

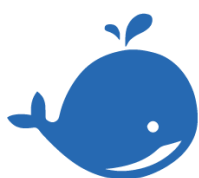
# 自制深度学习推理框架-表达式层的实现

赞助方：datawhale

作者：[傅莘莘](#)、[散步](#)、陈前

本章的代码：<https://github.com/zjhelloworldss/kuiperdatawhale.git>

补充材料：<https://www.bilibili.com/video/BV1HY4y1Z7S3/>



# Datawhale

# KuiperInfer

## 表达式的定义

**PNNX** 中的表达式就是一个二元的计算过程，类似如下：

```
1 output_mid = input1 + input2;  
2 output = output_mid * input3;
```

在 **PNNX** 的表达式层（Expression Layer）中，提供了一种计算表达式，该表达式能够在一定程度上折叠计算过程并消除中间变量。例如，在残差结构中的add操作在 **PNNX** 中就是一个表达式层。

下面是 **PNNX** 中对上述过程的计算表达式表示，其中的 **@0** 和 **@1** 代表之前提到的计算数 **RuntimeOperand**，用于表示计算表达式中的输入节点。

```
1 mul(@2, add(@0, @1));
```

尽管这个抽象表达式看起来比较简单，但实际上可能存在更为复杂的情况，例如以下的例子。因此，在这种情况下，我们需要一个强大而可靠的表达式解析和语法树构建功能。

```
1 add(add(mul(@0, @1), mul(@2, add(add(add(@0,
    @2), @3), @4))), @5);
```

## 词法解析

### 词法定义

词法解析的目的是将 **add(@0, mul(@1, @2))** 拆分为多个 Token，拆分后的 Token 依次为：

1. Identifier: **add**
2. Left bracket: **(**
3. Input number: **@0**
4. Comma: **,**
5. Identifier: **mul**
6. Left bracket: **(**
7. Input number: **@1**
8. Comma: **,**
9. Input number: **@2**
10. Right bracket: **)**

Token的类型定义如下:

```
1 enum class TokenType {  
2     TokenUnknown = -9,  
3     TokenInputNumber = -8,  
4     TokenComma = -7,  
5     TokenAdd = -6,  
6     TokenMul = -5,  
7     TokenLeftBracket = -4,  
8     TokenRightBracket = -3,  
9 };
```

Token的定义如下, 包括以下变量:

1. Token类型, 包括add (加法), mul (乘法), bracket (左右括号) 等;
2. Token在原句子中的开始和结束位置, 即 `start_pos` 和 `end_pos`;

对于表达式**add(@0, mul(@1, @2))**, 我们可以将它切分为多个Token, 其中Token(add)的 `start_pos` 为0, `end_pos` 为3。  
Token(left bracket)的 `start_pos` 为3, `end_pos` 为4。  
Token(@0)的 `start_pos` 为4, `end_pos` 为5, 以此类推。

```

1 // 词语Token
2 struct Token {
3     TokenType token_type =
TokenType::TokenUnknown;
4     int32_t start_pos = 0; // 词语开始的位置
5     int32_t end_pos = 0;   // 词语结束的位置
6     Token(TokenType token_type, int32_t
start_pos, int32_t end_pos)
7         : token_type(token_type),
start_pos(start_pos), end_pos(end_pos) {
8
9     }
10 };

```

最后，在词法解析结束后，我们需要将这些 Token（词语）按照它们的出现顺序和层级关系组成一棵语法树。

```

1 // 语法树的节点
2 struct TokenNode {
3     int32_t num_index = -1;
4     std::shared_ptr<TokenNode> left = nullptr;
5     // 语法树的左节点
6     std::shared_ptr<TokenNode> right = nullptr;
7     // 语法树的右节点
8     TokenNode(int32_t num_index,
std::shared_ptr<TokenNode> left,
std::shared_ptr<TokenNode> right);
9     TokenNode() = default;
10 };

```

# 词法解析

## 判断句子是否为空

```
1 CHECK(!statement_.empty()) << "The input
   statement is empty!";
```

## 移除句子中的空格

```
1 statement_.erase(std::remove_if(statement_.begin
   (), statement_.end(),
2                                     [](char c) {
   return std::isspace(c); }),
3                                     statement_.end());
4 CHECK(!statement_.empty()) << "The input
   statement is empty!";
```

如果表达式层中有表达式为 `add(@0, @1)`，我们删除其中的空格后就会得到新的表达式 `add(@0,@1)`。

## 逐个解析句子的字符

```
1 for (int32_t i = 0; i < statement_.size();) {
2     char c = statement_.at(i);
3     if (c == 'a') {
4         CHECK(i + 1 < statement_.size() &&
   statement_.at(i + 1) == 'd')
5         << "Parse add token failed, illegal
   character: "
6         << statement_.at(i + 1);
7         CHECK(i + 2 < statement_.size() &&
   statement_.at(i + 2) == 'd')
8         << "Parse add token failed, illegal
   character: "
```

```

9         << statement_.at(i + 2);
1         Token token(TokenType::TokenAdd, i, i +
0 3);
1         tokens_.push_back(token);
1         std::string token_operation =
2             std::string(statement_.begin() + i,
3 statement_.begin() + i + 3);
1         token_strs_.push_back(token_operation);
4         i = i + 3;
5     }

```

假设字符 `c` 表示当前的字符。如果 `c` 等于字符 'a'，根据我们的词法规定，Token 中以 'a' 开头的情况只有 add。因此，我们需要判断接下来的两个字符是否分别是 'd' 和 'd'。如果不是，则报错。如果是的话，则初始化一个新的 Token，并保存其在表达式中的初始和结束位置。

举个例子，如果表达式中的单词以 'a' 开头，那么它只能是 add，而不能是其他词汇表之外的单词，例如 `axc` 等情况。

```

1 CHECK(i + 1 < statement_.size() &&
  statement_.at(i + 1) == 'd')
2     << "Parse add token failed, illegal
  character: "
3     << statement_.at(i + 1);
4 CHECK(i + 2 < statement_.size() &&
  statement_.at(i + 2) == 'd')
5     << "Parse add token failed, illegal
  character: "
6     << statement_.at(i + 2);
7 Token token(TokenType::TokenAdd, i, i + 3);
8 tokens_.push_back(token);
9 std::string token_operation =
1     std::string(statement_.begin() + i,
0 statement_.begin() + i + 3);
1 token_strs_.push_back(token_operation);

```

如果在第一行中，我们判断第二个字符是否为 'd'；若是，在第二行中，我们判断第三个字符是否也是 'd'。如果满足条件，我们将初始化一个 Token 实例，并保存该单词在句子中的起始位置和结束位置。

