



北京化工大学

编译原理项目报告

2022-2023 学年第 2 学期

基于 C 语言的编译器简易实现

计科 2003-2020040227-谭植文 (组长)

计科 2004-2020040177-刘晔宁

更新日期: 2023 年 6 月 25 日

目录

1	前言	1
2	项目目标	1
3	开发环境	1
4	实现方法	1
4.1	主程序	2
4.1.1	基本思路	2
4.1.2	算法流程	2
4.1.3	解决问题	2
4.2	词法分析	3
4.2.1	基本思路	3
4.2.2	算法流程	3
4.2.3	解决问题	4
4.3	语法分析	5
4.3.1	基本思路	5
4.3.2	算法流程	5
4.3.3	解决问题	5
4.4	语义分析	6
4.4.1	基本思路	6
4.4.2	算法流程	6
5	结果	6
5.1	文件目录结构	6
5.2	使用方法	7
5.3	实际例子的输入输出效果	7
5.3.1	输入	7
5.3.2	输出	8
5.3.3	查看语法树	8
6	项目分工	9

1 前言

在编译原理课程的学习中，了解到编译器主要的工作原理，我们小组根据在此学习的基础上，完成课程项目作业。

2 项目目标

本项目的目标是实现一个简单的 C 语言编译器。

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 代码优化
- 目标代码生成

3 开发环境

- 操作系统：Windows 11
- 开发工具：Visual Studio Code
- 开发语言：C++
- 编译器：MinGW 8.1.0
- graphviz 8.0.3

4 实现方法

4.1 主程序

4.1.1 基本思路

1. 读取词法文件，生成对应的 DFA
2. 读取源文件，对源文件进行词法分析，输出 token 序列

3. 根据语法文件，生成 LR(1) 分析表
4. 根据 token 序列作为输入，进行语法分析，输出语法生成树
5. 遍历语法树，递归进行语义分析，输出语义分析结果

4.1.2 算法流程

```
// 词法分析
while 读取词法文件：
    生成对应的 DFA
while 读取源文件：
    if 当前字符为字母：
        识别标识符
    else if 当前字符为数字：
        识别数字
    else if 当前字符为运算符：
        识别运算符
    else if 当前字符为界符：
        识别界符
    else:
        识别错误
输出词法分析结果 token 序列

//语法分析
while 读取语法文件：
    生成 LR(1) 分析表
将 token 序列作为输入，进行语法分析
输出语法分析结果

//语义分析
遍历语法树，递归进行语义分析
```

4.1.3 解决问题

- 识别界符时，需要考虑界符的组合，如<=、>=、==、!=、&&、||等
- 对于某些界符，需考虑与 DFA 生成时所需的符号是否产生冲突，例如\、*、(、)、|等。利用其他符号做替换

4.2 词法分析

4.2.1 基本思路

1. 将正则表达式转换为后缀表达式

2. 将后缀表达式转换为 NFA
3. 将 NFA 转换为 DFA
4. 最小化 DFA

4.2.2 算法流程

// 正则表达式转为后缀表达式

建立运算符栈用于运算符的存储，此运算符遵循越往栈顶优先级越高的原则

while 扫描表达式：

if 当前字符是字母(优先级为0的符号)：

直接输出

else if 当前字符为运算符或者括号(优先级不为0的符号)，则判断：

if 当前运算符为 '('：

直接入栈

else if ')':

出栈并顺序输出运算符直到遇到第一个 '('，遇到的第一个 '(' 出栈但不输出；

else:

if 栈顶元素是 '('：

当前元素直接入栈；

else if 栈顶元素优先级 >= 当前元素优先级：

while(栈顶元素优先级 > 当前元素优先级)：

出栈并顺序输出运算符

当前元素入栈

else if 栈顶元素优先级 < 当前元素优先级：

当前元素直接入栈

顺序出栈并输出运算符直到栈元素为空

// 后缀表达式转为 NFA

建立状态栈用于存储状态，建立状态转换表用于存储状态转换关系

while 读取后缀表达式：

if 元素是字符：

创建两个状态，分别作为状态转换表的起始状态和终止状态

将状态对入栈

else if 元素是 '|':

创建两个状态，分别作为状态转换表的起始状态和终止状态

从栈中取两个状态对

根据法则进行链接

将状态对入栈

else if 元素是 '*':

创建两个状态，分别作为状态转换表的起始状态和终止状态

从栈中取两个状态对

根据法则进行链接

将状态对入栈

else if 元素是 '':

创建两个状态，分别作为状态转换表的起始状态和终止状态

从栈中取状态对

```
根据法则进行链接  
将状态对入栈
```

// NFA 转 DFA

```
建立 DFA 状态集合, 用于存储 DFA 的状态  
建立 DFA 状态队列  
NFA 开始状态闭包入队  
while 队列非空:  
    取队首状态闭包  
    for 每个输入符号:  
        计算输入符号对应的状态闭包  
        if 该状态闭包不在 DFA 状态集合中:  
            将该状态闭包加入 DFA 状态集合  
            将该状态闭包入队  
        记录状态闭包对应的 DFA 状态  
        记录状态闭包对应的 DFA 状态转换  
    出队
```

