【编译原理/龙书】用Rust写一个SQL Parser【1】

引言 & 简述

啊...这里是闭关 ~~修炼~~ 偷懒半个月的Makiror。上一篇博客上在期中考前两天完成的，完成后对自己的算法思维很自信，开心愉快了几天，然后我的数学考~砸~啦~（笑着笑着就哭了）
好啦其实没有那么糟糕，我本来想周末写文章的，但我跑去Hong Kong参加 Google I/O Extended Watch Party 2023 了，然后跑去其他地方度假了，后来又有一些奇奇怪怪的事情，所以往后再写了。

如标题所写，这次的主题还是 *Compilers: Principles, Techniques, and Tools*，《编译原理》，之后我们将其称为龙书。 内容主要会涉及到龙书第三章-词法分析（Lexical Analysis）和第四章-语法分析（Syntactic Analysis），其中为了更好地理解，我有针对性地选读过 *Modern Compiler Implementation in C*《现代编译原理 C语言描述》的相关章节，之后我们将其称为虎书。有看过我上一篇文章 *【Crystal入门与实践】LA、FA - 有穷自动机与正则引擎实现* 的小伙伴应该对这两本书不陌生了。 我们将通过里面的算法和一些小小Rust魔法来实现一个SQL解析器，这是Part 1，当然很多东西只要在本章疏通后面就很简单了。因为这个项目的内容涉及到的知识点不多就那些，但是代码量和重复运用是很多的，所以我不会像以往一样把完整代码都写在文章内，只会写某些关键内容的定义和实现。此外我会把这个SQL Parser开源，完整代码你可以通过文末Github Link查看。

尽管本文没有门槛要求，我仍然建议你多少看过书上的定义再来看这篇文章，很多定义类的东西看过后再实操会清晰很多，这些书我觉得也没有网上说的那么难理解（很多人说这些书如同“天书”一般，我一直很不认同这些说法，不要给自己没必要的心理暗示和焦虑），理解了就没什么好难的了。相比博客，这些书对于算法的解释是很详细的。

实践目标

标题很明确了，我们的目标是做一个Rust SQL Parser，其实工程量是很巨大的，目前我只是打算支持一些基本语句，毕竟重点是实现的算法，如果要实现完整的SQL Parser工程量是非常巨大的。此外，为了统一标准，本文的SQL语法只参考了SQL 92。
在Part 1我们会完成一个SQL语句的词法分析器和Select语句及其子句的语法分析器，可以输入一个Select语句并对其分析。（有些子句分析适用于其他语句，可重复使用）文末会列出本文为止该SQL Parser支持的内容，其余更新关注仓库即可。
我大概想了一下，即便是这样的内容，也能涵盖大部分语句的语法分析所需要的算法了，所以拓展是非常简单的，Part 2估计也不是针对其他语句分析的实现了。

什么是SQL?

在此简短地介绍一下SQL（Structured Query Language，结构化查询语言），它是一种操作关系型数据库的语言，基本上摸过数据库的人都对此不陌生了。它是一种非过程化的语言，我们可以用它来完成对数据库的各种操作，例如基本的增删查改。

将代码变成魔法

> 不同于以往的风格，这样的标题怎样?

编译器简述

我很记得初中时，啊对 是初一的计算机课，就有说过计算机能直接识别并运行的是什么——机器码，我们平时常用的高级编程语言是不能让计算机直接执行的。但是现在已经2023年了，很显然在绝大部分情况我们都能根据需求使用各种各样的编程语言来编写代码，以更自然更人类的方法表达告诉计算机它要做什么，而不是像早期一样手写01 。这是再普通不过的常识了。
计算机看不懂高级语言，它只知道01，但是高级语言更亲近人类的思维和表达，所以人们更愿意使用它。编译器充当了一个翻译者的角色，其实就是把代码变成计算机能看懂的机器码。

编译器是如何工作的?

编译器的工作流程主要分为几步：

- 词法分析 (Lexical Analysis)：编译器会对源代码进行词法分析，将一条语句中的每个单词（称为记号或标记）分离出来。也叫词汇分析。分离出来的单词在下一步将被用于语法分析。

- 语法分析 (Syntax Analysis)：在语法分析阶段，编译器会根据语言的语法规则，对词法分析器分离出的标记序列进行分类，并按照内部结构进行组织，这也叫语法制导翻译。

- 语义分析 (Semantic Analysis)：在语义分析阶段，编译器会进一步检查源代码的语义是否正确，并进行一些常量与变量的求值。

- 代码优化 (Code Optimization)：在优化阶段，编译器会尝试更好地组织和修改程序，以使其更有效率、更快更小。代码优化器可以执行各种转换和重写操作，例如删除未使用的代码、提取循环不变量、将表达式重组以提高运行速度等。

- 目标代码生成 (Code Generation)：在目标代码生成阶段，编译器将根据优化后的程序生成目标

代码。目标代码是汇编语言或机器码等底层指令。最后它们被写入文件或内存中作为可执行程序使用。

对于我们这次要做的SQL Parser而言，会涉及到前三步，后面就有点不一样了（大体相同），就是查询优化（Query Optimization）和执行计划生成 （Execution Plan Generation），前者看名字就知道什么意思了，后者就是根据最佳查询计划生成执行计划，例如一组指令等。而在本文我们会完成前两步。

词法分析

我们需要做的

输入一个SQL语句，分析的第一步就是词法分析(Lexical Analysis)。在上次的文章 *【Crystal入门与实践】LA、FA - 有穷自动机与正则引擎实现* 中也有提及，甚至可以说那篇文章的重点就是词法分析。但是其实我们这次要做的东西不太一样，上次我们通过各种闭包（Closure）等操作构建有穷自动机（Finite automata）再将其确定化和简化，实现正则语法的分析和匹配。但是对于SQL而言，词法分析就简单很多了（毕竟本文的压轴是语法分析），我们主要是写一个词法分析其将SQL语句转换为关键字、操作符、字面量等。。

词法分析器

Token意为『最小的语法单元』，但是你阅读一些相关的中文/中翻书籍可能会称其为“单词”“记号”，麻烦死了简直，所以在这里记住这个概念，后面直接叫Token就行了。
词法分析器（Lexer）的任务很简单，就是读入一段输入并将它们转换为一个Token序列，这也意味着它会抛弃多余的空白符和本来就无关的注释（这玩意是给人看的），把有实际意义的语句给分析成Token，不然这些问题扔给语法分析器处理的话会很麻烦。

类型定义

在讲算法之前我们先做点枯燥无味的东西。
就目前来看，我们只需要六种类型的Token：关键字（Keyword）、符号（Symbol）、数字（Number）、变量（Variable）、函数（Function）、布尔值（Bool）、空值（Null）、字面量（Identifier）。其中，不符合前七者条件的都会被归于字面量类型。

> 完整的数据类型定义均可在仓库的 `src/datatype/` 目录中查阅

Keyword就是诸如Select, Insert, Update, Delete之类的语句关键字，或者是From, Where, GroupBy, OrderBy这样的子句关键字（这类子句我们称之为clause，我们后面需要将它们拆分来进行语法分析），再或

者是Asc, Desc这样的排序关键字。你应该大概知道它们是些什么东西了，无论如何，我们先使用一个枚举类写出当前的Keyword枚举：

```
// src/datatype/keyword.rs

#[derive(Debug, PartialEq, Clone)]
pub enum Keyword {
    Select,
    Insert,
    Update,
    Delete,
    From,
    ...
}
```

当我们在之后的分析需要输出错误提示，例如`Unexpected token: 'WHERE'`之类的时候，你肯定不希望看到一堆`Unexpected token: 'Keyword(WHERE)'`这样的输出，用户哪需要在意这个Token是什么类型哦，所以我们要写一个方法重载（Method Overloading），实现fmt::Display trait 的 fmt 方法，用于将 Keyword 类型的值转换为字符串，因为这玩意是个枚举类，所以这个方法要考虑到它的每一个变体（Variant）。
所以我们在 fmt 方法中使用了 match 来挨个匹配不同的变体，然后分别调用 write! 宏将对应的字符串写入到 f 参数中，最终返回一个 fmt::Result 类型的值。这样，当我们使用 println 之类的宏时时，就可以得到期望的Keyword值输出了。

```
// src/datatype/keyword.rs

impl fmt::Display for Keyword {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Self::Select => write!(f, "SELECT"),
            Self::Insert => write!(f, "INSERT"),
            Self::Update => write!(f, "UPDATE"),
            Self::Delete => write!(f, "DELETE"),
            Self::From => write!(f, "FROM"),
            ...
        }
    }
}
```

输入的词经过一系列处理后会变成字符串，所以我们需要写一个函数来把字符串转换为Keyword。
因为某些Keyword是中间有空格的，尽管在分析时我们要抛弃无用的空格，我们在这个函数先假设它给的是一个含有必要空格的输入。所以以输入的空格来分割创建一个迭代器，在大部分单个单词的Keyword就直接转换即可。标准SQL是对大小写不敏感的，所以我们要在那之前先将输入的内容转为全大写。

```
// src/datatype/keyword.rs

pub fn to_keyword(s: &str) -> Option<Keyword> {
```

```
    let string = s.to_uppercase();
    let mut iter = string.split_whitespace();
    let first = iter.next()?;

    match first.as_str() {
        "SELECT" => Some(Keyword::Select),
        "INSERT" => Some(Keyword::Insert),
        "UPDATE" => Some(Keyword::Update),
        "DELETE" => Some(Keyword::Delete),
        "FROM" => Some(Keyword::From),
        ...
    }
}
```

我们以Group By这种Keyword为例，处理很简单，当匹配到'GROUP'时就尝试在迭代器获取下一个'BY'，若没有就照返回，这样的话那个孤苦伶仃的Group应该会被Lexer当作是字面量，然后在之后的语法分析出错。其他多个词的Keyword同理。

```
        "GROUP" => {
            if let Some(next) = iter.next() {
                if next == "BY" {
                    return Some(Keyword::GroupBy);
                }
            }
            None
        }
```

另外需要我们枚举定义的就是符号（Symbol）和函数（Function），按照上面Keyword那套定义就好了。后面我们有一些会用到的其他方法到时候再说。

```
// src/datatype/symbol.rs

#[derive(Debug, PartialEq, Clone)]
pub enum Symbol {
    Comma,
    Dot,
    Asterisk,
    Plus,
    ...
}
```

```
// src/datatype/function.rs

#[derive(Debug, PartialEq, Clone)]
pub enum FunctionT {
    Sum,
    Avg,
    Count,
```

```
    Max,
    Min,
    Concat,
}
```

