课程实验: 设计上下文无关文法的变换算法

0. 小组成员

- A 算法设计及撰写报告
- B 参与程序代码的实现
- C 参与程序代码的实现
- D 参与程序代码的实现
- GitHub 开源项目地址

1. 实验目的

- 编程实现上下文无关文法的变换算法，用于消除文法中的 ε 产生式、单产生式、以及无用符号。

2. 实验内容

- 上下文无关文法对产生式的右部没有限制，这种完全自由的形式有时会对文法分析带来不良影响，而对文法的某些限制形式在应用中更方便。通过上下文无关文法的变换，在不改变文法的语言生成能力的前提下，可以消除文法中的 ε 产生式、单产生式、以及无用符号。
- 编程实现消除上下文无关文法中的 ε 产生式、单产生式、以及无用符号的算法。输入是一个上下文无关文法，输出是与该文法等价的没有 ε 产生式、单产生式、无用符号的上下文无关文法。

3. 实验环境描述

- 操作系统 Windows 11 专业版 64 位, Kernel: 10.0.22000.0
- C++ 编译器 GCC version 11.2.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders)

4. 设计思路

- 使用有向图来存储所有符号之间的推导关系，使用图中的点来代表字符，有向边来表示推导关系。
- 用代码实现实现文法中的推导关系，以及如何处理特殊样例比如环形的推导关系等等。
- 按照消去无用符号、消去空字符、消去单产生式的步骤逐步处理和化简文法产生式。
- 每个消去的步骤中，都主要依赖图论中的深度优先搜索算法来实现产生式的替换和化简。

5.核心算法

我们按照顺序分别讲述如何消去无用符号、消去空产生式、消去单产生式的算法细节和代码实现，

以及为什么要使用图的搜索算法，其他算法的思路为什么不对，会有哪些特殊样例导致算法失效等等问题。

5.1 将文法产生式转换为图信息存储

在实现具体的化简过程之前，我们先考虑如何来用数据结构存储文法的逻辑关系，且以一个较为复杂，能包含较多推导关系，测试算法鲁棒性比较好的的文法例子开始:

$$
\begin{array}{ll}
S \to XYX|A|B & A \to bBX|aCbC \\
B \to bbb|BX|D & D \to C \\
X \to aX|\epsilon & Y \to bY|\epsilon \\
C \to \epsilon & E \to eE|F \\
F \to fF|G & G \to gG
\end{array}
$$

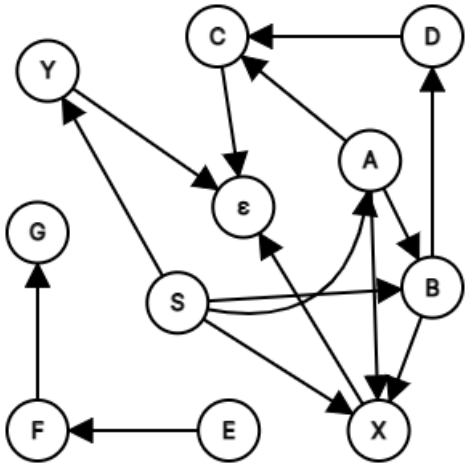
在上述例子中，从起始符 S 开始，其可以推导至起始符 X, Y, A, B，同时 A 也能推导至起始符 B, X, C 等等，在消去无用符号中，我们需要考虑哪些起始符是根本不会被其他起始符推导到的“落单”符号，以及在消去空产生式中，我们需要找到所有能通过链式推导关系能到达 ϵ 的起始符，比如 $B \to D \to C \to \epsilon$ 这样的关系。

这启发我们使用图这种数据结构来存储整个字符集合，然后用图中的有向边来表示推导关系，比如第一行产生式 $S \to XYX|A|B$ 为例，我们需要在图中从表示字符 S 的点向表示字符 X, Y, A, B 的点分别连一条有向边来表示推导关系，代码初始声明如下:

```
const int N = 50;                                   //起始字符集最大数量
int h[N], e[N], idx, ne[N];                         //存储图所需要的邻接表数组
bool reachable[N];                                  //初步推导可达关系
map<string, vector<string> > all_production;//存储所有产生式的哈希表
map<string, int> start_symbol;                      //将字符映射为代表图中点数字的哈希表
map<int, string> start_symbol_rev;                  //数字到字符的双向映射哈希表
int number = 1;                                     //当前的产生式数量，初始随意赋值为 1
bool epsilon_reachable[N];                          //存储字符能否到达 epsilon 空状态
```

