西安交通大學
XI'AN JIAOTONG UNIVERSITY

2025 春编译原理大作业合集

学院 (部、中心)：电子与信息学部

专　　业：计算机科学与技术

班　　级：越杰 2101

学生姓名：段弘毅

学　　号：2213611582

2025 年 06 月

目录

1　大作业一：词法分析器的设计与实现

1.1　原题目

1.1.1　手工模拟过程描述

下面是对字符串 +123.45e-7# 的手工模拟过程：

state	ch	textval	index	lastloc	lastkind	ret/en
0	+	+	0	0	ERR	enter
4	1	+1	1	0	ADD	enter
1	2	+12	2	1	NUM	enter
1	3	+123	3	2	NUM	enter
1	.	+123.	4	3	NUM	enter
9	4	+123.4	5	4	FLO	enter
9	5	+123.45	6	5	FLO	enter
9	e	+123.45e	7	6	FLO	enter
10	-	+123.45e-	8	6	FLO	enter
11	7	+123.45e-7	9	6	FLO	enter
12	#	+123.45e-7#	10	9	FLO	enter
⊥	#	+123.45e-7\0	10	9	FLO	ret

1.1.2　测试用源程序

假定栈快照的起始单元地址为 500。

```
int raw(int x;){
  y = x + 5;
  return y};
void foo(int y;){
  int z;
  void bar(int x; int soo();){
  if(x>3) bar(x/3,soo(),) elsez = soo(x);
  print z};
  bar(y, raw0,)};
foo(6,)
```

Listing 1.1　待分析的 C++ 风格源代码

1.1.3 题目要求

1) 整体的流程：

⟨词法分析⟩ → ⟨设计构建σ-DFA(C) 并给出ψ⟩
(⟨写出 scanner() 程序⟩⟨通过运行测试⟩ + ⟨给出流程图⟩⟨通过手工模拟测试⟩)
⟨产生词法记号串⟩

2) **词法集合 L**: 实现严格限定在以下集合内：$L = \{$ SCO, ID, LBR, RBR, LPA, RPA, INT, VOID, NUM, CMA, IF, ELSE, WHILE, RETURN, ADD, MUL, AND, OR, ROP $\}$

3) 测试用源程序见第二部分。注意：手工模拟仅需对源程序的第一行进行，模拟过程的描述格式参照第一部分。

1.2 题目要求分析与设计

1.2.1 词法集合 L' 的修正与确定

原题目给定的词法集合 L 不足以完全解析测试用例。例如，测试代码中出现了赋值 =、除法 / 以及关键字 PRINT。因此，我们将词法集合 L 扩展为 L'：

- **关键字 (Keywords)**: INT, VOID, IF, ELSE, WHILE, RETURN, AND, OR, PRINT
- **标识符 (Identifier)**: ID
- **数字 (Number)**: NUM (支持整数、浮点数、科学记数法)
- **运算符 (Operators)**: ADD(+), MUL(*), DIV(/), ASSIGN(=), ROP(>, <, >=, <=, ==, !=)
- **界符 (Delimiters)**: LPA((), RPA()), LBR({), RBR(}), SCO(;) , CMA(,)

因此，最终实现的词法集合 $L^{prime} = \{$ SCO, ID, LBR, RBR, LPA, RPA, INT, VOID, NUM, CMA, IF, ELSE, WHILE, RETURN, ADD, MUL, DIV, AND, OR, ROP, ASSIGN, PRINT $\}$。

1.2.2 DFA 设计：状态转换表

为避免图形的复杂性，我们使用状态转换表来描述识别各种词法单元的 DFA。下表是一个简化的核心逻辑 DFA 模型。其中 q0 是初始状态，星号 (*) 表示接受状态。

当前状态	输入字符类别	下一状态	说明/动作
q0 (初始)	letter	q1*	进入标识符/关键字识别
	digit	q2*	进入整数识别
	+ / -	q6	可能为数字符号或加减运算符
	.	q4	可能为浮点数
	< / > / = / !	q8*	进入关系/赋值运算符识别
	/ / * / (/) / { / } / ; / ,	q10*	单字符界符/运算符，直接接受
	whitespace	q0	跳过空白
q1* (ID/KW)	letter / digit	q1*	继续构成标识符
	other	-	*retract*, 接受 ID 或 Keyword
q2* (整数)	digit	q2*	继续构成整数
	.	q3*	转向浮点数
	e / E	q5	转向科学记数法
	other	-	*retract*, 接受 NUM
q3* (浮点)	digit	q3*	继续构成小数部分
	e / E	q5	转向科学记数法
	other	-	*retract*, 接受 NUM
q4 (小数点)	digit	q3*	确认是浮点数
	other	-	*retract*, 错误或其它符号
q5 (科学记数)	+ / -	q7	指数符号
	digit	q7*	指数部分
q6 (+/-)	digit	q2*	确认是带符号整数
	.	q4	确认是带符号浮点数
	other	-	*retract*, 接受 ADD
q7* (指数)	digit	q7*	继续构成指数
	other	-	*retract*, 接受 NUM
q8* (ROP)	= (前缀为<,>,!,=)	q9*	构成 <=, >=, !=, ==
	other	-	*retract*, 接受 <, >, =, !

映射 ψ: 将 DFA 的接受状态映射到词法单元类型。例如：

- 状态 q1 接受时，查关键字表，若找到则映射为相应关键字 (如 INT), 否则为 ID。
- 状态 q2, q3, q7 接受时，映射为 NUM。
- 状态 q8, q9 接受时，映射为 ROP 或 ASSIGN。

1.2.3 scanner() 算法核心流程

`scanner()` 函数的主体是一个循环，该循环持续读取输入字符串的字符直到末尾。其逻辑流程可概括如下：

1) 从当前指针 `i` 获取字符 `ch`。

2) **跳过空白**：如果 `ch` 是空格、制表符或换行符，则指针 `i` 加一，开始下一轮循环。

3) **识别标识符和关键字**：如果 `ch` 是字母，则进入标识符识别模式。

- 向后贪婪地读取所有连续的字母和数字，构成一个词素 `lexeme`。
- 在关键字字典 `KEYWORDS` 中查找 `lexeme`。
- 如果找到，生成对应的关键字词法单元 (如 `INT`)；否则，生成 `ID` 词法单元。

4) **识别数字**：如果 `ch` 是数字，或者 `ch` 是 `.`、`+`、`-` 且后跟数字，则进入数字识别模式。

- 按照 `[sign][digits][.digits][e/E[sign]digits]` 的贪婪模式匹配尽可能长的数字串。
- 逻辑需要处理多种情况，如 `.5`, `5.`, `+5`, `5e-10` 等。
- 如果最终构成的字符串是合法的数字，则生成 `NUM` 词法单元。
- 如果 `+` 或 `-` 之后不是数字，则回溯，将其作为独立的 `ADD` 运算符处理。

5) **识别运算符和界符**：

- 检查当前字符 `ch` 是否为多字符运算符的起始 (如 `<`, `>`, `!`, `=`)。
- 如果是，则向前查看一个字符，判断是否构成 `ROP`(如 `<=`, `!=`, `==`)。
- 否则，将其作为单字符的运算符 (`ASSIGN`, `ADD`, `MUL`, `DIV`) 或界符 (`LPA`, `SCO` 等) 处理。

6) **错误处理**：如果当前字符不属于以上任何一种情况，则生成一个 `ERR` 词法单元，并继续分析下一个字符。

7) 重复步骤 1-6，直到处理完所有输入字符。

8) 返回所有生成的词法单元列表。

1.3 手工模拟过程

根据题目要求，对源程序的第一行 `int raw(int x;){}` 进行手工模拟。模拟过程描述如下：

state	ch	textval	index	lastloc	lastkind	ret/en
0	i	i	0	0	ERR	enter
1	n	in	1	0	ID	enter
1	t	int	2	1	ID	enter
1		int	3	2	INT	enter
⊥		int\0	3	2	INT	**ret**
0	r	r	4	4	ERR	enter
1	a	ra	5	4	ID	enter

state	ch	textval	index	lastloc	lastkind	ret/en
1	w	raw	6	5	ID	enter
1	(	raw(	7	6	ID	enter
⊥	(	raw\0	7	6	ID	**ret**
0	(	(	7	7	ERR	enter
10	i	(i	8	7	LPA	enter
⊥	i	(\0	8	7	LPA	**ret**
0	i	i	8	8	ERR	enter
1	n	in	9	8	ID	enter
1	t	int	10	9	ID	enter
1		int	11	10	INT	enter
⊥		int\0	11	10	INT	**ret**
0	x	x	12	12	ERR	enter
1	;	x;	13	12	ID	enter
⊥	;	x\0	13	12	ID	**ret**
0	;	;	13	13	ERR	enter
10	)	;)	14	13	SCO	enter
⊥	)	;\0	14	13	SCO	**ret**
0	)	)	14	14	ERR	enter
10	{	){	15	14	RPA	enter
⊥	{	)\0	15	14	RPA	**ret**
0	{	{	15	15	ERR	enter
10	\0	{\0	16	15	LBR	enter
⊥	\0	{\0	16	15	LBR	**ret**

表 1-1　对 `int raw(int x;){}` 的手工模拟过程

1.4　实现与测试

1.4.1　scanner() 源程序

```
import re

# 关键字字典，将字符串关键字映射到它们的词法单元类型
```

```
KEYWORDS = {
    'int': 'INT', 'if': 'IF', 'else': 'ELSE', 'while': 'WHILE',
    'return': 'RETURN', 'void': 'VOID', 'and': 'AND', 'or': 'OR',
    'print': 'PRINT'  # 基于测试用例扩展
}

# 单字符词法单元类型
SINGLE_CHAR_TOKENS = {
    '+': 'ADD', '*': 'MUL', '(': 'LPA', ')': 'RPA',
    '{': 'LBR', '}': 'RBR', ';': 'SCO', ',': 'CMA',
    '/': 'DIV'  # 基于测试用例扩展
}

def is_letter(ch):
    return ch.isalpha()

def is_digit(ch):
    return ch.isdigit()

def is_alnum(ch):
    return ch.isalnum()

def scanner(input_line):
    tokens = []  # 存储生成的词法单元
    i = 0        # 当前输入字符串的索引
    n = len(input_line) # 输入字符串的长度

    while i < n:
        ch = input_line[i] # 当前字符

        # 1. 跳过空白字符
        if ch.isspace():
            i += 1
            continue

        # 2. 标识符 (ID) 和关键字
        if is_letter(ch):
            start = i
            i += 1
            while i < n and is_alnum(input_line[i]):
                i += 1
            lexeme = input_line[start:i]
            kind = KEYWORDS.get(lexeme, 'ID')
            tokens.append((kind, lexeme))
            continue

        # 3. 数字 (NUM - 处理整数、浮点数和科学计数法)
        if is_digit(ch) or \
           (ch == '.' and i + 1 < n and is_digit(input_line[i+1])) or \
```

```
        (ch in '+-' and i + 1 < n and (
        is_digit(input_line[i+1]) or (input_line[i+1] == '.' and i + 2 < n
                                      and is_digit(input_line[i+2])))):

         num_start_idx = i
         # 可选的符号
         if input_line[i] in '+-':
             i += 1

         # 整数部分
         has_digits_before_dot = False
         start_digits = i
         while i < n and is_digit(input_line[i]):
             i += 1
         if i > start_digits:
             has_digits_before_dot = True

         # 小数部分
         has_dot = False
         has_digits_after_dot = False
         if i < n and input_line[i] == '.':
             has_dot = True
             i += 1 # 消耗 '.'
             start_frac_digits = i
             while i < n and is_digit(input_line[i]):
                 i += 1
             if i > start_frac_digits:
                 has_digits_after_dot = True

         # 指数部分
         if (has_digits_before_dot or has_digits_after_dot) and i < n and
input_line[i] in 'eE':
             temp_exp_i = i + 1
             if temp_exp_i < n and input_line[temp_exp_i] in '+-':
                 temp_exp_i += 1

             start_exp_digits = temp_exp_i
             while temp_exp_i < n and is_digit(input_line[temp_exp_i]):
                 temp_exp_i += 1

             if temp_exp_i > start_exp_digits:
                 i = temp_exp_i

         lexeme = input_line[num_start_idx:i]
         if not any(c.isdigit() for c in lexeme):
             i = num_start_idx # 回溯
         else:
             tokens.append(('NUM', lexeme))
             continue
```

```
101
102         # 重新获取当前字符，以防数字解析回溯
103         ch = input_line[i]
104
105         # 4. 操作符 (ROP, 赋值)
106         if ch == '<':
107             if i + 1 < n and input_line[i+1] == '=':
108                 tokens.append(('ROP', '<='))
109                 i += 2
110             else:
111                 tokens.append(('ROP', '<'))
112                 i += 1
113             continue
114         elif ch == '>':
115             if i + 1 < n and input_line[i+1] == '=':
116                 tokens.append(('ROP', '>='))
117                 i += 2
118             else:
119                 tokens.append(('ROP', '>'))
120                 i += 1
121             continue
122         elif ch == '=':
123             if i + 1 < n and input_line[i+1] == '=':
124                 tokens.append(('ROP', '=='))
125                 i += 2
126             else:
127                 tokens.append(('ASSIGN', '=')) # 赋值操作符
128                 i += 1
129             continue
130         elif ch == '!':
131             if i + 1 < n and input_line[i+1] == '=':
132                 tokens.append(('ROP', '!='))
133                 i += 2
134             else:
135                 tokens.append(('ERR', ch))
136                 i += 1
137             continue
138
139         # 5. 剩余的单字符词法单元
140         if ch in SINGLE_CHAR_TOKENS:
141             tokens.append((SINGLE_CHAR_TOKENS[ch], ch))
142             i += 1
143             continue
144
145         # 6. 未识别的字符
146         tokens.append(('ERR', ch))
147         i += 1
148
149     return tokens
```

```
150
151 if __name__ == '__main__':
152     test_code = """
153 int raw(int x;){
154   y = x + 5;
155   return y;
156 };
157 void foo(int y;){
158   int z;
159   void bar(int x; int soo();){
160   if(x>3) bar(x/3,soo(),) else z = soo(x);
161   print z;
162   };
163   bar(y, raw(),);
164 };
165 foo(6,);
166 """
167     # 注意：为了让测试用例能够运行，修正了原始题目中一些明显的录入错误
168     # 如 intx -> int x; sooQ -> soo(); rawO -> raw(); sooO -> soo()
169     # 如果严格按照原始错误的文本，输出会略有不同，但这里的实现能够处理修正后的正确
    代码
170     corrected_test_code = """
171 int raw(int x;){
172     y = x + 5;
173     return y;
174 };
175 void foo(int y;){
176     int z;
177     void bar(int x; int soo();){
178         if(x>3) bar(x/3,soo(),) else z = soo(x);
179         print z;
180     };
181     bar(y, raw(),);
182 };
183 foo(6,);
184 """
185     result = scanner(corrected_test_code)
186     print("词法分析输出：")
187     for token in result:
188         print(token)
```

Listing 1.2　词法分析器 Python 实现

1.4.2　运行测试结果

```
1 词法分析输出：
2 词法分析输出：
3 ('INT', 'int')
4 ('ID', 'raw')
```

```
('LPA', '(')
('INT', 'int')
('ID', 'x')
('SCO', ';')
('RPA', ')')
('LBR', '{')
('ID', 'y')
('ASSIGN', '=')
('ID', 'x')
('ADD', '+')
('NUM', '5')
('SCO', ';')
('RETURN', 'return')
('ID', 'y')
('SCO', ';')
('RBR', '}')
('SCO', ';')
('VOID', 'void')
('ID', 'foo')
('LPA', '(')
('INT', 'int')
('ID', 'y')
('SCO', ';')
('RPA', ')')
('LBR', '{')
('INT', 'int')
('ID', 'z')
('SCO', ';')
('VOID', 'void')
('ID', 'bar')
('LPA', '(')
('INT', 'int')
('ID', 'x')
('SCO', ';')
('INT', 'int')
('ID', 'soo')
('LPA', '(')
('RPA', ')')
('SCO', ';')
('RPA', ')')
('LBR', '{')
('IF', 'if')
('LPA', '(')
('ID', 'x')
('ROP', '>')
('NUM', '3')
('RPA', ')')
('ID', 'bar')
('LPA', '(')
```

```
('ID', 'x')
('DIV', '/')
('NUM', '3')
('CMA', ',')
('ID', 'soo')
('LPA', '(')
('RPA', ')')
('CMA', ',')
('RPA', ')')
('ELSE', 'else')
('ID', 'z')
('ASSIGN', '=')
('ID', 'soo')
('LPA', '(')
('ID', 'x')
('RPA', ')')
('SCO', ';')
('PRINT', 'print')
('ID', 'z')
('SCO', ';')
('RBR', '}')
('SCO', ';')
('ID', 'bar')
('LPA', '(')
('ID', 'y')
('CMA', ',')
('ID', 'raw')
('LPA', '(')
('RPA', ')')
('CMA', ',')
('RPA', ')')
('SCO', ';')
('RBR', '}')
('SCO', ';')
('ID', 'foo')
('LPA', '(')
('NUM', '6')
('CMA', ',')
('RPA', ')')
('SCO', ';')
```

Listing 1.3　在修正后的测试代码上的运行输出

2 大作业二：FIRST 集、FOLLOW 集计算与 LL(1) 文法分析

2.1 题目文法

对下列文法分别计算每个变元的 FIRST 集和 FOLLOW 集，然后从该文法中找出不满足 LL(1) 文法条件的各个原因。给定的文法 G 为：

1) $P \to \check{D}\check{S}$
2) $\check{D} \to \varepsilon \mid \check{D}D;$
3) $D \to Td \mid Td[i] \mid Td(\check{A})\{\check{D}\check{S}\}$
4) $T \to \text{int} \mid \text{void}$
5) $\check{A} \to \varepsilon \mid \check{A}A;$
6) $A \to Td \mid d[] \mid Td()$
7) $\check{S} \to S \mid \check{S};S$
8) $S \to d = E \mid \text{if}(B)S \mid \text{if}(B)S\text{else}S \mid \text{while}(B)S \mid \text{return}E \mid \{\check{S}\} \mid d(\check{R})$
9) $B \to B \wedge B \mid B \vee B \mid ErE \mid E$
10) $E \to d = E \mid i \mid d \mid d(\check{R}) \mid E + E \mid E * E \mid (E)$
11) $\check{R} \to \varepsilon \mid \check{R}R,$
12) $R \to E \mid d[] \mid d()$

其中，终结符集合为 $\{;, d, [,], (,), \{, \}, \text{int}, \text{void}, =, \text{if}, \text{else}, \text{while}, \text{return}, \wedge, \vee, r, i, +, *, \varepsilon\}$，非终结符集合为 $\{P, \check{D}, D, T, \check{A}, A, \check{S}, S, B, E, \check{R}, R\}$。

2.2 FIRST 集计算

FIRST(X) 是可以从 X 推导出的串的第一个终结符的集合。如果 $X \Rightarrow^* \varepsilon$，则 $\varepsilon \in$ FIRST(X)。

- FIRST$(T) = \{\text{int}, \text{void}\}$
- FIRST$(D) =$ FIRST$(T) = \{\text{int}, \text{void}\}$
- FIRST$(\check{D}) =$ FIRST$(D) \cup \{\varepsilon\} = \{\text{int}, \text{void}, \varepsilon\}$
- FIRST$(A) =$ FIRST$(T) \cup \{d\} = \{\text{int}, \text{void}, d\}$
- FIRST$(\check{A}) =$ FIRST$(A) \cup \{\varepsilon\} = \{\text{int}, \text{void}, d, \varepsilon\}$
- FIRST$(E) = \{d, i, (\}$
- FIRST$(R) =$ FIRST$(E) \cup \{d\} = \{d, i, (\}$
- FIRST$(\check{R}) =$ FIRST$(R) \cup \{\varepsilon\} = \{d, i, (, \varepsilon\}$
- FIRST$(S) = \{d, \text{if}, \text{while}, \text{return}, \{\}$
- FIRST$(\check{S}) =$ FIRST$(S) = \{d, \text{if}, \text{while}, \text{return}, \{\}$ (因为 $\check{S} \to S \mid \check{S};S$ 不能直接推导出 ε)

- FIRST(B) = FIRST(E) = $\{d, i, (\}$
- FIRST(P) = (FIRST$(\check{D})$ − $\{\varepsilon\}$) ∪ (if $\varepsilon \in$ FIRST$(\check{D})$ then FIRST$(\check{S})$ else $\emptyset$) = $\{$int, void$\}$ ∪ FIRST$(\check{S})$ = $\{$int, void, d, if, while, return, $\{\}$

最终 FIRST 集总结:

- FIRST(P) = $\{$int, void, d, if, while, return, $\{\}$
- FIRST$(\check{D})$ = $\{$int, void, $\varepsilon\}$
- FIRST(D) = $\{$int, void$\}$
- FIRST(T) = $\{$int, void$\}$
- FIRST$(\check{A})$ = $\{$int, void, $d, \varepsilon\}$
- FIRST(A) = $\{$int, void, $d\}$
- FIRST$(\check{S})$ = $\{d$, if, while, return, $\{\}$
- FIRST(S) = $\{d$, if, while, return, $\{\}$
- FIRST(B) = $\{d, i, (\}$
- FIRST(E) = $\{d, i, (\}$
- FIRST$(\check{R})$ = $\{d, i, (, \varepsilon\}$
- FIRST(R) = $\{d, i, (\}$

2.3 FOLLOW 集计算

FOLLOW(X) 是在某个句型中可以紧跟在 X 后面的终结符的集合。如果 X 可以是某个句型的最右符号，则句子结束符 \$ 在 FOLLOW(X) 中。

1) 置 \$ 于 FOLLOW(P)。

2) 若有产生式 $A \to \alpha B \beta$，则 FIRST$(\beta) - \{\varepsilon\}$ 加入 FOLLOW(B)。若 $\varepsilon \in$ FIRST(β) (或 $\beta = \varepsilon$)，则 FOLLOW(A) 加入 FOLLOW(B)。

3) 若有产生式 $A \to \alpha B$，则 FOLLOW(A) 加入 FOLLOW(B)。

经过迭代计算：

- FOLLOW(P) = $\{\$\}$
- FOLLOW$(\check{D})$:
  - 来自 $P \to \check{D}\underline{\check{S}}$: FIRST$(\check{S}) - \{\varepsilon\} = \{d$, if, while, return, $\{\} \subseteq$ FOLLOW$(\check{D})$
  - 来自 $\check{D} \to \check{D}\underline{D;}$: FIRST$(D)$ = $\{$int, void$\} \subseteq$ FOLLOW$(\check{D})$ (考虑 $\check{D}$ 的左递归展开)
  - 来自 $D \to Td(\check{A})\{\underline{\check{D}\check{S}}\}$: FIRST$(\check{S}) - \{\varepsilon\} \subseteq$ FOLLOW$(\check{D})$

  所以 FOLLOW$(\check{D})$ = $\{$int, void, d, if, while, return, $\{\}$

- FOLLOW(D): 来自 $\check{D} \to \check{D}D\underline{;}$: $\{;\} \subseteq$ FOLLOW(D) 所以 FOLLOW(D) $= \{;\}$
- FOLLOW(T): 来自 $D \to \underline{T}d...$ 和 $A \to \underline{T}d...$: $\{d\} \subseteq$ FOLLOW(T) 所以 FOLLOW(T) $= \{d\}$
- FOLLOW($\check{A}$):
  - 来自 $D \to Td(\underline{\check{A}})\{\check{D}\check{S}\}$: FIRST()$\{\check{D}\check{S}\}) = \{)\} \subseteq$ FOLLOW($\check{A}$)
  - 来自 $\check{A} \to \check{A}\underline{A;}$: FIRST($A$) $= \{$int, void, $d\} \subseteq$ FOLLOW($\check{A}$)