// 最小化 DFA

```
建立最小化分析表  
将非终态和终态分别编号为 0 和 1  
while 新生成状态不同:  
    根据 DFA 状态转换表将状态分为不同的等价类  
    根据等价类, 生成新的 DFA 状态转换表  
    根据新的 DFA 状态转换表生成新的 DFA 状态集合
```

4.2.3 解决问题

- 对于图的数据结构的设计, 最开始将状态和边分别存储, 这样会导致遍历时需要多次循环, 效率低下。最终更改为邻接表的存储, 提高了效率。
- 在寻找闭包等需要递归操作时, 采用了队列的方式, 将递归转换为迭代, 提高了效率。
- 最小化 DFA 时, 路径的对应关系复杂, 容易出错

4.3 语法分析

4.3.1 基本思路

1. 采用 LR(1) 分析法
2. 对文法进行拓广
3. 根据文法构建 DFA
4. 根据 DFA 构建 LR(1) 分析表

4.3.2 算法流程

```
// 对文法进行拓广
增加一个新的开始符号 S' 和一个新的产生式 S' -> S

// 根据文法构建 DFA
建立状态栈用于存储状态，建立状态转换表用于存储状态转换关系
构建开始状态闭包以及对应搜索符
while 有新状态生成：
    for 每一条产生式：
        根据 dot 位置和搜索符计算新状态闭包
        if 状态核心项相同：
            指向该状态
        else:
            新建状态

// 根据 DFA 构建 LR(1) 分析表
建立 LR(1) 分析表
```

4.3.3 解决问题

- 在寻找相同状态时，应当是核心项中所有产生式完全相同，之前产生错误即只比较了其中一项，导致状态减少
- 在求解 First 和 Follow 集合时，需要考虑空串的情况，即将当前符号后所有符号遍历
- 在构建 DFA 时，状态添加新文法需要将文法闭包添加到状态闭包中，否则会出现错误

4.4 语义分析

4.4.1 基本思路

1. 采用递归下降分析法
2. 遍历语法树，递归进行语义分析
3. 采用深度优先遍历

4.4.2 算法流程

```
void DFA(root):
    if root is leaf:
        root.value = token.value
```

```
for each child:
    DFA(child)
    root.value = root.value + child.value
```

5 结果

5.1 文件目录结构

```
compiler
├── grammer_analysis
│   ├── include
│   └── src
├── input
│   ├── grammer.txt
│   └── lextion.txt
├── lexical_analysis
│   ├── include
│   └── src
├── output
│   ├── analysis_process.txt
│   ├── grammer_tree.dot
│   └── lex_result.txt
├── report
├── main.cpp
├── sample.tan
├── .gitignore
└── CmakeLists.txt
```

5.2 使用方法

1. 在终端打开根目录
2. 依次输入以下命令

```
mkdir build
cmake -G "MinGW Makefiles" -Bbuild . // -G参数指定生成器，可以去掉
cd build
make
```

3. 在根目录下会生成 main.exe

4. 终端回到根目录，运行程序

```
cd ..  
./main.exe
```

5.3 实际例子的输入输出效果

5.3.1 输入

- 词法分析./input/lexicon.txt
- 语法分析./input/grammer.txt
- 分析源程序./sample.tan

```
int main(int argc, char* argv[100])  
{  
    static double pi = 3;  
    const int a = 7;  
    const int b = 9;  
    int c = a+b;  
    int d[10] = {6,54,52,34,55,68,10,90,78,56};  
    int i;  
    int sum=0;  
  
    if(a+b>10)  
    {  
        printf("true");  
    }  
    else  
    {  
        printf("false");  
    }  
  
    for (i=0;i<10;i++)  
    {  
        sum=sum+d[i];  
    }  
  
    printf(c);  
    printf(sum);  
    printf("HelloWorld");  
    return 0;  
}
```

5.3.2 输出

- token 序列./output/lex_result.txt

- 分析过程./output/analysis_process.txt
- 语法树./output/grammer_tree.txt



图 5.1: 语法树缩略图

5.3.3 查看语法树

1. 安装 Graphviz
2. 在终端打开 output 目录
3. 在终端输入以下命令

```
// 图片较大, 建议生成 svg 图片, 也可根据需要生成对应的格式
// 复制命令注意符号格式不同
dot -Tsvg grammer_tree.dot -o grammer_tree.svg
```

6 项目分工

- 谭植文 65%:
主要完成了主程序、词法分析和语法分析的代码编写
- 刘晔宁 35%:
主要完成了词法分析规则、部分 C 语言文法和报告的撰写

参考文献

- [1] Author. Article Title. <http://www.arxiv.org>, 2023.
- [2] Author. Project. <http://www.baidu.com>, 2023.
- [3] Author. 网络笔记标题. <https://blog.csdn.net>, 2023.