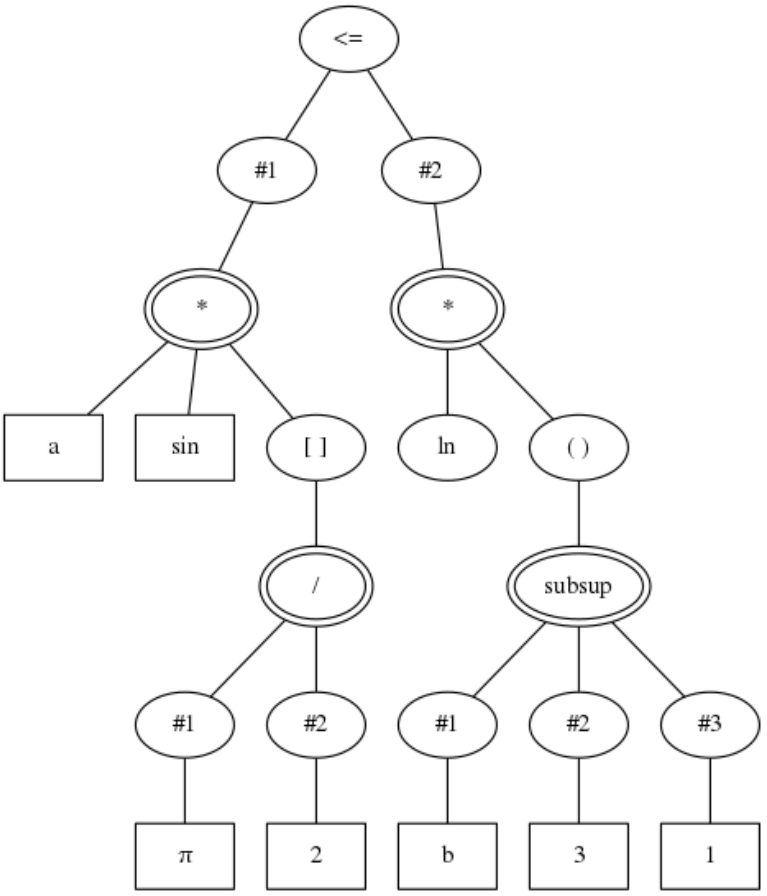
公式

$$a \cdot \sin\left[\frac{\pi}{2}\right] = \ln(b_1^3)$$

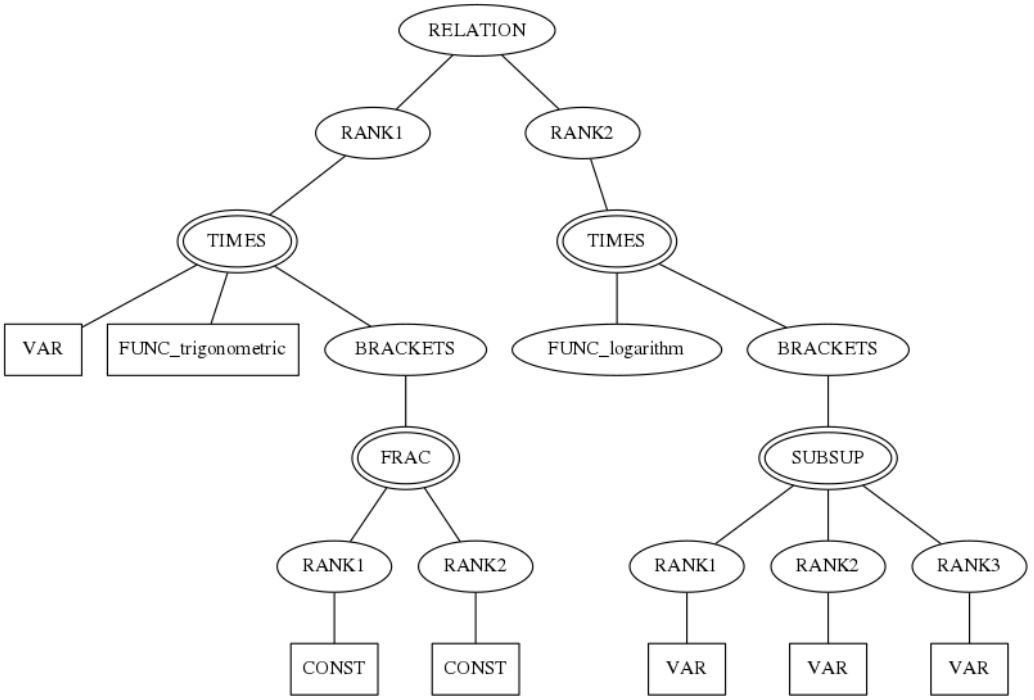
在 parse 以后的 operator tree:



