编译大作业第二部分报告

1. 设计思路

沿用第一部分的IR结构，在此基础上进行求导计算。

在设计上主要有以下几个难点：

- 如何初始化所求微分数组
- 如何计算某一特定张量的微分
- 如何确定函数的所有参数
- 如何处理变量名相同但下标不同的张量
- 如何变换下标使得等号左侧下标不存在运算

接下来将着重着眼于这几个难点进行分析

2. 实现方法

初始化所求微分数组

程序开始时，需要将所求微分数组清空。

具体实现时，先找到所求微分数组的范围，之后用临时变量遍历整个数组，赋值为0即可。

以第六个样例的代码为例：

```
2   for (int i = 0; i < 2; ++i) {
3     for (int j = 0; j < 16; ++j) {
4       for (int k = 0; k < 7; ++k) {
5         for (int l = 0; l < 7; ++l) {
6           dB[i][j][k][l] = 0;
7         }
8       }
9     }
10  }
```

计算某一特定张量的微分

首先我们可以将输入的表达式抽象为一棵二叉树，我们要求导的张量位于二叉树的某个叶结点。

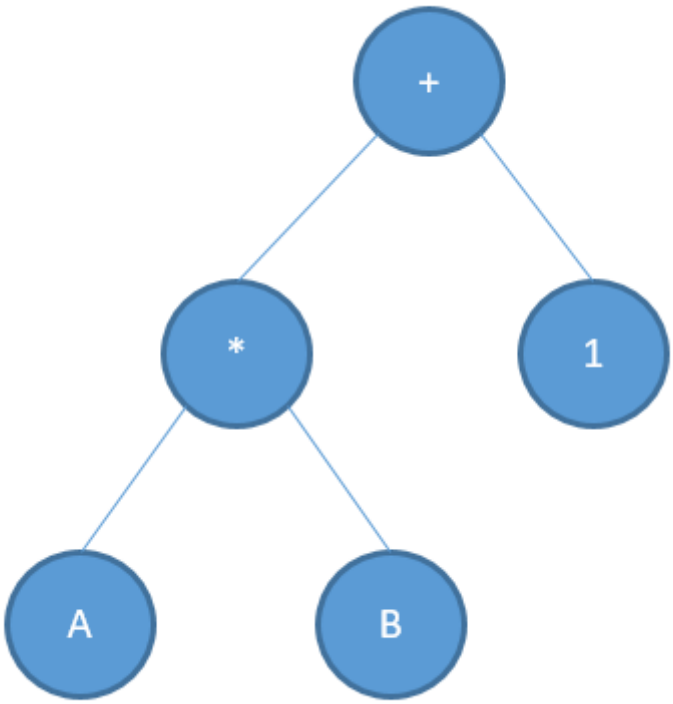
从该点向上走，初始时表达式为1，如果走到的点代表加减法运算，就忽略该点继续向上；如果走到的点是乘法，则乘上另一侧的表达式。

最终走到根，就求出了该点对应张量的导数，再将导数乘上等式左侧张量的微分，即算出所求张量的微分。

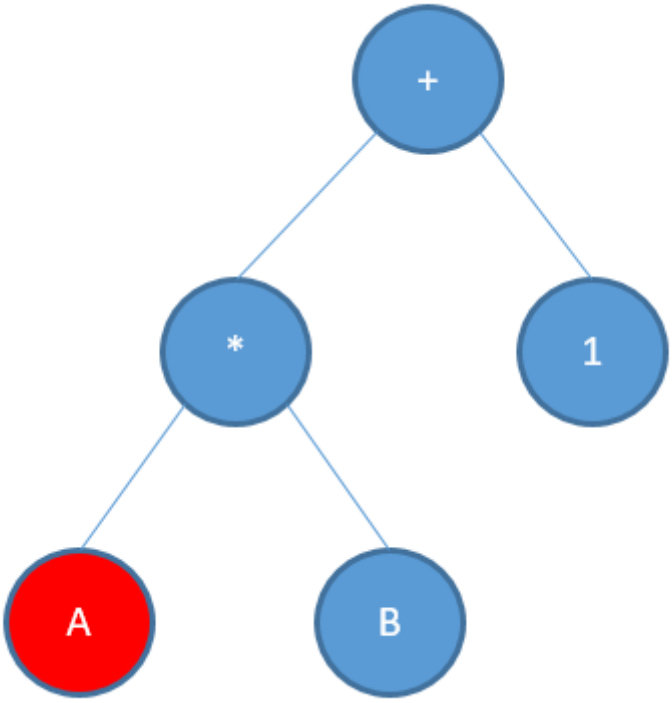
以第一个样例为例进行说明：

```
{
    "name": "grad_case1",
    "ins": ["A", "B"],
    "outs": ["C"],
    "data_type": "float",
    "kernel": "C<4, 16>[i, j] = A<4, 16>[i, j] * B<4, 16>[i, j] + 1.0;",
    "grad_to": ["A"]
}
```

