GBNF Guide

GBNF (GGML BNF) is a format for defining formal grammars to constrain model outputs in `llama.cpp`. For example, you can use it to force the model to generate valid JSON, or speak only in emojis. GBNF grammars are supported in various ways in `examples/main` and `examples/server`.

GBNF（GGML BNF）是一个用于定义正式语法的格式，这种格式可以用来限制`llama.cpp`中的模型输出。正式语法是一种系统，用来描述一种语言中有效字符串的结构和组成，广泛应用于编程语言和数据格式的设计中。GBNF是基于BNF（巴科斯-诺尔范式）的一种扩展，BNF是一种用于表示上下文无关文法的标记法，常用于描述计算机语言的语法。

在`llama.cpp`的实现中，GBNF可以用于定义模型输出的约束条件。例如，你可以使用GBNF来确保模型输出是有效的JSON格式，或者仅使用表情符号进行交流。这种格式支持的实现可以在`examples/main`和`examples/server`中找到。

- **应用场景**：在需要确保生成的文本满足特定语法规则的场合（如编程语法检查、自动生成代码片段等），使用GBNF可以提前定义好规则，使得模型的输出自动符合这些规则。
- **数学原理**：GBNF通过一系列的产生式规则来定义，每个规则指定了一个符号的可能替换方式。这些规则共同定义了所有有效字符串的结构，确保输出只包含符合预定义语法的结果。
- **计算机实现**：在`llama.cpp`中，你可以通过修改模型的配置或参数来指定使用特定的GBNF语法。在解析和生成文本时，模型会参考这些语法规则，确保所有输出都严格遵守定义的结构和约束。

在具体实施中，GBNF的使用涉及到将这些语法规则编码到`llama.cpp`的处理逻辑中，可能需要额外的语法解析器支持，以确保运行时的文本生成严格符合预设的语法模式。

Background

Backus-Naur Form (BNF) is a notation for describing the syntax of formal languages like programming languages, file formats, and protocols. GBNF is an extension of BNF that primarily adds a few modern regex-like features.

巴科斯-诺尔范式（BNF，Backus-Naur Form）是一种用于描述形式语言语法的符号系统。形式语言包括编程语言、文件格式和通信协议等。BNF通过简洁的规则集描述语言的结构，使得理解和实现这些语言的解析器变得更加直观和简单。这些规则通常由非终结符、终结符、替代（|）、连接（空格）和特殊符号（如括号）构成。

BNF 的组成

- **终结符**：语言的基本符号，不能再被分解，如编程语言中的具体字符或关键字。
- **非终结符**：由一个或多个终结符和/或其他非终结符组合成的复杂结构。
- **产生式**：规则，形式为“<非终结符> ::= 表达式”，表示非终结符可以由表达式中的元素构成。

例如，一个简单的BNF规则来描述简单的算术表达式可能是这样的：

```
<expression> ::= <number> | <expression> "+" <expression>
<number> ::= "1" | "2" | "3" | ... | "9"
```

这表明一个表达式可以是一个数字，或者是两个表达式通过加号连接的组合。

GBNF：BNF的扩展

GBNF（Generalized Backus-Naur Form）是BNF的一个扩展，增加了类似于现代正则表达式的特征。这些特征使得GBNF在表达更复杂的语法规则时更加灵活和强大。GBNF的主要增强包括：

- **重复和可选元素**：允许元素出现多次或可选出现，类似于正则表达式中的`*`（零次或多次）、`+`（一次或多次）和`?`（零次或一次）。
- **字符类**：可以定义一个字符的集合，让任何属于这个集合的字符都能匹配，如`[a-zA-Z]`匹配所有大小写字母。

这些增强使得GBNF比传统的BNF更适合描述复杂的文本模式，如复杂的数据文件格式或者复杂的编程语言特征。

应用实例

例如，如果我们要使用GBNF描述一个简单的日期格式（如`YYYY-MM-DD`），我们可能会写出这样的规则：

```
<date> ::= <year> "-" <month> "-" <day>
<year> ::= [0-9]{4}
<month> ::= "0"[1-9] | "1"[0-2]
<day> ::= "0"[1-9] | [12][0-9] | "3"[01]
```

这里使用了类似正则表达式的语法来指定数字的重复次数和范围，使得描述更精确且简洁。总结来说，BNF及其扩展GBNF为我们提供了一种强大的工具，可以精确且清晰地描述语言的语法结构，这对于开发相关的软件工具和系统至关重要。

Basics

In GBNF, we define *production rules* that specify how a *non-terminal* (rule name) can be replaced with sequences of *terminals* (characters, specifically Unicode code points) and other non-terminals. The basic format of a production rule is `nonterminal ::= sequence...`.

在GBNF中，我们定义了所谓的“产生式规则”（production rules），这些规则指定了一个非终结符（non-terminal，通常是一个规则名）如何可以被替换为终结符（terminals，即具体的字符，特指Unicode编码点）和其他非终结符的序列。产生式规则的基本格式是`nonterminal ::= sequence...`。这里，我们一步步详细解析这个定义和它的组成部分。

非终结符（Non-terminal）

非终结符是GBNF语法中的一个核心概念。它代表了语法中的一个变量或者占位符，它不直接对应于最终的语言输入，而是定义了一个可以进一步替换和展开的规则。非终结符通常用尖括号`< >`括起来标示，或者直接以名称出现，取决于具体语法的定义。

