (число комбинаций из N по M):

```
! r_comb (N M) = (0 < M && M < N) ? r_comb (N-1, M) + r_comb
(N-1, M-1) : 1;
r_comb (8, 3);
```

```
! r_comb:(N M) [2] = if:(c_and:(lt:(0, M), lt:(M, N)), add:
(.r_comb:(sub:(N, 1), M), .r_comb:(sub:(N, 1), sub:(M,
1))), 1) => r_comb:;
r_comb:(8, 3) => 56;
```

Факториалы и биномиальные коэффициенты легко могут быть вычислены нерекурсивным путем, но есть ситуации, когда обойтись без рекурсии весьма затруднительно. Неплохой пример – *функция Аккермана*, определенная для двух неотрицательных аргументов (при вычислении которой даже для небольших значений аргументов происходит значительное число вложенных вызовов):

```
! ack (m n) = m ? (ack (m-1, n ? ack (m, n-1) : 1)) : n+1;
```

```
! ack:(m n) [2] = if:(m, .ack:(sub:(m, 1), if:(n, .ack:(m,
```

```
sub:(n, 1)), 1)), add:(n, 1)) => ack:;
```

```
ack (3, 5);
```

```
ack:(3, 5) => 253;
```

Семейства функторов

Также нередка ситуация, когда нужно создать несколько *взаимно-рекурсивных функторов*, т. е. таких, каждый из которых прямо или косвенно обращается к остальным. Здесь, однако, есть проблема: язык требует, чтобы декларация каждого функтора *предшествовала* его первому использованию, но в группе функторов, связанных взаимной рекурсией, хотя бы один должен быть вызван до того, как будет декларирован. Для решения этой проблемы есть специальный языковый механизм – семейства *функторов*.
Общий синтаксис описания семейства функторов таков:

```
! { functor1 functor2 functor3 … functorM } = {
     (par11 … par1N) : [loc11 … loc1N] = body1,
     (par21 … par2N) : [loc21 … loc2N] = body2,
     …
     (parM1 … parMN) : [locM1 … locMN] = bodyM
     }
```

Как легко видеть, синтаксис группового описания функторов является обобщением синтаксиса одиночного функтора, и состоит из двух частей: заголовочной (перечисляющей имена функторов) и собственно реализационной. Здесь:

functor1 … functorM — идентификаторы описываемых функторов;
parn1 … parnN и locn1 … locnN — списки *параметров и локальных переменных* для каждого из M описываемых функторов;
body1 … bodyM — тела для каждого из M функторов.

Обратите внимание на то, что списки имен функторов, и их реализаций заключены в фигурные скобки (причем имена разделяются пробелами, а реализации – запятыми).
Конечно, в семейство можно объединить описание любой группы функторов, не обязательно связанных взаимной рекурсией – но именно в последнем случае применение семейств наиболее оправдано. Приведем пример: семейства из двух функторов (**is_even**/**is_odd**), вычисляющих четность/нечетность произвольного целого числа (для чего каждый из них рекурсивно обращается к другому).

```
! { is_even is_odd } = {
     ` is_even => `
     (n) = abs(n) <= 1 ? n == 0 : is_odd  (n > 0 ? n-1 : n+1),

     ` is_odd => `
     (n) = abs(n) <= 1 ? n <> 0 : is_even (n > 0 ? n-1 : n+1)
     };
```

```
{
! is_even:(n) [1] = if:(le:(abs:n, 1), eq:(n, 0), .
is_odd:if:(gt:(n, 0), sub:(n, 1), add:(n, 1)));
! is_odd:(n) [1] = if:(le:(abs:n, 1), ne:(n, 0), .
is_even:if:(gt:(n, 0), sub:(n, 1), add:(n, 1)));
} => (is_even:, is_odd:);
```

```
for_inc (i, -3..4, <: (i, ': ', is_even(i), ', ', is_odd(i),
'\n'));
```

```
-3: 0, 1
-2: 1, 0
-1: 0, 1
0: 1, 0
1: 0, 1
2: 1, 0
3: 0, 1
for_inc:(i, (neg:3, 4), f_put:((), i, ": ", is_even:i, ",
", is_odd:i, "
")) => 6;
```

Конечно, такой способ проверки четности, мягко выражаясь, неэффективен. В реальной жизни лучше использовать что-нибудь вроде:

```
! is_even(n) = n %% 2 == 0;
! is_odd(n) = n %% 2 <> 0;
```

Локальные функторы

Описания функторов могут быть *вложенными*: в теле любого функтора может быть описано любое число других функторов. Как и в случае параметров и локальных переменных, все эти описания локализуются в данном функторе, т. е. недоступны вне него (если только не использовать *квалификаторы*, описанным в разделе о классах). При этом – как и для глобальных переменных и функторов – пространства имен для локальных переменных и для функторов являются *независимыми*: переменные и функторы могут иметь одинаковые имена, не конфликтуя друг с другом.

Каждому локальному функтору доступны все локальные переменные (как и другие локальные функторы, декларированные до него) того функтора, в который он вложен. (В этом отношении AWL похож на другие языки, в которых поддерживается локальность функций – например, Паскаль.) Заметим, что если внешний функтор является рекурсивным, и был вызван более одного раза, то доступны будут только переменные *последнего* (т. е. глубже всего вложенного) вызова.

На глубину вложенности функторов также не накладывается никаких ограничений. Язык поощряет использование локальных функторов: их локализация уменьшает загруженность таблиц имен, снижает риск конфликта между ними и часто позволяет экономить на количестве передаваемых аргументов.

Приведем достаточно простой пример локального описания функтора. Функтор permute работает как *генератор перестановок*: permute(N) находит все N! перестановок целых чисел от 0 до N-1 (и выводит каждую на консоль). Сам рекурсивный перебор перестановок выполняет внутренний функтор r_perm (заметьте, что ему доступны локальные переменные permute, такие, как N и perm).

```
` Перестановки чисел 0..N `
! permute (N):[perm] = {
` Текущая перестановка `
perm = 0 [*] N;

` Рекурсивный перебор `
! r_perm (n):[i] =
    n <> N ? {
          perm[n] = n;
          n ++;
          for_dec (i, 1..n, perm[i] :=: perm[i-1]);
          r_perm (n);
          for_inc (i, 1..n, {
               perm[i] :=: perm[i-1];
               r_perm (n)
               })
          }
     :
     ` Конец рекурсии: вывести перестановку `
    {
     <: '[ ';
     for_inc (i, N, <: (perm[i], ' '));
     <: ']\n';
     };

` Входная точка рекурсии `
r_perm (0);
};
```

Работает permute следующим образом:

```
permute (3);
```

```
[ 2 1 0 ]
[ 1 2 0 ]
[ 1 0 2 ]
[ 2 0 1 ]
[ 0 2 1 ]
[ 0 1 2 ]
permute:3 => ();
```

Ссылки на функторы и косвенные вызовы

Все встречавшиеся нам вызовы функторов были *прямыми* — они имели синтаксическую форму термов, т. е. имя вызываемого функтора в них всегда указывалось явно. Однако, может возникнуть потребность в обращении к функтору, который определяется только в процессе выполнения программы. Для этого в языке присутствует специальный тип данных – *ссылки на функторы*, которые дают возможность косвенного обращения к ним.
Значение ссылочного типа может ссылаться на произвольный функтор (встроенный, определенный пользователем или даже класс, о чем подробнее далее). Поскольку пространство имен функторов отделено от переменных, для ссылки на функтор требуется специальная нотация:

```
! functor
```

Это выражение возвращает значением ссылку на функтор functor (как и префиксы термов, имя functor относится к соответствующему пространству имен). С этой ссылкой можно

обращаться как с выражением любого другого типа (т. е. можно присваивать переменным, передавать другому функтору как параметр или возвращать оттуда как значение, и пр.) Но, прежде всего, ссылку на функтор можно *вызвать* с определенным набором аргументов, что делает встроенный функтор apply:

```
apply (f_expr, args)
```