在很多图论算法的实现中，一般都是用数字来表示点的编号，这样利于代码的实现，而文法中使用的都是单个字符或者字符串，所以我们需要两个哈希表 `map<string, int> start_symbol`，以及 `map<int, string> start_symbol_rev` 来双向存储字符和数字之前的双射关系，便于快速转换。

哈希表 `map<string, vector<string> > all_production` 存储所有的产生式，这个哈希表的 Key 值为每一行文法左边的起始字符，Value 值存储所有能从左边起始字符直接到达的产生式，这样单纯的文字描述有点抽象，我们画个图来形象化上面这个文法吧:



使用 csacademy.com/app/graph_editor 绘制

对，就是这样，好像有点复杂。看起来我们只在图中存储了所有起始字符之间的推导关系，没有考虑终结符号。确实如此，因为终结符号并不需要化简什么，需要我们处理和化简的逻辑只存在于起始符号之间。对于终结符，我们只需要存一下每个起始字符能到的终结符串就行了，最后消去化简之后简单处理连接一下字符串即可。接下来我们就可以从图论的角度继续讲述如何化简文法了，先从消去无用符号开始。

5.2 消去无用符号

其实消去无用符号一共有两种产生式需要考虑，第一种就是不会被其他产生式到达的那些符号，比如上图中的 E, F, G 就是，虽然 E 可以到达 F, G, 但是从 S 出发却到达不了它们三个点，用术语来讲它们三个在图中自己构成了一个孤立的连通块，图中一共有 2 个连通块。

还有一种呢就是特殊的 “单递归” 产生式，比如下面这个文法:

$$
\begin{aligned}
&S \rightarrow A|B|C \\
&A \rightarrow Bd|Cf|D|c \\
&C \rightarrow A|E \\
&B \rightarrow bB|D|E \\
&D \rightarrow dD|E \\
&E \rightarrow eE
\end{aligned}
$$

我们重点观察一下最后三行产生式，其中起始符号 B, D, E 都有类似于 $X \rightarrow aX$ 这样的右递归式子，这些产生式永远都不会到达终结符号，所以也要把这些产生式消去，他们也是无用的。但是消去这种式子好像并不简单，观察一下产生式 $B \rightarrow bB|D|E$ ，如果我们想知道起始字符 B 能否通过一系列的推导到达终止字符，或者到达 ϵ ，我们就需要知道 B 可到达的字符 D, E 是否能到达终止字符或者 ϵ 。

同样的逻辑，想知道 D 能否到达终结符，我们继续还得知道 D 能到达的 E 能否到达终结符号.....等等一直这样递归下去, 直到假设我们可以确认 D, E 一直推导下去，他们俩有一个确实可以到达终结符号，那么此时 B 才能说是可以到达终结符号的，B 才是有用的，不能被删去。反之如果 D, E 它们俩都不能到达，那么就可以知道 B 也不可到达终结符，就得把 B, D, E 全部删掉了。

仔细思考一下这个过程，这是完全递归的，对于每个起始字符，我们都需要递归地去推导一次它是否可以到达终止字符或者空字符。其实这个过程本质就是图论中的深度优先搜索 (Depth-first search) 算法，我们从起点 S 出发，开始搜索 S 的邻接点 (图中的直接邻居) 能否可达，然后 S 的邻居又去搜索它们的邻接点，直到最终搜索到临界的终点，这些终点就是类似 $A \rightarrow aa|bb$ 这样只含有终结字符或者 ϵ 的产生式，然后就可以回溯了，返回给上一层递归搜索，这样我们从 S 开始做一次 DFS 就可以找出这些要消去的 “单递归” 产生式了，主要代码如下:

```
bool dfs(int u) {  //判断点 u开始能否到达终结符
    if (reachable[u]) return true;
    // u右边只有终结符组成的字符串返回 true
    if (all_production.find(start_symbol_rev[u]) == all_production.end())
        return false;
    auto vec = all_production[start_symbol_rev[u]];
    for (auto t : vec) {
        bool flag = true;
        for (auto c : t) {
            if (c >= 'A' && c <= 'Z') {
                flag = false;
                break;
            }
        }
        if (flag) {
            reachable[u] = true;
            break;
        }
    }
    // u能到的起始符也是可达的返回 true
    // A -> B | C
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
```

```
        reachable[u] |= dfs(j);
    }
    if (reachable[u]) return true;
    return false;  //有环的情况 A->C, C->A => A->A 也是不可达的
}
```

