使用 TextFormats 规格语言为现有数据设计文本表示

Giorgio Gonnella[1,2] ✉

[1]汉堡大学生物信息学中心（ZBH），汉堡 Bundesstrasse 43，邮编 20146 [2]哥廷根乔治-奥古斯特大学微生物学与遗传学研究所，哥廷根 Goldschmidtstr. 1，邮编 37077

arXiv:2301.13551v1 [cs.PL] 31 Jan 2023

TextFormats 是一套软件系统，可通过多种编程语言（C/C++、Python、Nim）和 Unix 命令行，高效、便捷地创建文本格式规范。要使用某种格式，必须先创建一份以 TextFormats 规格语言（TFSL）编写的规范。该规范为格式的每个部分定义数据类型。

TextFormats 规范中数据类型定义的语法基于文本表示。因此，该系统非常适合描述现有格式。不过，在创建用于表示现有数据的新文本格式时，用户可以根据值的类型和表示方式的选择，采用不同的定义。

本研究探讨了 TextFormats 规格语言中可能采用的定义语法，以用于创建标量值（例如字符串、数值、布尔值）和复合数据结构（例如数组、映射）的文本表示。研究结果以系统化的方式呈现，并针对可表示的不同类型的值逐一提供示例和使用建议。

TextFormats | 数据类型定义 | 解析器 | 文本表示 | 数据格式 | 领域特定语言 | 软件库 | 格式规范 | 文件格式 | 数据类型 | 文本格式 | 教程

联系邮箱： ***giorgio.gonnella@uni-goettingen.de***

TextFormats（Gonnella，2022）是一个用于定义文本格式的软件库和一组工具。尽管它最初是为表示生物信息学格式而开发的，但它是一款可应用于各种领域的通用软件。使用领域特定语言 TFSL（TextFormats 规格语言）定义格式后，可使用 TextFormats 规范解析该格式，也可使用面向多种编程语言（Nim、Python、C/C++）的 API，或通过提供的 Unix 命令行工具在脚本中写入该格式的数据。

在 TFSL 中，如表 1 所示，使用多种定义来描述不同类型数据的文本表示。这些定义不仅取决于值的类型，也取决于数据及其表示的其他特征。例如，它们取决于可能值的集合：对于字符串，单个值使用常量表示，较小的值集合使用值表示，而所有匹配某个正则表达式的字符串则使用正则表达式表示。有时，同一组字符串也可以使用不同类型的定义（例如 values、regex 和 regexes）。对于某些复合数据值，所用定义的类型取决于组成元素的语义和数据类型是否以及如何编码在文本表示中（例如 composed_of 与 labeled_list 或 tagged_list）。因此，探究这些不同定义类型在表示不同类型值时的使用场景颇有意义。

这很重 需要注意的是，两者并不总是一一对应 o-one : 所表示值的类型与其文本表示之间。例如，文本表示“I”可以表示字符串“I”（例如英语中的第一人称单数代词），也可以表示用罗马数字表示的整数值，而该整数通常用阿拉伯数字“1”表示。

本文系统地探讨了可在文本格式的各个部分中表示的不同类型的值，并报告了在规范中准确描述数据所需的相应 TextFormats 定义。针对每种值类型，本文都提供了 TFSL 定义的实际示例，以阐释所介绍的概念。

数据值的类型

在文本格式的语境中，可以表示各种类型的值。需要考虑的一个重要区别是：数据点在逻辑上代表的是不可再分的原子单位，还是可以分解为多个组件。如果属于前者，则称为标量值；如果属于后者，则称为复合值。

标量值是单一、不可再分的数据单位。此类别涵盖多种子类型，包括数值数据、分类数据、布尔值和字符串。数值数据包括整数和浮点值，用于表示数学数值。另一方面，分类数据会将每个数据点归入一个预定义的有限值集合中的