вычисляет значение первого аргумента (f_expr), и (если оно является ссылкой на функтор), вызывает этот функтор, передавая ему аргумент(ы) args, и возвращает значение-результат. Проще говоря, apply выполняет *косвенный вызов* того функтора, на который ссылается f_expr. (Если значением f_expr не является ссылка на функтор, apply считается ошибкой.) Как обычно, для выражения **apply** (f_expr, args) предусмотрена более компактная синтаксическая запись:

```
f_expr ! args
```

Очевидно, что любой терм вида **functor**(args) можно записать (более вычурно и менее эффективно) в виде **apply**(!**functor**, args). Например, выражение **mul**(X, Y) делает то же, что и **apply**(!**mul**, X, Y), или (что то же самое) (!**mul**) ! (X, Y). Вот более полезный пример: выражение:

```
(!add, !sub, !mul, !div) [index] ! (A, B)
```

возвращает сумму, разность, произведение или частное аргументов A и B, в зависимости от значения index (предполагая, что последнее находится в диапазоне 0..4).
Определим функтор reduce, семантика которого аналогична встроенной операции [=]: он «сворачивает» список L путем последовательного применения функтора-аргумента op к каждой паре элементов – до тех пор, пока в результате не останется скаляр.

```
! reduce (op L) : [val i] = {
    val = L[0];
    for_inc (i, 1..#L, val = op ! (val, L[i]));
    val
    };
```

В отличие от операции [=], reduce может быть применен и к пользовательским функторам.

```
reduce (!add, [1 2 3 4 5]);
  reduce:(add:, 1, 2, 3, 4, 5) => 15;

reduce (!mul, [1 2 3 4 5]);
  reduce:(mul:, 1, 2, 3, 4, 5) => 120;
```

Анонимные функторы

Механизм ссылок на функторы открывает еще одну возможность: создания безымянных функторов непосредственно в тех местах, где требуется ссылка на них. Таким образом, если функтор требуется только раз, нет необходимости декларировать его отдельно и придумывать для для него уникальное имя. В языках типа LISP подобные конструкции называются лямбда-выражениями.
Анонимный функтор является выражением (принадлежащим к приоритетной группе унарных операций), имеющим следующий синтаксис:

```
! (par1 par2 … parN) : [loc1 loc2 … locN] = functor_body
```

Как можно видеть, описание анонимного функтора практически аналогично описанию обычного: очевидная разница состоит в том, что у анонимного функтора отсутствует имя, а его тело *должно* быть синтаксически замкнутым выражением. Приведем несколько примеров:

```
func_ref = !(x y) = (x*x - y*y);
```

```
set:(func_ref, ! (x y) [2] = sub:(mul:(x, x), mul:(y, y)))
=> ! (x y) [2] = sub:(mul:(x, x), mul:(y, y));
```

Значением переменной func_ref станет ссылка на анонимный функтор, возвращающий разность квадратов двух своих аргументов. Со значением func_ref допустимы те же операции, что и с любым значением функторного типа – например, вызов через apply:

```
func_ref ! (20, 10);
func_ref ! (5, 4);
```

```
apply:(func_ref, 20, 10) => 300;
apply:(func_ref, 5, 4) => 9;
```

Конечно, имя переменной func_ref принадлежит пространству имен переменных (а не функторов), и она в любой момент может стать ссылкой на другой функтор (или, вообще, любое другое выражение). В отличие от явно декларированных функторов (время жизни которых связано с их областью определения), анонимные функторы существуют до тех пор, пока активна хотя бы одна ссылка на них – если все ссылки исчерпаны, анонимный функтор уничтожается.

В языке имеется несколько встроенных функторов, предполагающих, что их операндами являются ссылки на другие функторы (безразлично, встроенные или пользовательские). Например, встроенный функтор **l_map** выполняет операцию “отображения” списка-операнда, определяемого функтором-операндом: вызов **l_map** (F, L) предполагает, что F вычисляется в ссылку на функтор, а L – в список. Результатом вычисления является *новый список* (L не изменяется!), состоящих из результатов применения F к L. (Иными словами, справедлив инвариант: `(l_map(F, L)) [I] == apply (F, L[I])`.) Например:

```
l_map (!sin, (0, pi(1/3), pi(1/2), pi(1)));
```

```
l_map:(sin:, 0, pi:div:(1, 3), pi:div:(1, 2), pi:1) => (0.,
0.8660254, 1., 1.2246064e-016);
```

```
l_map (!(x) = (3*x + 2), [7 2 6 5 9]);
```

```
l_map:(! (x) [1] = add:(mul:(3, x), 2), 7, 2, 6, 5, 9) =>
(23, 8, 20, 17, 29);
```

```
l_map (!(str) = ('<' +$ str +$ '>'), [10 'aa' 20 'bb' 30
'cc']);
```

```
l_map:(! (str) [1] = s_cat:(s_cat:("<", str), ">"), 10,
"aa", 20, "bb", 30, "cc") => ("<10>", "<aa>", "<20>",
"<bb>", "<30>", "<cc>");
```

Объекты и классы

В AWL присутствуют основные механизмы для объектно-ориентированного программирования. Так, в языке присутствует такой тип данных, как *объекты*. Каждый

объект является экземпляром некоторого класса, который должен быть предварительно декларирован. Поддерживается наследование и полиморфизм, посредством виртуальных функторов.
(Содержание этой главы является относительно новым дополнением к языку, и может измениться в будущем.)

Описание класса

Каждый класс должен быть предварительно декларирован. *Описание (декларация) класса* очень похоже на описание функтора (сходство отнюдь не случайно). В самом общем случае оно имеет следующий вид:

```
!! [superclass] class (par1 par2 par3 … parN): [memb1 memb2
memb3 … membN]
= constructor
~ destructor
# { virtual1 … virtualN }
{ decl1, decl2 … declN }
```

Декларация класса начинается с *двух* (в отличие от функтора) восклицательных знаков.
За ними следуют:

- необязательное *имя суперкласса* (**superclass**),
- *имя* определяемого класса (**class**),
- необязательный *список параметров класса* (*par1 … parN*),
- необязательный *список компонент класса* (*memb1 … membN*),
- необязательные *конструктор* (constructor) и *деструктор* (destructor),
- необязательный *список собственных деклараций* класса (decl1 … declN).

Рассмотрим элементы описания класса по порядку.
Имя класса (class) принадлежит к функторному пространству имен, и должно быть уникально в этом пространстве, т. е. не может совпадать с именем функтора или другого класса в той же области видимости (но никак не конфликтует с именами переменных). Само описание класса, как и описание функтора, создает собственную область видимости, вложенную в его окружение. Класс может иметь *суперкласс*, и в этом случае является его *наследником* (подробнее механизм наследования также объяснен ниже).
Список *компонент* класса содержит переменные (компоненты), принадлежащие каждому из объектов данного класса. Естественно, у каждого объекта данного класса будет собственный набор компонент. Абсолютно то же относится к *параметрам класса* — разница состоит лишь в том, что параметры класса дополнительно автоматически инициализируются списком аргументов при создании экземпляра класса (об этом подробнее ниже). Имена компонент и параметров принадлежат области видимости класса (т. е. не мешают внешним переменным с теми же именами, но могут временно замещать их). Синтаксис списка параметров практически ничем не отличается от аналогичного списка для функторов (допуская, в частности, латентную передачу и инициализаторы по умолчанию).
Дополнительно, класс может иметь *конструктор* и *деструктор*, обеспечивающие корректную инициализацию и уничтожение объектов данного класса (об этом также подробнее ниже).
Наконец, *собственные декларации* класса — это, чаще всего, функторы (но возможно и другие классы), локализованные в среде декларируемого класса. Заметим, что разделителем в этом списке является *запятая* (не точка с запятой, как обычно!).