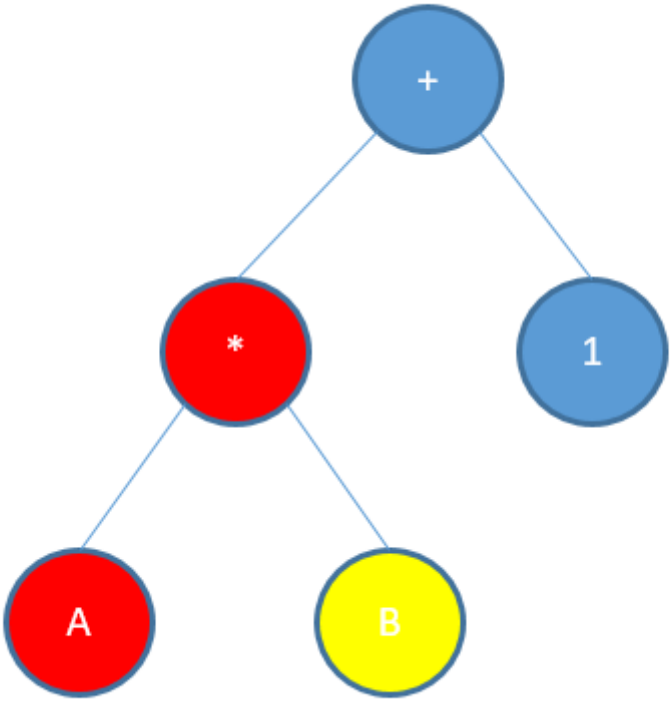
将表达式右侧建树，我们得到：



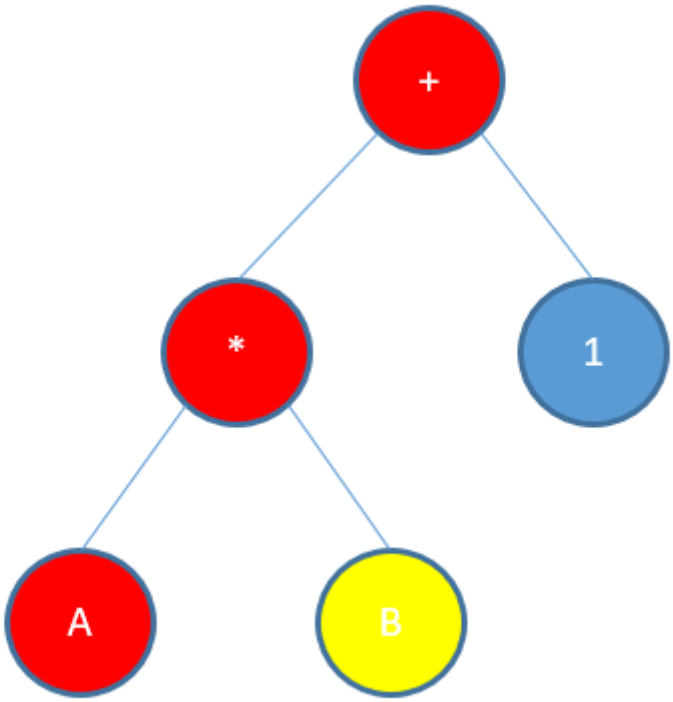
我们要对A求导，初始时表达式为1：



向上走一步，遇到乘法，将表达式乘上右侧对应的表达式，变为$1 \times B[i, j]$：



再向上走一步，遇到加法，直接忽略即可：



最终导数表达式为$1 \times B[i,j]$，乘上$dC[i,j]$，最终表达式为$dA[i,j] = dC[i,j] \times 1 \times B[i,j]$

代码的具体实现中采用的是dfs找到确定张量对应的点后向上回溯的方法。

确定函数的所有参数

和第一部分不同，第二部分的参数并不能直接通过ins和outs来确定。

比如样例一中，A是ins中的张量

```
{
    "name": "grad_case1",
    "ins": ["A", "B"],
    "outs": ["C"],
    "data_type": "float",
    "kernel": "C<4, 16>[i, j] = A<4, 16>[i, j] * B<4, 16>[i, j] + 1.0;",
    "grad_to": ["A"]
}
```

但却没有出现在函数的参数中

```
void grad_case1(float (&B)[4][16], float (&dC)[4][16], float (&dA)[4][16]) {
```

一定会出现的参数为outs和grad_to中张量的微分。

那么如何确定ins中的张量是否在函数的参数中呢?