终结符（Terminal）

终结符是另一个重要的概念，它们是构成语言的基本单位。在GBNF中，终结符通常是指Unicode的码点，这些码点直接对应于可输入的字符。在产生式规则中，终结符是语法分析的终点，不会进一步被替换或展开。

产生式规则（Production Rule）

产生式规则是GBNF中描述如何从非终结符到终结符和/或其他非终结符转换的规则。每个规则的基本形式是`nonterminal ::= sequence...`，这里的`::=`可以被理解为“被定义为”。右侧的`sequence`可以包含一个或多个终结符和非终结符，描述了当解析到左侧的非终结符时，应该如何生成或解释该非终结符。

示例

例如，考虑一个定义简单算术表达式的GBNF规则：

```
<expression> ::= <number> | <expression> "+" <expression>
<number> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

在这个例子中：

- `<expression>` 和 `<number>` 是非终结符。
- `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"` 和 `"+"` 是终结符。
- 规则`<expression> ::= <number> | <expression> "+" <expression>`表示一个表达式可以是一个单独的数字，或者是两个表达式通过加号连接的组合。

结论

通过定义这样的产生式规则，GBNF允许我们构建复杂的语法结构，用于解析或生成满足特定语法规则的字符串或代码。这种强大的语法描述能力使得GBNF成为描述复杂文本模式和语言结构的有力工具。

Example

Before going deeper, let's look at some of the features demonstrated in `grammars/chess.gbnf`, a small chess notation grammar:

```
# `root` specifies the pattern for the overall output
root ::= (
    # it must start with the characters "1. " followed by a sequence
    # of characters that match the `move` rule, followed by a space, followed
    # by another move, and then a newline
    "1. " move " " move "\n"

    # it's followed by one or more subsequent moves, numbered with one or two
digits
    ([1-9] [0-9]? ". " move " " move "\n")+
)

# `move` is an abstract representation, which can be a pawn, nonpawn, or castle.
# The `[+#]?` denotes the possibility of checking or mate signs after moves
move ::= (pawn | nonpawn | castle) [+#]?

pawn ::= ...
nonpawn ::= ...
castle ::= ...
```

在提供的grammars/chess.gbnf例子中，定义了一个用于解析棋谱的GBNF语法。这个语法旨在识别标准的国际象棋记谱法（例如用于棋谱的电子记录或分析）。以下是对这个例子中各个部分的详细解释：

根规则（Root Rule）

在root规则中，定义了整个输出的模式：

```
root ::= (
    "1. " move " " move "\n"
    ([1-9] [0-9]? ". " move " " move "\n")+
)
```

这段代码详细说明了棋局的起始和随后的每一步应如何被记录和解析：

- "1. " 表示棋局从"1. "开始，这是国际象棋记谱法的标准开局标记。
- move " " move "\n" 表示一对走子记录，分别为白方和黑方的移动，之后跟一个换行符。
- 在首次移动之后，([1-9] [0-9]? ". " move " " move "\n")+ 规定了棋局中的后续步骤。这里使用了正则表达式风格的语法，表示连续的移动，每个移动以数字开头（可能是一位或两位数，表示步数），后跟一个点和空格，然后是一对走子和换行。

移动规则（Move Rule）

```
move ::= (pawn | nonpawn | castle) [+#]?
```

move规则定义了一步棋的可能构成，这里包括三种可能的走子方式：pawn（兵的移动），nonpawn（非兵的移动），以及castle（王车易位）。[+#]? 表示每次移动后可能附加的棋盘状态标记，其中+代表将军，#代表将死。

具体的走子规则

虽然具体的pawn、nonpawn、castle规则在这个示例中没有展开，但它们通常会具体定义如下：

- pawn：定义了兵的移动，可能包括前进、吃过路兵等。
- nonpawn：定义了除兵外其他棋子（如王后、车、马等）的移动规则。
- castle：特别定义了王车易位，可能包括大易位和小易位。

总结

这个GBNF规则集提供了一个非常结构化和具体的方式来解析国际象棋的记谱，使得计算机程序能够有效读取和理解国际象棋的走子记录。通过这种方式，可以在棋局分析、自动重放和进一步的学习中使用棋谱数据。

Non-Terminals and Terminals

Non-terminal symbols (rule names) stand for a pattern of terminals and other non-terminals. They are required to be a dashed lowercase word, like move, castle, or check-mate.

Terminals are actual characters (code points). They can be specified as a sequence like `"1"` or `"O-O"` or as ranges like `[1-9]` or `[NBKQR]`.

在GBNF中，非终结符（Non-terminals）和终结符（Terminals）是构建语法规则的基础元素，它们共同定义了语法的结构和语言的构成。下面详细解释这些概念及其在GBNF中的应用。

非终结符（Non-Terminals）

非终结符是用来代表一组模式的符号，这些模式包括终结符和其他非终结符。在GBNF中，非终结符通常需要满足以下特征：

- **名称规范**：非终结符通常是由小写字母和连字符组成的词，如`move`, `castle`, 或 `check-mate`。这种命名方式有助于区分非终结符和终结符，并保持语法的清晰和组织性。
- **模式代表**：非终结符不直接对应到具体的字符，而是代表一个更大的构造或模式，这个模式本身可能由多个终结符和其他非终结符组成。例如，在棋盘游戏中，`move`可以代表任何合法的移动动作，这个动作的具体表现则由更详细的规则来定义。

终结符（Terminals）

终结符是指直接表示为字符或字符序列的元素，这些字符是构成语言的实际文本单位：

- **字符表示**：终结符可以是具体的字符或字符序列，如`"1"`或`"O-O"`。这些字符直接对应于文本中的实际字符，是语法分析的最基本单元。
- **字符范围**：终结符也可以通过字符范围来定义，例如`[1-9]`表示任何一个从1到9的数字，`[NBKQR]`表示国际象棋中的特定棋子（骑士、主教、王后、车和国王）。
- **Unicode 码点**：由于终结符对应于Unicode码点，因此它们可以包括任何有效的Unicode字符。这使得GBNF能够适用于包括多种语言和特殊符号在内的广泛文本。

应用示例

在定义一个语法规则时，你可能会遇到如下的定义：