因为代码确实很长 (整个程序近 500 行)，大片附着在报告里面会丧失可读性，所以我只截取核心的代码来讲述，注意观察一下上面整个函数的第 $22-25$ 行代码:

```
for (int i = h[u]; ~i; i = ne[i]) {
    int j = e[i];
    reachable[u] |= dfs(j);
}
```

u 是当前搜索的点，然后我们去遍历点 u 的邻居，继续递归搜索这些邻接点，如果点 u 有任意一个邻居可以到达终结符，那么 u 肯定在图中存在一条有向路径能够到达终结符, 这很容易理解。所以这里使用 "或" 的逻辑来实现。

最后一行代码 `return false` 处理了默认递归的返回值，如果能执行到这行代码，说明图中出现了环形推导关系，比如 $A \to B, B \to A$, 这样本质就是 $A \to A$ ，这样的产生式暂且保留不会被消去。

数组 `reachable[u]` 标记了点 u 在搜索结束后能否到达终结符。对于图中孤立的连通块，从起点 S 出发是肯定搜索不到他们的，所以这个 DFS 的过程可以同时消去上述两种无用符号，这样无用符号就被消除完了。回顾一下这个过程，我们只在图上做了一次 DFS, 所以这个过程的算法时间复杂度是 $O(n)$ 的，n 是字符集大小 。

5.3 消去 ϵ 产生式

对于消去空产生式，我们同样用一开始举例的比较长的文法来说:

$$
\begin{array}{ll}
S \to XYX|A|B & A \to bBX|aCbC \\
B \to bbb|BX|D & D \to C \\
X \to aX|\epsilon & Y \to bY|\epsilon \\
C \to \epsilon & E \to eE|F \\
F \to fF|G & G \to gG
\end{array}
$$

能直接推导到达 ϵ 的起始字符有 X, Y, C, 然后进而得知能够到达 X, Y, C 的那些字符也能到达 ϵ ，然后一直逆推下去，我们就可以直到到底有哪些符号可以递推到 ϵ 了，不过这是逆向思考，我们并不需要这样。因为已经消去了无用符号，所以目前还留在图中的点都在同一个连通块中，我们从起点 S 出发正向搜索一次的效果是一样的，核心代码如下所示:

```
bool dfs_epsilon(int u) {
    if (epsilon_reachable[u]) return true;
    auto vec = all_production[start_symbol_rev[u]];
    for (auto t : vec) {
        if (t == "epsilon") {
            epsilon_reachable[u] = true;
        }
        bool flag = true;
        for (auto c : t) {
            if (c >= 'a' && c <= 'z') {
                flag = false;
                break;
```

```
            }
        }
        if (flag) {
            set<string> S;
            for (auto c : t) {
                if (string(1, c) != start_symbol_rev[u])
                    S.insert(string(1, c));
            }
            bool res = true;
            for (auto str : S) {
                // A -> XY, X, Y 均可达epsilon, A才能->epsilon
                res &= dfs_epsilon(start_symbol[str]);
            }
            if (res) {
                epsilon_reachable[u] = res;
            }
        }
    }
    return epsilon_reachable[u];
}
```

这个代码的逻辑和消除无用产生式的并没有太大区别，就是在图中递归搜索就行了，递归回溯的临界点就是搜索到那些直接可以推导到 ϵ 的字符。唯一的区别就是要考虑一种特殊的情况，比如文法中的:

$$S \to XYX|A|B$$

观察右边第一项 XYX, 搜索结束之后要确保 X, Y 都可以到达 ϵ，那么 XYX 整体才可以到达 ϵ，所以在上述代码中的 $22-25$ 行:

```
for (auto str : S) {
    // A -> XY, X, Y 均可达epsilon, A才能->epsilon
    res &= dfs_epsilon(start_symbol[str]);
}
```

