

编译原理课程设计

代码说明文档

罗文水

计算机科学与工程学院

专业：计算机科学与技术

2021. 05

目录

一 文档结构说明	3
二 问题一：词法分析文档说明	3
三 问题二：语法分析文档说明	4
四 问题三：语义分析文档说明	5
五 程序运行方法	5
5.1 词法分析.....	5
5.2 语法分析	6
5.3 语义分析	7
六 程序使用与扩展方法	8

一 文档结构说明

本次课程设计中涉及的文档以及结构如下所示：

1.1 文件夹 lexical:

- 1.1.1 文件 code.txt
- 1.1.2 文件 code2.txt
- 1.1.3 文件 code3.txt
- 1.1.1 文件 lexical-rule.txt
- 1.1.2 文件 lexical-rule2.txt
- 1.1.3 文件 lexical-rule3.txt
- 1.1.4 代码文档 lexicalAnalysis.cpp
- 1.1.5 头文件 lexicalAnalysis.h
- 1.1.6 文件 problem3_code.txt
- 1.1.7 文件 token.txt

1.2 文件夹 grammer

- 1.2.1 文件 error_file.txt
- 1.2.2 文件 grammer_rule.txt
- 1.2.3 文件 grammer_rule2.txt
- 1.2.4 代码文档 grammerAnalysis.txt
- 1.2.5 头文件 grammerAnalysis.h
- 1.2.6 文件 test_rule.txt
- 1.2.7 文件 test_rule2.txt
- 1.2.8 文件 test_rule3.txt
- 1.2.9 文件 test_rule4.txt
- 1.2.10 文件 test_rule5.txt
- 1.2.11 文件 test_rule6.txt

1.3 文件夹 semantic

- 1.3.1 文件 error_file.txt
- 1.3.2 文件 rule.txt
- 1.3.3 代码文档 semanticAnalysis.cpp
- 1.3.4 头文件 semanticAnalysis.h
- 1.3.5 测试文件 test1.cpp

二 问题一：词法分析文档说明

词法分析程序文件主要为 lexicalAnalysis.cpp，其中大部分数据结构抽离并放置于 lexicalAnalysis.h 中。

在 lexicalAnalysis.cpp 中，有两个宏分别用于测试和调试输出，如下所示：

```
#ifndef DEBUG
|   #define DEBUG
#endif

#ifndef TEST
|   #define TEST
#endif
```

将其中 DEBUG 宏取消注释之后将会得到 NFA 转 DFA 过程描述，即子集法算法每一步的执行过程（包括标记 DFA 节点，弧转换等）。

将其中 TEST 宏取消注释之后将会得到四个单项 DFA 的测试输出结果，即 cpp 文档中构造的测试序列（标识符、运算符、限定符、常量）分别在 DFA 上的测试结果，返回结果是 false 或者 true 表示符号串是否符合词法分析文件 lexical-rule2.txt 中指定的规则。程序的一部分测试输出如下所示（测试科学计数法）：

```
4234 : true
44323e10 : true
.1E10 : true
.1e-10 : true
103213.3213e+10 : true
.3123 : true
4234i : true
44323e10i : true
.231E10i : true
.312e-10i : true
.3213e+10i : true
e10 : false
e++10 : false
. : false
e : false
.43f3e10 : false
.e10 : false
```

将两个宏同时注释掉之后将会得到程序需要的输出，包含读取的产生式，字母表，非终结符对整数的索引表，NFA 转移表，DFA 状态转移表以及 token 序列。

词法分析的产生式规则输入文档为 lexical-rule2.txt。其中包括四个规则，分别对应于常量、标识符、限定符和运算符规则，对于每一类，第一个符号对应于开始符号，而由于三型文法，终止状态只有一个，故无需指定，由程序制定（以非终结符数量为标准进行自动指定）。

本文中的产生式规则应该满足：对任意产生式，左侧只有一个非终结符，右侧非终结符在左，终结符号在右，且都是一个字符。

三 问题二：语法分析文档说明

词法分析主要程序代码在 grammarAnalysis.cpp 中，其中大部分数据结构抽离在

grammerAnalysis.h 中。在 cpp 程序文件中，同样有一个 DEBUG 宏用于控制程序的过程性输出，本程序中的 DEBUG 宏用于控制如下三种过程性输出：

- 求 FIRST 集合算法过程中，对于每个非终结符是否能推出 epsilon 的判断，该算法分为三个阶段进行迭代，直到所有的非终结符都确定是否能推出 epsilon 则终止迭代。
- FIRST 集合求解的迭代过程，该过程中算法一共有五种情况，一旦 first 集合发生变化就控制输出，直到 FIRST 集合全部收敛。
- 可归前缀 DFA 状态转移图，使用 map 进行存储，debug 过程输出 map 中每个节点可经过弧转换到达状态的信息（以该信息构建 ACTION 和 GOTO 表）。

本问题中，error_file.txt 用于输出错误信息。test-rule.txt、test-rule2.txt、test-rule3.txt 用于测试符号的 FIRST 集合是否求解正确（经过本算法的测试，能正确求解所有合法二型文法的 FIRST 集合）。其次，test-rule4.txt、test-rule5.txt、test-rule6.txt 用于测试项目集规范族求解算法以及 LR(1) 分析是否正确，经过本程序的测试，所有合法 LR(1) 文法均能通过测试并自动构建可归前缀自动机。

注意：使用测试 FIRST 集合的文件进行测试时会在随后求 LR 过程中报 assertion 错误，因为只求 FIRST 集合而不指定 LR 分析串会导致本人定义的 assert 错误，并导致操作系统自陷。但过程中对于 FIRST 集合等的求解结果都将在 assertion 错误之前显示。测试项目集规范族算法时，ACTION 和 GOTO 表也将在 assertion 错误之前显示。

注：所有的测试样例都为《编译原理》教材中的典型例子。

在正确的按照流程执行问题一、二的代码后，将不会遇到 assertion 错误，并显示正确结果。

本程序的产生式规则文件为 **grammer-rule2.txt**，该文档中规定产生式规则，非终结符号使用英文双引号进行包围，终结符号使用单引号进行标注。文档结尾有一个\$符号表示产生式录入结束。Grammer-rule.txt 为产生式测试文件，其中的产生式为部分 C 语言的产生式，测试正确后方修改为自定义语言产生式。

四 问题三：语义分析文档说明

语义分析文档根据问题二 LR(1) 分析法程序进行二次开发。产生式规则在 semantic 文件夹下的 rule.txt 中指定（属性文法）。错误信息提示文件在 semantic 文件夹下的 error_file.txt 文件中指定。

五 程序运行方法

5.1 词法分析

进入目录 lexical，进入 lexicalAnalysis.cpp 文件，该程序使用 C++ 编写，编辑环境为 vscode，使用 **C++11 标准**，使用任何可以运行 **C++11 标准** 代码的环境并且 **点击运行** 即可得到结果（保证在环境中能够链接到目录下的 txt 文件以及头文件）。对于本词法分析，使用的分析程序在 code.txt 中。正确运行后会得到如下 tokenList（部分）并且该结果将会保存到 token.txt 文件中，若出现错误则错误信息将打印在控制台：

token 表:

```
(1, 关键字 , 'func' )
(1, 标识符 , 'main' )
(1, 限定符 , '(' )
(1, 关键字 , 'Void' )
(1, 标识符 , 'args' )
(1, 限定符 , ';' )
(1, 关键字 , 'Double' )
(1, 标识符 , 'args2' )
(1, 限定符 , ';' )
(1, 关键字 , 'String' )
(1, 标识符 , 'args3' )
(1, 限定符 , ')' )
(1, 运算符 , '-' )
(1, 运算符 , '>' )
(1, 关键字 , 'Void' )
(1, 限定符 , '{' )
(2, 标识符 , 'function' )
```

说明: 其余文档 lexical_rule 、lexical_rule3 文档都是产生式的测试文档，在运行时不必考虑。而 code2.txt 与 code3.txt 也是程序编写过程中的测试文档，若不修改代码，则无需考虑，直接运行默认程序。

5. 2 语法分析

前提条件：执行完词法分析程序，并且不改变默认的文件目录。

执行方法：进入 grammer 文件夹中，进入 grammerAnalysis.cpp 文件，该程序使用 C++ 编写，编辑环境为 vscode，使用 **C++11 标准**，使用任何可以运行 **C++11 标准** 代码的环境并且**点击运行**即可得到结果（保证在环境中能够链接到目录下的 txt 文件以及头文件）。使用默认的程序以及语法分析的产生式，直接运行得到程序的分析结果。包括 ACTION 和 GOTO 分析表以及程序的 LR(1) 识别过程，若出现错误将会将错误同时显示在控制台以及 error_file.txt 文件中。结果部分如下图所示：

```
278 : { 0 , 2 , 3 }      # FUNCTION func      test () -> String { } #      S5(action)
279 : { 0 , 2 , 3 , 5 }   # FUNCTION func [identifier]      () -> String { } #      S6(action)
280 : { 0 , 2 , 3 , 5 , 6 } # FUNCTION func [identifier] ( ) -> String { } #      S7(action)
281 : { 0 , 2 , 3 , 5 , 6 , 7 } # FUNCTION func [identifier] ( )      -> String { } #      S19(action)
282 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 } # FUNCTION func [identifier] ( ) -> String { } #      S24(action)
283 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 } # FUNCTION func [identifier] ( ) -> String { } #      S31(action)
284 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 , 31 } # FUNCTION func [identifier] ( ) -> String { } #      r6(ac
285 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 , 27 } # FUNCTION func [identifier] ( ) -> DATATYPE { } #      S38
286 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 , 27 , 38 } # FUNCTION func [identifier] ( ) -> DATATYPE { } #      S38
287 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 , 27 , 38 , 40 } # FUNCTION func [identifier] ( ) -> DATATYPE { SENTENCE
288 : { 0 , 2 , 3 , 5 , 6 , 7 , 19 , 24 , 27 , 38 , 40 , 58 } # FUNCTION func [identifier] ( ) -> DATATYPE { SENTENC
289 : { 0 , 2 , 2 }      # FUNCTION FUNCTION      #      r1(action)      4(goto)
290 : { 0 , 2 , 2 , 4 }   # FUNCTION FUNCTION X     #      r2(action)      4(goto)
291 : { 0 , 2 , 4 }      # FUNCTION X          #      r2(action)      1(goto)
292 : { 0 , 1 }          # X          #      acc(action)
成功识别
程序的运行时间 (含I/O): 2.89575s
```