定义键	描述
适用于标量数据	
常量	单一、不变的值
值	一组值中的一个
正则表达式	匹配提供的正则表达式
正则表达式	匹配提供的正则表达式之一
整数	有符号整数，可选范围验证
无符号_整数	无符号整数，可选范围验证
浮点数	浮点值，可选范围验证
用于复合数据	
列表_	有序集合；元素数据类型和语义与位置无关
由_组成	有序集合；元素数据类型和语义取决于位置
带标签列表_	键值对集合；元素数据类型和语义取决于键
带标签的_列表	标签名/类型码/值三元组；语义取决于标签名，数据类型取决于类型码
多选项	
以下选项之一_	不同格式之一，由单独的定义描述

表 1. TextFormats 规格语言中可用的数据类型定义键，按待定义数据类型的类别（标量或复合型）划分，以及用于定义单个元素的多选项（之一_选项）。

可能值。布尔值是分类数据的一个示例，用于表示二元条件，其值可以为真或假。最后，字符串是由字符组成的序列，可用于表示各种值。

复合值由多个组件组成，每个组件可以是标量，也可以本身是复合值。复合数据类型提供了一种将数据组织成复杂结构的方式，从而能够表示结构化的多层信息。例如，复合数据类型可以由若干数值组成，每个数值代表不同的数据点；也可以由多个组件组成，每个组件代表数据的不同方面。复合数据类型的重要特点是，它们能够以有意义的方式表示多方面信息，以便进行解析和分析。因此，解析器必须了解每个组件的语义和数据类型，这些信息可以来自外部约定，也可以通过文本表示本身包含的元数据（例如标签）提供。

标量值的定义

数值。有几种专用于数值变量的数据类型定义。要定义数值，需考虑以下标准。首先，该值是否为整数。其次，可能值的范围是多少。第三，由于数值在许多上下文中（例如 C 语言）采用不同的内部表示方式，如果该值是整数，则还需考虑它是否为有符号数。

任意值。如果所有有效值都可以接受，只要它们处于使用该数据的编程语言或环境中的数据类型范围内，那么就使用预定义的整数、无符号_整数和浮点数。由于它们是预定义的，因此不需要编写定义，只需将关键字（例如 integer）插入规范中需要定义的位置即可。通常，这种做法用于复合数据类型或别名，如以下示例所示：

```
数据类型: num1: 无符号_整数
list1: {列表_的: 整数}
```

范围内的值。如果只接受给定范围内的值，可以使用 min 和/或 max：

```
数据类型: num2: 整数: {最小值: -1, 最大值: 1}num3: 无符号_整数: {最小值: 0, 最大值: 100}num4: 浮点数: {最小值: 0, 最大值: 1}
```

默认情况下，可接受值包含边界值，但可以使用 (min|max)_excluded 将其排除。这主要适用于浮点值，因为对于整数，只需将边界值增加或减少 1 即可。

```
datatypes:
  num5: float: {min: 0, max: 100,
                min_excluded: true,
                max_excluded: true}
```

值集合的元素。如果只需表示给定值集合中的元素，则使用值定义键，如以下示例所示：

```
数据类型:num6: {值: [1,2,3]}
```

这种定义的一个局限是，由于必须枚举所有可能的值，当集合中的元素数量过多时，其性能会非常差。

唯一可能的值。如果只允许接受单个值，则使用常量定义：

```
数据类型:
  num7: {常量:3}
```

罗马数字。TFSL 对非常规表示形式的数值支持非常有限。唯一可以轻松表示的情况是所有可能的值都可以枚举，因此其数量有限；否则性能会受到负面影响。在这种情况下，可以通过为每个元素指定到相应数值的转换来使用值定义。下面给出一个示例，用于表示从 1 到 3 的罗马数字：

```
数据类型: num8: 值:-
"I":1- "II":2-
"III":3
```

在其他情况下（例如，必须能够表示范围内的任意值或所有数值），无法直接在 TextFormats 中定义值的解析或格式化。因此，必须将数据类型定义为字符串（例如使用正则表达式），并由调用代码处理，如以下示例所示：

```
datatypes:
  num9: {regex: "^[MDCLXVI]+$"}
```

