编译原理课件作业

第一章

1. **词法分析**：
   - 将源代码分解成最小的语法单元（如关键字、标识符、运算符等），称为“词法记号”或“标记”（token）。

   语法分析：
   - 构建抽象语法树（AST）或其他中间表示结构来表示程序的结构。

   语义分析：
   - 检查程序是否符合语义规则，如变量是否已经声明、类型匹配等。
   - 标注符号表以保存变量和函数的信息。

   **中间代码生成：
   - 将语法树或其他中间表示转换为中间代码，通常是介于高级语言和机器码之间的抽象形式。

   代码优化：
   - 优化中间代码以提高执行效率和减少内存使用。
   - 进行常见优化，如去除冗余代码、循环优化等。

   **目标代码生成：
   - 将中间代码转换为目标平台的机器代码或汇编代码。

   **目标代码优化：
   - 对生成的目标代码进行进一步优化，以提高性能。
2.

   编译器前端主要负责分析源代码并检查其正确性。前端的任务是将代码转换为中间表示，同时检测和报告代码中的错误。其主要任务包括：

   1. 词法分析：
      - 将源代码分解为标记（tokens）。
      - 识别代码中的关键字、标识符、运算符等基本单元。
   2. 语法分析：
      - 检查标记的结构是否符合语言的语法规则。
      - 构建抽象语法树（AST）或解析树来表示程序结构。
   3. 语义分析：
      - 确保程序符合语义规则，如类型检查、变量范围检查。
      - 构建和维护符号表以存储标识符和相关信息。
   4. 中间代码生成：
      - 将语法树转换为中间代码，例如三地址码（Three-Address Code）或中间表示（IR）。

   编译器后端主要负责将中间代码转换为目标机器的可执行代码。它的任务侧重于代码生成和优化，以提高程序的性能和效率。其主要任务包括：

1. 代码优化：
   - 对中间代码进行优化，减少执行时间和内存使用。
   - 包括循环展开、公共子表达式消除等优化技术。
2. 目标代码生成：
   - 将优化后的中间代码转换为目标平台的机器代码或汇编代码。
3. 目标代码优化：
   - 针对特定架构进行进一步优化，如寄存器分配、指令调度。
4. 寄存器分配和指令选择：
   - 将中间代码中的变量映射到物理寄存器。
   - 选择适合的指令来提高执行效率。

区别：

- 前端侧重于代码分析和转换为中间表示，主要是为了确保代码的正确性和标准化。
- 后端侧重于生成高效的目标代码，主要是为了在特定硬件平台上提高程序的性能。

3. 语法分析是根据语言的语法规则，分析由词法分析器生成的标记（tokens）序列，以确定它们是否形成有效的结构。语法分析的输出通常是一棵抽象语法树（AST），用于表示程序的语法结构。
   自上而下分析、自下而上分析、语法制导翻译

4. 作用：

   将源代码分解为标记：将连续的字符流分解为可以识别的标记，如关键字、标识符、数字、运算符、分隔符等。

   去除空白和注释：过滤掉对语法无关的空格和注释，以简化后续分析过程。

   标记识别和分类：识别源代码中的单词，并将它们分类为特定的标记类型，如`identifier`（标识符）、`keyword`（关键字）、`number`（数字）、`operator`（运算符）等。

   错误检测：在识别过程中发现词法错误，如不合法的标识符或数字格式时，给出适当的错误信息。

   输入：词法分析器的输入是源代码的字符序列。这些字符可以是程序员编写的代码文件中的所有字符。

   输出：词法分析器的输出是一个标记序列，每个标记包括两部分：标记类型：表示该标记的类别，如关键字、标识符、运算符等。；标记的值（或属性值）：用于标识实际的字符串或相关信息，如标识符的名称或数字的具体值。