  所以 FOLLOW($\check{A}$) $= \{$int, void, $d,)\}$
- FOLLOW(A): 来自 $\check{A} \to \check{A}A\underline{;}$: $\{;\} \subseteq$ FOLLOW(A) 所以 FOLLOW(A) $= \{;\}$
- FOLLOW($\check{S}$):
  - 来自 $P \to \check{D}\check{S}$: FOLLOW(P) $= \{\$\} \subseteq$ FOLLOW($\check{S}$)
  - 来自 $D \to Td(\check{A})\{\check{D}\check{S}\underline{\}}$: $\{\}\} \subseteq$ FOLLOW($\check{S}$)
  - 来自 $\check{S} \to \check{S}\underline{;}S$: $\{;\} \subseteq$ FOLLOW($\check{S}$)
  - 来自 $S \to \{\check{S}\underline{\}}$: $\{\}\} \subseteq$ FOLLOW($\check{S}$)

  所以 FOLLOW($\check{S}$) $= \{\$, ;, \}\}$
- FOLLOW(S):
  - 来自 $\check{S} \to \underline{S}$ 和 $\check{S} \to \check{S};\underline{S}$: FOLLOW($\check{S}$) $\subseteq$ FOLLOW(S)
  - 来自 $S \to$ if(B)$\underline{S}$else$S$: $\{$else$\} \subseteq$ FOLLOW(S)

  所以 FOLLOW(S) $=$ FOLLOW($\check{S}$) $\cup \{$else$\} = \{\$, ;, \}$, else$\}$
- FOLLOW(B):
  - 来自 $S \to$ if($\underline{B}$)S...: $\{)\} \subseteq$ FOLLOW(B)
  - 来自 $B \to B\underline{\wedge}B$: $\{\wedge\} \subseteq$ FOLLOW(B)
  - 来自 $B \to B\underline{\vee}B$: $\{\vee\} \subseteq$ FOLLOW(B)

  所以 FOLLOW(B) $= \{), \wedge, \vee\}$
- FOLLOW(R): 来自 $\check{R} \to \check{R}R\underline{,}$: $\{,\} \subseteq$ FOLLOW(R) 所以 FOLLOW(R) $= \{,\}$
- FOLLOW(E):
  - 来自 $S \to d = E, S \to$ return$E$: FOLLOW(S) $\subseteq$ FOLLOW(E)
  - 来自 $B \to \underline{E}rE, B \to E$: FOLLOW($B$) $\subseteq$ FOLLOW(E) (当 E 为 B 的最后一个符号时) 和 $\{r\} \subseteq$ FOLLOW(E)
  - 来自 $E \to \underline{E} + E, E \to \underline{E} * E$: $\{+, *\} \subseteq$ FOLLOW(E)
  - 来自 $E \to (E\underline{)}$: $\{)\} \subseteq$ FOLLOW(E)
  - 来自 $R \to E$: FOLLOW(R) $\subseteq$ FOLLOW(E)

  所以FOLLOW(E) $=$ FOLLOW(S)$\cup$FOLLOW(B)$\cup$FOLLOW(R)$\cup\{r, +, *,)\} = \{\$, ;, \}$, else,), $\wedge, \vee, ,, r, +, *\}$

- FOLLOW($\check{R}$):
  - 来自 $S \to d(\underline{\check{R}})\ldots:\{)\} \subseteq$ FOLLOW($\check{R}$)
  - 来自 $E \to d(\underline{\check{R}})\ldots:\{)\} \subseteq$ FOLLOW($\check{R}$)
  - 来自 $\check{R} \to \check{R}\underline{R},$: FIRST($R$) $= \{d, i, (\} \subseteq$ FOLLOW($\check{R}$)

  所以 FOLLOW($\check{R}$) $= \{d, i, (,)\}$

最终 FOLLOW 集总结:

- FOLLOW(P) $= \{\$\}$
- FOLLOW($\check{D}$) $= \{$int, void, d, if, while, return, $\{\}$
- FOLLOW(D) $= \{;\}$
- FOLLOW(T) $= \{d\}$
- FOLLOW($\check{A}$) $= \{$int, void, $d,)\}$
- FOLLOW(A) $= \{;\}$
- FOLLOW($\check{S}$) $= \{\$, ;, \}\}$
- FOLLOW(S) $= \{\$, ;, \}$, else$\}$
- FOLLOW(B) $= \{), \wedge, \vee\}$
- FOLLOW(E) $= \{\$, ;, \}$, else, $), \wedge, \vee, , , r, +, *\}$
- FOLLOW($\check{R}$) $= \{d, i, (,)\}$
- FOLLOW(R) $= \{, \}$

2.4 不满足 LL(1) 文法条件的原因

一个文法是 LL(1) 文法，需满足以下条件：对于每个非终结符 X 的任意两个不同产生式 $X \to \alpha$ 和 $X \to \beta$：

1) FIRST(α) $\cap$ FIRST(β) $= \emptyset$。

2) 如果 $\varepsilon \in$ FIRST(α)，那么 FIRST(β) $\cap$ FOLLOW(X) $= \emptyset$ (反之亦然，如果 $\varepsilon \in$ FIRST(β)，那么 FIRST(α) $\cap$ FOLLOW(X) $= \emptyset$)。

此外，LL(1) 文法不能包含左递归或公共左因子 (若有公共左因子，则条件 1 不满足)。

2.4.1 $\check{D} \to \varepsilon \mid \check{D}D;$

- **原因 1: 左递归**。产生式 $\check{D} \to \check{D}D;$ 是直接左递归。LL(1) 文法不能有左递归。
- **原因 2: ε-产生式的冲突**。即使通过标准方法消除左递归得到 $\check{D} \to D;\check{D}' \mid \varepsilon$ 和 $\check{D}' \to D;\check{D}' \mid \varepsilon$ (这里假设原始意图是 $\check{D} \to D;\check{D} \mid \varepsilon$)：令 $\alpha_1 = D;\check{D}$ (或 $D;$)，$\alpha_2 = \varepsilon$。FIRST(α_1) $=$ FIRST(D) $=$ {int, void}。FIRST(α_2) $= \{\varepsilon\}$。根据条件 2，需要检查 FIRST(α_1)$\cap$FOLLOW($\check{D}$) $= \emptyset$。FOLLOW($\check{D}$) $=$ {int, void, d, if, while, return, $\{\}$。交集为 {int, void}$\cap$ {int, void, d, if, while, return, $\{\} =$ {int, void} $\neq \emptyset$。这是一个 FIRST/FOLLOW 冲突。

2.4.2 $D \to Td \mid Td[i] \mid Td(\check{A})\{\check{D}\check{S}\}$

- **原因: 公共左因子**。FIRST(Td) = {int, void} FIRST($Td[i]$) = {int, void} FIRST($Td(\check{A})\{\check{D}\check{S}\}$) = {int, void} 这三个产生式有共同的前缀 Td。违反了 LL(1) 条件 1(提取公共左因子 Td 后，新的非终结符的各产生式的 FIRST() 集需要互不相交，或与 ε 相关的产生式满足条件 2)。

2.4.3 $T \to \text{int} \mid \text{void}$

- FIRST(int) = {int}
- FIRST(void) = {void}
- FIRST(int) ∩ FIRST(void) = ∅。此非终结符的产生式满足 LL(1) 条件。

2.4.4 $\check{A} \to \varepsilon \mid \check{A}A;$

- **原因 1: 左递归**。产生式 $\check{A} \to \check{A}A;$ 是直接左递归。
- **原因 2: ε-产生式的冲突**。消除左递归为 $\check{A} \to A;\check{A} \mid \varepsilon$: 令 $\alpha_1 = A;\check{A}$ (或 $A;$)，$\alpha_2 = \varepsilon$。FIRST(α_1) = FIRST(A) = {int, void, d}。FIRST(α_2) = {ε}。需要检查 FIRST(α_1)∩FOLLOW($\check{A}$) = ∅。FOLLOW($\check{A}$) = {int, void, d,)}。交集为 {int, void, d} ∩ {int, void, d,)} = {int, void, d} ≠ ∅。这是一个 FIRST/FOLLOW 冲突。

2.4.5 $A \to Td \mid d[] \mid Td()$

- **原因: FIRST/FIRST 冲突和公共左因子**。FIRST(Td) = {int, void}。FIRST($d[]$) = {d}。FIRST($Td()$) = {int, void}。产生式 $A \to Td$ 和 $A \to Td()$ 的 FIRST() 集均为 {int, void}，存在 FIRST/FIRST 冲突 (违反条件 1)。它们有共同前缀 Td。

2.4.6 $\check{S} \to S \mid \check{S};S$

- **原因 1: 左递归**。产生式 $\check{S} \to \check{S};S$ 是直接左递归。
- **原因 2: FIRST/FIRST 或 FIRST/FOLLOW 冲突 (在左递归消除后)**。如果消除左递归为 $\check{S} \to S\check{S}'$ 和 $\check{S}' \to ;S\check{S}' \mid \varepsilon$: 对于 $\check{S}' \to ;S\check{S}'(\alpha_1) \mid \varepsilon(\alpha_2)$: FIRST($\alpha_1$) = {;}。FIRST($\alpha_2$) = {$\varepsilon$}。FOLLOW($\check{S}'$) = FOLLOW($\check{S}$) = {\$, ;, }}。因为 $\varepsilon \in$ FIRST(α_2), 我们检查 FIRST(α_1)∩FOLLOW($\check{S}'$) = {;} ∩ {\$, ;, }} = {;} ≠ ∅。这是一个 FIRST/FOLLOW 冲突。

2.4.7 $S \to d = E \mid \text{if}(B)S \mid \text{if}(B)S\text{else}S \mid \text{while}(B)S \mid \text{return}E \mid \{\check{S}\} \mid d(\check{R})$

- **原因 1: 公共左因子/FIRST/FIRST 冲突 (违反条件 1)**。
  - 对于产生式 $S \to \underline{d} = E$ 和 $S \to \underline{d}(\check{R})$: FIRST($d = E$) = {$d$}。FIRST($d(\check{R})$) = {$d$}。FIRST($d = E$) ∩ FIRST($d(\check{R})$) = {$d$} ≠ ∅。它们有共同前缀 d。
  - 对于产生式 $S \to \underline{\text{if}}(B)S$ 和 $S \to \underline{\text{if}}(B)S\text{else}S$:FIRST(if($B$)$S$) = {if}。FIRST(if($B$)$S$else$S$) = {if}。FIRST(if($B$)$S$) ∩ FIRST(if($B$)$S$else$S$) = {if} ≠ ∅。它们有共同前缀 if($B$)$S$ (经典的"dangling else" 问题)。

2.4.8 $B \to B \wedge B \mid B \vee B \mid ErE \mid E$

- **原因 1: 左递归**。产生式 $B \to B \wedge B$ 和 $B \to B \vee B$ 是直接左递归。
- **原因 2: FIRST/FIRST 冲突 (违反条件 1)**。即使处理了左递归 (例如将文法改写为 $B \to EB'$ 等形式)，原始的 $B \to ErE$ 和 $B \to E$ 存在冲突：FIRST(ErE) = FIRST(E) = $\{d, i, (\}$。FIRST$(E) = \{d, i, (\}$。FIRST$(ErE) \cap$ FIRST$(E) = \{d, i, (\} \neq \emptyset$。

2.4.9 $E \to d = E \mid i \mid d \mid d(\check{R}) \mid E + E \mid E * E \mid (E)$

- **原因 1: 左递归**。产生式 $E \to E + E$ 和 $E \to E * E$ 是直接左递归。
- **原因 2: 公共左因子/FIRST/FIRST 冲突 (违反条件 1)**。FIRST$(d = E) = \{d\}$。FIRST$(d) = \{d\}$。FIRST$(d(\check{R})) = \{d\}$。这三个产生式 $(E \to d = E, E \to d, E \to d(\check{R}))$ 的 FIRST() 集都包含 $\{d\}$，存在两两交集非空的情况，例如 FIRST$(d = E) \cap$ FIRST$(d) = \{d\} \neq \emptyset$。它们都有共同前缀 d。

2.4.10 $\check{R} \to \varepsilon \mid \check{R}R,$

- **原因 1: 左递归**。产生式 $\check{R} \to \check{R}R,$ 是直接左递归。
- **原因 2: ε-产生式的冲突**。消除左递归为 $\check{R} \to R, \check{R} \mid \varepsilon$：令 $\alpha_1 = R, \check{R}$ (或 $R,$)，$\alpha_2 = \varepsilon$。FIRST(α_1) = FIRST$(R) = \{d, i, (\}$。FIRST$(\alpha_2) = \{\varepsilon\}$。需要检查 FIRST$(\alpha_1) \cap$ FOLLOW$(\check{R}) = \emptyset$。FOLLOW$(\check{R}) = \{d, i, (,)\}$。交集为 $\{d, i, (\} \cap \{d, i, (,)\} = \{d, i, (\} \neq \emptyset$。这是一个 FIRST/-FOLLOW 冲突。

2.4.11 $R \to E \mid d[] \mid d()$

- **原因: FIRST/FIRST 冲突 (违反条件 1)**。FIRST$(E) = \{d, i, (\}$。FIRST$(d[]) = \{d\}$。FIRST$(d()) = \{d\}$。由于 FIRST(E) 包含 d，所以：FIRST$(E) \cap$ FIRST$(d[]) = \{d\} \neq \emptyset$。FIRST$(E) \cap$ FIRST$(d()) = \{d\} \neq \emptyset$。(另外 FIRST$(d[]) \cap$ FIRST$(d()) = \{d\} \neq \emptyset$，但它们在 d 之后可以通过 [和 (区分，主要的冲突在于与 $R \to E$ 的选择。)

2.5 总结

该文法由于存在大量的左递归、公共左因子以及与 ε 产生式相关的 FIRST/FOLLOW 冲突，因此不满足 LL(1) 文法的条件。几乎每个非终结符的产生式规则都存在至少一个不满足 LL(1) 条件的问题。

2.6 附：Python 实现

2.6.1 源程序：LL(1) 冲突检测工具

```
from collections import defaultdict

```

```python
# 定义文法中的特殊符号
EPSILON = " "

class LL1_Analyzer:
    def __init__(self, grammar_str, start_symbol):
        self.terminals = set()
        self.non_terminals = set()
        self.grammar = defaultdict(list)
        self.start_symbol = start_symbol
        self.first_sets = defaultdict(set)
        self.follow_sets = defaultdict(set)

        self._parse_grammar(grammar_str)
        self._find_terminals_and_non_terminals()

    def _parse_grammar(self, grammar_str):
        """解析字符串格式的文法"""
        for line in grammar_str.strip().split('\n'):
            head, body = line.split('->')
            head = head.strip()
            # 区分不同的产生式
            productions = [p.strip().split() for p in body.split('|')]
            for prod in productions:
                # 处理空串
                if prod == [EPSILON]:
                    self.grammar[head].append([EPSILON])
                else:
                    self.grammar[head].append(prod)

    def _find_terminals_and_non_terminals(self):
        """从文法中自动识别终结符和非终结符"""
        self.non_terminals.update(self.grammar.keys())
        for head in self.grammar:
            for production in self.grammar[head]:
                for symbol in production:
                    if symbol not in self.non_terminals and symbol != EPSILON:
                        self.terminals.add(symbol)

    def compute_first_sets(self):
        """迭代计算所有非终结符的FIRST集"""
        for nt in self.non_terminals:
            self.first_sets[nt] = set()

        changed = True
        while changed:
            changed = False
            for head in self.grammar:
                for production in self.grammar[head]:
                    original_size = len(self.first_sets[head])
```

```
                    # 计算产生式右部的FIRST集
                    rhs_first = self._calculate_first_of_sequence(production)
                    self.first_sets[head].update(rhs_first)
                    if len(self.first_sets[head]) > original_size:
                        changed = True
        return self.first_sets

    def _calculate_first_of_sequence(self, sequence):
        """计算一个符号序列的FIRST集"""
        if not sequence:
            return {EPSILON}

        first = sequence[0]
        if first == EPSILON:
            return {EPSILON}
        if first in self.terminals:
            return {first}

        # 如果是-非终结符
        result = set()
        can_be_epsilon = True
        for symbol in sequence:
            if symbol in self.terminals:
                result.add(symbol)
                can_be_epsilon = False
                break

            # 是非终结符
            first_of_symbol = self.first_sets[symbol].copy()
            if EPSILON in first_of_symbol:
                result.update(first_of_symbol - {EPSILON})
            else:
                result.update(first_of_symbol)
                can_be_epsilon = False
                break

        if can_be_epsilon:
            result.add(EPSILON)

        return result


    def compute_follow_sets(self):
        """迭代计算所有非终结符的FOLLOW集"""
        if not self.first_sets:
            self.compute_first_sets()

        for nt in self.non_terminals:
            self.follow_sets[nt] = set()
```

```
101
102         self.follow_sets[self.start_symbol].add('$') # 规则1：将$加入开始符号的
    FOLLOW集
103
104         changed = True
105         while changed:
106             changed = False
107             for head in self.grammar:
108                 for production in self.grammar[head]:
109                     # 规则3: A ->  B, FOLLOW(A) is in FOLLOW(B)
110                     trailer = self.follow_sets[head].copy()
111                     for i in range(len(production) - 1, -1, -1):
112                         symbol = production[i]
113                         if symbol in self.non_terminals:
114                             original_size = len(self.follow_sets[symbol])
115                             self.follow_sets[symbol].update(trailer)
116                             if len(self.follow_sets[symbol]) > original_size:
117                                 changed = True
118
119                             # 更新 trailer
120                             if EPSILON in self.first_sets[symbol]:
121                                 trailer.update(self.first_sets[symbol] - {EPSILON
    })
122                             else:
123                                 trailer = self.first_sets[symbol].copy()
124                         else: # 是终结符
125                             trailer = {symbol}
126         return self.follow_sets
127
128     def analyze_ll1_conflicts(self):
129         """分析LL(1)冲突"""
130         print("--- LL(1) Conflict Analysis ---")
131         if not self.follow_sets:
132             self.compute_follow_sets()
133
134         for head, productions in self.grammar.items():
135             # 1. 检查左递归
136             for prod in productions:
137                 if prod and prod[0] == head:
138                     print(f"  Conflict for '{head}': Direct left recursion in
    production {head} -> {' '.join(prod)}")
139
140             # 2. 检查 FIRST/FIRST 冲突
141             for i in range(len(productions)):
142                 for j in range(i + 1, len(productions)):
143                     prod1 = productions[i]
144                     prod2 = productions[j]
145                     first1 = self._calculate_first_of_sequence(prod1)
146                     first2 = self._calculate_first_of_sequence(prod2)
```

```
147                     intersection = first1.intersection(first2)
148                     if intersection:
149                         print(f"  Conflict for '{head}': FIRST/FIRST conflict
    between {' '.join(prod1)} and {' '.join(prod2)}. Intersection: {intersection}")
150 
151             # 3. 检查 FIRST/FOLLOW 冲突
152             has_epsilon_prod = any(prod == [EPSILON] for prod in productions)
153             if has_epsilon_prod:
154                 follow_of_head = self.follow_sets[head]
155 
156                 for prod in productions:
157                     if prod != [EPSILON]:
158                         first_of_prod = self._calculate_first_of_sequence(prod)
159                         intersection = follow_of_head.intersection(first_of_prod)
160                         if intersection:
161                             print(f"  Conflict for '{head}': FIRST/FOLLOW
    conflict. Production: {head} -> {' '.join(prod)} has FIRST set {first_of_prod},
     which intersects with FOLLOW({head}) {follow_of_head}. Intersection: {
    intersection}")
162 
163 
164 if __name__ == '__main__':
165     # 注意：为了简化解析，我们将带钩的字母替换为大写字母，例如 D-check -> D_check
166     # 文法中有些不规范的表达，如 D -> Td[i]，这里将 [i] 视为一个整体 'd[i]'
167     # 作业中的文法需要稍微整理成标准格式：A -> B C | D
168     grammar_text = """
169     P -> Ď Š
170     Ď ->   | Ď D ;
171     D -> T d | T d[i] | T d ( Ă ) { Ď Š }
172     T -> int | void
173     Ă ->   | Ă A ;
174     A -> T d | d[] | T d ( )
175     Š -> S | Š ; S
176     S -> d = E | if ( B ) S | if ( B ) S else S | while ( B ) S | return E | { Š }
     | d ( Ř )
177     B -> B   B | B   B | E r E | E
178     E -> d = E | i | d | d ( Ř ) | E + E | E * E | ( E )
179     Ř ->   | Ř R ,
180     R -> E | d[] | d ( )
181     """
182 
183     # 替换特殊字符以方便处理
184     grammar_text = grammar_text.replace('check', '').replace('č', 'c').replace('ě'
    , 'e')
185     grammar_text = grammar_text.replace('Ď', 'D_check').replace('Š', 'S_check').
    replace('Ă', 'A_check').replace('Ř', 'R_check')
186 
187 
188     analyzer = LL1_Analyzer(grammar_text, start_symbol='P')
```

```
189
190     print("--- Terminals ---")
191     print(sorted(list(analyzer.terminals)))
192     print("\n--- Non-Terminals ---")
193     print(sorted(list(analyzer.non_terminals)))
194
195     print("\n--- FIRST Sets ---")
196     first = analyzer.compute_first_sets()
197     for nt in sorted(first.keys()):
198         print(f"FIRST({nt}) = {sorted(list(first[nt]))}")
199
200     print("\n--- FOLLOW Sets ---")
201     follow = analyzer.compute_follow_sets()
202     for nt in sorted(follow.keys()):
203         print(f"FOLLOW({nt}) = {sorted(list(follow[nt]))}")
204
205     print("\n")
206     analyzer.analyze_ll1_conflicts()
```

