Erlang: Строковые

Андрей Ушаков показывает, что строки строкам рознь: в Erlang они значительно удобнее и безопаснее.

Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

Строки – это один из основных типов данных в языках программирования, наряду с целыми и действительными числами. В отличие от чисел, строки – более сложный тип данных, и от поддержки их в языке зависит и удобство работы с ними, и безопасность. В качестве примера языка, в котором со строками работать неудобно и небезопасно, стоит привести язык C, в котором нет отдельного строкового типа данных. Даже для элементарной операции конкатенации строк мы вынуждены использовать функцию библиотеки языка – **concat**. К тому же многие функции для работы со строками небезопасны, и именно из-за этих неудобных и небезопасных функций появляется большое количество ошибок, наподобие ошибки переполнения буфера. Давайте разберемся, насколько удобно работать со строковыми данными в языке **Erlang**.

В языке **Erlang** специального типа данных для строк, такого как **std::string** в C++ или **java.lang.String** в Java, нет: строки в **Erlang** – это либо списки целых чисел, либо битовые строки. На первый взгляд это кажется шагом назад по сравнению с современными («мейнстримовыми») языками программирования. Однако давайте более пристально взглянем на тип данных «строка», например, в C++ или Java: строковый тип данных – это контейнер, который содержит набор символов и набор методов для манипуляции этими символами. Язык **Erlang** не является объектно-ориентированным языком, поэтому в нем разделены данные и методы для обработки этих данных; в нашем случае, данные – это набор символов, а методы для обработки этих данных содержатся в модулях, таких как **lists**, **string**, **unicode**. Поэтому нет ничего страшного в том, что строки в **Erlang** – это всего лишь либо список целых чисел, либо битовая строка: главное – насколько удобно нам работать с ними.

Один из наиболее важных вопросов, возникающих при работе со строками – это вопрос о том, какие наборы символов или кодировки поддерживаются строковым типом данных. Язык **Erlang** поддерживает все виды юникода (**UTF-8**, **UTF-16**, **UTF-32**) и кодировку символов **ISO-latin-1 (ISO8859-1)**.

По поводу юникода нужно сказать следующее: поскольку одно из представлений строк – это списки целых чисел, то каждое целое число в этом списке должно быть действительной кодовой точкой и представлять один символ. Только при соблюдении этого условия список целых чисел будет считаться строкой в юникод (список целых чисел может и не являться представлением строки юникод, что очевидно). Что же касается поддержки строк в других кодировках (отличных от юникода и **ISO-latin-1 (ISO8859-1)**) – такой поддержки нет. Но не все так плохо: у нас есть два варианта, что с этим делать.

Во-первых, мы можем перекодировать символы строки в заданной кодировке в символы юникод (да, при этом таблицу соответствия между символами заданной кодировки и символами юникод нам придется задавать самим). Во-вторых, мы можем работать со строками в заданной кодировке как со списком однобайтовых целых чисел. При этом наши возможности будут ограничены функциональностью модулей **lists** и **string**, а более сложная функциональность работы со строками, такая как регулярные выражения, форматирование, ввод и вывод будет недоступна (по вполне понятным причинам).

Во многих задачах этого достаточно – например, в задаче перевода текста из одной однобайтовой кодировки в другую.

Про кодировку **ISO-latin-1 (ISO8859-1)** следует сказать еще следующее: в этой кодировке записывается исходный текст программы. Компилятор ограничивает имена функций и переменных, атомы, строковые константы символами из этой кодировки. Однако ожидается, что в скором времени это ограничение будет снято, и в исходном тексте программы можно будет применять юникод.

Следующий, не менее важный вопрос – как задавать символьные и строковые константы. Если нам достаточно символов из **ISO-latin-1** (**ISO8859-1**), то все очень просто. Символьные константы получаются при помощи символа **$** и следующего за ней символа: например, **$0** – символьная константа для символа **"0"**. Помимо обычных печатаемых символов, символьной константой может быть и непечатаемый символ с использованием escape-последовательностей (при помощи символа обратного слэша "\"); например, **$\s** – для пробела, **$\n** – для новой строки, **$** – для самого обратного слэша. Символьные константы – это «синтаксический сахар» времени компиляции; значением константы является численное представление символа в кодировке **ISO-latin-1 (ISO8859-1)**. Так, например, значением символьной константы **$0** является **48**.