5. 中间代码：编译器在将源代码转换为目标代码时生成的中间形式。它是一种介于源语言和目标机器语言之间的代码表示，旨在提高编译器的移植性和可优化性。

   三地址码、抽象语法树、控制流图、静态单赋值形式、四元式和三元式、P代码

第二章

1. `^(?:[01]{2})*$`

2. ```c
   #include <stdio.h>
   #include <string.h>
   #include <stdbool.h>

   // 定义一个函数来判断输入字符串是否匹配正则表达式
   bool matchesRegex(const char* str) {
       // 此处应实现一个自动机或正则匹配算法
       int len = strlen(str);
   ```
   ```

```
    int i = 0;

    // 简单的实现示例，假设已经有状态转移表
    // state 0: 初始状态
    // state 1: 匹配第一个字符 (a|b)
    // state 2: 匹配 (aa|bb)*
    // state 3: 匹配结尾的 (a|b)

    int state = 0;
    while (i < len) {
        char c = str[i];
        switch (state) {
            case 0:
                if (c == 'a' || c == 'b') state = 1;
                else return false;
                break;
            case 1:
                if (c == 'a' || c == 'b') state = 2;
                else return false;
                break;
            case 2:
                if ((c == 'a' && str[i + 1] == 'a') || (c == 'b' && str[i + 1] == 'b')) {
                    i++; // 跳过额外字符
                    state = 2;
                } else if (c == 'a' || c == 'b') {
                    state = 3;
                } else {
                    return false;
                }
                break;
            case 3:
                return i == len - 1;
            default:
                return false;
        }
        i++;
    }
    return (state == 3);
}

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    if (matchesRegex(str)) {
        printf("The string matches the pattern.\n");
    } else {
        printf("The string does not match the pattern.\n");
    }
    return 0;
}

```

3. 初始准备：

- 确保 DFA 只有一个初始状态和一个接受状态。如果有多个接受状态，添加一个新的接受状态，连接到原来的所有接受状态。

状态消除：

- 依次消除所有中间状态（不是初始或接受状态）。

生成正则表达式：

- 当只剩下初始状态和接受状态时，得到的路径即为最终的正则表达式。

4. 检查是否有等价状态：
   - 如果 DFA 中存在等价状态（在相同输入条件下可到达相同或等价的状态），则该 DFA 不是最小化的。
   - 两个状态是等价的，如果它们在相同的输入串下都接受或都拒绝。

   使用等价状态划分方法：
   - 进行状态划分，将 DFA 的状态集合分成接受状态和非接受状态两个初始组。
   - 重复以下过程，直到不能再细分：
     - 将每个组细分为子组，使得每个子组中的状态在相同输入下转移到同一组的状态。
   - 如果在划分过程中没有两个状态属于不同组但转移行为一致，则 DFA 已最小化。

第三章

1.

状态	a	b	$	S
0	S1			G0
1		S2		G1
2			R2	
3			ACC	

第一步：

- 当前输入符号是 `a`，在状态 0 查找 ACTION 表，发现移入 `S1`。
- 执行移入操作，栈变为 `[S，a]`，输入变为 `aabbb`，当前状态为 `1`。

第二步：

- 当前输入符号是 `a`，在状态 1 查找 ACTION 表，发现移入 `S1`。
- 执行移入操作，栈变为 `[S，a，S]`，输入变为 `abbb`，当前状态为 `1`。

第三步：

- 当前输入符号是 `a`，在状态 1 查找 ACTION 表，发现移入 `S1`。
- 执行移入操作，栈变为 `[S，a，S，a]`，输入变为 `bbb`，当前状态为 `1`。

第四步：

- 当前输入符号是 `b`，在状态 1 查找 ACTION 表，发现移入 `S2`。
- 执行移入操作，栈变为 `[S，a，S，a，S]`，输入变为 `bb`，当前状态为 `2`。

第五步：

- 当前输入符号是 `b`，在状态 2 查找 ACTION 表，发现移入 `S2`。
- 执行移入操作，栈变为 `[S，a，S，a，S，b]`，输入变为 `b`，当前状态为 `2`。

第六步：

- 当前输入符号是 b，在状态 2 查找 ACTION 表，发现移入 S2。
- 执行移入操作，栈变为 [S，a，S，a，S，b，b]，输入变为 空，当前状态为 2。

第七步：

- 当前输入已为空，在状态 2 查找 ACTION 表，发现规约操作 R2。
- 规约 S → ε，栈变为 [S，a，S，a，S，b]，输入变为 空，当前状态为 3。

第八步：

- 当前状态是 3，输入为空，接受操作。

2. FIRST(E)={id,(,+,∗}，FOLLOW}(E) = {), $, +, * }
产生式：

E→E+E

E→E∗E

E→(E)

E→id

3.