我们需要用 “与” 逻辑来实现，而不是消除无用符号的 “或” 逻辑，这是唯一的区别。这样从起点 S 深度优先搜索一次，就可以处理出所有能够通过链式推导到达 ϵ 的字符集合了，我们使用数组 `epsilon_reachable[u]` 来标记点 u 是否能够到达 ϵ, 处理完之后就得替换空产生式了，我们得先删掉产生式中的 ϵ，然后去用这些产生式替换其余所有产生式中含有这些能到达 ϵ 的字符。

如何替换呢? 对于上述文法， X, Y 都能推导到 ϵ, 所以对于产生式 XYX, 就可以有这些替换的结果:

$$XYX|XY|YX|XX|X|Y$$

这是在暴力地去枚举要用 ϵ 替换哪些字符，考虑替换 1 个的组合，替换 2 个的组合，替换 3 个的组合等等等等，所以替换过程的时间复杂度为 $C_n^1 + C_n^2 + C_n^3 + \ldots + C_n^n \sim O(2^n)$, 替换过程的核心代码如下, 这是从最终源码的第 218 行开始的一小块代码:

```
for (auto &pro : all_production) {
    auto &to_state = pro.second;
    vector<string> new_right;
    vector<string> to_temp;
    if (to_state.empty()) continue;  // C -> _
    for (auto str : to_state) {      // str : XYX
        vector<int> pos;             //所有空项的下标
        for (int i = 0; str[i]; i++) {
```

```
            if (all_ep.count(string(1, str[i]))) {
                pos.push_back(i);
            }
        }
        if (!pos.empty()) {
            for (int i = 1; i < (1 << pos.size()); i++) { //指数枚举
                string temp = str;
                for (int k = 0; k < pos.size(); k++) {
                    if ((i >> k) & 1) { //位运算技巧处理
                        temp[pos[k]] = '#';
                    }
                }
                to_temp.push_back(temp);
            }
        }
    }
    for (auto t : to_temp) to_state.push_back(t);
}
```

最终替换完所有含有 ϵ 的产生式，就彻底消除了空产生式，此算法的复杂度瓶颈在于指数地枚举替换，目前来看这一步没有什么可以优化时间复杂度的可能性。

5.4 消去单产生式

为什么我们要把消除单产生式放在最后而不是把消除无用符号放在最后呢？这样其实并不会影响最终结果的正确性，而且会很大程度减少代码实现的复杂程度。这是因为每消去一次空产生式，我们需要重新建图，在图中删去那些不可达的点，以及最终替换 ϵ 之后的产生式，让图中保存信息始终和产生式保持同步。如果我们不先删除无用符号，就会让图中的点之间的关系变得难以维护。在最后消去单产生式之后，在代码实现上顺便会将无用符号删除，所以不用担心。同样以一个文法的例子来讲述消除单产生式的思路:

$$
\begin{aligned}
&S \to A \\
&A \to B|bS|a \\
&B \to AB|\epsilon
\end{aligned}
$$

这个文法是从上课的课件 4.2 章节上摘下来的，只有短短三行，但是确实是个非常强的样例, 我们开始写了好几个版本的代码都没法正确处理这个文法。因为仔细推导一下这个文法，会有很多的环形推导关系，以及自环推导，比如 $A \to B, B \to A, B \to B$, 稍不注意就会漏掉一些产生式。

这个文法中的单产生式有 $S \to A, A \to B$，要处理 S 就得先处理替换 A 为非单产生式，同样要处理 A 就得先处理 B 将其替换为非单产生式。所以仔细想一想这个过程依然是递归的，所以我们依旧使用深度优先搜索的思路。从 S 开始递归搜索，将其能推导到的所有产生式都全部替换成非单产生式才行，递归搜索的临界回溯边界是那些右边推导项全为非单产生式的字符，比如 $X \to aB|cc|Dd$ 这样的产生式遇到就可以回溯了，核心代码如下:

```
string dfs_unit(int u, int fa) {
    auto &vec = all_production[start_symbol_rev[u]];
    bool flag = true;
    for (auto &t : vec) {
        if (t.size() == 1 && t[0] >= 'A' &&
            t[0] <= 'Z' && t != start_symbol_rev[u] &&
            start_symbol[t] != fa) {
            //不要搜索自环
```

```
            flag = false;
            t = dfs_unit(start_symbol[t], u);
        }
    }
    if (flag) {  //右边没有单产生式了
        string res;
        for (auto t : vec) {
            if (t.size() > 1 || (t.size() == 1 && t[0] >= 'a' && t[0] <=
'z'))
                res += "#" + t;
        }
        return res;
    }
    return "";
}
```