同样地，如果某个字符 `c` 是 'm'，我们需要判断接下来的字符是否是 'u' 和 'l'。如果不满足条件，则说明我们的表达式中出现了词汇表之外的单词（因为词汇表只允许以 'm' 开头的单词是 "mul"）。如果满足条件，我们同样会初始化一个 Token 实例，并保存该单词的起始和结束位置，以及 Token 的类型。

```

1 else if (c == '@') {
2     CHECK(i + 1 < statement_.size() &&
  std::isdigit(statement_.at(i + 1)))
3     << "Parse number token failed, illegal
  character: " << c;
4     int32_t j = i + 1;

```

```

5     for (; j < statement_.size(); ++j) {
6         if (!std::isdigit(statement_.at(j))) {
7             break;
8         }
9     }
1    Token token(TokenType::TokenInputNumber, i,
0    j);
1    CHECK(token.start_pos < token.end_pos);
1    tokens_.push_back(token);
2    std::string token_input_number =
3    std::string(statement_.begin() + i,
    statement_.begin() + j);
1    token_strs_.push_back(token_input_number);
4    i = j;

```

如果第一个字符是 '@'，我们需要读取 '@' 后面的所有数字，例如对于 @31231，我们需要读取 @ 符号之后的所有数字。如果紧跟在 '@' 后面的字符不是数字，则报错。如果是数字，则将这些数字全部读取并组成一个单词（Token）。

```

1    else if (c == ',') {
2        Token token(TokenType::TokenComma, i, i +
    1);
3        tokens_.push_back(token);
4        std::string token_comma =
5        std::string(statement_.begin() + i,
    statement_.begin() + i + 1);
6        token_strs_.push_back(token_comma);
7        i += 1;
8    }

```

如果第一个字符是 ',' 逗号，那么我们直接读取这个字符作为一个新的Token。



最后，在正确解析和创建这些 Token 后，我们将它们放入名为 `tokens` 的数组中，以便进行后续处理。

```
1 tokens_.push_back(token);
```

## 语法解析

### 语法树的定义

```
1 struct TokenNode {
2     int32_t num_index = -1;
3     std::shared_ptr<TokenNode> left = nullptr;
4     std::shared_ptr<TokenNode> right = nullptr;
5     TokenNode(int32_t num_index,
6         std::shared_ptr<TokenNode> left,
7         std::shared_ptr<TokenNode> right);
8     TokenNode() = default;
9 };
10
```

在进行语法分析时，我们可以根据词法分析得到的 `token` 数组构建抽象语法树。抽象语法树是一个由二叉树组成的结构，每个节点都存储了操作符号或值，并通过左子节点和右子节点与其他节点连接。

对于表达式 "add (@0, @1)"，当 `num_index` 等于 1 时，表示计算数为 @0；当 `num_index` 等于 2 时，表示计算数为 @1。若 `num_index` 为负数，则说明当前节点是一个计算节点，如 "mul" 或 "add" 等。

以下是一个简单的示例：

```
1      add
2     /  \
3   @0   @1
```

在这个示例中，根节点是 "add"，左子节点是 "@0"，右子节点是 "@1"。这个抽象语法树表示了一个将 "@0" 和 "@1" 进行相加的表达式。

通过将词法分析得到的 `token` 数组解析并构建抽象语法树，我们可以进一步对表达式进行语义分析和求值等操作。

## 递归向下的解析

语法解析的过程是递归向下的,定义在 `Generate_` 函数中。

```
1 std::shared_ptr<TokenNode>
  ExpressionParser::Generate_(int32_t &index) {
2     CHECK(index < this->tokens_.size());
3     const auto current_token = this-
  >tokens_.at(index);
4     CHECK(current_token.token_type ==
  TokenType::TokenInputNumber
5         || current_token.token_type ==
  TokenType::TokenAdd || current_token.token_type
  == TokenType::TokenMul);
```

这个函数处理的对象是词法解析的Token（单词）数组，因为 `Generate_` 是一个递归函数，所以 `index` 参数指向Token数组中的当前处理位置。

`current_token` 表示当前被处理的Token，它作为当前递归层的第一个Token，必须是以下类型之一。

```
1 TokenInputNumber = 0,
2 TokenAdd = 2,
3 TokenMul = 3,
```

如果当前Token的类型是输入数字类型，那么会直接返回一个操作数Token作为叶子节点，不再进行下一层递归（如下）。例如，在表达式add(@0, @1)中的@0和@1被归类为输入数字类型的Token，在解析到这两个Token时会直接创建并返回语法树节点TokenNode。

```
1  if (current_token.token_type ==
    TokenType::TokenInputNumber) {
2      uint32_t start_pos =
    current_token.start_pos + 1;
3      uint32_t end_pos = current_token.end_pos;
4      CHECK(end_pos > start_pos);
5      CHECK(end_pos <= this-
    >statement_.length());
6      const std::string &str_number =
    std::string(this->statement_.begin() +
7      start_pos, this->statement_.begin() + end_pos);
8      return std::make_shared<TokenNode>
    (std::stoi(str_number), nullptr, nullptr);
9
1 }

```

如果当前Token的类型是mul或者add，我们需要进行下一层递归来构建对应的左子节点和右子节点。

例如，在处理add(@1,@2)时，遇到add token之后，如下的第一行代码，我们需要做以下的两步：

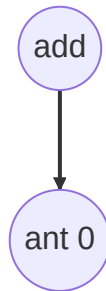
1. 首先判断是否存在左括号（left bracket）
2. 然后继续向下递归以获取@1，如下的第14行到17行代码，但由于@1代表的是数字类型，递归后立即返回，如以上代码块中第一行对数字类型Token的处理。

```

1  else if (current_token.token_type ==
   TokenType::TokenMul || current_token.token_type
   == TokenType::TokenAdd) {
2      std::shared_ptr<TokenNode> current_node =
   std::make_shared<TokenNode>();
3      current_node->num_index = -
   int(current_token.token_type);
4
5      index += 1;
6      CHECK(index < this->tokens_.size());
7      // 判断add之后是否有( left bracket
8      CHECK(this->tokens_.at(index).token_type ==
   TokenType::TokenLeftBracket);
9
1     index += 1;
10    CHECK(index < this->tokens_.size());
11    const auto left_token = this-
12 >tokens_.at(index);
13    // 判断当前需要处理的left token是不是合法类型
14    if (left_token.token_type ==
   TokenType::TokenInputNumber
15     || left_token.token_type ==
   TokenType::TokenAdd || left_token.token_type ==
   TokenType::TokenMul) {
16        // (之后进行向下递归得到@@
17        current_node->left = Generate_(index);
18    } else {
19        LOG(FATAL) << "Unknown token type: " <<
   int(left_token.token_type);
20    }
21 }

```

在第17行当左子树递归构建完毕后，将它赋值到add节点的左子树上。对于表达式 `add(@0, @1)`，我们将左子树连接到 `current_node` 的 `left` 指针中，随后我们开始构建右子树。