Строковые константы определяются как набор символов, заключенных в двойные кавычки ". Естественно, что в строковых константах можно использовать escape-последовательности, и вполне естественно, что для символа двойной кавычки есть escape-последовательность **\"**, чтобы этот символ можно было использовать внутри строковых констант. Значение строковой константы – это список из целых чисел, соответствующих символам

данные

в строковой константе в кодировке **ISO-latin-1 (ISO8859-1)**. Строковые константы, в отличие от символьных констант, являются «синтаксическим сахаром» как времени компиляции, так и времени выполнения. Если список целых чисел может быть представлен в виде строковой константы (если все числа соответствуют некоторым символам), то среда выполнения **Erlang** будет представлять такой список в виде строковой константы. Так, например, список **[48, 49, 50]** будет представлен в виде строки **"012"** (это можно проверить, если в консоли **Erlang** ввести **[48, 49, 50].** – результатом будет строка **"012"**); а список **[0, 300]** среда **Erlang** в виде строки не представляет. Если же нам необходимы строковые и символьные константы в юникоде, то все обстоит намного хуже (в текущей версии среды **Erlang**; в будущем все может измениться). Единственный способ задать такие константы – использовать escape-последовательность **\x{X…}**, где **X…** – число в шестнадцатеричном представлении. Данная escape-последовательность задает символ по его численному представлению. Так, например, символ **"0"** можно задать следующей escape-последовательностью: **\x{30}**. Есть еще два варианта escape-последовательностей, позволяющие задать символ по его численному представлению: **\000** и **\xXX**. Первая позволяет задать символ по его численному восьмеричному представлению в диапазоне от 000 до 777, вторая – по его численному шестнадцатеричному представлению в диапазоне от 00 до FF.

> «Существует "синтаксический сахар" времени компиляции.»

Другой способ работать со строковыми данными в языке **Erlang** – это использовать битовые строки (см. LXF148). Как мы помним, битовые строки состоят из сегментов и определяются следующим образом: **<<E1, …, En >>**. Такое определение справедливо как в выражениях, так и в операции соответствия шаблону [pattern matching], с той разницей, что в выражениях сегменты сами являются выражениями, а в операции соответствия шаблону сегменты могут быть как выражениями, так и неинициализированными переменными. С каждым сегментом может быть связан спецификатор типа; полный список спецификаторов см. в LXF148.

Рассмотрим, как мы можем определять строковые данные в битовых строках. Во-первых, можно просто задать внутри битовой строки обычную строковую константу; при этом строковая константа преобразовывается в однобайтовые сегменты, количество которых равно количеству символов в строке, а их значения – численным представлениям символов в кодировке **ISO-latin-1 (ISO8859-1)**. Более того, «синтаксический сахар» преобразования в строковые константы в битовых строках проявляется и во время выполнения: если битовая строка может быть представлена в виде строковой константы, то среда выполнения **Erlang** ее так и будет представлять. Так, например, битовую строку **<<48, 49, 50>>** среда выполнения **Erlang** представляет в следующем виде **<<"012">>**. Во-вторых, для сегмента мы можем задать один из следующих спецификаторов типа: **utf8**, **utf16**, **utf32**. Для сегментов **utf16** и **utf32** мы можем указать спецификатор порядка следования байтов (через символ дефис «-»): **big** или **little**. Как мы помним из предыдущего номера, есть еще один спецификатор порядка следования байт – **native**, который означает, что порядок следования байт определяется процессором в момент выполнения; но данный спецификатор не работает для сегментов **utf16** и **utf32**, генерируя ошибку времени выполнения (на данной версии среды выполнения **Erlang**). Существует «синтаксический сахар» времени компиляции, позволяющий при использовании строковой константы внутри битовой строки задать для нее спецификаторы типа и порядка следования байт (естественно, что спецификаторы будут применены к каждому сегменту, получаемому из этой строковой константы). Так, например, можно задать следующее выражение для битовой строки: **<<"01"/utf16-little>>**; это выражение будет преобразовано в следующую битовую строку: **<<48, 0, 49, 0>>**. При конструиро- »

Кодировка символов

» **Юникод** (англ. **Unicode**) – стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Стандарт предложен в 1991 году некоммерческой организацией «Консорциум Юникода» (англ. Unicode Consortium, Unicode Inc.). Применение этого стандарта позволяет закодировать очень большое число символов из разных письменностей: в документах **Unicode** могут соседствовать китайские иероглифы, математические символы, буквы греческого алфавита, латиницы и кириллицы; при этом становится ненужным переключение кодовых страниц.