这里涉及到一些字符串处理的细枝末节问题我不详细赘述，本质就是字符串各种连接处理替换，和文法消除的逻辑没有太大关系，所以一笔带过。这里的 DFS 单产生式的代码从源代码的第 309 行开始。同时在搜索过程中还要注意不要搜索自环，也就是 $A \to A$ 这种图中的边，会让程序死循环下去。消除单产生式由于同样只做了一次 DFS, 所以其时间复杂度同样为 $O(n)$, n 为字符集大小。

至此我们就分别按消除无用符号，空产生式，单产生式，无用符号的步骤结束了整个化简过程，核心算法的部分就这些了，其实总结一下就是我们全程只用到了深度优先搜索这样的递归思路去处理，没有其他什么复杂高深的算法涉及到里面去。

6. 程序的输入输出和执行效果

在这里就是要测试程序的鲁棒性，以及是否能处理各种极端的样例，以及会导致算法某个细节没有考虑完全而失效无法处理的情况。这也是为什么整个程序近 500 行，因为要处理很多 corner case，我们在代码的注释中都详细有说明。对于每个化简的步骤，消除无用符号，消除空产生式，消除单产生式，我们都构造了很多比较强的样例来测试，具体文件在 GitHub 代码仓库的此处, 这里就分别挑典型的样例来测一下。

- 本地编译以及执行程序:

```
g++ main.cpp -std=c++14 -fexec-charset=GBK -o test

./test
```

这里加上编译器参数 `-fexec-charset=GBK` 是因为 Windows 系统默认字符集没有开启 utf-8 beta 选项，为了在终端中正确输出中文，需要以 GBK 编码来编译。如果你的 Windows 系统开启了 utf-8 beta 选项，就不需要了，默认自己不去修改是没有开启的, 注意编译器版本不要太老，需要至少支持 C++11 标准。

输入格式很简单，第一行先输入起始字符的个数，第二行输入所有起始字符，用空格隔开，第三行输入文法的产生式行数，接下来就是所有的产生式了，一个一行就行。注意空字符用字符串 `epsilon` 替代，因为特殊符号 ϵ 并不是标准 ASCII 字符，测试如下所示。

- 实验文档要求中的简单样例:

```
5
S A B C D
5
S -> a | bA | B | ccD
A -> abB | epsilon
B -> aA
C -> ddC
D -> ddd
```

一共有 5 个起始字符 S, A, B, C, D，然后有 5 行产生式，这个例子就是实验要求 word 文档里面的，很简单没什么极端的情况，甚至都不需要递归处理，运行化简结果如下:

```
PS C:\Users\x\code\CFG-Simplification> .\test.exe

请输入起始符号数量: 5

请输入所有起始符: S A B C D

请输入产生式的数量: 5

请输入所有产生式, 确保文法最初从起始符 S 开始, 每个一行, 空字符使用 epsilon 替代:

S -> a | bA | B | ccD
A -> abB | epsilon
B -> aA
C -> ddC
D -> ddd

化简结果如下:

A -> abB
B -> a | aA
D -> ddd
S -> a | aA | b | bA | ccD
```

- 实验报告开头引入的比较复杂的样例，这个样例有 10 个起始字符，10 个产生式，涵盖了所有比较特殊的空产生式情况，输入如下所示:

```
10
S A B X Y C D E F G
10
S -> XYX | A | B
A -> bBX | aCbC
B -> bbb | BX | D
D -> C
X -> aX | epsilon
Y -> bY | epsilon
C -> epsilon
E -> eE | F
F -> fF | G
G -> gG
```

测试结果如下:

```
PS C:\Users\x\code\CFG-Simplification> .\test.exe

请输入起始符号数量: 10

请输入所有起始符: S A B X Y C D E F G

请输入产生式的数量: 10

请输入所有产生式, 确保文法最初从起始符 S 开始, 每个一行, 空字符使用 epsilon 替代:

S -> XYX | A | B
A -> bBX | aCbC
B -> bbb | BX | D
D -> C
X -> aX | epsilon
Y -> bY | epsilon
C -> epsilon
E -> eE | F
F -> fF | G
G -> gG

化简结果如下:

A -> aCb | aCbC | ab | abC | b | bB | bBX | bX
B -> BX | a | aX | bbb
S -> BX | XX | XY | XYX | YX | a | aCb | aCbC | aX | ab | abC | b | bB | bBX
| bX | bY | bbb
X -> a | aX
Y -> b | bY
```