```
number ::= [0-9]+
```

这里，`number`是一个非终结符，代表一个或多个数字的序列。`[0-9]+`则是终结符，表示匹配从0到9的任意数字，且这种匹配可以重复一次或多次。

结论

非终结符和终结符在定义形式语法时扮演着核心角色，非终结符提供了构造的抽象表示，而终结符则提供了具体的字符实现。这种区分使得GBNF能够有效地描述复杂的文本模式和语言结构，适用于从简单的数据格式到复杂的编程语言的广泛应用。

Characters and character ranges

Terminals support the full range of Unicode. Unicode characters can be specified directly in the grammar, for example `hiragana ::= [ぁ-ゟ]`, or with escapes: 8-bit (`\xXX`), 16-bit (`\uXXXX`) or 32-bit (`\UXXXXXXXX`).

Character ranges can be negated with `^`:

```
single-line ::= [^\n]+ "\n"`
```

在GBNF语法中，终结符支持全范围的Unicode字符，这使得其能够用于定义多种语言和特殊字符的模式。这部分内容介绍了如何在GBNF中直接使用Unicode字符，如何使用转义序列指定字符，以及如何定义字符范围和它们的否定形式。

Unicode 字符的使用

在GBNF中，可以直接在语法中使用Unicode字符，例如定义日语平假名的范围：

```
hiragana ::= [ぁ-ゟ]
```

这条规则定义了一个名为`hiragana`的非终结符，它包含了从`ぁ`到`ゟ`的所有平假名字符。这种方式非常直观，允许直接在规则中看到哪些字符被包括在内。

使用转义序列

当需要指定特定的Unicode字符而不使用它的可视形式时，可以使用转义序列。这在字符不易直接键入或阅读时尤其有用。GBNF支持以下几种转义序列：

- **8位转义**：`\xXX`，其中`XX`是两个十六进制数字，表示一个字符的8位Unicode码点。
- **16位转义**：`\uXXXX`，其中`XXXX`是四个十六进制数字，表示一个字符的16位Unicode码点。
- **32位转义**：`\UXXXXXXXX`，其中`XXXXXXXX`是八个十六进制数字，表示一个字符的32位Unicode码点。

这些转义序列使得可以精确地指定需要的字符，无论它们在编码表中的位置如何。

字符范围和否定

在GBNF中，字符范围可以通过在方括号`[]`中列出起止字符来定义，如上面的平假名例子所示。此外，还可以使用`^`来否定一个字符范围，表示“除了这些字符以外的所有字符”。例如：

```
single-line ::= [^\n]+ "\n"
```

这条规则定义了一个名为`single-line`的非终结符，它匹配除了换行符`\n`以外的一个或多个字符，最后以一个换行符结束。这常用于处理需要在一行内读取所有内容的情况，直到遇到行结束符。

总结

通过支持全范围的Unicode字符和提供灵活的字符指定方式（直接使用、字符范围、转义序列），GBNF使得语法设计者可以精确控制哪些字符可以出现在语言的特定部分。这种灵活性是处理多语言文本和特殊编码需求的关键。

Sequences and Alternatives

The order of symbols in a sequence matters. For example, in `"1. " move " " move "\n"`, the `"1. "` must come before the first `move`, etc.

Alternatives, denoted by `|`, give different sequences that are acceptable. For example, in `move ::= pawn | nonpawn | castle`, `move` can be a `pawn` move, a `nonpawn` move, or a `castle`.

Parentheses `()` can be used to group sequences, which allows for embedding alternatives in a larger rule or applying repetition and optional symbols (below) to a sequence.

在GBNF语法中，序列（Sequences）和选择（Alternatives）是构建复杂语法结构的基础工具。通过这些构造，可以定义语法的精确流程和可选路径，以匹配多样化的输入模式。下面将详细解释这些概念及其在GBNF中的应用。

序列（Sequences）

序列在GBNF中是指一系列符号（可以是终结符或非终结符）的严格排列。在定义序列时，每个符号的出现顺序是非常重要的，因为它直接影响了语法解析的流程和结果。

例如，在规则 `"1. " move " " move "\n"` 中，序列的组成表明：

- 首先，必须有字符串 `"1. "`。
- 紧接着是一个符合 `move` 规则的移动描述。
- 然后是一个空格 `" "`。
- 接着是另一个符合 `move` 规则的移动描述。
- 最后，必须以换行符 `"\n"` 结束。

这种严格的顺序确保了文本的解析可以按照预期的结构进行，对于编程语言的解析器、文件格式解析等场景尤为重要。

选择（Alternatives）

选择是通过竖线 `|` 表示的，用来定义不同的序列替代方案，任一方案都可被接受作为合法的输入。这允许语法具有更大的灵活性和适应性。

以规则 `move ::= pawn | nonpawn | castle` 为例，这里的选择表示 `move` 可以是以下三种情况之一：

- `pawn`：兵的移动。
- `nonpawn`：非兵类棋子的移动。
- `castle`：王车易位。

选择机制使得单一规则可以覆盖多种情况，增加了语法的表达能力。

分组和嵌套（Grouping with Parentheses）

括号 `()` 在GBNF中用于分组，允许将多个元素组合成一个单独的单元。这对于嵌入选择到更大的规则中、或者将重复和可选符号应用于整个序列非常有用。

例如，假设我们有以下规则：