状态	a	c	b	d	$	S	T
0	R1	S4				2	
1			S3				
2					ACC		
3				S6			
4		S5					
5	R2						
6	R3						

初始状态：

- 输入：acccdbd
- 栈：[S]（初始状态为 S）
- 当前状态：0

步骤 1：

- 当前输入符号是 a，在状态 0 查找 ACTION 表，发现移入 S1。
- 执行移入操作，栈变为 [S，a]，输入变为 cccdbd，当前状态为 1。

步骤 2：

- 当前输入符号是 c，在状态 1 查找 ACTION 表，发现移入 S4。
- 执行移入操作，栈变为 [S，a，T，c]，输入变为 ccdbd，当前状态为 4。

步骤 3：

- 当前输入符号是 c，在状态 4 查找 ACTION 表，发现移入 S5。
- 执行移入操作，栈变为 [S，a，T，c，T，c]，输入变为 cdbd，当前状态为 5。

步骤 4：

- 当前输入符号是 d，在状态 5 查找 ACTION 表，发现规约 T → cTd。

- 执行规约操作，栈变为 [S，a，T，c，T]，当前状态为 4。

步骤 5：

- 当前输入符号是 d，在状态 4 查找 ACTION 表，发现移入 S6。
- 执行移入操作，栈变为 [S，a，T，c，T，d]，输入变为 bd，当前状态为 6。

步骤 6：

- 当前输入符号是 b，在状态 6 查找 ACTION 表，发现规约 T → ε。
- 执行规约操作，栈变为 [S，a，T，c，T]，当前状态为 4。

步骤 7：

- 当前输入符号是 b，在状态 4 查找 ACTION 表，发现规约 T → ε。
- 执行规约操作，栈变为 [S，a，T]，当前状态为 2。

步骤 8：

- 当前输入符号是 b，在状态 2 查找 ACTION 表，发现规约 S → aSb。
- 执行规约操作，栈变为 [S]，当前状态为 0。

步骤 9：

- 当前输入符号是 b，在状态 0 查找 ACTION 表，发现规约 S → T。
- 执行规约操作，栈变为 [S]，输入为空，接受。

4.

状态	**id**	+	*	(	)	$	E	T	F
0	S1			S2			3	4	5
1		R1				ACC			
2	S1			S2			6	7	8
3		S4							
4	S1			S2					

步骤 1：

- 输入：id * (id + id)
- 当前栈：[E]
- 当前状态：0
- 当前符号是 id，在状态 0 查找 ACTION 表，发现移入 S1，执行移入操作，栈变为 [E，id]，输入变为 * (id + id)，当前状态为 1。

步骤 2：

- 输入：* (id + id)
- 当前栈：[E，id]
- 当前状态：1
- 当前符号是 *，在状态 1 查找 ACTION 表，发现进行规约 E → id，执行规约操作，栈变为 [E]，当前状态变为 0。

步骤 3：

- 输入：* (id + id)
- 当前栈：[E]

- 当前状态：0
- 当前符号是 *，在状态 0 查找 ACTION 表，发现移入 S4，执行移入操作，栈变为 [E，*]，输入变为 (id + id)，当前状态为 4。

步骤 4：

- 输入：(id + id)
- 当前栈：[E，*,]
- 当前状态：4
- 当前符号是 (，在状态 4 查找 ACTION 表，发现移入 S2，执行移入操作，栈变为 [E，*，(,]，输入变为 id + id)，当前状态为 2。

步骤 5：

- 输入：id + id)
- 当前栈：[E，*，(,]
- 当前状态：2
- 当前符号是 id，在状态 2 查找 ACTION 表，发现移入 S1，执行移入操作，栈变为 [E，*，(，id]，输入变为 + id)，当前状态为 1。

步骤 6：

- 输入：+ id)
- 当前栈：[E，*，(，id]
- 当前状态：1
- 当前符号是 +，在状态 1 查找 ACTION 表，发现进行规约 E → id，执行规约操作，栈变为 [E，*，(，]，当前状态变为 0。

步骤 7：

- 输入：+ id)
- 当前栈：[E，*，( ]
- 当前状态：0
- 当前符号是 +，在状态 0 查找 ACTION 表，发现移入 S4，执行移入操作，栈变为 [E，*，(，，+]，输入变为 id)，当前状态为 4。

5.