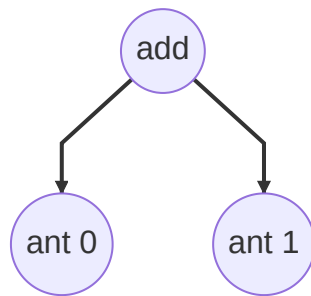


```
1      index += 1;
2      // 当前的index指向add(@1,@2)中的逗号
3      CHECK(index < this->tokens_.size());
4      // 判断是否是逗号
5      CHECK(this->tokens_.at(index).token_type ==
TokenType::TokenComma);
6
7      index += 1;
8      CHECK(index < this->tokens_.size());
9      // current_node->right = Generate_(index);构
建右子树
1     const auto right_token = this-
0 >tokens_.at(index);
1     if (right_token.token_type ==
1 TokenType::TokenInputNumber
1         || right_token.token_type ==
2 TokenType::TokenAdd || right_token.token_type
== TokenType::TokenMul) {
1         current_node->right = Generate_(index);
3     } else {
4         LOG(FATAL) << "Unknown token type: " <<
5 int(left_token.token_type);
1     }
6
```

```
1    index += 1;
8    CHECK(index < this->tokens_.size());
9    CHECK(this->tokens_.at(index).token_type ==
0 TokenType::TokenRightBracket);
2    return current_node;
```

随后我们需要判断@0之后是否存在comma token，如上代码中的第五行。在构建右子树的过程中，对于表达式`add(@1,@2)`，当`index`指向逗号的位置时，首先需要判断是否存在逗号。接下来，我们开始构建右子树，在右子树的向下递归分析中，会得到@2作为一个叶子节点。

当右子树构建完成后，将该节点（即`Generate_`返回的`TokenNode`，此处为一个叶子节点，其数据为@1）放置于`current_node`的`right`指针中。



## 一个例子

我们以一个简单点的例子为开始，假设现在表达式层中的表达式是：`add(@0,@1)`。在词法解析模块中，这个表达式将被构建成一个单词(Token)数组，如以下：

- add
- (
- @0

- ,
- @1
- )

在词法解析结束之后，这个表达式将被传递到语法解析模块中，用于构建抽象语法树。 `Generate_` 函数首先检查Token数组中的当前单词(Token)是否是以下类型的一种：

```
1 CHECK(index < this->tokens_.size());
2 const auto current_token = this-
  >tokens_.at(index);
3 CHECK(current_token.token_type ==
  TokenType::TokenInputNumber ||
4       current_token.token_type ==
  TokenType::TokenAdd ||
5       current_token.token_type ==
  TokenType::TokenMul);
```

当前的索引为0，表示正在处理Token数组中的"add"单词。针对这个输入，我们需要判断其后是否是"左括号"来确定其合法性。如果是合法的（**add**单词之后总存在括号），我们将构建一个左子树。因为对于一个add调用，它的后面总是跟着一个左括号"("，如下方代码的第8行。

```

1  else if (current_token.token_type ==
    TokenType::TokenMul ||
2      current_token.token_type ==
    TokenType::TokenAdd) {
3      std::shared_ptr<TokenNode> current_node =
    std::make_shared<TokenNode>();
4      current_node->num_index =
    int(current_token.token_type);
5
6      index += 1;
7      CHECK(index < this->tokens_.size()) <<
    "Missing left bracket!";
8      CHECK(this->tokens_.at(index).token_type ==
    TokenType::TokenLeftBracket);
9
1     index += 1;
10    CHECK(index < this->tokens_.size()) <<
1    "Missing correspond left token!";
1    const auto left_token = this-
2    >tokens_.at(index);

```

在以上代码的第8行中，我们对'add'之后的一个Token进行判断，如果是左括号则匹配成功，开始匹配括号内的元素。对于输入 `add(@0, @1)`，在第10行中，当对索引进行+1操作后，我们需要开始解析括号内左侧的元素 `left_token`。

随后我们开始递归构建表达式的左子树：



```

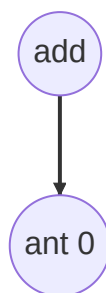
1  if (left_token.token_type ==
    TokenType::TokenInputNumber ||
2      left_token.token_type == TokenType::TokenAdd
    ||
3      left_token.token_type ==
    TokenType::TokenMul) {
4      current_node->left = Generate_(index);
5  }

```

对于当前的例子，当前索引(index)指向的单词是@0。在这种情况下，由于索引指向的位置是一个输入数字

@0(TokenType::TokenInputNumber)的类型，所以该节点进入递归调用后将直接返回。

根据前文给出的例子，'add'的左子树构建完毕后，下一步我们需要判断中add(@0,@1)的@0之后是否存在逗号，下方代码的第二行



```

1  index += 1;
2  CHECK(index < this->tokens_.size()) << "Missing
    comma!";
3  CHECK(this->tokens_.at(index).token_type ==
    TokenType::TokenComma);

```

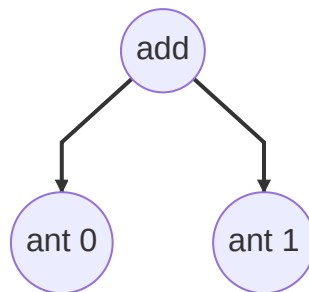
接下来，我们要为如上的二叉树构建右子树：

```

1  const auto right_token = this-
    >tokens_.at(index);
2  if (right_token.token_type ==
    TokenType::TokenInputNumber ||
3      right_token.token_type ==
    TokenType::TokenAdd ||
4      right_token.token_type ==
    TokenType::TokenMul) {
5      current_node->right = Generate_(index);
6  } else {
7      LOG(FATAL) << "Unknown token type: " <<
        int(right_token.token_type);
8  }

```

同样，由于当前索引(index)指向的位置是@1，它是一个输入数据类型，所以该节点在进入递归调用后将直接返回，并成为add节点的右子树，如下方代码所示。



```

1  std::shared_ptr<TokenNode>
    ExpressionParser::Generate_(int32_t &index) {
2      CHECK(index < this->tokens_.size());
3      ...
4      ...
5      如果是Input Number就直接返回
6
7      if (current_token.token_type ==
        TokenType::TokenInputNumber) {

```

```

8         uint32_t start_pos =
current_token.start_pos + 1;
9         uint32_t end_pos =
current_token.end_pos;
1        CHECK(end_pos > start_pos);
0        CHECK(end_pos <= this-
1 >statement_.length());
1        const std::string &str_number =
2        std::string(this-
3 >statement_.begin() + start_pos, this-
>statement_.begin() + end_pos);
1        return std::make_shared<TokenNode>
4 (std::stoi(str_number), nullptr, nullptr);
1
5    }
6 }

```

## 一个更复杂些的例子

如果现在有一个表达式 `add(mul(@0,@1),@2)`，那么我们应该如何对其进行语法解析呢？在词法解析中，它将被分割成以下的数个单词组成的数组：

1. add
2. left bracket
3. mul
4. left bracket
5. @0
6. comma
7. @1
8. right bracket