Стандарт состоит из двух основных разделов: универсальный набор символов (англ. **UCS**, universal character set) и семейство кодировок (англ. **UTF**, Unicode transformation format). Универсальный набор символов задает однозначное соответствие символов кодам – элементам кодового пространства, представляющим неотрицательные целые числа. Семейство кодировок определяет машинное представление последовательности кодов **UCS**.

Коды в стандарте Юникод разделены на несколько областей. Область с кодами от **U+0000** до **U+007F** содержит символы набора **ASCII** с соответствующими кодами. Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы. Часть кодов зарезервирована для использования в будущем. Под символы кириллицы выделены области знаков с кодами от **U+0400** до **U+052F**, от **U+2DE0** до **U+2DFF**, от **U+A640** до **U+A69F**.

Более подробно см. http://ru.wikipedia.org/wiki/Unicode

» **ISO/IEC 8859-1** (также известная как **ISO 8859-1** и **Latin-1**) – кодовая страница, предназначенная для западноевропейских языков; она базируется на символьном наборе популярных в прошлом терминалов VT220. Кодовые позиции 0–31 (0×0–0×1F) и 127–159 (0×7F–0×9F) не определены. По образцу **ISO 8859-1** сделаны все остальные кодировки серии **ISO 8859**.

» **ISO-8859-1** – кодировка, зарегистрированная IANA в 1992 г. В отличие от **ISO/IEC 8859-1**, кодовые позиции 0–31 и 127–159 здесь заполнены управляющими символами (большинство из которых, впрочем, все равно никто не использует). В HTML **ISO-8859-1** является кодировкой по умолчанию (в XHTML, однако, кодировкой по умолчанию является **UTF-8**).

В Юникоде первые 256 кодовых позиций совпадают с **ISO-8859-1**.

Более подробно см. http://ru.wikipedia.org/wiki/ISO8859-1

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

вании сегментов со спецификатором типа **utf8**, **utf16** или **utf32** значение сегмента должно быть действительной кодовой точкой. Это означает, что значение сегмента должно принадлежать одному из следующих диапазонов: **0…16#D7FF**, **16#E000…16#FFFD**, или **16#10000…16#10FFFF**. То же самое справедливо и для операции соответствия шаблону. Если значение сегмента не принадлежит одному из указанных диапазонов, то мы получим ошибку времени выполнения. Так, например, выражение **<<16#D800/utf16>>** и операция соответствия шаблону **<<UniChar/utf16>> = <<16#D800:16/integer>>** дадут ошибку времени выполнения.

Как мы видели выше, строковые данные в языке **Erlang** могут быть представлены при помощи двух типов данных: списки и битовые строки. Для создания экземпляров этих типов данных есть мощная и гибкая техника – конструирование списков (или List Comprehensions) и битовых строк (или Bit String Comprehensions) соответственно. Кратко напомним основную идею этой техники (более подробно см. **LXF147** и **LXF148**). В выражении конструирования используются две сущности: генераторы и фильтры. Первые генерируют список или битовую строку и предоставляют последовательный доступ к элементам или сегментам через специально объявленную переменную (в выражении конструирования могут одновременно появляться как генераторы списков, так и генераторы битовых строк). Вторые позволяют отфильтровывать ненужные элементы списков или сегменты битовых строк генераторов, пользуясь для доступа к ним объявленные в генераторах переменные. Как это работает? Среда выполнения **Erlang** последовательно генерирует все возможные комбинации элементов списков или сегментов битовых строк генераторов в порядке их появления в выражении конструирования, и для каждой комбинации применяются все объявленные фильтры; если для какой-либо комбинации все фильтры вернули **true**, то эта комбинация попадает в итоговое выражение в выражении конструирования.

> «Для создания экземпляров типов данных есть мощная техника.»

