



北京邮电大学

Beijing University of Posts and Telecommunications

# 形式语言与自动机 实验二

## 上下文无关文法与下推自动机

### 实验报告

学    院：	计算机学院
组长班级：	2022211301
组长姓名：	卢安来
时    间：	2024 年 5 月 31 日

# 目录

<b>形式语言与自动机 实验二 上下文无关文法与下推自动机 实验报告 .....</b>	<b>1</b>
1. 实验目的.....	4
1.1 关于上下文无关文法.....	4
1.2 关于下推自动机.....	4
2. 实验内容.....	4
2.1 关于上下文无关文法.....	4
2.2 关于下推自动机.....	4
3. 实验小组成员和分工.....	4
4. 实验设计.....	5
4.1 开发环境描述.....	5
4.2 消除无用符号.....	5
4.2.1 设计思路 .....	5
4.2.2 代码实现 .....	6
4.3 消除 $\epsilon$ 产生式.....	8
4.3.1 设计思路 .....	8
4.3.2 代码实现 .....	9
4.4 消除单产生式.....	11
4.4.1 设计思路 .....	11
4.4.2 代码实现 .....	12
4.5 空栈接受 PDA 到 CFG 的转换 .....	13
4.5.1 设计思路 .....	13
4.5.2 代码实现 .....	14
4.6 终态接受 PDA 到空栈接受 PDA 的转换.....	16
4.6.1 设计思路 .....	16

4.6.2	代码实现 .....	16
4.7	输入输出和流程设计 .....	17
4.7.1	CFG 输入格式 .....	17
4.7.2	PDA 输入格式 .....	18
4.7.3	输出格式 .....	19
4.7.4	执行流程 .....	19
4.8	测试案例设计 .....	20
5.	全过程测试 .....	21
5.1	测试环境描述 .....	21
5.2	测试过程 .....	21
5.2.1	CFG 化简功能测试 .....	21
5.2.2	PDA 转化功能测试 .....	23
6.	改进的思路和方法 .....	25
6.1	增加更多的 CFG 转换算法 .....	25
6.2	图形化输入输出界面 .....	25
6.3	集成更多的自动化功能 .....	25
7.	实验总结 .....	26
<b>附录 A：参考文献 .....</b>		<b>27</b>

## 1. 实验目的

### 1.1 关于上下文无关文法

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

### 1.2 关于下推自动机

编程实现由下推自动机构造等价的上下文无关文法的算法。

## 2. 实验内容

### 2.1 关于上下文无关文法

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。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  $\varepsilon$  产生式、单产生式、以及无用符号。

本实验要求编程实现消除上下文无关文法中的  $\varepsilon$  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  $\varepsilon$  产生式、单产生式、无用符号的上下文无关文法。

### 2.2 关于下推自动机

下推自动机和上下文无关文法是用于描述上下文无关语言的两种方式。

本实验要求编程实现由下推自动机构造等价的上下文无关文法的算法。输入为一个下推自动机，输出为与该下推自动机等价的没有  $\varepsilon$  产生式、单产生式、无用符号的上下文无关文法。

## 3. 实验小组成员和分工

见表 3-1 小组成员和分工。

表 3-1 小组成员和分工

班级	学号	姓名	实验分工	贡献度 <sup>1</sup>
2022211301	2022212720	卢安来	方案设计 消除单产生式算法实现 实验报告文档撰写	25%
	2022210985	王若竹	空栈接受 PDA 到 CFG 的转化 终态接受 PDA 转空栈接受	25%
	2022211363	谢牧航	消除 $\varepsilon$ 产生式算法实现 输入输出界面设计	25%
	2022212263	苏柏闻	消除无用符号算法实现 代码审查和测试设计	25%

## 4. 实验设计

### 4.1 开发环境描述

见表 4-1 开发环境描述。

表 4-1 开发环境描述

配置项目	描述	备注
编程语言和版本 测试工具 版本控制工具 代码托管平台	Python 3.12.3 64-bit	依赖包见 requirements.txt
	Pytest 8.1.1	
	Git	
	GitHub	

### 4.2 消除无用符号

#### 4.2.1 设计思路

设上下文无关文法  $G = (N, T, P, S)$ ，如果对  $X \in N \cup T$ ，某些  $\alpha, \beta, \omega \in T^*$ ，当  $X$  出现在推导  $S \xRightarrow[G]{*} \alpha X \beta \xRightarrow[G]{*} \omega$ ，则  $X$  是有用符号，如果  $X$  不出现在  $S \xRightarrow[G]{*} \omega$  中，则  $X$  是无用符号。

根据定义判定一个符号是否有用，需从两个方面进行分析：

<sup>1</sup>根据实验分工进行 Shapley 方法计算所得。因为我们的团队协作十分紧密，因此笔者认定每位成员的边际效益均为 1。

其一，在给定一个上下文无关文法  $G$  中，对每一个非终结符  $A$ ，能否由  $A$  推导出某些终结字符串。

其二， $A$  是否能出现在由起始符  $S$  开始的推导句型之中。

因此，设计算法 4-1 和算法 4-2 用于消除无用符号。

#### 算法 4-1 找出有用的非终结符

##### Algorithm 1 找出有用的非终结符

**Input:**  $G = (N, T, P, S)$ : 上下文无关文法

**Output:**  $N'$ : 有用的非终结符集合

```

1:  $N' \leftarrow \{\}$                                 ▷ 初始化有用的非终结符集合
2:  $N'' \leftarrow \{A \mid A \rightarrow \omega \text{ 且 } \omega \in T^*\}$     ▷ 找到直接生成终结字符串的非终结符
3: while  $N' \neq N''$  do
4:    $N' \leftarrow N''$                                 ▷ 更新有用的非终结符集合
5:    $N'' \leftarrow N' \cup \{A \mid A \rightarrow \alpha \text{ 且 } \alpha \in (T \cup N')^*\}$     ▷ 找到间接生成终结字符串的非终结符
6: end while
7: return  $N''$                                 ▷ 返回最终的有用非终结符集合

```

#### 算法 4-2 找出有用的符号

##### Algorithm 2 找出有用的符号

**Input:**  $G = (N, T, P, S)$ : 上下文无关文法

**Output:**  $N'$ : 有用的符号的集合

```

1:  $N' \leftarrow \{S\}$                                 ▷ 初始化有用符号集合为起始符号
2:  $N'' \leftarrow N' \cup \{X \mid X \in N \text{ 且存在 } A \rightarrow aX\beta \in P, \text{ 其中 } A \in N'\}$ 
3: while  $N' \neq N''$  do
4:    $N' \leftarrow N''$                                 ▷ 更新有用符号集合
5:    $N'' \leftarrow N' \cup \{X \mid X \in N \text{ 且存在 } A \rightarrow aX\beta \in P, \text{ 其中 } A \in N'\}$     ▷ 找到所有出现在推导中的符号
6: end while
7: return  $N'$ 