5.3 语义分析

前提条件：修改 lexical/lexicalAnalysis.cpp 中的字符串常量定义（文件名称定义），如下图所示，将其中的 code_file=initial_code，修改为 code_file=problem3 以链接问题三的计算式文件（从而生成问题三的输入 token 词例表）。在 **C++11 环境**下运行代码。

```
const string problem3 = "problem3_code.txt";
const string initial_code = "code.txt";
const string code_file = initial_code;
```

计算式书写文件：lexical/lexicalAnalysis.cpp 文件（需符合指定文法，否则会提示错误）。

执行方法：进入 semantic/semanticAnalysis.cpp 文件中，在 **C++11 环境**下运行代码即可得到程序输出，程序输出包括 LR (1) 分析过程以及语义动作、最终的四元式序列。四元式的输出结果应当如下所示（默认表达式的分析，有两种表述形式的四元式，情况分别列出）：

```
t1:=10
t2:=t1
t3:=t2
t4:=10
t5:=t4
t6:=321
t7:=t6
t8:=t7
t9:=213
t10:=t9
t12:=321
t13:=t12
t15:=t14
t17:=321
t19:=t18
t20:=t19
t21:=t20
t22:=32
t23:=t22
t24:=t23
t25:=321
t26:=t25
t28:=t27
t31:=t30
t32:=t31
t33:=t32
```

```
( := ,10 , _ , t1 )
```

```
( := ,t1 ,_ ,t2 )
( := ,t2 ,_ ,t3 )
( := ,10 ,_ ,t4 )
( := ,t4 ,_ ,t5 )
( := ,321 ,_ ,t6 )
( := ,t6 ,_ ,t7 )
( := ,t7 ,_ ,t8 )
( := ,213 ,_ ,t9 )
( := ,t9 ,_ ,t10 )
( - ,t8 ,t10 ,t11 )
( := ,321 ,_ ,t12 )
( := ,t12 ,_ ,t13 )
( + ,t11 ,t13 ,t14 )
( := ,t14 ,_ ,t15 )
( * ,t5 ,t15 ,t16 )
( := ,321 ,_ ,t17 )
( * ,t16 ,t17 ,t18 )
( := ,t18 ,_ ,t19 )
( := ,t19 ,_ ,t20 )
( := ,t20 ,_ ,t21 )
( := ,32 ,_ ,t22 )
( := ,t22 ,_ ,t23 )
( := ,t23 ,_ ,t24 )
( := ,321 ,_ ,t25 )
( := ,t25 ,_ ,t26 )
( + ,t24 ,t26 ,t27 )
( := ,t27 ,_ ,t28 )
( * ,t21 ,t28 ,t29 )
( + ,t3 ,t29 ,t30 )
( := ,t30 ,_ ,t31 )
( := ,t31 ,_ ,t32 )
( := ,t32 ,_ ,t33 )
```

六 程序使用与扩展方法

程序对于任意词法规则的构造与扩展需要满足如下条件：

- 词法分析中产生式规则仅修改 lexical-rule2.txt 文件，其中每一类规则的修改都在指定区域内，比如对于常量的产生式修改必须在[scientific-format]以及下一个类型的表示符号之间修改。
- epsilon 使用@符号表示。
- 产生式右侧，第一个符号是非终结符，第二个符号是终结符，产生式左边第一个符号是非终结符，且每一个符号只占一个字符（本文档书写规则，同时也是三型文法的要求，非终结符与终结符号的大小写无关，只关注位置）。

程序对于任意语法分析规则的构造与扩展需要满足如下条件:

- 语法规则在 grammer-rule2.txt 文件中修改。
- 产生式满足非终结符用英文双引号进行包括，终结符使用英文单引号进行包括。
- 标识符务必使用[identifier]并使用英文单引号进行包括。
- epsilon 使用@符号表示。并且使用英文单引号进行包括。
- 若修改成自身指定规则的语言，需关注词法和语法分析程序头文件中的 keywordList 表，修改其中的关键字，否则可能导致关键字识别成标识符的问题。

南京理工大学

《编译原理课程设计》报告

姓名: 罗文水 学号: 918106840738

学院(系): 计算机科学与工程学院

专业: 计算机科学与技术

实验名称: 课程设计 2 编译原理

2021 年 5 月

目录

一 设计要求与缘由	4
1.1 设计要求.....	4
1.2 设计缘由.....	5
二 词法分析	5
2.1 产生式规则与解释.....	5
2.2 数据结构	10
2.2.1 DFA 单个状态数据结构 DFASState	10
2.2.2 DFA 数据结构 DFA	10
2.2.3 NFA 数据结构 NFA	10
2.2.4 Token 类型数据结构	11
2.2.5 Token 数据结构	11
2.2.6 DFA 处理数据结构 Processor.....	11
2.3 词法分析中的算法与程序解释.....	13
2.3.1 产生式读取算法.....	14
2.3.2 epsilon 闭包求解算法---广度优先搜索	15
2.3.3 子集法求 DFA	15
2.4 案例分析.....	16
2.4.1 NFA 转 DFA 测试	16
2.4.2 词法分析正确性测试	16
三 语法分析	19
3.1 产生式规则与解释.....	19
3.2 数据结构.....	21
3.2.1 符号数据结构 Symbol.....	21
3.2.2 产生式数据结构 production.....	22
3.2.3 产生式集成数据结构 allProduction.....	23
3.2.4 项目数据结构 Item.....	24
3.2.5 项目集数据结构 ItemSet.....	25
3.2.6 ACTION 和 GOTO 元素数据结构 Element	26

3.2.7 可归前缀 DFA 构造数据结构 Processor.....	26
3.3 语法分析中的算法与程序解释.....	28
3.3.1 整体程序流程.....	28
3.3.2 标记非终结符能否推出 epsilon 算法.....	29
3.3.3 求所有文法符号串的 FIRST 集算法.....	30
3.3.4 项目集规范族构造算法与 ACTION & GOTO 表构造算法.....	30
3.3.5 LR(1) 符号串分析算法与实现方案.....	32
3.4 案例分析.....	32
3.4.1 求 FIRST 算法书上案例.....	32
3.4.2 求 LR(1) 项目集规范族算法案例	34
四 语义分析.....	36
4.1 产生式规则与解释.....	36
4.2 数据结构.....	37
4.2.1 扩展产生式数据结构 production	37
4.3 表达式语义分析中算法.....	38
4.3.1 属性文法读入算法.....	38
4.3.2 LR(1) 分析算法以及四元式构造算法.....	38
五 推广与改进.....	43
六 总结与心得.....	43

一 设计要求与缘由

1.1 设计要求

本次课程设计有如下三项任务：

任务一：创建一个词法分析程序，该程序支持分析常规单词。

必须使用 DFA 或者 NFA 来实现此程序。程序有两个输入：一个文本文档，包括一组 3 型文法的产生式；一个源代码文本文档，包括一组需要识别的字符串（程序代码）。程序的输出是一个 token 表，该表由 5 种 token 组成：关键词、标识符、常量、限定符和运算符。

词法分析的推荐处理逻辑：根据用户输入的正规文法，生成 NFA，再确定化生成 DFA，根据 DFA 编写识别 token 的程序，从头到尾从左至右识别用户输入的源代码，生成 token 列表（三元组：所在行号，类别，token 内容）。

要求：词法分析程序可以准确识别：科学计数法形式的常量，复数常量，可检查整数常量的合法性，标识符的合法性，尽量符合真实常用高级语言的规则。

任务二：创建一个使用 LL(1) 或者 LR(1) 方法的语法分析程序。

程序有两个输入：1) 一个是文本文档，其中包含 2 型文法的产生式集合；2) 任务一词法分析程序生成的 token 令牌表。程序的输出包括：YES 或者 NO（源代码字符串符合此 2 型文法，或者源代码字符串不符合此 2 型文法）；错误提示文件，如果有语法错误，标示出错行号并给出大致出错的原因。

语法分析程序的推荐处理逻辑：根据用户输入的 2 型文法生成 ACTION 和 GOTO 表，设计合适的数据结构，判断 token 序列。

任务三：创建符合属性文法规则的语义分析程序。

程序有两个输入：1) 一个是文本文档，其中包含 2 型文法（上下文无关文法 + 属性文法，包含语义规则注释，可以简单以表达式计算语义为例）的产生式集合；2) 任务一词法分析程序输出的 token 令牌表。

程序输出：四元式序列，可以利用优化技术生成优化后的四元式序列。

1.2 设计缘由

编译技术（或编译器、编译系统）在计算机科学与技术的发展历史中发挥了巨大作用，是计算机系统的核心支撑软件。“编译原理”一直以来也是国内外大学计算机相关专业的重要课程，其知识结构贯穿程序设计语言、系统环境以及体系结构，能以相对独立的视角体现从软件到硬件以及软硬件协同的整机概念；同时，其理论基础又涉及形式语言与自动机、数据结构与算法等计算机学科的许多重要方面，不愧为联系计算机科学理论与计算机系统的典范。计算机科学家 A. V. Aho 和 J. D. Ullman 在他们的著作中说道：“在每一个计算机科学家的研究生涯中，这些原理和技术都会反复用到。”

本次编译原理课程设计是基于“编译原理”课程的一次扩展，提出的三个问题分别涉及词法分析，语法分析，语义分析等众多方面，在设计过程中，我们能够通过代码加深对编译原理各个过程的理解，对编译原理的基础理解与实现水平将会得到提升，同时也加深对算法与数据结构的理解，加强编码能力与实现能力。

二 词法分析

2.1 产生式规则与解释

词法分析产生文件 `lexical-rule2.txt` 文件中，产生式规则如下：

[scientific-format]

A->dB

A->.G

B->dB

B->.C

B->eD

B->ED

B->@

B->i

C->dC

C->eD

C->ED

C->@
C->i
D->+E
D->-E
D->dF
E->dF
F->dF
F->i
F->@
G->dC
[delimiter]
A->,
A->;
A->:
A->(
A->)
A->{
A->}
A->[
A->]
A->#
A->'
A->"
[operator]
A->+
A->-
A->*
A->/
A->%
A->=
A-><E
E-><
E->@
A->>F