Создание объектов класса

После того, как новый класс декларирован, создать его новый экземпляр (объект) очень

просто:

```
Object1 = class (arg1, arg2, … argN);
```

Практически, синтаксис создания объекта ничем не отличается от синтаксиса вызова функтора. При выполнении такого вызова происходит следующее: создается новый объект класса class, и все его компоненты-параметры (par1 … parN) инициализируются соответствующими аргументами (arg1 … argN). Происходит это абсолютно так же, как и при вызове функтора — в частности, аргументы могут передаваться как по значению (имеет место по умолчанию), так и латентно (если соответствующий параметр снабжен флажком @). Все компоненты класса, не инициализированные аргументами (и не имеющие инициализвторов по умолчанию), получат значение **undef**. После инициализации всех параметров, вызывается *конструктор*, если он присутствует (подробнее об этом позже). В завершение, ссылка на созданный объект возвращается в качестве значения выражения (в данном случае, она присваивается переменной Object1).
Таким образом, процесс создания объекта во многом похож на вызов функтора. Разница в том, что при вызове функтора он вычисляется и выдается результат; при «вызове» класса новый инициализированный объект выдается в качестве результата. (Можно даже сказать, что класс — это функтор с приостановленным выполнением.)
Как и вызов функтора, создание объекта может быть *непрямым*, т. е. через обращение к apply:

```
Object2 = (!class) ! (arg1, arg2, … argN);
```

Здесь Object2 инициализируется так же, как и Object1 выше. Очевидно, что в данном случае вместо (!class) может быть любое функторное выражение, задающее класс неявным образом, что позволяет создавать объекты неизвестного заранее класса.

Понятие текущего класса и квалификаторы

При работе с классами возникает проблема области видимости: поскольку все компоненты и внутренние декларации класса локализованы в нем, снаружи они по умолчанию недоступны. В общем случае, для того, чтобы обратиться к областям видимости классов (и не только классов), имеется специальная языковая конструкция (*квалификатор*):

```
Class !! Expr
```

Здесь **Class** — имя класса, *Expr* — выражение (которое должно быть синтаксически замкнутым). В остальном *Expr* может быть произвольным, но обычно оно имеет какие то ссылки на локальные декларации класса **Class** (его компоненты или локальные для него функторы). Результат — выражение *Expr*, интерпретированное в области видимости **Class**. Заметим, что эта языковая конструкция имеет эффект только при компиляции. (Именно поэтому мы говорим об «интерпретации» а не об «вычислении» операнда *Expr*.) Этой операции не соответствуют никакие средства периода выполнения (т. е. не существует никакой «функторной» записи для **Class** !! *Expr*). Хотя это очевидно, напомним, что в качестве *Expr* может быть целый блок — тогда все конструкции в этом блоке интерпретируются применительно к области видимости класса **Class**.
Внутри класса (в его собственных декларациях функторов и пр.), нет необходимости прибегать к такой записи: описываемый класс и так является текущим.
Наконец, квалификаторы могут быть вложены друг в друга. Можно даже написать что-нибудь вроде:

```
Class1 !! Class2 !! … ClassN !! Expr
```

что позволяет интерпретировать *Expr* в контексте всех классов **Class1** ... **ClassN**. При этом в

Expr будут доступны все, что декларировано в перечисленных классах (до тех пор, пока никакие имена не конфликтуют друг с другом). Если же конфликт имен возникает, он разрешается в пользу самых внутренних описаний: т. е. дублированные имена в **ClassN** переопределяют имена в **Class1**, но не наоборот.
Также заметим, что первым операндом этой операции может быть не только имя класса. Она равно применима к пользовательским функторам, и к встроенным пространствам имен (о них подробно далее). Так, с помощью такой записи можно обратиться к локальной переменной или параметру функтора за пределами области его видимости.

Текущий экземпляр класса и его переопределение

Одним из фундаментальных понятий AWL является понятие *текущего экземпляра* класса. Понятие это является довольно специфичным для AWL: в большинстве ОО-языков, таких, как C++ или Java, точных аналогов нет.
У каждого определенного пользователем класса имеется собственный *текущий экземпляр*, который является одним из его атрибутов, аналогично компонентам. После декларирования нового класса его текущий экземпляр *не определен*. Важно то, что в языке в принципе нет способа постоянно переопределить текущий экземпляр класса. Подобное может быть сделано только *временно*, с помощью операции “точка” (**with**). Синтаксис этой операции прост:

```
Object . Expr
```

или (в равносильной функторной записи):

```
with (Object, Expr)
```

Семантика этой операции немного сложнее. Если кратко, ее можно описать так: «*вычислить значение выражения Expr с Object в качестве текущего экземпляра его класса*». Если говорить более развернуто, то при выполнении этой операции выполняются следующие действия:

- Прежде всего, вычисляется значение Object;
- если результат является объектом некого класса (обозначим его как Class), то…
- ... *предыдущий текущий экземпляр* класса Class запоминается;
- ... объект Object становится *новым текущим экземпляром* класса Class;
- ... вычисляется *значение* Expr;
- ... восстанавливается *старый текущий экземпляр* класса Class;
- ... вычисленное значение Expr возвращается как значение выражения.

Итак, текущий экземпляр класса можно изменить только *временно*. Однако, здесь уместно вспомнить, что “нет ничего более постоянного, чем временное”: поскольку выражение Expr может быть *произвольным* (в т. ч. вызов функтора, или блок) можно выполнить произвольно большой объем работы с Object в качестве текущего экземпляра для Class. Наконец, крайне важно то, что механизм, реализуемый функтором **with**, полностью *реентрантен*, т. е. изменения текущего экземпляра (для любого класса или классов) могут быть вложенными на сколь угодно большую глубину.
Формально в AWL отсутствуют «методы» классов (как в C++ или Java), фактически они и не нужны. Каждый функтор, локальный для класса, практически работает как метод класса в этих языках. Принципиальная разница заключается в том, что в этих языках «текущий экземпляр» класса определен только в пределах метода (т. к. является неявным аргументом при вызове метода). В AWL это понятие *глобальное* (и с вызовом функторов, строго говоря, никак не связанное). Таким образом, в AWL можно легко создать метод (скажем **new_meth**) класса Class и вне его декларации (что невозможно в C++ или Java):

```
! new_meth (Params) = Class !! Body;
```

Здесь операция !! позволяет работать в Body со «внутренностями» класса Class так же просто, как и в собственных локальных функторах класса Class. Вызывать new_meth так же просто, как и любой локальный функтор из Class:

```
Object . new_meth (Args);
```

Таким образом, операцию “точка” можно использовать практически также, как и ее аналог в большинстве ОО-языков — но, на самом деле, ее семантика значительно шире. Как и в других языках, в AWL можно использовать эту операцию для обращения к компоненте:

```
` обращение к memb1 в объекте Object `
Object . Memb1;
```

или для вызова функтора-метода:

```
` вызов func1 с параметрами a,b,c и Object как текущим
объектом `
Object . func1 (a, b, c);
```

Однако, следующие примеры перевести на другие языки будет затруднительно:

```
` сумма двух компонент объекта Object `
Object . (memb1 + memb2);

` произведение двух функторов (вызванных с Object как текущим
объектом) `
Object . (func1 (a, b, c) * func2 (d, e));
```

Наконец, заметим, что вторым операндом для **with** может быть целый блок:

```
Object . { expr1; expr2; … exprN };
```

Здесь выполняется последовательность expr1 ... exprN с Object как текущим объектом данного класса. По сути, это очень похоже на оператор **with** Паскаля/Delphi (в C++/Java аналогичных средств нет) или **inspect** в Simula.
Перечисленным, однако, все возможности далеко не исчерпываются. Так, в AWL легко создать функтор, являющийся «методом» не одного, а двух или более классов (фактически, просто работающий с текущими экземплярами этих классов):

```
! mutual_method_AB (Params) = ClassA !! ClassB !! Body;
```