```

分析可知，算法 4-1 和算法 4-2 的时间复杂度为  $\Theta(n)$ ，产生的新文法的规模为  $\Theta(n)$ ，其中  $n$  为原文法的规模。

## 4.2.2 代码实现

见 `src/algorithm/eliminate_useless_symbol.py` 或下方代码。

```

from exceptions import *
from datastructure.cfg import *

__all__: list[str] = ["eliminate_useless_symbol"]

def _is_left_contained(
    cfg: CFG, production: production_t, symbols: set[nonterminal_t]
) -> bool:
    left, _ = production
    return all(symbol not in cfg.N or symbol in symbols for symbol in left)

```

```
def _is_right_contained(
    cfg: CFG, production: production_t, symbols: set[nonterminal_t]
) -> bool:
    _, right = production
    return all(symbol not in cfg.N or symbol in symbols for symbol in right)

def _get_reachable(cfg: CFG, symbols: set[nonterminal_t]) ->
    set[nonterminal_t]:
    return {
        symbol
        for production in cfg.P
        if _is_left_contained(cfg=cfg, production=production, symbols=symbols)
        for _, right in [production]
        for symbol in right
        if symbol in cfg.N
    }

def _get_reachable_nonterminal(
    cfg: CFG,
) -> set[nonterminal_t]:
    current_set: set[nonterminal_t] = {cfg.S}
    next_set: set[nonterminal_t] = (
        _get_reachable(cfg=cfg, symbols=current_set) | current_set
    )
    while current_set != next_set:
        current_set = next_set
        next_set = _get_reachable(cfg=cfg, symbols=current_set) | current_set
    return current_set

def _get_productive(cfg: CFG, symbols: set[nonterminal_t]) ->
    set[nonterminal_t]:
    return {
        symbol
        for production in cfg.P
        if _is_right_contained(cfg=cfg, production=production,
symbols=symbols)
        for left, _ in [production]
        for symbol in left
        if symbol in cfg.N
    }

def _get_productive_nonterminal(
    cfg: CFG,
) -> set[nonterminal_t]:
    current_set: set[nonterminal_t] = set()
    next_set: set[nonterminal_t] = (
        _get_productive(cfg=cfg, symbols=current_set) | current_set
    )
    while current_set != next_set:
        current_set = next_set
        next_set = _get_productive(cfg=cfg, symbols=current_set) | current_set
    return current_set
```

```

def _reduce_grammar_by_nonterminal(cfg: CFG, nonterminals: set[nonterminal_t])
-> CFG:
    new_productions: set[production_t] = {
        production
        for production in cfg.P
        if all(
            symbol in cfg.T or symbol in nonterminals
            for part in production
            for symbol in part
        )
    }

    new_terminals: set[terminal_t] = {
        symbol
        for production in new_productions
        for part in production
        for symbol in part
        if symbol not in nonterminals
    }

    new_nonterminals: set[nonterminal_t] = nonterminals
    new_start = cfg.S
    return CFG(N=new_nonterminals, T=new_terminals, P=new_productions,
S=new_start)

def eliminate_useless_symbol(cfg: CFG) -> CFG:
    if not cfg.is_valid():
        raise CFGInvalid("Invalid CFG.")
    nonterminals: set[nonterminal_t] = _get_productive_nonterminal(cfg=cfg)
    if cfg.S not in nonterminals:
        raise CFGEmptyError("CFG is empty.")
    cfg = _reduce_grammar_by_nonterminal(cfg=cfg, nonterminals=nonterminals)
    nonterminals: set[nonterminal_t] = _get_reachable_nonterminal(cfg=cfg)
    if cfg.S not in nonterminals:
        raise CFGEmptyError("CFG is empty.")
    cfg = _reduce_grammar_by_nonterminal(cfg=cfg, nonterminals=nonterminals)
    return cfg

```

## 4.3 消除 $\varepsilon$ 产生式

### 4.3.1 设计思路

对上下文无关文法来说，如果带有形如  $A \rightarrow \varepsilon$  的生成式（即  $\varepsilon$  产生式），会给推导带来麻烦，一般情况下应删除。但有一个例外，如果上下文无关文法  $G$  生成的语言  $L(G)$  中含有  $\varepsilon$ ，那就不能删除生成式  $S \rightarrow \varepsilon$ 。



因此，对于上下文无关文法  $G = (N, T, P, S)$ ，如果生成式  $P$  中无任何  $\varepsilon$  产生式，或只有一个生成式  $S \rightarrow \varepsilon$ ，且  $S$  不在任何生成式的右部，则称  $G$  为无  $\varepsilon$  文法。

因此，设计算法 4-3 用于消除  $\varepsilon$  产生式。

算法 4-3 消除  $\varepsilon$  产生式

**Algorithm 3** 消除上下文无关文法中的  $\varepsilon$  生成式

**Input:** 上下文无关文法  $G = (N, T, P, S)$

**Output:** 消除  $\varepsilon$  生成式后的上下文无关文法  $G' = (N', T, P', S')$

```

1: 确定可空的非终结符集合  $N_\varepsilon$                                 ▷ 找出所有可以生成空串的非终结符
2: 初始化  $P'$  为  $\emptyset$                                           ▷ 初始化新的生成式集合
3: for all  $A \rightarrow \alpha \in P$  do                                  ▷ 遍历每个生成式
4:   if  $\alpha \neq \varepsilon$  then                                      ▷ 跳过  $\varepsilon$  生成式
5:     找出  $\alpha$  中属于  $N_\varepsilon$  的符号位置                      ▷ 标记可空符号的位置
6:     for 每个可能的子集 do                                    ▷ 生成所有子集的组合
7:       构造去除某些位置符号后的  $\alpha'$                       ▷ 去除子集中符号后得到新的右部
8:       if  $\alpha' \neq \varepsilon$  then                              ▷ 确保新右部非空
9:          $P' \leftarrow P' \cup \{A \rightarrow \alpha'\}$           ▷ 添加新的生成式
10:      end if
11:    end for
12:  end if
13: end for
14: if  $S \in N_\varepsilon$  并且  $S$  出现在某些生成式的右侧 then        ▷ 处理开始符号可空且出现在右侧的情况
15:   引入新的开始符号  $S'$                                     ▷ 引入新的开始符号
16:    $N' \leftarrow N \cup \{S'\}$                                 ▷ 更新非终结符集合
17:    $P' \leftarrow P' \cup \{S' \rightarrow S, S' \rightarrow \varepsilon\}$   ▷ 添加新的生成式
18: else
19:    $N' \leftarrow N$                                            ▷ 非终结符集合不变
20:   if  $S \in N_\varepsilon$  then                                       ▷ 处理开始符号可空的情况
21:      $P' \leftarrow P' \cup \{S \rightarrow \varepsilon\}$                 ▷ 添加  $\varepsilon$  生成式
22:   end if
23:    $S' \leftarrow S$                                            ▷ 开始符号不变
24: end if
25: return  $(N', T, P', S')$                                     ▷ 返回新的文法