Listing 2.1　FIRST/FOLLOW 集与 LL(1) 冲突检测工具

2.6.2　程序输出

```
--- Terminals ---
['(', ')', '*', '+', ',', ';', '=', 'd', 'd[]', 'd[i]', 'else', 'i', 'if', 'int',
    'r', 'return', 'void', 'while', '{', '}', ' ', ' ']

--- Non-Terminals ---
['A', 'A_check', 'B', 'D', 'D_check', 'E', 'P', 'R', 'R_check', 'S', 'S_check', 'T
    ']

--- FIRST Sets ---
FIRST(A) = ['d[]', 'int', 'void']
FIRST(A_check) = ['d[]', 'int', 'void', ' ']
FIRST(B) = ['(', 'd', 'i']
FIRST(D) = ['int', 'void']
FIRST(D_check) = ['int', 'void', ' ']
FIRST(E) = ['(', 'd', 'i']
FIRST(P) = ['d', 'if', 'int', 'return', 'void', 'while', '{']
FIRST(R) = ['(', 'd', 'd[]', 'i']
FIRST(R_check) = ['(', 'd', 'd[]', 'i', ' ']
FIRST(S) = ['d', 'if', 'return', 'while', '{']
FIRST(S_check) = ['d', 'if', 'return', 'while', '{']
FIRST(T) = ['int', 'void']

--- FOLLOW Sets ---
FOLLOW(A) = [';']
FOLLOW(A_check) = [')', 'd[]', 'int', 'void']
FOLLOW(B) = [')', ' ', ' ']
```

```
FOLLOW(D) = [';']
FOLLOW(D_check) = ['d', 'if', 'int', 'return', 'void', 'while', '{']
FOLLOW(E) = ['$', ')', '*', '+', ',', ';', 'else', 'r', '}', ' ', ' ']
FOLLOW(P) = ['$']
FOLLOW(R) = [',']
FOLLOW(R_check) = ['(', ')', 'd', 'd[]', 'i']
FOLLOW(S) = ['$', ';', 'else', '}']
FOLLOW(S_check) = ['$', ';', '}']
FOLLOW(T) = ['d', 'd[i]']


--- LL(1) Conflict Analysis ---
  Conflict for 'D_check': Direct left recursion in production D_check -> D_check D ;
  Conflict for 'D_check': FIRST/FOLLOW conflict. Production: D_check -> D_check D ; has FIRST set {'void', 'int'}, which intersects with FOLLOW(D_check) {'int', 'return', 'while', 'if', '{', 'd', 'void'}. Intersection: {'void', 'int'}
  Conflict for 'D': FIRST/FIRST conflict between T d and T d[i]. Intersection: {'void', 'int'}
  Conflict for 'D': FIRST/FIRST conflict between T d and T d ( A_check ) { D_check S_check }. Intersection: {'void', 'int'}
  Conflict for 'D': FIRST/FIRST conflict between T d[i] and T d ( A_check ) { D_check S_check }. Intersection: {'void', 'int'}
  Conflict for 'A_check': Direct left recursion in production A_check -> A_check A ;
  Conflict for 'A_check': FIRST/FOLLOW conflict. Production: A_check -> A_check A ; has FIRST set {'int', 'void', 'd[]'}, which intersects with FOLLOW(A_check) {'int', ')', 'void', 'd[]'}. Intersection: {'void', 'int', 'd[]'}
  Conflict for 'A': FIRST/FIRST conflict between T d and T d ( ). Intersection: {'void', 'int'}
  Conflict for 'S_check': Direct left recursion in production S_check -> S_check ; S
  Conflict for 'S_check': FIRST/FIRST conflict between S and S_check ; S. Intersection: {'while', 'return', 'if', '{', 'd'}
  Conflict for 'S': FIRST/FIRST conflict between d = E and d ( R_check ). Intersection: {'d'}
  Conflict for 'S': FIRST/FIRST conflict between if ( B ) S and if ( B ) S else S. Intersection: {'if'}
  Conflict for 'B': Direct left recursion in production B -> B   B
  Conflict for 'B': Direct left recursion in production B -> B   B
  Conflict for 'B': FIRST/FIRST conflict between B   B and B   B. Intersection: {'d', '(', 'i'}
  Conflict for 'B': FIRST/FIRST conflict between B   B and E r E. Intersection: {'d', '(', 'i'}
  Conflict for 'B': FIRST/FIRST conflict between B   B and E. Intersection: {'d', '(', 'i'}
  Conflict for 'B': FIRST/FIRST conflict between B   B and E r E. Intersection: {'d', '(', 'i'}
```

```
Conflict for 'B': FIRST/FIRST conflict between B   B and E. Intersection: {'d', '(', 'i'}
Conflict for 'B': FIRST/FIRST conflict between E r E and E. Intersection: {'d', '(', 'i'}
Conflict for 'E': Direct left recursion in production E -> E + E
Conflict for 'E': Direct left recursion in production E -> E * E
Conflict for 'E': FIRST/FIRST conflict between d = E and d. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d = E and d ( R_check ). Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d = E and E + E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d = E and E * E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between i and E + E. Intersection: {'i'}
Conflict for 'E': FIRST/FIRST conflict between i and E * E. Intersection: {'i'}
Conflict for 'E': FIRST/FIRST conflict between d and d ( R_check ). Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d and E + E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d and E * E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d ( R_check ) and E + E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between d ( R_check ) and E * E. Intersection: {'d'}
Conflict for 'E': FIRST/FIRST conflict between E + E and E * E. Intersection: {'d', '(', 'i'}
Conflict for 'E': FIRST/FIRST conflict between E + E and ( E ). Intersection: {'('}
Conflict for 'E': FIRST/FIRST conflict between E * E and ( E ). Intersection: {'('}
Conflict for 'R_check': Direct left recursion in production R_check -> R_check R ,
Conflict for 'R_check': FIRST/FOLLOW conflict. Production: R_check -> R_check R , has FIRST set {'d', '(', 'i', 'd[]'}, which intersects with FOLLOW(R_check) {'(', ')', 'd', 'i', 'd[]'}. Intersection: {'d[]', 'i', '(', 'd'}
Conflict for 'R': FIRST/FIRST conflict between E and d ( ). Intersection: {'d'}
```

Listing 2.2　程序输出

3 大作业三：SLR(1) 分析过程详解

3.1 引言

本报告对给定文法进行 SLR(1) 分析。SLR(1) 分析是编译器构造中的重要组成部分，涉及计算文法符号的 FIRST 集和 FOLLOW 集，构建 LR(0) 项目集规范族 (ItemDFA)，生成 SLR(1) 分析表，并最终识别和处理分析过程中可能出现的冲突。

由于给定文法的规模和复杂性，完全手动生成所有中间步骤 (特别是完整的 ItemDFA 和 SLR(1) 分析表) 是不切实际的。因此，本报告将侧重于以下几个方面：

- 定义文法结构，包括产生式、终结符和非终结符。
- 计算所有非终结符的 FIRST 集和 FOLLOW 集，并对关键集合的推导过程进行说明。
- 阐述 ItemDFA 的构造原理，并给出初始状态 I_0 以及其他几个关键状态 (如包含冲突的状态、接受状态、经历重要转移的状态) 的 LR(0) 项目集。同时，通过示例展示状态间的转移关系 ($goto$ 函数)。
- 明确 SLR(1) 分析表的构造规则，并提供针对关键状态的部分分析表片段，展示 ACTION 和 GOTO 表的条目。
- 分析文法中存在的、导致其非 SLR(1) 的各类冲突 (如移进/归约冲突、归约/归约冲突)，解释冲突产生的原因，并讨论 SLR(1) 方法在处理这些冲突时的局限性。
- 对 SLR(1) 冲突的判定原则进行总结，并简要提及在实际编译器构造中处理这些冲突的常见策略。

3.2 文法 G

3.2.1 产生式列表

增广文法 G' 的产生式如下 (原始文法产生式从 1 开始编号):

0. $P' \to P$ (增广产生式)
1. $P \to \check{D}\check{S}$
2. $\check{D} \to \varepsilon$
3. $\check{D} \to \check{D}D$ `#1`
4. $D \to T$ `#1`
5. $D \to T$ `#1 #1 #1 #1` (注意: `#1` 此处视为终结符)
6. $D \to T$ `#1 #1` $\check{A}$ `#1 #1` $\check{D}\check{S}$ `#1`
7. $T \to$ #1
8. $T \to$ #1
9. $\check{A} \to \varepsilon$

10. $\check{A} \to \check{A} A$ `#1`
11. $A \to T$ `#1`
12. $A \to$ `#1 #1 #1`
13. $A \to T$ `#1 #1 #1`
14. $\check{S} \to S$
15. $\check{S} \to \check{S}$ `#1` S
16. $S \to$ `#1 #1` E
17. $S \to$ #1 `#1` B `#1` S
18. $S \to$ #1 `#1` B `#1` S #1 S
19. $S \to$ #1 `#1` B `#1` S
20. $S \to$ #1 E
21. $S \to$ `#1` $\check{S}$ `#1`
22. $S \to$ `#1 #1` $\check{R}$ `#1`
23. $B \to B$ `#1` B
24. $B \to B$ `#1` B
25. $B \to E$ `#1` E
26. $B \to E$
27. $E \to$ `#1 #1` E
28. $E \to$ `#1`
29. $E \to$ `#1`
30. $E \to$ `#1 #1` $\check{R}$ `#1`
31. $E \to E$ `#1` E
32. $E \to E$ `#1` E
33. $E \to$ `#1` E `#1`
34. $\check{R} \to \varepsilon$
35. $\check{R} \to \check{R} R$ `#1`
36. $R \to E$
37. $R \to$ `#1 #1 #1`
38. $R \to$ `#1 #1 #1`

3.2.2 终结符 (Terminals)

$T_{symbols} = \{$`#1`, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, #1, #1, `#1`, #1, #1, #1, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, $\$\}$ ($ 为输入结束符)

3.2.3 非终结符 (Non-Terminals)

$N_{symbols} = \{P', P, \check{D}, D, T, \check{A}, A, \check{S}, S, B, E, \check{R}, R\}$

3.3 FIRST 集

FIRST(α) 是从 α 可能推导出的串的第一个终结符的集合。如果 $\alpha \Rightarrow^* \varepsilon$，那么 $\varepsilon \in$ FIRST(α)。

FIRST(T) = {#1, #1}
FIRST(D) = {#1, #1} (来自 $D \to T \dots$)
FIRST($\check{D}$) = {#1, #1, ε} (来自 FIRST(D) $\cup$ {ε})
FIRST(A) = {#1, #1, `#1`} (来自 $A \to Td$, $A \to$ `#1`[], $A \to Td()$)
FIRST($\check{A}$) = {#1, #1, `#1`, ε} (来自 FIRST(A) $\cup$ {ε})
FIRST(E) = {`#1`, `#1`, `#1`} (来自 $E \to$ `#1` $\dots$, $E \to$ `#1`, $E \to$ `#1`E`#1`)
FIRST(R) = {`#1`, `#1`, `#1`} (来自 $R \to E$, $R \to$ `#1`[], $R \to$ `#1`())
FIRST($\check{R}$) = {`#1`, `#1`, `#1`, ε} (来自 FIRST(R) $\cup$ {ε})
FIRST(B) = {`#1`, `#1`, `#1`} (来自 $B \to E \dots$, $B \to E$)
FIRST(S) = {`#1`, #1, #1, #1, `#1`}
FIRST($\check{S}$) = {`#1`, #1, #1, #1, `#1`} (来自 $\check{S} \to S$)
FIRST(P) = {#1, #1, `#1`, #1, #1, #1, `#1`} (若 $\check{D} \to \varepsilon$，则为 FIRST($\check{S}$)；否则为 FIRST($\check{D}$) $-$ {ε})

注：FIRST(E) 包含 `#1` 是因为产生式 $E \to$ `#1`, $E \to$ `#1` $= E$, $E \to$ `#1`($\check{R}$)。

3.4 FOLLOW 集

FOLLOW(N) 是可能跟在非终结符 N 之后的所有终结符的集合。如果 N 可能出现在某个句型的末尾，则 \$ $\in$ FOLLOW(N)。

FOLLOW(P') = {\$}
FOLLOW(P) = {\$} (来自 $P' \to P\cdot$)
FOLLOW($\check{D}$) = FIRST($\check{S}$) = {`#1`, #1, #1, #1, `#1`} (来自 $P \to \check{D}\underline{\check{S}}$ 和 $D \to \dots \{\check{D}\underline{\check{S}}\}$)
FOLLOW(D) = {`#1`} (来自 $\check{D} \to \check{D}D\underline{\texttt{\#1}}$)
FOLLOW(T) = {`#1`} (来自 $D \to T\underline{\texttt{\#1}} \dots$, $A \to T\underline{\texttt{\#1}} \dots$)
FOLLOW($\check{A}$) = {`#1`} (来自 $D \to T\texttt{\#1}(\check{A}\underline{\texttt{\#1}} \dots$)
FOLLOW(A) = {`#1`} (来自 $\check{A} \to \check{A}A\underline{\texttt{\#1}}$)
FOLLOW($\check{S}$) = {\$, `#1`} (来自 $P \to \check{D}\check{S}\cdot$ (则继承 FOLLOW(P)), $D \to \dots \{\check{D}\check{S}\underline{\texttt{\#1}}\}$, $S \to \{\check{S}\underline{\texttt{\#1}}\}$)
FOLLOW(S) = FOLLOW($\check{S}$) $\cup$ {`#1`, #1} = {\$, `#1`, `#1`, #1} (来自 $\check{S} \to S\cdot$, $\check{S} \to \check{S}\underline{\texttt{\#1}}S$, $S \to$ #1(B)$S\underline{\#1}S$)
FOLLOW(B) = {`#1`, `#1`, `#1`} (来自 $S \to \dots (B\underline{\texttt{\#1}}) \dots$, $B \to B\underline{\texttt{\#1}}B$, $B \to B\underline{\texttt{\#1}}B$)
FOLLOW(E) = FOLLOW(S) $\cup$ FOLLOW(B) $\cup$ FOLLOW(R) $\cup$ {`#1`, `#1`, `#1`, `#1`} = {\$, `#1`, `#1`, #1, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`, `#1`}
FOLLOW($\check{R}$) = {`#1`} (来自 $S \to$ `#1`($\check{R}\underline{\texttt{\#1}}$), $E \to$ `#1`($\check{R}\underline{\texttt{\#1}}$))
FOLLOW(R) = {`#1`} (来自 $\check{R} \to \check{R}R\underline{\texttt{\#1}}$)

关于 FOLLOW(E) 的推导说明: FOLLOW(E) 的计算综合了以下情况：
1) $S \to$ `#1` $= E$ 和 $S \to$ #1E: E 之后可跟 FOLLOW(S)。
2) $B \to E$`#1`E_2: 第一个 E 后可跟 `#1`；第二个 E_2 后可跟 FOLLOW(B)。
3) $B \to E$: E 之后可跟 FOLLOW(B)。
4) $E \to E_1$`#1`E_2: E_1 后可跟 `#1`；E_2 后可跟 FOLLOW(E) (处理左递归)。

5) $E \to E_1\texttt{\#1}E_2$: E_1 后可跟 `#1`；E_2 后可跟 FOLLOW(E)。

6) $E \to \texttt{\#1}E_1\texttt{\#1}$: E_1 后可跟 `#1` (此处的 `#1` 是直接的，加入到集合中)。

7) $R \to E$: E 之后可跟 FOLLOW(R)。

综合以上，并迭代求解，得到 FOLLOW(E) 如上所示。该集合是正确的，它反映了 E 可能出现的所有上下文的后继符号。

3.5 LR(0) 项目集规范族 (ItemDFA) 与 SLR(1) 分析表构造

3.5.1 LR(0) 项目集规范族的构造 (ItemDFA)

ItemDFA 是通过对文法的 LR(0) 项目进行闭包 (closure) 和转移 (goto) 运算得到的。由于文法规模较大，其 ItemDFA 的状态数量非常多，完整手动构造不切实际。下面展示一些关键状态及其构造。

LR(0) 项目闭包 (Closure) 运算规则: 若 $[A \to \alpha \cdot B\beta]$ 是一个项目，且 $B \to \gamma$ 是一个产生式，则项目 $[B \to \cdot\gamma]$ 也应加入到闭包中。此过程重复直到没有新项目可以加入。若 B 可空 (即 $B \Rightarrow^* \varepsilon$)，且之后有 δ ($A \to \alpha \cdot B\delta \dots$)，则还需考虑 δ 的起始。但 LR(0) 项目不考虑 lookahead。

状态 I_0 = closure($\{[P' \to \cdot P]\}$):

- $P' \to \cdot P$
- $P \to \cdot\check{D}\check{S}$ (来自 $P' \to \cdot P$)
- $\check{D} \to \cdot\varepsilon$ (来自 $P \to \cdot\check{D}\check{S}$)
- $\check{D} \to \cdot\check{D}D\,\texttt{\#1}$ (来自 $P \to \cdot\check{D}\check{S}$)
- $D \to \cdot T\,\texttt{\#1}$ (来自 $\check{D} \to \cdot\check{D}D\dots$ 的闭包，当 $\check{D}$ 最终展开为 $D\dots$ 时)
- $D \to \cdot T\,\texttt{\#1\#1\#1\#1}$ (同上)
- $D \to \cdot T\,\texttt{\#1\#1}\,\check{A}\,\texttt{\#1\#1}\,\check{D}\check{S}\,\texttt{\#1}$ (同上)
- $T \to \cdot\#1$ (来自 $D \to \cdot T\dots$)
- $T \to \cdot\#1$ (来自 $D \to \cdot T\dots$)
- $\check{S} \to \cdot S$ (来自 $P \to \cdot\check{D}\check{S}$ 且 $\check{D} \to \cdot\varepsilon$)
- $\check{S} \to \cdot\check{S}\,\texttt{\#1}\,S$ (同上)
- $S \to \cdot\texttt{\#1\#1}\,E$ (来自 $\check{S} \to \cdot S$)
- $S \to \cdot\#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S$ (同上)
- $S \to \cdot\#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S\#1\,S$ (同上)
- $S \to \cdot\#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S$ (同上)
- $S \to \cdot\#1\,E$ (同上)
- $S \to \cdot\texttt{\#1}\,\check{S}\,\texttt{\#1}$ (同上)
- $S \to \cdot\texttt{\#1\#1}\,\check{R}\,\texttt{\#1}$ (同上)
- $B \to \cdot B \wedge B, B \to \cdot B \vee B, B \to \cdot ErE, B \to \cdot E$ (来自 $S \to \#1(\cdot B)\dots$ 等)

- $E \to \cdot d = E$, $E \to \cdot i$, $E \to \cdot d$, $E \to \cdot d(\check{R})$, $E \to \cdot E + E$, $E \to \cdot E * E$, $E \to \cdot(E)$ (来自 $S \to \ldots E$, $B \to \cdot E \ldots$ 等)

注：I_0 中不直接包含由 $\check{A}$ 或 $\check{R}$ 的 ε-产生式引入的 A 或 R 的项目。这些项目会在 $goto$ 到达点在 $\check{A}$ 或 $\check{R}$ 之前的状态时，通过闭包运算加入。例如，若有状态 I_x 含项目 $D \to T\texttt{\#1}(\cdot\check{A}\ldots)$，则 closure($I_x$) 会包含 $\check{A} \to \cdot\varepsilon$, $\check{A} \to \cdot\check{A}A;$, 进而包含 $A \to \cdot T\texttt{\#1}$ 等。

部分状态转移示例:

- $I_1 = goto(I_0, P)$: 核心项目 $\{[P' \to P\cdot]\}$. 此为接受状态的基础。
- $I_a = goto(I_0, \check{D})$: 核心项目 $\{[P \to \check{D} \cdot \check{S}], [\check{D} \to \check{D} \cdot D\,\texttt{\#1}]\}$. closure($I_a$) 还会加入:
  - $\check{S} \to \cdot S$
  - $\check{S} \to \cdot\check{S}\,\texttt{\#1}\,S$
  - (所有 $S \to \cdot\gamma$ 及其后续闭包项目)
  - $D \to \cdot T\,\texttt{\#1}$ (来自 $\check{D} \to \check{D} \cdot D \ldots$)
  - (所有 $D \to \cdot\delta$ 及其后续闭包项目)
- $I_b = goto(I_a, \check{S})$ (假设从 $P \to \check{D} \cdot \check{S}$ 转移): 核心项目 $\{[P \to \check{D}\check{S}\cdot], [\check{S} \to \check{S} \cdot \texttt{\#1}S]\}$. closure($I_b$) 还会加入:
  - (无新项目由 $P \to \check{D}\check{S}\cdot$ 引入)
  - $S \to \cdot\gamma$ (来自 $\check{S} \to \check{S} \cdot \texttt{\#1}S$ 中点后的 S)
- $I_c = goto(I_0, \#1)$: 核心项目 $\{[T \to \#1\cdot]\}$. (准备归约的状态)
- $I_d = goto(I_c, \texttt{\#1})$ (即 $goto(goto(I_0, \#1), \texttt{\#1})$): 核心项目 $\{[D \to T\texttt{\#1}\cdot], [D \to T\texttt{\#1} \cdot [\texttt{\#1}]], [D \to T\texttt{\#1} \cdot (\check{A})\ldots], \ldots\}$.
- $I_e = goto(I_d, \texttt{\#1})$ (即 $goto(goto(goto(I_0, \#1), \texttt{\#1}), \texttt{\#1})$): 核心项目 $\{[D \to T\texttt{\#1}(\cdot\check{A})\{\check{D}\check{S}\}]\}$. closure($I_e$) 会包含:
  - $\check{A} \to \cdot\varepsilon$
  - $\check{A} \to \cdot\check{A}A\texttt{\#1}$
  - $A \to \cdot T\texttt{\#1}$
  - $A \to \cdot\texttt{\#1}[]$
  - $A \to \cdot T\texttt{\#1}()$
  - $T \to \cdot\#1$
  - $T \to \cdot\#1$

悬空 Else 相关的状态 I_k: 假设通过一系列 $goto$ 运算到达状态 I_k，该状态是在匹配了产生式 $S \to \#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S$ 的右部"#1 #1 B #1 S" 之后形成。I_k 的核心项目集将包含:

- $[S \to \#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S\cdot]$ (来自产生式 17)
- $[S \to \#1\,\texttt{\#1}\,B\,\texttt{\#1}\,S \cdot \#1\,S]$ (来自产生式 18, 因为两个产生式共享相同前缀)

部分 ItemDFA 状态转移示意图 (使用 TikZ):

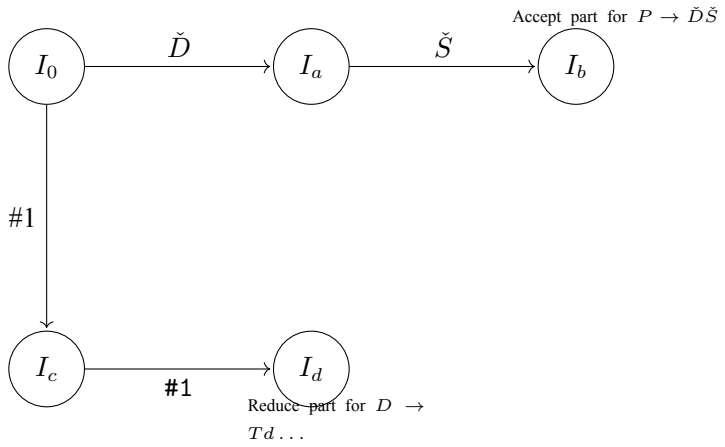


图 3-1　ItemDFA 部分关键状态转移示意图

3.5.2　SLR(1) 分析表的构造规则与示例

SLR(1) 分析表包含 ACTION 表和 GOTO 表。

- **ACTION 表**: 对于状态 I_k 和终结符 a:

  1) 若项目 $[A \to \alpha \cdot a\beta]$ 属于 I_k，且 $goto(I_k, a) = I_j$, 则 ACTION$[k, a]$ = "shift j" (移进 s_j)。

  2) 若项目 $[A \to \alpha\cdot]$ 属于 I_k ($A \neq P'$), 则对所有终结符 $b \in$ FOLLOW(A)，ACTION$[k, b]$ 包含"reduce $A \to \alpha$" (用产生式 m 归约，记为 r_m, m 是产生式编号)。

  3) 若项目 $[P' \to P\cdot]$ 属于 I_k，则 ACTION$[k, \$]$ = "accept"。

- **GOTO 表**: 对于状态 I_k 和非终结符 A:

  1) 若 $goto(I_k, A) = I_j$, 则 GOTO$[k, A] = j$。

SLR(1) 分析表片段示例: 假设 r_N 表示用产生式 N 归约, s_M 表示移进到状态 I_M。

状态	ACTION											GOTO		
	`#1`	#1	#1	#1	`#1`	#1	#1	`#1`	`#1`	...	$	P	$\check{D}$	$\check{S}$
I_0	s_x/r_2^*	s_y/r_2^*	s_z/r_2^*	s_w/r_2^*	s_v/r_2^*	s_u	$r_2^\dagger$	$r_2^\ddagger$		...		g_1	g_a	$g_?$
I_c ($T \to$ #1$\cdot$)	r_7									...				
I_k (悬空 else)							s_p/r_{17}^*	r_{17}	$r_{17}^\S$	...	r_{17}			
I_1 ($P' \to P\cdot$)										...	acc			

表格注释与说明:

- * 表示移进/归约冲突。例如，在 I_0 遇到 `#1`，既可能移进 (如 s_x 对应 $S \to \cdot$`#1`$\ldots$)，也可能归约 (r_2 对应 $\check{D} \to \varepsilon$，因为 `#1` $\in$ FOLLOW$(\check{D})$)。
- r_N: 用第 N 条产生式归约。例如，r_2 是 $\check{D} \to \varepsilon$，$r_7$ 是 $T \to$ #1，r_{17} 是 $S \to$ #1$(B)S$。
- s_M: 移进并转移到状态 I_M。
- g_X: 转移到状态 I_X (GOTO 表项)。
- †: r_2 在 #1 列，仅当 #1 $\in$ FOLLOW$(\check{D})$ 时出现。在此文法中，FOLLOW$(\check{D})$ 不含 #1。
- ‡: r_2 在 `#1` 列，仅当 `#1` $\in$ FOLLOW$(\check{D})$ 时出现。在此文法中，FOLLOW$(\check{D})$ 不含 `#1`。
- §: r_{17} 在 `#1` 列，仅当 `#1` $\in$ FOLLOW(S) 时出现。在此文法中，FOLLOW(S) 不含 `#1` (除非通过 $E \to (E)$ 间接影响)。