Function的枚举类定义稍微麻烦一点，在后面语法分析时我们为了方便会使用携带调用参数的函数，但是这不应该是在词法分析阶段完成的，所以在词法分析阶段我们只需要把这类关键字专门分出一个类型的Token，但是为了防止后面和另外一个Function（携带调用参数的）有冲突，这个定义后面加上T（指Token，我承认这是我不得不做的一个举措，不然我不喜欢这样）
然后我们简单的定义一个Token枚举类（省略外部源文件引入）：

```
// src/datatype/token.rs

#[derive(Debug, PartialEq, Clone)]
pub enum Token {
    Keyword(Keyword),
    Symbol(Symbol),
    Function(FunctionT),
    Identifier(String),
    Variable(String),
    Number(String),
    Null,
}
```

其他的。。。直接去Github看吧，这种实在是没什么好讲的，就是一堆枚举...所以现在我们来讲处理输入。

处理输入和迭代转换

计算机本不认识什么Token，我们更不能简单粗暴地使用空格分隔输入，因为很多东西不是这么简单的。例如变量前缀@、逗号什么的，所以我们需要另外一种做法来精准地把一串输入转为Token。
首先，我们需要毫无偏见地把输入的字符串，转成每个单独的Char。然后，Rust有一种可允许在不移动迭代器指针的情况下查看下一个元素的迭代器类型，我们叫它Peekable 迭代器。

```
// src/lexer.rs

pub fn lex(text: &str) -> Vec<Token> {
    let mut tokens: Vec<Token> = Vec::new();
    let mut chars = text.chars().peekable();
}
```

你看Rust文档就能知道，Rust标准迭代器Iterator不支持Peek操作，所以Rust另外提供了这种迭代器，实际上它会包装这个实现了Iterator trait的迭代器，然后我们就能在后面进行迭代操作了。
Char迭代，意味着它是单个单个字符读取的，但是我们有很多Keyword啊什么的不止单个字符，所以我们要写一个函数，使其能读取我们期望的结果并返回。我们要接触一个概念叫闭包（Closure），在Rust中，闭包

是一种可以捕获其环境并可以作为参数传递的函数类型，我们将在这个函数用到这个概念。

collect_until

这个函数会从字符迭代器中读取字符，直到满足某个条件为止。

输入：

- chars：一个可变的字符迭代器，类型为 *Peekable<Chars>*
- condition：一个接受两个参数的闭包：当前读取到的字符和已经读取到的字符串，其返回值为bool。

输出：result：读取到的字符串

我们使用peek方法读取迭代器的下一个Char，这样读取并不会消耗它。若读取到的字符满足给定的终止条件则跳出循环，否则将读取到的字符加入结果的字符串中，并使用next方法消耗目标字符，继续循环。
或者迭代器为空时会跳出循环。

```
// src/lexer.rs

pub fn collect_until<F>(chars: &mut std::iter::Peekable<std::str::Chars>, condition: F) -> String
where
    F: Fn(char, String) -> bool,
{
    let mut result = String::new();
    while let Some(&c) = chars.peek() {
        if condition(c, result.clone()) {
            break;
        }
        result.push(c);
        chars.next();
    }
    result
}
```

它的时间复杂度为 O(n)，其中 n 是读取到满足终止条件的字符所需的字符数。

lex

将SQL语句分析类型并转换为Token序列。

输入：text：SQL语句，类型为 *&str*
输出：tokens：Token序列，类型为 *Vec<Token>*

然后我们先尝试写最简单的模式匹配，若遇到空格、制表符、回车符或换行符就跳过，若遇到注释符号则判

断迭代器的再下一个元素是否也是‘-’，若是则消耗掉两个元素并将接下来的内容直到换行都当作是注释内容，将它们忽略掉。

```
    while let Some(&token) = chars.peek() {
        match token {
            ' ' | '\n' | '\r' | '\t' => {
                chars.next();
            }
            '-' if chars.nth(1).map_or(false, |c| c == '-') => {
                chars.next();
                if let Some('-') = chars.peek() {
                    chars.next();
                } else {
                    tokens.push(Token::Symbol(Symbol::Minus));
                    continue;
                }
                let _ = collect_until(&mut chars, |c, _| c == '\n').trim().to_string();
            }
        }
    }
```

peek是基于迭代器状态机实现的，调用时会返回迭代器的当前状态并恢复原本的状态，所以它不能直接查看下下个元素。而可能你会想到nth方法可以用来获取迭代器指定位置的元素，但很可惜，它会消耗掉前面的元素，所以我们在这里只能老老实实地写『如果下一个不是-，则这只是一个减号，将它作为减号处理』

然后是单引号和双引号括住的字面量，后面的所有内容直到再遇到单引号或双引号就是无论怎样都会当作字面量处理。

```
            '\'' | '"' => {
                if let Some(quote) = chars.next() {
                    let literal = collect_until(&mut chars, |c, _| c == quote);
                    tokens.push(Token::Identifier(literal));
                    chars.next();
                }
            }
```

@开头的代表变量，读取一段文本直到它不是数字字母和下划线为止，得到的内容就是变量名。

```
            '@' => {
                chars.next();
                let text = collect_until(&mut chars, |c, _| !c.is_alphanumeric() && c != '_');
                tokens.push(Token::Variable(text));
            }
```

若字符是ASCII 数字字符，则以Number类型的Token加入序列。

```
            token if token.is_ascii_digit() => {
                let num = collect_until(&mut chars, |c, _| !c.is_ascii_digit() && c != '.');
                tokens.push(Token::Number(num));
            }
```

在转换符号之前，我们要为Token和Char写拓展方法，这样我们可以更简洁地判断条件，首先这是判断Token是否是终结符

```
impl Token {
    pub fn is_terminator(&self) -> bool {
        match self {
            Token::Symbol(Symbol::Semicolon) | Token::Symbol(Symbol::Slash) => true,
            _ => false
        }
    }
}
```

然后我们为Char类型写一个trait，分别是判断Char是否是符号类型、转换成符号类型Token、是否有后接符号。

```
// src/datatype/token.rs

pub trait SqlCharExt {
    fn is_symbol(&self) -> bool;
    fn as_symbol(&self) -> Option<Symbol>;
}

impl SqlCharExt for char {
    fn is_symbol(&self) -> bool {
        if to_symbol(&self.to_string().as_str()).is_some() {
            return true
        }
        false
    }
    fn as_symbol(&self) -> Option<Symbol> {
        if let Some(symbol) = to_symbol(&self.to_string().as_str()) {
            return Some(symbol)
        }
        None
    }
}
```

```
// src/datatype/symbol.rs

pub trait SymbolExtChar {
    fn has_next(&self, chars: &mut std::iter::Peekable<std::str::Chars>) -> bool;
}
```

```
impl SymbolExtChar for char {
    fn has_next(&self, chars: &mut std::iter::Peekable<std::str::Chars>) -> bool {
        match self {
            '!' | '<' | '>' => {
                if chars.nth(1).map_or(false, |c| c == '=') {
                    return true;
                }
                return false;
            }
            _ => return false,
        }
    }
}
```

这边解释一下为什么要专门判断符号是否有后接符号，因为有一些Symbol是多个符号组成的，例如`>=`，所以在一般Symbol模式匹配中要使用collect_until，但是有些符号是只能单独存在的，我们如果直接collect_until就会出现问题，例如一个四则运算`(1+2)*3`，`)`和`*`连在了一起，这也会被不严谨的collect_until给收集自一起，然后在转换时就会出现错误（因为没有符号是`)*`）。而有可能存在后接符号的字符只有三个（!><）且只有可能后接=，我们只需要判断下一个符号是不是=就行了。

然后就能写判断过程了：

```
            token if token.is_symbol() => {
                let mut symbol = token.to_string();

                if token.has_next(&mut chars) {
                    symbol.push(chars.next().take().unwrap());
                } else {
                    chars.next();
                }

                if let Some(s) = symbol.as_symbol() {
                    tokens.push(Token::Symbol(s));
                }
            }
```

在最后一步，我们收集字符直到它不再是数字或字母，且它没有可能的后接关键字。我们为String定义几个拓展方法：

```
// src/datatype/token.rs

pub trait SqlStringExt {
    fn is_keyword(&self) -> bool;
    fn is_function(&self) -> bool;
    fn as_keyword(&self) -> Option<Keyword>;
    fn as_symbol(&self) -> Option<Symbol>;
    fn as_function(&self) -> Option<FunctionT>;
```

```
    fn as_bool(&self) -> Option<bool>;
}
```

```
// src/datatype/keyword.rs

pub trait SqlStringExt {
    fn has_suffix(&self) -> bool;
}

...
    fn has_suffix(&self) -> bool {
        match self.to_uppercase().as_str() {
            "GROUP"
            | "ORDER"
            | "INNER"
            | "LEFT"
            | "OUTER"
            | "RIGHT"
            | "FULL" => true,
            _ => false,
        }
    }
...
```

前五个完全按照上面Char的拓展方法改词就行了，没什么必要讲，重点是has_suffix，若收集到的字符是这些一定要后接一个关键字的关键字，那就继续收集，如果没有的话会连带下一个空格被收集然后被当成字面量。正常情况下这样就能收集到一个带有空格的完整Keyword，例如`GROUP BY`，Keyword反而比较简单，因为它们是一定会后带关键字，不像symbol还有单独存在的情况。

很简单了，如果收集到的结果不能作为任何函数/关键字就当成字面量处理。另外别忘了引用单独写的trait。

```
            _ => {
                let text = collect_until(&mut chars, |c, result| !c.is_alphanumeric() && c != '_' && !result.has_suffix() );
                if let Some(function) = text.as_function() {
                    tokens.push(Token::Function(function));
                } else if let Some(keyword) = text.as_keyword() {
                    tokens.push(Token::Keyword(keyword));
                } else if let Some(bool) = text.as_bool() {
                    tokens.push(Token::Bool(bool));
                } else if text.to_uppercase() == "NULL" {
                    tokens.push(Token::Null);
                } else {
                    tokens.push(Token::Identifier(text));
                }
            }
```