其中：

1. commutative operator 需要在儿子节点上添加 pseudo node (e.g. #1, #2 ...)，以在 leaf-root path 中区分儿子的顺序
2. 图中双边的节点叫做“gener node”，之后会对他们进行一些单独 index，以此来提供对任意（或多个）gener node(s) 替换成 wildcard variable(s) 之后的 wildcard 搜索功能

上图 operator tree 是按照各个节点所对应的"symbol"来绘制的。其实每个"symbol"经过 tokenization 后可以得到所谓的"token"，如果用 token 来表示这棵树：



在我们的搜索引擎中，token 用来扩大搜索范围，而 symbol 用来在这个范围内给搜索结果打分。比如对于 query: $a \cdot \ln(b)$，表达式 $\lambda \cdot \ln(b)$ 和 $x \times \log(y)$ 都可以被搜索到（他们的 token 表示都相同），但是前者的排位在搜索结果中会更高(因为前者的 symbol 更多地符合 query)。

对于树中的每个节点：

```
struct optr_node {
    union {
        commutative;  /* boolean (for non-leaves) */
        wildcard;     /* boolean (for leaves, e.g. \ldots or user specified wildcard) */
    };
    symbol_id;
    token_id;
    sons;          /* number of sons (for leaves it is zero) */
    rank;          /* the rank among brothers (for root it is zero) */
    br_hash;       /* node to root symbols' HASH value */
};
```

不同于图示，在 operator tree 阶段不会加入实际的 pseudo node，而是以 rank 变量的形式记录儿子节点次序，pseudo node 要等到生成 leaf-root path 的时候才被实际“插入”。这样做的目的其一是降低 in-memory operator tree 所耗的内存，另一方面防止 parser debug 的时候 operator tree 节点太多不利于肉眼观察。

从这个 operator tree 上，我们提取 leaf-root paths (lr_path):

```
struct lr_paths {
    list lr_paths;
    n_lr_path;      /* total number of leaf-root paths from original tree */
};

struct lr_path {
    list lr_path_nodes;
    path_id;      /* less than the number of leaves from original tree */
    lf_wildcard;      /* if leaf is wildcard (meaningful only in query path) */
    lf_symbol_id;
};

struct lr_path_node {
    token_id;
    sons;     /* number of sons in original tree */
    fr_hash;  /* father to root symbols' HASH value, non-zero on leaf or gener node */
    ge_hash;  /* gener node HASH, non-zero on gener node */
    /* gener HASH: the sum of gener-leaf HASHes from all its descendants. */
};
```

其中，gener node 的充分条件是：

1. 是内节点（不是 root 也不是 leaf）
2. 儿子数目大于一 或者 括号、根号这类可能只有一个 operand 的 operator

根据 leaf-root path，我们按 path 顺序中的 token 以及 sons 来指定 posting list 所在的目录路径。比如图中最左边的那条

leaf-root path 将会在 index 的时候定位到 `${index_dir}/VAR/TIMES/3/RANK1/1/RELATION/2`，并把这条 path 的 posting_item 和 lr_path_info_item 结构 append 到（经过压缩）该位置的目录里的两个二进制文件 `${index_dir}/VAR/TIMES/3/RANK1/1/RELATION/2/{posting.bin, pathinfo.bin}`.