(Понятно, что один из двух квалификаторов можно опустить, если этот функтор декларирован локально в ClassA или ClassB.) Обратиться к нему можно так: предполагая, что ObjectA принадлежит к ClassA, а ObjectB к ClassB:

```
ObjectA . ObjectB . mutual_method_AB (Args);
```

или же

```
ObjectB . ObjectA . mutual_method_AB (Args);
```

(в данном случае, очевидно, порядок не играет никакой роли). Думаю, очевидно, что если ObjectA/ObjectB (или оба) здесь опущены, будет использован текущий объект для данного класса (или их обоих).
Теперь приведем более конкретные примеры работы с классами. Для многих задач, например, компьютерной трехмерной графики необходимы векторы в трехмерном

пространстве. Определим полезный класс Vector3D:

```
!! Vector3D (X Y Z) {
     ! length () = sqr (X*X + Y*Y + Z*Z)
};
```

```
! Vector3D.length:() [0] = sqr:add:(add:(mul:(Vector3D.X,
Vector3D.X), mul:(Vector3D.Y, Vector3D.Y)), mul:
(Vector3D.Z, Vector3D.Z)) => Vector3D.length:;

!! Vector3D:(X Y Z) {3} => Vector3D:;
```

В данный момент этот класс весьма примитивен: вектор Vector3D содержит три координатных компоненты X, Y, Z (они же являются его параметрами), и внутренний функтор (если хотите, «метод») **length**, вычисляющий длину вектора по известной всем формуле. Поскольку операция **with** здесь нигде явно не употребляется, функтор работает с *текущим экземпляром* Vector3D. Например, если мы создадим несколько векторов:

```
Vec1 = Vector3D (10, 12, 15);
Vec2 = Vector3D (-3, -5, -11);
```

```
set:(Vec1, Vector3D:(10, 12, 15)) => Vector3D:{10, 12, 15};
set:(Vec2, Vector3D:(neg:3, neg:5, neg:11)) => Vector3D:
{-3, -5, -11};
```

то вычислить длины этих векторов можно следующим образом:

```
Vector3D!!Vec1.length();
Vector3D!!Vec2.length();
```

```
with:(Vec1, Vector3D.length:()) => 21.656408;
with:(Vec2, Vector3D.length:()) => 12.4499;
```

Однако, возможен альтернативный подход к вычислению длины: с явной передачей вектора как параметра внешнему функтору (назовем его **length1**).

```
! length1(Vec) = Vector3D!! sqr (Vec.X*Vec.X + Vec.Y*Vec.Y +
Vec.Z*Vec.Z);
```

```
! length1:(Vec) [1] = sqr:add:(add:(mul:(with:(Vec,
Vector3D.X), with:(Vec, Vector3D.X)), mul:(with:(Vec,
Vector3D.Y), with:(Vec, Vector3D.Y))), mul:(with:(Vec,
Vector3D.Z), with:(Vec, Vector3D.Z))) => length1:;
```

Соответственно, использовать **length1** предполагается так:

```
length1(Vec1);
length1(Vec2);
```

```
length1:Vec1 => 21.656408;
length1:Vec2 => 12.4499;
```

Предыдущая реализация **length1** корректна, но выполнена «в стиле C++/Java», т. е. не очень изящна. На самом деле, поскольку мы все время обращаемся к одному объекту Vec, мы может вынести его «за скобки» (т. е. объединить все операции **with** в одну). «В стиле AWL» этот же код может быть реализован компактнее и эффективнее так:

```
! length1(Vec) = Vector3D!! sqr (Vec.(X*X + Y*Y + Z*Z));
```

или даже так:

```
! length1(Vec) = Vector3D!! Vec.sqr (X*X + Y*Y + Z*Z);
```

Рассмотрим теперь ситуацию, когда функтор класса должен работать с *несколькими* экземплярами этого класса. Например, для векторов совершенно необходимы операции векторного *сложения* и *вычитания*. Реализовать их проще всего так (предполагаем, что этот код добавляется в декларацию класса Vector3D):

```
! add (VA VB) = Vector3D(VA.X + VB.X, VA.Y + VB.Y, VA.Z +
VB.Z),
! sub (VA VB) = Vector3D(VA.X - VB.X, VA.Y - VB.Y, VA.Z -
VB.Z)
```

```
! Vector3D.add:(VA VB) [2] = .Vector3D:(add:(with:(VA,
Vector3D.X), with:(VB, Vector3D.X)), add:(with:(VA,
Vector3D.Y), with:(VB, Vector3D.Y)), add:(with:(VA,
Vector3D.Z), with:(VB, Vector3D.Z))) => Vector3D.add:;
! Vector3D.sub:(VA VB) [2] = .Vector3D:(sub:(with:(VA,
Vector3D.X), with:(VB, Vector3D.X)), sub:(with:(VA,
Vector3D.Y), with:(VB, Vector3D.Y)), sub:(with:(VA,
Vector3D.Z), with:(VB, Vector3D.Z))) => Vector3D.sub:;
```

Как и выше, это описание совершенно правильно, но неоптимально. В данном случае, «вынести за скобки» оба операнда, очевидно, не получится — но можно вынести любой из них по выбору. Таким образом, **add** можно переписать следующим образом:

```
! add(VA VB) = VA.Vector3D(X + VB.X, Y + VB.Y, Z + VB.Z),
```

или же так:

```
! add(VA VB) = VB.Vector3D (VA.X + X, VA.Y + Y, VA.Z + Z),
```

Аналогично можно поступить и с **sub**.
Каким бы из перечисленных способов эти функторы не были определены, работа с ними вполне тривиальна:

```
Vector3D!! add (Vec1, Vec2);
Vector3D!! sub (Vec2, Vec1);
```

```
Vector3D.add:(Vec1, Vec2) => Vector3D:{7, 7, 4};
Vector3D.sub:(Vec2, Vec1) => Vector3D:{-13, -17, -26};
```

Получение атрибутов классов и объектов

Если нужно узнать, к какому классу принадлежит некий объект, это очень просто:

```
class_of (Object);
```

В качестве результата вызова этого функтора возвращается *ссылка на класс* (т. е. функторное выражение), к которому принадлежит объект Object (если результат вычисления Object не является объектом, выводится сообщение об ошибке и возвращается **undef**). Например, для определенных выше объектов:

```
class_of (Vec1);
class_of (Vec2);
```

```
class_of:Vec1 => Vector3D:;
class_of:Vec2 => Vector3D:;
```

Функтор **self** выполняет до некоторой степени обратную операцию: он позволяет выяснить, какой объект в данный момент является текущим для класса, заданного аргументом.

```
self (class_Expr);
```

Здесь первый операнд (class_Expr) — функторное выражение, ссылающееся на некий класс. В качестве результата возвращается ссылка на текущий объект данного класса (или **undef**, если он в данный момент не определен). Например:

```
self (!Vector3D);
```

позволяет узнать текущий экземпляр класса Vector3D. В **self** операнд может быть опущен — тогда вместо него берется *текущий класс*.
Иными словами, **self**() возвращает ссылку на текущий экземпляр текущего же класса — то же самое, что в C++ или Java обеспечивает **this**.

Конструкторы и деструкторы

Дополнительными атрибутами класса в AWL могут быть *конструктор* и/или *деструктор*. Их основная задача — обеспечить правильную инициализацию и/или уничтожение экземпляров данного класса.
Подход к этой проблеме в AWL сильно отличается от принятого в C++ или Java: в отличие от этих языков, конструкторы и деструкторы AWL не являются функциональными объектами — это просто выражения (которые, как обычно, могут быть и блоками кода). У класса по определению не может быть более одного конструктора, а у конструктора, очевидно, не может быть параметров (зато они могут быть у класса как такового). Если конструктор есть, он вызывается (вычисляется) после того, как параметры класса получили значения из списка аргументов. Функции конструктора могут быть такими:

- инициализировать те внутренние компоненты класса, которые не входят в список аргументов и потому не инициализированы явно;
- компоненты, инициализированные как параметры, тоже иногда нужно подкорректировать (например, чтобы они соответствовали требованиям логической целостности объекта);
- наконец, конструктор может работать с внешними по отношению к классу данными (или с внешними по отношению к программе ресурсами — например, файлами или окнами графической системы).