```
number ::= ("0" | [1-9] [0-9]*)
```

这里，括号用于分组表示一个数字可以是：

- `"0"`，或者
- 一个非零数字开头，后面可以跟任意多个数字（包括0个）。

总结

序列和选择是GBNF语法定义中非常强大的工具，它们为语法的精确描述提供了必要的结构。序列确保符号以特定的顺序出现，而选择则提供了灵活性，使得语法可以适应多种可能的输入情况。括号的使用进一步增强了这些工具的功能，允许复杂的嵌套和分组，使得GBNF可以有效地应对复杂和多变的语法需求。

Repetition and Optional Symbols

- `*` after a symbol or sequence means that it can be repeated zero or more times (equivalent to `{0,}`).
- `+` denotes that the symbol or sequence should appear one or more times (equivalent to `{1,}`).
- `?` makes the preceding symbol or sequence optional (equivalent to `{0,1}`).
- `{m}` repeats the precedent symbol or sequence exactly `m` times
- `{m,}` repeats the precedent symbol or sequence at least `m` times
- `{m,n}` repeats the precedent symbol or sequence at between `m` and `n` times (included)
- `{0,n}` repeats the precedent symbol or sequence at most `n` times (included)

在GBNF中，重复和可选符号是用来控制语法中元素出现次数的工具，它们为语法规则提供了更大的灵活性和表达力。这些符号允许开发者精确地指定一个元素应当重复的次数，或者它的出现是否可选。以下是对每个重复和可选符号的详细解释：

`*` - 零次或多次重复

使用 `*` 后缀表示一个符号或序列可以重复零次或多次。这类似于正则表达式中的 `{0,}`。这意味着序列可以完全不出现，或者出现任意次数。例如：

```
<digits> ::= [0-9]*
```

这条规则表示 `<digits>` 可以是任意长度的数字序列，包括一个空序列。

`+` - 一次或多次重复

使用 `+` 后缀表示一个符号或序列必须至少出现一次，可以多次重复。这与正则表达式中的 `{1,}` 相同。例如：

```
<words> ::= <word>+
```

这条规则要求 `<words>` 至少包含一个 `<word>`，但可以包含更多。

`?` - 可选符号

使用 `?` 后缀使得一个符号或序列成为可选的，即它们可以出现零次或一次，相当于 `{0,1}`。这在定义可能省略的语法元素时非常有用。例如：

```
<line-end> ::= "\r"?
```

这条规则表示 `<line-end>` 可以包含一个可选的回车符 `\r`。

`{m}` - 精确 `m` 次重复

这种形式表示前面的符号或序列必须恰好重复 `m` 次。例如：

```
<year> ::= [0-9]{4}
```

表示 `<year>` 必须由四位数字组成。

`{m,}` - 至少 `m` 次重复

此形式表示符号或序列至少出现 `m` 次，可以无限次重复。例如：

```
<paragraph> ::= <sentence>{2,}
```

要求 `<paragraph>` 至少包含两个句子。

`{m,n}` - `m` 到 `n` 次重复（含）

这种形式表示符号或序列的出现次数必须在 `m` 和 `n` 之间，包括 `m` 和 `n`。例如：

```
<password> ::= [a-zA-Z0-9]{6,12}
```

表示 `<password>` 必须由6到12个字母或数字组成。

`{0,n}` - 最多 `n` 次重复

这种形式表示符号或序列最多重复 `n` 次。例如：

```
<optional-args> ::= <arg>{0,3}
```

表示 `<optional-args>` 可以包含0到3个 `<arg>`。

这些重复和可选符号在定义复杂的语法规则时非常有用，允许语法设计者为特定的语言特征提供严格的数值约束，从而增加语法的精确性和灵活性。

Comments and newlines

Comments can be specified with `#`:

```
# defines optional whitespace
ws ::= [ \t\n]+
```

Newlines are allowed between rules and between symbols or sequences nested inside parentheses. Additionally, a newline after an alternate marker | will continue the current rule, even outside of parentheses.

在GBNF语法中，注释和换行符的使用是为了增强语法定义的可读性和组织性。注释可以帮助解释和澄清规则的用途和功能，而适当的换行符使用则可以使复杂的规则更加清晰易读。以下是对如何在GBNF中使用注释和换行符的详细解释：

注释（Comments）

在GBNF中，注释是通过井号（#）符号来标记的，与许多编程语言中的注释用法相似。注释从井号开始，直到行末。它们通常用于解释代码的功能或提供有关规则的其他说明，但在语法解析过程中会被忽略。

例如：

```
# defines optional whitespace
ws ::= [ \t\n]+
```

这里的注释“defines optional whitespace”解释了ws这一规则的功能，即定义了可选的空白符，包括空格、制表符和换行符。

换行符（Newlines）

换行符在GBNF语法定义中被用来分隔规则以及规则内部的不同元素，以增加清晰度和可读性。在GBNF中，你可以在规则之间、规则内的符号或序列之间使用换行符，以及在括号内嵌套的元素之间使用换行符。

此外，GBNF中还有一种特殊用法，允许在选择标记|后使用换行符继续当前规则，即使在括号外部也是如此。这使得可以在不同的行上列出多个选择分支，从而在定义复杂的选择结构时保持规则的可读性。

例如：

```
expression ::=
    term "+" term
  | term "-" term
  | term
```

在这个例子中，|后的换行符使得每个选择可以单独位于一行，有助于清晰地展示所有可能的替代规则。

总结

注释和换行符的正确使用是GBNF语法定义的重要方面，不仅有助于其他开发者或用户理解和维护语法规则，也增加了语法本身的组织性和可读性。通过注释，开发者可以传达额外的信息或意图，而恰当的换行符使用则确保了复杂语法的清晰表达。

The root rule

In a full grammar, the `root` rule always defines the starting point of the grammar. In other words, it specifies what the entire output must match.