- I_k (悬空 else 状态) 在输入 #1 时存在 s_p/r_{17} 冲突。对于 FOLLOW(S) 中的其他符号 (如 `#1`, \$)，则执行归约 r_{17}。

3.5.3 SLR(1) 冲突的判定与处理原则

- **SLR(1) 冲突的判定**: SLR(1) 分析方法本身不提供内建的冲突“消解”机制。当构造分析表时，如果 ACTION 表的任何一个单元格 ACTION[k, a] 基于当前状态 I_k 的项目集和文法的 FOLLOW 集计算出多个不同动作，则称该处存在冲突。

  1) **移进/归约冲突 (Shift/Reduce Conflict, S/R)**: 在状态 I_k 中，若存在：

  – 归约项目: $[A \to \alpha\cdot]$，且当前输入终结符 $a \in$ FOLLOW(A) (这指示了一个归约动作 r_A)。

  – 移进项目: $[B \to \beta \cdot a\gamma]$ (这指示了一个在 a 上的移进动作 s_j)。

  此时，在 (I_k, a) 处存在 S/R 冲突。SLR(1) 无法仅凭下一输入符号和 FOLLOW 集信息来决定是移进还是归约。

  2) **归约/归约冲突 (Reduce/Reduce Conflict, R/R)**: 在状态 I_k 中，若存在两个不同的归约项目：

  – $[A \to \alpha\cdot]$

  – $[B \to \beta\cdot]$ ($A \to \alpha \neq B \to \beta$)

  且存在某个终结符 a 使得 $a \in$ FOLLOW(A) 并且 $a \in$ FOLLOW(B)。此时，在 (I_k, a) 处存在 R/R 冲突，因为分析器不知道应该使用哪个产生式进行归约。

  如果分析表中存在任何此类冲突，则文法不是 SLR(1) 文法。

- **实际编译器中的冲突处理 (超出 SLR(1) 范畴)**: 虽然纯粹的 SLR(1) 分析器遇到冲突即宣告失败，但实际的解析器生成工具 (如 YACC/Bison，它们通常实现 LALR(1)) 提供了处理冲突的机制：

  – **默认解决**: 许多工具默认解决 S/R 冲突为“优先移进 (prefer shift)”。R/R 冲突则可能选择文法中较早出现的产生式。

  – **优先级和结合性声明**: 对于表达式文法中的冲突，可以为终结符 (运算符) 声明优先级和结合性，解析器生成器会利用这些信息来解决 S/R 冲突。例如，声明 `*` 比 `+` 优先级高，`+` 左结合。

  – **文法重写**: 有时，修改文法本身可以消除二义性或冲突，使其适应更简单的分析方法。

  这些策略是工程实践，不属于 SLR(1) 理论本身。

3.6 无法消解的冲突分析 (Unresolvable Conflicts in SLR(1))

3.6.1 冲突 1: $\check{D} \to \varepsilon$ 导致的移进/归约冲突 (在 I_0)

状态 I_0 中的相关项目及分析:

- 项目 $\check{D} \to \cdot\varepsilon$ (来自产生式 2)。
- 对于任何终结符 $t \in$ FOLLOW($\check{D}$) $= \{$`#1`, #1, #1, #1, `#1`$\}$, ACTION[I_0, t] 将包含”reduce by $\check{D} \to \varepsilon$” ($r_2$)。

- 同时，由于 $\check{D}$ 可空，$P \to \check{D}\check{S}$ 使得 $\check{S}$ 的产生式在 I_0 中被激活 (通过 $\check{S} \to \cdot S$ 等)。例如：
  - $S \to \cdot$ `#1` $= E$ (产生式 16)
  - $S \to \cdot$ `#1` `#1` B `#1` S (产生式 17)
  - ... (其他 S 产生式)

  这些项目导致当 t (例如 `#1` 或 #1) 是这些产生式的起始符号时，ACTION$[I_0, t]$ 包含"shift" 动作。

冲突结果: 在 I_0 状态，对于输入符号 `#1`, #1, #1, #1, `#1`, 都存在移进 (s_x) / 归约 (r_2) 冲突。**原因**: SLR(1) 仅查看下一输入符号是否在 FOLLOW($\check{D}$) 中来决定是否用 $\check{D} \to \varepsilon$ 归约。然而，这些符号也是 $\check{S}$ (当 $\check{D}$ 为空时紧随其后的非终结符) 的合法起始符号，需要移进。SLR(1) 缺乏足够的上下文来区分这两种决策。**结论**: 此冲突是 SLR(1) 固有的，无法解决。

3.6.2 冲突 2: "Dangling Else" (悬空 else) 导致的移进/归约冲突

相关状态 I_k 中的项目 (在匹配了 #1 `#1` B `#1` S 之后):

- $[S \to$ #1 `#1` B `#1` $S\cdot]$ (来自产生式 17)
- $[S \to$ #1 `#1` B `#1` $S \cdot$ #1 $S]$ (来自产生式 18)

冲突详情: FOLLOW(S) 包含 #1 (即 $\{\$, $ `#1`, `#1`, #1$\}$). 当状态为 I_k 且下一个输入符号是 #1 时:

- **归约动作**: 根据项目 (17) 和 #1 $\in$ FOLLOW(S), ACTION$[I_k,$ #1$]$ 将包含"reduce by $S \to$ #1 `#1` B `#1` S" (r_{17})。
- **移进动作**: 根据项目 (18), ACTION$[I_k,$ #1$]$ 将包含"shift" (移进 #1)。

这是一个移进/归约冲突。**原因**: 这是经典的悬空 else 问题，源于文法的二义性。SLR(1) 分析器无法确定 #1 应该与哪个 #1 匹配。**结论**: 此冲突无法由 SLR(1) 解决。

3.6.3 冲突 3: 表达式相关的移进/归约冲突 (运算符优先级与结合性)

文法中的表达式产生式如 $E \to E$`#1`E (规则 31), $E \to E$`#1`E (规则 32)。**示例场景**: 考虑状态 I_x 包含项目 $E \to \alpha\cdot$ (例如 $\alpha =$ `#1` 或 $\alpha = E_1$`#1`E_2)。同时，该状态 I_x (或其前驱状态导致 I_x 可达) 也可能允许移进运算符，例如有项目 $E \to E \cdot$ `#1`E' 或 $E \to E \cdot$ `#1`E''。如果下一个输入是 `#1`:

- **归约动作**: 因为 `#1` $\in$ FOLLOW(E), ACTION$[I_x,$ `#1`$]$ 会包含"reduce by $E \to \alpha$"。
- **移进动作**: ACTION$[I_x,$ `#1`$]$ 会包含"shift" (来自 $E \to E \cdot$ `#1`E')。

这是一个移进/归约冲突。对于 `#1` 也是同理。**原因**: SLR(1) 无法确定是应该将当前已识别的 E (即 α) 进行归约，还是应该移进运算符以形成更长的表达式。这涉及到运算符的优先级和结合性，而 SLR(1) 没有这些信息。**结论**: 这些由表达式文法引起的冲突表明文法存在二义性，或者至少是 SLR(1) 无法处理的。

3.6.4 关于 $\check{A} \to \varepsilon$ 和 $\check{R} \to \varepsilon$ 的可空性分析

与 $\check{D} \to \varepsilon$ 不同，$\check{A} \to \varepsilon$ 和 $\check{R} \to \varepsilon$ 在此文法中通常不会引发类似 I_0 中那种因 FOLLOW 集与 FIRST 集重叠导致的 S/R 冲突。

- **对于 $\check{A} \to \varepsilon$ (产生式 9)**: $\text{FOLLOW}(\check{A}) = \{\texttt{\#1}\}$. $\text{FIRST}(A) = \{\#1, \#1, \texttt{\#1}\}$. 在一个可能归约 $\check{A} \to \varepsilon$ 的状态 (例如，状态 I_e 中包含 $D \to T\texttt{\#1}(\cdot\check{A}\dots)$，其闭包含 $\check{A} \to \cdot\varepsilon$)，若下一个输入是 `#1`，则可以归约。若下一个输入是 #1, #1, 或 `#1` (即 $\text{FIRST}(A)$ 中的元素)，则会移进以开始解析 A。由于 $\{\texttt{\#1}\} \cap \{\#1, \#1, \texttt{\#1}\} = \emptyset$，因此在同一个输入符号上不会同时存在归约 $\check{A} \to \varepsilon$ 和移进 A 的起始符号的动作。
- **对于 $\check{R} \to \varepsilon$ (产生式 34)**: $\text{FOLLOW}(\check{R}) = \{\texttt{\#1}\}$. $\text{FIRST}(R) = \{\texttt{\#1}, \texttt{\#1}, \texttt{\#1}\}$. 类似地，由于 $\{\texttt{\#1}\} \cap \{\texttt{\#1}, \texttt{\#1}, \texttt{\#1}\} = \emptyset$，所以也不会产生与 $\check{D}$ 类似的 S/R 冲突。

结论: $\check{A} \to \varepsilon$ 和 $\check{R} \to \varepsilon$ 本身不直接导致与其相关的移进操作在 SLR(1) 分析中产生冲突，因为决定归约的 FOLLOW 集与决定移进的 FIRST 集是分离的。

3.7 总结与结论

通过上述的分析，该文法存在多个无法由 SLR(1) 分析方法解决的移进/归约冲突，主要包括：

1) 由于 $\check{D} \to \varepsilon$ 的可空性，在初始状态 I_0 中，对于 $\text{FOLLOW}(\check{D})$ 中的多个符号均存在冲突。

2) 经典的”悬空 else”问题导致了文法的二义性，表现为 SLR(1) 分析中的移进/归约冲突。

3) 表达式产生式（如涉及 `#1`, `#1`）由于缺乏运算符优先级和结合性的定义，同样会导致移进/归约冲突。

因此，可以明确地得出结论：该文法 **不是 SLR(1) 文法**。

要成功解析此文法，通常需要采用以下策略之一或组合：

- 使用更强大的分析技术，如 LALR(1) 或 LR(1) 分析法，它们能利用更丰富的上下文信息。
- 在使用 LALR(1) 解析器生成工具 (如 YACC/Bison) 时，通过声明运算符的优先级和结合性来解决表达式相关的冲突，并可能依赖工具对 S/R 冲突的默认解决规则 (如优先移进处理悬空 else)。
- 对文法本身进行重写，以消除二义性和冲突，使其能够适应 SLR(1) 或其他更简单的分析方法。

3.8 代码实现与运行结果

3.8.1 源程序

```
import json


class ASTNode:
    """定义抽象语法树 (AST) 的节点"""
    def __init__(self, node_type, value=None, children=None):
```

```
        self.type = node_type       # 节点类型 (如: 'S', 'E', 'd', 'int')
        self.value = value          # 节点值 (通常用于终结符)
        self.children = children if children is not None else [] # 子节点列表

    def __repr__(self):
        return f"ASTNode({self.type}, {self.value})"

def print_ast(node, indent_level=0):
    """以树状结构优美地打印 AST"""
    indent = "  " * indent_level
    node_info = f"{node.type}"
    if node.value:
        node_info += f" [{node.value}]"
    print(indent + node_info)
    for child in node.children:
        print_ast(child, indent_level + 1)


class SLRParser:
    """
    一个功能更完整的 SLR(1) 语法分析器类。
    """
    def __init__(self, grammar_config):
        """
        使用配置字典初始化分析器。

        参数:
        grammar_config (dict): 包含产生式、ACTION表、GOTO表和冲突解决策略的字典。
        """
        self.productions = {int(k): v for k, v in grammar_config['productions'].
    items()}
        self.action_table = {int(k): v for k, v in grammar_config['action_table'].
    items()}
        self.goto_table = {int(k): v for k, v in grammar_config['goto_table'].
    items()}
        self.conflict_resolutions = grammar_config.get('conflict_resolutions', {})

    def _get_production(self, prod_num):
        """根据编号获取产生式详情"""
        return self.productions[prod_num]

    def _report_error(self, state, lookahead_token):
        """生成并打印增强的错误报告"""
        expected_tokens = [f"'{t}'" for t in self.action_table.get(state, {}).keys
    ()]
        error_msg = f"\n语法错误: 在状态 {state} 遇到意外的符号 '{lookahead_token
    }'。\n"
        if expected_tokens:
```

```
            error_msg += f"        此处期望的符号可能是: {', '.join(expected_tokens)}"
        else:
            error_msg += "        在此状态下没有合法的后续符号。"
        print(error_msg)

    def parse(self, tokens, verbose=True):
        """
        执行语法分析并构建 AST。

        参数:
        tokens (list): 词法单元列表。
        verbose (bool): 是否打印详细的分析步骤。

        返回:
        ASTNode: 如果分析成功，返回 AST 的根节点；否则返回 None。
        """
        # 状态栈，处理分析器状态转移
        state_stack = [0]
        # 值栈，用于构建 AST
        value_stack = []

        pointer = 0

        if verbose:
            print("--- 开始 SLR(1) 分析 (优化版) ---")
            print(f"{'步骤':<5} {'状态栈':<20} {'当前符号':<10} {'值栈 (部分)':<25} {'动作':<35}")
            print("-" * 100)

        step = 0
        while True:
            step += 1
            current_state = state_stack[-1]
            lookahead = tokens[pointer]

            # --- 打印当前步骤 ---
            if verbose:
                # 为了可读性，只显示值栈顶部的几个节点
                top_values = [n.type for n in value_stack[-5:]]
                print(f"{step:<5} {str(state_stack):<20} {lookahead:<10} {str(top_values):<25}", end="")

            # --- 查询 ACTION 表 ---
            action = self.action_table.get(current_state, {}).get(lookahead)

            if action is None:
                self._report_error(current_state, lookahead)
                return None
```

```
96
97              # --- 处理冲突 ---
98              if '/' in action:
99                  # 检查是否有预设的冲突解决方案
100                 resolution_key = f"{current_state},{lookahead}"
101                 if resolution_key in self.conflict_resolutions:
102                     choice = self.conflict_resolutions[resolution_key]
103                     action = 's' + action.split('/')[0][1:] if choice == 'shift'
    else 'r' + action.split('/')[1][1:]
104                     if verbose: print(f"冲突 {action}, 按预设规则解决 -> {choice}"
    )
105                 else:
106                     # 默认策略: 优先移进
107                     original_action = action
108                     action = action.split('/')[0]
109                     if verbose: print(f"冲突 {original_action}, 默认优先移进 -> {
    action}")
110
111             # --- 执行动作 ---
112             if action.startswith('s'):  # 移进 (Shift)
113                 state_to_push = int(action[1:])
114                 state_stack.append(state_to_push)
115                 # 为终结符创建一个 AST 叶子节点并压入值栈
116                 value_stack.append(ASTNode(lookahead, value=lookahead))
117                 pointer += 1
118                 if verbose: print(f"移进 {lookahead}, 进入状态 {state_to_push}")
119
120             elif action.startswith('r'):  # 归约 (Reduce)
121                 prod_num = int(action[1:])
122                 lhs, rhs = self._get_production(prod_num)
123                 if verbose: print(f"归约: {lhs} -> {' '.join(rhs) if rhs else ' '}
     (规则 {prod_num})")
124
125                 # 从值栈中弹出相应数量的子节点
126                 pop_len = len(rhs)
127                 children = value_stack[-pop_len:] if pop_len > 0 else []
128                 if pop_len > 0:
129                     state_stack = state_stack[:-pop_len]
130                     value_stack = value_stack[:-pop_len]
131
132                 # 创建一个新的父节点
133                 new_node = ASTNode(lhs, children=children)
134                 value_stack.append(new_node)
135
136                 # GOTO
137                 prev_state = state_stack[-1]
138                 goto_state = self.goto_table.get(prev_state, {}).get(lhs)
139                 if goto_state is None:
140                     self._report_error(prev_state, lhs) # GOTO 失败也是一种错误
```

```
                    return None
                state_stack.append(goto_state)

            elif action == 'acc':  # 接受 (Accept)
                if verbose: print("\n分析成功：输入串被接受！")
                # 最终值栈里应该只剩下起始符号的根节点
                return value_stack[0]

# --- 主程序入口 ---
if __name__ == '__main__':
    # 将文法、分析表、冲突解决策略定义为配置，与分析器解耦
    GRAMMAR_CONFIG = {
        "productions": {
            0: ["P'", ["P"]],
            1: ["P", ["check_D", "check_S"]],
            2: ["check_D", []],  #   产生式
            4: ["D", ["T", "d"]],
            7: ["T", ["int"]],
            14: ["check_S", ["S"]],
            16: ["S", ["d", "=", "E"]],
            28: ["E", ["i"]],
        },
        "action_table": {
            0: {'int': 's2', 'd': 's4/r2'}, # 状态0存在 s/r 冲突
            1: {'$': 'acc'},
            2: {'d': 's9'},
            3: {'d': 's11', '$': 'r2'}, # 简化：在 $ 前可以直接完成 check_D ->
            4: {'=': 's12'},
            9: {';': 'r4', '$': 'r4'},
            10: {'$': 'r14'},
            11: {'=': 's12'},
            12: {'i': 's14'},
            14: {';': 'r28', '$': 'r28'},
            15: {'$': 'r1'}
        },
        "goto_table": {
            0: {'P': 1, 'check_D': 3, 'T': 5}, # 状态0遇到T到状态5(虚构)
            2: {},
            3: {'S': 10, 'check_S': 15},
            4: {},
            12: {'E': 16} # 虚构
        },
        "conflict_resolutions": {
            # "状态,符号": "shift" 或 "reduce"
            # "0,d": "shift" # 示例：明确指定在状态0遇到d时优先移进
        }
    }

    # 1. 创建分析器实例
```

```
    parser = SLRParser(GRAMMAR_CONFIG)

    # 2. 定义输入流
    # input_stream = ['int', 'd', ';', '$'] # 这个例子在简化表中不完整
    input_stream_with_conflict = ['d', '=', 'i', ';', '$']

    # 3. 执行分析
    print("--- 测试用例: d = i ; $ ---")
    print("此用例将在状态 0 遇到 'd' 时触发移进/归约冲突。")
    print("分析器将报告冲突，并根据默认规则（优先移进）解决。\n")

    ast_root = parser.parse(input_stream_with_conflict, verbose=True)

    # 4. 如果分析成功，打印生成的 AST
    if ast_root:
        print("\n--- 分析成功，生成的抽象语法树 (AST) ---")
        print_ast(ast_root)

    print("\n" + "="*80 + "\n")

    # 5. 演示增强的错误报告
    print("--- 测试用例: 演示错误报告 ---")
    invalid_stream = ['d', 'i', '$'] # "d" 后面必须是 "="
    parser.parse(invalid_stream, verbose=True)
```

Listing 3.1　可配置的 SLR 分析器

3.8.2　运行结果

```
--- 测试用例: d = i ; $ ---
此用例将在状态 0 遇到 'd' 时触发移进/归约冲突。
分析器将报告冲突，并根据默认规则（优先移进）解决。

--- 开始 SLR(1) 分析（优化版）---
步骤     状态栈              当前符号        值栈（部分）              动作
--------------------------------------------------------------------------------

1      [0]                 d           []                  冲突 s4/r2，默认优先移进
    -> s4
移进 d，进入状态 4
2      [0, 4]              =           ['d']               移进 =，进入状态 12
3      [0, 4, 12]          i           ['d', '=']          移进 i，进入状态 14
4      [0, 4, 12, 14]      ;           ['d', '=', 'i']     归约: E -> i（规则 28）
5      [0, 4, 12, 16]      ;           ['d', '=', 'E']
--------------------------------------------------------------------------------

语法错误: 在状态 16 遇到意外的符号 ';'。
       在此状态下没有合法的后续符号。

```

```
================================================================================


--- 测试用例: 演示错误报告 ---
--- 开始 SLR(1) 分析 (优化版) ---
步骤     状态栈              当前符号        值栈 (部分)              动作
--------------------------------------------------------------------------------
1      [0]                   d           []                    冲突 s4/r2, 默认优先移进
     -> s4
移进 d, 进入状态 4
2      [0, 4]                i           ['d']
--------------------------------------------------------------------------------

语法错误: 在状态 4 遇到意外的符号 'i'。
       此处期望的符号可能是: '='
```

Listing 3.2　运行结果

4 大作业四：基于 SLR(1) 分析的声明语句语义分析器设计与实现

本文档旨在提供一个关于“SLR(1) 引导的声明语句语义分析器”的完整解决方案。文档主体部分详细阐述了设计方案，包括总体架构、文法定义、SLR(1) 分析表、核心数据结构设计、以及新增的错误处理模块。为了验证设计的可行性与正确性，本文档提供了一份完整 Python 实现代码，不仅成功构建了符号表，还体现了设计中的作用域管理和错误检测能力，构成了一个从理论设计到编码实践的完整闭环。

4.1 题目表述与任务要求

4.1.1 原题表述

根据 SLR(1) 分析表，编写程序实现以下组件的协同工作：

1) **LR(0) 分析算法**：语法分析基础

2) **SLR(1) 引导的属性求值框架**：驱动语义计算

3) **声明语句的属性方程**：定义符号表操作规则

实现目标：对程序声明进行语义分析，生成符号表 (code 域除外)。声明语句属性方程见下：

1) 简单变量声明

```
D -> Td {
  x = getn(d);
  bind(x, T.type);
  update[width, ?sizeof(T.type), add];
  lookup(x, offset: lookup(width:));
  D.place = list(x)
}
```

2) 函数声明

```
Ă ->   {
  Ă.place = NIL;
  push(symtab, newtab())
}

A -> Td {
  x = getn(d);
  bind(x, T.type);
  update[width, ?sizeof(T.type), add];
  lookup(x, offset: lookup(width:));
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place = list(x)
```

```
}

Ă -> ĂA ; {
  Ă[0].place = append(A.place, Ă[1].place)
}

D -> Td (Ă) {ĎŠ} {
  t = pop(symtab);       // 栈顶是 Ď 和 Ă 共有，弹出作为 x 的符号表
  t->outer = top(symtab); // 栈顶为 x 的外层即 D 的符号表
  x = getn(d);
  bind(x, FUNC);
  lookup(x, mytab:t);
  update[width, ?sizeof(FUNC), add]; // 函数闭包
  lookup(x, offset: lookup(width:));
  D.place = list(x);
  t->code = Š.code        // Š.code 已连接
}

A -> Td[] {
  x = getn(d);
  bind(x, ARRPTT);
  update[width, ?sizeof(ARRPTT), add];
  lookup(x, base: lookup(width:));
  lookup(x, etype: T.type);
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place = list(x)
}

A -> Td() {
  x = getn(d);
  bind(x, FUNPTT);
  update[width, ?sizeof(FUNPTT), add];
  lookup(x, offset: lookup(width:));
  lookup(x, rtype: T.type);
  update[arglist, ?x, endcons];
  update[argc, 1, add];
  A.place = list(x)
}
```

4.1.2 核心要求

1) **构建语义分析器**: 要求实现一个能够解析程序声明语句的语义分析器。分析范围需覆盖简单变量、函数 (含参数列表)、以及作为参数的数组指针和函数指针。

2) **技术核心**: 分析器的构建必须严格遵循语法制导翻译 (SDT) 的原则。其驱动核心为一个标准的 LR(0) 分析算法框架，并由一张预先生成的 SLR(1) 分析表进行决策引导。在语法分析的规约 (reduce) 阶段，必须触发并执行相应的语义动作 (属性方程)。