状态	**a**	**b**	**c**	**$**	**S**
0	S1	S2	S3		4
1				R1	
2				R2	
3				ACC	
4	S1	S2	S3		4
5	S1	S2	S3		5

步骤 1：

- 输入：ababcc
- 当前栈：[S]
- 当前状态：0

- 当前符号是 a，在状态 0 查找 ACTION 表，发现移入状态 S1，执行移入操作，栈变为 [S，a]，输入变为 babcc，当前状态为 1。

步骤 2：

- 输入：babcc
- 当前栈：[S，a]
- 当前状态：1
- 当前符号是 b，在状态 1 查找 ACTION 表，发现移入状态 S2，执行移入操作，栈变为 [S，a，b]，输入变为 abcc，当前状态为 2。

步骤 3：

- 输入：abcc
- 当前栈：[S，a，b]
- 当前状态：2
- 当前符号是 a，在状态 2 查找 ACTION 表，发现移入状态 S1，执行移入操作，栈变为 [S，a，b，a]，输入变为 bcc，当前状态为 1。

步骤 4：

- 输入：bcc
- 当前栈：[S，a，b，a]
- 当前状态：1
- 当前符号是 b，在状态 1 查找 ACTION 表，发现移入状态 S2，执行移入操作，栈变为 [S，a，b，a，b]，输入变为 cc，当前状态为 2。

步骤 5：

- 输入：cc
- 当前栈：[S，a，b，a，b]
- 当前状态：2
- 当前符号是 c，在状态 2 查找 ACTION 表，发现移入状态 S3，执行移入操作，栈变为 [S，a，b，a，b，c]，输入变为 c，当前状态为 3。

步骤 6：

- 输入：c
- 当前栈：[S，a，b，a，b，c]
- 当前状态：3
- 当前符号是 c，在状态 3 查找 ACTION 表，发现移入状态 S3，执行移入操作，栈变为 [S，a，b，a，b，c，c]，输入变为空，当前状态为 3。

步骤 7：

- 输入为空，栈中已没有其他符号，分析完成，接受。

6.

步骤 1：

- 输入：acbcc
- 当前栈：[S]
- 当前状态：S
- 当前符号是 a，在 LL(1) 分析表中查找 S 对应 a 的产生式，选择产生式 S → aSb。
- 执行移入操作，将 S 替换为 aSb，栈变为 [a，S，b]，输入变为 cbcc。

步骤 2：

- 输入：`cbcc`
- 当前栈：`[a, S, b]`
- 当前状态：`S`
- 当前符号是 `c`，在 LL(1) 分析表中查找 `S` 对应 `c` 的产生式，选择产生式 `S → cSc`。
- 执行移入操作，将 `S` 替换为 `cSc`，栈变为 `[a, c, S, c, b]`，输入变为 `bcc`。

步骤 3：

- 输入：`bcc`
- 当前栈：`[a, c, S, c, b]`
- 当前状态：`S`
- 当前符号是 `b`，在 LL(1) 分析表中查找 `S` 对应 `b` 的产生式，选择产生式 `S → ε`。
- 执行规约操作，将 `S` 规约为空串，栈变为 `[a, c, c, b]`，输入变为 `cc`。

步骤 4：

- 输入：`cc`
- 当前栈：`[a, c, c, b]`
- 当前状态：`c`
- 当前符号是 `c`，在 LL(1) 分析表中查找 `S` 对应 `c` 的产生式，选择产生式 `S → cSc`。
- 执行移入操作，将 `S` 替换为 `cSc`，栈变为 `[a, c, c, S, c, b]`，输入变为 `c`。

步骤 5：

- 输入：`c`
- 当前栈：`[a, c, c, S, c, b]`
- 当前状态：`S`
- 当前符号是 `c`，在 LL(1) 分析表中查找 `S` 对应 `c` 的产生式，选择产生式 `S → cSc`。
- 执行移入操作，将 `S` 替换为 `cSc`，栈变为 `[a, c, c, c, S, c, b]`，输入变为空。

步骤 6：

- 输入为空，栈已清空，分析完成，接受。

7.