布尔变量。布尔值是只能包含两个值之一的变量：true 或 false。它们通常表示为一对字符串，例如 "T" 和 "F"，或 "0" 和 "1"。

单一表示。以下定义展示了如何为布尔变量定义一种数据类型，并为两个解码后的值（true 或 false）分别指定一个字符串值。这些表示可以通过值定义和值定义映射提供：

```
数据类型:
  boolean1: 值:  ["T": true, "F":
  false}
```

多种表示。有时，对于布尔值的两个值中的每一个，某种格式都接受多个字符串表示。在这种情况下，会使用值定义或正则表达式定义，并且必须指定规范表示：

```
数据类型: boolean2: 正则表达式: -"[Tt]
(rue)?": true- "[Ff](alse)?":
falsecanonical: {"T": true, "F":
false}
```

存在或缺失。在其他情况下，布尔变量的值会通过指定元素在文本表示中的存在或缺失来编码。在这种情况下，可以使用带有默认值（空键）的常量定义：

```
datatypes:
  boolean3:
    constant: {"$": true}
    empty: false
```

处理未定义状态。在某些情况下，可能存在三态变量状态。例如，变量可以取一个布尔值或一个特殊未定义值。在这种情况下，只需将更多选项添加到值定义映射中：

```
datatypes:
  boolean4:
    values: ["T": true, "F": false,
             "NA": null]
```

字符串值。本节介绍如何定义字符串变量中存储的值的数据类型。

任意字符串。如果格式的某个组成部分可以包含任意字符串，则可以使用预定义的字符串数据类型。别名或复合定义中会用到它：

```
datatypes:
  str1: string
  list2: {list_of: string}
```

分类标签。分类变量的值只能从有限个不同的值中取值，这些值可以逐一列出。上文已处理了某些分类值的情况，例如布尔值和具有固定数量可能值的数值变量。这些情况使用值定义来定义。如果类别存储为字符串，也使用同类定义：

```
datatypes:
  str2: {values: ["ABC", "DEF"]}
```

单一可能值。如果字符串变量只能包含一个值，则使用常量定义：

```
datatypes:
  str3: {constant: "XYZ"}
```

正则表达式。如果字符串必须匹配给定的正则表达式，则使用正则表达式类型的定义：

```
datatypes:
  str4: {regex: "[ABC]{1,3}"}
```

在某些情况下，将一个正则表达式拆分成多个分别定义的部分会更容易。无论使用单个还是多个正则表达式，可匹配的字符串范围都保持不变。使用多个正则表达式的优点是，单个正则表达式可能更易读，而且将字符串映射到给定值时，可以对每个部分进行不同处理：

数据类型:
  str5:

    正则表达式: - "[ABC]
    {1,3}"- "[DEF]{5,7}"

空字符串。如果也需要处理空字符串，可以使用 empty 提供其值（此选项适用于任何类型的定义）。该选项具有最高优先级，因此，即使提供了一个也能匹配空字符串的正则表达式，空字符串的情况也会按照空选项中的定义处理。例如，在以下情况下，空字符串会得到解码值 0:

```
datatypes:
  str6: {regex: "\d*", empty: "0"}
```

结构化字符串。在某些情况下，尽管某个值应解码为字符串，而不再解析为更小的元素，但它仍具有内部结构。在这种情况下，创建一个定义（例如下文所述的复合数据类型定义）可能很有用，然后使用 as_string: true 选项告知 TextFormats，该定义仅用于验证，而不用于解析。

例如，下面这个字符串定义（包含无符号整数，并以“.”分隔）使用了相对复杂的正则表达式:

```
datatypes:
  str7:
    regex: "(0|[1-9][0-9]*)
            (\.(0|[1-9][0-9]*))*"
```