最后这个tokens就是返回值了，我们的词法分析部分就完成了，很简单。

语法规则

结束了无聊透顶的简单的词法分析器，来本篇的重点——语法分析，我们先从语法规则讲起吧，先从理论下手。

> 数学公式编辑有些繁琐，这都是阿喵的心血啊www

上下文无关文法

上下文无关语法是本文一个很重要的概念，此处我们先了解它的基本概念，后面我们会详细讨论和大量用到。
文法规则是一种形式化的方法，用于描述一种语言的结构和语法。在计算机科学中，我们通常使用上下文无关文法（Context-Free Grammar）来描述语言的语法规则。

先来了解一下它的形式定义：
一个上下文无关文法是一个四元组 $G = \langle V, \Sigma, P, S \rangle$：

- 一个非终结符集合 V
- 一个终结符集合 Σ
- 一个产生式集合 P，它描述了语法规则中的所有产生式，每个产生式都是 A → α 的规则，其中 A 是一个非终结符号，α 是一个由终结符号和非终结符号组成的字符串。
- 一个起始符号（一定是非终结符号）S

终结符是不可再分割的最小符号，一般会是语言的实际元素（例如数字、特定符号、字面量）。非终结符是能再分割成更小符号的符号，它代表语言中的语法结构，一般是语法成分（表达式、语句等）。
产生式（Production）描述了如何由文法符号生成另一个或多个文法符号的规则，上面我们提到的 P 是产生式集合，其中产生式分为左部和右部，用箭头连接。左部是一个非终结符，右部是终结符或者非终结符。

例如，这是一个非常简单的上下文无关文法示例，可以表示加减运算的表达式。

$$E \to E + E \mid E - E \mid num$$

其中：

- $V = E$
- $\Sigma = \{+, -, num\}$
- $P = \{E \to E + E, E \to E - E, E \to num\}$
- $S = E$

E表示一个表达式，它可以是一个数字 num，两个表达式 E 相加或两个表达式 E 相减。
它之所以叫『上下文无关』文法，是因为它的产生式规则生成的符号串，不依赖于符号串出现的上下文环

境，左部在任何情况下都可以被替换为右部。现在我们假设有一个上下文无关文法，其中E代表（expression），T代表项（term），它的文法规则很简单：

$$\begin{gathered} E \rightarrow T \\ T \rightarrow x \mid y \end{gathered}$$

在这个文法规则中，无论 E 出现在什么位置都能被替换为 T ，T 同理，可以被替换成 x 或者 y ，也就是替换成的过程只取决于这个非终结符本身，这就叫做『上下文无关』。需要注意的是，x 和 y 是终结符，所以它们不能再被替换了。
要强调的是，一个上下文无关文法可以有多个右部，多个右部用 | 符号分割，可以以这样的形式表示。

$$E \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$$

紧接着我们来了解一下另外一种上下文无关文法的，有规范的表示方法。

BNF范式

巴克斯-诺尔范式（Backus-Naur Form，简称BNF）是一种上下文无关文法的表示方法，由约翰·巴克斯（John Backus，上次讲到的图灵奖论文的作者）提出。实际上就是一种定义语法规则的方法，上面的例子用BNF写就是：

$$\begin{aligned} < E > ::= & < E > + < E > \\ & \mid < E > - < E > \\ & \mid num \end{aligned}$$

非终结符使用尖括号<>括起来，它本来就是上下文无关文法的一种，所以大体是一样的。
通过使用BNF范式，我们也可以清晰地描述一个语言的语法，这里了解即可。

语法分析

推导和表示

经过前面的内容，我们已经知道了如何表示语法，所以我们要了解如何进一步的语句展开以便于我们后面继续分析。
推导（Derivation）是从起始符开始，根据产生式规则将非终结符替换为其对应产生式中的任意一个右部，直到推出一个仅包含终结符的符号串的过程。我们再用回刚才的例子。

$$E \rightarrow E + E \mid E - E \mid num$$

输入串和被解析的token序列如下：

$$num + num - num$$

$$num, +, num, -, num$$

第一个token是 num ，符合 $E \rightarrow\ num$，所以我们将这个规则应用于这个token。第二个token是+，符合$E \rightarrow E + E$，同理使用将规则应用于第二个token。此时我们就得到了推导式$num + E$ ，后面三个Token同理，这是整个推导式：

$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow num + E \\ &\rightarrow num + E - E \\ &\rightarrow num + num - E \\ &\rightarrow num + num - num \end{aligned}$$

推导必须具有唯一性，对于一个符合文法规则的输入串，推导过程是唯一的，否则它就是具有二义性（我们后面再讨论）。推导是一个有序的过程，并且可逆，每一步推导都依赖于前一步，所以我们也可以进行反向推导。

我们上面的推导过程叫做最左推导（Leftmost derivation），即每次都选择最左侧的非终结符进行推导。相应的还有一种叫最右推导（Rightmost derivation），即每次都选择最右侧的非终结符进行推导，推导式如下

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E + E - E \\ &\rightarrow E + E - num \\ &\rightarrow E + num - num \\ &\rightarrow num + num - num \end{aligned}$$

两种推导的过程是不唯一的，但是推导结果是一样的。我们可以用 $x \underset{lm}{\Rightarrow} y$ 表示 x 经过一步最左推导得到 y 。例如我们将上面最左推导的第二步 $E \rightarrow num + E$ 带入就是这样的。

$$E + E \underset{lm}{\Rightarrow} num + E$$

最右推导也有相同的表示方法，上面最右推导例子的第二步应该这样表示：

$$E + E - E \underset{rm}{\Rightarrow} E + num - E$$

我们使用 $x \overset{*}{\Rightarrow} y$ 表示 x 经过0步或者多步可以推导出 y，若$x = y$，则可以视为 x 可以通过0步推导的到 y

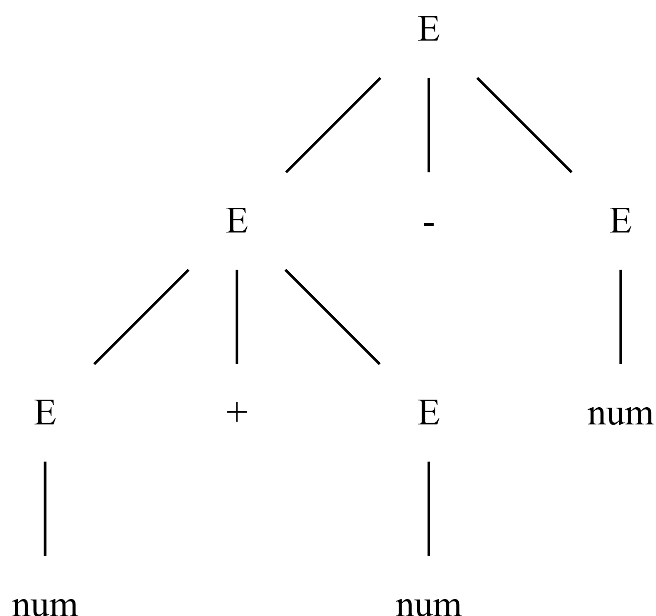
，以上面的最左推导为例，我们可以将它写作：

$$E \underset{lm}{\overset{*}{\Rightarrow}} num + num - num$$

分析树

分析树（Parse Tree）是一种用于表示上下文无关文法（CFG）推导过程的树形结构。其中树根是文法的起始符，叶子节点是文法的终结符，内部节点是文法的非终结符。分析树包含了所有的语法细节（例如终结符与非终结符的关系）和语法规则。通过从根节点到叶子节点的遍历，可以得到输入句子。而从叶子节点到根节点的遍历，则是对输入句子进行分析的过程，每一步都是使用一个产生式将一个非终结符替换为它所对应的符号序列。

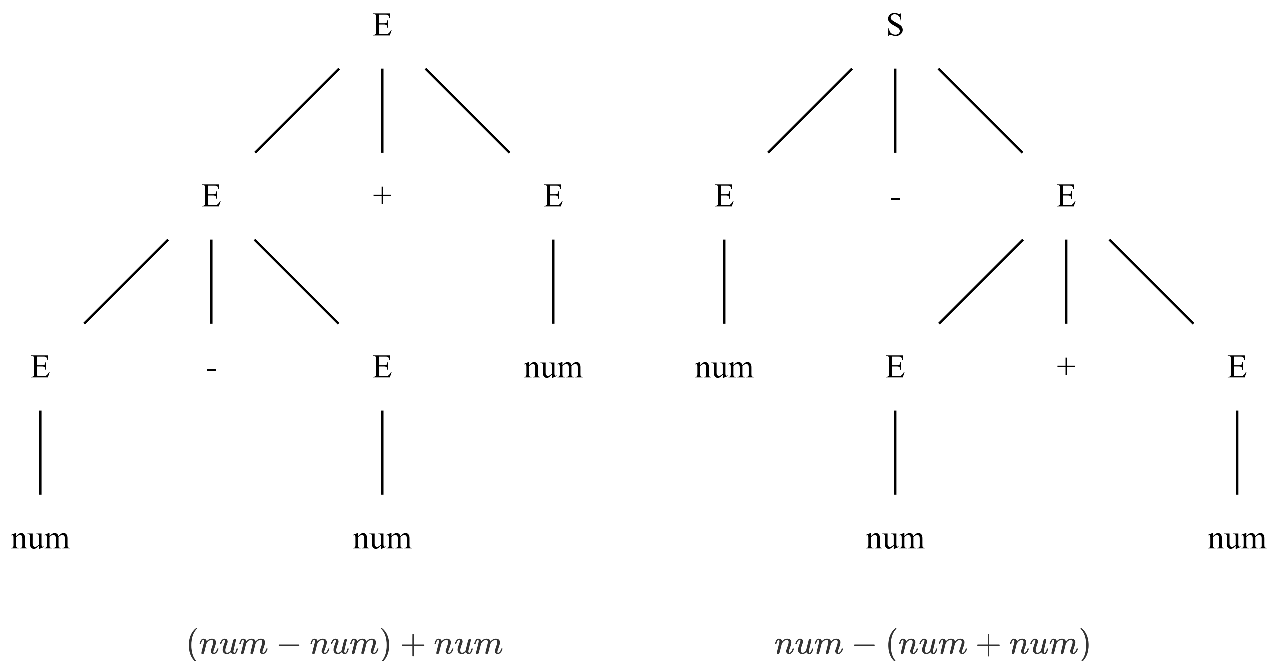
构建分析树实质就是不断展开非终结符，我们在后面将实际尝试如何操作。我们先为刚才的例子建立 $num + num - num$ 一个分析树，根据它的展开我们可以建立出一个这样的分析树：