```
# a grammar for lists
root ::= ("- " item)+
item ::= [^\n]+ "\n"
```

在GBNF语法定义中，`root`规则起着至关重要的作用，它定义了整个语法的起点。这意味着所有解析的输入都必须符合`root`规则所描述的模式。这里详细解释一下提供的例子，并说明`root`规则如何作为整个语法的基础。

`root`规则的作用

`root`规则定义了整个语法分析的起始点，也就是说，它规定了整个输出必须匹配的模式。在任何GBNF定义的语法中，解析器都会从`root`规则开始，尝试将输入文本与此规则相匹配。因此，`root`规则的定义直接影响到整个文本的解析结果。

示例解析

在提供的例子中，定义了一个用于解析列表的语法：


```
# a grammar for lists
root ::= ("- " item)+
item ::= [^\n]+ "\n"
```


解释各个组成部分

- `root ::= ("- " item)+`
  - 这条规则表示`root`由一个或多个`"- " item`序列组成。每个序列由一个短划线加空格`"- "`开始，后跟一个`item`。
  - `+`操作符表明至少有一个这样的序列存在，因此，至少需要有一个以`"- "`开始的列表项。
- `item ::= [^\n]+ "\n"`
  - `item`规则定义了列表中每一项的内容。这里使用`[^\n]+`表示匹配任何不是换行符的字符序列，直到遇到换行符，这意味着每个`item`可以包含任意非换行的字符序列。
  - `"\n"`确保了每个项目后面跟着一个换行符，这是列表每一项的结束标记。

语法的功能和应用

这个语法规则非常适合解析简单的带有项目符号的文本列表。例如，它可以成功解析以下形式的文本：

```
- item 1
- item 2
- item 3
```

每一行都以"- "开始，并且以换行符结束，符合item的定义。这种类型的语法定义可以用在需要提取文本文件中列表信息的场景，如处理购物列表、任务清单等。

总结

通过定义精确的root规则，GBNF语法为特定格式的文本提供了强有力的解析能力。在这个例子中，root规则通过确定性的结构定义，确保了所有合法的输入都符合预期的列表格式，从而使得解析过程既高效又可靠。

Next steps

This guide provides a brief overview. Check out the GBNF files in this directory (grammars/) for examples of full grammars. You can try them out with(本指南提供了简要概述。查看此目录 (grammars/) 中的 GBNF 文件以获取完整语法的示例。您可以使用):