以下定义与之等效，但可读性更高:

```
数据类型:
  str8: 列表_的: {正则表达式: 无符号_整数}
  以_分隔: "."最小值_长度: 1作为_字符串:
  true
```

首字母缩略词。默认情况下，字符串会直接编码为其自身，也就是说，解析过程涉及验证，但不会修改值本身。在某些情况下，字符串表示中应当使用与解码后的值不同的字符串：例如，我们可能希望展开一个首字母缩略词。为此，可以使用解码映射:

```
数据类型: str9: 值: - "USA": "美国
"- "UK": "英国"
```

如果多个编码值解码后得到相同的解码值（对于正则表达式来说总是如此），则必须指定规范的编码形式，以便编码器知道应使用哪一种形式。例如:

```
数据类型: str10: 正则表达式: - "U(SA?|
sa)": "United States"- "U[Kk]": "
United Kingdom"规范形式: - "USA": "
UnitedStates"- "UK": "
UnitedKingdom"
```

复合值的定义

复合数据有几种不同的类型。这里我们区分以下 3 种情况，分别处理:

1. 有序元素列表，在语义上彼此等价；2. 字典或映射，即开放的关联数据结构，其中不同元素可能具有不同的语义；3. 对象或结构体，通常包含不同且语义各异的属性

值得注意的是，这几类复合值的数据类型名称取决于上下文（例如编程语言）以及底层数据结构，也就是数据在内存中的存储方式。

TextFormats 中的复合数据兼容性。复合值可以将其他复合值作为组件。这种层次结构可以用树来表示，其深度可能没有限制，并可在文本表示中通过缩进或嵌套的括号对来标记。然而，要与当前版本的 TextFormats 实现兼容，某种格式必须表示一种正则语言（由外部库处理的格式部分除外；目前这指的是 JSON）。因此，复合数据类型的定义树必须在定义时确定，而且 TFSL 不允许循环定义。

列表、数组、序列、集合。这里我们讨论复合数据值，其中元素是有序的（顺序可能有意义，但不一定有意义），但所有元素在语义上都被视为等价，并且元素数据类型和可表示值的集合不依赖于它们在列表中的位置。

如果顺序很重要，复合值通常会根据底层数据结构和上下文（例如编程语言）存储在数组或列表数据结构（例如链表）中，因此称为列表（例如 Python）、数组（例如 C、Python）或序列（例如 Nim、YAML）。在 TextFormats 中，这类值使用 list_of 数据类型定义进行描述。

当各元素的语义彼此相同时，TextFormats 中复合值的定义使用键列表_，如下例所示：

```
datatypes:
  list3:
    list_of: {regex: [A-Z]}
```

元素分隔符。通常，元素之间的分隔字符串可使单个元素得以解析；该字符串不会出现在元素本身中。可通过选项 splitted_by 指定：

```
datatypes:
  list4:
    list_of: unsigned_integer
    splitted_by: ","
```

分隔符转义。不过，在某些情况下，分隔字符串也可能出现在元素本身中，例如通过转义来处理。此时，定义中会使用分隔符选项，并且例如可以使用正则表达式来定义元素：

```
datatypes:
  list5:
    list_of:
      regex: "(\\\:|[A-Za-z0-9 _])*"
        # allows : escaped by \
    separator: ":"
```

例如，根据上述定义，字符串
元素 1:元素2:元素_3:元素\:\:4
将被解析为四个元素：elem1、elem2、elem_3 和 elem\:\:4。

定长元素。如果列表中的元素长度都相同，通常不需要分隔符。不过，如果存在分隔符，它也可以出现在元素文本本身中，因为不存在混淆的风险。在这种情况下，可使用分隔符选项，例如：

数据类型：列表6：列表：_正则表达式：
"[: 0-9]{3}"分隔符：
":"

根据前面的定义，每个元素的大小为 3，因此分隔符可能包含在元素中也无关紧要。例如，字符串 001:0:::002:2:1:112:::: 会被解析为六个元素：001、0::、002、2:1、112 和 :::。