在文法无歧义的情况下分析树是唯一的，若一个语句存在多个不同的分析树，它就是有二义性的。考虑语句：

$$num - num + num$$

减法和加法优先级是相同的，都是左结合的，所以这个语句有如下两个分析树，左边是左结合的，右边是右结合的。且分别对应这两个语句。



$$(num - num) + num \qquad num - (num + num)$$

二义性文法很容易给后续的工作带来一些问题，所以我们得考虑如何解决它，我们会在后面实践部分详细讨论如何消除二义性。

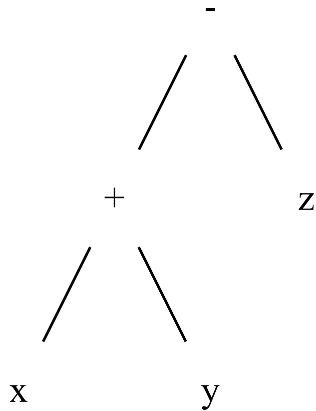
抽象语法树

抽象语法树（Abstract Syntax Tree，AST），是语法分析后生成的树型结构，用于表示程序的语法结构。要注意的是，它和分析树不是一个东西，分析树有一个完整的推导过程，而AST忽略了冗余信息，只保留语法结构中的重要部分。

为了表达更具体，我们将 x, y, z 代入 $num + num - num$来举例，代入后这个语句：

$$x + y - z$$

在AST中只保留语法结构中的重要部分，所以这个语句的AST很简单，只有关键的项和运算符。



AST的节点可以分为这三类：

- 叶子节点（Leaf Node）：表示终结符。叶子节点没有子节点，其属性通常包含了节点的值和数据类型。
- 内部节点（Internal Node）：表示非终结符。内部节点具有子节点，每个子节点表示内部节点所包含的语法元素。
- 根节点（Root Node）：表示整个 AST 的根节点，它包含了整个程序的语法结构，也就是，根节点是特殊的内部节点。

先说一下，我们后面用来解析SQL的一些表达式时，这个AST就是例如运算表达式之类的，语法分析的输出结果。现在我们已经了解了一些基本的语法分析表示了，接下来要一边coding一边细讲一些细节。

数据结构定义

在本章我们将完成一个Select语句的解析器，其中比较具有代表性的就是条件表达式，无论如何我们先建立一个结构来定义Select语句的内容，我们将支持的Select语句参数如下：

```
Select [ ALL | DISTINCT ] { * | column  [ AS alias ] [, ...] } [ FROM from_item [ AS alias ] [, ...] ] [ WHERE condition ]
[GROUP BY column [ AS alias ] [, ...]] [HAVING condition] [ORDER BY column [ASC | DESC] [, ...]]
```

简单解释一下：

- DISTINCT：可选项，表示查询结果是否要去重。
- column：必选项，用于指定要查询的列。
- FROM：必选项，用于指定要查询的表或视图。
- WHERE：可选项，用于指定查询条件，可以使用比较运算符、逻辑运算符和其他操作符来组合条件。
- GROUP BY：可选项，用于指定分组依据，通常与聚合函数一起使用。
- HAVING：可选项，用于对分组后的数据进行进一步的汇总和筛选。

- ORDER BY：可选项，用于指定查询结果的排序方式。

先用结构体结构的枚举类列出来：

```
// src/models/structs.rs

#[derive(Debug)]
pub enum Statement {
    Select {
        distinct: bool,
        projections: Column,
        table: Vec<(Expression, Option<Expression>)>,
        filter: Option<Condition>,
        group_by: Column,
        having: Option<Condition>,
        order_by: Option<Vec<(String, Sort)>>
    }
}
```

其中column是枚举类，因为要考虑到`*`的情况，比起直接使用Vec，会更明确。

```
#[derive(Debug)]
pub enum Column {
    AllColumns,
    Columns(Vec<String>),
}
```

筛选条件可有可无，所以我们使用Option枚举类表示，为了方便表示连续的条件，我们将Condition定义为一个枚举类，其中And、Or、Not是递归类型。
值得注意到是，Rust的类型系统是静态系统，所以在编译器就要确定类型的大小，所以我们不能直接定义递归类型，所以我们要用指针类型Box来定义，你可以将它看作是一个固定大小的指针类型，所以就不需要在意递归类型的大小了。

```
#[derive(Debug, Clone)]
pub enum Condition {
    And {
        left: Box<Condition>,
        right: Box<Condition>,
    },
    Or {
        left: Box<Condition>,
        right: Box<Condition>,
    },
    Not(Box<Condition>),
    Comparison {
        left: Value,
        operator: Symbol,
        right: Expression,
```

```
    }
}
```

这样做有一个好处，就是我们只需要一个Condition就足够了，考虑到用户就只需要递归解析即可。因为等号左边的有可能是变量、字面量、数值、空值等，所以这种数据类型我们同一叫它“Value”，在另外一个源文件定义它并引入，后面这个源文件还有用。

```
// src/models/data.rs

#[derive(Debug, Clone)]
pub enum Value {
    Identifier(String),
    Number(String),
    Variable(String),
    Null,
}
```

Expression只带有一个ast，所以我们先去定义AST的结构好了。AST在定义上没有严格要求节点数，在我们目前的情况，每个节点只需要两个节点即可，后面我们会用到的一些方法，内容是很显然的，所以文章内只展示方法名、参数、返回值，数据结构的定义。

```
// src/models/structs.rs

#[derive(Debug, Clone)]
pub struct Expression {
    pub ast: ASTNode,
}

impl Expression {
    pub fn new(left: ASTNode, symbol: Symbol, right: ASTNode) -> Self ...
    pub fn new_with_ast(ast: ASTNode) -> Self ...
    pub fn new_with_symbol(s: Symbol) -> Self ...
    pub fn new_left(node: NodeType) -> Self ...
    pub fn new_unary_op(s: Symbol, expr: ASTNode) -> Self ...
}
```

```
// src/models/ast.rs

#[derive(Debug, Clone)]
pub struct ASTNode {
    pub node: NodeType,
    pub left: Option<Box<ASTNode>>,
    pub right: Option<Box<ASTNode>>,
}

impl ASTNode {
    pub fn default(node: NodeType) -> Self ...
    pub fn new(node: NodeType, left: Option<Box<ASTNode>>, right: Option<Box<ASTNode>>) -> Self ...
```

```
    pub fn new_node(node: NodeType) -> Self ...
    pub fn set_left(&mut self, node: ASTNode) ...
    pub fn set_right(&mut self, node: ASTNode) ...
}
```

我们还没有做好对函数类型的定义，对于节点类型的定义暂且先这样，Statement（语句）和Function类型的节点应该有Box，因为它们是递归类型。其实后面我们最麻烦和讨厌的就是递归来递归去。

```
#[derive(Debug, Clone)]
pub enum NodeType {
    Statement(Box<Statement>),
    Symbol(Symbol),
    Value(Value),
    Function(Box<Function>),
}
```

你可能注意到了我们没有在这里区分什么叶子节点内部节点，它们都是NodeType，我们前面提到了AST中的节点是有分类型的，但是如果我们在该步骤就把它们分开的话语法分析会很麻烦，而我们已经将必要区分的数据类型区分开了，所以这个问题交给后面的类型检查来完成就好了。

函数的分析

我们先要解决一个分析起来相对麻烦的类型——函数。说它相对的麻烦原因是，这玩意不像其他类型直接解析成Token就行了，它可能有各种各样的参数、有参数类型和长度要求，所以不是在词法分析器就能解决的。不过词法分析器已经帮我们做的尽可能多了，它已经帮我们分析好Function的Token了，所以我们可以很简单地知道『接下来要分析的是函数』。我们假设这是一个表达式和它被lexer分析出的Token序列：

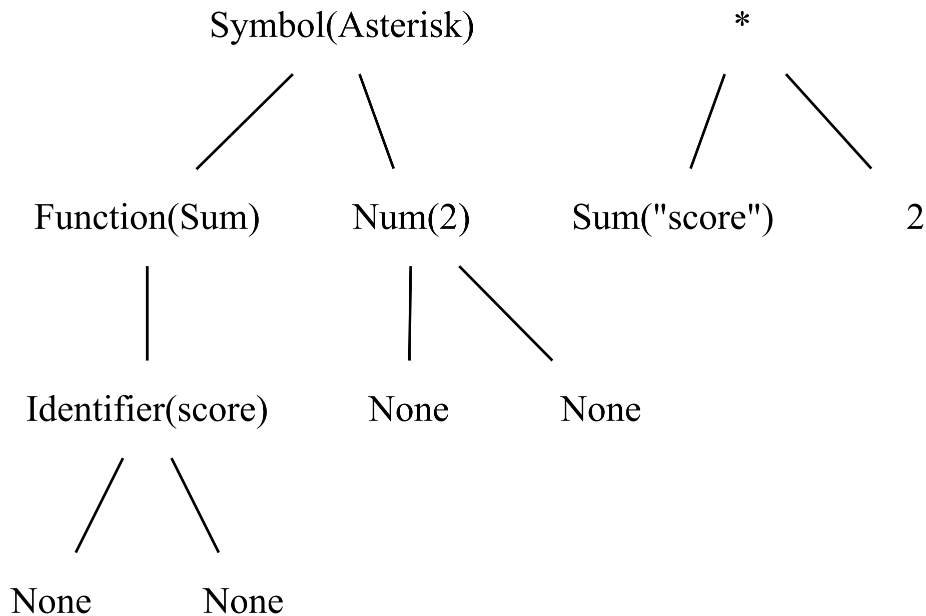
$$Sum(score) * 2$$

```
FunctionT(Sum), Symbol(LeftParen), Identifier("score"), Symbol(RightParen), Symbol(Asterisk), Num(2)
```

函数不应该直接被解析成带有值的Token，这样的话如果出现语法错误还得让Lexer做错误处理，太讨厌了，所以干脆让函数的关键字用Keyword的同样套路被解析，其他的就爱咋咋吧，所以你会看到括号和里面的参数都是分开的Token。

> 并不能说无法，而是其中的一些分析过程更应该属于语法分析

NodeType有Function类型，所以我们可以直接将携带值的函数作为节点，所以我们期望的AST很简单，下图左边是每个节点在程序中的数据类型，右边是它更具有可读性的样子。