9. comma

10. @2

11. right bracket

当以上的数组被输入到语法解析中后，index的值等于0. 随后我们再判断index指向位置的单词类型是否符合要求。

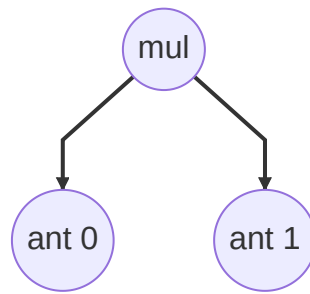
```
1 CHECK(current_token.token_type ==  
  TokenType::TokenInputNumber ||  
2     current_token.token_type ==  
  TokenType::TokenAdd ||  
3     current_token.token_type ==  
  TokenType::TokenMul);
```

如果该表达式的第一个单词是"add"，那么我们就像之前的例子一样，将它作为二叉树的左子树进行构建。

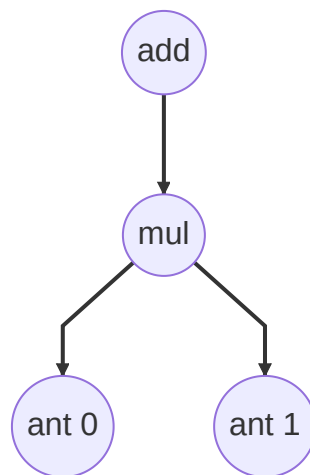
```
1 if (left_token.token_type ==  
  TokenType::TokenInputNumber ||  
2     left_token.token_type == TokenType::TokenAdd  
  ||  
3     left_token.token_type ==  
  TokenType::TokenMul) {  
4     current_node->left = Generate_(index);
```

已知表达式为 `add(mul(@0,@1),@2)`，在处理完这个表达式的左括号之后，当前指向的标记是"mul"，它不属于输入参数类型。因此，在调用 `Generate_` 函数时，我们将对"mul"子表达式进行递归分析。

对"mul"子表达式解析的方式和对 `add(@0,@1)` 解析的方式相同，"mul"子表达式的分析结果如下图所示：



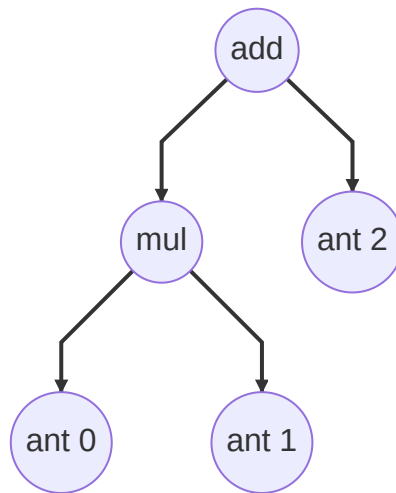
在子表达式的解析完成并返回后，我们将这颗子树插入到当前节点的左指针上(**`current_node->left = Generate_(index)`**)



随后我们开始解析 `add(mul(@0,@1),@2)` 表达式中@2以及其之后的部分作为add的右子树。

```
1  if (right_token.token_type ==  
    TokenType::TokenInputNumber ||  
2    right_token.token_type ==  
    TokenType::TokenAdd ||  
3    right_token.token_type ==  
    TokenType::TokenMul) {  
4    current_node->right = Generate_(index);  
5  } else {  
6    LOG(FATAL) << "Unknown token type: " <<  
    int(right_token.token_type);  
7  }
```

在第4行调用 `Generate_` 之后，由于@2是一个输入数类型，不再进行递归分析，所以它将被直接返回并赋值给 `current_node->right`。



```
1 std::shared_ptr<TokenNode>
  ExpressionParser::Generate_(int32_t &index) {
2     CHECK(index < this->tokens_.size());
3     ...
4     ...
5     如果是Input Number就直接返回
6
7     if (current_token.token_type ==
TokenType::TokenInputNumber) {
8         uint32_t start_pos =
current_token.start_pos + 1;
9         uint32_t end_pos =
current_token.end_pos;
10        CHECK(end_pos > start_pos);
11        CHECK(end_pos <= this-
>statement_.length());
12        const std::string &str_number =
std::string(this-
>statement_.begin() + start_pos, this-
>statement_.begin() + end_pos);
```

```
1         return std::make_shared<TokenNode>
4 (std::stoi(str_number), nullptr, nullptr);
1
5     }
6 }
```

最终，我们成功完成了这个较为复杂的二叉树构建例子。

## 单元测试1

### 词法解析

```
1 TEST(test_parser, tokenizer) {
2     using namespace kuiper_infer;
3     const std::string &str = "";
4     ExpressionParser parser(str);
5     parser.Tokenizer();
6     const auto &tokens = parser.tokens();
7     ASSERT_EQ(tokens.empty(), false);
8
9     const auto &token_strs = parser.token_strs();
10    ASSERT_EQ(token_strs.at(0), "add");
11    ASSERT_EQ(tokens.at(0).token_type,
12    TokenType::TokenAdd);
13
14    ASSERT_EQ(token_strs.at(1), "(");
15    ASSERT_EQ(tokens.at(1).token_type,
16    TokenType::TokenLeftBracket);
17
18    ASSERT_EQ(token_strs.at(2), "@0");
19    ASSERT_EQ(tokens.at(2).token_type,
20    TokenType::TokenInputNumber);
21
22    ASSERT_EQ(token_strs.at(3), ",");
```

```
20    ASSERT_EQ(tokens.at(3).token_type,  
TokenType::TokenComma);  
21  
22    ASSERT_EQ(token_strs.at(4), "mul");  
23    ASSERT_EQ(tokens.at(4).token_type,  
TokenType::TokenMul);  
24  
25    ASSERT_EQ(token_strs.at(5), "(");  
26    ASSERT_EQ(tokens.at(5).token_type,  
TokenType::TokenLeftBracket);  
27  
28    ASSERT_EQ(token_strs.at(6), "@1");  
29    ASSERT_EQ(tokens.at(6).token_type,  
TokenType::TokenInputNumber);  
30  
31    ASSERT_EQ(token_strs.at(7), ",");  
32    ASSERT_EQ(tokens.at(7).token_type,  
TokenType::TokenComma);  
33  
34    ASSERT_EQ(token_strs.at(8), "@2");  
35    ASSERT_EQ(tokens.at(8).token_type,  
TokenType::TokenInputNumber);  
36  
37    ASSERT_EQ(token_strs.at(9), ")");  
38    ASSERT_EQ(tokens.at(9).token_type,  
TokenType::TokenRightBracket);  
39  
40    ASSERT_EQ(token_strs.at(10), ")");  
41    ASSERT_EQ(tokens.at(10).token_type,  
TokenType::TokenRightBracket);  
42 }
```