Во всех перечисленных случаях использование конструктора оправдано.
У деструктора, очевидно, совершенно противоположные задачи. Деструктор вызывается (вычисляется) непосредственно перед уничтожением данного объекта класса. Если класс задействовал какие-то внешние ресурсы, деструктор может их освободить.
Приведем пример весьма тривиального класса (экземпляры которого, однако, умеют информировать о собственном “рождении” и “смерти”).

```
!! traced_class (A B)
= { <: ['[created: traced_class (' A ',' B ')]\n' ] }
~ { <: ['[destroyed: traced_class (' A ',' B ')]\n'] };
```

Поэкспериментировав с объектами traced_class, можно убедиться в том, что их создание и уничтожение происходят во вполне определенные моменты времени (в AWL применяется детерминированное управление памятью, в отличие от Java и некоторых других языков).

```
tr_a = traced_class (101, 303);
tr_b = traced_class ("aabb", "ddee");
```

```
[created: traced_class (101,303)]
set:(tr_a, traced_class:(101, 303)) => traced_class:{101,
303};
[created: traced_class (aabb,ddee)]
set:(tr_b, traced_class:("aabb", "ddee")) => traced_class:
{"aabb", "ddee"};
```

```
tr_a = tr_b = ();
```

```
[destroyed: traced_class (aabb,ddee)]
[destroyed: traced_class (101,303)]
set:(tr_a, set:(tr_b, ())) => ();
```

Производные классы

До сих пор мы для простоты изложения намеренно не рассматривали вопрос о *наследовании*. Все рассматриваемые нами классы были *оригинальными*, т. е. не имели явного “предка”. В отличие от многих ОО-языков, в AWL не существует ни “метаклассов”, ни абстрактных классов вроде Object, от которых происходят все остальные. Если класс не имеет явно объявленного предка – значит, его создание происходит “с нуля”.

Однако, класс также может быть и *производным* от любого другого. Если в декларации перед именем класса стоит имя суперкласса (в квадратных скобках) – значит, декларируемый класс является потомком указанного, автоматически наследуя при этом все его атрибуты, в т. ч. компоненты и локальные функторы. За счет этого обычно наследуется и большая часть функциональности, заложенной в суперклассе.

Производный класс, естественно, может дополнительно иметь собственный набор компонент и локальных функторов. Имена их *дополняют* соответствующие пространства имен суперкласса, а в случае конфликтов (которых, конечно, желательно избегать) локально переопределяют их.

Описанные ранее квалификаторы работают с производными классами вполне предсказуемым образом. Именно: если первым операндом является имя производного класса, то при этом автоматически “подключаются” (т. е. становятся доступными) все его классы-предки, вплоть до корневого.

Соответственно, если объект производного класса является первым операндом в **with**, то он временно становится текущим экземпляром не только для своего класса, но и для всех его предков. Все старые экземпляры, конечно, при этом сохраняются при входе в **with** и восстанавливаются при выходе из него.

Объект производного класса может быть инициализирован списком аргументов. При этом, однако, возникает вопрос: каким образом инициализируется суперкласс? Ответ простой: *первым аргументом в списке* всегда является инициализатор для суперкласса (в общем случае – также список), все дальнейшие аргументы передаются производному классу. Это правило применяется рекурсивно (т.е. для нескольких уровней наследования первый аргумент передается суперклассу, первый аргумент первого аргумента – суперклассу суперкласса и т. п.) Если суперклассу не требуется явный список параметров, можно задать вместо него **undef** ().

Аналогично работают конструкторы и деструкторы. Именно, если у суперклассов предусмотрены конструкторы, все они вызываются для каждого создаваемого объекта. При этом, самый внешний конструктор вызывается первым, а самый внутренний – последним (и каждый из конструкторов вызывается после инициализации параметров на данном уровне иерархии). Если у производного класса есть деструкторы, все они вызываются при уничтожении объекта данного класса, в порядке, противоположном конструкторам: первым вызывается деструктор для самого внутреннего класса, последним – для самого внешнего.

Наконец, в языке существует полиморфизм, обеспечиваемый виртуальными функторами, подробно описанными в следующей главе.

Виртуальные функторы

Этот механизм обеспечивает полиморфный вызов функторов, т. е. возможность создать для производного класса альтернативную версию одного или нескольких функторов-методов. Такие функторы мы будем называть *виртуальными*.
Для тех, кто знаком с C++ или Java такая концепция, конечно, в целом не нова. Важно рассмотреть отличия. Прежде всего, в AWL и декларация, и реализация виртуальных функторов имеют специальный синтаксис.
Каждый виртуальный функтор должен быть декларирован в качестве такового в суперклассе. Список виртуальных функторов является необязательной частью декларации любого класса, если он задан, то следует после конструктора/деструктора, но до фигурных скобок, содержащих локальные декларации класса. Сам список начинается с '#', за этим символом следует список виртуальных функторов, объявленных в данном классе. Этот список содержит только *имена*: все прочие атрибуты виртуальных функторов считаются специфичными для реализации, и потому при декларации не требуются.
Вот простейший пример:

```
!! MySuperClass
# { virtual1 virtual2 }
{
…
};
```

Здесь класс **MySuperClass** объявляет два виртуальных функтора (**virtual1** и **virtual2**), которые будут доступны как в нем, так и во всех производных от него классах. (Класс, в котором виртуальный функтор объявлен, мы будем называть его *оригинатором*.) Декларация этих функторов ничего не говорит об их параметрах и тем более их реализации: она лишь объявляет имена этих функторов виртуальными.
Но, конечно, чтобы эти функторы делали что-то полезное, надо дать им *реализации*. Класс-оригинатор и любой производный от него класс может реализовать любые из своих виртуальных функторов: синтаксис реализации отличается от обычной декларации функтора тем, что перед именем стоит символ '#'. Например, в теле **MySuperClass** можно написать следующее:

```
! #virtual1 = <: "Greetings from SuperClass!!!\n"
```

Наличие # перед именем функтора важно: без него это будет воспринято как декларация отдельного функтора, никак не связанного с ранее объявленными виртуальными. Если у этого класса имеются потомки, в любом из них virtual1 можно переопределить:

```
!! [MySuperClass] MySubClass {
! #virtual1 = <: "Greetings from SubClass!\n"
};
```

Полагаю, то, как работает механизм виртуального вызова, вполне очевидно. При вызове **virtual1** вызывается та версия этого метода, которая зависит от текущего экземпляра класса **MySuperClass** (принимая во внимание, разумеется, и все производные от него классы). Таким образом, **MySuperClass**().**virtual1**() выведет текст "Greetings from SuperClass!", а **MySubClass**().**virtual1**() соответственно "Greetings from SubClass!". Конечно, в потомках класса **MySubClass** метод **virtual1** также может быть переопределен, например может выводить что-то еще. Если виртуальный метод не был явно переопределен в субклассе, заимствуется версия из суперкласса. Виртуальный метод вообще может быть оставлен без

определения (как в нашем примере **virtual2**): вызов неопределенного виртуального метода не считается ошибкой, и не имеет никакого эффекта, возвращая **undef**.
Наконец, любой из производных классов может определять дополнительные уникальные виртуальные методы, которые могут иметь реализацию в нем или в его подклассах.
Приведем более практичный пример: пусть мы хотим работать с абстракциями, представляющими собой простые геометрические фигуры. Для этого логично создать абстрактный класс **Shape**:

```
` Абстрактный геометрический объект `
!! Shape
     # {
     put       ` вывести информацию о фигуре `
     perimeter ` вычислить периметр `
     area      ` вычислить площадь `
     };
```