状态	**id**	+	*	(	)	$	**E**
0	S1			S2			3
1		S4	S5		R1	R1	
2	S1			S2			3
3					R6		7
4		S4	S5		R2	R2	3
5		S4	S5		R3	R3	3
6	S1			S2			3

步骤 1：

- 输入：`id*(id+id)`

- 当前栈：[E]
- 当前状态：0
- 当前符号是 id，在状态 0 查找 **ACTION** 表，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id]，输入变为 *(id+id)，当前状态为 1。

步骤 2：

- 输入：*(id+id)
- 当前栈：[E，id]
- 当前状态：1
- 当前符号是 *，在状态 1 查找 **ACTION** 表，发现移入状态 S5。
- 执行移入操作，栈变为 [E，id，*]，输入变为 (id+id)，当前状态为 5。

步骤 3：

- 输入：(id+id)
- 当前栈：[E，id，*，*]
- 当前状态：5
- 当前符号是 (，在状态 5 查找 **ACTION** 表，发现移入状态 S2。
- 执行移入操作，栈变为 [E，id，*，*，(]，输入变为 id+id)，当前状态为 2。

步骤 4：

- 输入：id+id)
- 当前栈：[E，id，*，*，(]
- 当前状态：2
- 当前符号是 id，在状态 2 查找 **ACTION** 表，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id，*，*，(，id]，输入变为 +id)，当前状态为 1。

步骤 5：

- 输入：+id)
- 当前栈：[E，id，*，*，(，id]
- 当前状态：1
- 当前符号是 +，在状态 1 查找 **ACTION** 表，发现移入状态 S4。
- 执行移入操作，栈变为 [E，id，*，*，(，id，+]，输入变为 id)，当前状态为 4。

步骤 6：

- 输入：id)
- 当前栈：[E，id，*，*，(，id，+]
- 当前状态：4
- 当前符号是 id，在状态 4 查找 **ACTION** 表，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id，*，*，(，id，+，id]，输入变为空，当前状态为 1。

步骤 7：

- 输入为空，栈已清空，分析完成，接受。

8.

状态	**a**	**b**	**c**	**$**	**S**
0	S1	S2	S2	R1	4

状态	a	b	c	$	S
1	S1	S2	S3	R2	4
2	S1	S2	S3	R3	4
3					
4	R1	R2	R3		

步骤 1：

- 输入：`ababcc`
- 当前栈：`[S]`
- 当前状态：`0`
- 当前符号是 `a`，在 **ACTION** 表中查找，发现移入状态 `S1`。
- 执行移入操作，栈变为 `[S，a]`，输入变为 `babcc`，当前状态为 `1`。

步骤 2：

- 输入：`babcc`
- 当前栈：`[S，a]`
- 当前状态：`1`
- 当前符号是 `b`，在 **ACTION** 表中查找，发现移入状态 `S2`。
- 执行移入操作，栈变为 `[S，a，b]`，输入变为 `abcc`，当前状态为 `2`。

步骤 3：

- 输入：`abcc`
- 当前栈：`[S，a，b]`
- 当前状态：`2`
- 当前符号是 `a`，在 **ACTION** 表中查找，发现移入状态 `S1`。
- 执行移入操作，栈变为 `[S，a，b，a]`，输入变为 `bcc`，当前状态为 `1`。

步骤 4：

- 输入：`bcc`
- 当前栈：`[S，a，b，a]`
- 当前状态：`1`
- 当前符号是 `b`，在 **ACTION** 表中查找，发现移入状态 `S2`。
- 执行移入操作，栈变为 `[S，a，b，a，b]`，输入变为 `cc`，当前状态为 `2`。

