AI+Coding Kata

AI in Coding helps a lot with high-level tasks like prototyping, reasoning and finding bugs.

Please use your favourite tools (no limitations) to implement as much of this spec as possible in a language of your choice. This should be a parser that can parse any document in this spec. Imagine that your team will have to support this code for a few years, so you want to do a thorough job here.

There are no limits here. If you spot something unusual - use your best judgement.

BizDocumentAI Spec

Let's define a simple document format that could describe a contract, procedure, or any other business document in a structured way. It may be used to load this business data into AI Assistants (like in Enterprise RAG Challenge). We'll work with the documents.

Our documents will consist of **blocks**. A block is a logical piece of text (like a paragraph). It can optionally have a **head**, **number**, and **body**. A block's body can contain:

- Another block
- Text

- A list
- A dictionary

Blocks can contain heterogeneous content—texts, other blocks, dictionaries, etc. Lists can contain only similar block items that also have a **number**.

Document Layout

The document below describes a simple text format that can be deterministically parsed into JSON objects. This document is also a **test suite**! Code admonitions always come in pairs: first **input** and then **json**.

When the parser is implemented, parsed input should always produce output that is structurally similar to the expected JSON. The headline before the code blocks is the name of the text.

Python Data Structures

Below is an example of how you might structure your data models in Python using Pydantic:

```
from typing import List, Optional, Union, Dict, Literal
from pydantic import BaseModel, Field

# This type alias helps with readability and forward references.
ContentNode = Union[str, "Block", "ListBlock", "Dictionary"]

class Dictionary(BaseModel):
    """
    A distinct dictionary structure for key-value pairs.
    """
    kind: Literal["dict"]
    items: Dict[str, str] = Field(default_factory=dict)
```

```
class Block(BaseModel):
    """
    A general-purpose container for a 'section' or item.

    - 'number' can store a section number (e.g., "5", "5.1") if applicable.
    - 'head' is an optional heading for the block.
    - 'body' can hold any mix of strings, sub-blocks, dictionaries, or lists.
    """
    kind: Literal["block"]
    number: Optional[str] = None
    head: Optional[str] = None
    body: List[ContentNode] = Field(default_factory=list)

class ListBlock(BaseModel):
    """
    A container for a list of items, each item being a 'Block'.
    """
    kind: Literal["list"]
    items: List[Block] = Field(default_factory=list)

# Important for forward references within union types
Block.model_rebuild()
```

Specifications

Empty text

Empty text results in an empty document block.

Input:

(there is no content)

JSON:

```
{
  "kind": "block"
}
```

Body

Plain text goes into the block body straight away. Different paragraphs are separated by new lines.

Input:

```
First paragraph.
Second paragraph.
```

JSON:

```
{
  "kind": "block",
  "body": [
    "First paragraph.",
    "Second paragraph."
  ]
}
```

Note that we strip and skip empty lines!

Input:

```
First paragraph.

Second paragraph.
```

(An empty line in between)

JSON:

```
{
  "kind": "block",
  "body": [
    "First paragraph.",
    "Second paragraph."
  ]
}
```

Head

Text marked with `<head>` goes directly into the head of the current block.

Input:

```
<head>Test Document</head>
Content
```

JSON:

```
{
  "kind": "block",
  "head": "Test Document",
  "body": [
    "Content"
  ]
}
```

Blocks

You've seen that the document is parsed into a **root block**. But everything is a block, and blocks can be nested explicitly:

Input:

```
<head>AI Coding Kata</head>
Let's get started with the kata
<block>
<head>Preface</head>
Here is a little story
</block>
```

JSON:

```
{
  "kind": "block",
  "head": "AI Coding Kata",
  "body": [
    "Let's get started with the kata",
    {
      "kind": "block",
      "head": "Preface",
```

```
      "body": [
        "Here is a little story"
      ]
    }
  ]
}
```

Dictionaries

Dictionaries are used to capture key-value pairs. By default, they are separated by `:`.