包围字符串。在某些情况下，列表的第一个元素之前和/或最后一个元素之后会出现常量字符串，例如左括号和右括号：

```
datatypes:
  list7:
    list_of: unsigned_integer
    splitted_by: ","
    prefix: "("
    suffix: ")"
```

此定义允许解析类似 (1,2,3,4) 的字符串表示。

元素数量。在某些情况下，列表的元素数量存在最小值和/或最大值，或者是固定数量。可以使用验证规则强制执行这一点，如以下示例所示：

```
datatypes:
  list8:   # e.g. 0;-1;32
    list_of: integer
    splitted_by: ";"
    length: 3
  list9:
    list_of: integer
    splitted_by: ";"
    min_length: 5
    max_length: 7
```

空列表。同样支持空列表。如果存在前缀和后缀，则可以识别空列表，无需特殊处理。如果没有包围字符串，则空列表的表示是空字符串。这种情况可以通过使用空选项来处理：

```
datatypes:
  list9:
    list_of: {regex: "[A-Z]"}
    empty: []
``
```

预定义表示。通过使用解码映射和/或默认解码值（见下文），可以将给定的字符串解码为预定义列表。如果同一值有多个表示形式，则必须通过规范选项指定编码时应使用哪一种，如下例所示：

```
datatypes:
  list10:
    values:
      "a": ["a"]
      "1a": ["a"]
      "2a": ["a", "a"]
      "3a": ["a", "a", "a"]
    empty: []
    canonical: {"1a": ["a"]}
```

集合与多重集。有时，如果元素的顺序并不重要，就可以使用另一种集合类型，并采用不同类型的数据结构在内存中表示它们，例如哈希表。这类集合可用作 Python 中的集合以及 Nim 中的哈希集合。

TextFormats 不会对集合进行特殊处理。这意味着集合中的元素会被视为列表。因此，忽略元素顺序的等价操作必须在外部实现。同样，集合元素的唯一性（相对于多重集而言）也必须在外部验证。

异构列表。one_of 定义可以与 list_of 定义结合使用，以实现包含不同类型元素的列表；这些元素的类型不依赖于其位置顺序，也不会通过键或类型代码显式标注。相反，元素本身的格式会揭示其类型。例如，下面定义了一个列表，其中包含整数或单个大写字符：

```
数据类型: list11: 列表: _
二选一: _- 整数- 正则表达式:
"[A-Z]"拆分_依据: ","
```

一个可由前述定义处理的字符串示例为：1,-3,A,5,B,-2。

映射、字典、关联数组。另一类复合值中，元素的语义和数据类型可能各不相同；不同实例可能包含或不包含某些元素，有时还会包含多个相同类型的元素。这类复合值通常存储在开放式关联数据结构中，这些结构的名称和底层数据结构各不相同，例如字典（Python）、表（Nim）、哈希表（C）、对象（JSON）或映射（YAML）。

在文本格式中，这类数据可以用不同方式表示。例如，语义和数据类型可以在文本表示中明确指定，也可以由文本格式本身隐式表达。根据具体情况，TextFormats 中要使用的定义类型也有所不同（例如，含有 one_of 元素的 list_of、带标签列表或带标记列表）。

按格式确定语义。如果列表元素的语义由格式决定，则可以给出列表_定义，其中元素使用一_定义来定义。这与上文“异构列表”段落中所举的情况相同。在这种实践中并不常见的情况下，解析结果以及要传递给编码函数的数据都必须是列表。因此，需要进行一些外部预处理或后处理，将数据转换为映射，或从映射转换回来。

键值对。在许多情况下，集合包含不同类型的元素，并且每个元素的语义都与该元素的值一同明确给出。因此，每个元素都以由键和值组成的元组形式呈现。虽然这种情况下也可以使用列表_，但如果可能的键集合及其对应的数据类型是预先已知的，TextFormats 还提供了一种专门的列表定义。