Вся функциональность заложена в виртуальные функторы: **put** позволяет выводить информацию в стандартный вывод, **perimeter** вычисляет ее периметр, **area** – ее площадь. Теперь нетрудно создать набор конкретных геометрических фигур:

```
` Прямоугольник (со сторонами A, B) `
!! [Shape] Rectangle (A B) {
     ! #put = <: ['Rectange (' A ' * ' B ')'],
     ! #perimeter = 2*(A + B),
     ! #area = A*B
     };
```

```
` Квадрат (со стороной S) `
!! [Shape] Square (S) {
     ! #put = <: ['Square [' S ']'],
     ! #perimeter = 4*S,
     ! #area = S*S
     };
```

```
` Круг (с радиусом R) `
!! [Shape] Circle (R) {
     ! #put = <: ['Circle [' R ']'],
     ! #perimeter = 2*pi(R),
     ! #area = pi(R*R)
     };
```

```
` Треугольник (со сторонами A, B, C) `
!! [Shape] Triangle (A B C) {
     ! #put = <: ['Triangle (' A ',' B ',' C ')'],
     ! #perimeter = A + B + C,
     ! #area : [p] = {
          `(формула Герона...)`
          p = (A + B + C) / 2;
          sqr (p*(p - A)*(p - B)*(p - C))
          }
     };
```

Теперь легко можно создать список из геометрических фигур.

```
Figures = (
     Square ((), 5),
     Rectangle ((), 6, 9),
     Circle ((), 10),
     Triangle ((), 5, 12, 13)
     );
```

```
set:(Figures, Square:((), 5), Rectangle:((), 6, 9), Circle:
((), 10), Triangle:((), 5, 12, 13)) => (Square:{Shape:{}->
5}, Rectangle:{Shape:{}-> 6, 9}, Circle:{Shape:{}-> 10},
Triangle:{Shape:{}-> 5, 12, 13});
```

(Заметьте, что первый аргумент в списке инициализаторов для каждой фигуры – **undef**, предназначенный для суперкласса **Shape**).
Работать со списком фигур можно так:

```
Shape!!l_loop (fig, Figures, fig.{
     put ();
     <: ": ";
     <: ("периметр = ", perimeter(), "; ");
     <: ("площадь = ", area());
     <: "\n";
     });
```

```
Square [5]: периметр = 20; площадь = 25
Rectange (6 * 9): периметр = 30; площадь = 54
Circle [10]: периметр = 62.831853; площадь = 314.15927
Triangle (5,12,13): периметр = 30; площадь = 30.
l_loop:(fig, Figures, with:(fig, {
    Shape.put:();
    f_put:((), ": ");
    f_put:((), "периметр = ", Shape.perimeter:(), "; ");
    f_put:((), "площадь = ", Shape.area:());
    f_put:((), "
");
```

Массивы

Создание массива

В AWL предусмотрен механизм массивов. *Массив* – это структурный тип данных, содержащий упорядоченный набор значений. Каждое из них доступно по целочисленному индексу (или по набору индексов, так как массив в общем может быть многомерным). В отличие от списков, размеры массива задаются при его создании (и их изменение после

создания требует больше усилий).
Обычно новый массив создается обращением к примитиву **array** (у него есть синоним: **a_create**):

```
` создать вектор из 50 элементов `
Vector1 = array (50);

` создать матрицу из 40 строк и 70 столбцов `
Matrix2 = array (40, 70);
```

Операндом для **array** является *список размеров* массива (который вырождается в скаляр, когда массив одномерен). Любой из размеров массива может быть задана не константой (как в приведенных случаях), а произвольным выражением (вычисляемым как целое). Когда массив имеет более одной размерности, начальные элементы списка мы будем называть *внешними* размерами массива, а конечные – *внутренними*. Количество измерений массива мы будем называть его *рангом*: например, ранг линейного массива равен 1, а прямоугольной матрицы – 2. Максимальный допустимый ранг массива – 16 измерений.
Конечно, ни один из размеров массива не должен быть отрицательным. Однако, любой размер массива может быть равен 1 или даже 0. (В последнем случае массив не будет содержать ни единого элемента, и, хотя полезность такого массива сомнительна, как частный случай это допускается). Заметим, что (в отличие от списков) массив с размерами 0 или 1 не "вырождается" в примитивные элементы данных (как скаляр или **undef**): он всегда остается массивом со структурой, с котором создавался.
Массивы в AWL являются *гетерогенными*, т. е. могут содержать данные любого типа (причем разные элементы могут иметь разные типы). Для только что созданного массива все его элементы инициализированы в **undef**.
Интерпретатор выводит массив в виде списка его размеров (в том же порядке, что и в конструкторе), заключенных в квадратные скобки и разделенных пробелами. Заметим, что значения собственно элементов по умолчанию не выводятся. Например, для вышесозданных массивов:

```
Vector1 => [50];
Matrix2 => [40 70];
```

Конечно, вместо массива может быть использован список (а вместо многомерного массива – иерархия вложенных списков). У каждого из подходов есть свои сильные и слабые стороны. Массивы обеспечивают более быстрый доступ к элементам (для списков время доступа к элементу *I* имеет порядок *O(I);* для массива оно не зависит от индекса элемента), и по сравнению со списками занимают немного меньше памяти. Однако, если добавление новых элементов в список или их исключение из него реализуются просто (и эффективность этих операций не зависит от количества элементов в списке), то изменение размеров массива является относительно трудоемкой операцией (тем более медленной, чем больше элементов есть в массиве).
Предикат **is_array**(V) возвращает истинное значение, если результат *V* – произвольный массив.

Доступ к элементам массива

Чаще всего работа с массивом выполняется поэлементно, для чего используется примитив **a_elem**. Вызов

```
a_elem (Array, Index0, Index1 ... IndexN)
```

предоставляет доступ (мутабельный) к элементу массива *Array* с индексами *Index0...IndexN*. Порядок индексов соответствует порядку размеров массива: *Index0* соответствует самому

внешнему измерению, *IndexN* – самому внутреннему. Если количество индексов в списке меньше ранга массива, то недостающие значения индексов считаются нулевыми (а если оно больше, то "лишние" индексы просто игнорируются). Элементы нумеруются с нуля: если размер массива по соответствующему измерению равен *N*, то минимальный индекс равен 0, а максимальный – *N-1*. Если один из индексов оказывается за пределами соответствующего размера массива, возвращается **undef**.
Как и для многих операций, для **a_elem** есть компактный синтаксис. Эта запись:

```
Array {Index0, Index1 ... IndexN}
```

равносильна приведенной выше. Например:

```
Vector1{25} = 22.33;
++ Matrix2{8, 12};
```

Получение размеров массива

Несколько примитивных операций позволяют получить информацию о ранге и размерах массива. Примитив **a_dims**(Array) возвращает список размеров массива *Array* (в порядке конструктора, от внешних к внутренним).

```
a_dims (Vector1);
a_dims (Matrix2);
```

```
a_dims:Vector1 => 50;
a_dims:Matrix2 => (40, 70);
```

Примитив **a_rank**(Array) возвращает ранг массива *Array*. (Это значение всегда равно #**a_dims** (Array)). Например:

```
a_rank (Vector1);
a_rank (Matrix2);
```

```
a_rank:Vector1 => 1;
a_rank:Matrix2 => 2;
```

Наконец, результатом **a_total**(Array) является общее количество элементов в массиве *Array*. Оно всегда равно произведению его размерностей.

```
a_total (Vector1);
a_total (Matrix2);
```

```
a_total:Vector1 => 50;
a_total:Matrix2 => 2800;
```

Заполнение массива

Если есть необходимость присвоить всем элементам массива одно и то же значение, удобно использовать примитив **a_fill**(Array, Value). В результате значение *Value* присваивается всем элементам массива *Array*. Например:

```
a_fill (Matrix2, 0);
```

полностью обнуляет все элементы матрицы *Matrix2*.
Заметим, что **a_fill**(Array) вызывает полную очистку массива *Array*, присваивая всем его элементам значение **undef**. Также заметим, что все элементы массива являются ссылками на единственный экземпляр *Value* – никакого копирования не происходит.