- 课件中提到的只有三行的文法，一个较强的样例:

```
3
S A B
3
S -> A
A -> B | bS | a
B -> AB | epsilon
```

运行结果如下:

```
PS C:\Users\x\code\CFG-Simplification> .\test.exe

请输入起始符号数量: 3

请输入所有起始符: S A B

请输入产生式的数量: 3

请输入所有产生式, 确保文法最初从起始符 S 开始, 每个一行, 空字符使用 epsilon 替代:

S -> A
A -> B | bS | a
B -> AB | epsilon
```

```
14
15 化简结果如下：
16
17 A -> AB | a | b | bS
18 B -> AB | a | b | bS
19 S -> AB | a | b | bS
```

这和课件中的化简结果完全一致，如下图所示：

- 另外一个比较复杂的样例，涵盖了消除无用符号里面的特殊单递归式：

```
1  9
2  S A P B C D E F G
3  9
4  S -> a | E | F | bA | B | ccD | Pc
5  A -> abB | epsilon | P | F
6  P -> ppP | A | C | E | G
7  B -> aA
8  C -> ddC | D
9  D -> ddd
10 E -> eE | F
11 F -> fF | G
12 G -> gG
```

化简结果如下：

```
1  请输入起始符号数量: 9
2
3  请输入所有起始符: S A P B C D E F G
4
5  请输入产生式的数量: 9
6
7  请输入所有产生式, 确保文法最初从起始符 S 开始, 每个一行, 空字符使用 epsilon 替代:
8
9  S -> a | E | F | bA | B | ccD | Pc
10 A -> abB | epsilon | P | F
```

```
P -> ppP | A | C | E | G
B -> aA
C -> ddC | D
D -> ddd
E -> eE | F
F -> fF | G
G -> gG

化简结果如下:

A -> abB
B -> a | aA
C -> ddC | ddd
D -> ddd
P -> abB | ppP
S -> Pc | a | aA | b | bA | ccD
```

7. 改进思路和方法

虽然构造了很多比较强的测试样例，以及考虑了样例可能会导致的边界情况，但是毕竟上下文无关文法要处理的情况还有不少，可能程序中还有没有考虑到的 corner case, 一些很极端的样例可能会得到不完整的化简结果。

当然报告里面所描述的算法就是很基本的按照步骤 [1] [2] 分别消去无用符号，空产生式，单产生式。除了消除空产生式的步骤里面需要以 $O(2^n)$ 的复杂度去暴力枚举所有空产生式来替换以外，其余的两个步骤都是使用搜索从而是稳定 $O(n)$ 线性复杂度的。所以替换空产生式应该是整个算法的瓶颈。

除了这样按照三个步骤依次化简之外，还有更快的算法吗？出于猎奇心理我还是去 Google 搜索了一下相关的论文，貌似对于 context-free grammar simplification 这个问题并没有太多的学术界研究，搜索到的往往都是一些名校公开的 lecture notes，所讲述的方法都和上课讲述的几个算法一模一样，但是还是找到了一篇 1994 年的论文 ***Simplification of context-free grammar through Petri net*** [3]，我粗略浏览了一遍作者提到的 Petri net 计算模型，论文的 abstract 如下:

> The object of this paper is to exploit a PN model to simplify a context-free grammar (CFG). This is done first by representing the CFG by a PN. Then algorithms to eliminate λ- and unit-productions are developed. The proposed technique is novel in the sense that it provides a better representation and understanding of the problem. It is simple, requires less computation and is easily implemented on computers.

但是似乎对于算法复杂度并没有显著的提升, 只是引入 PN 计算模型来优化减少计算的步骤。

还有就是参考了 LLVM 编译器内部对于 CFG 化简的实现 [4], 编译器源码的代码量是巨大的，毕竟一个编译器要实现的必须是完整的，高效的，处理了所有情况的算法，我也没有再继续去花时间读 LLVM 的源码。

Reference:

- [1] Tutorial in English 1
- [2] Tutorial in English 2
- [3] Simplification of context-free grammar through Petri net
- [4] LLVM Simplify and canonicalize the CFG