在这种情况下，会使用带标签列表定义，如以下示例所示：

```
数据类型: map1: 带标签列表: _等
级: 无符号_整数名称: 字符串拆分
_依据: ";"
```

每个键的可选键集合及其值的数据类型，都以映射形式列在标记为_列表的键下。可以指定一个内部分隔字符串_，用于分隔键和值（默认为 :）。内部分隔符不能为空，也不能出现在键中，但可以出现在值中。采用键值列表的格式通常都满足这一条件。

单实例键。默认情况下，名称可以在列表中出现多次。因此，元素值在解码后的值中总是以列表形式给出。在某些情况下，所有或部分名称只能出现一次。可以通过将这些名称列在 single 键下强制执行此限制：

```
datatypes:
  map2:
    labeled_list:
      rank: unsigned_integer
      name: string
    splitted_by: ";"
    single: [rank, name]
```

必需键。此外，默认情况下，某些名称可以不出现在元素集合中。如果某些名称必须出现，则将它们列在 required 键下：

```
datatypes:
  map3:
    labeled_list:
      rank: unsigned_integer
      name: string
    splitted_by: ";"
    internal_separator: "="
    required: [name]
```

SAM 风格标签。在某些情况下，集合中的每个值都附带名称和类型代码，即以值/名称/类型代码三元组的形式表示。一个典型的例子是 SAM 风格标签，它们常见于近期的生物信息学格式中，例如 GFA（GFA 格式规范工作组，2016年，2018年）和

VCF（Danecek 等，2011），这些标签最初是在 SAM 格式中定义的（Li 等，2009）。

与键值情形的区别在于，名称定义了值的语义，但并非所有名称（不同于带标签值列表）都必须预先定义。由于名称不一定是预定义的，因此必须明确给出类型，所以文本表示中会包含类型代码。每个类型代码都与一个数据类型定义相关联。

对于这些情形，使用 tagged_list 定义键，在该键下，通过映射在 tagged_listkey 下给出所有可用的数据类型代码及其关联的数据类型定义。

此处给出了一个带标签列表定义的示例：

```
数据类型:map4:tagged_
list:i: 整数f: 浮点数标签名:
"[A-Z]"internal_分隔符: "."
splitted_by: ";"
```

上述定义例如可以处理字符串表示 A.i.12;B.f.1.3，并将其解析为两个元素：A 的值为 12，B 的值为 1.3。

广义标签。有效名称及其格式由正则表达式指定。内部分隔符键的默认值为冒号（:），且必须是非空字符串。它不能出现在标签名和类型代码中（但可以出现在值中）。这一限制是合理的，例如 SAM 标签中就符合此限制。

预定义表示。通过使用解码映射和/或默认解码值（见下文），可以将给定字符串解码为预定义的映射/字典。如果一个数据值有多个字符串表示，则必须指定其中的规范表示，以便用于编码。

数据类型：map5：值："ax": {"a":"x", "b": "y"} "a": {"a":"x", "b":"y"} "ay": {"a","y", " b":"y"} "bx": {"a":"x", "b":"x"} 规范："ax": {"a":"x", "b":"y"} "ay": {"a","y", "b":"y"} " bx": {"a":"x", "b":"x"} 空值：{"a":"z","b": " z"}

隐式条目。在某些情况下，解码后的字典应包含给定的常量键值对，而该键值对并未显式编码。这些条目通过以下方式指定：

隐式映射，可用于 composed_of、labeled_list 和 tagged_list 定义：

```
数据类型：map6：由以下内容组成_：-名
称：字符串- 拷贝数：无符号_整数按_拆
分：","隐式：{类型："rRNA"}
```

例如，此定义可以处理字符串表示形式 "16S,2"，该字符串会被解析为包含 3 个键的映射 {"name": "16S", copies: 2, type: "rRNA"}。