引入 sons 的目的是，搜索时可以通过比较 sons 来对一些 分支搜索进行 pruning。但是为了避免对可能 path 的过多穷举尝试（比如搜索 path 的某一节点 sons = 3, 那么该目录下的 3 到 maxSons 子目录都要被递归访问以获得所有可能结果），我们规定当 sons 超过一定阈值的时候，不再增加目录，而使用类似于 gtMaxSons 这样的目录名称代替更高数字的目录。

压缩前的 posting_item 和 lr_path_info_item 数据结构：

```
struct posting_item {
    doc_id:           32; /* delta and bit pack compression */
    frml_id:          32; /* bit pack compression */
    path_info_pos:    32; /* delta and bit pack compression */
};

struct path_info_item {
    union {
        lf_symbol_id;  /* in case of leaf-root path */
        ge_hash;       /* in case of gener-root path */
    };

    path_id;           /* maximum 64 leaves per tree */
    /* path_id is
     * 1: leaf path_id in case of leaf-root path;
     * 2: one of path_id from its descendant paths, in case of gener-root path.
     */

    fr_hash;           /* father-to-root HASH value */
    n_lr_path;         /* total number of leaf-root paths from original tree */
    eoi:               /* boolean: end of leaf-root path info item? */
};
```

我们需要三个 `${index_dir}` 来分别完成对 query 的 wildcard 搜索、普通数学搜索 还有基于 key-value database 的 term lookup 搜索（term lookup 搜索不仅仅是为了 general text 搜索，还有对数学公式中的数字进行精确加速的额外功能）：

- Wildcard index: 所有 gener-root path 单独 index 到这里
- Token index: 即 leaf-root path 的 index
- Dict index: 里面除了 index 数学公式里的 CONST token 开头的 leaf-root path，还作为 general text term 的 index directory。Dict index 通过 key-value 进行 term lookup。对于数学公式，对应的 value 是 leaf-root path index；对于 general text term，对应的 value 是传统搜索引擎的 term posting list.

数学公式里如果有带 CONST token 的 leaf-root path，则优先通过 dict index 搜索，如果搜索结果不够多则退化到 token index 搜索；带 wildcard token 的 leaf-root path 则通过 wildcard index 和 token index 里的 VAR/CONST 共同结合搜索（前者搜索匹配的 subexpression，后者搜索匹配的单个叶子节点）。

对于任意的 query path 和 index，搜索会进行以下几个阶段：

1. AND and math score 阶段: 分散开（distributed）搜索 index tree 的某一个广度搜索 level 的各个节点，AND merge 所有 query paths 定位的 postings，通过 Mark and Cross 算法得到的匹配度s和匹配的深度比d、广度比b对数学表达式进行 relevance score（拟定 math relevance = $\frac{s^2}{\ln(b \times d)}$），将 math relevance score 以 vector-space weight 的形式存储在 in-memory 的结构 mem_posting_ele 里面:

   ```
   struct mem_posting_ele {
       doc_id;
       vec_weight;
   };
   ```

   在 AND 之后、打分前可以过滤一些不可能的 hits：比如通过 compare n_lr_path（防止$a + a + a$能够搜索出$a + a$），或者检测（比如维护一个 HASH table) tuple $(\mathbf{docID}, \mathbf{frmlID})$之前是否已经存在于 ranking heap, so that we make sure IDs are unique in ranking set: $a + b$ will not get $a + b + \frac{a+b}{2}$ twice。
   期间不停检查 number of examed posting item 有没有达到最大阈值，超过的话直接进入下一阶段，并在结束 insert heap 之后结束搜索
2. Level OR 阶段: 通过 OR merge 汇总上一阶段在这个 level 各个节点得到的 math hits.
3. General AND/OR and score 阶段: 把上一步得到的 math posting 和其他 math postings 以及 general-text term postings 进行 AND/OR merge 操作，使用 vector space model 进行 query / doc 的整体评分，然后插入 heap（如果大于 heap min element 的话），并且保持 heap 不超过一个最大 size（有利于提高 ranking heap 的最小分，从而增加不用更深搜索的概率）。
4. 在进入下个 index tree level 前看看下一个 level 的 score 有没有可能超过 ranking heap 的最小分，能的话继续算法并回到阶段1，否则结束搜索。

在 Mark and Cross 的时候，ge_hash 会使得不同 wildcard symbols（比如 `\?x` 和 `\?y` ）所代表的 gener 子树不同。对于 gener, 我们用它子孙的其中一条 path_id 来代表它在 Mark and Cross 表中的位置，这就使得 Mark and Cross 中可能会出现 path_id 的重叠，这个时候我们我们以 gener node 的 path_id 为优先。

TODO

fix {\geq}
parser: star vs times 做一个区分

mem_trace
result highlight