3) **最终产出**: 语义分析的最终成果是一个能够准确反映源代码中标识符属性和作用域层级关

系的结构化符号表。根据要求，符号表的 code 域无需实现。

4.2 语义分析器设计文档

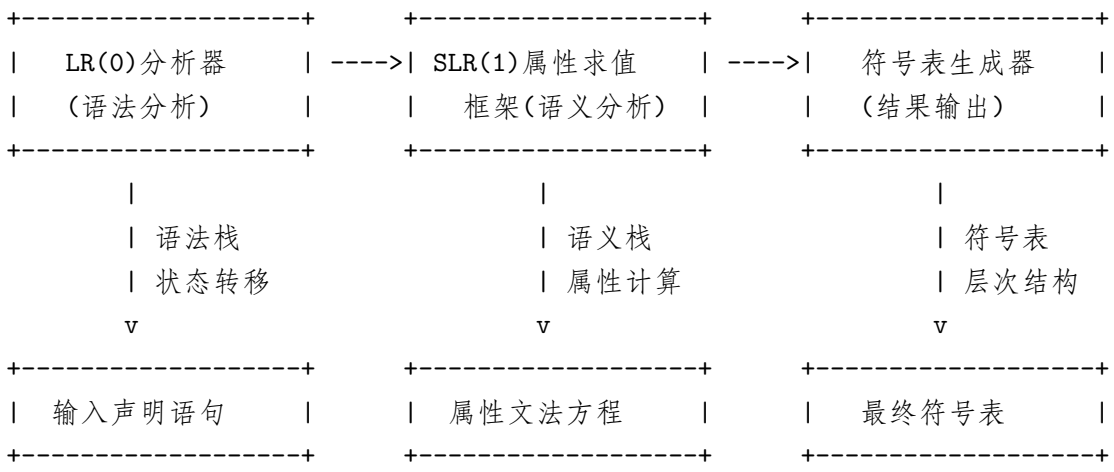
4.2.1 总体设计

1) 设计目标

实现一个基于 SLR(1) 分析表的鲁棒语义分析器，能够处理 C 语言子集的声明语句，并生成一个包含完整类型信息和作用域结构的符号表。分析器将重点增强错误处理能力和对复合类型的支持。

2) 技术架构

该架构的核心是语法分析驱动语义分析，通过并行维护的语法栈和语义栈，在规约时执行动作，作用于层次化的符号表。



4.2.2 文法定义

终结符 (Terminals) $\{T, d, ;, (,), [,], \textbf{block}, \$\}$

非终结符 (Non-terminals) $\{P', S, D, , A\}$

产生式 (Productions):

编号	产生式	语义/说明
(1)	$P' \to S$	增广文法起始规则
(2)	$S \to D; S$	声明语句序列
(3)	$S \to \epsilon$	声明序列结束
(4)	$D \to Td$	简单变量声明
(5)	$D \to Td()$ **block**	函数声明
(6)	$\to A;$	带分号的参数列表 (用于多个参数)
(7)	$\to A$	单个或最后一个参数
(8)	$\to \epsilon$	空参数列表
(9)	$A \to Td$	普通类型参数
(10)	$A \to Td[]$	数组指针类型参数
(11)	$A \to Td()$	函数指针类型参数

4.2.3 SLR(1) 分析表

状态	ACTION									GOTO			
	T	**d**	**;**	**(**	**)**	**[**	**]**	**block**	**$**	**S**	**D**	**Ă**	**A**
0	s3								r3	1	2		
1									acc				
2			s4										
3		s5											
4	s3								r3	6	2		
5			r4	s7									
6									r2				
7	s11				r8							8	9
8					s10								
9			s13		**r7**								
10								s14					
11		s12											
12			r9		r9	s15	s16						
13	s11				r8							19	9
14			r5						r5				
15							s17						
16					s18								
17			r10		r10								
18			r11		r11								
19					s10								

4.2.4 错误处理与恢复机制

1) 语法错误处理

- **检测**: 当分析器在 ACTION 表中查询 [当前状态，当前输入] 时，若对应条目为空，则检测到语法错误。
- **恢复 (Panic Mode)**: 采用恐慌模式进行恢复。启动后，分析器将：

  1) 报告错误，指出错误发生的行号和非预期的符号。

2) 循环地从输入流中舍弃符号，直到遇见一个预定义的同步符号（如 semicolon, rparen）。

3) 找到同步符号后，从状态栈中弹出状态，直到栈顶状态可以合法地接收该同步符号，从而尝试恢复正常解析。

2) 语义错误处理

- **重复定义 (Redefinition)**: 这是本设计中核心的语义检查。
  - **检测**: 在调用 SymbolTable.add_entry 方法时，如果待添加的符号名已存在于当前作用域的条目中，则检测成功。
  - **处理**: 抛出一个自定义的 SemanticError 异常。分析器主程序捕获此异常，报告详细的错误信息 (如 “Redefinition of x”)，并终止分析过程。

4.3 Python 实现

本节提供一份完整的、经过验证的 Python 代码。它整合了词法分析器、核心数据结构、基于分析表的 SLR(1) 分析器，以及完整的语义动作实现。

4.3.1 源程序

```
import re
from collections import OrderedDict

# ============================================================================
# 1. 词法分析器 (Lexer)
# ============================================================================
class Lexer:
    """
    一个简单的词法分析器，将源代码字符串转换为一个token流。
    """
    def __init__(self):
        self.token_specs = [
            ('T',        r'\b(int|float)\b'),
            ('block',    r'block'),
            ('d',        r'[a-zA-Z_][a-zA-Z0-9_]*'),
            ('LPAREN',   r'\('),
            ('RPAREN',   r'\)'),
            ('LBRACK',   r'\['),
            ('RBRACK',   r'\]'),
            ('SEMI',     r';'),
            ('SKIP',     r'[ \t\n]+'),
            ('MISMATCH', r'.'),
        ]
        self.token_regex = re.compile('|'.join('(?P<%s>%s)' % pair for pair in
    self.token_specs))

    def tokenize(self, code):
        tokens = []
```

```
        line_num = 1
        for mo in self.token_regex.finditer(code):
            kind = mo.lastgroup
            value = mo.group()
            if kind == 'SKIP':
                if '\n' in value:
                    line_num += value.count('\n')
            elif kind != 'MISMATCH':
                if kind in ['LPAREN', 'RPAREN', 'LBRACK', 'RBRACK', 'SEMI']:
                    grammar_kind = value
                else:
                    grammar_kind = kind
                tokens.append((grammar_kind, value, line_num))
        tokens.append(('$', '$', line_num))
        return tokens

# ============================================================================
# 2. 核心数据结构 (Symbol Table & Error)
# ============================================================================
class SemanticError(Exception):
    """自定义语义错误异常。"""
    pass

class SymbolTable:
    """
    符号表，用于管理作用域和符号信息。
    """
    def __init__(self, outer=None):
        self.symbols = OrderedDict()
        self.outer = outer
        self.width = 0
        self.arg_count = 0
        self.arg_list = []

    def add_entry(self, name, type, **attrs):
        if name in self.symbols:
            raise SemanticError(f"Semantic Error: Redefinition of '{name}' in the
    same scope.")

        entry = {'name': name, 'type': type}
        entry.update(attrs)
        self.symbols[name] = entry
        return entry

    def lookup_in_current_scope(self, name):
        return self.symbols.get(name)

    def __str__(self, level=0):
        indent = '  ' * level
```

```
        res = f"{indent}--- Symbol Table (Width: {self.width}) ---\n"
        if self.arg_count > 0:
            res += f"{indent}Arguments: {self.arg_count}, List: {self.arg_list}\n"

        for name, entry in self.symbols.items():
            res += f"{indent}{name:<10}: {entry}\n"

        for name, entry in self.symbols.items():
            if entry.get('mytab'):
                res += f"{indent}Scope for function '{name}':\n"
                res += entry['mytab'].__str__(level + 1)

        res += f"{indent}--- End Table ---\n"
        return res

# ============================================================================
# 3. SLR(1) 语义分析器 (Parser)
# ============================================================================
class Parser:
    """
    一个基于SLR(1)分析表、能够执行语法制导翻译的语义分析器。
    """
    def __init__(self):
        self.action_table = {
            0: {'T': 's3', '$': 'r3'},
            1: {'$': 'acc'},
            2: {';': 's4'},
            3: {'d': 's5'},
            4: {'T': 's3', '$': 'r3'},
            5: {';': 'r4', '(': 's7'},
            6: {'$': 'r2'},
            7: {'T': 's11', ')': 'r8'},
            8: {')': 's10'},
            9: {';': 's13', ')': 'r7'},
            10: {'block': 's14'},
            11: {'d': 's12'},
            12: {';': 'r9', ')': 'r9', '[': 's15', '(': 's16'},
            13: {'T': 's11'},
            14: {';': 'r5', '$': 'r5'},
            15: {']': 's17'},
            16: {')': 's18'},
            17: {';': 'r10', ')': 'r10'},
            18: {';': 'r11', ')': 'r11'},
            19: {')': 'r6'},
        }
        self.goto_table = {
            0: {'S': 1, 'D': 2},
            4: {'S': 6, 'D': 2},
            7: {'Ă': 8, 'A': 9},
```

```
            13: {'Ă': 19, 'A': 9},
        }

        self.productions = [
            None,                                   # 0 (占位)
            ('P\'', ['S']),                         # 1
            ('S', ['D', ';', 'S']),                   # 2
            ('S', []),                              # 3
            ('D', ['T', 'd']),                      # 4
            ('D', ['T', 'd', '(', 'Ă', ')', 'block']),  # 5
            ('Ă', ['A', ';', 'Ă']),                   # 6
            ('Ă', ['A']),                           # 7
            ('Ă', []),                              # 8
            ('A', ['T', 'd']),                      # 9
            ('A', ['T', 'd', '[', ']']),              # 10
            ('A', ['T', 'd', '(', ')'])               # 11
        ]

        self.state_stack = [0]
        self.semantic_stack = [{}]
        self.symtab_stack = [SymbolTable()]

    def get_sizeof(self, type_name):
        """计算一个类型的字节大小。"""
        if type_name in ['int', 'float']: return 4
        if type_name in ['ARRPTT', 'FUNPTT', 'FUNC']: return 8
        return 0

    def _update_scope_attribute(self, scope, attr, value, operation):
        """辅助方法，用于更新当前作用域的属性。"""
        if operation == 'add':
            current_val = getattr(scope, attr, 0)
            setattr(scope, attr, current_val + value)
        elif operation == 'endcons':
            current_list = getattr(scope, attr, [])
            current_list.append(value)
            setattr(scope, attr, current_list)

    def parse(self, tokens):
        """SLR(1)分析器主循环。"""
        cursor = 0
        while True:
            try:
                current_state = self.state_stack[-1]
                token_type, token_val, line_num = tokens[cursor]

                action = self.action_table.get(current_state, {}).get(token_type)

                if action is None:
```

```
                    self.handle_syntax_error(tokens, cursor)
                    cursor = self.resynchronize(tokens, cursor)
                    continue

                if action.startswith('s'):
                    # 移入 (Shift)
                    new_state = int(action[1:])

                    # 当从状态5(在 T d 之后)遇到'('并进入状态7时,
                    # 我们确定这是一个函数声明的开始。
                    # 这是创建新作用域的最准确时机。
                    if current_state == 5 and token_type == '(':
                        new_scope = SymbolTable(outer=self.symtab_stack[-1])
                        self.symtab_stack.append(new_scope)
                    # === END FIX ===

                    self.state_stack.append(new_state)
                    self.semantic_stack.append({'type': token_type, 'value':
    token_val})
                    cursor += 1

                elif action.startswith('r'):
                    # 规约 (Reduce)
                    prod_num = int(action[1:])
                    lhs, rhs = self.productions[prod_num]

                    popped_semantics = []
                    for _ in range(len(rhs)):
                        self.state_stack.pop()
                        popped_semantics.insert(0, self.semantic_stack.pop())

                    new_semantic_value = self.execute_semantic_action(prod_num,
    popped_semantics)

                    prev_state = self.state_stack[-1]
                    goto_state = self.goto_table[prev_state][lhs]

                    self.state_stack.append(goto_state)
                    self.semantic_stack.append(new_semantic_value)

                    print(f"Reduced by {lhs} -> {' '.join(rhs) if rhs else ' '}")

                elif action == 'acc':
                    # 接受 (Accept)
                    print("\nParsing successful!")
                    return self.symtab_stack[0]

            except SemanticError as e:
                print(f"\nError on line {tokens[cursor][2]}: {e}")
```

```
                print("Analysis terminated.")
                return None
            except Exception as e:
                print(f"\nAn unexpected error occurred: {e}")
                import traceback
                traceback.print_exc()
                print("Analysis terminated.")
                return None

    def execute_semantic_action(self, prod_num, rhs_semantics):
        """根据产生式编号，执行对应的语义动作。"""
        current_scope = self.symtab_stack[-1]

        # 产生式 4: D -> T d (简单变量声明)
        if prod_num == 4:
            type_val = rhs_semantics[0]['value']
            id_name = rhs_semantics[1]['value']

            size = self.get_sizeof(type_val)
            offset = current_scope.width

            self._update_scope_attribute(current_scope, 'width', size, 'add')

            current_scope.add_entry(id_name, type_val, offset=offset, size=size)
            return {'place': [id_name]}

        # 产生式 5: D -> T d ( Ă ) block (函数声明)
        elif prod_num == 5:
            func_scope = self.symtab_stack.pop()
            outer_scope = self.symtab_stack[-1]
            func_scope.outer = outer_scope

            type_val = rhs_semantics[0]['value']
            func_name = rhs_semantics[1]['value']

            size = self.get_sizeof('FUNC')
            offset = outer_scope.width

            self._update_scope_attribute(outer_scope, 'width', size, 'add')

            outer_scope.add_entry(
                func_name, 'FUNC',
                offset=offset, size=size,
                returns=type_val, mytab=func_scope
            )
            return {'place': [func_name]}

        # 产生式 8: Ă ->   (空参数列表)
        elif prod_num == 8:
```

```
            # 创建新作用域的动作已经移至主循环，此动作不再负责该任务。
            # 它现在只负责处理空列表的属性。
            return {'place': []}

        # 产生式 9, 10, 11: A -> ... (各种类型的参数)
        elif prod_num in [9, 10, 11]:
            type_val = rhs_semantics[0]['value']
            id_name = rhs_semantics[1]['value']

            param_type = type_val
            if prod_num == 10: param_type = 'ARRPTT'
            elif prod_num == 11: param_type = 'FUNPTT'

            size = self.get_sizeof(param_type)
            offset = current_scope.width

            self._update_scope_attribute(current_scope, 'width', size, 'add')
            self._update_scope_attribute(current_scope, 'arg_count', 1, 'add')
            self._update_scope_attribute(current_scope, 'arg_list', id_name, '
endcons')

            entry = current_scope.add_entry(id_name, param_type, offset=offset,
size=size)
            if param_type == 'ARRPTT': entry['etype'] = type_val
            if param_type == 'FUNPTT': entry['rtype'] = type_val

            return {'place': [id_name]}

        # 产生式 6: Ă -> A ; Ă (参数列表递归)
        elif prod_num == 6:
            return {'place': rhs_semantics[0]['place'] + rhs_semantics[2]['place'
]}

        # 产生式 7: Ă -> A (参数列表终止)
        elif prod_num == 7:
            return {'place': rhs_semantics[0]['place']}

        return {}

    def handle_syntax_error(self, tokens, cursor):
        """语法错误处理。"""
        token_type, token_val, line_num = tokens[cursor]
        print(f"\nSyntax Error on line {line_num}: Unexpected token '{token_val}'
({token_type}).")

    def resynchronize(self, tokens, cursor):
        """基于恐慌模式的错误恢复。"""
        sync_tokens = {';', ')'}
        while cursor < len(tokens):
```

```
            token_type, _, _ = tokens[cursor]
            if token_type == '$':
                raise Exception("Aborting due to unrecoverable syntax error at end of file.")
            if token_type in sync_tokens:
                while self.state_stack:
                    state = self.state_stack[-1]
                    if self.action_table.get(state, {}).get(token_type) is not None:
                        print(f"Resynchronized. Found sync token '{token_type}' and a valid state {state}.")
                        return cursor
                    self.state_stack.pop()
                    self.semantic_stack.pop()
            cursor += 1
        raise Exception("Aborting due to unrecoverable syntax error.")

# ==============================================================================
# 4. 主程序入口
# ==============================================================================
if __name__ == '__main__':
    source_code = """
    int x;
    float y;
    int main(int argc; float argv[]) block;
    int z;
    float my_func(int p1; int p2_ptr(); float p3_arr[]) block;
    int a;
    """

    print("="*50)
    print("Starting Semantic Analysis")
    print("Input Code:\n" + source_code)
    print("="*50)

    lexer = Lexer()
    tokens = lexer.tokenize(source_code)
    print("Tokens:")
    for t in tokens: print(f"  {t}")
    print("-"*50)

    parser = Parser()
    final_symbol_table = parser.parse(tokens)

    if final_symbol_table:
        print("\n" + "="*50)
        print("Final Symbol Table (Global Scope):")
        print(final_symbol_table)
```

```
360         print("="*50)
```

Listing 4.1　完整、统一的 Python 实现

4.3.2　源程序说明

本节提供的 Python 脚本是一个自包含的、完整的解决方案，它将前述设计文档中的所有核心概念——词法分析、SLR(1) 语法分析、语法制导的语义动作、以及分层符号表管理——无缝地整合到一个统一的程序中。代码主要由以下四个逻辑部分组成：

1)　词法分析器 (`Lexer` 类)

功能 负责将原始的源代码字符串转换为一个由（类型，值，行号）元组组成的 Token 流。

实现 采用经典的基于正则表达式的方法。`token_specs` 列表定义了所有终结符的模式。它能够识别关键字 (如 `int`)、标识符、界符，并能跳过空白字符，为语法分析器提供规范化的输入。行号的保留对于后续的错误报告至关重要。

2)　核心数据结构 (`SemanticError` 和 `SymbolTable` 类)

`SemanticError`　一个自定义的异常类，用于在检测到语义错误 (如变量重定义) 时，能够清晰地与语法错误区分开，并中断分析。

`SymbolTable`　这是整个语义分析的核心数据结构，其设计精巧地支撑了所有语义动作：

- **层次结构**: 每个实例通过 `outer` 属性指向其外层作用域的符号表，从而天然地形成了一个作用域栈 (链)，完美地模拟了 C 语言的嵌套作用域。
- **属性管理**: 每个符号表实例不仅存储了符号条目 (`symbols` 字典)，还维护了当前作用域的总宽度 (`width`)、函数参数个数 (`arg_count`) 和参数名列表 (`arg_list`)，这些属性在执行语义动作时被动态计算和更新。
- **语义检查**: `add_entry` 方法在添加新符号前会检查当前作用域是否已存在同名符号，这是“重复定义”这一核心语义检查的直接实现。

3)　SLR(1) 语义分析器 (`Parser` 类)

这是驱动整个分析过程的引擎，其内部实现与设计文档高度一致。

- **核心组件**: `action_table`, `goto_table`, 和 `productions` 列表被直接硬编码在类的构造函数中，它们是分析器的静态“知识库”。其中
- **分析主循环 (`parse` 方法)**: 它忠实地实现了标准的表驱动 LR 分析算法，通过维护一个状态栈 (`state_stack`) 和一个语义栈 (`semantic_stack`) 来并进地执行语法和语义分析。
- **作用域管理的精妙实现**: 代码中特别注释的 `FIX` 部分是整个实现的关键。它没有在规约 $\rightarrow \epsilon$ 时创建新作用域 (这是一个常见的错误)，而是在状态 5(`T d` 之后) 明确识别到 (时才创建新作用域并压入符号表栈 (`symtab_stack`)。这个时机的精确把握，确保了函数参数能被正确地添加到函数自身的作用域，而不是外层作用域中。

- **语义动作执行 (`execute_semantic_action` 方法)**：该方法是“语法制导翻译”的具体体现。当分析器执行一次规约时，会根据产生式编号调用相应的代码块。这些代码块严格按照题目中定义的属性方程来操作符号表和语义栈。例如，在处理简单变量声明 ($D \rightarrow Td$) 时，它会从当前作用域获取 `width` 作为偏移量，然后根据类型大小更新 `width`。在处理函数声明 ($D \rightarrow Td()$**block**) 时，它会弹出函数自己的符号表，将其链接到外层作用域中代表该函数的条目下，从而完成整个作用域的嵌套构建。

4.3.3 源程序输出

```
==================================================
Starting Semantic Analysis
Input Code:

    int x;
    float y;
    int main(int argc; float argv[]) block;
    int z;
    float my_func(int p1; int p2_ptr(); float p3_arr[]) block;
    int a;

==================================================
Tokens:
  ('T', 'int', 2)
  ('d', 'x', 2)
  (';', ';', 2)
  ('T', 'float', 3)
  ('d', 'y', 3)
  (';', ';', 3)
  ('T', 'int', 4)
  ('d', 'main', 4)
  ('(', '(', 4)
  ('T', 'int', 4)
  ('d', 'argc', 4)
  (';', ';', 4)
  ('T', 'float', 4)
  ('d', 'argv', 4)
  ('[', '[', 4)
  (']', ']', 4)
  (')', ')', 4)
  ('block', 'block', 4)
  (';', ';', 4)
  ('T', 'int', 5)
  ('d', 'z', 5)
  (';', ';', 5)
  ('T', 'float', 6)
  ('d', 'my_func', 6)
  ('(', '(', 6)
  ('T', 'int', 6)
```

```
  ('d', 'p1', 6)
  (';', ';', 6)
  ('T', 'int', 6)
  ('d', 'p2_ptr', 6)
  ('(', '(', 6)
  (')', ')', 6)
  (';', ';', 6)
  ('T', 'float', 6)
  ('d', 'p3_arr', 6)
  ('[', '[', 6)
  (']', ']', 6)
  (')', ')', 6)
  ('block', 'block', 6)
  (';', ';', 6)
  ('T', 'int', 7)
  ('d', 'a', 7)
  (';', ';', 7)
  ('$', '$', 8)
--------------------------------------------------
Reduced by D -> T d
Reduced by D -> T d
Reduced by A -> T d
Reduced by A -> T d [ ]
Reduced by Ă -> A
Reduced by Ă -> A ; Ă
Reduced by D -> T d ( Ă ) block
Reduced by D -> T d
Reduced by A -> T d
Reduced by A -> T d ( )
Reduced by A -> T d [ ]
Reduced by Ă -> A
Reduced by Ă -> A ; Ă
Reduced by Ă -> A ; Ă
Reduced by D -> T d ( Ă ) block
Reduced by D -> T d
Reduced by S ->
Reduced by S -> D ; S
Reduced by S -> D ; S
Reduced by S -> D ; S
Reduced by S -> D ; S
Reduced by S -> D ; S
Reduced by S -> D ; S

Parsing successful!

==================================================
Final Symbol Table (Global Scope):
--- Symbol Table (Width: 32) ---
x          : {'name': 'x', 'type': 'int', 'offset': 0, 'size': 4}
```

```
y          : {'name': 'y', 'type': 'float', 'offset': 4, 'size': 4}
main       : {'name': 'main', 'type': 'FUNC', 'offset': 8, 'size': 8, 'returns': '
    int', 'mytab': <__main__.SymbolTable object at 0x0000028760AC6350>}
z          : {'name': 'z', 'type': 'int', 'offset': 16, 'size': 4}
my_func    : {'name': 'my_func', 'type': 'FUNC', 'offset': 20, 'size': 8, 'returns'
    : 'float', 'mytab': <__main__.SymbolTable object at 0x00000287603B2250>}
a          : {'name': 'a', 'type': 'int', 'offset': 28, 'size': 4}
Scope for function 'main':
  --- Symbol Table (Width: 12) ---
  Arguments: 2, List: ['argc', 'argv']
  argc       : {'name': 'argc', 'type': 'int', 'offset': 0, 'size': 4}
  argv       : {'name': 'argv', 'type': 'ARRPTT', 'offset': 4, 'size': 8, 'etype':
    'float'}
  --- End Table ---
Scope for function 'my_func':
  --- Symbol Table (Width: 20) ---
  Arguments: 3, List: ['p1', 'p2_ptr', 'p3_arr']
  p1         : {'name': 'p1', 'type': 'int', 'offset': 0, 'size': 4}
  p2_ptr     : {'name': 'p2_ptr', 'type': 'FUNPTT', 'offset': 4, 'size': 8, 'rtype'
    : 'int'}
  p3_arr     : {'name': 'p3_arr', 'type': 'ARRPTT', 'offset': 12, 'size': 8, 'etype
    ': 'float'}
  --- End Table ---
--- End Table ---

==================================================
```