对象、结构体。本节将处理值的集合，其中每个实例都包含相同的元素集合（可能存在例外），用于表示数据的不同方面。每个元素都有其自身的数据类型和语义。

这类复合值通过结构体（C、Python）或类的实例（Python、Nim）表示。其他可能的表示形式是上一段中提到的形式（映射、哈希表），并最终添加验证，以确保所有且仅有正确的元素存在。

在 TextFormats 中，这类数据的描述通过 composed_of 定义完成。在 definition 键下给出一个元组列表，每个元组分别包含名称和定义。请注意，这是一个列表（因此在 YAML 中使用 -），而不是映射，因为元素的顺序很重要，它定义了每个元素分别对应什么：

```
d数据类型：obj1：由_组成：-第一：无符号_整
  数- 第二：浮点数- 第三：{正则表达式："[
  A-Za-z0-9]"}拆分_通过：" "
```

包围字符串。对于列表，composed_of 定义可以包含前缀和/或后缀选项，用于在第一个元素之前和/或最后一个元素之后定义包围字符串（例如圆括号）。

元素分隔符。splitted_by 和分隔符选项用于描述如何分隔复合值中的单个元素。它们的用法与列表一节中已经说明的相同，即，splitted_by 用于分隔符不会出现在元素文本中的情况，而 separator 则用于其他情况，例如转义分隔符可能出现在元素中，或者元素大小可通过其格式识别的情况，如定长元素。

多个分隔符。在某些情况下，不同的元素对之间会使用不同的分隔符。此时，可以将它们指定为额外的常量元素，并在

解码后的字典中使用 hide_constants 选项将其隐藏，如以下示例所示：

数据类型：obj2：由_组成：- x：无符号_整数 - sep1：{常量：";"} - y：浮点数 - sep2：{常量："|"} - z：{正则表达式："[A-Za-z]"} 隐藏常量：true_

由前一个定义解析的字符串表示示例为 1;2.0 |A。由此得到的映射包含三个元素，因为常量分隔符仅用于解析：{x:1, y:2.0, z:A}。

可选分隔元素。某些元素可以是可选的，即有时会从元素序列中省略。如果序列由非空分隔符字符串分隔，则可以通过该分隔符的存在来识别空元素。在这种情况下，元素定义中会使用空选项，如以下示例所示：

```
d数据类型: obj3: 由_组成: - first: {值:
  [1,2], 空: 0}- second: {值:  [A, B], 空:
 C}以_分隔: ";"
```

可选尾随元素。在某些情况下，文本表示开头的指定数量的元素可能是必需的，而后续元素则可以出现或不出现（仅当它们之后的元素出现时，这些元素才是必需的）。在这种情况下，可使用 required 选项（必需元素的数量）：

```
datatypes:obj4:composed_of:- first: {值:
  [1,2], empty:0}- second: {值:  [A,B],
 empty:C}splitted_by: ";"required: 1
```

可选内部元素。在这种情况下，某个内部元素可能缺失（其关联分隔符也一并缺失，或者不使用分隔符），但不会产生歧义，例如，因为序列中的下一个元素具有一种可与可选元素区分的类型，或者因为元素总数会随可选元素是否存在而变化。

在这种情况下，用户必须使用 one_of 定义，为该结构提供多个备选定义（即包含和不包含可选元素的定义）。此外，为了让对象或结构体的所有实例都具有相同的一组值，可以使用隐式选项。

```
datatypes:
  obj5:
    one_of:
    - composed_of:
        - name: string
      - expressed:
          values: {"+": true, "-": false}
     splitted_by: ","
     implicit: {"copies": 1}
    - composed_of:
      - name: string
      - copies: unsigned_integer
      - expressed:
          values: {"+": true, "-": false}
      splitted_by: ","
```

上述定义会将字符串表示形式“X,+”解析为映射 { name: X, expressed: true, copies: 1}，而“X,2,+”则会被解析为 {name: X, expressed: true, copies: 2}。