我们对表达式 `add(@0,mul(@1,@2))` 进行了词法切分，并得到了一个Tokens数组。通过逐一比对该数组，我们确认了词法分析器在这个案例下的正常工作。如果你在以上的表达式中加入了错误的符号或者单词，例如 `add(@0,mcl(@1,@2))`，以上的单元测试一定会报错。

## 语法解析

```
1 TEST(test_parser, generate1) {
2     using namespace kuiper_infer;
3     const std::string &str = "add(@0,@1)";
4     ExpressionParser parser(str);
5     parser.Tokenizer();
6     int index = 0; // 从0位置开始构建语法树
7     // 抽象语法树:
8     //
9     //      add
10    //      /  \
11    //  @0    @1
12
13    const auto &node = parser.Generate_(index);
14    ASSERT_EQ(node->num_index,
15    int(TokenType::TokenAdd));
16    ASSERT_EQ(node->left->num_index, 0);
17    ASSERT_EQ(node->right->num_index, 1);
18 }
```

```
1 TEST(test_parser, generate2) {
2     using namespace kuiper_infer;
3     const std::string &str =
4     "add(mul(@0,@1),@2)";
5     ExpressionParser parser(str);
```

```

5     parser.Tokenizer();
6     int index = 0; // 从0位置开始构建语法树
7     // 抽象语法树:
8     //
9     //         add
10    //        /  \
11    //      mul   @2
12    //     /  \
13    //   @0   @1
14
15    const auto &node = parser.Generate_(index);
16    ASSERT_EQ(node->num_index,
17    int(TokenType::TokenAdd));
18    ASSERT_EQ(node->left->num_index,
19    int(TokenType::TokenMul));
20
21    ASSERT_EQ(node->left->left->num_index, 0);
22    ASSERT_EQ(node->left->right->num_index, 1);
23
24    ASSERT_EQ(node->right->num_index, 2);
25 }

```

在以上的代码中，我们的 `Generate_` 函数构造了一棵层次结构正确的语法树，同学们也可以自行尝试构建更复杂的语法树。

## 对语法树的转换

### 逆波兰式

我们来以一个简单的例子来说明，对于计算式 `add(@0, @1)`，首先遇到的节点是 `add`，但在遇到 `add` 时缺少进行计算所需的具体数据 `@0` 和 `@1`。

因此，我们需要进行逆波兰转换，将操作数放在前面，计算放在后面。该转换的实现非常简单，只需对原有的二叉树进行后续遍历即可：

```
1 void ReversePolish(const
  std::shared_ptr<TokenNode> &root_node,
2
  std::vector<std::shared_ptr<TokenNode>>
  &reverse_polish) {
3     if (root_node != nullptr) {
4         ReversePolish(root_node->left,
reverse_polish);
5         ReversePolish(root_node->right,
reverse_polish);
6         reverse_polish.push_back(root_node);
7     }
8 }
```

逆波兰式化后的表达如下：

对于 `add (@0,@1)`，逆波兰式为： `@0,@1,add`

对于 `add(mul(@0,@1),@2)`，逆波兰式为： `@0,@1,mul,@2,add`

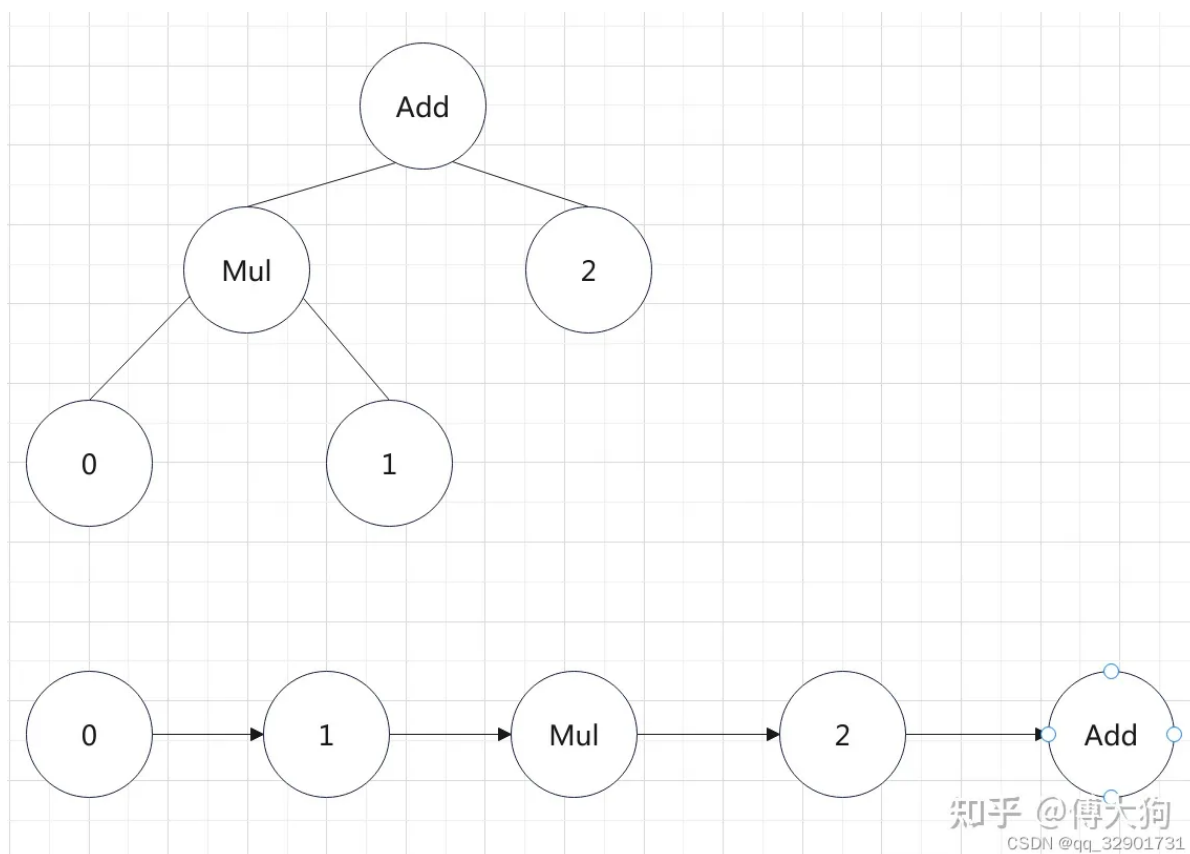
通过逆波兰转换，可以将原式转换为计算式的输入数放在前面，操作符号放在后面的形式。逆波兰式的特点是消除了括号的需求，使得计算顺序更加清晰和直观。