Давайте рассмотрим простой пример, чтобы показать, как это работает. Предположим, нам необходимо построить все возможные комбинации «согласная буква – гласная буква» для латинского алфавита (для простоты предположим, что нам достаточно таких комбинаций для заглавных букв), тогда данную задачу с использованием выражения конструирования списка можно решить следующим способом:

```
[[C] ++ [V] || V <- “AEIOU”, C <- lists:seq($B, $Z), not
lists:member(C, “AEIOU”)]].
```

Результатом будет список всех возможных таких пар: **[“BA”, “BE”, …, “ZU”]**.

Вспоминая в очередной раз, что строки есть списки или битовые строки, мы можем для работы со строками использовать BIF для работы со списками или битовыми строками, а также функции из модулей **lists** или **binary** (в зависимости от того, как представлены строковые данные). Т. к. об этих BIF и функциях уже говорилось (см. **LXF147** и **LXF148**), и будет говориться еще не раз в будущих статьях, то не будем останавливаться на них сейчас. Вместо этого поговорим о специфичных для строковых данных BIF и функциях из модулей **erlang** и **string**. Специфичные для строковых данных BIF и функции из модуля **erlang** – это функции по преобразованию объектов других типов в строковые данные и из строковых данных. Это BIF по преобразованию целых и действительных чисел в строку и обратно: **integer_to_list/1**, **float_to_list/1**, **list_to_integer/1**, **list_to_float/1**; это BIF по преобразованию атомов в строки и обратно: **atom_to_list/1**, **list_to_atom/1**, **list_to_existing_atom/1**; это BIF, позволяющие преобразовать атомы в строки в виде битовых строк и наоборот: **atom_to_binary/2**, **binary_to_atom/2**, **binary_to_existing_atom/2**; и, наконец, это вспомогательные BIF и функции из модуля **erlang** для преобразования в строку некоторых объектов, таких как идентификатор процесса Pid, для целей отладки. BIF, преобразующие строку в атом, содержат два семейства функций: **XXX_to_atom** и **XXX_to_existing_atom**. Разница между ними заключается в поведении, когда атом, в которой преобразовывают строку, не существует: в этом случае функции из первого семейства создают новый атом, а функции из второго семейства генерируют ошибку времени выполнения.

Поговорим про функции из модуля **string**. Эти функции позволяют решать часто встречающиеся задачи при работе со строками: преобразование букв в верхний (**string:to_upper/1**) и нижний регистры (**string:to_lower/1**), преобразование строки в целое (**string:to_integer/1**) и действительное число (**string:to_float/1**) со строкой-остатком, выделение подстроки (**string:sub_string/2**, **string:sub_string/3**), очистка строки от ненужных символов слева и/или справа (**string:strip/1**, **string:strip/2**, **string:strip/3**), разбиение строки на список лексем (**string:tokens/2**), соединение нескольких строк вместе (**string:join/2**) и еще множество других задач.

Рассмотрим наиболее интересные функции на примерах. Пусть мы хотим разбить строку на лексемы, и разделителями между лексемами может быть один из символов **“0”** и **“-”**. Тогда следующий вызов поможет решить интересующую нас задачу: **string:tokens(“10203-4”, “0-”)** и вернет нам следующий список лексем: **[“1”, “2”, “3”, “4”]**. Заметьте, что второй параметр функции **string:tokens/2** – это список символов разделителей, а, как мы помним, список символов – это всегда строка, составленная из этих символов. Теперь пусть перед нами стоит следующая задача: у нас есть строка, начинающаяся на некоторое целое число, мы хотим распарсить эту строку так, чтобы в результате получилось число и строка-остаток после распарсивания. Сделать это можно следующим образом: **{Number, Rest} = string:to_integer(“33*22”)**, при этом **Number** будет иметь распарсенное значение **33**, а остаток **Rest** – **“*22”**. Далее, пусть нам необходимо очистить строку спереди и сзади от «паразитных» символов – например, нулей. Сделать это можно так: **string:strip(“0001234500”, both, $0)**; результатом будет следующая строка **“12345”**. Функции преобразования букв в верхний (**string:to_upper/1**) и нижний регистры (**string:to_lower/1**) работают только для символов из кодировки **ISO-latin-1 (ISO8859-1)**. Это означает, что для строк в юникоде, например, для кириллицы нам придется писать преобразование в верхний и нижний регистр самим. Про функции из модуля **string** еще можно добавить следующее: несмотря на то, что модуль предназначается для работы именно со строками, большинство его функций прекрасно работают со списками любых целых чисел (а не только тех, которые представляют некоторые символы). Более того, некоторые функции прекрасно работают со списками любых объектов. Наиболее полезная в этом плане функция, с точки зрения автора, это функция разделения списка на лексемы **string:tokens/2**. Так, например, следующий вызов разделяет список атомов на лексемы по заданному списку разделителей: **string:tokens([cd, fd, aa, ab, de, aa, fd, ab], [aa, ab])**. В результате мы получаем следующий список лексем: **[[cd, fd], [de], [fd]]**.