在标准的SQL中，函数的参数是支持表达式的，所以函数参数的类型是Expression而不是Value。分析的结果应该是一个带有参数的函数，我们用一个枚举类列出支持的函数：

```
// src/models/ast.rs

use super::structs::{Statement, Expression};


#[derive(Debug, Clone)]
pub enum Function {
    Sum(Expression),
    Avg(Expression),
    Count(Expression),
    Max(Expression),
    Min(Expression),
    Concat(Vec<Expression>),
}
```

这里是我做的不够好，按照内容来说ast.rs和structs.rs应该是一个源文件的，但是我为了更方便区分就分开了，导致ast不得不引用structs的一些内容（本来我打算是structs单方面依赖ast的），我对这玩意实在有点强迫症，所以很在意。

为FunctionT写一个方法获得参数数量，为0时代表2个以上（多个）

```
// src/datatype/function.rs

impl FunctionT {
    pub fn arg_len(&self) -> u8 {
        match self {
```

```
            Self::Sum
            ...
            | Self::Min => 1,
            Self::Concat => 0,
        }
    }
}
```

考虑到方便检查参数数量，新建Function的方法统一以参数序列传入，若只需要一个参数则取第一个元素即可，根据函数来选择参数。为了处理参数数量错误的情况，我们先定义一个错误类，并且将它定义为ParseError的子类。

```
// Cargo.toml

[dependencies]
thiserror = "1.0.24"
```

```
// src/models/error.rs

use thiserror::Error;

#[derive(Error, Debug)]
pub enum StructError {
    #[error("Incorrect number of args: expect {0}")]
    IncorrectArgCount(u8),

    #[error("Incorrect number of args: expect {0} or more")]
    ExpectMoreArg(u8),
}
pub type Result<T> = std::result::Result<T, StructError>;
```

```
// src/parser/error.rs

use thiserror::Error;

#[derive(Error, Debug)]
pub enum ParseError {
    #[error("{0}")]
    StructError(#[from] StructError),

    #[error("Unexpected token: '{0}'")]
    UnexpectedToken(Token),

    #[error("Missing token: '{0}'")]
    MissingToken(Token),
}

pub type Result<T> = std::result::Result<T, ParseError>;
```

在new一个Function时就要检查参数数量，若不符合预期就返回错误。除了参数量和函数不等的情况，还要考虑函数预期2个或多个参数的情况，因为我们用0表示，所以当参数量不符合条件且为0时则需要返回ExpectMoreArg错误，注意两个错误类型的参数是它们的期望值而不是得到的值。

```
// src/models/ast.rs

impl Function {
    pub fn new(function: FunctionT, args: Vec<Expression>) -> Result<Self> {
        let arg_len = function.arg_len();
        if (args.len() != arg_len.into() && arg_len != 0) || (arg_len == 0 && args.len() < 2) {
            if arg_len == 0 {
                return Err(StructError::ExpectMoreArg(2));
            }
            return Err(StructError::IncorrectArgCount(arg_len));
        }

        Ok(match function {
            FunctionT::Sum => Self::Sum(args[0].clone()),
            ...
            FunctionT::Concat => Self::Concat(args.clone()),
        })
    }
}
```

先简单写一个函数，判断期望的token，若不期望则返回错误，这样我们可以直接用?来调用它。然后让函数parse_function最开始先检查函数关键字和左括号。

```
// src/parser/expression_parser.rs

fn parse_function(iter: &mut Peekable<IntoIter<Token>>) -> Result<Function> {
    let function = match iter.peek() {
        Some(Token::Function(f)) => f.clone(),
        Some(t) => return Err(ParseError::UnexpectedToken(t.clone())),
        _ => return Err(ParseError::MissingFunction),
    };
    iter.next();
    match_token(&iter.next(), Token::Symbol(Symbol::LeftParen))?;
}

fn match_token(value: &Option<Token>, expect: Token) -> Result<()> {
    return match value {
        Some(_) => Ok(()),
        None => return Err(ParseError::MissingToken(expect))
    }
}
```

然后我们只需要解析函数参数的序列即可，函数参数用逗号分隔，当遇到逗号时消耗掉它本身继续，我们假设有一个函数 parse_expression 可以正确解析一个合格的表达式并消耗掉应该消耗掉的括号，那我们只需要循环调用直到遇到右括号为止即可。最后将Token和参数列表传入Function::new方法即可（它已经自带了参

数检查）。

```
loop {
    match iter.peek() {
        Some(Token::Symbol(Symbol::RightParen)) => break,
        _ => ()
    }
    match parse_expression(iter) {
        Ok(e) => {
            args.push(e);
            if let Some(Token::Symbol(Symbol::Comma)) = iter.peek() {
                iter.next();
            }
        },
        Err(e) => return Err(e),
    }
}
iter.next();
return Ok(Function::new(function, args)?);
```

要注意的是在这个源文件使用的错误类型是ParserError，所以在匹配参数时错误的话会被包装在ParserError::StructError里面返回而不是一个单独的StructError，所以千万不要搞错这两个错误类型的Result别名。这两个源文件不在同一个父目录，一般你没有理解错误的情况也不会搞混。

自顶向下分析简介

自顶向下分析是一种语法分析方式，从文法的开始符号开始，递归尝试将输入符号串匹配到文法中的某个产生式直到匹配到终结符，或者无法进一步匹配。

我们先来简单地从理论了解递归下降分析，以便后续的工作。递归下降分析是自顶向下分析的一种具体实现，基本思想就是每个函数对应一个非终结符并递归调用分析。其实很简单的道理，我们从一个简单的例子开始，之前的文法：

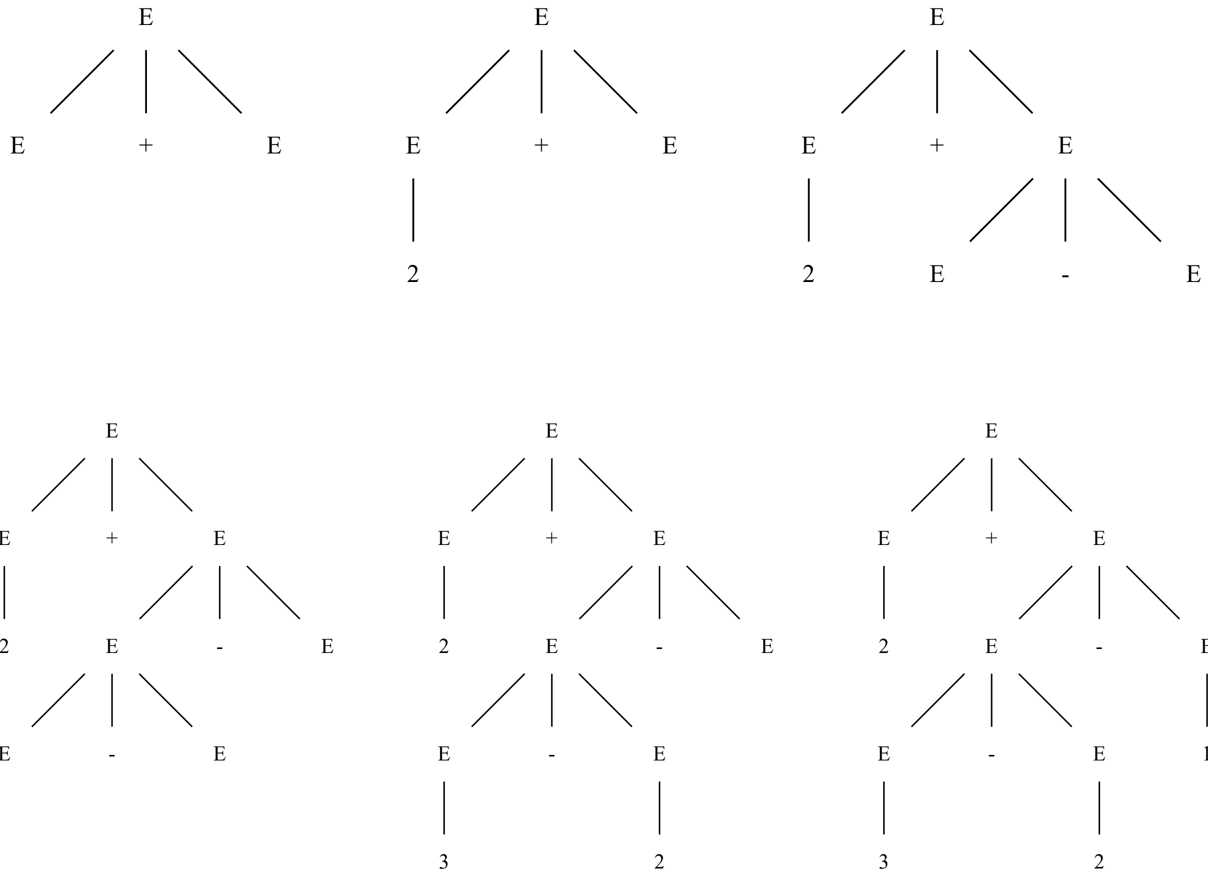
$$E \to E + E \mid E - E \mid num$$

考虑输入和其最左推导步骤：

$$2 + (3 - 2) - 1$$

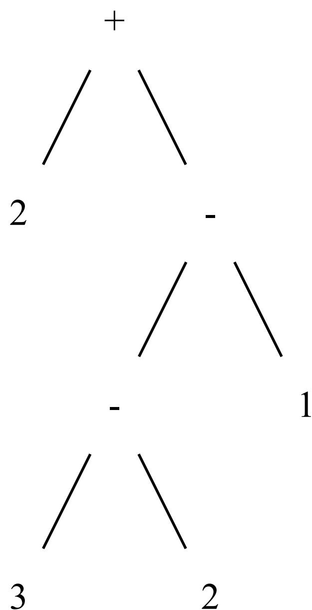
$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow 2 + E \\
&\rightarrow 2 + E - E \\
&\rightarrow 2 + (E - E) - E \\
&\rightarrow 2 + (3 - E) - E \\
&\rightarrow 2 + (3 - 2) - E \\
&\rightarrow 2 + (3 - 2) - 1
\end{aligned}
$$

在推导中，第二个表达式 E 是非终结符，我们就展开它并且继续推导直到推导到终结符再继续，这个展开继续推导的方法就叫做递归下降分析。分析树分析过程如下，其中第4-5是 $2 + (E - E) - E \underset{lm}{\overset{*}{\Rightarrow}} \rightarrow 2 + (3 - 2) - E$，两步推导出终结符，分开的话得画7个树就还是这样好了XD