Преобразование между массивами и списками

Примитивный функтор **a_save**(Array) в определенном смысле сохраняет все содержимое массива, разворачивая его в линейный список-результат. Порядок элементов в списке соответствует *внутреннему порядку* элементов массива: т. е. элементы в списке упорядочены так, что цикл по самому внешнему измерению является самым внешним (и выполняется один раз), а цикл по внутреннему измерению – самым внутренним (выполняемым максимальное количество раз).
Заметим, что финальные элементы списка, равные **undef**, в этот список не включаются. В случае, когда массив полностью пуст, результатом **a_save** является **undef**.
Обратную операцию выполняет примитивный функтор **a_load**. В результате **a_load**(Array, List) в массив *Array* последовательно загружаются элементы из списка *List*. Элементы упорядочены также, как и в **a_save** (таким образом, применение **a_load** с результатом **a_save** полностью восстанавливает содержимое массива). Все избыточные элементы в списке *List* игнорируются.

Словари

Создание словаря

Словарь (также *хэш*) – это неупорядоченный набор *значений*, связанных с *ключами*. При этом к каждому из значений словаря можно получить доступ по ключу, т. е. словарь можно рассматривать как массив неограниченного размера, индексированный произвольными элементами данных вместо целых чисел. Каждую пару "ключ-значение" мы будем называть *элементом* словаря. В процессе работы со словарем можно добавлять новые элементы, удалять существующие, и менять значения, связанные с ключами.
Для создания нового словаря используется примитив **hash** (или **h_create**). Его результатом является новый словарь; его параметрами (необязательными) является внутренняя емкость хэша (кол-во "карманов"), и минимум/максимум допустимой загрузки из расчета на один "карман". Эти параметры позволяют подстраивать эффективность хэша для конкретной задачи (обычно они опускаются). Например:

```
Hash1 = hash (10);
Hash1;
```

```
set:(Hash1, hash:10) => <0 / 10>;
Hash1 => <0 / 10>;
```

Как легко видеть, объекты-словари выводятся системой в виде <N / V>, где N – общее количество элементов в словаре, а V – его внутренняя емкость.
Созданный обращением к **hash** словарь изначально является пустым. Предикат **is_hash**(V) возвращает истинное значение, если результат *V* – произвольный словарь.

Доступ к элементам словаря

Чаще всего работа со словарями выполняется поэлементно. Выполнение примитива **h_elem**:

```
h_elem (Hash, Key)
```

предоставляет мутабельный доступ к элементу словаря *Hash*, связанному с ключом *Key*. Если элемента с таким значением ключа не было, он автоматически создается (со значением **undef**). Как уже говорилось, в качестве ключей в любом словаре могут использоваться выражения произвольного типа (в т. ч. скаляры и списки из скаляров). Например:

```
h_elem (Hash1, 1) = 'one';
h_elem (Hash1, 2) = 'two';
h_elem (Hash1, 3) = 'three';
h_elem (Hash1, 'one') = 1;
h_elem (Hash1, 'two') = 2;
h_elem (Hash1, 'three') = 3;
```

позволяет занести в словарь *Hash1* ряд элементов с заданными значениями. Поскольку в словаре может быть только один ключ с заданным значением, если в словаре уже присутствовали элементы с заданными ключами, их значения заменяются на новые. Значение элемента, как и ключ, может иметь произвольный тип.

При поиске ключа в словаре используется *сравнение на идентичность*. В частности, два скаляра считаются идентичными, если совпадают их типы и значения. Скаляры с различными типами считаются различными, т. е. неявные приведения, определенные для скалярных операций, в этом случае не применяются (например, 3, 3.0 и “3” - это три различных ключа). Списки считаются идентичными, если попарно идентичны их элементы.

Если необходим доступ к элементу только на чтение, можно использовать примитив **h_lookup** (*Hash*, *Key*). В отличие от **h_elem**, если ключ *Key* отсутствует, он не создается, а результатом в этом случае будет **undef**. Например:

```
(h_lookup (Hash1, 1), h_lookup (Hash1, 2), h_lookup (Hash1,
3));
(h_lookup (Hash1, 'three'), h_lookup (Hash1, 'two'), h_lookup
(Hash1, 'one'));
```

```
(h_lookup:(Hash1, 1), h_lookup:(Hash1, 2), h_lookup:(Hash1,
3)) => ("one", "two", "three");
(h_lookup:(Hash1, "three"), h_lookup:(Hash1, "two"),
h_lookup:(Hash1, "one")) => (3, 2, 1);
```

После создания нового элемента словаря, он остается в нем независимо от того, какие значения ему присваиваются. В частности, значению элемента можно присвоить **undef**, однако, это не приводит к удалению элемента из словаря.

Для этой цели следует использовать специальный примитив: **h_remove**. Выполнение **h_remove** (Hash, Key) удаляет из словаря *Hash* элемент с ключом *Key*, возвращая значение, которое было с ним связано. Если элемента с таким ключом не было, ничего не изменяется, и возвращается **undef**.

```
h_remove (Hash1, 1);
h_remove (Hash1, 2);
h_remove (Hash1, 3);
```

```
h_remove:(Hash1, 1) => "one";
h_remove:(Hash1, 2) => "two";
h_remove:(Hash1, 3) => "three";
```

Для того, чтобы узнать текущее количество элементов в словаре *Hash*, можно использовать **h_count** (Hash).

Наконец, обращение к **h_clear** (Hash) быстро и полностью очищает словарь *Hash*, удаляя из него все элементы.

Итераторы по словарю

Для эффективного перебора всех элементов словаря, удобнее всего использовать примитив **h_loop**.

```
h_loop (Hash, Variable, Body)
```

выполняет (или вычисляет) операнд *Body* для каждого элемента словаря *Hash*, предварительно присвоив выражению *Variable* (оно должно быть мутабельным) данный элемент в виде списка (*Ключ*, *Значение*). Порядок перебора элементов словаря не определен. В качестве значения **h_loop** (как и большинство других итераторов) возвращает последнее значение *Body*.
Например, таким образом можно вывести все элементы словаря (для *Hash1* из предыдущего раздела):

```
h_loop (Hash1, elem, <: (l_head(elem), " => ", l_tail(elem),
"\n"));
```

```
two => 2
three => 3
one => 1
```

Преобразование между словарями и списками

Как и для массивов, предусмотрены примитивы, позволяющие преобразовывать словари в списки ключей и/или значений (и наоборот).
Операция **h_save**(Hash) возвращает содержимое словаря *Hash* в виде списка (всегда открытого), элементами которого являются (как для **h_loop**) списки вида (*Ключ*, *Значение*). Если нужен только список ключей словаря, можно использовать примитив **h_keys**(Hash); если нужен только список значений – примитив **h_values**(Hash) (оба примитива также возвращают открытые списки). Порядок элементов во всех случаях следует считать неопределенным (однако, он одинаков для **h_keys**, **h_values** и **h_save**,).
Например, для рассмотренного выше *Hash1*:

```
h_keys(Hash1);
h_values(Hash1);
h_save(Hash1);
```

```
h_keys:Hash1 => ("one", "three", "two", );
h_values:Hash1 => (1, 3, 2, );
h_save:Hash1 => (("one", 1), ("three", 3), ("two", 2), );
```

Примитив **h_load** выполняет операцию, совершенно обратную **h_save**. **h_load**(*Hash*, *List*), получая в качестве аргумента *List* открытый список пар (*Ключ*, *Значение*), аналогичный возвращаемому **h_save**, загружает их как элементы в словарь *Hash*. Например:

```
h_load (Hash1,
   ['four' 4],
   ['five' 5],
   ['six' 6],
   );
```

```
h_load:(Hash1, ("four", 4), ("five", 5), ("six", 6), ) =>
<6 :: 10>;
```

```
h_save (Hash1);
```

```
h_save:Hash1 => (("six", 6), ("one", 1), ("three", 3),
("four", 4), ("two", 2), ("five", 5), );
```

Порядок элементов в **h_load** не имеет значения, но если в списке повторяются ключи, более поздние элементы с этими ключами замещают более ранние.