Пойдем дальше. Мы помним, что строковые данные в языке **Erlang** могут быть представлены в виде списков и битовых строк. Ожидаемый формат хранения строковых данных в битовых стро-

ках в языке **Erlang** – **UTF-8**. Это означает, что функции библиотек ожидают, что строковые данные, которые передаются им в качестве параметров, будут представлены либо в виде списков, либо в виде битовых строк в формате **UTF-8** (это означает, что все сегменты битовой строки имеют спецификатор типа **utf8**).

В связи с этим возникает вопрос: а как можно преобразовать данные из одного формата хранения в другой? Задача подобного преобразования очень не простая (за исключением преобразования данных из одной однобайтовой кодировки в другую), но эта задача уже решена разработчиками языка **Erlang** в модуле **unicode**. Функции из этого модуля позволяют преобразовать строковые данные как в представление в виде списков, так и в представление в виде битовых строк в любой кодировке. Эти функции преобразования возвращают одно из трех возможных значений: преобразованный объект, кортеж **{error, Transforned, Rest}** в случае ошибки преобразования и кортеж **{incomplete, Transformed, Rest}**, когда данных для преобразования недостаточно (здесь **Transformed** – это объект, содержащий часть преобразованных данных, **Rest** – непреобразованные данные). Третий вариант возвращаемого значения позволяет нам преобразовывать данные порциями, начиная с непреобразованного остатка, если он есть (а это полезно, когда мы читаем данные порциями).

Давайте рассмотрим на нескольких примерах, как преобразовывать данные из одного формата в другой. Здесь мы преобразовываем данные, хранящиеся в битовой строке в кодировке **utf16**, в список: **unicode:characters_to_list(<<"12"/utf16>>, utf16)**; в результате мы получаем строку **"12"** (в представлении в виде списка). А теперь рассмотрим пример, когда мы преобразовываем данные порциями (по 2 байта): **unicode:characters_to_list(<<10, 16#d7>>, utf8)**; т. к. такую битовую строку мы полностью преобразовать не можем, то в результате получаем следующий кортеж **{incomplete, "\n", <<16#d7>>}**. Это означает, что когда мы продолжим преобразование, начать нам нужно будет с остатка после этого преобразования. И, наконец, рассмотрим преобразование из битовой строки в формате **utf8** в битовую строку в формате **utf16** с порядком следования байт **big** (от старшего к младшему): **unicode:characters_to_binary(<<"12"/utf8>>, utf8, {utf16, big})**. В результате мы получаем битовую строку **<<0, 49, 0,50>>**.

Модуль **unicode** содержит еще одну функциональность – работа с BOM (маркером порядка байт). Это кодовая точка **16#FEFF**, закодированная так же, как и все остальные строковые данные. В разных кодировках юникод с разным порядком байт эта кодовая точка выглядит по-разному (и занимает разный размер, но не более 4 байт). Если строковые данные (пришедшие из внешнего источника, т. к. иначе мы знаем кодировку и порядок следования байт) содержат первой кодовой точкой BOM, то, прочитав первые 4 байт строковых данных, мы можем извлечь BOM и понять кодировку юникод и порядок следования байт. Чтобы не делать этого вручную, призовем на помощь функцию **unicode:bom_to_encoding/1**, которая принимает битовую строку и возвращает кортеж **{Encoding, Length}**, где **Encoding** – это кодировка, а **Length** – длина BOM. Понятно, что длина исходной битовой строки должна быть не меньше 4 байт. Также в модуле **unicode** есть метод **unicode:encoding_to_bom/1**, позволяющий создать BOM для заданной кодировки и порядка следования байт.