```

分析可知，算法 4-3 的时间复杂度为  $\Theta(2^n)$ ，产生的新文法的规模为  $\Theta(2^n)$ ，其中  $n$  为原文法的规模。

### 4.3.2 代码实现

见 `src/algorithm/eliminate_epsilon_production.py` 或下方代码。

```

from exceptions import *
from datastructure.cfg import *

__all__: list[str] = ["eliminate_epsilon_production"]

def _is_right_nullable(
    cfg: CFG, production: production_t, nullable_symbols: set[nonterminal_t]
) -> bool:

```

```

_, right = production
return all(symbol in nullable_symbols for symbol in right)

def _get_nullable(
    cfg: CFG, nullable_nonterminals: set[nonterminal_t]
) -> set[nonterminal_t]:
    return {
        symbol
        for production in cfg.P
        if _is_right_nullable(
            cfg=cfg, production=production,
            nullable_symbols=nullable_nonterminals
        )
        for left, _ in [production]
        for symbol in left
        if symbol in cfg.N
    }

def _get_nullable_nonterminal(cfg: CFG) -> set[nonterminal_t]:
    current_set: set[nonterminal_t] = set()
    next_set: set[nonterminal_t] = (
        _get_nullable(cfg=cfg, nullable_nonterminals=current_set) |
        current_set
    )
    while current_set != next_set:
        current_set = next_set
        next_set: set[nonterminal_t] = (
            _get_nullable(cfg=cfg, nullable_nonterminals=current_set) |
            current_set
        )
    return current_set

def eliminate_epsilon_production(cfg: CFG) -> CFG:
    if not cfg.is_valid():
        raise CFGInvalid("Invalid CFG.")
    nullable_nonterminals: set[nonterminal_t] = _get_nullable_nonterminal(cfg)
    new_productions: set[production_t] = set()
    for left_hand, right_hand in cfg.P:
        if not right_hand:
            continue
        nullable_indices: list[int] = [
            i for i, symbol in enumerate(right_hand) if symbol in
            nullable_nonterminals
        ]
        for i in range(0, 2 ** len(nullable_indices)):
            new_right_hand = list(right_hand)
            for j, index in enumerate(nullable_indices):
                if not (i & (1 << j)):
                    new_right_hand[index] = ""
            filtered_right_hand = tuple(
                symbol for symbol in new_right_hand if symbol is not ""
            )
            if filtered_right_hand:

```

```

        new_productions.add((left_hand, filtered_right_hand))
    if cfg.S in nullable_nonterminals and any(cfg.S in right for _, right in
cfg.P):
        new_start: nonterminal_t = cfg.S + ""
        new_nonterminals: set[nonterminal_t] = cfg.N | {new_start}
        new_productions.add(((new_start,), (cfg.S,)))
        new_productions.add(((new_start,), ()))
    else:
        new_nonterminals = cfg.N
        if cfg.S in nullable_nonterminals:
            new_productions.add(((cfg.S,), ()))
        new_start = cfg.S
    return CFG(N=new_nonterminals, T=cfg.T, P=new_productions, S=new_start)

```

## 4.4 消除单产生式

### 4.4.1 设计思路

在生成式中，形式为  $A \rightarrow B$  的生成式，其中  $A, B$  为非终结符，生成式的右边仅是单个非终结符，称为单生成式。在文法的变换中，可将单生成式删除。如果文法  $G$  是无  $\varepsilon$  生成式的上下文无关文法，但存在单生成式，可利用算法构成一个无单生成式的等效文法  $G$ 。

因此，设计算法 4-4 用于消除单产生式。

算法 4-4 消除单产生式

**Algorithm 4** 消除单生成式

**Input:**  $G = (N, T, P, S)$

▷ 上下文无关文法

**Output:**  $G' = (N, T, P', S)$

▷ 消除单生成式后的文法

1: for all  $A \in N$  do

2:    $N_A \leftarrow \{A\}$

▷ 初始化集合

3:    $N' \leftarrow \{C \mid B \rightarrow C \text{ 且 } B \in N_A\} \cup N_A$

4:   repeat

5:      $N_A \leftarrow N'$

6:      $N' \leftarrow \{C \mid B \rightarrow C \text{ 且 } B \in N_A\} \cup N_A$

7:   until  $N_A = N'$

8: end for

9:  $P' \leftarrow \{\}$

▷ 初始化生成式集合

10: for all  $B \rightarrow \alpha \in P$  do

11:   if  $\alpha \notin N$  then

12:     for all  $A \in N_B$  do

13:        $P' \leftarrow P' \cup \{A \rightarrow \alpha\}$

14:     end for

15:   end if

16: end for

17: return  $(N, T, P', S)$

▷ 返回消除单生成式后的文法

分析可知，算法 4-4 的时间复杂度为  $\Theta(n^2)$ ，产生的新文法的规模为  $\Theta(n^2)$ ，其中  $n$  为原文法的规模。

## 4.4.2 代码实现

见 `src/algorithm/eliminate_unit_production.py` 或下方代码。

```
from exceptions import *
from datastructure.cfg import *

__all__: list[str] = ["eliminate_unit_production"]

def _is_unit_production(cfg: CFG, production: production_t) -> bool:
    left, right = production
    is_left_unit: bool = len(left) == 1 and left[0] in cfg.N
    is_right_unit: bool = len(right) == 1 and right[0] in cfg.N
    return is_left_unit and is_right_unit

def _get_unit_reachable(
    unit_productions: set[production_t], symbols: set[nonterminal_t]
) -> set[nonterminal_t]:
    return {left[0] for left, right in unit_productions if right[0] in symbols}

def _get_unit_production_left_side(
    unit_productions: set[production_t], symbol: nonterminal_t
) -> set[nonterminal_t]:
    current_set: set[nonterminal_t] = {symbol}
    next_set: set[str] = (
        _get_unit_reachable(unit_productions=unit_productions,
                           symbols=current_set)
        | current_set
    )
    while current_set != next_set:
        current_set = next_set
        next_set: set[str] = (
            _get_unit_reachable(unit_productions=unit_productions,
                               symbols=current_set)
            | current_set
        )
    return current_set

def eliminate_unit_production(cfg: CFG) -> CFG:
    if not cfg.is_valid():
        raise CFGInvalid("Invalid CFG.")
    unit_productions: set[production_t] = {
        production
        for production in cfg.P
        if _is_unit_production(cfg=cfg, production=production)
    }
    new_productions: set[production_t] = set()
    for symbol in cfg.N:
```

```

    unit_reachable_symbols: set[nonterminal_t] =
_get_unit_production_left_side(
    unit_productions=unit_productions, symbol=symbol
)
rights: set[tuple[symbol_t, ...]] = {
    right
    for left, right in cfg.P
    if left == (symbol,) and (len(right) != 1 or (right[0] in cfg.T))
}
for right in rights:
    for left_symbol in unit_reachable_symbols:
        new_productions.add(((left_symbol,), right))
return CFG(N=cfg.N, T=cfg.T, P=new_productions, S=cfg.S)