在递归下降分析中，我们遇到为非终结符的展开，则继续递归分析直到展开到终结符，正如上图的第3、4步，将一个需要继续展开的E展开为$E - E$ （符合文法定义），并**始终选择最左**的非终结符继续展开，直到全部展开为终结符。

所以，推导后，去除冗余信息，最后这个输入的抽象语法树长这样：



我们上面例子的推导方式是最左推导，但是并不属于最左递归下降分析，我们后面会较具体的了解左递归。

表达式处理

做表达式处理时，我们会面对很多常见的语法分析问题，所以这一小节会概括很大一部分关键的理论知识，我觉得一边写代码一边讲才是最好的。

在SQL中，表达式可以是变量、数值、字面量、函数等，其中的类型要求会根据情况细分，而类型检查是单独的一个功能，所以在现阶段我们只需要想着分析就好了。对于四则运算，我们可以写出一个很直观的上下文无关文法：

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid value$$

此处value代表没有子节点的Node，数据在NodeType携带，NodeType的Value类型一定会是终结符。Function的参数是Expression，所以它里面还会带有对应数量的表达式里的AST，但是Function自身是没有子节点的。总之这些没有子节点的东西我们统称为 $value$ 。

> 这里的value含义并不严谨，更准确的说应该叫表达式，但是这样会有更多歧义，姑且先这么叫吧。

但是分析器并不知道什么四则运算优先顺序，它只认识这个上下文无关文法，所以我们不能直接用那套直接推导，我们要再改写文法：

$$E->E+T \mid E-T \mid T$$
$$T->T*F \mid T/F \mid F$$
$$F->(E) \mid value$$

其中 E 是表达式（Expression）， T 表示项（Term）， F 表示因子（Factor），这样改写是因为四则运算是由表达式、项和因子构成的，其中表达式是最高级别的，包含了项和表达式的运算，项包含了因子和项的运算，因子则是最基本的单位，可以是数字或者带括号的表达式。
这个文法用BNF范式可以这样写

$$
\begin{aligned}
&expr ::= < expr > + < term > \mid < expr > - < term > \mid < term > \\
&term ::= < term > * < factor > \mid < term > / < factor > \mid < factor > \\
&factor ::= (< expr >) \mid value
\end{aligned}
$$

大概就是那么简单，我们先写出并不困难的实现再分析和讨论这个实现方式。我们在前面已经定义了Expression等的数据结构，$factor$ 在文法定义中是表达式中的基本元素，所以我们可以很简单地写出一个函数解析表达式中最基本的元素。只需要用一个迭代器并根据Token类型进行相应的处理。但是对于左括号和加减符号这种需要递归下降分析的处理函数还没写好，暂时先留todo。

```
// src/parser/expression_parser.rs

fn parse_factor(iter: &mut Peekable<IntoIter<Token>>) -> Result<Expression> {
    if let Some(token) = iter.peek() {
        let result = match token {
            Token::Identifier(ref s) => Ok(Expression::new_left(NodeType::Value(Value::Identifier(s.clone())))),
            Token::Number(ref s) => Ok(Expression::new_left(NodeType::Value(Value::Number(s.clone())))),
            Token::Variable(ref v) => Ok(Expression::new_left(NodeType::Value(Value::Variable(v.clone())))),
            Token::Function(_) => {
                let function = parse_function(iter)?;
                return Ok(Expression::new_left(NodeType::Function(Box::new(function))));
            },
            Token::Symbol(Symbol::LeftParen) => todo!()
            Token::Symbol(Symbol::Plus) | Token::Symbol(Symbol::Minus) => todo!()
            _ => Err(ParseError::UnexpectedToken(token.clone())),
        };
        iter.next();
        return result;
    } else {
        return Err(ParseError::IncorrectExpression);
    }
}
```

$term$ 在数学中就是指一个项，所以parse_next_term函数的作用是解析乘除法表达式中的项。解析过程很简单，首先解析出下一个 $factor$ ，再循环检查下一个 Token 是否为乘除号。如果是，则解析出下一个 $factor$ ，并用已有的“左表达式”和当前的符号构建一个新的Expression。如果不是，则跳出循环，返回这个“左表达式”。

```
// src/parser/expression_parser.rs

fn parse_next_term(iter: &mut Peekable<IntoIter<Token>>) -> Result<Expression> {
    let mut left_expr = parse_factor(iter)?;
    while let Some(token) = iter.peek() {
        let symbol = match token {
            Token::Symbol(s) => s.clone(),
            _ => break,
        };
        match symbol {
            Symbol::Asterisk | Symbol::Slash => {
                iter.next();
                let right_expr = parse_factor(iter)?;
                left_expr = Expression::new( left_expr.ast, symbol, right_expr.ast );
            }
            _ => break,
        }
    }
    Ok(left_expr)
}
```

注意构建表达式时用的是它里面的AST，而不是直接传入表达式本身。

最后的 $expr$ 的解析函数负责解析加减法表达式，再循环检查下一个 Token是否是加减号，如果是，则解析出下一个 $term$，则已有的“左表达式”和当前的符号构建一个新的Expression。否则跳出循环并返回这个“左表达式”。

```
// src/parser/expression_parser.rs

pub fn parse_expression(iter: &mut Peekable<IntoIter<Token>>) -> Result<Expression> {
    let mut left_expr = parse_next_term(iter)?;
    while let Some(token) = iter.peek() {
        let symbol = match token {
            Token::Symbol(s) => s.clone(),
            _ => break,
        };
        match symbol {
            Symbol::Plus | Symbol::Minus => {
                iter.next();
                let right_expr = parse_next_term(iter)?;
                left_expr = Expression::new( left_expr.ast, symbol, right_expr.ast );
            }
            _ => break,
        }
    }
    Ok(left_expr)
}
```

实质上在 $term$ 的解析函数中， $left_expr$ 只是一个包含一个表示因子的AST的Expression，在更高层的调用中需要的只是AST本身，Expression类型本身是只有一个AST的，我用这样的类型包装它是为了可以更直

观的表示数学表达式，而不是把AST传来传去，同时这种做法限制了AST的结构，在之后我们可以用来更方便地检查类型。同理，在 $expr$ 的解析函数中，它也只是一个包含一个表示项的AST的Expression，不过到这一步这个Expression就是被完整解析的表达式了。

再回到 $factor$ 的处理函数：

```
            Token::Symbol(Symbol::LeftParen) => {
                iter.next();
                let expr = parse_expression(iter)?;
                return match iter.next() {
                    Some(Token::Symbol(Symbol::RightParen)) => Ok(expr),
                    _ => Err(ParseError::MissingToken(Token::Symbol(Symbol::RightParen))),
                };
            }
            Token::Symbol(Symbol::Plus) | Token::Symbol(Symbol::Minus) => {
                let t = token.clone();
                iter.next();
                let expr = parse_factor(iter)?;
                return Ok(Expression::new_unary_op(
                    t.as_symbol().take().unwrap(),
                    expr.ast
                ));
            }
```

一个合格的括号里会包含一个表达式，所以直接递归调用即可。而当在解析因子时遇到加/减，则将下一因子解析为一元加/减的表达式，这样也就支持了有负号的表达式的分析。

二义性问题

我们前面在分析树那已经提到过二义性问题了，而二义性文法是很常见的，例如我们前面一开始就提到的文法就是一个典型。

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid value$$

所以我才会说『分析器并不知道什么四则运算优先顺序，它只认识这个上下文无关文法，所以我们不能直接用那套直接推导』，这就是一个二义性文法。我们通过添加运算符优先级、指定结合性规则来消除二义性，具体的说，改写的文法：

$$\begin{aligned} E &- > E + T \mid E - T \mid T \\ T &- > T * F \mid T / F \mid F \\ F &- > (E) \mid value \end{aligned}$$

在改写的文法中，我们先是定义了运算符优先级，加减法的优先级低于乘除法，在递归调用时，优先级较高

的乘除法（即 $term$ ）总是会被优先处理。然后是定义了结合性规则，用于确定相同优先级的操作符在表达式中如何组合，我们定义了的加减法和乘除法都是做结合，这样的话分析其就能根据我们定义的规则来正确分析表达式，避免二义性。

除了四则运算以外，函数法分析往往也是二义性的重灾区，我们前面函数分析方式使用括号明确指定了参数的范围和个数，以及使用严格的token匹配来避免函数调用语句的二义性。

> 个人觉得可以去龙书看看“悬空else”问题更好理解二义性

左递归问题

在递归下降分析法中，直接按照左递归文法规则解析的话会导致递归调用陷入死循环，当然这是有很多方法可以避免的，所以我将介绍这个问题、常见的解决方法和我们表达式处理的方法。
在一个产生式中，左侧非终结符号能够直接或间接地推导出自身，那这就是一个左递归的文法，例如：

$$E \to E + \alpha \mid \alpha$$

产生式的左部非终结符 E 出现在了右部的第一位，且可以通过 E 推导出它本身，如果我们直接按照这个文法规则进行递归下降分析，则会陷入死循环。首先最简单的，我们可以考虑将其替换为等价的非左递归的产生式，将文法改写为：

$$\begin{aligned} E &\to \alpha\ E' \\ E' &\to +\ \alpha\ E' \mid \varepsilon \end{aligned}$$

实质上就是将这个左递归产生式拆分成 $E \to \alpha\ E'$ 和 $E' \to +\ \alpha\ E' \mid \varepsilon$ 两个产生式，其中 ε 表示一个不产生任何终结符的产生式（即空串）。在这个新产生式中，非终结符 E' 出现在了右侧的第二个位置，因此它不再是左递归的产生式，而是一个右递归产生式。按照这样的思路，右递归并不是不会导致递归调用的死循环，而是对于自顶向下分析法而言，这能保证递归调用发生在产生式的末尾，避免陷入死循环后还要回溯处理。
我们进行表达式解析的上下文无关文法也是一个左递归文法。

$$\begin{aligned} &E->E+T \mid E-T \mid T \\ &T->T*F \mid T/F \mid F \\ &F->(E) \mid value \end{aligned}$$