Образцы

Сопоставление с образцами

В языке также имеется механизм *образцов*, выполняющих функции, аналогичные регулярным выражениям во многих других языках. Но, в отличие от них, регулярные выражения AWL не кодируются в виде символьных строк: они являются самостоятельным и полноправным типом данных. Образец, созданный однажды, может использоваться многократно.
Основные операции, определенные для образцов – различные виды *сопоставления* и *поиска*. Простейший вид сопоставления – это проверка соответствия начала строки заданному образцу, для чего используется примитив **rx_match**. Выполнение **rx_match**(Pattern, String) возвращает длину начального фрагмента строки *String*, который удалось успешно сопоставить с *Pattern*. Если сопоставить не удалось, результатом является -1. Заметим, что эту ситуацию необходимо отличать от удачного сопоставления пустой строки, при котором результатом является 0.
Более употребительны поисковые операции: **rx_findfirst** и **rx_findlast**, обе операции также имеют два параметра: *Pattern* и *String*. Операции выполняют поиск образца *Pattern* в заданной строке *String* (соответственно, вперед с начала для **rx_findfirst** или назад с конца для **rx_findlast**). Для обеих операций результатом (*Range*) является диапазон индексов, определяющих положения заданного контекста в строке *String* (т. е. *String* $[*Range*] полностью сопоставляется с образцом *Pattern*). Если образец не найден, в качестве результата возвращается **undef**.
Все образцы создаются с помощью операций-конструкторов. Некоторые операции конструируют примитивные образцы с простой семантикой сопоставления. Другие операции позволяют конструировать более сложные образцы из своих операндов.
Рассмотрим основные конструкторы образцов по порядку.

Литеральные образцы

Простейшие операции-конструкторы позволяют создавать образцы, соответствующие фиксированным символам и символьным строкам. Конструктор **rx_char**(Code) возвращает примитивный образец, соответствующий одному символу с кодом *Code* (из набора ASCII/Unicode). Аналогично, конструктор **rx_string**(String) возвращает примитивный образец, соответствующий исключительно строке *String*.
Например:

```
rx_char (48);
rx_string ("Hello");
```

```
rx_char:48 => '0';
rx_string:"Hello" => "Hello";
```

Операция **rx_string** может применяться неявно: если для любой из перечисленных ниже операций операндом-образцом является строка, к ней неявно применяется **rx_string**, превращая ее в строковый образец.

Альтернация образцов

Операция *альтернации* позволяет создать образец, сопоставляющийся с одной из двух альтернатив. Выполнение **rx_alt**(Pat1, Pat2) возвращает образец, сопоставляющийся либо с *Pat1*, либо с *Pat2*. Например:

```
rx_alt (“Hello”, “Goodbye”);
```

создает образец, сопоставляющийся со строкой “Hello” или “Goodbye”.

```
rx_alt:("Hello", "Goodbye") => <"Hello" | "Goodbye">;
```

Вложенные альтернации позволяют создать образец, сопоставляющийся с произвольным количеством альтернатив. Например:

```
rx_alt (“Yes”, rx_alt (“No”, “Maybe”));
```

создает образец, сопоставляющийся со строками “Yes”, “No” и “Maybe”.

```
rx_alt:("Yes", rx_alt:("No", "Maybe")) => <"Yes" | <"No" |
"Maybe">>;
```

Приведем пример поиска такого образца:

```
rx_findfirst (rx_alt ('ab', 'ba'), 'abbbbba');
rx_findlast (rx_alt ('ab', 'ba'), 'abbbbba');
```

```
rx_findfirst:(rx_alt:("ab", "ba"), "abbbbba") => (0, 2);
rx_findlast:(rx_alt:("ab", "ba"), "abbbbba") => (5, 7);
```

Конкатенация образцов

Операция *конкатенации* позволяет создать новый образец, успешно сопоставляющийся с конкатенацией двух образцов-операндов. Выполнение **rx_cat**(Pat1, Pat2) возвращает образец, сопоставляющийся с любой конкатенацией образцов *Pat1* и *Pat2*.
Так, например,

```
rx_cat (rx_alt (“hello”, “goodbye”), “!”);
```

возвращает образец, сопоставляющийся с строкой “hello!” или “goodbye!”. Аналогично, выполнение

```
rx_cat (rx_alt (“aa”, “bb”), rx_alt (“cc”, “dd”));
```

возвращает образец, сопоставляющийся с одной из четырех строк: “aacc”, “bbcc”, “aadd”, “bbdd”.

```
rx_cat:(rx_alt:("aa", "bb"), rx_alt:("cc", "dd")) => <<"aa"
| "bb"> & <"cc" | "dd">>;
```

Очевидно, что применение конкатенации не имеет явного смысла, когда оба операнда являются символьными или строковыми литералами.

Репликация образца

Операция *репликации* позволяет создать новый образец, сопоставляющийся с повторением заданного образца (т. е. его последовательной конкатенацией с самим собой) определенное число раз. Результатом **rx_rep**(Range, MinMax, Pat) является образец, сопоставляющийся с некоторым числом повторений образца *Pat*. Допустимое число повторений задается параметром-диапазоном *Range*: минимальное число повторений задается нижней границей диапазона, максимальное – верхней границей. Если нижняя граница диапазона опущена, предполагается 0; если опущена верхняя граница, максимальное число повторений не лимитировано, вплоть до бесконечности. Параметр *MinMax* рассматривается как логическое значение: при его ложности, предполагается минимальное количество повторений, при его

истинности – максимальное.
Приведем примеры:

```
rx_rep (2..5, 0, "()");
```

```
rx_rep:((2, 5), 0, "()") => <"()" * 2.>>.5>;
```

Результат сопоставляется с повторением строки “()” как минимум 2 и как максимум 5 раз подряд (при этом предпочтение отдается минимуму повторов).

```
rx_rep (10, 1, "/-/");
```

```
rx_rep:(10, 1, "/-/") => <"/-/" * 0.<<.10>;
```

Результат сопоставляется с повторением строки “/-/” не более 10 раз подряд (предпочтение отдается максимуму повторов).

Произвольный символ

Функтор **rx_any**() является одной из простейших операций конструкторов: он возвращает образец, сопоставляющийся с произвольным символом. (Параметров у **rx_any** нет.)
Этот образец чаще всего используется в качестве операнда для **rx_rep**: например, **rx_rep** (10..20, 0, **rx_any**()) возвращает образец, сопоставляющийся с произвольной (предпочтительно, короткой) последовательностью, не менее 10 и не более 20 символов длиной.

Оценка длины образца

Имеется полезная операция, позволяющая оценить, со строкой какой длины может быть успешно сопоставлен образец. Результатом **rx_length**(Pattern) является список-диапазон, границы которого определяют минимальную и максимальную длину контекста, с которым может быть успешно сопоставлен образец *Pattern*.
Например:

```
rx_length (rx_alt ('aa', 'bbcc'));
rx_length (rx_cat (rx_alt ('hello', 'goodbye'), '.'));
rx_length (rx_rep (4..7, 0, rx_alt ('ad', 'ret')));
```

```
rx_length:rx_alt:("aa", "bbcc") => (2, 4);
rx_length:rx_cat:(rx_alt:("hello", "goodbye"), ".") => (6,
8);
rx_length:rx_rep:((4, 7), 0, rx_alt:("ad", "ret")) => (8,
21);
```