```

## 4.5 空栈接受 PDA 到 CFG 的转换

### 4.5.1 设计思路

下推自动机和上下文无关文法,是用于描述上下文无关语言的不同方式。当给定一个上下文无关文法  $G$  和由它产生的语言  $L(G)$ , 必存在一个下推自动机  $M$  接受且仅接受语言  $L(M)$ , 使  $L(M) = L(G)$ , 反之亦然。

现有一空栈接受 PDA  $M$ , 要构造一个上下文无关文法  $G$ , 它产生  $M$  接受的所有字符串。换句话说, 如果一个字符串能使  $M$  从它的起始状态转移到一个接受状态, 则  $G$  也应该产生这个字符串。

因此, 设计算法 4-5 用于将空栈接受的 PDA 转化为相应的 CFG。

## 算法 4-5 将空栈接受的 PDA 转化为 CFG

<b>Algorithm 5</b> 构造上下文无关文法 $G$ 使 $L(G) = L(M)$	
<b>Input:</b> 下推自动机: $M = (Q, T, \Gamma, \delta, q_0, Z, \emptyset)$	
<b>Output:</b> 上下文无关文法: $G = (N, T, P, S)$	
1: $N \leftarrow \{[q, A, p] \mid q, p \in Q, A \in \Gamma\} \cup \{S\}$	▷ 构造非终结符集合
2: $P \leftarrow \{\}$	▷ 初始化生成式集合
3: <b>for all</b> $q \in Q$ <b>do</b>	
4: $P \leftarrow P \cup \{S \rightarrow [q_0, Z, q]\}$	▷ 添加初始生成式
5: <b>end for</b>	
6: <b>for all</b> $(q, a, A) \in Q \times (T \cup \{\varepsilon\}) \times \Gamma$ <b>do</b>	
7: <b>for all</b> $(p, \beta) \in \delta(q, a, A)$ <b>do</b>	
8: $\beta = B_1 B_2 \dots B_m$	▷ 其中 $B_i \in \Gamma$
9: <b>for all</b> 序列 $q_1, q_2, \dots, q_m \in Q$ <b>do</b>	
10: <b>if</b> $m > 0$ <b>then</b>	
11: $P \leftarrow P \cup \{[q, A, q_{m+1}] \rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \dots [q_m, B_m, q_{m+1}]\}$	▷ 添加生成式
12: <b>else</b>	
13: $P \leftarrow P \cup \{[q, A, p] \rightarrow a\}$	▷ 添加生成式
14: <b>end if</b>	
15: <b>end for</b>	
16: <b>end for</b>	
17: <b>end for</b>	
18: <b>for all</b> $(q, \varepsilon, A) \in \delta(q, \varepsilon, A)$ <b>do</b>	
19: $P \leftarrow P \cup \{[q, A, p] \rightarrow \varepsilon\}$	▷ 添加 $\varepsilon$ 生成式
20: <b>end for</b>	
21: <b>return</b> $(N, T, P, S)$	▷ 返回上下文无关文法

分析可知，算法 4-5 的时间复杂度为  $\Theta(n^3)$ ，产生的文法的规模为  $\Theta(n^3)$ ，其中  $n$  为给定的空栈接受 PDA 的规模。

## 4.5.2 代码实现

见 [src/algorithm/transfer\\_epda\\_to\\_cfg.py](#) 或下方代码。

```
from datastructure.cfg import *
from datastructure.pda import *
import itertools

__all__: list[str] = ["transfer_epda_to_cfg"]

def _eliminate_none(input: tuple[str, ...]) -> tuple[str, ...]:
    return tuple(filter(None, input))

def transfer_epda_to_cfg(epda: PDA) -> CFG:
    assert epda.is_epda(), "The input PDA is not an empty-accepted PDA."

    Q = epda.Q
    T = epda.T
    T_star = T | {""}
    delta = epda.delta
    q0 = epda.q0
    Z0 = epda.Z0

    cfg_S = "S"
    cfg_N: set[nonterminal_t] = set(cfg_S)
    cfg_T: set[character_t] = T
    cfg_P: set[production_t] = set()
```

```

que: list[tuple[state_t, stack_symbol_t, state_t]] = []
vis: set[tuple[state_t, stack_symbol_t, state_t]] = set()

for q in Q:
    cfg_N.add(f"{q0}_{Z0}_{q}")
    cfg_P.add(((("S",), (f"{q0}_{Z0}_{q}",))),)
    que.append((q0, Z0, q))

while len(que):
    q, A, p = que.pop()

    if (q, A, p) in vis:
        continue
    vis.add((q, A, p))

    for a in T_star:
        if (q, a, A) not in delta:
            continue
        for q1, B in delta[q, a, A]:
            m = len(B)
            left_hand: tuple[str] = (f"{q}_{A}_{p}",)
            right_hand_prefix = (a,)
            if m > 1:
                states: set[state_t] = Q
                all_combinations = itertools.product(states, repeat=m - 1)
                for combination in all_combinations:
                    pre: state_t = q1
                    nonterminals: tuple[nonterminal_t, ...] = tuple()
                    for i in range(m - 1):
                        if (pre, B[i], combination[i]) not in vis:
                            que.append((pre, B[i], combination[i]))
                            nonterminals +=
                                (f"{pre}_{B[i]}_{combination[i]}",)
                        pre = combination[i]
                    if (pre, B[m - 1], p) not in vis:
                        que.append((pre, B[m - 1], p))
                    nonterminals += (f"{pre}_{B[m-1]}_{p}",)
                    cfg_N.update(nonterminals)
                    cfg_P.add(
                        (
                            left_hand,
                            _eliminate_none(right_hand_prefix +
nonterminals),
                        )
                    )
            elif m == 1:
                if (q1, B[0], p) not in vis:
                    que.append((q1, B[0], p))
                    cfg_P.add(
                        (
                            left_hand,
                            _eliminate_none(right_hand_prefix +
(f"{q1}_{B[0]}_{p}",)),
                        )
                    )

```

```

elif m == 0 and q1 == p:
    cfg_P.add((left_hand, _eliminate_none(right_hand_prefix)))

return CFG(cfg_N, cfg_T, cfg_P, cfg_S)