但是我们在parse_expression和parse_next_term函数中，使用循环来代替原先的递归调用，并在循环中依次处理每个产生式右部的项，从而实现了右递归的语法规则，也就是将左递归产生式转换为等价右递归产生式，以避免左递归问题，上述文法被转换成了这样的等价右递归文法：

$$
\begin{aligned}
&E->T\ E' \\
&E'->+T\ E'\ |\ -\ T\ E'\ |\ \varepsilon \\
&T->F\ T' \\
&T'->*\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon \\
&F->(E)\ |\ value
\end{aligned}
$$

在我们之前的左递归文法 $E \to E + \alpha \mid \alpha$ 严格来说算是直接左递归，即产生式右部的第一个符号是产生式左部本身，属于一种左递归。还有一种左递归叫做间接左递归，即通过一系列产生式会间接导致左递归的产生式，比如：

$$
\begin{aligned}
&E->T\alpha\ |\ \beta \\
&T->E\gamma\ |\ \delta
\end{aligned}
$$

这样的间接左递归文法同样会为自顶向下的语法分析带来麻烦，所以要用同样的方式消除它的左递归。
综上所述，我们需要实现确定的自顶向下分析，对于文法有要求无二义性和无左递归。

预测分析

预测分析是递归下降分析法的一种，它通过使用预测分析表来指导语法分析过程，根据当前输入符号预测应该使用哪个产生式进行推导，以实现对输入的分析。
按照之前对上下文无关文法的形式定义，一个上下文无关文法是一个四元组 $G = \langle V, \Sigma, P, S \rangle$，我们为文法 G 构建预测分析表 M，其中 $M[A, \alpha]$表示在分析非终结符 A 时，若输入符号为 α 时则应用产生式 $P[A->\alpha]$ 进行推导。

首先是一个很重要的概念：FIRST集合，指一个文法中每个非终结符号的可能首字符集合，再次考虑表达式文法，这是一个无二义性，无左递归的LL文法，即从左到右扫描，从左到右推导，最左推导。

$$
\begin{aligned}
&E->T\ E' \\
&E'->+T\ E'\ |\ -\ T\ E'\ |\ \varepsilon \\
&T->F\ T' \\
&T'->*\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon \\
&F->(E)\ |\ value
\end{aligned}
$$

对于诸如+-*$value$等的终结符，它的FIRST就是它本身。然后：

- 非终结符E'，它有三个产生式（右部），其中包含终结符+-和空串，所以E的FIRST集合为$\{+, -, \varepsilon\}$

- 非终结符T'有三个产生式，其中包含终结符*/和空串，所以T的FIRST集合为 $\{*, /, \varepsilon\}$
- 非终结符F有两个产生式，其中(E)的产生式中第一个字符为终结符(，所以取它本身，而$value$是终结符，所以也可以直接取它本身，所以F的FIRST集合为$\{(, value\}$
- 非终结符 E, T 的右部经过推导后得到的都是 F 的FIRST集合，所以它们的FIRST集合为 $\{(, value\}$

然后是FOLLOW集合，指文法某个非终结符号推导时，它后面可能出现的终结符号集合。在接下来的举例中，符号 $ 代表输入串的结束符。

想了一段时间如何清晰简短的表述FOLLOW集合的计算方法，ググ过一些中文的Blog，太多讲一半的或是偷换概念的（恐怕作者自己都没搞懂）那种。这里引用一个 *Compiler Design Tutorial* 的例子，个人觉得是比较通俗清晰的：

> 1. FOLLOW(S) = { $ } // where S is the starting Non-Terminal
> 2. If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).
> 3. If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
> 4. If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

这边要补充解释一下，引用文中的Є和ε都是epsilon的小写形式，严格来说Є是西里尔字母，而ε是希腊字母，在这个场景下它们可以互换使用。

- 非终结符E 是文法的起始符，所以表示结束符的$必须加入进去，并且在 $F->(E) \mid value$ 中，它后面有个终结符)，所以它的FOLLOW集合为 $\{\$,)\}$
- 非终结符E'同理为 $\{\$,)\}$
- 因为 $E'->+T\ E' \mid -\ T\ E' \mid \varepsilon$ 是一个产生式，且 $FIRST(E')$ 包含ε，所以 $FOLLOW(T)$ 包含 $\{FIRST(E') - \varepsilon\} \cup\ FOLLOW(E')$，参考规则4，即 $\{+, -, \$,)\}$
- 非终结符 T' 同理为 $\{+, -, \$,)\}$
- 因为 $T'->*\ F\ T' \mid /\ F\ T' \mid \varepsilon$ 是一个产生式，且 $FIRST(T')$ 包含ε，所以 $FOLLOW(F)$ 包含 $\{FIRST(T') - \varepsilon\} \cup\ FOLLOW(T')$，即 $\{*, /, +, -, \$,)\}$

它们的构造目的很显然，在理论中这两个集合算完便可以构造一个预测分析表进行语法分析。我们再往回看之前的parse_factor函数，虽然因为它没有直接构建分析表，不符合预测分析的定义，但是它通过查看迭代器下一个元素再预测下一步如何展开和推导，所以它也算是运用了预测分析的思想，只不过是手写匹配和对应的展开。严格来说它算一个基于LL文法的自顶向下语法分析。

预测分析好处还是很多的，能在早期阶段就检测到语法错误，能更高效的分析，不过要记住的是这玩意仅限于LL文法这种特殊的上下文无关文法。

在分析方面的理论知识已经写了不少了...当然只是围绕我们的表达式文法，龙书虎书还有提到很多其他分析方法和文法类型，以后如果有机会就用它们来写东西再讲吧（不要乱挖坑啊喂）。最后我们把一些简单的分析写完即可完成实战部分。

条件表达式分析

我们前面已经定义好了条件表达式的数据结构，其中没有嵌套关系的是Comparison，也就是最直接的条件表达式，所以我们先单独写一个函数解析Comparison。只要依次匹配并解析三个字段即可，而且很方便的是，解析表达式已经自带了各种嵌套的处理，所以我们只需要解析即可。

```
// src/parser/clause_parser.rs

fn parse_comparison(iter: &mut Peekable<IntoIter<Token>>) -> Result<Condition> {
    let left = match iter.peek() {
        Some(Token::Identifier(_)) | Some(Token::Symbol(Symbol::LeftParen))| Some(Token::Variable(_)) | Some(Token::Function(_)) =>
        Some(t) => return Err(ParseError::UnexpectedToken(t.clone())),
        None => return Err(ParseError::MissingComparator),
    };
    let operator = match iter.peek() {
        Some(Token::Symbol(t)) if t.is_comparator() => t.clone(),
        Some(t) => return Err(ParseError::UnexpectedToken(t.clone())),
        None => return Err(ParseError::MissingComparator),
    };
    iter.next();
    let right = parse_expression(iter)?;
    Ok(Condition::Comparison {
        left,
        operator,
        right,
    })
```

```
// src/parser/error.rs

    #[error("Missing comparator")]
    MissingComparator,
```

对于AND、OR、NOT这种嵌套的条件表达式，我们又要写 ~~好写爱写~~ 的循环处理，但是有了前面Expression解析的经验，这个很简单，其实就是一回事。但是在token名的部分你要稍微注意一下所有权问题。
take().unwrap()，我承认已经有良好错误处理习惯的你看到它会难受，但是编译器不敢确定这个时候Left是Some类型而不是None，所以我们只能通过这种方式强行将它剥离Option，但是在正常的程序逻辑下这个事情是不会发生的，如果Left是None，在之前就已经返回一个错误了。

```
// src/parser/clause_parser.rs

fn parse_condition(iter: &mut Peekable<IntoIter<Token>>) -> Result<Condition> {
    let mut left: Option<Condition> = None;

    while let Some(token) = iter.peek() {
        match token {
            Token::Keyword(Keyword::And) | Token::Keyword(Keyword::Or)| Token::Keyword(Keyword::Not) => {
                if !left.is_some() {
                    return Err(ParseError::IncorrectCondition);
                }
                let current_token = token.clone();
                iter.next();
                let next_condition = parse_condition(iter)?;
                left = match current_token {
                    Token::Keyword(Keyword::And) => {
                        Some(Condition::And {
                            left: Box::new(left.take().unwrap().clone()),
                            right: Box::new(next_condition)
                        })
                    },
                    Token::Keyword(Keyword::Or) => {
                        Some(Condition::Or {
                            left: Box::new(left.take().unwrap().clone()),
                            right: Box::new(next_condition)
                        })
                    },
                    Token::Keyword(Keyword::Not) => Some(Condition::Not(Box::new(next_condition))),
                    _ => return Err(ParseError::UnknownError),
                };
            },
            ...
    }
}
```

```
// src/parser/error.rs

    #[error("Incorrect condition")]
    IncorrectCondition,
    #[error("Unknown error")]
    UnknownError,
```

在这段代码中UnknownError是不会发生的，但是编译器不知道，所以我们只能创造这样的一个错误类型给他匹配，但是如果这个错误真的在这里发生了，那程序逻辑就有问题了。

这是剩下的类型匹配，注意这里要自己消耗掉左右括号。

```
            Token::Symbol(Symbol::LeftParen) => {
                iter.next();
                let next_condition = parse_condition(iter)?;
                if let Some(Token::Symbol(Symbol::RightParen)) = iter.next() {
```

```
                left = Some(next_condition);
            } else {
                return Err(ParseError::MissingToken(Token::Symbol(Symbol::RightParen)));
            }
        },
        token if token.is_terminator() => break,
        Token::Symbol(Symbol::RightParen) | Token::Keyword(_) => break,
        Token::Symbol(_) | Token::Number(_) => {
            return Err(ParseError::UnexpectedToken(token.clone()));
        }
        Token::Identifier(_) | Token::Variable(_) | Token::Function(_) | Token::Bool(_) => {
            left = Some(parse_comparison(iter)?);
        }
        t => return Err(ParseError::UnexpectedToken(t.clone())),
    }
}

if let Some(r) = left {
    return Ok(r);
}
return Err(ParseError::IncorrectCondition);
```

子句分析

在SQL中允许表名和列名以AS作为关键字定义别名，而且还要支持表达式，所以我们专门为此写一个函数来处理有AS的表达式，后面的就会很简单。

另外记得实现Keyword的is_clause()方法，如果是子句关键字就返回true，否则返回false。这个函数没什么特殊的，只是在解析expression的基础上做了别名解析。

```
fn parse_items_with_alias(iter: &mut Peekable<IntoIter<Token>>) -> Result<Vec<(Expression, Option<Expression>)>> {
    let mut columns = Vec::new();
    loop {
        match iter.peek() {
            Some(Token::Keyword(k)) if k.is_clause() => break,
            Some(s) if s.is_terminator() => break,
            _ => ()
        }
        match parse_expression(iter) {
            Ok(e) => {
                let mut alias = None;
                if let Some(Token::Keyword(Keyword::As)) = iter.peek() {
                    iter.next();
                    alias = Some(parse_expression(iter)?);
                }
                columns.push((e, alias));
                if let Some(Token::Symbol(Symbol::Comma)) = iter.peek() { iter.next(); }
            },
            Err(e) => return Err(e),
        }
    }
```

```
    return Ok(columns);
}
```