联合体。在某些情况下，格式中的一个元素可以用多种不同方式表示。在 Python 等动态类型语言中，任何变量都可以存储此类值。在 C 中，此类值可以存储为联合体；在 Nim 中，则可以存储为变体对象。

在 TextFormats 中，此类值的类型可以通过定义 one_of 类型来指定。例如，以下定义允许将无符号整数值表示为数字（当它为 >=1 时），否则表示为浮点数：

```
数据: yp
  num10:
    one_of:
      - unsigned_integer: {min: 1}
      - float
```

请注意，one_of 键的内容是一个 YAML 列表；列表中元素的顺序决定了定义的优先顺序（使用第一个适用的定义）。

结论

本文分析了如何按照 TextFormats 库的规范，在文本格式中表示不同类型的数据。研究表明，编程语言（如 C 和 Python）中可使用的大多数值类型，也可以用 TextFormats 表示。

在 TextFormats 规范中，同一种值有时会使用不同类型的数据类型定义来表示。标量值和复合值都是如此。例如，在上述示例中，布尔值有时使用 values 类型的定义表示，有时则使用正则表达式，甚至常量，具体取决于它们在格式中的表示方式。在复合值的示例中，例如带标签元素集合，有时会使用 list_of 表示，但在其他情况下

则使用专门的列表定义 keys tagged_list 和带标签列表。_

这是因为，TextFormat 中用于数据类型定义的语法侧重于文本表示，而非所表示值的类型。基于值的类型对定义类型进行系统综述，在定义新格式时尤其有用；它也补充了库文档中包含的 TFSL 语法手册。后者在为现有格式编写规范时更为有用。

参考资料

Giorgio Gonnella。Textformats：简化生物信息学中文本格式的定义与解析。PLOS ONE，17(5):1–17，2022年5月。doi: 10.1371/journal.pone.0268910。URL https://doi.org/10.1371/journal.pone.0268910。

GFA 格式规范工作组。GFA 格式规范，2016年。URL http://gfa-spec.github.io/GFA-spec/GFA1.html。

GFA 格式规范工作组。图形化片段组装（GFA）2.0 格式规范，2018年。URL http://gfa-spec.github.io/GFA-spec/GFA2.html。

Petr Danecek、Adam Auton、Goncalo Abecasis、Cornelis A. Albers、Eric Banks、Mark A. DePristo、Robert E. Hand- saker、Gerton Lunter、GaborT. Marth、Stephen T. Sherry, Gilean McVean、Richard Durbin 和千人基因组计划分析组。变异调用格式与 VCFtools。Bioinformatics，27(15):2156–2158，2011年6月。ISSN 1367-4803。doi: 10.1093/bioinformatics/btr330。URL https://doi.org/10.1093/bioinformatics/btr330。

Heng Li、Bob Handsaker、Alec Wysoker、Tim Fennell、Jue Ruan、Nils Homer、Gabor Marth、Goncalo Abecasis、Richard Durbin，以及 1000 Genome Project Data Processing Subgroup。序列比对/映射格式与 SAMtools。Bioinformatics，25(16):2078–2079，2009年6月。ISSN 1367-4803。doi: 10.1093/bioinformatics/btp352。URL https://doi.org/10.1093/bioinformatics/btp352。

致谢

Giorgio Gonnella 获得了德国研究联合会资助 GO 3192/1-1 的支持，资助项目为“通过收集和验证关联规则，自动表征微生物基因组和宏基因组”。资助方未参与研究设计、数据收集与分析、发表决定或论文撰写。

作者贡献

这些贡献遵循贡献者角色分类法指南：https://casrai.org/credit/。概念化：G.G.；数据整理：G.G.；正式分析：G.G.；经费获取：G.G.；研究：G.G.；方法学：G.G.；项目管理：G.G.；资源：G.G.；软件：G.G.；监督：G.G.；验证：G.G.；可视化：G.G.；撰写——初稿：G.G.；撰写——审阅与编辑：G.G。

竞争性财务利益

作者声明不存在竞争性财务利益。