F->>
F->@
A->+B
A->-B
A->*B
A->/B
A->%B
A->=B
A-><B
A->>B
B->=
A->+C
C->+
A->-D
D->-
A->&G
G->&
G->=
G->@
[identifier]
A->aB
A->_B
B->aB
B->dB
B->_B
B->@
[end]

该文件中共包含四种类型 token 的产生式规则，标识符号 [scientific-format] 表示常量，标识符号 [delimiter] 表示限定符，标识符号 [operator] 表示运算符，标识符号 [identifier] 表示标识符。对于关键字类型的 token，识别方案是按照标识符识别文法进行识别，每次识别出标识符之后判断是否是关键字。从而将其判断为标识符类型或者关键字类型。

在文法的书写规则中，使用 a 表示 a-z 以及 A-Z 之间的任意字符，使用 d 表

示 0-9 之间的任意数字。

根据上述文法，每一类的 NFA 自动机形式分别如下所示：

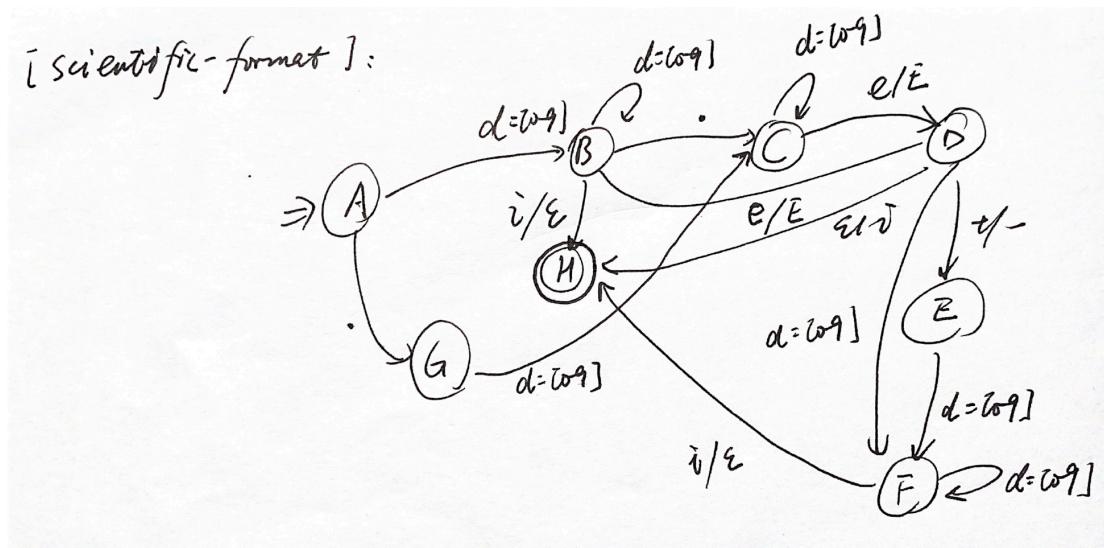


图 1. 常量识别 NFA (根据文法绘制)

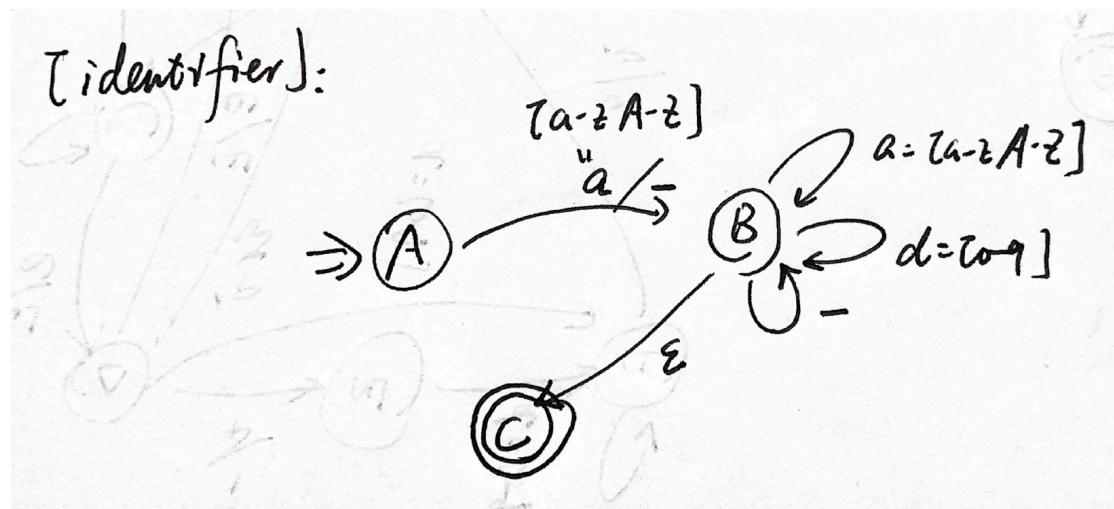


图 2. 标识符识别 NFA (根据文法绘制)

[operator]:

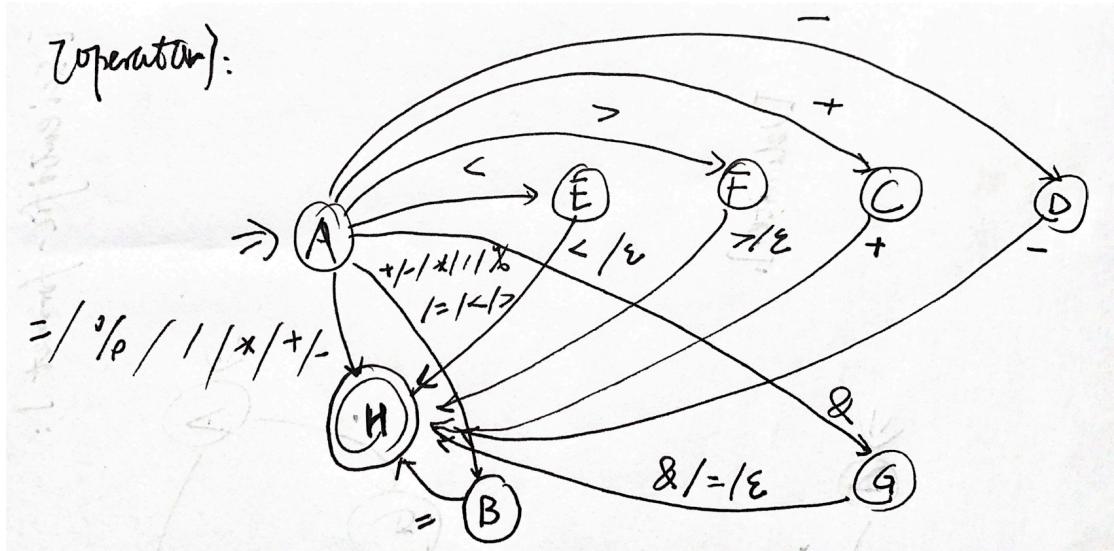


图 3. 运算符识别 NFA (根据文法绘制)

[delimiter]:

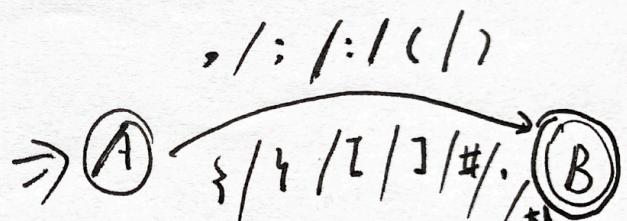


图 4. 限定符识别 NFA

从图 1 中可以得知，本词法分析程序对于常量的识别规则有如下几种：科学计数法形式：如 3123、121e12, 12e+312, 23e-12 等，复数形式的常量，如 123i, 21321e+12i 等形式。

此外，从图 2 中可以得知，本词法分析程序设定的标识符产生式识别规则为：标识符开头是由下划线_或者大小写英文字母，随后的字符由大小写英文字母、下划线_或者数字 0-9 构成。

从图 3 中可以得知，本词法分析程序设定的运算符识别规则是可以识别如下符号：< / << / > / >> / + / ++ / - / -- / & / &= / && / + / - / * / / / % / =。任意自定义的运算符都能通过修改文法得到。

从图 4 中可以得知，本词法分析程序设定的标识符识别规则是可以识别如下符号：, / ; / : / (/) / { / } / [/] / # / ' / "。

2.2 数据结构

本词法分析中主要用到的数据结构有如下几种：

2.2.1 DFA 单个状态数据结构 DFAState

数据结构的定义如下所示：

```
struct DFAState {  
    bool marked;  
    vector<int> states;  
    map<char,int> moves;  
};
```

其中 marked 表示当前状态在进行子集法（求与 NFA 等价的 DFA）算法时是否已经考虑过弧转换，考虑过则标记 marked=true，否则标记 marked=false；

states 表示当前 DFA 状态表示的原 NFA 中的状态集合。使用 vector 容器易于对元素数量进行管理以及运用 C++STL 中的函数。

moves 中存放的是当前状态经过弧 a 转换之后到状态 b 构成的对 (a, b)。使用 map<char, int>数据结构，弧转移时转移弧为字符类型，转移之后所能到达的状态是 int 类型 DFA 状态编号。

2.2.2 DFA 数据结构 DFA

数据结构的定义如下所示：

```
typedef map<int, DFAState> DFA;
```

map<int,DFAState>数据结构表示每一个 DFA 状态 int 形式的索引都对应于一个 DFAState 类型的数据结构。从而将 DFA 每个状态与 DFAState 建立映射。

2.2.3 NFA 数据结构 NFA

NFA 数据结构如下所示：

```
typedef map<int, map<char, vector<int>>> NFA;
```

表示每一个 NFA 状态都对应于一个 map<char, vector<int>>映射，表示每个状态能经过一个 char 类型的弧转换到一系列状态，这些状态使用 vector<int> 数据结构保存。

2.2.4 Token 类型数据结构

Token 的类型数据结构表示如下：

```
enum TokenType{constant,identifier,keyword,delimiter,operand};
```