```
# 使用--grammar-file指定语法文件从而影响model的输出内容
./llama-cli -m <model> --grammar-file grammars/some-grammar.gbnf -p 'Some prompt'
```

llama.cpp can also convert JSON schemas to grammars either ahead of time or at each request, see below.（llama.cpp 还可以提前或在每次请求时将 JSON 模式转换为语法，见下文。）

Troubleshooting

Grammars currently have performance gotchas (see https://github.com/ggerganov/llama.cpp/issues/4218).（语法目前存在性能缺陷，查看https://github.com/ggerganov/llama.cpp/issues/4218）

Efficient optional repetitions（高效的可选重复）

A common pattern is to allow repetitions of a pattern x up to N times.

While semantically correct, the syntax x? x? x?.... x? (with N repetitions) may result in extremely slow sampling. Instead, you can write x{0,N} (or (x (x (x ... (x)?...)?)?)? w/ N-deep nesting in earlier llama.cpp versions).

在处理GBNF语法时，一个常见的性能问题涉及到可选重复的模式，即允许某个模式x重复多次，但次数有上限。这部分内容讨论了这一问题的性能隐患以及如何更有效地处理这种情况。

性能缺陷的描述

如上文所示，使用形如x? x? x?.... x?（x的可选出现，重复N次）的语法在语义上是正确的，但可能导致极其缓慢的解析性能。这是因为每个x?的存在增加了解析器的计算负担，尤其是在x可以选择出现或完全不出现的情况下，解析器需要检查所有可能的组合，这导致性能以指数级增长。

高效的解决方案

使用范围重复

为了提高效率，推荐使用`x{0,N}`的形式来替代连续的`x?`重复。这种写法直接告诉解析器`x`可以出现从0到N次，解析器因此可以更高效地计算出所有合法的重复次数，而不需要单独评估每一次可能的省略。

使用嵌套结构

另一个较早的解决方案是使用嵌套结构`(x (x (x ... (x)?...)?)?)?`，这在早期的`llama.cpp`版本中可能更常见。这种方法通过递归地嵌套可选的`x`来实现重复，虽然这比直接使用范围重复的写法`{0,N}`在逻辑上更复杂一些，但它提供了一种处理早期版本中可能不支持范围重复的方法。

总结

在设计使用GBNF语法的解析器时，考虑到性能优化是非常重要的，特别是在处理可能重复的模式时。选择合适的语法结构可以显著影响解析的效率和响应时间。通过使用`x{0,N}`或合适的嵌套结构，可以有效地减少计算复杂度和提高性能，这对于实际应用中处理大量数据或复杂语法结构至关重要。

Using GBNF grammars

You can use GBNF grammars:

- In llama-server's completion endpoints, passed as the `grammar` body field
- In llama-cli, passed as the `--grammar` & `--grammar-file` flags
- With llama-gbnf-validator tool, to test them against strings.

GBNF语法的使用非常灵活，可以在多种不同的工具和环境中应用，以增强文本处理和解析功能。下面是对上述内容中提到的几种使用GBNF语法的方式的详细解释：

1. 在 `llama-server` 中使用GBNF语法

`llama-server` 是一个服务端应用，通常用来处理网络请求。在这种环境下，GBNF语法可以直接通过请求的`grammar`体字段传递给服务端的完成端点（completion endpoints）。这使得客户端可以动态指定语法规则，服务端根据这些规则来处理和响应请求。例如，如果你在开发一个需要理解或生成特定格式文本的Web服务，可以使用GBNF语法来确保生成的文本符合预定的格式。

2. 在 `llama-cli` 中使用GBNF语法

`llama-cli` 是一个命令行工具，它提供了直接在命令行界面下测试和运行GBNF语法的能力。使用`--grammar`和`--grammar-file`标志，用户可以在命令行中直接指定一个GBNF语法规则，或者指定一个包含GBNF规则的文件。这种方式适合于开发和测试阶段，开发者可以快速试验和调整GBNF规则。

3. 使用 `llama-gbnf-validator` 工具

`llama-gbnf-validator` 是一个专门的工具，用于测试GBNF语法是否能正确匹配指定的字符串。这是一个非常有用的工具，尤其在开发复杂的GBNF规则时，能够帮助开发者验证和调试语法规则。通过这种方式，可以确保GBNF规则按预期工作，有效地捕捉或生成正确的文本格式。

应用示例

例如，如果你正在开发一个应用程序，需要解析用户输入的复杂命令或数据，可以使用`llama-cli`来测试你的GBNF规则。一旦规则通过测试，可以将其部署到`llama-server`，以便实时处理用户请求。同时，使用`llama-gbnf-validator`来进行持续的测试和验证，确保随着应用需求的变化，GBNF规则仍然有效。

总结

通过在不同的环境和工具中应用GBNF语法，开发者可以构建更健壮、更灵活的文本处理和解析功能。无论是在服务端处理复杂的请求，在命令行界面快速测试语法，还是在开发过程中验证规则的正确性，GBNF语法提供了强大的支持。这种灵活性和强大的功能使得GBNF成为处理复杂文本数据的有力工具。

JSON Schemas → GBNF

`llama.cpp` supports converting a subset of https://json-schema.org/ to GBNF grammars:

- In llama-server:
  - For any completion endpoints, passed as the `json_schema` body field
  - For the `/chat/completions` endpoint, passed inside the `result_format` body field (e.g. `{"type", "json_object", "schema": {"items": {}}}`)
- In llama-cli, passed as the `--json` / `-j` flag
- To convert to a grammar ahead of time:
  - in CLI, with examples/json_schema_to_grammar.py
  - in JavaScript with json-schema-to-grammar.mjs (this is used by the server's Web UI)

Take a look at tests to see which features are likely supported (you'll also find usage examples in https://github.com/ggerganov/llama.cpp/pull/5978, https://github.com/ggerganov/llama.cpp/pull/6659 & https://github.com/ggerganov/llama.cpp/pull/6555).

```
llama-cli \
  -hfr bartowski/Phi-3-medium-128k-instruct-GGUF \
  -hff Phi-3-medium-128k-instruct-Q8_0.gguf \
  -j '{
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "name": {
                "type": "string",
                "minLength": 1,
                "maxLength": 100
            },
            "age": {
                "type": "integer",
                "minimum": 0,
                "maximum": 150
            }
        },
        "required": ["name", "age"],
        "additionalProperties": false
    },
    "minItems": 10,
    "maxItems": 100
  }' \
  -p 'Generate a {name, age}[] JSON array with famous actors of all ages.'
```

▶ Details
Show grammar

You can convert any schema in command-line with:

```
examples/json_schema_to_grammar.py name-age-schema.json
```

```
char ::= [^"\\\x7F\x00-\x1F] | [\\] (["\\bfnrt] | "u" [0-9a-fA-F]{4})
item ::= "{" space item-name-kv "," space item-age-kv "}" space
item-age ::= ([0-9] | ([1-8] [0-9] | [9] [0-9]) | "1" ([0-4] [0-9] | [5] "0"))
space
item-age-kv ::= "\"age\"" space ":" space item-age
item-name ::= "\"" char{1,100} "\"" space
item-name-kv ::= "\"name\"" space ":" space item-name
root ::= "[" space item ("," space item){9,99} "]" space
space ::= | " " | "\n" [ \t]{0,20}
```

Here is also a list of known limitations (contributions welcome):

- `additionalProperties` defaults to `false` (produces faster grammars + reduces hallucinations).
- `"additionalProperties": true` may produce keys that contain unescaped newlines.
- Unsupported features are skipped silently. It is currently advised to use the command-line Python converter (see above) to see any warnings, and to inspect the resulting grammar / test it w/ llama-gbnf-validator.
- Can't mix `properties` w/ `anyOf` / `oneOf` in the same type (https://github.com/ggerganov/llama.cpp/issues/7703)
- prefixItems is broken (but items works)
- `minimum`, `exclusiveMinimum`, `maximum`, `exclusiveMaximum`: only supported for `"type": "integer"` for now, not `number`
- Nested `$ref`s are broken (https://github.com/ggerganov/llama.cpp/issues/8073)
- patterns must start with `^` and end with `$`
- Remote `$ref`s not supported in the C++ version (Python & JavaScript versions fetch https refs)
- `string` formats lack `uri`, `email`
- No `patternProperties`

And a non-exhaustive list of other unsupported features that are unlikely to be implemented (hard and/or too slow to support w/ stateless grammars):

- `uniqueItems`
- `contains` / `minContains`
- `$anchor` (cf. dereferencing)
- `not`
- Conditionals `if` / `then` / `else` / `dependentSchemas`

A word about additionalProperties

> [!WARNING] The JSON schemas spec states `object`s accept additional properties by default. Since this is slow and seems prone to hallucinations, we default to no additional properties. You can set `"additionalProperties": true` in the the schema of any object to explicitly allow additional properties.

If you're using Pydantic to generate schemas, you can enable additional properties with the `extra` config on each model class:

```
# pip install pydantic
import json
from typing import Annotated, List
from pydantic import BaseModel, Extra, Field
class QAPair(BaseModel):
    class Config:
        extra = 'allow'  # triggers additionalProperties: true in the JSON schema
    question: str
    concise_answer: str
    justification: str

class Summary(BaseModel):
    class Config:
        extra = 'allow'
    key_facts: List[Annotated[str, Field(pattern='- .{5,}')]]
    question_answers: List[Annotated[List[QAPair], Field(min_items=5)]]

print(json.dumps(Summary.model_json_schema(), indent=2))
```

▶ Show JSON schema & grammar

```
{
  "$defs": {
    "QAPair": {
      "additionalProperties": true,
      "properties": {
        "question": {
          "title": "Question",
          "type": "string"
        },
        "concise_answer": {
          "title": "Concise Answer",
          "type": "string"
        },
        "justification": {
          "title": "Justification",
          "type": "string"
        }
      },
      "required": [
        "question",
        "concise_answer",
```

```
        "justification"
      ],
      "title": "QAPair",
      "type": "object"
    }
  },
  "additionalProperties": true,
  "properties": {
    "key_facts": {
      "items": {
        "pattern": "^- .{5,}$",
        "type": "string"
      },
      "title": "Key Facts",
      "type": "array"
    },
    "question_answers": {
      "items": {
        "items": {
          "$ref": "#/$defs/QAPair"
        },
        "minItems": 5,
        "type": "array"
      },
      "title": "Question Answers",
      "type": "array"
    }
  },
  "required": [
    "key_facts",
    "question_answers"
  ],
  "title": "Summary",
  "type": "object"
}
```

```
QAPair ::= "{" space QAPair-question-kv "," space QAPair-concise-answer-kv ","
space QAPair-justification-kv ( "," space ( QAPair-additional-kv ( "," space
QAPair-additional-kv )* ) )? "}" space
QAPair-additional-k ::= ["] ( [c] ([o] ([n] ([c] ([i] ([s] ([e] ([_] ([a] ([n]
([s] ([w] ([e] ([r] char+ | [^"r] char*) | [^"e] char*) | [^"w] char*) | [^"s]
char*) | [^"n] char*) | [^"a] char*) | [^"_] char*) | [^"e] char*) | [^"s] char*)
| [^"i] char*) | [^"c] char*) | [^"n] char*) | [^"o] char*) | [j] ([u] ([s] ([t]
([i] ([f] ([i] ([c] ([a] ([t] ([i] ([o] ([n] char+ | [^"n] char*) | [^"o] char*) |
[^"i] char*) | [^"t] char*) | [^"a] char*) | [^"c] char*) | [^"i] char*) | [^"f]
char*) | [^"i] char*) | [^"t] char*) | [^"s] char*) | [^"u] char*) | [q] ([u] ([e]
([s] ([t] ([i] ([o] ([n] char+ | [^"n] char*) | [^"o] char*) | [^"i] char*) |
[^"t] char*) | [^"s] char*) | [^"e] char*) | [^"u] char*) | [^"cjq] char* )? ["]
space
QAPair-additional-kv ::= QAPair-additional-k ":" space value
QAPair-concise-answer-kv ::= "\"concise_answer\"" space ":" space string
```