```

## 4.6 终态接受 PDA 到空栈接受 PDA 的转换

### 4.6.1 设计思路

本报告「4.5 空栈接受 PDA 到 CFG 的转换」中给出的算法仅适用于空栈接受 PDA，为了将所有的 PDA 都转化为 CFG，还需要实现终态接受 PDA 到空栈接受 PDA 的转换。

因此，设计算法 4-6 用于终态接受 PDA 到空栈接受 PDA 的转换。

算法 4-6 终态接受 PDA 到空栈接受 PDA 的转换

**Algorithm 6** 构造以空栈接受的下推自动机  $M'$

**Input:** 下推自动机  $M = (Q, T, \Gamma, \delta, q_0, Z, F)$  以终止状态接受语言

**Output:** 下推自动机  $M' = (Q' = Q \cup \{q_i, q_f\}, T, \Gamma' = \Gamma \cup \{Z_0\}, \delta', q_i, Z_0, \emptyset)$  以空栈接受语言

```

1:  $\delta' \leftarrow \delta$  ▷ 初始化  $\delta'$  为  $\delta$  的副本
2:  $\delta' \leftarrow \delta' \cup \{(q_i, \varepsilon, \Gamma_0) \rightarrow (q_0, Z_0 Z)\}$  ▷ 初始状态  $q_i$  将  $Z_0$  压入栈底并进入  $M$  的初始状态
3: for all  $(q, a, X) \in Q \times (T \cup \{\varepsilon\}) \times \Gamma$  do
4:   for all  $(p, \gamma) \in \delta(q, a, X)$  do
5:      $\delta' \leftarrow \delta' \cup \{(q, a, X) \rightarrow (p, \gamma)\}$  ▷  $M'$  模拟  $M$  的转移
6:   end for
7: end for
8: for all  $q \in F$  do
9:   for all  $X \in \Gamma \cup \{Z_0\}$  do
10:     $\delta' \leftarrow \delta' \cup \{(q, \varepsilon, X) \rightarrow (q_f, \varepsilon)\}$  ▷ 当  $M$  进入终止状态时,  $M'$  转移到  $q_f$  并清空栈
11:   end for
12: end for
13: for all  $X \in \Gamma \cup \{Z_0\}$  do
14:    $\delta' \leftarrow \delta' \cup \{(q_f, \varepsilon, X) \rightarrow (q_f, \varepsilon)\}$  ▷  $q_f$  状态下继续清空栈
15: end for
16: return  $(Q \cup \{q_i, q_f\}, T, \Gamma \cup \{Z_0\}, \delta', q_i, Z_0, \emptyset)$  ▷ 返回构造好的下推自动机  $M'$ 

```

分析可知，算法 4-6 的时间复杂度为  $\Theta(n)$ ，产生的空栈接受 PDA 的规模为  $\Theta(n)$ ，其中  $n$  为给定的终态接受 PDA 的规模。

### 4.6.2 代码实现

见 src/algorithm/transfer\_fpda\_to\_epfa.py 或下方代码。

```

from datastructure.pda import *

__all__: list[str] = ["transfer_fpda_to_epfa"]

def transfer_fpda_to_epfa(fpda: PDA) -> PDA:

```



```

epda_q0 = fpda.q0 + ""
epda_Z0 = fpda.Z0 + ""
epda_qf = fpda.q0 + ""

epda_Q = fpda.Q | {epda_q0, epda_qf}
epda_T = fpda.T
epda_Gamma = fpda.Gamma | {epda_Z0}
epda_delta = fpda.delta
epda_F = set[state_t]()

epda_delta[(epda_q0, "", epda_Z0)] = {(fpda.q0, (fpda.Z0, epda_Z0))}
for Z in epda_Gamma:
    for f in fpda.F:
        if (f, "", Z) not in epda_delta:
            epda_delta[(f, "", Z)] = set()
            epda_delta[(f, "", Z)].add((epda_qf, ()))
        epda_delta[(epda_qf, "", Z)] = {(epda_qf, ())}

return PDA(
    Q=epda_Q,
    T=epda_T,
    Gamma=epda_Gamma,
    delta=epda_delta,
    q0=epda_q0,
    Z0=epda_Z0,
    F=epda_F,
)

```

## 4.7 输入输出和流程设计

### 4.7.1 CFG 输入格式

本次设计的用户交互界面为命令行输入输出而非图形化界面。具体而言，用户输入要求如下。

对于用户输入的 CFG，要求符合公认的 CFG 的形式化定义，即要求用户按照  $G = (N, T, P, S)$  的格式和顺序输入，其中  $N$  为非终结符集， $T$  为终结符集， $P$  为生成式集， $S$  为起始符号， $\varepsilon$  通过留空表示，具体输入格式要求如。

表格 4-2 CFG 输入格式

概念	格式要求
$N$	用大括号括起，用逗号分隔的字符串
$T$	用大括号括起，用逗号分隔的字符串
单个生成式	用 $\rightarrow$ 分隔左右，符号用空格分隔

概念	格式要求
S	字符串，不含空格

此外，CFG 还有诸如  $N \cap T = \emptyset, S \in N$  之类的数据要求，如果不符合这些要求，程序将会给出提示并要求重新输入。

下面给出了实验指导书中给出的测试用例的输入方法，其中黑色字为程序提示，红色字体指示用户输入内容。

```

please input your CFG
N = {S, A, B, C, D}
T = {a, b, c, d}
P: (input one empty line to end, use -> to separate the left and right sides
S -> a | b A | B | c c D
A -> a b B |
B -> a A
C -> d d C
D -> d d d
S: S

```

### 4.7.2 PDA 输入格式

对于用户输入的 PDA，要求符合公认的 PDA 的形式化定义，即要求用户按照  $M = (Q, T, \Gamma, \delta, q_0, Z_0, F)$  的格式和顺序输入，其中  $Q$  为状态集， $T$  为输入符号集， $\Gamma$  为栈符号集， $\delta$  为状态转移函数， $q_0$  为初始状态， $Z_0$  为初始栈符号， $F$  为接受状态集合， $\varepsilon$  通过留空表示，具体输入格式要求如。

表格 4-3 PDA 输入格式

概念	格式要求
$Q$	用大括号括起，用逗号分隔的字符串
$T$	用大括号括起，用逗号分隔的字符串
$\Gamma$	用大括号括起，用逗号分隔的字符串
转移函数单点定义	格式形如 $(?, ?, ?) \rightarrow \{(? , ?)\}$
$q_0$	字符串，不含空格
$Z_0$	字符串，不含空格
$F$	用大括号括起，用逗号分隔的字符串