步骤 5：

- 输入：`cc`
- 当前栈：`[S，a，b，a，b]`
- 当前状态：`2`
- 当前符号是 `c`，在 **ACTION** 表中查找，发现移入状态 `S3`。
- 执行移入操作，栈变为 `[S，a，b，a，b，c]`，输入变为 `c`，当前状态为 `3`。

步骤 6：

- 输入：`c`

- 当前栈：[S，a，b，a，b，c]
- 当前状态：3
- 当前符号是 c，在 **ACTION** 表中查找，发现可以进行规约操作 S → c（R3）。
- 执行规约，栈变为 [S，a，b，a，b]，输入变为空。

步骤 7：

- 输入为空，栈已清空，分析完成，接受。

9.

状态	**id**	+	*	(	)	$	**E**	**T**	**F**
0	S1			S2			3	4	5
1		R3			R3	R3			
2	S1			S2			3	4	5
3			S5						
4	S1			S2			6	7	8
5		R2			R2	R2			
6		R1			R1	R1			

步骤 1：

- 输入：id*(id+id)
- 当前栈：[E]
- 当前状态：0
- 当前符号是 id，在 **ACTION** 表中查找，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id]，输入变为 *(id+id)，当前状态为 1。

步骤 2：

- 输入：*(id+id)
- 当前栈：[E，id]
- 当前状态：1
- 当前符号是 *，在 **ACTION** 表中查找，发现移入状态 S5。
- 执行移入操作，栈变为 [E，id，*]，输入变为 (id+id)，当前状态为 5。

步骤 3：

- 输入：(id+id)
- 当前栈：[E，id，*，*]
- 当前状态：5
- 当前符号是 (，在 **ACTION** 表中查找，发现移入状态 S2。
- 执行移入操作，栈变为 [E，id，*，*，(]，输入变为 id+id)，当前状态为 2。

步骤 4：

- 输入：id+id)
- 当前栈：[E，id，*，*，(]
- 当前状态：2

- 当前符号是 id，在 **ACTION** 表中查找，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id，*，*，(，id]，输入变为 +id)，当前状态为 1。

步骤 5：

- 输入：+id)
- 当前栈：[E，id，*，*，(，id]
- 当前状态：1
- 当前符号是 +，在 **ACTION** 表中查找，发现移入状态 S4。
- 执行移入操作，栈变为 [E，id，*，*，(，id，+]，输入变为 id)，当前状态为 4。

步骤 6：

- 输入：id)
- 当前栈：[E，id，*，*，(，id，+]
- 当前状态：4
- 当前符号是 id，在 **ACTION** 表中查找，发现移入状态 S1。
- 执行移入操作，栈变为 [E，id，*，*，(，id，+，id]，输入变为空，结束。

10. 步骤 1：分析符号 S
    - 当前符号是 S，栈顶是 S，输入符号是 a。
    - 根据预测分析表，S → aSb 对应于 a，执行移入操作。
    - 执行后：
      - 栈：Sb
      - 输入：acccdbd

    步骤 2：分析符号 S
    - 当前符号是 S，栈顶是 S，输入符号是 a。
    - 根据预测分析表，S → aSb 对应于 a，执行移入操作。
    - 执行后：
      - 栈：SbSb
      - 输入：cccd

    步骤 3：分析符号 S
    - 当前符号是 S，栈顶是 S，输入符号是 c。
    - 根据预测分析表，S → T 对应于 c，执行移入操作。
    - 执行后：
      - 栈：bSbT
      - 输入：cccd

    步骤 4：分析符号 T
    - 当前符号是 T，栈顶是 T，输入符号是 c。
    - 根据预测分析表，T → cTd 对应于 c，执行移入操作。
    - 执行后：
      - 栈：Tdb
      - 输入：cccd

    步骤 5：分析符号 T

- 当前符号是 `T`，栈顶是 `T`，输入符号是 `d`。
- 根据预测分析表，`T → ε`，执行规约操作，弹出 `T`。
- 执行后：
  - 栈：`bSb`
  - 输入：`cccd`