```
QAPair-justification-kv ::= "\"justification\"" space ":" space string
QAPair-question-kv ::= "\"question\"" space ":" space string
additional-k ::= ["] ( [k] ([e] ([y] ([_] ([f] ([a] ([c] ([t] ([s] char+ | [^"s]
char*) | [^"t] char*) | [^"c] char*) | [^"a] char*) | [^"f] char*) | [^"_] char*)
| [^"y] char*) | [^"e] char*) | [q] ([u] ([e] ([s] ([t] ([i] ([o] ([n] ([_] ([a]
([n] ([s] ([w] ([e] ([r] ([s] char+ | [^"s] char*) | [^"r] char*) | [^"e] char*) |
[^"w] char*) | [^"s] char*) | [^"n] char*) | [^"a] char*) | [^"_] char*) | [^"n]
char*) | [^"o] char*) | [^"i] char*) | [^"t] char*) | [^"s] char*) | [^"e] char*)
| [^"u] char*) | [^"kq] char* )? ["] space
additional-kv ::= additional-k ":" space value
array ::= "[" space ( value ("," space value)* )? "]" space
boolean ::= ("true" | "false") space
char ::= [^"\\\x7F\x00-\x1F] | [\\] (["\\bfnrt] | "u" [0-9a-fA-F]{4})
decimal-part ::= [0-9]{1,16}
dot ::= [^\x0A\x0D]
integral-part ::= [0] | [1-9] [0-9]{0,15}
key-facts ::= "[" space (key-facts-item ("," space key-facts-item)*)? "]" space
key-facts-item ::= "\"" "- " key-facts-item-1{5,} "\"" space
key-facts-item-1 ::= dot
key-facts-kv ::= "\"key_facts\"" space ":" space key-facts
null ::= "null" space
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]? integral-part)?
space
object ::= "{" space ( string ":" space value ("," space string ":" space value)*
)? "}" space
question-answers ::= "[" space (question-answers-item ("," space question-answers-
item)*)? "]" space
question-answers-item ::= "[" space question-answers-item-item ("," space
question-answers-item-item){4,} "]" space
question-answers-item-item ::= QAPair
question-answers-kv ::= "\"question_answers\"" space ":" space question-answers
root ::= "{" space key-facts-kv "," space question-answers-kv ( "," space (
additional-kv ( "," space additional-kv )* ) )? "}" space
space ::= | " " | "\n" [ \t]{0,20}
string ::= "\"" char* "\"" space
value ::= object | array | string | number | boolean | null
```

If you're using Zod, you can make your objects to explicitly allow extra properties w/ `nonstrict()` / `passthrough()` (or explicitly no extra props w/ `z.object(...).strict()` or `z.strictObject(...)`) but note that zod-to-json-schema currently always sets `"additionalProperties": false` anyway.

```ts
import { z } from 'zod';
import { zodToJsonSchema } from 'zod-to-json-schema';

const Foo = z.object({
  age: z.number().positive(),
  email: z.string().email(),
}).strict();

console.log(zodToJsonSchema(Foo));
```

▶ Show JSON schema & grammar

```
{
  "type": "object",
  "properties": {
    "age": {
      "type": "number",
      "exclusiveMinimum": 0
    },
    "email": {
      "type": "string",
      "format": "email"
    }
  },
  "required": [
    "age",
    "email"
  ],
  "additionalProperties": false,
  "$schema": "http://json-schema.org/draft-07/schema#"
}
```

```
age-kv ::= "\"age\"" space ":" space number
char ::= [^"\\\x7F\x00-\x1F] | [\\] (["\\bfnrt] | "u" [0-9a-fA-F]{4})
decimal-part ::= [0-9]{1,16}
email-kv ::= "\"email\"" space ":" space string
integral-part ::= [0] | [1-9] [0-9]{0,15}
number ::= ("-"? integral-part) ("." decimal-part)? ([eE] [-+]? integral-part)?
space
root ::= "{" space age-kv "," space email-kv "}" space
space ::= | " " | "\n" [ \t]{0,20}
string ::= "\"" char* "\"" space
```

`llama.cpp`支持将JSON Schema的一个子集转换为GBNF语法。这种转换功能允许用户将结构化的JSON Schema定义直接转换成形式语法，这对于确保特定类型的JSON数据符合预期格式非常有用。以下详细解释这种转换功能及其应用方式：

在`llama-server`中使用

在`llama-server`中，可以通过向服务器的完成端点发送`json_schema`体字段来使用JSON Schema。这允许服务端根据传递的JSON Schema动态生成相应的GBNF语法，进而控制输出数据的结构。此外，在`/chat/completions`端点，可以在`result_format`体字段中传递JSON Schema，例如：`{"type": "json_object", "schema": {"items": {}}}`。这使得服务端能根据定义的结构生成符合规范的JSON对象。

在`llama-cli`中使用

通过命令行界面`llama-cli`，用户可以使用`--json`或`-j`标志来指定JSON Schema。这种方式便于在本地测试和验证JSON Schema与GBNF的转换是否符合预期。

预先转换

为了方便开发者预先转换JSON Schema到GBNF语法，`llama.cpp`提供了两种工具：

- 在命令行使用`json_schema_to_grammar.py`脚本。该脚本允许用户将JSON文件中的Schema转换成GBNF语法。
- 在JavaScript环境使用`json-schema-to-grammar.mjs`模块。该模块在`llama-server`的Web UI中使用，允许在浏览器中直接进行转换。

测试和验证

为了验证转换的GBNF语法是否正确，可以查看`llama.cpp`的`tests/test-json-schema-to-grammar.cpp`测试文件，这些测试用例展示了哪些功能是受支持的，并提供了使用示例。此外，`llama-gbnf-validator`工具可用于测试特定的字符串是否符合生成的GBNF语法。

已知的限制

尽管转换功能强大，但存在一些已知限制，例如：

- `additionalProperties`默认为`false`，这有助于生成更快的语法并减少生成错误数据的风险。
- 不支持的特性将被静默跳过。建议使用命令行Python转换器来查看任何警告，并检查结果语法。
- 某些JSON Schema功能（如`uniqueItems`、`contains`等）可能因为实现难度或性能问题不会被支持。

使用场景举例

假设需要生成一个包含特定属性的JSON数组，如名字和年龄，每个条目都需要符合特定的约束（如名字长度和年龄范围），这时可以使用上述工具和方法来确保生成的数据符合预定义的JSON Schema，这对于自动化测试、数据验证等场景非常有用。通过指定`--json`标志和相应的Schema，`llama-cli`能够按需生成严格符合规范的数据，有助于提高开发效率和数据质量。