此外，PDA 需要满足一些数据要求，如果不符合这些要求，程序将会给出提示并要求重新输入。

下面给出了实验指导书中给出的测试用例的输入方法，其中黑色字为程序提示，红色字体指示用户输入内容。

```
please input your PDA
Q = {q0, q1}
T = {a, b}
Gamma = {B, Z0}
delta: (input one empty line to end, use (?, ?, ?) -> {(?, ?), ...} format)
(q0, b, Z0) -> {(q0, B Z0)}
(q0, b, B) -> {(q0, B B)}
(q0, a, B) -> {(q1, )}
(q1, a, B) -> {(q1, )}
(q1, , B) -> {(q1, )}
(q1, , Z0) -> {(q1, )}

q0 = q0
Z0 = Z0
F = {}
```

### 4.7.3 输出格式

输出格式与 CFG 输入格式类似，见本报告 4.7.1 CFG 输入、图 5-5 CFG 化简结果或图 5-9 PDA 转化结果。

### 4.7.4 执行流程

程序处理流程如图 4-1。可以看出，程序对用户的输入的合法性进行了判定，如果输入非法则将会显示提示信息并要求用户重新输入，具有极好的健壮性。

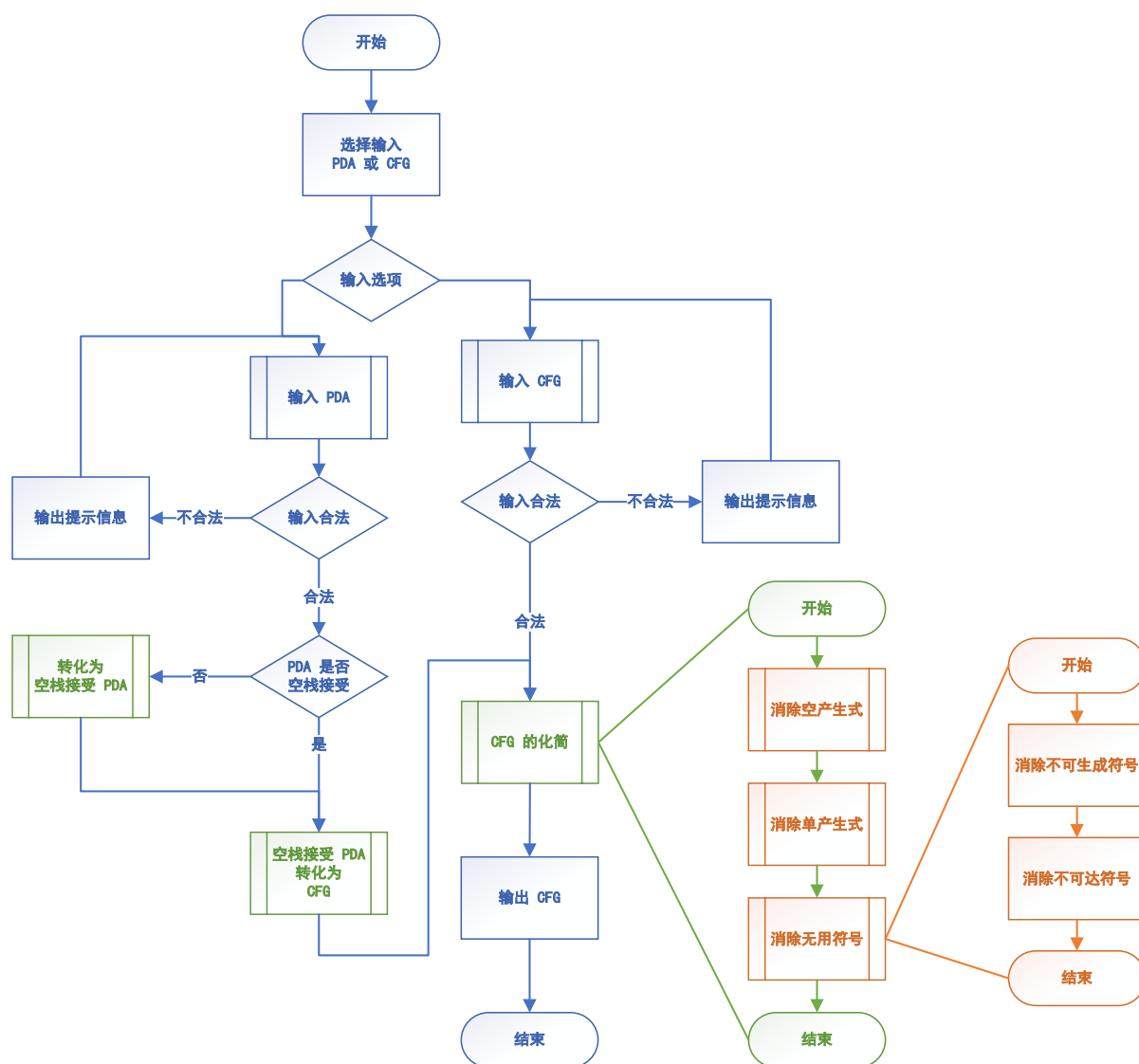


图 4-1 程序处理流程

## 4.8 测试案例设计

为了确保程序的正确性和执行效率，我们精心设置了一些测试案例，包含单元测试（见 `tests/test_*.py`）和整体测试案例，此处因篇幅安排不展开叙述，测试过程见本报告第 5 章 全过程测试。