Input:

```
<dict sep=":">
Key One: Value One
Key Two: Value Two
Key Three: Value Three
</dict>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "dict",
      "items": {
        "Key One": "Value One",
        "Key Two": "Value Two",
        "Key Three": "Value Three"
      }
    }
```

```
  ]
}
```

We can also have a non-standard separator and empty values:

Input:

```
<dict sep="-">
Title - AI Coding - for TAT
Kata Number -
</dict>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "dict",
      "items": {
        "Title": "AI Coding - for TAT",
        "Kata Number": ""
      }
    }
  ]
}
```

Lists

Lists are very important! By default, each non-empty line is a list item. They go inside the root block.

There are multiple kinds:

- `.` for **ordered** lists that are dot-separated
- `*` for **bulleted** lists

Note that the list item's text goes into `head` and the item number goes into `number`.

Ordered Lists

Input:

```
<list kind=".">

1. First
2. Second
</list>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        { "kind": "block", "number": "1.", "head": "First" },
        { "kind": "block", "number": "2.", "head": "Second" }
      ]
    }
  ]
}
```

As a convenience, **nested lists** are automatically detected:

Input:

```
<list kind=".">

1. First
2. Second
2.1. Subitem 1
2.2. Subitem 2
</list>
```

JSON:


```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        {
          "kind": "block",
          "number": "1.",
          "head": "First"
        },
        {
          "kind": "block",
          "number": "2.",
          "head": "Second",
          "body": [
            {
              "kind": "list",
              "items": [
                { "kind": "block", "number": "2.1.", "head": "Subitem 1" },
                { "kind": "block", "number": "2.2.", "head": "Subitem 2" }
              ]
            }
          ]
```

```
        }
      ]
    }
  ]
}
```

Unordered lists

We can have unordered lists too:

Input:

```
<list kind="*">
• First
• Second
• Third
</list>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        { "kind": "block", "number": "•", "head": "First" },
        { "kind": "block", "number": "•", "head": "Second" },
        { "kind": "block", "number": "•", "head": "Third" }
      ]
    }
  ]
}
```

And nesting can be done with `"o"`:

Input:

```
<list kind="*">
• First
    o Subitem
• Second
• Third
</list>
```

JSON:


```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        {
          "kind": "block",
          "number": "•",
          "head": "First",
          "body": [
            {
              "kind": "list",
              "items": [
                {
                  "kind": "block",
                  "number": "o",
                  "head": "Subitem"
                }
              ]
            }
          ]
        },
        {
          "kind": "block",
```

```
          "number": "•",
          "head": "Second"
        },
        {
          "kind": "block",
          "number": "•",
          "head": "Third"
        }
      ]
    }
  ]
}
```

Mixed lists

We can mix lists, but we need to designate different types separately with tags.

Input:

```
<list kind=".">

1. Beginning
2. Main
2.1. Subsection
<list kind="*">
* Bullet 1
* Bullet 2
</list>
3. Ending
</list>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        { "kind": "block", "number": "1.", "head": "Beginning" },
        {
          "kind": "block",
          "number": "2.",
          "head": "Main",
          "body": [
            {
              "kind": "list",
              "items": [
                { "kind": "block", "number": "*", "head": "Bullet 1" },
                { "kind": "block", "number": "*", "head": "Bullet 2" }
              ]
            }
          ]
        },

        { "kind": "block", "number": "3.", "head": "Ending" }
      ]
    }
  ]
}
```


Lists with content

Lists can also have additional content. If something in the current list doesn't match the prefix, then it is treated as a **block body**:

Input:

```
<list kind=".">

1. First
First body
2. Second
Some more text
<dict sep=":">
Key: Value
Another Key: Another Value
</dict>
</list>
```

JSON:

```
{
  "kind": "block",
  "body": [
    {
      "kind": "list",
      "items": [
        {
          "kind": "block",
          "number": "1.",
          "head": "First",
          "body": [
            "First body"
          ]
        },
        {
          "kind": "block",
          "number": "2.",
          "head": "Second",
          "body": [
            "Some more text",
            {
              "kind": "dict",
              "items": {
                "Key": "Value",
```

```
                "Another Key": "Another Value"
              }
            }
          ]
        }
      ]
    }
  }
```

Published: April 06, 2025.