## 整体过程回顾

经过这样的转换，可以确保在每次遇到计算节点时所需的操作数已经准备就绪。

1. 首先，传入一个表达式字符串，例如`add(mul(@0,@1),@2)`

2. 接下来，对`add(mul(@0,@1),@2)`进行词法分析，将其拆分为多个tokens，在拆分过程中需要进行词法校验。
3. 然后，根据已知的tokens数组，通过递归向下遍历进行语法分析，从而得到相应的计算二叉树。计算二叉树的各个节点可以是`add`、`mul`或者`@0`、`@1`等。
4. 最后，对计算二叉树进行逆波兰变换，得到的逆波兰式如下：`@0、@1、mul、@2、add`。



## 单元测试2

在这个单元测试中，我们需要将抽象语法树通过后续遍历转换为可执行的队列。在可执行队列中，各个元素满足操作数在前、操作符号在后的特点。对于如下的一个表达式：

`add(mul(@0,@1),@2)`，在构建完成后，我们得到了以下形式的二叉树。该二叉树对应的逆波兰式表达式为：

`@0,@1,mul,@2,add.`

```

1
2         add
3       /   \
4     mul   @2
5   /     \
6  @0     @1

```

```

1 TEST(test_parser, reverse_polish) {
2     using namespace kuiper_infer;
3     const std::string &str =
4     "add(mul(@0,@1),@2)";
5     ExpressionParser parser(str);
6     parser.Tokenizer();
7     // 抽象语法树:
8     //
9     //         add
10    //       /   \
11    //     mul   @2
12    //   /     \
13    //  @0     @1
14
15    const auto &vec = parser.Generate();
16    for (const auto &item : vec) {
17        if (item->num_index == -5) {
18            LOG(INFO) << "Mul";
19        } else if (item->num_index == -6) {
20            LOG(INFO) << "Add";
21        } else {
22            LOG(INFO) << item->num_index;
23        }
24    }
25 }

```

输出结果:

```
1 I20230805 12:51:25.513495 4751
  test_expression.cpp:116] 0
2 I20230805 12:51:25.513556 4751
  test_expression.cpp:116] 1
3 I20230805 12:51:25.513599 4751
  test_expression.cpp:112] Mul
4 I20230805 12:51:25.513639 4751
  test_expression.cpp:116] 2
5 I20230805 12:51:25.513679 4751
  test_expression.cpp:114] Add
```

## 表达式层的实现

```
1 class ExpressionLayer : public NonParamLayer {
2     public:
3         explicit ExpressionLayer( std::string
  statement);
4
5         InferStatus Forward(
6             const
  std::vector<std::shared_ptr<Tensor<float>>>&
  inputs,
7
  std::vector<std::shared_ptr<Tensor<float>>>&
  outputs) override;
8
9         static ParseParameterAttrStatus GetInstance(
1            const std::shared_ptr<RuntimeOperator>&
0 op,
1            std::shared_ptr<Layer>&
1 expression_layer);
1
2     private:
```

```

3   std::string statement_;
4   std::unique_ptr<ExpressionParser> parser_;
5 };

```

我们可以看到在表达式层面有两个类内变量。第一个是需要解析的表达式 `statement_`，第二个是之前提到的用于词法分析和语法分析的 `ExpressionParser`，需要解析的表达式 `statement_` 会被传入到解析器 `parser_` 中。

## ExpressionParser的定义

我们再来看一下 `ExpressionParser` 的定义：

```

1  class ExpressionParser {
2  public:
3      explicit ExpressionParser(std::string
statement)
4          : statement_(std::move(statement)) {}
5      /**
6       * 词法分析
7       * @param retokenize 是否需要重新进行语法分析
8       */
9      void Tokenizer(bool retokenize = false);
10
11      ...
12      ...
13
14  private:
15      std::shared_ptr<TokenNode> Generate_(int32_t&
index);
16      // 被分割的词语数组
17      std::vector<Token> tokens_;
18      // 被分割的字符串数组
19      std::vector<std::string> token_strs_;

```

```

9 // 待分割的表达式
0 std::string statement_;
2 };

```

在 `ExpressionParser` 中，就像我们上文中所说的一样，`Tokenizer` 用于将表达式分割为多个单词，例如将 `add(@0,@1)` 分割为以下的几个单词：

1. add
2. left bracket
3. input number(@0)
4. comma
- ...

## ExpressionParser 生成语法树

```

1 std::vector<std::shared_ptr<TokenNode>>
  ExpressionParser::Generate() {
2     if (this->tokens_.empty()) {
3         this->Tokenizer(true);
4     }
5     int index = 0;
6     // 构建语法树
7     std::shared_ptr<TokenNode> root =
  Generate_(index);
8     CHECK(root != nullptr);
9     CHECK(index == tokens_.size() - 1);
10
11     // 转逆波兰式, 之后转移到expression中
12     std::vector<std::shared_ptr<TokenNode>>
  reverse_polish;
13     ReversePolish(root, reverse_polish);

```



```
14
15     return reverse_polish;
16 }
```

在以上的代码中，我们首先调用 `Generate_` 方法。该方法通过对 `tokens` 数组进行语法分析，生成一棵抽象语法树。然后，我们对这棵抽象语法树进行逆波兰排序，得到最终的执行序列。

这些内容我们已经在前文中详细地讲述过了，这里不再详细展开。

## 表达式层的注册

我们已经在前面的几章内容中介绍了层或算子的注册过程。在定义完表达式层的计算过程后，我们需要将其注册到推理框架中。这样，框架就能够使用我们定义的表达式层进行计算。

```
1 LayerRegistererWrapper
  kExpressionGetInstance("pnnx.Expression",
2
  ExpressionLayer::GetInstance);
```

我们再来看一下它的初始化过程：

```
1 ParseParameterAttrStatus
  ExpressionLayer::GetInstance(
2     const std::shared_ptr<RuntimeOperator>& op,
3     std::shared_ptr<Layer>& expression_layer) {
4     CHECK(op != nullptr) << "Expression operator
  is nullptr";
5     const auto& params = op->params;
6     if (params.find("expr") == params.end()) {
```

```

7         return
ParseParameterAttrStatus::kParameterMissingExpr
;
8     }
9
10    auto statement_param =
11
12        std::dynamic_pointer_cast<RuntimeParameterString>(params.at("expr"));
13    if (statement_param == nullptr) {
14        LOG(ERROR) << "Can not find the expression
parameter";
15        return
ParseParameterAttrStatus::kParameterMissingExpr
;
16    }
17    if (statement_param->type !=
RuntimeParameterType::kParameterString) {
18        LOG(ERROR) << "Can not find the expression
parameter";
19        return
ParseParameterAttrStatus::kParameterMissingExpr
;
20    }
21    expression_layer =
std::make_shared<ExpressionLayer>
(statement_param->value);
22    return
ParseParameterAttrStatus::kParameterAttrParseSu
ccess;
23 }

```

先来看一个实例：

**NODE PROPERTIES**✕

type

pnnx.Expression

name

pnnx\_expr\_12

**ATTRIBUTES**

#1

(4,1,4,4)f32

#2

(4,1,4,4)f32

#3

(4,1,4,4)f32

expr

add(@0,@1)