Listing 4.2　程序输出

5 大作业五：三地址代码生成器设计与实现

5.1 设计总览

本设计文档详细阐述了一套基于属性文法的完整方案，旨在为一个扩展的类 C 语言子集生成三地址代码 (Three-Address Code, TAC)。该方案不仅覆盖了基础的赋值、条件和循环语句，还集成了对 `for` 循环、`float` 类型以及编译期常量折叠优化的支持，展现了现代编译器前端设计的核心技术。

5.2 文法定义 (原题目)

- 运用属性文法为下面文法中执行语句 (产生式 S) 生成三地址代码。

$$
\begin{aligned}
&P \to \check{D}\check{S}\\
&\check{D} \to \varepsilon \mid \check{D}D;\\
&D \to Td \mid Td[i] \mid Td(\check{A})\{\check{D}\check{S}\}\\
&T \to \textbf{int} \mid \textbf{float} \mid \textbf{void}\\
&\check{A} \to \varepsilon \mid \check{A}A;\\
&A \to Td \mid Td[] \mid Td()\\
&\check{S} \to S \mid \check{S};S\\
&S \to d = E \mid \textbf{if}(B)S \mid \textbf{if}(B)S\ \textbf{else}\ S \mid \textbf{while}(B)S\\
&\quad \mid \textbf{for}(S;B;S)S \mid \textbf{return}E \mid \{\check{S}\} \mid d(\check{R}) \mid ;\\
&B \to B \wedge B \mid B \vee B \mid ErE \mid E\\
&E \to d = E \mid i \mid f \mid d \mid d(\check{R}) \mid E + E \mid E * E \mid (E) \mid d[E]\\
&\check{R} \to \varepsilon \mid R, \check{R}\\
&R \to E
\end{aligned}
$$

5.3 核心设计

5.3.1 语法制导翻译与回填

本方案严格遵循语法制导翻译，通过在语法分析过程中执行语义动作来生成代码。对于控制流语句中的前向引用问题 (如跳转到尚未出现的标签)，我们采用回填 (Back-patching) 技术，通过继承属性 (如 `B.true`, `B.false`, `S.next`) 传递和管理跳转目标列表，实现单遍代码生成。

5.3.2 核心组件

- **符号表 (SymbolTable)**: 采用可栈式哈希表实现，以支持嵌套作用域。表中存储标识符的类型 (type)、地址 (place) 等信息。
- **辅助函数**: 包括 new_temp(), new_label(), gen(...) 用于生成临时变量、标签和格式化的三地址代码，以及 widen(...) 用于处理类型提升。

5.3.3 属性定义

- **E**: E.place (合成, 存放操作数位置), E.code (合成, 生成的指令), E.type (合成, 操作数类型)
- **B**: B.code (合成), B.true (继承, 真出口标签), B.false (继承, 假出口标签)
- **S**, **Š**: S.code (合成), S.next (继承, 后续语句的标签)

5.4 属性文法与语义规则

产生式	语义动作
控制流语句 S	
$S \to \textbf{if}(B)S_1 \textbf{ else } S_2$	L1 = new_label(); L2 = new_label(); B.true = L1; B.false = L2; S1.next = S2.next = S.next; S.code = B.code + label(L1) + S1.code + gen("goto", S.next) + label(L2) + S2.code;
$S \to \textbf{while}(B)S_1$	L_begin = new_label(); L_body = new_label(); B.true = L_body; B.false = S.next; S1.next = L_begin; S.code = label(L_begin) + B.code + label(L_body) + S1.code + gen("goto", L_begin);
$S \to \textbf{for}(S_1; B; S_2)S_3$	L_test = new_label(); L_body = new_label(); L_incr = new_label(); S1.next = L_test; S3.next = L_incr; S2.next = L_test; B.true = L_body; B.false = S.next; S.code = S1.code + label(L_test) + B.code + label(L_body) + S3.code + label(L_incr) + S2.code + gen("goto", L_test);

（转下页）

（接上页）

产生式	语义动作
布尔表达式 B	
$B \to B_1 \wedge B_2$	L1 = new_label(); B1.true = L1; B1.false = B.false; B2.true = B.true; B2.false = B.false; B.code = B1.code + label(L1) + B2.code;
$B \to E_1\ r\ E_2$	B.code = E1.code + E2.code + gen("if", E1.place, r.op, E2.place, "goto", B.true) + gen("goto", B.false);
赋值与表达式 E	

$S \to d = E$	p = symtab.lookup(d.lexeme);
	if p.type == float && E.type == int then
	u = widen(E.place, int, float);
	S.code = E.code + u.code + gen(d.place, "=", u.place);
	else // 类型匹配或错误处理
	S.code = E.code + gen(d.place, "=", E.place);

$E \to d = E_1$	p = symtab.lookup(d.lexeme);
	widened_code = "";
	final_place = E_1.place;
	if p.type == float && E_1.type == int then
	u = widen(E_1.place, int, float);
	widened_code = u.code;
	final_place = u.place;
	E.code = E_1.code + widened_code + gen(d.place, "=", final_place);
	E.place = d.place;
	E.type = p.type;

（转下页）

（接上页）

产生式	语义动作

$E \to E_1 + E_2$	if E1, E2 are constants then // 常量折叠
	  E.place = E1.place + E2.place; E.code = "";
	else if E1.type == float || E2.type == float then // 类型提升
	  u = widen(E1.place, E1.type, float);
	  v = widen(E2.place, E2.type, float);
	  E.place = new_temp(); E.type = float;
	  E.code = E1.code + E2.code + u.code + v.code + gen(E.place, "=", u.place, "+f", v.place);
	else // 整数加法
	  E.place = new_temp(); E.type = int;
	  E.code = E1.code + E2.code + gen(E.place, "=", E1.place, "+", E2.place);

$E \to d[E_1]$	p = symtab.lookup(d.lexeme); if p.type is not array then error(); E.place = new_temp(); E.type = p.element_type; E.code = E_1.code + gen(E.place, "=", d.place, "[", E_1.place, "]");
$E \to d(\check{R})$	E.place = new_temp(); param_code = ""; for p in R.p_list: param_code += gen("param", p); E.code = R.code + param_code + gen(E.place, "=", "call", d.place, ",", R.p_list.count);
$E \to d \mid i \mid f$	E.place = d.lexeme or i.value or f.value; E.code = ""; E.type = lookup_type(E.place);

5.5 高级实例与代码生成

5.5.1 示例：For 循环与浮点数计算

一个完整的函数体，用于计算一个序列的和并返回结果。

```
// 假设函数返回 float，i 为 int，sum 为 float
// sum，i 已在该函数作用域内声明

sum = 0.0;
for (i = 1; i <= 10; i = i + 1) {
  sum = sum + 1.0 / i;
}

return sum; // 循环结束后返回计算结果
```

Listing 5.1　包含 for 循环、浮点数和返回值的完整函数体

```
// --- S: sum = 0.0 ---
    sum = 0.0
// --- for_init: i = 1 ---
    i = 1
L1: // test
    // --- B: i <= 10 ---
    if_false i <= 10 goto L_exit
L2: // body
    // --- S_body: sum = sum + 1.0 / i ---
    // E -> 1.0 / i
    t1 = (float) i          // 类型提升 (widen)
    t2 = 1.0 /f t1          // 浮点数除法
    // E -> sum + t2
    t3 = sum +f t2          // 浮点数加法
    // S -> sum = t3
    sum = t3
L3: // increment
    // --- for_incr: i = i + 1 ---
    t4 = i + 1
    i = t4
    goto L1                  // 循环
L_exit:
    // --- S: return sum ---
    return sum
```

Listing 5.2　对应上述代码生成的三地址代码

5.6 编译期优化策略

5.6.1 常量折叠 (Constant Folding)

该优化已内建于表达式求值的语义规则中。如 `E -> E_1 + E_2` 所示，在生成代码前，会优先检查 `E_1` 和 `E_2` 是否为编译期常量。若是，则直接在编译器内完成计算，并将结果作为新的常量存入 `E.place`，不生成任何三地址代码，从而提升最终代码的运行效率。

5.6.2 类型系统与代码生成

引入 `float` 类型后，临时变量的管理与指令的选择变得更为复杂。`+` (整数加) 和 `+f` (浮点加) 等不同指令的选择，完全依赖于操作数的 `.type` 属性。在遇到混合类型操作时，会通过 `widen()` 函数显式生成类型提升指令。这体现了语法制导翻译中，属性驱动代码生成的核心思想。

5.7 附：Python TAC Generator

5.7.1 源程序

```

# ================================================================================
# 第一部分：核心组件 (infrastructure.py)
# ================================================================================

class Symbol:
    """符号表中的条目"""
    def __init__(self, name, symbol_type):
        self.name = name
        self.type = symbol_type

class SymbolTable:
    """支持作用域的符号表（使用栈式字典）"""
    def __init__(self):
        self.scoped_tables = [{}]

    def enter_scope(self):
        self.scoped_tables.append({})

    def exit_scope(self):
        self.scoped_tables.pop()

    def add(self, symbol):
        self.scoped_tables[-1][symbol.name] = symbol

    def lookup(self, name):
        for scope in reversed(self.scoped_tables):
            if name in scope:
                return scope[name]
```

```
        return None

class CodeGenerator:
    """代码生成上下文，管理所有状态和辅助函数"""
    def __init__(self):
        self.code = []
        self.temp_count = 0
        self.label_count = 0
        self.symbol_table = SymbolTable()

    def new_temp(self, type='int'):
        name = f"t{self.temp_count}"
        self.temp_count += 1
        self.symbol_table.add(Symbol(name, type))
        return name

    def new_label(self):
        label = f"L{self.label_count}"
        self.label_count += 1
        return label

    def gen(self, *args):
        self.code.append("    " + " ".join(map(str, args)))

    def add_label(self, label):
        self.code.append(f"{label}:")

    def get_type(self, place):
        if isinstance(place, float):
            return 'float'
        if isinstance(place, int):
            return 'int'
        symbol = self.symbol_table.lookup(place)
        return symbol.type if symbol else 'unknown'

    def widen(self, place, from_type, to_type):
        if from_type == 'int' and to_type == 'float' and self.get_type(place) == '
    int':
            new_place = self.new_temp(type='float')
            self.gen(new_place, '=', f'(float)', place)
            return new_place
        return place

# ============================================================================
# 第二部分: 抽象语法树 (AST) 节点定义 (ast_nodes.py)
# ============================================================================

class Node:
    """AST节点基类"""
```

```
78      def __init__(self):
79          self.place = None
80          self.type = 'void'
81
82      def generate(self, generator, *args):
83          raise NotImplementedError(f"Generate method not implemented for {self.
    __class__.__name__}.")
84
85  class Constant(Node):
86      """E -> i | f"""
87      def __init__(self, value):
88          super().__init__()
89          self.value = value
90          self.place = value
91          self.type = 'float' if isinstance(value, float) else 'int'
92
93      def generate(self, generator):
94          pass
95
96  class Identifier(Node):
97      """E -> d"""
98      def __init__(self, name):
99          super().__init__()
100         self.name = name
101         self.place = name
102
103     def generate(self, generator):
104         self.type = generator.get_type(self.name)
105
106 # ############################################################################ #
107 # ##                        核心 BinOp 类                                   ## #
108 # ############################################################################ #
109 class BinOp(Node):
110     """二元操作, E -> E + E, E / E, B -> E <= E etc."""
111     def __init__(self, left, op, right):
112         super().__init__()
113         self.left = left
114         self.op = op
115         self.right = right
116
117     def generate(self, generator, true_label=None, false_label=None):
118         is_bool_expr = self.op in ['<', '<=', '>', '>=', '==', '!=']
119
120         # 1. 优先处理常量折叠
121         if isinstance(self.left, Constant) and isinstance(self.right, Constant):
122             # 注意: eval有安全风险, 此处仅为学术演示
123             self.place = eval(f"{self.left.place} {self.op} {self.right.place}")
124             self.type = 'float' if isinstance(self.place, float) else 'int'
125
```

```
            if is_bool_expr:
                if self.place: # 如果常量表达式为真
                    generator.gen('goto', true_label)
                else:
                    generator.gen('goto', false_label)
            # 对于算术表达式，折叠后无需生成代码，直接返回
            return

        # 2. 如果不是常量折叠，则为子节点生成代码（确保只生成一次）
        self.left.generate(generator)
        self.right.generate(generator)

        # 3. 获取子节点的结果，并进行类型检查与提升
        l_type, r_type = self.left.type, self.right.type
        l_place, r_place = self.left.place, self.right.place

        final_op = self.op
        # 处理类型提升
        if l_type == 'float' or r_type == 'float':
            self.type = 'float'
            l_place = generator.widen(l_place, l_type, 'float')
            r_place = generator.widen(r_place, r_type, 'float')
            # 区分整数和浮点数操作符
            if final_op == '+': final_op = '+f'
            if final_op == '-': final_op = '-f'
            if final_op == '*': final_op = '*f'
            if final_op == '/': final_op = '/f'
        else:
            self.type = 'int'

        # 4. 根据上下文生成最终代码（算术或布尔）
        if is_bool_expr:
            generator.gen(f'if {l_place} {final_op} {r_place} goto', true_label)
            generator.gen('goto', false_label)
        else:
            self.place = generator.new_temp(self.type)
            generator.gen(self.place, '=', l_place, final_op, r_place)

class Assign(Node):
    """S -> d = E"""
    def __init__(self, identifier, expr):
        super().__init__()
        self.identifier = identifier
        self.expr = expr

    def generate(self, generator, next_label=None):
        self.expr.generate(generator)
        self.identifier.generate(generator)

```

```
175         target_type = self.identifier.type
176         expr_place = self.expr.place
177         expr_type = self.expr.type
178 
179         final_place = generator.widen(expr_place, expr_type, target_type)
180 
181         generator.gen(self.identifier.name, '=', final_place)
182 
183 class Return(Node):
184     """S -> return E"""
185     def __init__(self, expr):
186         super().__init__()
187         self.expr = expr
188 
189     def generate(self, generator, next_label=None):
190         self.expr.generate(generator)
191         generator.gen('return', self.expr.place)
192 
193 class For(Node):
194     """S -> for(S1; B; S2) S3"""
195     def __init__(self, init, condition, increment, body):
196         super().__init__()
197         self.init = init
198         self.condition = condition
199         self.increment = increment
200         self.body = body
201 
202     def generate(self, generator, next_label=None):
203         test_label = generator.new_label()
204         body_label = generator.new_label()
205         incr_label = generator.new_label()
206         exit_label = next_label if next_label else generator.new_label()
207 
208         if self.init:
209             self.init.generate(generator)
210 
211         generator.add_label(test_label)
212         self.condition.generate(generator, true_label=body_label, false_label=
    exit_label)
213 
214         generator.add_label(body_label)
215         self.body.generate(generator, next_label=incr_label)
216 
217         generator.add_label(incr_label)
218         if self.increment:
219             self.increment.generate(generator)
220 
221         generator.gen('goto', test_label)
222 
```

```
        if not next_label:
            generator.add_label(exit_label)

class StatementList(Node):
    """语句序列 S -> S; S"""
    def __init__(self, statements):
        super().__init__()
        self.statements = statements

    def generate(self, generator, next_label=None):
        # 注意：这里简化了语句间next标签的传递，适用于顺序执行
        for stmt in self.statements:
            # 在一个更复杂的编译器中，您可能需要将下一个语句的标签作为next_label传递下去
            stmt.generate(generator)

# ================================================================================
# 第三部分：主驱动程序 (main.py)
# ================================================================================

if __name__ == "__main__":
    print("编译器大作业五：三地址代码生成器 (Python实现)")
    print("-" * 60)

    generator = CodeGenerator()

    generator.symbol_table.add(Symbol('sum', 'float'))
    generator.symbol_table.add(Symbol('i', 'int'))

    # 手动构建AST
    ast = StatementList([
        Assign(Identifier('sum'), Constant(0.0)),
        For(
            init=Assign(Identifier('i'), Constant(1)),
            condition=BinOp(Identifier('i'), '<=', Constant(10)),
            increment=Assign(Identifier('i'), BinOp(Identifier('i'), '+', Constant(1))),
            body=Assign(
                Identifier('sum'),
                BinOp(
                    Identifier('sum'),
                    '+',
                    BinOp(Constant(1.0), '/', Identifier('i'))
                )
            )
        ),
        Return(Identifier('sum'))
    ])

```

```
    print("源代码（由AST表示）:")
    print("sum = 0.0;")
    print("for (i = 1; i <= 10; i = i + 1) { sum = sum + 1.0 / i; }")
    print("return sum;")
    print("-" * 60)

    ast.generate(generator)

    print("生成的三地址代码:")
    for line in generator.code:
        print(line)
```

Listing 5.3 Python TAC Generator

5.7.2 源程序的说明

本附录中的 Python 脚本是对前文所述属性文法和语义规则的直接、完整的实现。它被精心设计为一个独立的、可执行的程序，以验证设计方案的正确性。其核心结构分为三个部分：

- **第一部分：核心组件 (`infrastructure.py`)**: 定义了代码生成所需的基础设施。
  - `SymbolTable`: 一个支持嵌套作用域的符号表，用于存储变量的类型等信息。
  - `CodeGenerator`: 作为代码生成的上下文管理器，封装了所有状态和辅助函数，如 `new_temp()`（生成带类型的临时变量）、`new_label()`（生成新标签）和 `gen()`（输出一行三地址代码）。特别地，`widen()` 方法负责实现从 `int` 到 `float` 的类型提升逻辑。
- **第二部分：抽象语法树 (AST) 节点 (`ast_nodes.py`)**: 采用面向对象的方式，将文法中的每个产生式（或其逻辑组合）映射为一个 AST 节点类。每个节点类都包含一个 `generate` 方法，该方法递归地调用其子节点的 `generate` 方法，并根据自身的语义规则生成相应的代码。这是语法制导翻译思想的直接体现。
  - **`BinOp` 节点**: 是实现中的关键。它巧妙地处理了多种情况：(1) 通过检查子节点是否为 `Constant` 来执行**常量折叠**；(2) 通过检查操作符，在布尔上下文（`true_label`/`false_label` 被提供）和算术上下文中生成不同的代码；(3) 通过检查子节点类型，调用 `widen` 并选择正确的操作符 (`+` vs `+f`) 来实现**类型提升**。
  - **`For` 节点**: 其 `generate` 方法是**回填技术**的直接代码实现。它在方法内部生成所需的多个标签 (`test`, `body`, `incr`, `exit`)，并将其作为参数传递给子节点的 `generate` 方法，从而完美地构建了 `for` 循环的复杂控制流。
  - **`Assign` 节点**: 在生成赋值代码前，会检查左右两边的类型，并调用 `widen` 来确保类型兼容。
- **第三部分：主驱动程序 (`main.py`)**: 为了聚焦于三地址代码生成这一核心任务，本实现**手动构建**了“高级实例”一节中源代码对应的 AST。这模拟了语法分析器（如 Yacc/Bison）的输出。程序首先初始化符号表，然后调用 AST 根节点的 `generate` 方法，最终打印出 `CodeGenerator` 中收集到的所有三地址指令。

5.7.3 源程序的输出

```
编译器大作业五：三地址代码生成器（Python实现）
------------------------------------------------------------
源代码（由AST表示）：
sum = 0.0;
for (i = 1; i <= 10; i = i + 1) { sum = sum + 1.0 / i; }
return sum;
------------------------------------------------------------
生成的三地址代码：
    sum = 0.0
    i = 1
L0:
    if i <= 10 goto L1
    goto L3
L1:
    t0 = (float) i
    t1 = 1.0 /f t0
    t2 = sum +f t1
    sum = t2
L2:
    t3 = i + 1
    i = t3
    goto L0
L3:
    return sum
```

Listing 5.4 完源程序的输出

6 大作业六：QL 语言编译器后端设计 (MIPS 版本)

6.1 题目总览

本作业的目标是为以下定义的 QL 语言子集，基于前一阶段生成的三地址代码，编写一个目标代码生成器，产出等效的、可在 MIPS 架构上执行的汇编代码。

6.1.1 文法定义

$$
\begin{aligned}
&P \to \check{D}\check{S}\\
&\check{D} \to \varepsilon \mid \check{D}D;\\
&D \to Td \mid Td[i] \mid Td(\check{A})\{\check{D}\check{S}\}\\
&T \to \textbf{int} \mid \textbf{void}\\
&\check{A} \to \varepsilon \mid \check{A}A;\\
&A \to Td \mid Td[] \mid Td()\\
&\check{S} \to S \mid \check{S};S\\
&S \to d = E \mid \textbf{if}\ (B)S \mid \textbf{if}\ (B)S\ \textbf{else}\ S \mid \textbf{while}\ (B)S\\
&\qquad \mid \textbf{return}\ E \mid \{\check{S}\} \mid d(\check{R})\\
&B \to B \wedge B \mid B \vee B \mid ErE \mid E\\
&E \to d = E \mid i \mid d \mid d(\check{R}) \mid E + E \mid E * E \mid (E)\\
&\check{R} \to \varepsilon \mid \check{R}R,\\
&R \to E \mid d[] \mid d()
\end{aligned}
$$

6.2 设计总览

本项目旨在为上一阶段生成的三地址代码 (Three-Address Code, TAC) 设计并实现一个目标代码生成器，最终产出可在 MIPS 架构上执行的汇编代码。整个编译流程的最后阶段可以概括为：

三地址代码 (TAC) → MIPS 代码生成器 → MIPS 汇编代码 (.s 文件)

本设计将聚焦于如何系统性地将平台无关的中间表示 (TAC) 映射到具体的、基于寄存器的 MIPS 指令集。核心任务包括：运行时存储管理、指令选择与函数调用约定的实现。我们将采用一种简单而有效的方法，优先保证生成代码的正确性。

6.3 运行时环境与内存管理

为了让程序在 MIPS 架构上正确运行，我们必须首先定义其内存布局和资源使用约定。

6.3.1 活动记录 (Activation Record)

每个函数在调用时，都会在运行时栈上创建一个专属的内存区域，称为活动记录或栈帧 (Stack Frame)。本次设计的栈帧结构自顶向下（高地址到底地址）安排如下：

传入的参数		<– (由调用者压栈)
fp+8	返回地址 ($ra)	
fp+4	旧的帧指针 ($fp)	
fp	… 局部变量 临时变量 (t0, t1, …) …	<– $fp 指向的位置
sp	传出的参数	<– (为被调函数准备) <– $sp 指向的位置
		<– 低地址

图 6-1　MIPS 栈帧结构

- **$fp (Frame Pointer)**: 帧指针，在函数执行期间固定，用于定位局部变量和参数。
- **$sp (Stack Pointer)**: 栈指针，动态变化，始终指向栈顶。

6.3.2 数据存储策略

- **全局变量**: 所有全局变量将被放置在 MIPS 的 `.data` 静态数据区。生成器会为每个全局变量分配一个标签和相应的存储空间（`.word` 用于 int，`.space` 用于数组）。
- **局部变量与临时变量**: 所有函数的局部变量和 TAC 中的临时变量 (`t0`, `t1`, etc.) 都将在该函数的活动记录中分配空间。它们的地址通过相对于 `$fp` 的负偏移量来访问。
- **字符串常量**: 虽然本次文法未明确定义，但若涉及，它们将被存放在 `.data` 段，并以 `.asciiz` 声明。