## 5. 全过程测试

### 5.1 测试环境描述

我们只需要假设在测试环境中已经预装了 64 位的 Python 3.12.2，并且该环境具有稳定的互联网连接。

当然，在本报告附件中附带了打包的可执行程序，可以直接在 Windows 11 系统环境下运行而不需要其他配置。

### 5.2 测试过程

打开终端，切换到项目文件夹，笔者这里为 `D:\buptLab-cfg_pda`，终端启动后显示如图 5-1。

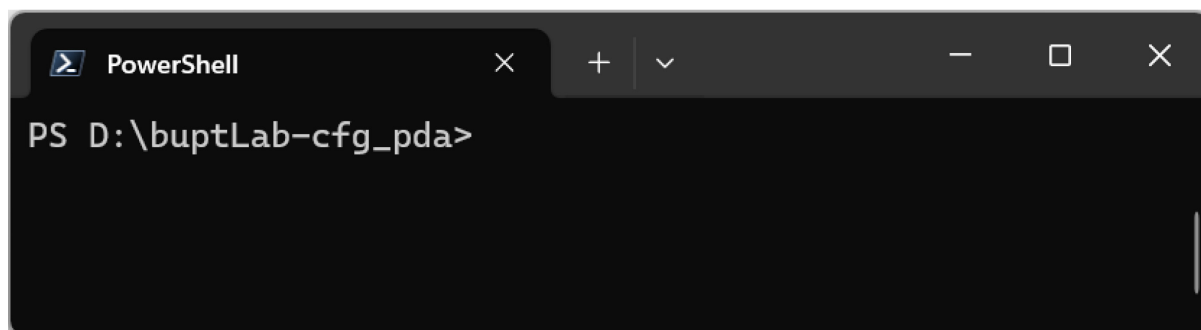


图 5-1 启动终端

键入命令 `python src/main.py`，启动程序，显示如图 5-2。

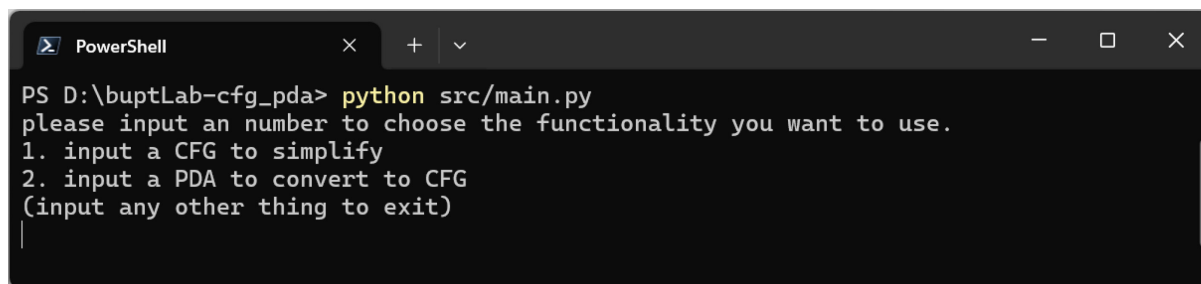
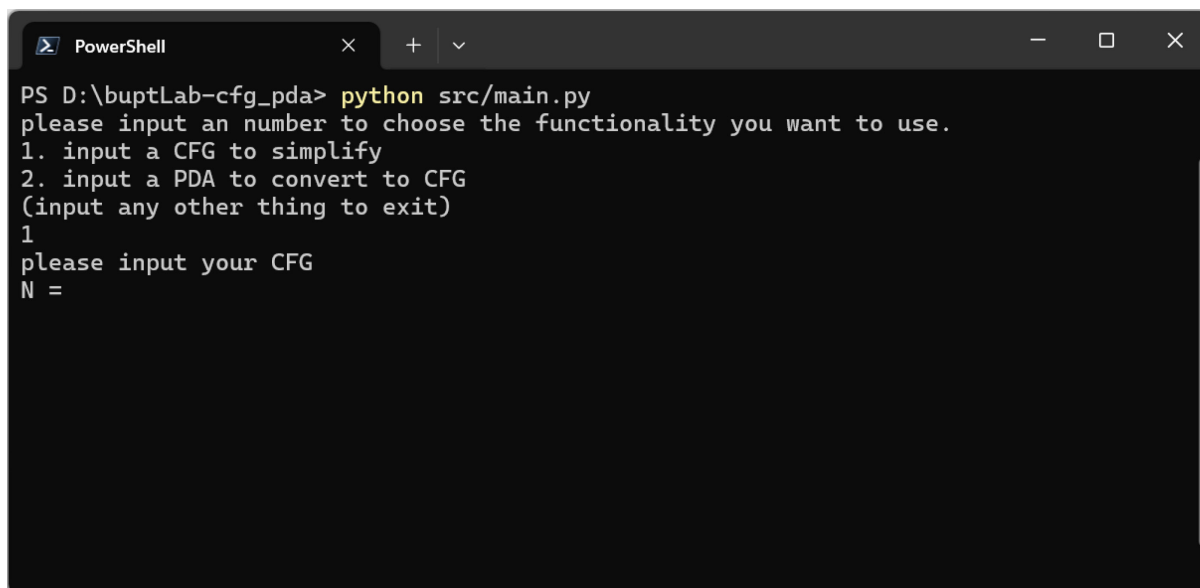


图 5-2 启动程序

#### 5.2.1 CFG 化简功能测试

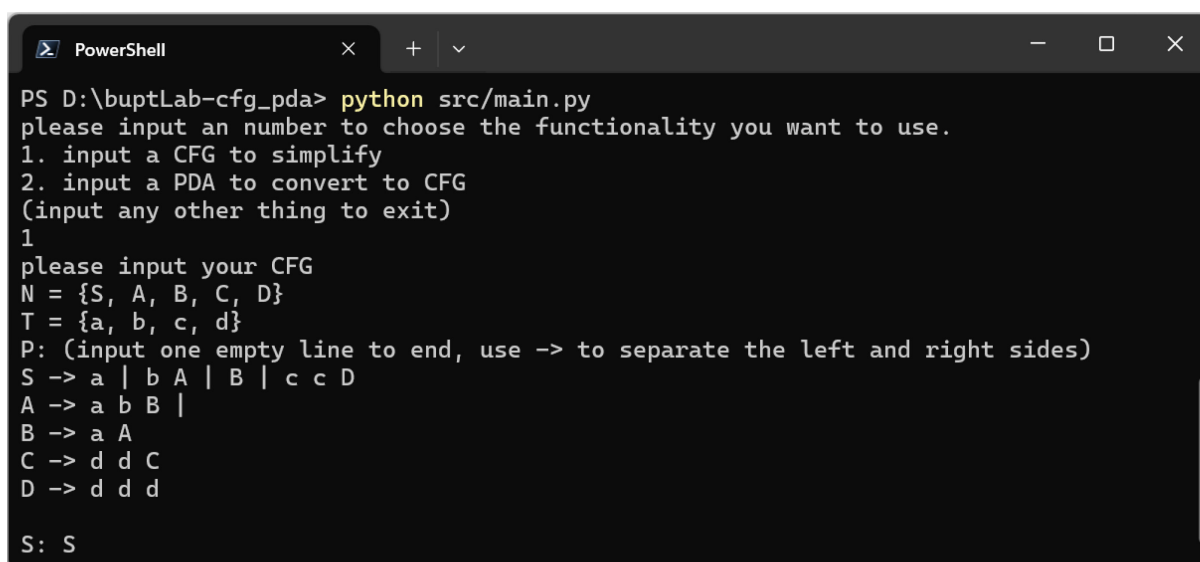
根据选项编号，键入 1 选择 CFG 简化功能，按下回车确认，显示如图 5-3。



```
PS D:\buptLab-cfg_pda> python src/main.py
please input an number to choose the functionality you want to use.
1. input a CFG to simplify
2. input a PDA to convert to CFG
(input any other thing to exit)
1
please input your CFG
N =
```

图 5-3 选择 CFG 化简功能

输入实验指导书中给定样例 CFG，显示如图 5-4。



```
PS D:\buptLab-cfg_pda> python src/main.py
please input an number to choose the functionality you want to use.
1. input a CFG to simplify
2. input a PDA to convert to CFG
(input any other thing to exit)
1
please input your CFG
N = {S, A, B, C, D}
T = {a, b, c, d}
P: (input one empty line to end, use -> to separate the left and right sides)
S -> a | b A | B | c c D
A -> a b B |
B -> a A
C -> d d C
D -> d d d

S: S
```