**INPUTS**

input

name: 1

1

name: 2

**OUTPUTS**

output

name: 3

在以上的代码中：

```
1 auto statement_param =  
2  
std::dynamic_pointer_cast<RuntimeParameterString>(params.at("expr"));
```

从 `PNNX` 中提取表达式字符串 `expr`，然后使用该字符串来实例化算子。

```
1 expression_layer =  
std::make_shared<ExpressionLayer>  
(statement_param->value);
```

# 表达式层的输入处理

在Expression Layer的Forward函数输入中，也就是在这个数组中，多个输入依次排布：

```
1 | const
   std::vector<std::shared_ptr<Tensor<float>>>&
   inputs
```

如果 `batch_size` 的大小为4，那么 `input1` 中的元素数量为4，`input2` 的元素数量也为4。换句话说，`input1` 中的数据都来源于同一批次的操作数1（operand 1），`input2` 中的数据都来源于同一批次的操作数2（operand 2）。

其中，`input1` 中的4(batch size = 4)个元素都是来自于操作数1，而 `input2` 中的4(batch size = 4)个元素都是来自于操作数2，它们在 `inputs` 数组参数中依次排列，如下图所示。

| inputs vec |            |
|------------|------------|
| input 1    | input 2    |
| batch size | batch size |

## 对于两个输入操作数

已知有如上的数据存储排布，在本小节中我们将讨论如何根据现有的数据完成 `add(@0, @1)` 计算。可以看到每一次计算的时候，都以此从 `input1` 和 `input2` 中取得一个数据进行加法操作，并存放在对应的输出位置。

| input1                |        |                         |        | input2                  |        |                         |        |
|-----------------------|--------|-------------------------|--------|-------------------------|--------|-------------------------|--------|
| batch size            |        |                         |        | batch size              |        |                         |        |
| input1                | input2 | input3                  | input4 | input5                  | input6 | input7                  | input8 |
| output1=input1+input5 |        | output2 = input2+input6 |        | output3 = input3+input7 |        | output4 = input4+input8 |        |

## 对于三个输入操作数

下图的例子展示了对于三个输入, `mul(add(@0, @1), @2)` 的情况:

| input1                              |        |        |        | input2                                |        |        |        | input3                             |         |         |         |
|-------------------------------------|--------|--------|--------|---------------------------------------|--------|--------|--------|------------------------------------|---------|---------|---------|
| batch size                          |        |        |        | batch size                            |        |        |        | batch size                         |         |         |         |
| input1                              | input2 | input3 | input4 | input5                                | input6 | input7 | input8 | input9                             | input10 | input11 | input12 |
| output1 = (input1+ input5) * input9 |        |        |        | output2 =( input2 + input6) * input10 |        |        |        | output4 = (input3+input7)* input11 |         |         |         |

每次计算的时候依次从 `input1`, `input2` 和 `input3` 中取出数据, 并作出相应的运算, 并将结果数据存放于对应的 `output` 中。我们简单说明一下:

1.  $output_1 = (input_1 + input_5) \times input_9$ , 对于第一个输出数据, 我们先从取出第一组输入 (@0) 中第一个输入数据  $input_1$ , 再从第二组输入 (@1) 中取得第一个输入数据  $input_5$ , 最后再从第三组输入 (@2) 中取得第一个输入数据  $input_9$ .
2.  $output_2 = (input_2 + input_6) \times input_{10}$ , 对于第一个输出数据, 我们先从取出第一组输入 (@0) 中第一个输入数据  $input_2$ , 再从第二组输入 (@1) 中取得第一个输入数据  $input_6$ , 最后再从第三组输入 (@2) 中取得第一个输入数据  $input_{10}$ .
3.  $output_3 = (input_3 + input_7) \times input_{11}$ , 对于第一个输出数据, 我们先从取出第一组输入 (@0) 中第一个输入数据  $input_3$ , 再从第二组输入 (@1) 中取得第一个输入数据  $input_7$ , 最后再从第三组输入 (@2) 中取得第一个输入数据  $input_{11}$ .
4.  $output_4$  同理。

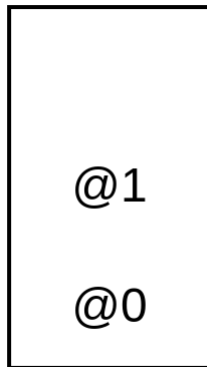
# 表达式层的计算过程

表达式层同样继承于算子的父类 `Layer`，并重写了其中的 `Forward` 方法。在 `Forward` 方法中，我们定义了表达式层的计算逻辑。假设现在有一个计算式为 `add(mul(@0, @1), @2)`，通过抽象语法树构建和逆波兰转换，我们已经得到了以下序列：

@0、@1、mul、@2、add

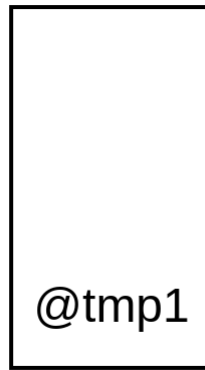
在 `Forward` 函数中，我们定义了一个栈式计算模块，并维护了一个输入数栈。输入数栈是一个先进后出的数据结构，用于存放表达式中的输入数。

对于给定的表达式，例如 `add(mul(@0, @1), @2)`，我们将前两个输入数依次压入输入数栈中。

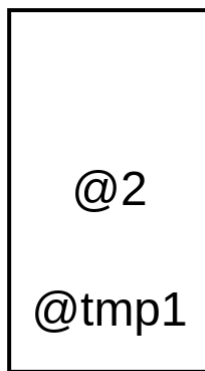


在序列中的下一个节点是 `mul`，它的作用是将两个输入数相乘。因此，我们需要从输入数栈中顺序地弹出两个输入数进行相乘操作。通过该操作，我们得到一个中间结果 `@tmp1`。接下来，我们需要将这个中间结果存放到输入数栈中，以便供后续步骤处理。

@0      Mul      @1



在序列中的下一个节点是@2，因为是一个输入操作数，我们将它存放到输入数栈中，图示如下：



在序列中的最后一个节点是 `add`，它是一个加法节点，需要两个输入数据。因此，它会将栈中的 `@2` 和 `@tmp1` 全部弹出，进行加法操作，得到整个计算序列最后的结果。