表示 Token 令牌的类型有如下四种：constant（常量）、identifier（标识符）、keyword（关键字）、delimiter（限定符）、operand（操作符）。使用 enum 类型容易通过变量名对令牌类型进行管理。

2.2.5 Token 数据结构

```
struct token{
    int line;//所在行号
    int type;//类型
    string value;//取值
    token(int line,int type,string value):
        line(line),type(type),value(value){}
    token(){}
};
```

token 数据结构中包含变量 line: 表示当前的 token 所在的行号；
type 表示当前的 token 类型，通过 2.1.3 中的 TokenType 进行指定，分别为任务一要求的五种类型之一。
value: 表示 token 令牌在程序文档中的实际取值。
token 构造函数：包括无参以及有参函数对 token 实例化。

2.2.6 DFA 处理数据结构 Processor

```
class Processor{
public:
    vector<string> rules;
    int nfa_init_state;
    int nfa_state_count;
    int dfa_state_count;
    vector<int> nfa_final_states;
    set<int> dfa_final_states;
```

```

vector<char> alphabet;
NFA nfa;
DFA dfa;
int to_mark;
map<char,int> indexTable;

public:
    bool vectorContain(vector<int> vec, int key);
    int is_a_state_of_DFA(vector<int> state);
    vector<int> eclosure(vector<int> T);
    vector<int> move(vector<int> T, char move);
    DFAState newDFAState(bool mark, vector<int> s);
    vector<int> findFinalDFAStates();
    void subsetMethod();
    void printDFA();
    void printNFA();
};


```

(1) 变量解释

- rules: vector<string> 类型, 用于存放从规则文档中得到的产生式, 以 string 类型存储在动态数组 vector 类型中。
- nfa_init_state: int 类型, 表示 NFA 初始的状态索引号。
- nfa_state_count: int 类型, 表示 NFA 的状态总数。
- dfa_state_count: int 类型, 表示 DFA 的状态总数。
- nfa_final_states: vector<int> 类型, 表示 DFA 终止状态集合 (在本词法分析的三型文法中, 实际上 NFA 终止状态只有一个)。
- dfa_final_states: set<int> 类型, 表示 DFA 终止状态集合, 使用 set 容器可以去除相同元素, 保持 set 集合中元素的唯一性。
- alphabet: vector<char> 类型, 表示符号表, 从产生式文法中自动总结出非终结符符号表。
- nfa: NFA 类型, 表示从产生式规则文件中自动识别出的不确定有限自动机, 等待被处理器的算法转变成 DFA。
- dfa: DFA 数据类型, 表示由 NFA 生成的 DFA。

- `to_mark`: int 类型, 表示下一个等待被标记的 DFA 的状态标号, 在运行子集法算法中作为控制变量使用。
- `indexTable`: map<char,int>类型, 表示非终结符号的索引表, 将非终结符转换成 int 类型的索引, 从 0 开始标注索引。易于对非终结符进行随机访问。

(2) 函数解释

- ✓ `bool vectorContain(vector<int> vec, int key)`: 判断容器 vector 中是否存在某个元素。
- ✓ `int is_a_state_of_DFA(vector<int> state)`: 判断一个向量 state 是否是 DFA 中已经标记过的一个状态。求子集法的辅助函数。
- ✓ `vector<int> eclosure(vector<int> T)` : 对状态 T 求 epsilon 闭包函数。
- ✓ `vector<int> move(vector<int> T, char move)` : 对状态 T 求经过 move 弧转换之后的状态集合。
- ✓ `DFAState newDFAState(bool mark, vector<int> s)` : 构造一个新的 DFA 状态并且设置当前 DFA 状态的标志位为 mark。状态值为集合 s。
- ✓ `vector<int> findFinalDFAStates()` : 求出 DFA 所以的终止状态, 以数组形式返回。
- ✓ **`void subsetMethod()`** : 最主要的函数, 用于运行子集法从 NFA 构建 DFA。
- ✓ `void printDFA()` : 在控制台输出 DFA, 包括其初始状态, 终止状态以及转移图等。
- ✓ `void printNFA()` : 在控制台输出 NFA。包括其初始状态, 转移表等。

2.3 词法分析中的算法与程序解释

本词法分析程序的总体流程如图 5 所示, 通过该流程, 没成功识别一个符号即打印到控制台并且输入到文件 token.txt 中。

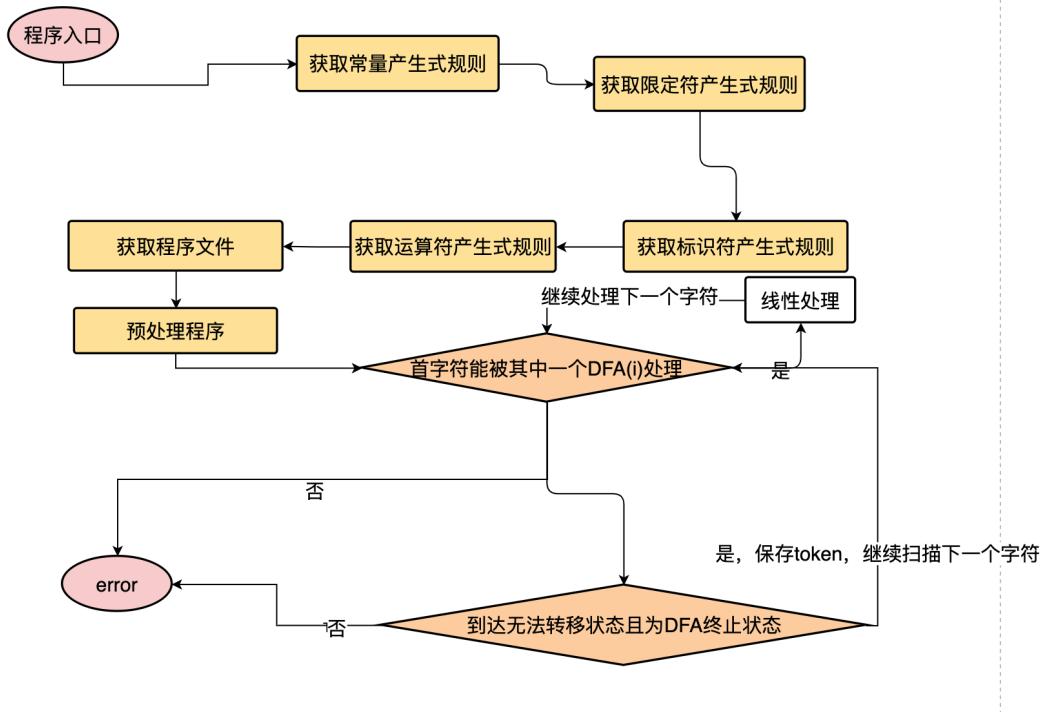


图 5. 词法分析总体流程

2.3.1 产生式读取算法

产生式读取算法的流程如图 6 所示，其中

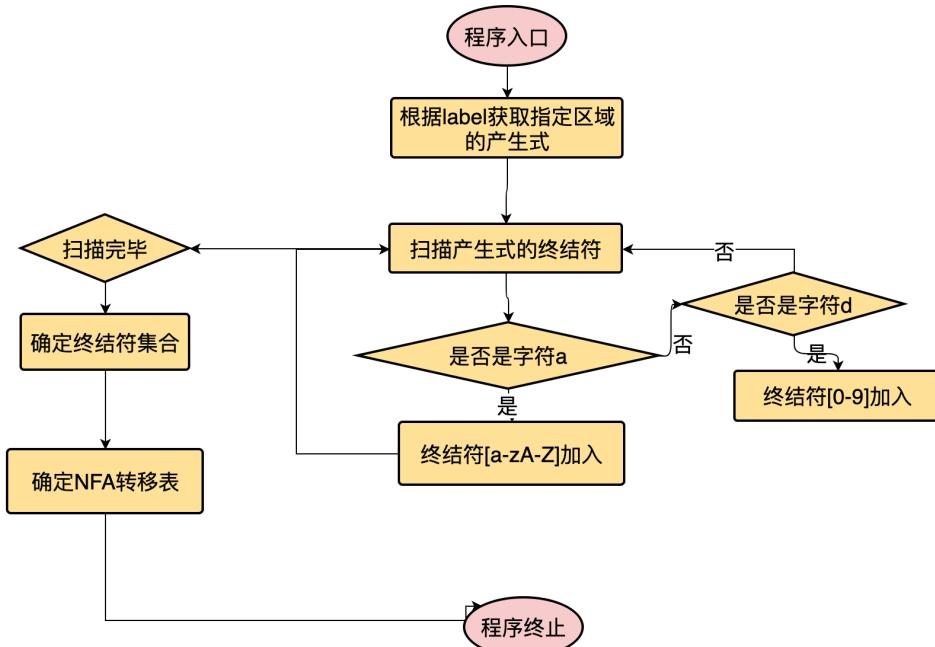


图 6. 产生式读取算法

2.3.2 epsilon 闭包求解算法---广度优先搜索

使用数据结构：队列；

算法输入： 需要求闭包的 $\text{vector} < \text{int} >$ T

算法输出： epsilonClosure(T)

算法流程：

1. 初始化时队列 q 中只有传入的 T 中的元素；定义集合 A，将 T 中元素放入 A 中；
2. q 为空则结束，至 5；否则至 3；
3. 取 q 中的元素 a；检查 a 的 epsilon 弧转换结果 b；如果 A 集合中不包含弧转换的结果 b，将 b 加入 A 集合，同时加入 q；
4. 跳转至 2；
5. 将 A 集合转换成 $\text{vector} < \text{int} >$ 返回。

2.3.3 子集法求 DFA

算法输入： NFA N(K, S_1, f, K0, Kt)

算法输出： DFA M(S, S_2, D, S0, St)

算法流程：

(1)

1. 开始，令 epsilon-closure(K0) 为 C 中的唯一成员，并且它是未标记的。
2. While(C 中存在尚未标记的子集 T) do

 标记 T:

```
        for 每一个输入字母 a do
            U:=epsilon-closure(move(T, a));
            if U 不在 C 中 then
                将 U 作为未被标记的子集加入 C 中
            end if
        end for
    end While
```

(2) M 和 N 的输入字母表是一致的；

(3) $S_0 = \text{epsilon-closure}(K_0)$;