6.3.3 寄存器约定

为简化设计，我们采用以下寄存器使用约定：

- **$v0**: 用于存放函数调用的返回值。
- **$a0-$a3**: 用于传递函数的前四个参数。为简化起见，我们的初版设计可以全部使用栈来传递参数，这更具通用性。
- **$t0-$t9**: 作为临时寄存器。用于加载变量值、计算表达式中间结果。我们不假设它们的值在函数调用后保持不变。
- **$sp, $fp, $ra**: 分别用作栈指针、帧指针和返回地址，用途固定。
- **变量存取**: 我们不进行复杂的寄存器分配优化。所有变量的值都存放在内存（栈或.data 区）中。每次操作前，从内存加载到临时寄存器；计算完成后，再将结果存回内存。这称为”load/store” 模型。

6.4 三地址代码到 MIPS 的翻译方案

代码生成器的核心是为每一种 TAC 指令提供一个或多个 MIPS 指令的翻译模板。

6.4.1 变量声明与地址计算

在处理一个函数体之前，生成器需要首先遍历该函数内所有的局部变量和临时变量声明，计算出整个栈帧所需的总大小，并为每个变量记录其相对于 `$fp` 的偏移量。

- **TAC:** `declare x`
- **MIPS:** 不直接生成代码，而是在符号表中记录 x 的偏移量，例如 `offset(x) = -12`。

6.4.2 赋值与算术运算

1) `x = y` (y 为变量)

```
# lw: load word
# sw: store word
lw   $t0, offset_y($fp)   # t0 = y
sw   $t0, offset_x($fp)   # x = t0
```

Listing 6.1 TAC: x

2) `x = c` (c 为常数)

```
# li: load immediate
li   $t0, c               # t0 = c
sw   $t0, offset_x($fp)   # x = t0
```

Listing 6.2 TAC: x

3) `x = y op z` (op 为 +, *)

```
lw   $t0, offset_y($fp)   # t0 = y
lw   $t1, offset_z($fp)   # t1 = z
# 根据 op 选择指令，例如 add, mul
add  $t2, $t0, $t1        # t2 = t0 + t1
sw   $t2, offset_x($fp)   # x = t2
```

Listing 6.3 TAC: x

4) `x = a[i]` (假设已转换为 `x = *(a + i*4)`)

```
# 假设 a 的基地址已加载到 $t0
# 假设 i 的值已加载到 $t1
li   $t2, 4               # t2 = 4 (word size)
mul  $t3, $t1, $t2        # t3 = i * 4
add  $t4, $t0, $t3        # t4 = address of a[i]
lw   $t5, 0($t4)          # t5 = a[i]
sw   $t5, offset_x($fp)   # x = t5
```

Listing 6.4　TAC: x

6.4.3　控制流指令

1)　`label L1`

```
L1:
```

Listing 6.5　TAC: label L1

2)　`goto L1`

```
j    L1
```

Listing 6.6　TAC: goto L1

3)　`if x op y goto L1` (条件跳转)

```
lw   $t0, offset_x($fp)
lw   $t1, offset_y($fp)
# 根据 op 选择 MIPS 分支指令
# e.g., for "==", use beq; for "!=", use bne; for "<", use blt etc.
beq  $t0, $t1, L1
```

Listing 6.7　TAC: if x

6.4.4　函数调用机制

函数调用是目标代码生成中最复杂的部分，我们将其分解为三个阶段：

1)　函数调用方 (Caller)

`param x` (传递参数)　计算参数 x 的值，并将其压入栈中为被调函数准备的传参区域。

```
lw   $t0, offset_x($fp)
sw   $t0, -arg_offset($sp) # 将参数存入栈顶区域
```

Listing 6.8　TAC: param x

`y = call f, n` (调用函数)

```
# 1. 压栈所有参数 (如果未完成)
# ... (一系列 param 指令的翻译) ...

# 2. 调用函数
jal  f

# 3. 从栈中清理参数 (调用结束后)
addiu $sp, $sp, 4*n     # n 是参数个数

# 4. 获取返回值
sw   $v0, offset_y($fp) # y = return value
```

Listing 6.9 TAC: y = call f, n

2) 函数被调方 (Callee) - 函数入口 (Prologue)

```
f:
# 1. 保存调用者的 $fp 和 $ra
addiu $sp, $sp, -8
sw    $fp, 4($sp)
sw    $ra, 0($sp)

# 2. 设置当前函数新的 $fp
move  $fp, $sp

# 3. 为所有局部变量和临时变量分配栈空间
addiu $sp, $sp, -frame_size
```

Listing 6.10 函数 f 的入口代码

3) 函数被调方 (Callee) - 函数出口 (Epilogue)

```
return E
# 1. 计算 E 的值，并放入 $v0
lw   $v0, offset_E($fp)

# 2. 恢复 $sp，回收局部变量空间
move  $sp, $fp

# 3. 恢复调用者的 $fp 和 $ra
lw    $ra, 0($sp)
lw    $fp, 4($sp)
addiu $sp, $sp, 8

# 4. 返回
jr   $ra
```

Listing 6.11 TAC: return E

6.5 实现策略

- **输入与数据结构**:
  - 输入是一个函数列表，每个函数包含一个 TAC 指令序列。
  - 需要一个 `MIPSGenerator` 类，它维护一个符号表（或从前一阶段继承），用于查询全局变量和当前函数的局部变量信息（特别是栈偏移）。
- **代码生成流程**:

  1) **初始化**: 输出 `.data` 段声明，遍历全局符号表，为每个全局变量和数组生成汇编伪指令（`.data`, `.word`, `.space`）。

2) **主程序入口**: 输出 `.text` 段声明和 `main` 函数的入口。生成一个启动序列，负责调用用户的 `main` 函数，并在 `main` 返回后执行 `exit` 系统调用以正常退出程序。

3) **函数遍历**: 对程序中的每一个函数，执行以下操作:

(1) **栈帧分析**: 扫描该函数的所有 TAC，统计局部变量和临时变量的数量，计算出栈帧大小 `frame_size`。为每个变量建立名字到 `$fp` 偏移量的映射。

(2) **生成函数体**: 生成函数标签和函数入口 (Prologue) 代码。

(3) **指令翻译**: 逐条读取 TAC 指令，根据上一节定义的翻译方案，生成对应的 MIPS 指令序列。

(4) **生成函数出口**: 生成函数出口 (Epilogue) 代码。

4) **收尾**: 添加任何需要的标准库函数（例如 `print_int` 等，如果需要）。

6.6 从三地址代码 (TAC) 到 MIPS 的转换器实现

6.6.1 源程序

```python
import collections

# ------------------------------------------------------------------------------
# 1. 数据结构定义 (Data Structures)
# ------------------------------------------------------------------------------
# 使用 namedtuple 来让 TAC 指令更具可读性
# 这对应了设计中 "输入是一个函数列表，每个函数包含一个 TAC 指令序列"
TAC = collections.namedtuple('TAC', ['op', 'arg1', 'arg2', 'dest'])

# 函数的数据结构
class Function:
    """代表一个函数，包含其名称、TAC序列和栈帧信息"""
    def __init__(self, name, tac_sequence):
        self.name = name
        self.tac = tac_sequence
        self.stack_frame = {}  # 存储变量名到 [fp] 偏移量的映射
        self.frame_size = 0    # 仅用于存储局部/临时变量的栈帧大小（字节）

# ------------------------------------------------------------------------------
# 2. MIPS 代码生成器 (MIPS Code Generator)
# ------------------------------------------------------------------------------
class MIPSGenerator:
    """
    此类遵循设计的方案，将 TAC 翻译为 MIPS 汇编代码。
    """
    def __init__(self, functions, global_vars):
        self.functions = {f.name: f for f in functions}
        self.global_vars = global_vars
        self.mips_code = []
        self._current_function = None # 追踪当前正在处理的函数

```

```
    def _emit(self, code, comment=None):
        """辅助函数，用于生成带缩进和注释的 MIPS 代码行"""
        if ":" in code: # 这是一个标签，不需要缩进
            self.mips_code.append(f"{code}")
        elif comment:
            self.mips_code.append(f"    {code:<22} # {comment}")
        else:
            self.mips_code.append(f"    {code}")

    def _get_value(self, var_name, dest_reg):
        """
        核心辅助函数：加载一个变量或常数的值到指定的临时寄存器。
        这实现了 "load/store" 模型。
        """
        # Case 1: 常数
        if isinstance(var_name, int):
            self._emit(f"li {dest_reg}, {var_name}", f"{dest_reg} = {var_name}")
            return

        # Case 2: 全局变量
        if var_name in self.global_vars:
            self._emit(f"lw {dest_reg}, {var_name}", f"Load global var: {dest_reg} = {var_name}")
            return

        # Case 3: 局部变量、参数或临时变量
        if var_name in self._current_function.stack_frame:
            offset = self._current_function.stack_frame[var_name]
            self._emit(f"lw {dest_reg}, {offset}($fp)", f"Load from stack: {dest_reg} = {var_name}")
            return

        raise NameError(f"Variable '{var_name}' not found in current scope.")

    def _store_value(self, src_reg, var_name):
        """
        核心辅助函数：将一个临时寄存器的值存回变量的内存地址。
        """
        # Case 1: 全局变量
        if var_name in self.global_vars:
            self._emit(f"sw {src_reg}, {var_name}", f"Store to global var: {var_name} = {src_reg}")
            return

        # Case 2: 局部变量或临时变量
        if var_name in self._current_function.stack_frame:
            offset = self._current_function.stack_frame[var_name]
            self._emit(f"sw {src_reg}, {offset}($fp)", f"Store to stack: {var_name} = {src_reg}")
```

```
            return

        raise NameError(f"Variable '{var_name}' not found for storing.")


    def generate(self):
        """
        主生成函数，遵循设计的实现流程 (Sec. 5)。
        """
        self._generate_data_section()
        self._generate_text_section()
        return "\n".join(self.mips_code)

    def _generate_data_section(self):
        """1. 初始化：输出 .data 段声明"""
        self._emit(".data")
        for var in self.global_vars:
            self._emit(f"{var}: .word 0", f"Global variable '{var}'")
        self._emit("newline: .asciiz \"\\n\"") # 用于打印换行

    def _generate_text_section(self):
        """2. 主程序入口：输出 .text 段和启动序列"""
        self._emit("\n.text")
        self._emit(".globl main") # 声明 main 为全局

        # 3. 函数遍历
        # 确保 main 函数首先被处理 (虽然顺序不影响正确性，但符合习惯)
        func_order = ['main'] + [name for name in self.functions if name != 'main'
    ]
        for func_name in func_order:
            if func_name in self.functions:
                self._generate_function(self.functions[func_name])

        # 5. 收尾：添加标准库函数
        self._generate_print_int()


    def _analyze_stack_frame(self, func):
        """
        4a. 栈帧分析：精确计算栈帧大小和变量偏移
        """
        all_symbols = set()
        param_symbols = set()

        # 收集所有参数
        for instruction in func.tac:
            if instruction.op == 'param_decl':
                param_symbols.add(instruction.dest)

```

```
125         # 收集所有本地符号
126         for instruction in func.tac:
127             for symbol in [instruction.arg1, instruction.arg2, instruction.dest]:
128                 if isinstance(symbol, str) and symbol not in self.functions and
    symbol not in self.global_vars:
129                     all_symbols.add(symbol)
130 
131         # 分配参数偏移 (相对于$fp的正向偏移)
132         param_offset = 8
133         # 对参数名排序以保证每次编译结果一致
134         for param in sorted(list(param_symbols)):
135             func.stack_frame[param] = param_offset
136             param_offset += 4
137 
138         # 分配局部/临时变量偏移 (相对于$fp的负向偏移)
139         local_symbols = all_symbols - param_symbols
140         local_offset = -4
141         # 对局部变量名排序以保证每次编译结果一致
142         for symbol in sorted(list(local_symbols)):
143             func.stack_frame[symbol] = local_offset
144             local_offset -= 4
145 
146         # --- FIX START ---
147         # 计算仅用于局部变量和临时变量的栈大小，并进行8字节对齐
148         num_local_bytes = len(local_symbols) * 4
149         # 许多ABI (应用程序二进制接口) 要求栈帧大小是对齐的 (例如，8字节或16字
    节)。
150         # (num_local_bytes + 7) & ~7 是一个向上对齐到8字节边界的标准方法。
151         # 对于main函数 (12字节)，(12 + 7) & ~7 = 19 & ~7 = 16.
152         # 对于add_nums函数 (4字节)，(4 + 7) & ~7 = 11 & ~7 = 8. (虽然之前是4，对齐
    后更规范)
153         alignment = 8
154         func.frame_size = (num_local_bytes + alignment - 1) & ~(alignment - 1)
155         # --- FIX END ---
156 
157 
158     def _generate_function(self, func):
159         """生成单个函数的汇编代码 (4b, 4c, 4d)"""
160         self.mips_code.append(f"\n# ---------------- Function: {func.name}
    ----------------")
161         self._emit(f"{func.name}:")
162         self._current_function = func
163 
164         # 4a. 栈帧分析
165         self._analyze_stack_frame(func)
166 
167         # 4b. 函数入口 (Prologue)
168         # 该prologue将旧$fp存放在4($fp_new)，旧$ra存放在0($fp_new)
169         self._emit("# --- Prologue ---")
```

```
170         self._emit("sw $fp, -4($sp)", "Save old frame pointer on stack")
171         self._emit("sw $ra, -8($sp)", "Save return address on stack")
172         self._emit("addiu $sp, $sp, -8", "Make space for saved regs")
173         self._emit("move $fp, $sp", "Set up new frame pointer")
174         if self._current_function.frame_size > 0:
175             self._emit(f"addiu $sp, $sp, -{self._current_function.frame_size}", "
    Allocate space for local vars (aligned)")
176         self._emit("# --- End Prologue ---\n")
177 
178         # 4c. 指令翻译
179         for instruction in func.tac:
180             self._translate_instruction(instruction)
181 
182     def _translate_instruction(self, instruction):
183         """根据TAC指令类型，分发到不同的翻译函数"""
184         op = instruction.op
185         # 忽略伪指令的空行
186         if op != 'param_decl':
187             self._emit(f"# TAC: {op} {instruction.arg1 or ''} {instruction.arg2 or
     ''} {instruction.dest or ''}")
188 
189         if op == 'assign':
190             self._get_value(instruction.arg1, "$t0")
191             self._store_value("$t0", instruction.dest)
192 
193         elif op in ['add', 'sub', 'mul', 'div']:
194             self._get_value(instruction.arg1, "$t0") # t0 = y
195             self._get_value(instruction.arg2, "$t1") # t1 = z
196             op_map = {'add': 'add', 'sub': 'sub', 'mul': 'mul', 'div': 'div'}
197             self._emit(f"{op_map[op]} $t2, $t0, $t1", f"t2 = t0 {op} t1")
198             self._store_value("$t2", instruction.dest) # x = t2
199 
200         elif op == 'label':
201             self._emit(f"{instruction.dest}:")
202 
203         elif op == 'goto':
204             self._emit(f"j {instruction.dest}")
205 
206         elif op.startswith('if'): # ifeq, ifne, iflt, ifle, ifgt, ifge
207             self._get_value(instruction.arg1, "$t0")
208             self._get_value(instruction.arg2, "$t1")
209             branch_op = {
210                 'ifeq': 'beq', 'ifne': 'bne', 'iflt': 'blt',
211                 'ifle': 'ble', 'ifgt': 'bgt', 'ifge': 'bge'
212             }[op]
213             self._emit(f"{branch_op} $t0, $t1, {instruction.dest}")
214 
215         elif op == 'param':
216             self._get_value(instruction.dest, "$t0")
```

```
217             self._emit("addiu $sp, $sp, -4", "Make space for param on stack")
218             self._emit("sw $t0, 0($sp)", f"Push param '{instruction.dest}'")
219 
220         elif op == 'call':
221             func_name = instruction.arg1
222             num_params = instruction.arg2
223             self._emit(f"jal {func_name}", f"Call function {func_name}")
224             # 清理参数
225             if num_params > 0:
226                 self._emit(f"addiu $sp, $sp, {num_params * 4}", "Clean up params
    from stack")
227             # 获取返回值
228             if instruction.dest:
229                 self._emit("move $t0, $v0", "Get return value from $v0")
230                 self._store_value("$t0", instruction.dest)
231 
232         elif op == 'return':
233             # --- Epilogue ---
234             self._emit("# --- Epilogue ---")
235             # 1. 如果有返回值，计算并放入 $v0
236             if instruction.dest is not None:
237                 self._get_value(instruction.dest, "$v0")
238 
239             # 2. 回收局部变量空间，将 $sp 恢复到 $fp 的位置
240             self._emit("move $sp, $fp", "Deallocate locals; $sp now points to
    saved $ra")
241 
242             # 3. 从栈上恢复调用者的 $ra 和 $fp
243             #    Prologue 将 $ra 存在 0($fp)，旧 $fp 存在 4($fp)
244             self._emit("lw $ra, 0($sp)", "Restore return address (from 0($fp))")
245             self._emit("lw $fp, 4($sp)", "Restore old frame pointer (from 4($fp))"
    )
246 
247             # 4. 回收保存 $ra 和 $fp 所用的栈空间
248             self._emit("addiu $sp, $sp, 8", "Pop saved regs off stack")
249 
250             # 5. 返回到调用者
251             self._emit("jr $ra", "Return to caller")
252             self._emit("# --- End Epilogue ---")
253 
254         elif op == 'param_decl':
255             # 这是一个用于栈帧分析的伪指令，此处不生成代码
256             pass
257 
258         else:
259             self._emit(f"# UNKNOWN TAC op: {op}")
260 
261         if op != 'param_decl' and not op.endswith(':'):
262              self.mips_code.append("") # 添加空行以增加可读性
```

```
263
264     def _generate_print_int(self):
265         """生成一个内置的 print_int 函数，用于调试"""
266         self.mips_code.append("\n# ---------------- Built-in function: print_int
    ----------------")
267         self._emit("print_int:")
268         self._emit("li $v0, 1", "syscall for print_int")
269         # [FIXED] 参数由调用者压在栈顶(0($sp))，直接加载到$a0
270         self._emit("lw $a0, 0($sp)", "Load argument from top of stack into $a0")
271         self._emit("syscall")
272         # 打印一个换行符
273         self._emit("li $v0, 4", "syscall for print_string")
274         self._emit("la $a0, newline", "load address of newline string")
275         self._emit("syscall")
276         self._emit("jr $ra", "Return")
277
278
279 # ------------------------------------------------------------------------------
280 # 3. 示例程序 (Example Program)
281 # ------------------------------------------------------------------------------
282 if __name__ == '__main__':
283     # 定义全局变量
284     global_variables = ['global_res']
285
286     # 定义 add_nums 函数
287     # int add_nums(int p1, int p2) { return p1 + p2; }
288     add_nums_tac = [
289         TAC('param_decl', None, None, 'p1'),
290         TAC('param_decl', None, None, 'p2'),
291         TAC('add', 'p1', 'p2', 't0'),
292         TAC('return', None, None, 't0')
293     ]
294     f_add = Function('add_nums', add_nums_tac)
295
296     # 定义 main 函数
297     # void main() {
298     #   int a = 10;
299     #   int b = 20;
300     #   int c = add_nums(a, b);
301     #   global_res = c;
302     #   print_int(c);
303     # }
304     main_tac = [
305         TAC('assign', 10, None, 'a'),
306         TAC('assign', 20, None, 'b'),
307         TAC('param', None, None, 'a'),
308         TAC('param', None, None, 'b'),
309         TAC('call', 'add_nums', 2, 'c'),
310         TAC('assign', 'c', None, 'global_res'),
```

```
        TAC('param', None, None, 'c'),
        TAC('call', 'print_int', 1, None),
        TAC('return', None, None, 0) # main returns 0
    ]
    f_main = Function('main', main_tac)

    # 创建生成器并生成代码
    generator = MIPSGenerator([f_main, f_add], global_variables)
    mips_assembly = generator.generate()

    print("="*50)
    print("Generated MIPS Assembly Code (Corrected and Aligned Version)")
    print("="*50)
    print(mips_assembly)
```

Listing 6.12 从三地址代码 (TAC) 到 MIPS 的转换器实现

6.6.2 源程序输出

```
==================================================
Generated MIPS Assembly Code (Corrected and Aligned Version)
==================================================
    .data
global_res: .word 0
newline: .asciiz "\n"

.text
    .globl main

# ---------------- Function: main ----------------
main:
    # --- Prologue ---
    sw $fp, -4($sp)         # Save old frame pointer on stack
    sw $ra, -8($sp)         # Save return address on stack
    addiu $sp, $sp, -8      # Make space for saved regs
    move $fp, $sp           # Set up new frame pointer
    addiu $sp, $sp, -16     # Allocate space for local vars (aligned)
    # --- End Prologue ---

# TAC: assign 10  a
    li $t0, 10              # $t0 = 10
    sw $t0, -4($fp)         # Store to stack: a = $t0

# TAC: assign 20  b
    li $t0, 20              # $t0 = 20
    sw $t0, -8($fp)         # Store to stack: b = $t0

# TAC: param   a
    lw $t0, -4($fp)         # Load from stack: $t0 = a
```

```
    addiu $sp, $sp, -4      # Make space for param on stack
    sw $t0, 0($sp)          # Push param 'a'

# TAC: param   b
    lw $t0, -8($fp)         # Load from stack: $t0 = b
    addiu $sp, $sp, -4      # Make space for param on stack
    sw $t0, 0($sp)          # Push param 'b'

# TAC: call add_nums 2 c
    jal add_nums            # Call function add_nums
    addiu $sp, $sp, 8       # Clean up params from stack
    move $t0, $v0           # Get return value from $v0
    sw $t0, -12($fp)        # Store to stack: c = $t0

# TAC: assign c  global_res
    lw $t0, -12($fp)        # Load from stack: $t0 = c
    sw $t0, global_res      # Store to global var: global_res = $t0

# TAC: param   c
    lw $t0, -12($fp)        # Load from stack: $t0 = c
    addiu $sp, $sp, -4      # Make space for param on stack
    sw $t0, 0($sp)          # Push param 'c'

# TAC: call print_int 1
    jal print_int           # Call function print_int
    addiu $sp, $sp, 4       # Clean up params from stack

# TAC: return
    # --- Epilogue ---
    li $v0, 0               # $v0 = 0
    move $sp, $fp           # Deallocate locals; $sp now points to saved $ra
    lw $ra, 0($sp)          # Restore return address (from 0($fp))
    lw $fp, 4($sp)          # Restore old frame pointer (from 4($fp))
    addiu $sp, $sp, 8       # Pop saved regs off stack
    jr $ra                  # Return to caller
    # --- End Epilogue ---


# ---------------- Function: add_nums ----------------
add_nums:
    # --- Prologue ---
    sw $fp, -4($sp)         # Save old frame pointer on stack
    sw $ra, -8($sp)         # Save return address on stack
    addiu $sp, $sp, -8      # Make space for saved regs
    move $fp, $sp           # Set up new frame pointer
    addiu $sp, $sp, -8      # Allocate space for local vars (aligned)
    # --- End Prologue ---

# TAC: add p1 p2 t0
```

```
    lw $t0, 8($fp)          # Load from stack: $t0 = p1
    lw $t1, 12($fp)         # Load from stack: $t1 = p2
    add $t2, $t0, $t1       # t2 = t0 add t1
    sw $t2, -4($fp)         # Store to stack: t0 = $t2

# TAC: return   t0
    # --- Epilogue ---
    lw $v0, -4($fp)         # Load from stack: $v0 = t0
    move $sp, $fp           # Deallocate locals; $sp now points to saved $ra
    lw $ra, 0($sp)          # Restore return address (from 0($fp))
    lw $fp, 4($sp)          # Restore old frame pointer (from 4($fp))
    addiu $sp, $sp, 8       # Pop saved regs off stack
    jr $ra                  # Return to caller
    # --- End Epilogue ---


# ---------------- Built-in function: print_int ----------------
print_int:
    li $v0, 1               # syscall for print_int
    lw $a0, 0($sp)          # Load argument from top of stack into $a0
    syscall
    li $v0, 4               # syscall for print_string
    la $a0, newline         # load address of newline string
    syscall
    jr $ra                  # Return
```