首先我们要将求导之后的式子求出来，如果一个张量在这个式子中，那么就是函数的参数，否则就不是函数的参数。

样例一中，张量A在求导后的式子中消失了，所以它不是函数的参数。

而我们观察样例2，

```
{
  "name": "grad_case2",
  "ins": ["A"],
  "outs": ["B"],
  "data_type": "float",
  "kernel": "B<4, 16>[i, j] = A<4, 16>[i, j] * A<4, 16>[i, j] + 1.0;",
  "grad_to": ["A"]
}
```

发现张量A在求导后依然存在，所以A是样例2函数的参数。

```
void grad_case2(float (&A)[4][16], float (&dB)[4][16], float (&dA)[4][16]) {
```

最终，所有ins中的张量放在参数最前面，之后是outs中张量的微分，最后才是grad_to中张量的微分。

一个张量有可能和它的微分同时为函数的参数。

处理变量名相同但下标不同的张量

在样例10中，出现了变量名相同但下标不同的张量：

```
{
    "name": "grad_case10",
    "ins": ["B"],
    "outs": ["A"],
    "data_type": "float",
    "kernel": "A<8, 8>[i, j] = (B<10, 8>[i, j] + B<10, 8>[i + 1, j] + B<10, 8>[i + 2, j]) / 3.0;",
    "grad_to": ["B"]
}
```

可以发现三个张量B有不同的下标，显然我们没法同时对他们求导。

那么怎么办呢?

我们可以分别对每一个B单独求导，然后再将三项加起来。

输出的代码为：

```
for (int i = 0; i < 8; ++i) {
  for (int j = 0; j < 8; ++j) {
    dB[i][j] = dB[i][j] + dA[i][j] * (1) / 3;
    dB[i + 1][j] = dB[i + 1][j] + dA[i][j] * (1) / 3;
    dB[i + 2][j] = dB[i + 2][j] + dA[i][j] * (1) / 3;
  }
}
```

用这种方法即可完美解决问题，但新的问题出现了：题目要求等号左边的下标不能有运算，这部分我们要如何处理呢?

变换下标使得等号左侧下标不存在运算

在6、8、10三组样例中，我们发现求微分的张量参数的下标中有运算。

根据题目要求，我们需要在求导后去除这些运算。

以样例10为例进行说明

```
for (int i = 0; i < 8; ++i) {
  for (int j = 0; j < 8; ++j) {
    dB[i][j] = dB[i][j] + dA[i][j] * (1) / 3;
    dB[i + 1][j] = dB[i + 1][j] + dA[i][j] * (1) / 3;
    dB[i + 2][j] = dB[i + 2][j] + dA[i][j] * (1) / 3;
  }
}
```

发现第二第三项dB中，有i+1和i+2作为下标，这不满足要求。

所以我们可以用新的循环变量来代替这两个下标。

代码实现时，我们将等式左侧张量的所有下标都替换成了新的循环变量，具体如下：

```
 7   for (int i = 0; i < 8; ++i) {
 8     for (int j = 0; j < 8; ++j) {
 9       for (int b = i; b < i + 1; ++b) {
10         for (int c = j; c < j + 1; ++c) {
11           dB[b][c] = dB[b][c] + dA[i][j] * (1) / 3;
12         }
13       }
14       for (int b = i + 1; b < i + 1 + 1; ++b) {
15         for (int c = j; c < j + 1; ++c) {
16           dB[b][c] = dB[b][c] + dA[i][j] * (1) / 3;
17         }
18       }
19       for (int b = i + 2; b < i + 2 + 1; ++b) {
20         for (int c = j; c < j + 1; ++c) {
21           dB[b][c] = dB[b][c] + dA[i][j] * (1) / 3;
22         }
23       }
24     }
25   }
```

发现等式左侧不再有运算。

3. 组内分工

四人共同讨论大作业思路并进行分工。

黄致焕：实现计算导数的代码

冯哲：实现初始化、确定函数参数代码

杨芳源：实现确定下标范围、输入输出代码

姜成烨：实现变换下标的代码