(4) DFA 终态 S_t 为 DFA 状态中与 NFA 终态交集不为空集的状态;

2.4 案例分析

2.4.1 NFA 转 DFA 测试

本词法分析程序中，一个将 NFA 转化成 DFA 的输出如图 7 所示。

状态总数: 8										
开始状态: 1 终止状态: 8										
NFA: % & * + - / < = > @										
1	{2,8}	{7}	{2,8}	{2,3,8}	{2,4,8}	{2,8}	{2,5}	{2,8}	{2,6}	{}
2	{}	{}	{}	{}	{}	{}	{8}	{}	{}	
3	{}	{}	{}	{8}	{}	{}	{}	{}	{}	
4	{}	{}	{}	{}	{8}	{}	{}	{}	{}	
5	{}	{}	{}	{}	{}	{8}	{}	{}	{8}	
6	{}	{}	{}	{}	{}	{}	{}	{8}	{8}	
7	{}	{8}	{}	{}	{}	{}	{8}	{}	{8}	
8	{}	{}	{}	{}	{}	{}	{}	{}	{}	
初始状态集合: {0}										
终止状态集合: {1,2,3,4,5,6,7}										
DFA: % & * + - / < = >										
0	{1}	{2}	{1}	{3}	{4}	{1}	{5}	{1}	{6}	
1	{}	{}	{}	{}	{}	{}	{}	{7}	{}	
2	{}	{7}	{}	{}	{}	{}	{}	{7}	{}	
3	{}	{}	{}	{7}	{}	{}	{}	{7}	{}	
4	{}	{}	{}	{}	{7}	{}	{}	{7}	{}	
5	{}	{}	{}	{}	{}	{}	{7}	{7}	{}	
6	{}	{}	{}	{}	{}	{}	{}	{7}	{7}	
7	{}	{}	{}	{}	{}	{}	{}	{}	{7}	

图 7. NFA 转 DFA 案例

2.4.2 词法分析正确性测试

(1) 测试案例 1

使用如下案例进行词法分析测试（注：由于本文语言自定义，所以在该式子中 Int64 是类型关键字：

Int64 a=1312e-1 + b

测试的 token 表输出为：

(1, 关键字 , 'Int64')

(1, 标识符 , 'a')

(1, 运算符 , '=')

(1, 常量 , '1312e-1')

(1, 运算符 , '+')

(1, 标识符 , 'b')

(2) 测试案例 2

```

func main(Void args,Double args2,String args3) -> Void{
    function((1+342.321),current);
    let number: Int64 = (312321.3123123E10+31321);
    function(.1,31231,321);
    let ans:Int32 = (43242+32423);
    var a:Int32 = 10;
    {
        let _fasd:String = 100;
        let a : Int32=10;
        while a<10 {
            i=(10+10);
            a=(1312E10+1);
        }
        while( b<100 )i=(i+10);
    }
    {
        4232;
        if a<b{
            a = 1;
        }
    }
}

func test()->String{};

```

测试得到 token 表输出为（从左至右，从下至上，词法中关键词在头文件 keywordsList 中指定，其余规则根据产生式确定）：

```

( 1 , 关键字 , 'func' )( 1 , 标识符 , 'main' )( 1 , 限定符 , '(' )
( 1 , 关键字 , 'Void' )( 1 , 标识符 , 'args' )( 1 , 限定符 , ',' )
( 1 , 关键字 , 'Double' )( 1 , 标识符 , 'args2' )
( 1 , 限定符 , ';' )( 1 , 关键字 , 'String' )( 1 , 标识符 , 'args3' )
( 1 , 限定符 , ')' )( 1 , 运算符 , '-' )( 1 , 运算符 , '>' )
( 1 , 关键字 , 'Void' )( 1 , 限定符 , '{' )( 2 , 标识符 , 'function' )
( 2 , 限定符 , '(' )( 2 , 限定符 , '(' )( 2 , 常量 , '1' )

```

(2, 运算符 ,'+')(2, 常量 ,'342.321')(2, 限定符 ,')
(2, 限定符 ,';')(2, 标识符 ,'*current*')(2, 限定符 ,')
(2, 限定符 ,';')(3, 关键字 ,'*let*')(3, 标识符 ,'*number*')
(3, 限定符 ,';')(3, 关键字 ,'*Int64*')(3, 运算符 ,' $=$ ')
(3, 限定符 ,'(')(3, 常量 ,' $312321.3123123E10$ ')(3, 运算符 ,'+')
(3, 常量 ,' 31321 ')(3, 限定符 ,')')(3, 限定符 ,';')
(4, 标识符 ,'*function*')(4, 限定符 ,'(')(4, 常量 ,'.1')
(4, 限定符 ,';')(4, 常量 ,' 31231 ')(4, 限定符 ,')')(4, 常量 ,' 321 ')
(4, 限定符 ,')')(4, 限定符 ,';')(5, 关键字 ,'*let*')
(5, 标识符 ,'*ans*')(5, 限定符 ,';')
(5, 关键字 ,'*Int32*')(5, 运算符 ,' $=$ ')
(5, 限定符 ,'(')(5, 常量 ,' 43242 ')(5, 运算符 ,'+')(5, 常量 ,' 32423 ')
(5, 限定符 ,')')(5, 限定符 ,';')(6, 关键字 ,'*var*')
(6, 标识符 ,'*a*')(6, 限定符 ,';')(6, 关键字 ,'*Int32*')(6, 运算符 ,' $=$ ')
(6, 常量 ,' 10 ')(6, 限定符 ,';')(7, 限定符 ,'{')(8, 关键字 ,'*let*')
(8, 标识符 ,'*_fasd*')(8, 限定符 ,';')(8, 关键字 ,'*String*')
(8, 运算符 ,' $=$ ')(8, 常量 ,' 100 ')(8, 限定符 ,';')
(9, 关键字 ,'*let*')(9, 标识符 ,'*a*')(9, 限定符 ,';')
(9, 关键字 ,'*Int32*')(9, 运算符 ,' $=$ ')(9, 常量 ,' 10 ')
(9, 限定符 ,';')(10, 关键字 ,'*while*')(10, 标识符 ,'*a*')
(10, 运算符 ,' $<$ ')(10, 常量 ,' 10 ')(10, 限定符 ,'{')
(11, 标识符 ,'*i*')(11, 运算符 ,' $=$ ')(11, 限定符 ,'(')(11, 常量 ,' 10 ')
(11, 运算符 ,'+')(11, 常量 ,' 10 ')(11, 限定符 ,')')
(11, 限定符 ,';')(12, 标识符 ,'*a*')(12, 运算符 ,' $=$ ')
(12, 限定符 ,'(')(12, 常量 ,' $1312E10$ ')(12, 运算符 ,'+')
(12, 常量 ,'*t*')(12, 限定符 ,')')(12, 限定符 ,';')
(13, 限定符 ,'}')(14, 关键字 ,'*while*')(14, 限定符 ,'{')
(14, 标识符 ,'*b*')(14, 运算符 ,' $<$ ')(14, 常量 ,' 100 ')
(14, 限定符 ,')')(14, 标识符 ,'*i*')(14, 运算符 ,' $=$ ')
(14, 限定符 ,'(')(14, 标识符 ,'*i*')(14, 运算符 ,'+')

```

( 14 , 常量 , '10' )( 14 , 限定符 , ')' )( 14 , 限定符 , ';' )
( 15 , 限定符 , '}' )( 16 , 限定符 , '{' )( 17 , 常量 , '4232' )( 17 , 限定符 , ';' )
( 18 , 关键字 , 'if' )( 18 , 标识符 , 'a' )( 18 , 运算符 , '<' )
( 18 , 标识符 , 'b' )( 18 , 限定符 , '{' )( 19 , 标识符 , 'a' )( 19 , 运算符 , '=' )
( 19 , 常量 , '1' )( 19 , 限定符 , ';' )( 20 , 限定符 , '}' )
( 21 , 限定符 , '}' )( 22 , 限定符 , '}' )( 23 , 关键字 , 'func' )
( 23 , 标识符 , 'test' )( 23 , 限定符 , '(' )( 23 , 限定符 , ')' )
( 23 , 运算符 , '-' )( 23 , 运算符 , '>' )( 23 , 关键字 , 'String' )
( 23 , 限定符 , '{' )( 23 , 限定符 , '}' )

```

三 语法分析

3.1 产生式规则与解释

本程序中使用的默认产生式规则如下所示。

```

"X"->"X"
"X"-> '@'
"X"->"FUNCTION""X"
"DATATYPE"->'Void'
"DATATYPE"->'Int'
"DATATYPE"->'Double'
"DATATYPE"->'String'
"DATATYPE"->'Int8'
"DATATYPE"->'Int16'
"DATATYPE"->'Int32'
"DATATYPE"->'Int64'
"FUNCTION"->'func"[identifier]"("PARAMETER_LIST")"->"DATATYPE"{"SENTENCES"}'
"PARAMETER_LIST"->"PARAMETER""OTHERS"
"PARAMETER_LIST"-> '@'
"OTHERS"->',"PARAMETER""OTHERS"
"OTHERS"-> '@'

```

```

"PARAMETER"->'[identifier]":"DATATYPE"
"SENTENCES"-> '@'
"SENTENCES"->"SENTENCE""SENTENCES"
"SENTENCE"->"EXPRESSION";'
"SENTENCE"->"DEFINITION";'
"SENTENCE"->"LOOP"
"SENTENCE"->'[identifier]"("CALLPARAMETER_LIST")";'
"SENTENCE"->"BRANCH"
"SENTENCE"->'{ "SENTENCES"}'
"SENTENCE"->'return';'
"SENTENCE"->'return'"EXPRESSION";'
"CALLPARAMETER_LIST"->"VALUE""OTHERS_2"
"CALLPARAMETER_LIST"-> '@'
"OTHERS_2"->,"VALUE""OTHERS_2"
"OTHERS_2"-> '@'
"BRANCH"->'if'("EXPRESSION")"{"SENTENCES"}'
"BRANCH"->'if'"EXPRESSION"{"SENTENCES"}'
"EXPRESSION"->"VALUE""op""VALUE"
"op"->'>'
"op"->'='
"op"->'<'
"op"->'+''
"EXPRESSION"->"VALUE"
"VALUE"->'('EXPRESSION")'
"VALUE"->'[identifier]'
"VALUE"->'[scientific-format]'
"DEFINITION"->"LIMIT"[identifier]":"DATATYPE"
"DEFINITION"->"LIMIT"[identifier]":"DATATYPE"="EXPRESSION"
"LIMIT"->'let'
"LIMIT"->'var'
"LOOP"->'while'("EXPRESSION")"SENTENCE"
"LOOP"->'while'"EXPRESSION"{"SENTENCES"}'
$
```