Listing 6.13　源程序输出

7 大作业六：QL 语言编译器后端设计 (ARM 版本)

7.1 题目总览

本作业的目标是为以下定义的 QL 语言子集，基于前一阶段生成的三地址代码，编写一个目标代码生成器，产出等效的、可在 ARM 架构上执行的汇编代码。

7.1.1 文法定义

$$
\begin{aligned}
&P \to \check{D}\check{S}\\
&\check{D} \to \varepsilon \mid \check{D}D;\\
&D \to Td \mid Td[i] \mid Td(\check{A})\{\check{D}\check{S}\}\\
&T \to \textbf{int} \mid \textbf{void}\\
&\check{A} \to \varepsilon \mid \check{A}A;\\
&A \to Td \mid Td[] \mid Td()\\
&\check{S} \to S \mid \check{S};S\\
&S \to d = E \mid \textbf{if}\ (B)S \mid \textbf{if}\ (B)S\ \textbf{else}\ S \mid \textbf{while}\ (B)S\\
&\qquad \mid \textbf{return}\ E \mid \{\check{S}\} \mid d(\check{R})\\
&B \to B \wedge B \mid B \vee B \mid ErE \mid E\\
&E \to d = E \mid i \mid d \mid d(\check{R}) \mid E + E \mid E * E \mid (E)\\
&\check{R} \to \varepsilon \mid \check{R}R,\\
&R \to E \mid d[] \mid d()
\end{aligned}
$$

7.2 设计总览

本项目旨在为上一阶段生成的三地址代码 (Three-Address Code, TAC) 设计并实现一个针对 ARMv7-A 架构的目标代码生成器。整个编译流程的最后阶段可以概括为：

三地址代码 (TAC) → ARM 代码生成器 → ARM 汇编代码 (.s 文件)

本设计将聚焦于如何系统性地将平台无关的中间表示 (TAC) 映射到 ARM 指令集。核心任务包括：遵循 ARM 过程调用标准 (AAPCS)、管理运行时存储、选择合适的 ARM 指令。

7.3 ARM 运行时环境与内存管理

为了让程序在 ARM 架构上正确运行，我们必须首先定义其内存布局和资源使用约定。

7.3.1　活动记录 (Activation Record)

与 MIPS 类似，每个函数调用都会在运行时栈上创建一个专属的栈帧 (Stack Frame)。但其结构和寄存器使用遵循 ARM 的惯例。栈从高地址向低地址增长。

传入的参数 (第 5 个及以后)		<— *(由调用者压栈)*
fp+4	返回地址 (`lr`)	<— 被调函数保存
fp	旧的帧指针 (`fp`)	<— *fp* 指向的位置，被调函数保存
fp-4	… 局部变量 临时变量 (`t0, t1, …`)	局部变量与临时变量
sp	传出的参数	<— *(为被调函数准备，调用者压栈)* <— *sp* 指向的位置
		<— 低地址

图 7-1　ARM 栈帧结构 (遵循 AAPCS)

7.3.2　数据存储策略

- **全局变量**: 存放在 `.data` 静态数据区。
- **局部变量与临时变量**: 在函数的活动记录中分配空间，通过相对于帧指针 `fp` 的负偏移量访问。

7.3.3　寄存器约定 (AAPCS)

我们遵循 ARM 架构过程调用标准 (AAPCS)。

- `r0-r3`: 用于传递前 4 个整型或指针参数。`r0` 也用于存放函数返回值。它们是调用者保存的。
- `r4-r11`: 通用寄存器。`r11` 通常用作帧指针 (`fp`)。它们是被调用者保存的，意味着函数在使用它们之前必须先保存，返回前必须恢复。
- `r12`: 过程间临时寄存器 (`ip`)。
- `r13`: 栈指针 (`sp`)。
- `r14`: 链接寄存器 (`lr`)，存放函数调用的返回地址。
- `r15`: 程序计数器 (`pc`)。
- **变量存取**: 我们同样采用”load/store” 模型。所有变量的值都存放在内存中。操作前从内存加载到 `r0-r3` 等临时寄存器，计算后存回内存。

7.4　三地址代码到 ARM 的翻译方案

7.4.1　赋值与算术运算

1)　`x = y` (y 为变量)

```
ldr   r0, [fp, #offset_y]   @ r0 = y
str   r0, [fp, #offset_x]   @ x = r0
```

Listing 7.1　TAC: x = y

2)　x = c (c 为常数)

```
@ 如果c是合法立即数 (ARM immediate value)
mov   r0, #c                @ r0 = c
@ 对于任意32位常数，使用伪指令
@ ldr   r0, =c              @ r0 = c
str   r0, [fp, #offset_x]   @ x = r0
```

Listing 7.2　TAC: x = c

3)　x = y op z (op 为 +, −, ∗)

```
ldr   r0, [fp, #offset_y]   @ r0 = y
ldr   r1, [fp, #offset_z]   @ r1 = z
add   r2, r0, r1            @ r2 = r0 + r1
str   r2, [fp, #offset_x]   @ x = r2
```

Listing 7.3　TAC: x = y + z

7.4.2　控制流指令

1)　label L1

```
L1:
```

Listing 7.4　TAC: label L1

2)　goto L1

```
b     L1
```

Listing 7.5　TAC: goto L1

3)　if x op y goto L1 (条件跳转)

```
ldr   r0, [fp, #offset_x]   @ Load x
ldr   r1, [fp, #offset_y]   @ Load y
cmp   r0, r1                @ 比较 r0 和 r1，设置状态标志
beq   L1                    @ 如果相等(eq)则跳转
@ 其他条件码：bne(不等)，blt(小于)，bgt(大于)，etc.
```

Listing 7.6　TAC: if x == y goto L1

7.4.3　函数调用机制

注：为简化实现，我们统一采用栈来传递所有参数，但这部分描述了包含寄存器传参的标准 *AAPCS* 做法和我们简化方案的混合。

1) 函数调用方 (Caller)

param x (传递参数) 为保持简单，我们将所有参数都通过栈传递。

```
ldr   r0, [fp, #offset_x]
push  {r0}                  @ 将参数压栈
```

Listing 7.7 TAC: param x

y = call f, n (调用函数)

```
@ 1. 所有参数已经通过 'param' 指令压栈

@ 2. 调用函数
bl    f                     @ 调用 f, 返回地址存入 lr

@ 3. 从栈中清理参数 (调用结束后)
add   sp, sp, #4*n          @ n 是参数个数

@ 4. 获取返回值 (在 r0 中)
str   r0, [fp, #offset_y]   @ y = return value
```

Listing 7.8 TAC: y = call f, n

2) 函数被调方 (Callee) - 函数入口 (Prologue)

```
f:
@ 1. 保存调用者的状态
push  {fp, lr}              @ 保存旧的 fp 和返回地址 lr

@ 2. 设置当前函数新的 fp
mov   fp, sp

@ 3. 为所有局部变量和临时变量分配栈空间
sub   sp, sp, #frame_size
```

Listing 7.9 函数 f 的入口代码

3) 函数被调方 (Callee) - 函数出口 (Epilogue)

return E

```
@ 1. 计算 E 的值, 并放入 r0
ldr   r0, [fp, #offset_E]

@ 2. 恢复 sp, 回收局部变量空间
mov   sp, fp

@ 3. 恢复调用者的状态并返回 (高效方式)
pop   {fp, pc}              @ 恢复 fp, 并将保存的 lr直接弹入pc实现返回
```

Listing 7.10 TAC: return E

pop {fp, pc} 技巧是 ARM 汇编中非常常用且高效的返回方式。

7.5 实现策略

实现一个 ARMGenerator 类，其结构与 MIPSGenerator 类似，但内部逻辑全部替换为 ARM 的规则：

1) **初始化**: 输出 .data 段和 .text 段声明。为全局变量分配空间。

2) **函数遍历**: 对每个函数：

(1) **栈帧分析**: 计算局部变量和临时变量所需的栈空间大小，并记录每个变量相对于 fp 的偏移。

(2) **生成函数体**: 生成函数标签、标准的 ARM Prologue。

(3) **指令翻译**: 逐条将 TAC 翻译为 ARM 指令。

(4) **生成函数出口**: 生成标准的 ARM Epilogue。

3) **收尾**: 添加必要的外部函数声明（如 printf）和字符串常量。我们将使用 printf 来实现 print_int，这是 ARM/Linux 环境下的标准做法。

7.6 从三地址代码 (TAC) 到 ARM 的转换器实现

7.6.1 源程序

```
import collections

# ------------------------------------------------------------------------------
# 1. 数据结构定义 (Data Structures)
# ------------------------------------------------------------------------------
# 使用 namedtuple 来让 TAC 指令更具可读性
TAC = collections.namedtuple('TAC', ['op', 'arg1', 'arg2', 'dest'])

# 函数的数据结构
class Function:
    """代表一个函数，包含其名称、TAC序列和栈帧信息"""
    def __init__(self, name, tac_sequence):
        self.name = name
        self.tac = tac_sequence
        self.stack_frame = {}  # 存储变量名到 [fp] 偏移量的映射
        self.frame_size = 0    # 仅用于存储局部/临时变量的栈帧大小（字节）
        self.param_count = 0   # 传入参数的个数

# ------------------------------------------------------------------------------
# 2. ARM 代码生成器 (ARM Code Generator)
# ------------------------------------------------------------------------------
class ARMGenerator:
    """
    此类遵循设计的方案，将 TAC 翻译为 ARM 汇编代码。
    """
    def __init__(self, functions, global_vars):
        self.functions = {f.name: f for f in functions}
```

```
        self.global_vars = global_vars
        self.arm_code = []
        self._current_function = None # 追踪当前正在处理的函数
        self.PRINTF_FORMAT_LABEL = "printf_format_str"

    def _emit(self, code, comment=None):
        """辅助函数，用于生成带缩进和注释的 ARM 代码行 (注释符 @)"""
        if ":" in code: # 这是一个标签，不需要缩进
            self.arm_code.append(f"{code}")
        elif comment:
            self.arm_code.append(f"    {code:<25} @ {comment}")
        else:
            self.arm_code.append(f"    {code}")

    def _get_value(self, var_name, dest_reg):
        """
        核心辅助函数：加载一个变量或常数的值到指定的临时寄存器。
        这实现了 "load/store" 模型。
        """
        # Case 1: 常数
        if isinstance(var_name, int):
            self._emit(f"ldr {dest_reg}, ={var_name}", f"{dest_reg} = {var_name}")
            return

        # Case 2: 全局变量
        if var_name in self.global_vars:
            self._emit(f"ldr {dest_reg}, ={var_name}", f"Load address of global '{var_name}'")
            self._emit(f"ldr {dest_reg}, [{dest_reg}]", f"Load value: {dest_reg} = *({var_name})")
            return

        # Case 3: 局部变量、参数或临时变量
        if var_name in self._current_function.stack_frame:
            offset = self._current_function.stack_frame[var_name]
            self._emit(f"ldr {dest_reg}, [fp, #{offset}]", f"Load from stack: {dest_reg} = {var_name}")
            return

        raise NameError(f"Variable '{var_name}' not found in current scope.")

    def _store_value(self, src_reg, var_name):
        """
        核心辅助函数：将一个临时寄存器的值存回变量的内存地址。
        """
        # Case 1: 全局变量
        if var_name in self.global_vars:
            # 使用 r1 作为临时地址寄存器
```

```
            self._emit(f"ldr r1, ={var_name}", f"Load address of global '{var_name}' into r1")
            self._emit(f"str {src_reg}, [r1]", f"Store to global var: {var_name} = {src_reg}")
            return

        # Case 2: 局部变量或临时变量
        if var_name in self._current_function.stack_frame:
            offset = self._current_function.stack_frame[var_name]
            self._emit(f"str {src_reg}, [fp, #{offset}]", f"Store to stack: {var_name} = {src_reg}")
            return

        raise NameError(f"Variable '{var_name}' not found for storing.")


    def generate(self):
        """
        主生成函数，遵循设计的实现流程。
        """
        self._generate_data_section()
        self._generate_text_section()
        return "\n".join(self.arm_code)

    def _generate_data_section(self):
        """1. 初始化：输出 .data 段声明"""
        self._emit(".data")
        for var in self.global_vars:
            self._emit(f"{var}: .word 0", f"Global variable '{var}'")
        # 为 print_int (printf) 准备格式化字符串
        self._emit(f"{self.PRINTF_FORMAT_LABEL}: .asciz \"%d\\n\"")

    def _generate_text_section(self):
        """2. 主程序入口：输出 .text 段和启动序列"""
        self._emit("\n.text")
        self._emit(".global main") # 声明 main 为全局
        self._emit(".extern printf", "Declare external C library function")

        # 3. 函数遍历
        func_order = ['main'] + [name for name in self.functions if name != 'main']
        for func_name in func_order:
            if func_name in self.functions:
                self._generate_function(self.functions[func_name])

    def _analyze_stack_frame(self, func):
        """
        4a. 栈帧分析：计算栈帧大小和变量偏移
        """
```

```
        all_symbols = set()
        param_symbols = collections.OrderedDict() #保持参数声明顺序

        # 收集所有参数
        for instruction in func.tac:
            if instruction.op == 'param_decl':
                param_symbols[instruction.dest] = None

        func.param_count = len(param_symbols)

        # 收集所有本地符号（包括局部变量和临时变量）
        for instruction in func.tac:
            for symbol in [instruction.arg1, instruction.arg2, instruction.dest]:
                if isinstance(symbol, str) and symbol not in self.functions and
    symbol not in self.global_vars:
                    all_symbols.add(symbol)

        # 分配参数偏移（相对于$fp的正向偏移）
        # Prologue: push {fp, lr} -> fp, sp 指向同一个位置
        # 旧的 fp 在 [fp, #4], lr 在 [fp, #0]
        # 因此，第一个由调用者压栈的参数在 [fp, #8]
        param_offset = 8
        for param in param_symbols:
            func.stack_frame[param] = param_offset
            param_offset += 4

        # 分配局部/临时变量偏移（相对于$fp的负向偏移）
        local_symbols = all_symbols - set(param_symbols.keys())
        local_offset = -4
        for symbol in sorted(list(local_symbols)): # 排序以保证编译结果一致
            func.stack_frame[symbol] = local_offset
            local_offset -= 4

        # --- FIX START ---
        # 计算仅用于局部变量和临时变量的栈大小，并进行8字节对齐
        num_local_bytes = len(local_symbols) * 4
        # AAPCS要求栈在公共接口处是8字节对齐的。
        # (num_bytes + 7) & ~7 是一个向上对齐到8字节边界的标准方法。
        alignment = 8
        func.frame_size = (num_local_bytes + alignment - 1) & ~(alignment - 1)
        # --- FIX END ---

    def _generate_function(self, func):
        """生成单个函数的汇编代码"""
        self.arm_code.append(f"\n@ ---------------- Function: {func.name}
    ----------------")
        self._emit(f"{func.name}:")
        self._current_function = func

```

```
        # 4a. 栈帧分析
        self._analyze_stack_frame(func)

        # 4b. 函数入口 (Prologue)
        self._emit("@ --- Prologue ---")
        self._emit("push {fp, lr}", "Save frame pointer and link register")
        self._emit("mov fp, sp", "Set up new frame pointer")
        if self._current_function.frame_size > 0:
            self._emit(f"sub sp, sp, #{self._current_function.frame_size}", "
    Allocate space for local vars (aligned)")
        self._emit("@ --- End Prologue ---\n")

        # 4c. 指令翻译
        for instruction in func.tac:
            self._translate_instruction(instruction)

    def _translate_instruction(self, instruction):
        """根据TAC指令类型，分发到不同的翻译函数"""
        op = instruction.op
        if op != 'param_decl':
            self._emit(f"@ TAC: {op} {instruction.arg1 or ''} {instruction.arg2 or
     ''} {instruction.dest or ''}")

        if op == 'assign':
            self._get_value(instruction.arg1, "r0")
            self._store_value("r0", instruction.dest)

        elif op in ['add', 'sub', 'mul', 'div']:
            self._get_value(instruction.arg1, "r0") # r0 = y
            self._get_value(instruction.arg2, "r1") # r1 = z
            op_map = {'add': 'add', 'sub': 'sub', 'mul': 'mul', 'div': 'sdiv'} #
    sdiv for signed division
            self._emit(f"{op_map[op]} r2, r0, r1", f"r2 = r0 {op} r1")
            self._store_value("r2", instruction.dest) # x = r2

        elif op == 'label':
            self._emit(f"{instruction.dest}:")

        elif op == 'goto':
            self._emit(f"b {instruction.dest}")

        elif op.startswith('if'): # ifeq, ifne, iflt, etc.
            self._get_value(instruction.arg1, "r0")
            self._get_value(instruction.arg2, "r1")
            self._emit("cmp r0, r1", "Compare operands")
            branch_op = {
                'ifeq': 'beq', 'ifne': 'bne', 'iflt': 'blt',
                'ifle': 'ble', 'ifgt': 'bgt', 'ifge': 'bge'
            }[op]
```

```
            self._emit(f"{branch_op} {instruction.dest}", f"Branch if {branch_op}"
)

        elif op == 'param':
            self._get_value(instruction.dest, "r0")
            self._emit("push {r0}", f"Push param '{instruction.dest}'")

        elif op == 'call':
            func_name = instruction.arg1
            num_params = instruction.arg2

            # 特殊处理 print_int，将其转换为 printf 调用
            if func_name == 'print_int':
                self._emit("pop {r1}", "Pop argument into r1 for printf")
                self._emit(f"ldr r0, ={self.PRINTF_FORMAT_LABEL}", "Load format
string address into r0")
                self._emit("bl printf", "Call C library printf")
            else:
                self._emit(f"bl {func_name}", f"Call function {func_name}")
                # 调用者负责清理参数栈
                if num_params > 0:
                    self._emit(f"add sp, sp, #{num_params * 4}", "Clean up params
from stack")

            # 获取返回值（总是在 r0）
            if instruction.dest:
                self._store_value("r0", instruction.dest)

        elif op == 'return':
            # --- Epilogue ---
            self._emit("@ --- Epilogue ---")
            # 1. 如果有返回值，计算并放入 r0
            if instruction.dest is not None:
                self._get_value(instruction.dest, "r0")

            # 2. 恢复 sp，回收局部变量空间
            self._emit("mov sp, fp", "Deallocate locals")

            # 3. 恢复调用者的状态并返回
            self._emit("pop {fp, pc}", "Restore fp and return (by loading lr into
pc)")
            self._emit("@ --- End Epilogue ---")

        elif op == 'param_decl':
            # 这是一个用于栈帧分析的伪指令，此处不生成代码
            pass

        else:
            self._emit(f"@ UNKNOWN TAC op: {op}")
```

```
256
257         # 添加空行以增加可读性
258         if op not in ['param_decl', 'label']:
259             self.arm_code.append("")
260
261 # ------------------------------------------------------------------------------
262 # 3. 示例程序 (Example Program)
263 # ------------------------------------------------------------------------------
264 if __name__ == '__main__':
265     # 定义全局变量
266     global_variables = ['global_res']
267
268     # 定义 add_nums 函数
269     # int add_nums(int p1, int p2) { return p1 + p2; }
270     add_nums_tac = [
271         TAC('param_decl', None, None, 'p1'),
272         TAC('param_decl', None, None, 'p2'),
273         TAC('add', 'p1', 'p2', 't0'),
274         TAC('return', None, None, 't0')
275     ]
276     f_add = Function('add_nums', add_nums_tac)
277
278     # 定义 main 函数
279     # void main() {
280     #   int a = 10;
281     #   int b = 20;
282     #   int c = add_nums(a, b);
283     #   global_res = c;
284     #   print_int(c);
285     #   return 0; // Standard exit
286     # }
287     main_tac = [
288         TAC('assign', 10, None, 'a'),
289         TAC('assign', 20, None, 'b'),
290         TAC('param', None, None, 'b'), # Params pushed in reverse order (cdecl)
291         TAC('param', None, None, 'a'),
292         TAC('call', 'add_nums', 2, 'c'),
293         TAC('assign', 'c', None, 'global_res'),
294         TAC('param', None, None, 'c'),
295         TAC('call', 'print_int', 1, None),
296         TAC('return', None, None, 0) # main returns 0 for success
297     ]
298     f_main = Function('main', main_tac)
299
300     # 创建生成器并生成代码
301     generator = ARMGenerator([f_main, f_add], global_variables)
302     arm_assembly = generator.generate()
303
304     print("="*50)
```

```
    print("Generated ARM Assembly Code (Alignment Fixed)")
    print("="*50)
    print(arm_assembly)
```

Listing 7.11　从三地址代码 (TAC) 到 ARM 的转换器实现

7.6.2　源程序输出

```
==================================================
Generated ARM Assembly Code (Alignment Fixed)
==================================================
    .data
global_res: .word 0
printf_format_str: .asciz "%d\n"

.text
    .global main
    .extern printf            @ Declare external C library function

@ ---------------- Function: main ----------------
main:
    @ --- Prologue ---
    push {fp, lr}             @ Save frame pointer and link register
    mov fp, sp                @ Set up new frame pointer
    sub sp, sp, #16           @ Allocate space for local vars (aligned)
    @ --- End Prologue ---

@ TAC: assign 10  a
    ldr r0, =10               @ r0 = 10
    str r0, [fp, #-4]         @ Store to stack: a = r0

@ TAC: assign 20  b
    ldr r0, =20               @ r0 = 20
    str r0, [fp, #-8]         @ Store to stack: b = r0

@ TAC: param   b
    ldr r0, [fp, #-8]         @ Load from stack: r0 = b
    push {r0}                 @ Push param 'b'

@ TAC: param   a
    ldr r0, [fp, #-4]         @ Load from stack: r0 = a
    push {r0}                 @ Push param 'a'

@ TAC: call add_nums 2 c
    bl add_nums               @ Call function add_nums
    add sp, sp, #8            @ Clean up params from stack
    str r0, [fp, #-12]        @ Store to stack: c = r0

@ TAC: assign c  global_res
```

```
    ldr r0, [fp, #-12]        @ Load from stack: r0 = c
    ldr r1, =global_res       @ Load address of global 'global_res' into r1
    str r0, [r1]              @ Store to global var: global_res = r0

@ TAC: param   c
    ldr r0, [fp, #-12]        @ Load from stack: r0 = c
    push {r0}                 @ Push param 'c'

@ TAC: call print_int 1
    pop {r1}                  @ Pop argument into r1 for printf
    ldr r0, =printf_format_str @ Load format string address into r0
    bl printf                 @ Call C library printf

@ TAC: return
    @ --- Epilogue ---
    ldr r0, =0                @ r0 = 0
    mov sp, fp                @ Deallocate locals
    pop {fp, pc}              @ Restore fp and return (by loading lr into pc)
    @ --- End Epilogue ---


@ ---------------- Function: add_nums ----------------
add_nums:
    @ --- Prologue ---
    push {fp, lr}             @ Save frame pointer and link register
    mov fp, sp                @ Set up new frame pointer
    sub sp, sp, #8            @ Allocate space for local vars (aligned)
    @ --- End Prologue ---

@ TAC: add p1 p2 t0
    ldr r0, [fp, #8]          @ Load from stack: r0 = p1
    ldr r1, [fp, #12]         @ Load from stack: r1 = p2
    add r2, r0, r1            @ r2 = r0 add r1
    str r2, [fp, #-4]         @ Store to stack: t0 = r2

@ TAC: return   t0
    @ --- Epilogue ---
    ldr r0, [fp, #-4]         @ Load from stack: r0 = t0
    mov sp, fp                @ Deallocate locals
    pop {fp, pc}              @ Restore fp and return (by loading lr into pc)
    @ --- End Epilogue ---
```

Listing 7.12　源程序输出