Осталось поговорить о такой необходимой функциональности, как создание строки по шаблону (с использованием спецификатора формата) и извлечение данных из строки в соответствии с шаблоном (опять же с использованием спецификаторов формата). Эта (и не только эта) функциональность реализована в модуле **io_lib**. Для создания строки по шаблону используется метод **io_lib:format(Format, Data)**, где **Format** – строка-шаблон, **Data** – список с объектами, которые необходимо вывести в соответствии с шаблоном. По логике работы, эта функция – аналог функции **printf** языка C. Строка формата – это строка (которая может быть как списком, так и битовой строкой), которая помимо обычных данных содержит спецификаторы формата. Полное описание спецификаторов формата слишком большое (его можно найти в документации), поэтому ограничимся тем, что приведем спецификатор, использующийся в примере: **~p** – позволяющий вывести объект в стандартном виде. Эта функция подставляет и форматирует объекты из списка **Data** в строку формата в соответствии со спецификаторами типа. Если количество объектов или их тип не совпадают с тем, что ожидается, то генерируется ошибка времени выполнения. Если все хорошо, то возвращается так называемый «глубокий» список, т. е. список, элементами которого тоже могут быть списки. Чтобы преобразовать этот список в строку, можно использовать функцию **lists:flatten/1**.

Рассмотрим небольшой пример. Вызов **lists:flatten(io_lib:format("this object ~p is tuple", [{1, value}]))** вернет нам следующую отформатированную строку: **"this object {1,value} is tuple"**. Теперь займемся проблемой получения данных из строки в соответствии с шаблоном: для этого используются методы **io_lib:fread(Format, String)** и **io_lib:fread(Continue, String, Format)**, где **Format** – шаблон, **String** – строка из которой осуществляется получение, **Continue** – кусок строки, оставшийся с предыдущей порции получения данных. Первый метод просто получает данные из строки в соответствии с шаблоном, а второй позволяет получать эти данные порциями.

Логика работы этих функций аналогична функции **scanf** языка C, за исключением, пожалуй, порционного получения данных. Опять же, полное описание спецификаторов формата слишком большое (и его можно найти в документации), поэтому здесь мы приведем два спецификатора, которые используются в примерах: **~d** считывает целое число по основанию **10**, **~a** считывает строку и преобразует ее в атом. Приведем пару примеров. В первом примере мы просто получаем из строки число и атом: **io_lib:fread("~d-~a", "123-atom")**. В результате мы получаем следующий кортеж: **{done, [123, atom], []}**; это означает, что данные из строки успешно извлечены. Во втором примере мы извлекаем данные порциями:

```
{more, Cont} = io_lib:fread([], "123", "~d-~a").
io_lib:fread(Cont, "-atom\n", "~d-~a").
```

Получается кортеж **{done, {ok, [123, atom]}, []}**; это означает, что данные из строк (порций) успешно извлечены.

В данной статье мы рассмотрели работу со строковыми данными и увидели, что язык **Erlang** и библиотеки позволяют гибко и безопасно решать многочисленные задачи. Помимо того, что было показано, используя библиотеки языка **Erlang**, мы можем осуществлять ввод и вывод, работать с регулярными выражениями и т. д.; возможно, мы рассмотрим некоторые из этих возможностей в одной из статей. А в следующий раз мы затронем темы, которые не вошли из-за нехватки места в статьи про базовые сущности языка. LXF

Полезные заметки

- Список всех escape-последовательностей, поддерживаемых языком **Erlang**: http://www.erlang.org/doc/reference_manual/data_types.html#id73109
- Документация на функцию **io_lib:format/2**: http://www.erlang.org/doc/man/io_lib.html#format-2
- Документация на спецификаторы формата для функции **io_lib:format/2**: http://www.erlang.org/doc/man/io.html#fwrite-1
- Документация на функции **io_lib:fread/2** и **io_lib:fread/3**: http://www.erlang.org/doc/man/io_lib.html#fread-2 http://www.erlang.org/doc/man/io_lib.html#fread-3
- Документация на спецификаторы формата для функций **io_lib:fread/2** и **io_lib:fread/3**: http://www.erlang.org/doc/man/io.html#fread-3