图 5-4 输入样例 CFG

按下回车，程序输出化简结果显示如图 5-5。

```
PowerShell
#=====
Here is the result.
N = { A, B, D, S }
T = { a, b, c, d }
P:
A -> a b B
B -> a | a A
D -> d d d
S -> a | a A | b | b A | c c D
S = S
Press any key to continue . . .
```

图 5-5 CFG 化简结果

经检验，程序输出无误，CFG 化简功能测试完毕。

### 5.2.2 PDA 转化功能测试

再次键入命令 `python src/main.py`，启动程序，显示如图 5-6。

```
PowerShell
Press any key to continue . . .
PS D:\buptLab-cfg_pda> python src/main.py
please input an number to choose the functionality you want to use.
1. input a CFG to simplify
2. input a PDA to convert to CFG
(input any other thing to exit)
```

图 5-6 启动程序

根据选项键入 2 选择 PDA 转化功能，按下回车，显示如图 5-7。

```
PowerShell
PS D:\buptLab-cfg_pda> python src/main.py
please input an number to choose the functionality you want to use.
1. input a CFG to simplify
2. input a PDA to convert to CFG
(input any other thing to exit)
2
please input your PDA
Q =
```

图 5-7 选择 PDA 转化功能

输入实验指导书中给定 PDA 样例，显示如图 5-8。

```
PowerShell
PS D:\buptLab-cfg_pda> python src/main.py
please input an number to choose the functionality you want to use.
1. input a CFG to simplify
2. input a PDA to convert to CFG
(input any other thing to exit)
2
please input your PDA
Q = {q0, q1}
T = {a, b}
Gamma = {B, Z0}
delta: (input one empty line to end, use (?, ?, ?) -> {(?, ?), ...} format)
(q0, b, Z0) -> {(q0, B Z0)}
(q0, b, B) -> {(q0, B B)}
(q0, a, B) -> {(q1, )}
(q1, a, B) -> {(q1, )}
(q1, , B) -> {(q1, )}
(q1, , Z0) -> {(q1, )}

q0 = q0
Z0 = Z0
F = {}
```

图 5-8 输入样例 PDA

按下回车，程序输出转化并化简结果如图 5-9。

```
PowerShell
#=====
Here is the result.
N = { S, q0_B_q1, q1_B_q1 }
T = { a, b }
P:
S -> b q0_B_q1
q0_B_q1 -> a | b q0_B_q1 | b q0_B_q1 q1_B_q1
q1_B_q1 -> a
S = S
Press any key to continue . . .
```

图 5-9 PDA 转化结果

经检验，程序输出无误，PDA 转化功能测试完毕。

至此，测试完成，按下任意键关闭程序，关闭终端，全过程测试结束。



## 6. 改进的思路和方法

本次实验虽然内容不多，但涉及的知识面仍然比较广泛，有许多值得改进的地方，下面列举几项改进思路和方法。

### 6.1 增加更多的 CFG 转换算法

在本实验中，由于时间限制，我们未能充分探索和实现 CFG 转换算法的多个方面。例如，消除左递归、将 CFG 转换为 Chomsky 范式或 Greibach 范式等重要的转换算法。在未来的工作中，我们将考虑实现这些缺失的算法，以增强系统的功能性和灵活性。

消除左递归是 CFG 转换中的一个重要步骤，它可以提高解析效率，避免在解析过程中出现无限循环。通过消除左递归，我们可以将 CFG 转换为等价的右递归或非递归形式，从而简化解析过程。

将 CFG 转换为 Chomsky 范式或 Greibach 范式也是重要的转换算法。Chomsky 范式是一种特殊的 CFG 形式，它具有更简单的产生式结构，便于进行进一步的分析和处理。Greibach 范式则是另一种特殊的 CFG 形式，它具有更严格的产生式规则，可以用于更高级的语言处理任务。

实现这些转换算法，将为我们未来的研究提供更多的可能性，助力我们将来在形式语言和编译原理领域的发展。

### 6.2 图形化输入输出界面

在本次实验中，我们未能实现图形化界面，这限制了用户交互的便捷性和直观性。图形化界面可以为用户提供更直观、更易于操作的方式来输入和查看 PDA 和 CFG，从而提高用户体验。

通过图形化界面，用户可以直接通过拖放元素来构建 CFG 和 PDA，或者通过可视化工具来观察 PDA 的状态转换过程。这样的界面设计可以大大简化用户的操作步骤，降低学习成本，提高用户的使用满意度。

### 6.3 集成更多的自动化功能

我们还可以提供自动化的测试工具，以验证 CFG 和 PDA 的正确性和性能。这些测试工具可以自动生成测试用例，对 CFG 和 PDA 进行全面的

测试，检查它们是否能够正确地处理各种输入。通过自动化测试，用户可以快速发现和修复潜在的问题，提高系统的可靠性和稳定性。

通过集成这些自动化功能，我们可以提供更强大、更高效的工具，以更好地应对复杂的实验任务。

## 7. 实验总结

**卢安来：**在本次实验中，我负责了方案设计、消除单产生式算法的实现。经过本次实验，我深入理解了消除单产生式算法的设计和实现，更深入地理解了上下文无关文法的组成和特性，还进一步锻炼了我使用编程语言进行程序设计的能力。此次实验极大地激发了我对计算理论中形式语言与自动机方面的兴趣，有效地助力了我在该领域的学习和探索。

**王若竹：**在本次实验中，我负责了 PDA 到 CFG 的转化。经过本次实验，我深入理解并实现了 PDA 到 CFG 的转化过程，并通过测试、调试等环节进一步锻炼了我的工程能力。此次实验让我更加熟悉了计算理论中的自动机与文法之间的转化算法，对我今后的学习、研究、开发都有很大的帮助。

**谢牧航：**在本次实验中，我负责了消除  $\epsilon$  产生式算法的实现以及用户界面的设计。经过本次实验，我深入理解了消除  $\epsilon$  产生式的过程，并通过编写代码、测试等环节进一步加深了我对上下文无关文法的理解。此次实验不仅仅加深了我对形式语言与自动机这一学科的理解，还提高了我的算法设计和编程实践能力。

**苏柏闻：**在本次实验中，我负责了消除无用符号算法的实现以及代码审查。经过本次实验，我深入理解了消除无用符号的过程，并通过编写代码、测试等环节进一步加深了我对上下文无关文法的理解。此次实验不仅加深了我对形式语言与自动机这一学科的理解，还提高了我的算法设计和编程实践能力。

## 附录 A：参考文献

- [1]. 杨娟, 石川, 王柏. (2017). 《形式语言与自动机》(第 2 版). 北京邮电大学出版社.
- [2]. Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation. Addison Wesley.
- [3]. CMSC 451 (2008). Automata Theory & Formal Languages PDA to CFG example.