步骤 6：分析符号 `b`

- 当前符号是 `b`，栈顶是 `b`，输入符号是 `b`。
- 执行移入操作，执行后：
  - 栈：`Sb`
  - 输入：`cccd`

步骤 7：分析符号 `S`

- 当前符号是 `S`，栈顶是 `S`，输入符号是 `c`。
- 根据预测分析表，`S → T` 对应于 `c`，执行移入操作。
- 执行后：
  - 栈：`bT`
  - 输入：`cccd`

步骤 8：分析符号 `T`

- 当前符号是 `T`，栈顶是 `T`，输入符号是 `c`。
- 根据预测分析表，`T → cTd` 对应于 `c`，执行移入操作。
- 执行后：
  - 栈：`Td`
  - 输入：`d`

步骤 9：分析符号 `T`

- 当前符号是 `T`，栈顶是 `T`，输入符号是 `d`。
- 根据预测分析表，`T → ε`，执行规约操作，弹出 `T`。
- 执行后：
  - 栈：`b`
  - 输入：`d`

步骤 10：分析符号 `b`

- 当前符号是 `b`，栈顶是 `b`，输入符号是 `d`。
- 执行移入操作。
- 执行后：
  - 栈：``
  - 输入：`d`

第四章

1. **E.val**：表示表达式 `E` 的值。

   T.val：表示项 `T` 的值。

F.val：表示因子 `F` 的值。

integer.val：表示整数常量 `integer` 的值。**E → E + T**：

- 该规则表示将 `E` 和 `T` 相加。
- **E.val = E1.val + T.val**

解释：当我们遇到 `E + T` 时，首先计算 `E` 和 `T` 的值，然后将它们相加，结果赋给新的 `E`。

E → T：

- 当 `E` 直接等于 `T` 时，`E` 的值就是 `T` 的值。
- **E.val = T.val**

解释：如果 `E` 是一个 `T`，那么 `E` 的值就是 `T` 的值。

T → T * F：

- 该规则表示将 `T` 和 `F` 相乘。
- **T.val = T1.val * F.val**

解释：当我们遇到 `T * F` 时，首先计算 `T` 和 `F` 的值，然后将它们相乘，结果赋给新的 `T`。

T → F：

- 当 `T` 直接等于 `F` 时，`T` 的值就是 `F` 的值。
- **T.val = F.val**

解释：如果 `T` 是一个 `F`，那么 `T` 的值就是 `F` 的值。

F → integer：

- 该规则表示 `F` 为一个整数常量，直接取其值。
- **F.val = integer.val**

解释：如果 `F` 是一个整数常量，那么 `F` 的值就是这个整数的值。

2. **id.type**：表示 `id` 变量的类型。

E.type：表示表达式 `E` 的类型。
步骤 1：定义属性

我们为每个非终结符定义属性，具体如下：

- **stmt.type**：表示 `stmt` 语句的类型。我们将使用它来检查赋值语句两边的类型是否匹配。
- **E.type**：表示表达式 `E` 的类型。`E` 可以是 `id` 或者是常量 `int`。
- **id.type**：表示 `id` 变量的类型。假设这是一个外部符号表提供的信息，即每个变量 `id` 都有一个类型。

步骤 2：翻译规则

1. **stmt → id = E**：
   - **stmt.type = E.type**
   - **E.type** 必须和 `id.type` 匹配，表示赋值的右边和左边类型要一致。
2. **E → id**：
   - **E.type = id.type**
   - 当 `E` 产生式为 `id` 时，`E.type` 应当等于 `id.type`，即 `E` 的类型与 `id` 的类型一致。
3. **E → int**：
   - **E.type = "int"**
   - 当 `E` 产生式为常量 `int` 时，`E.type` 为 `"int"`。

步骤 3：检查类型匹配

在语法分析的过程中，我们将检查赋值语句左右两边的类型是否匹配。如果 `stmt.type` 不等于 `E.type`，则抛出类型不匹配的错误。

第五章

无

第六章

无

第七章

无