这个函数只是把的结果包装成列的数据结构，要解析列的地方不止一处，所以我们单独定义它。

```
fn parse_items_with_alias(iter: &mut Peekable<IntoIter<Token>>) -> Result<Column> {
    Ok(Column::Columns(parse_items_with_alias(iter)?))
}
```

然后我们就能完成解析最开始的选定列的函数：

```
pub fn parse_projection(iter: &mut Peekable<IntoIter<Token>>) -> Result<Column> {
    if let Some(Token::Symbol(Symbol::Asterisk)) = iter.peek() {
        iter.next();
        return Ok(Column::AllColumns);
    }
    parse_columns(iter)
}
```

解析表同理，但是选定表的数量不能为0，至少为1，错误类型在ParseError补充定义即可。

```
pub fn parse_tables(iter: &mut Peekable<IntoIter<Token>>) -> Result<Vec<(Expression, Option<Expression>)>> {
    match iter.peek() {
        Some(Token::Keyword(Keyword::From)) => (),
        _  => return Err(ParseError::MissingToken(Token::Keyword(Keyword::From))),
    }
    iter.next();
    let tables = parse_items_with_alias(iter)?;
    if tables.len() == 0 { return Err(ParseError::MissingTable); }
    Ok(tables)
}
```

两个解析条件的语句就很简单了，我们只需要调用之前写的函数即可，但是要考虑到没有条件的情况所以有None的存在。

```
pub fn parse_where(iter: &mut Peekable<IntoIter<Token>>) -> Result<Option<Condition>> {
    match iter.peek() {
        Some(Token::Keyword(Keyword::Where)) => iter.next(),
        _  => return Ok(None),
    };
    let condition = parse_condition(iter)?;
    return Ok(Some(condition))
}

pub fn parse_having(iter: &mut Peekable<IntoIter<Token>>) -> Result<Option<Condition>> {
    match iter.peek() {
        Some(Token::Keyword(Keyword::Having)) => iter.next(),
```

```
        _  => return Ok(None),
    };
    let condition = parse_condition(iter)?;
    return Ok(Some(condition))
}
```

GROUP BY子句也是只需要解析即可。

```
pub fn parse_groupby(iter: &mut Peekable<IntoIter<Token>>) -> Result<Column> {
    match iter.peek() {
        Some(Token::Keyword(Keyword::GroupBy)) => (),
        _  => return Ok(Column::AllColumns),
    }
    iter.next();
    parse_columns(iter)
}
```

相对麻烦一点点的是ORDER BY子句，不过稍微动下脑子就知道结果是一个二元组序列即可，一个列名对应一个排序枚举类。记得补充错误类型，在匹配的时候要稍微注意一下所有权规则就行了。我稍微看了下其他SQL实现，有些的ORDER BY子句支持表达式，有些不支持，但是既然我们前面已经引入了列名的表达式支持和别名，我觉得这里先支持字面量的列名就够了。

```
pub fn parse_orderby(iter: &mut Peekable<IntoIter<Token>>) -> Result<Option<Vec<(String, Sort)>>> {
    match iter.peek() {
        Some(Token::Keyword(Keyword::OrderBy)) => iter.next(),
        _  => return Ok(None),
    };

    let mut order_by: Vec<(String, Sort)> = Vec::new();

    loop {
        match iter.peek() {
            Some(Token::Identifier(name)) => {
                let current_name = name.clone();
                iter.next();

                match iter.next() {
                    Some(t) => {
                        let sort = match t {
                            Token::Keyword(Keyword::Asc) => Sort::ASC,
                            | Token::Keyword(Keyword::Desc) => Sort::DESC,
                            _ => return Err(ParseError::UnexpectedToken(t.clone())),
                        };
                        let tuple = (current_name, sort);
                        order_by.push(tuple);
                    },
                    None => return Err(ParseError::MissingSort)
                }
            },
```

```
            Some(Token::Symbol(Symbol::Comma)) => {
                iter.next();
                continue;
            },
            Some(token) if token.is_terminator() => break,
            Some(token) => return Err(ParseError::UnexpectedToken(token.clone())),
            None => return Err(ParseError::MissingColumn)
        }
    }
    return Ok(Some(order_by));
}
```

Select语句分析

对于distinct这样的可有可无还有默认选项的参数，我们统一写一个函数来解析，现在只有Select语句，但以后这种东西多的是。
这个函数会另外接受一个Keyword序列和一个默认的Keyword，当迭代器的下一个元素是Keyword类型且属于提供的Keyword序列之一，那就返回那个元素，否则返回默认参数。这样的话只要我们把关键字定义在Keyword中，它就能精准地匹配类型，而不用我们以后遇到类似的问题都要写一坨match。

```
// src/parser/statement_parser.rs

fn parse_optional_args_or(
    iter: &mut Peekable<IntoIter<Token>>,
    args: Vec<Keyword>,
    default: Keyword,
) -> Keyword {
    if let Some(Token::Keyword(keyword)) = iter.peek() {
        if let Some(nodetype) = args.iter().find(|&a| a.clone() == keyword.clone()) {
            iter.next();
            return nodetype.clone();
        }
    }
    default
}
```

好，最后我们把我们的成果组合在一起，Select语句的分析就 完成辣！

```
pub fn parse_select(iter: &mut Peekable<IntoIter<Token>>) -> Result<Statement> {
    match_token(&iter.next(), Token::Keyword(Keyword::Select))?;
    let distinct = match parse_optional_args_or(iter, vec![Keyword::All, Keyword::Distinct], Keyword::All) {
        Keyword::Distinct => true,
        _ => false,
    };

    let projections = parse_projection(iter)?;
```

```
    let table = parse_tables(iter)?;
    let filter = parse_where(iter)?;
    let group_by = parse_groupby(iter)?;
    let having = parse_having(iter)?;
    let order_by = parse_orderby(iter)?;

    if let Some(terminator) = iter.next() {
        if !terminator.is_terminator() {
            return Err(ParseError::UnexpectedToken(terminator));
        }
    } else {
        return Err(ParseError::MissingTerminator);
    }

    return Ok(Statement::Select {
        distinct,
        projections,
        table,
        filter,
        group_by,
        having,
        order_by
    });
}
```

稍微注意一下，文章内的代码很多省略了错误类型定义，但是有示范，所以你自己定义就行了，如果不会就去仓库看吧（恼）
你可能注意到了在各种子句解析我并没有统一定义一个类来概括所有解析过程，只有一些零零散散的数据结构有类，这可能会让某些万物皆可面向对象的读者不习惯，这里只是我个人的做法咯，我觉得这里没必要使用一个类来概括，而我们在parser文件下还有一个mod.rs来定义一个模块导出。每个类似的子目录都是这样的，例如datatype和models，这个记得要做。

```
src/parser/mod.rs

mod clause_parser;
mod expression_parser;
pub mod statement_parser;
pub mod error;
```

单元测试

我们使用Rust内置的测试框架进行一个简单的单元测试，测试代码通常位于与被测试代码相同的模块中，但是它们被放在一个特殊的子模块中，名为 tests。测试代码可以使用相同的访问控制规则来访问被测试代码，但是测试代码本身不会被包含在最终的二进制文件中。
在那之前，我用一个Parser类把操作统一了，实际上只是单纯的整合词法分析和语法分析，见仓库 src/

parse.rs。

```
// tests/select.rs

use masql::parse::Parser;

#[test]
fn test_insert() {
    let mut p = Parser::new();
    let statement = p.parse("
    SELECT DISTINCT SUM(score) AS score, age
        -- test
        FROM students, teachers
        WHERE @age = (2-1) * SUM(score)
        GROUP BY name, age
        HAVING age > 14
        ORDER BY name ASC, age DESC;
    ");
    if let Err(e) = statement {
        println!("{}", e);
    } else if let Ok(r) = statement {
        println!("{:?}", r);
    }
}
```

运行测试并查看输出的命令。

```
cargo test -- --nocapture
```

Next和汇总

因为这次更重要的是理论，所以很长很繁琐的代码我都放仓库里了。虽然这次文章内示例的只有Select语句，但是同样的思路已经可以搞定大部分语句了，如果你想继续做就应该续写其他语句支持。下一篇的题材已经选好了，这个系列应该是以后才更新，如果更新就不会是继续重复语句支持了，在语法分析阶段我们不考虑类型，总之万物皆可Expression，所以我们必须有一个类型检查机制，我们会涉及到另外一个知识点，就期待吧。

简单汇总一下目前为止Select及子句的支持：

- All/Distinct参数
- */列选择，支持表达式和AS别名
- Tables，支持表达式和AS别名
- Filter：支持条件表达式，AND、OR、NOT的嵌套
- Group By：支持表达式和AS别名

- Having：支持条件表达式，AND、OR、NOT的嵌套
- Order By：支持ASC、DESC排序方式，列名仅支持字面量

不考虑类型检查，表达式支持数据类型为所有的AST节点类型，包括Symbol、Value、Function，其中Value包括字面量、数字、变量、布尔值、Null，Function参数支持表达式。
强调注意在现阶段不支持这些：

- 语句嵌套，例如Select嵌套
- CASE子句
- JOIN子句

但是JOIN子句是SQL非常常用的语句，所以我会尽快完成支持，关注仓库更新（文末链接）

参考书籍和相关链接

Compilers: Principles, Techniques, and Tools：https://www.dbscience.org/wp-content/uploads/2020/03/ALSUdragonbookcompilers.pdf

Modern Compiler Implementation in C
http://www.infouem.com.br/wp-content/uploads/2011/03/Modern-Compiler-Implementation-in-C.pdf

Compiler Design Tutorial：https://www.geeksforgeeks.org/compiler-design-tutorials/

Github repository：Ma ROR SQLParser：https://github.com/MAKIROR/Ma-R-SQLP

MAKIROR gzanan@gmail.com 02:32 28/05/2023