对产生式集合的解释如下：

- ✧ 程序可以是空程序，空程序能够通过 LR(1) 分析并被成功识别
- ✧ 程序只由有限个函数构成，其中函数的定义如 `func a(para1: Int, para2: Int)->Void` 形式，其中 `Void` 为返回值，`func` 为函数定义时的关键字。
- ✧ 数据类型在本语言中有：`Void`、`Int`、`Double`、`String`、`Int8`、`Int16`、`Int32`、`Int64`。均作为关键字存储在头文件的 `keywordsList` 中。
- ✧ 函数定义中参数的书写格式是 `para: Int`, 其中冒号为限定符，在冒号之后为变量的类型，即数据类型。
- ✧ 基本语句包括定义语句、循环语句、函数调用语句、分支语句、函数返回语句、基本块。
- ✧ 分支语句：形式可以是 `if(a<b){}` 或者是 `if a<b {}`，即分支判断的条件可以不在括号内
- ✧ 定义语句：必须以 `let` 或者 `var` 开头，其中在本语言（简称 X）中 `let` 表示 `const` 常量，`var` 表示可变变量类型。定义的语句必须如 `let a: Int32=100;` 形式。
- ✧ 循环语句：形式可以是 `while a<b` 跟着单条语句或者 `while(a<b){}`，其中大括号内可以跟着多条语句。
- ✧ 基本块：使用大括号包围起来的语句集合。

3. 2 数据结构

3. 2. 1 符号数据结构 Symbol

```
struct symbol{  
    string str;  
    int type;  
    string tokenType;  
    int can_generate_epsilon = -1;  
    set<string> first;  
    symbol(string str,int type):str(str),type(type){}  
    symbol(string str,int type,string tokenType):  
        str(str),type(type),tokenType(tokenType){}  
    symbol(){}
```

```

bool operator < (const symbol other) const{
    return this->str < other.str;
}
};

```

对该数据结构的解释如下：

- str: string 类型，表示该符号的字符串值。
 - type: int 类型，表示该符号的类型，有四种：非终结符、终结符、epsilon、前向搜索边界符号#。将四种类型都转化成 int 值存放在枚举变量 TYPE 中，枚举变量的声明如下：
- ```
enum TYPE{terminal=0,nonterminal=1,epsilon=2,front_search=3};
```
- tokenType: string 类型，表示该符号对应于 token 中的类型，有五种，即 token 中的五种，用于扩展使用。
  - can\_generate\_epsilon: int 类型，表示该符号能否推出 epsilon。能则为 1，不能则为 0，未确定则为-1。
  - first: set<string>类型，表示该符号的 FIRST 集合。
  - 重定义<运算符： 用于将 symbol 实例放入 set 中进行去重，为此必须设定 symbol 的大小关系比较方法。

### 3. 2. 2 产生式数据结构 production

```

class production{
public:
 int index;
 symbol left;
 vector<symbol> right = vector<symbol>();
};

public:
 void print();
};

```

对产生式数据结构的解释如下：

- index: int 类型，表示产生式的编号，从产生式文件中每读取一行设置一个产生式编号，从 0 开始。

- left: symbol 类型，表示产生式左侧的符号（由于是 2 型文法，产生式左侧只能有一个符号）。
- right: vector<symbol> 类型，表示产生式右侧的符号序列。
- print(): 产生式输出函数。

### 3. 2. 3 产生式集成数据结构 allProduction

```
class allProduction{
public:
 vector<production> productions;
 vector<symbol> terminal;
 vector<symbol> nonterminal;
 map<string,int> indexTable;
public:
 void initial_production_set();
 bool check_is_end();
 void markEpsilonProduction();
 void getFirstSet();
 void print_symbol_first();
 set<string> First(vector<symbol> symList);
 void printAllProduction();
};

```

#### 变量解释：

- productions : vector<production>类型，表示产生式数组。
- terminal: vector<symbol> 终结符数组。
- nonterminal: vector<symbol> 非终结符数组。
- indexTable: map<string,int>数据类型，表示每个非终结符在 nonterminal 数组中的索引值，用于构建映射表以随机获取任何非终结符的 symbol 数据，从而得到其能否推出空等数据。

#### 函数解释：

- ✧ initial\_production\_set(): 从产生式规则文件中读取产生式集合的函数。
- ✧ check\_is\_end(): 辅助函数，检查求每个非终结符能否推出 epsilon 过程是否

能够终止，也就是所有非终结符号能否推出 epsilon 是否全都不是未定(-1)。

- ✧ markEpsilonProduction(): 使用相应算法判断所有文法符号能否推导出 epsilon。
- ✧ getFirstSet(): 获取所有文法符号的 FIRST 集合。
- ✧ print\_symbol\_first(): 打印产生式序列对应的属性：所有文法符号的 FIRST 集合。
- ✧ First(vector<symbol> symList): 用于应用相关算法求任何符号序列的 FIRST 集合，传递的参数是符号列表。
- ✧ printAllProduction(): 输出函数，按照一定格式输出所有产生式。

### 3.2.4 项目数据结构 Item

```
class Item : public production{
public:
 int pointer = 0;
 set<string> frontSet = set<string>();
 bool canReduction = false;
 bool canShiftIn = false;

public:
 Item(production pro);
 static Item moveOneStep(Item item);
 bool checkCanReduction();
 bool checkCanShiftIn();
 bool isEpsilonItem();
 void printItem();
 static bool isEqual(Item item1, Item item2, bool in);
};
```

Item 数据结构扩展自 production，其中添加了一些关于 Item 的处理函数以及变量。解释如下：

#### 变量解释：

- ✓ pointer: int 数据类型，表示项目中项目点的位置，即当前项目已经被识别部分的长度。

- ✓ frontSet: set<string>类型，表示当前项目的前向搜索符号集合，使用 set 进行去重。
- ✓ canReduction: bool 类型，表示 Item 是否是规约项目。
- ✓ canShiftIn: bool 类型，表示 Item 是否是移进项目。

**函数解释：**

- ✓ Item(production pro): 构造函数，从产生式数据结构产生一个默认 Item。
- ✓ moveOneStep: 当前 Item 向右多识别一个符号。
- ✓ checkCanReduction(): 检查当前项目能否规约。
- ✓ checkCanShiftIn(): 检查当前项目能否移进。
- ✓ isEpsilonItem(): 判断当前项目是否是直接推出空的项目，用于在 LR(1) 项目集规范族构造算法中进行特判。
- ✓ printItem(): 按照一定格式打印 Item。
- ✓ static bool isEqual(Item item1, Item item2, bool in) : 判断两个 Item 是否相同，in=false 则不比较前向搜索符号集合。in=true 则比较前向搜索符号集合。

### 3.2.5 项目集数据结构 ItemSet

```
class ItemSet{
public:
 int state = -1;
 vector<Item> itemList = vector<Item>();
public:
 void printItemSet();
 bool checkCanInsert(Item item);
 bool operator == (const ItemSet &other) const;
};
```

对项目集数据结构的解释如下；

- state:int 类型，表示当前项目集在识别活前缀的过程中，在状态转移图中的编号。
- itemList: 项目集数组，表示一个 DFA 状态中有多个项目构成。
- printItemSet(): 打印项目集。
- checkCanInsert(Item item) : 检查在当前项目集中是否能够插入一条 Item。

- 重定义判等运算符：用于判断两个项目集是否相等。

### 3.2.6 ACTION 和 GOTO 元素数据结构 Element

```
class Element{
public:
 int type = -1;
 int index = -1;
public:
 Element();
 Element(int type,int index):type(type),index(index){}
 string transElement();
};
```

对 **Element** 数据结构的解释如下：

- ✧ type: int 数据类型，表示 ACTION 或者 GOTO 表中该项的类型，由如下枚举类型指定：

```
enum TYPE_OF_ACTION_GOTO{
 PRODUCTION=0,//表项中是产生式
 STATE=1,//表项中是状态编号
 ACCEPT=2,//表项中是接受状态
 ILLEGAL=3//表项为非法
};
```

- ✧ index: 若表项是产生式，则表示产生式的编号，若表项是状态，则表示状态编号。
- ✧ transElement(): 按照一定数据格式打印 Element。

### 3.2.7 可归前缀 DFA 构造数据结构 Processor

```
class Processor{
public:
 allProduction pros;
 vector<ItemSet> DFA;
 map<int, map<string,Element>> Action = map<int, map<string,Element>>();
 map<int, map<string,Element>> Goto;
```

```

map<int, map<string,int> > transferTable;
int errorPos = -1;

public:
 void generateDFA(allProduction prodctions);
 void generateActionAndGoto();
 vector<int> findAccpetState();
 ItemSet stateClosure(ItemSet kernel);
 static ItemSet merge(ItemSet item);
 static bool checkCanTrans(Item item);
 int isExist(ItemSet itemSet);
 void printTransTable();
 void printActionGoto();
 bool isIllegalList(vector<symbol> symList,vector<Token> tokenList);
 string errorMsg(Token token);
};


```

**对 Processor 数据结构的变量解释为:**

- ✓ pros : allProduction 数据类型，包括产生式集合以及一组产生式属性描述变量和处理函数。
- ✓ DFA: vector<ItemSet>数据类型，表示识别活前缀的自动机。
- ✓ Action: map<int, map<string,Element> >数据类型，表示 ACTION 表。
- ✓ Goto: map<int, map<string,Element> >数据类型，表示 GOTO 表。
- ✓ transferTable: map<int, map<string,Element> >数据类型，表示项目集规范族的转移图，用于构建 ACTION 和 GOTO 表。
- ✓ errorPos: 出现 LR(1)分析错误时错误的符号在输入符号序列中的位置，用于构造出错信息。