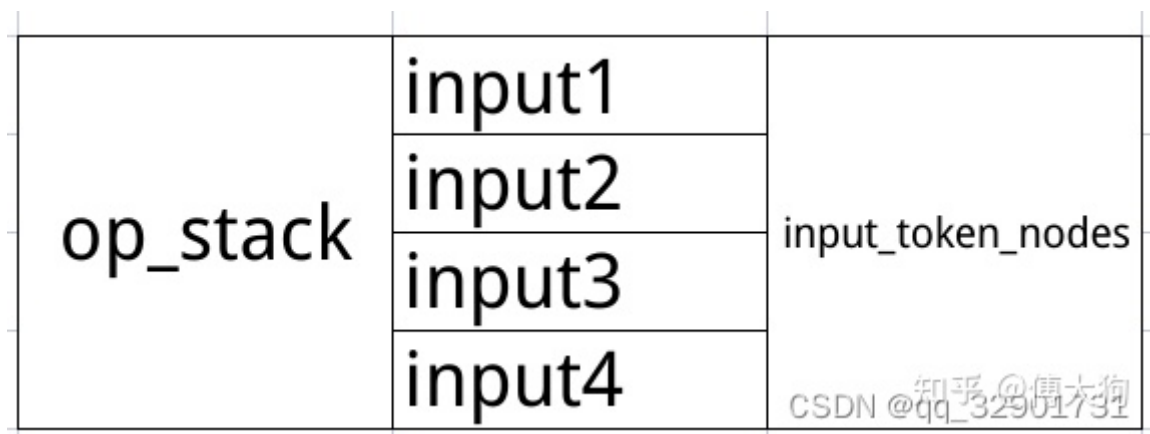
**我们再细讲一下：**

```

1  if (token_node->num_index >= 0) {
2      // process operator
3      uint32_t start_pos = token_node->num_index
      * batch_size;
4      std::vector<std::shared_ptr<Tensor<float>>>
input_token_nodes;
5      for (uint32_t i = 0; i < batch_size; ++i) {
6          CHECK(i + start_pos < inputs.size())
7              << "The " << i
8              << "th operand doesn't have
appropriate number of tensors";
9          input_token_nodes.push_back(inputs.at(i
+ start_pos));
10     }
11     op_stack.push(input_token_nodes);
12 }

```

根据输入的逆波兰式@0,@1,add，遇到的第一个节点是操作数是@0，所以栈op\_stack内的内存布局如下：



当按顺序遇到第二个节点（op）时，也就是操作数@1时，我们将从inputs中读取操作数并将其存放到input\_token\_nodes中。然后，将input\_token\_nodes这一批次的数据放入栈中。



|          |        |                   |
|----------|--------|-------------------|
| op_stack | input5 | input_token_nodes |
|          | input6 |                   |
|          | input7 |                   |
|          | input8 |                   |
|          | input1 | input_token_nodes |
|          | input2 |                   |
|          | input3 |                   |
|          | input4 |                   |

CSDN @知乎@傅大狗  
qq\_32901791

## 运算符的代码处理

```

1  const int32_t op = token_node->num_index;
2  ...
3  std::vector<std::shared_ptr<Tensor<float>>>
   input_node1 = op_stack.top();
4  ...
5  ...
6  op_stack.pop();
7
8  std::vector<std::shared_ptr<Tensor<float>>>
   input_node2 = op_stack.top();
9  CHECK(input_node2.size() == batch_size)
10     << "The second operand doesn't have
   appropriate number of tensors, "
11     "which need "
12     << batch_size;
13  op_stack.pop();

```

当节点（`op`）类型为操作符号时（也就是`num_index`小于0的时候），首先我们从栈（`op_stack`）中弹出两个批次的操作数。对于给定情况，`input_node1`存放的是`input1`至`input4`，而`input_node2`存放的是`input5`至`input8`。

```
1  std::vector<std::shared_ptr<Tensor<float>>>
   output_token_nodes(
2      batch_size);
3
4  for (uint32_t i = 0; i < batch_size; ++i) {
5      // do execution
6      if (op == int(TokenType::TokenAdd)) {
7          output_token_nodes.at(i) =
8              TensorElementAdd(input_node1.at(i),
9                               input_node2.at(i));
10     } else if (op == int(TokenType::TokenMul))
11     {
12         output_token_nodes.at(i) =
13             TensorElementMultiply(input_node1.at(i),
14                                   input_node2.at(i));
15     } else {
16         LOG(FATAL) << "Unknown operator type: "
17         << op;
18     }
19 }
20 op_stack.push(output_token_nodes);
```

在获取大小为`batch_size`的`input_node1`和`input_node2`后，流程将在`for(int i = 0...batch_size)`循环中对这两个输入进行两两操作（`input1+input5`，`input2+input6`），具体的操作类型定义于当前的`op`中。最后，我们将计算得到的结果放入输入数栈`op_stack`中。

# 单元测试3

```
1 TEST(test_expression, complex1) {
2     using namespace kuiper_infer;
3     const std::string& str =
4         "mul(@2,add(@0,@1))";
5     ExpressionLayer layer(str);
6     std::shared_ptr<Tensor<float>> input1 =
7         std::make_shared<Tensor<float>>(3, 224,
8         224);
9     input1->Fill(2.f);
10    std::shared_ptr<Tensor<float>> input2 =
11        std::make_shared<Tensor<float>>(3, 224,
12        224);
13    input2->Fill(3.f);
14
15    std::shared_ptr<Tensor<float>> input3 =
16        std::make_shared<Tensor<float>>(3, 224,
17        224);
18    input3->Fill(4.f);
19
20    std::vector<std::shared_ptr<Tensor<float>>>
21    inputs;
22    inputs.push_back(input1);
23    inputs.push_back(input2);
24    inputs.push_back(input3);
25
26    std::vector<std::shared_ptr<Tensor<float>>>
27    outputs(1);
28    outputs.at(0) =
29        std::make_shared<Tensor<float>>(3, 224, 224);
30    const auto status = layer.Forward(inputs,
31    outputs);
```

```

24 | ASSERT_EQ(status,
    | InferStatus::kInferSuccess);
25 | ASSERT_EQ(outputs.size(), 1);
26 | std::shared_ptr<Tensor<float>> output2 =
27 |     std::make_shared<Tensor<float>>(3, 224,
    | 224);
28 | output2->Fill(20.f);
29 | std::shared_ptr<Tensor<float>> output1 =
    | outputs.front();
30 |
31 | ASSERT_TRUE(
32 |     arma::approx_equal(output1->data(),
    | output2->data(), "absdiff", 1e-5));
33 | }

```

以上的表达式为 `mul(@2, add(@0, @1))`。我们将表示 `@0` 的 `input1` 赋值为2，将表示 `@1` 的 `input2` 赋值为3，并将表示 `@3` 的 `input3` 赋值为4，计算结果应该是  $(@0 + @1) * @2 = (2 + 3) * 4 = 20$ 。

## 本节课的作业

本次作业见代码 `test/test_expr_homework.cpp` 中的单元测试

- 词法和语法解析中支持 `sin`(三角函数) 操作
  - 词法分析: **TEST**(test\_parser, tokenizer\_sin)
  - 语法分析: **TEST**(test\_parser, generate\_sin)
- 如果操作符是单输入数，例如问题1中的 `sin` 函数，我们的 `Forward` 函数应该做出什么改动能获得正确的计算结果。
  - 完成 **TEST**(test\_expression, complex1)