**对 Processor 数据结构的函数解释为:**

- generateDFA(allProduction prodctions): 从产生式集合中构建识别活前缀的 DFA。
- generateActionAndGoto():产生 ACTION 和 GOTO 表。
- findAccpetState():查找 DFA 的终态。
- ItemSet stateClosure(ItemSet kernel): 传入项目集核，并通过该核构建项目闭

包。

- static ItemSet merge(ItemSet item): 项目集闭包计算完毕之后每个 item 的前向搜索符号集合只有一个符号, 将所有 item 除前向搜索符号集合之外相同的部分进行合并, 用于显示输出。
- static bool checkCanTrans(Item item): 检查产生式是否通过弧进行转移。
- int isExist(ItemSet itemSet): 判断一个项目集在 DFA 中是否存在, 存在则返回项目集在 DFA 中的编号, 不存在则返回-1。
- printTransTable(): 打印 DFA 转移图, 即转移函数。
- printActionGoto(): 打印 ACTION 和 GOTO 表。
- bool isIllegalList(vector<symbol> symList, vector<Token> tokenList): 判断符号序列是否是该 LR(1) 文法的符号序列。
- string errorMsg(Token token): 根据出错时正在扫描的 token, 构造出错信息。

### 3.3 语法分析中的算法与程序解释

#### 3.3.1 整体程序流程

语法分析的程序执行流程如图 7 所示:

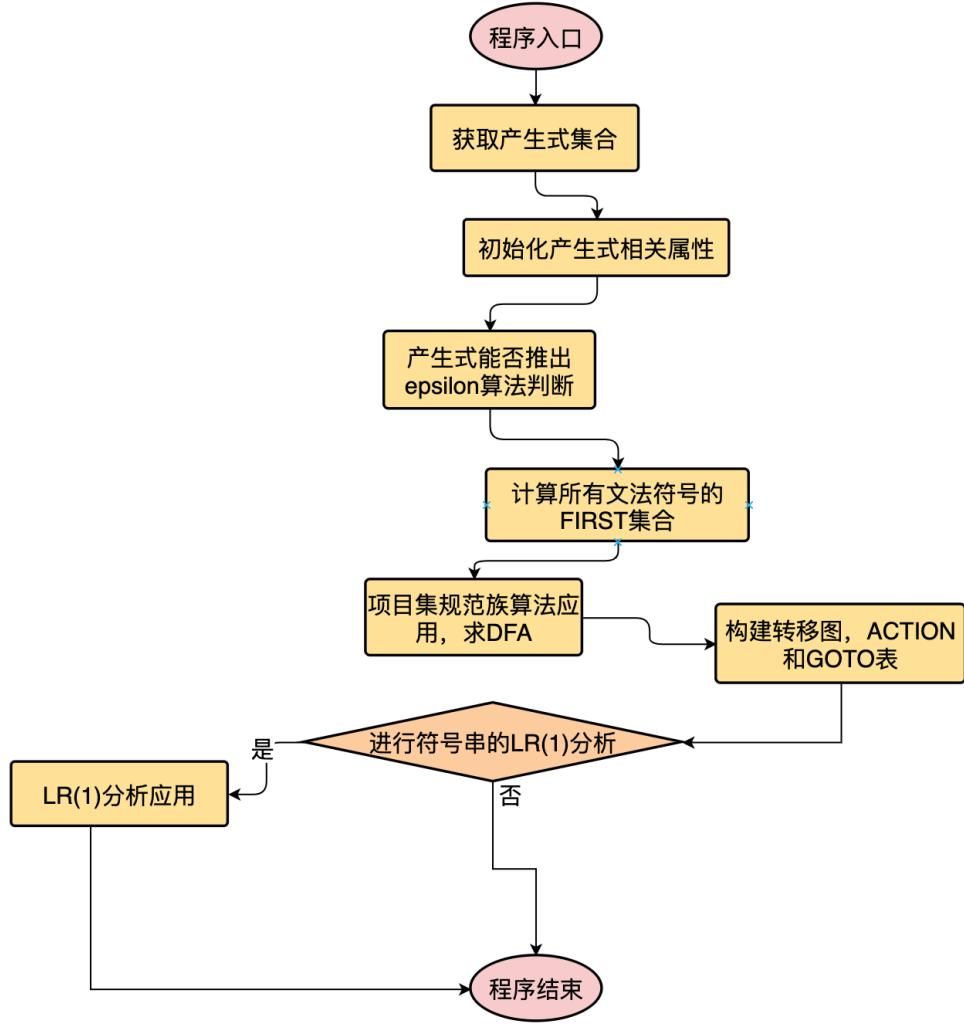


图 7. 语法分析程序整体流程

### 3.3.2 标记非终结符能否推出 epsilon 算法

标记非终结符能够推出 epsilon 的算法一共有如下四个流程：

- (1) 将 X[] 中对应于每一个非终结符的标志设置初值未定。
- (2) 扫描文法中的产生式。
  - a) 删除所有右部含有终结符的产生式，若这使得以某一个非终结符为左部的所有产生式都被删除，则将数组中对应于该非终结符的标记值改为“否”，说明该非终结符不能推出 epsilon。
  - b) 若某一非终结符的某一个产生式右部为 epsilon，则将数组中对该非终结符的标志置为“是”，并从文法中删除该非终结符的所有产生式。
- (3) 扫描产生式右部的每一个符号。

- a) 若所扫描到的非终结符在数组中对应的标志是“是”，则删去该非终结符；若这使产生式右部为空，则将产生式左部的非终结符在数组中对应标志改为“是”，并删除以该终结符为左部的所有产生式。
  - b) 若所扫描到的非终结符在数组中对应的表示是“否”，则删去该非终结符；若这使得产生式左部非终结符的有关产生式都被删除，则把数组中该非终结符对应的标志改为“否”。
- (4) 重复(3)，直到扫描完一遍文法的产生式，数组中非终结符对应的特征没有再改变为之。

### 3.3.3 求所有文法符号串的 FIRST 集算法

首先，计算所有文法符号 X（即非终结符以及终结符）的 FIRST 集合。

- (1) 如果 X 是终结符，则  $\text{FIRST}(X) = \{X\}$ ；
- (2) 如果 X 是非终结符，且有产生式  $X \rightarrow a \dots$ , a 为终结符，则 a 放入  $\text{FIRST}(X)$  中；
- (3) 如果 X 是非终结符，且  $X \rightarrow \epsilon$ ，则  $\epsilon$  放入  $\text{FIRST}(X)$  中。
- (4) 若  $X, Y_1, Y_2 \dots$  都是文法的符号，而有产生式  $X \rightarrow Y_1 Y_2 \dots Y_n$ ，当  $Y_i$  不能推导出  $\epsilon$  时，将  $\text{FIRST}(Y_1) - \{\epsilon\}, \text{FIRST}(Y_2) - \{\epsilon\} \dots$  以及  $\text{FIRST}(Y_i)$  放入  $\text{FIRST}(X)$  中。
- (5) 当(4)中所有  $i$  都有  $Y_i$  经过有限步推导出  $\epsilon$ ，则  $\text{FIRST}(Y_1), \text{FIRST}(Y_2) \dots \text{FIRST}(Y_n)$  与  $\{\epsilon\}$  都放入  $\text{FIRST}(X)$  中。
- (6) 反复运用算法(2)-(5)直到所有符号的 FIRST 集合都收敛。

求出每个文法符号的 FIRST 集合之后，可以求出任意符号串的 FIRST 集合。若符号串 a 都由文法符号构成，且  $a = X_1 X_2 \dots X_n$ ；当某个  $X_i$  不能推导  $\epsilon$  时， $\text{FIRST}(a)$  中应该包括  $\text{FIRST}(X_j) - \{\epsilon\}$  ( $j$  从 1 到  $i-1$ )，并且包括  $\text{FIRST}(X_i)$ 。

若对于符号串 a，每一个  $X_i$  都能推导出  $\epsilon$ ，则  $\text{FIRST}(a)$  中应当包括所有符号  $X_i$  的 FIRST 集合。

### 3.3.4 项目集规范族构造算法与 ACTION & GOTO 表构造算法

项目集规范族构造算法如下：

以初始符号  $S' \rightarrow S$ , # 属于初始项目集中, 把#作为前向搜索符号, 表示活前缀为 a (若 a 是有关 S 产生式的某一右部) 要归约成 S 是必须面临输入符号#才行。对初始项目  $S' \rightarrow S$  计算闭包之后, 再利用转换函数逐步求解整个文法的 LR(1) 项目集族。具体构造步骤如下:

### (1) 构造 LR(1) 项目集的闭包函数。

1. 假定 I 是项目集, I 的任何项目都属于 CLOSURE(I);
2. 若有项目  $A \rightarrow a \cdot B b, t$  属于 CLOSURE(I),  $B \rightarrow s$  是文法中的一个产生式, 且 b 是文法符号的序列, 则当 m 为 FIRST(bt) 中的元素时,  $B \rightarrow \cdot s, m$  也属于 CLOSURE(I)。
3. 重复 2, 直到 CLOSURE(I) 不再增大为止。

### (2) 构造转换函数。

LR(1) 转换函数的构造为:

$$G_0(I, X) = CLOSURE(J);$$

其中 I 是 LR(1) 的项目族, X 是文法符号,

$$J = \{ \text{任何形如 } [A \rightarrow aX \cdot b, m] \text{ 的项目} \mid [A \rightarrow aX \cdot b, m] \text{ 属于 } I \}$$

对文法  $G'$  的 LR(1) 项目集族的构造仍以  $[S' \rightarrow \cdot S, \#]$  为初态集的初始项目, 然后对其求闭包和转换函数, 直到项目集不再增大为止。

也就是对状态 I 经过符号 X 后转向状态 J, 求出 J 的核后, 对核求闭包即为 CLOSURE(J)。

### ACTION 和 GOTO 表的构造

若已构造出某文法的 LR(1) 项目集族 C:  $C = \{I_0, I_1, I_2, \dots, I_n\}$ ;

其中  $I_k$  的 k 为分析器的状态, 则 ACTION 和 GOTO 表的构造方法如下所示:

- (1) 若项目  $[A \rightarrow a \cdot bc, d]$  属于  $I_k$ , 且  $G_0(I_k, b) = I_j$  ( $G_0$  为转换函数), 其中 b 是终结符, 则置  $\text{ACTION}[k, b] = S_j$ ,  $S_j$  的含义是把输入符号和状态 j 分别移入文法符号栈和状态栈。
- (2) 若项目  $[A \rightarrow a \cdot, d]$  属于  $I_k$ , 且  $G_0(I_k, d) = r_j$ , 其中 d 是终结符,  $r_j$  的含义是把当前栈顶符号串 a 规约成 A, j 在文法中位产生式  $A \rightarrow a$  的编号。
- (3) 若项目  $[S' \rightarrow S \cdot, \#]$  属于  $I_k$ , 则置  $\text{ACTION}[k, \#] = \text{acc}$ , 表示“接受”。
- (4) 若  $G_0(I_k, A) = I_j$ , 其中 A 是非终结符, 则置  $\text{GOTO}[k, A] = j$ , 表示转入 j 状态, 置当前文法符号栈顶为 A, 状态栈顶为 j。

(5) 凡是不能用规则(1)-(4)填入分析表中的元素，均置报错标志。

### 3.3.5 LR(1)符号串分析算法与实现方案

1. 设置状态栈：StateStack
2. 设置符号栈：SymbolStack
3. 对于输入符号串，从左至右线性扫描，当前栈顶状态 S，查看下一个输入符号 a，动作为 ACTION[S, a]。若 ACTION[S, a] 为状态 Sj，则将状态 Sj 压入状态栈中，将 a 压入符号栈中，若 ACTION[S, a] 为产生式 rj，则查看产生式 rj 右侧符号数量，令为 t，将符号栈和状态栈弹出 t 个元素，若 rj 产生式规约之后为 A，则查看 ACTION[S1, A]，其中 S1 为弹出 t 个状态之后状态栈的栈顶，则下一步动作为 ACTION[S1, A] 指定动作。
4. 通过算法步骤 3 进行迭代，直到遇到 acc 或者遇到非法的 ACTION。

## 3.4 案例分析

### 3.4.1 求 FIRST 算法书上案例

#### (1) 案例 1

产生式(其中 epsilon 使用@符号代替)：

```
"S" -> "A" "B"
"S" -> "b" "C"
"A" -> '@'
"A" -> 'b'
"B" -> '@'
"B" -> 'a" "D"
"C" -> "A" "D"
"C" -> 'b'
"D" -> 'a" "S"
"D" -> 'c'
$
```

FIRST 集合求解结果：

$$\text{FIRST}(A) = \{ @, b \}$$

$\text{FIRST}(B) = \{ @, a \}$   
 $\text{FIRST}(C) = \{ a, b, c \}$   
 $\text{FIRST}(D) = \{ a, c \}$   
 $\text{FIRST}(S) = \{ @, a, b \}$   
 $\text{FIRST}(a) = \{ a \}$   
 $\text{FIRST}(b) = \{ b \}$   
 $\text{FIRST}(c) = \{ c \}$

## (2) 案例 2

产生式(其中 epsilon 使用@符号代替):

```

"E" -> "E"
"E" -> "E" + "T"
"E" -> "T"
"T" -> "T" * "F"
"T" -> "F"
"F" -> 'i'
"F" -> '("E")'
$
```

FIRST 集合求解结果:

$\text{FIRST}(E) = \{ (, i \}$   
 $\text{FIRST}(E') = \{ (, i \}$   
 $\text{FIRST}(F) = \{ (, i \}$   
 $\text{FIRST}(T) = \{ (, i \}$   
 $\text{FIRST}(\ ) = \{ ( \}$   
 $\text{FIRST}(\ )) = \{ ) \}$   
 $\text{FIRST}(*) = \{ * \}$   
 $\text{FIRST}(+) = \{ + \}$   
 $\text{FIRST}(i) = \{ i \}$

## (3) 案例 3

```

"E" -> "T" "E1"
"E1" -> '+' "T" "E1"
"E1" -> '@'
"T" -> "F" "T1"
"T1" -> '*' "F" "T1"
```

```

"T1"->'@'
"F"->'i'
"F"->('E')
$
```

FIRST 集合求解结果:

```

FIRST(E)= { (, i }
FIRST(E1)= { + , @ }
FIRST(F)= { (, i }
FIRST(T)= { (, i }
FIRST(T1)= { * , @ }
FIRST(())= { ()
FIRST()))= {) }
FIRST(*)= { * }
FIRST(+)= { + }
FIRST(i)= { i }
```

根据上述文法以及结果可以发现，通过本语法分析计算的各种类型文法的 FIRST 集合都正确。

### 3.4.2 求 LR(1) 项目集规范族算法案例

#### (1) 案例 1

```

"S'"->"S"
"S"->"B""B"
"B"->a"B"
"B"->'b'
$
```

对于上述产生式规则，构建的 LR(1) 项目集规范族如下所示：

状态： 0

```

S' -> ·S , { # }
S -> ·BB , { # }
B -> ·aB , { a,b }
B -> ·b , { a,b }
```

状态： 1

$S' \rightarrow S^\cdot, \{ \# \}$

状态: 2

$S \rightarrow B^\cdot B, \{ \# \}$

$B \rightarrow \cdot aB, \{ \# \}$

$B \rightarrow \cdot b, \{ \# \}$

状态: 3

$B \rightarrow a \cdot B, \{ a, b \}$

$B \rightarrow \cdot aB, \{ a, b \}$

$B \rightarrow \cdot b, \{ a, b \}$

状态: 4

$B \rightarrow b \cdot, \{ a, b \}$

状态: 5

$S \rightarrow BB^\cdot, \{ \# \}$

状态: 6

$B \rightarrow a \cdot B, \{ \# \}$

$B \rightarrow \cdot aB, \{ \# \}$

$B \rightarrow \cdot b, \{ \# \}$

状态: 7

$B \rightarrow b \cdot, \{ \# \}$

状态: 8

$B \rightarrow aB \cdot, \{ a, b \}$

状态: 9

$B \rightarrow aB \cdot, \{ \# \}$

上述结果与书上结果完全一致，同时，在 test-rule5.txt 以及 test-rule6.txt 文件中同样有书上的两个案例，根据结果可以发现，与书上的案例结果完全相同。

### (3) 案例 2

有如下的测试程序片段：

```
func mine()->Void{}
```

LR(1)预测分析结果如下所示：

```

func mine () -> Void { } 的 LR(1)识别过程
步驟 狀態棧 符號棧 輸入字符串 ACTION GOTO
1 : { 0 } # func mine () -> Void { } # S3(action)
2 : { 0 , 3 } # func mine () -> Void { } # S5(action)
3 : { 0 , 3 , 5 } # func [identifier] () -> Void { } # S6(action)
4 : { 0 , 3 , 5 , 6 } # func [identifier] () -> Void { } # r13(action) -l(goto)
5 : { 0 , 3 , 5 , 6 , 7 } # func [identifier] (PARAMETER_LIST) -> Void { } # S10(action)
6 : { 0 , 3 , 5 , 6 , 7 , 10 } # func [identifier] (PARAMETER_LIST) -> Void { } # S14(action)
7 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 } # func [identifier] (PARAMETER_LIST) -> Void { } # S25(action)
8 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 } # func [identifier] (PARAMETER_LIST) -> Void { } # S28(action)
9 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 , 28 } # func [identifier] (PARAMETER_LIST) -> Void { } # r3(action) 27(goto)
10 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 , 27 } # func [identifier] (PARAMETER_LIST) -> DATATYPE { } # S36(action)
11 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 , 27 , 36 } # func [identifier] (PARAMETER_LIST) -> DATATYPE { } # r17(action) -l(goto)
12 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 , 27 , 36 , 37 } # func [identifier] (PARAMETER_LIST) -> DATATYPE { SENTENCES } # S54(action)
13 : { 0 , 3 , 5 , 6 , 7 , 10 , 14 , 25 , 27 , 36 , 37 , 54 } # func [identifier] (PARAMETER_LIST) -> DATATYPE { SENTENCES } # r11(action) 2(goto)
14 : { 0 , 2 } # FUNCTION # r1(action) 1(goto)
15 : { 0 , 2 , 4 } # FUNCTION X # r2(action) 1(goto)
16 : { 0 , 1 } # X # acc(action)

成功识别

```

图 8. 程序片段 LR(1) 分析结果

## 四 语义分析

本语义分析程序仅仅对于表达式属性文法计算结果并给出四元式序列。产生式的规则如 4.1 所示。本语义动作的指定规则：如果是二元运算，运算符只能是 +,\* 或者是 -。可以改变二元运算中符号的名称、运算数的前后顺序等。

语义分析程序基于问题二 LR(1) 分析进行开发，在规约过程中加上了产生式的语义动作，从而在语义动作的过程中构造四元式序列。

### 4.1 产生式规则与解释

对于语义分析，本程序中使用的产生式规则如下所示（参考书中的表达式计算 S 属性文法）。

```

"E'->"E" {print("E".val)}
"E"->"E""+"T" {"E".val:="E".val+"T".val}
"E"->"E""*""T" {"E".val:="E".val*T".val}
"E"->"E""-""T" {"E".val:="E".val-T".val}
"E"->"T" {"E".val:="T".val}
"T"->"T""*""F" {"T".val:="T".val*F".val}
"T"->"F" {"T".val:="F".val}
"F"->'("E")' {"F".val:="E".val}
"F"->[scientific-format]' {"F".val:='d'.lexval}
$

```

从中可见本产生式规则可以实现整数的表达式计算，可以实现不超过 INT\_MAX 范围的整数之间的加法、减法、乘法运算。其中.lexval 表示变量字面值。.val 表示符号的值，print 语义为打印值。

对于综合属性文法的解释为：对于产生式  $A \rightarrow a$  的产生式，其语义动作为  $a = f(b_1, b_2, b_3, \dots, b_n)$  且  $a$  是  $A$  的一个属性，则  $a$  是  $A$  的综合属性，只含有综合属性的文法为 S-属性文法。

## 4. 2 数据结构

### 4. 2. 1 扩展产生式数据结构 production

```
class production{
public:
 int index;
 symbol left;
 vector<symbol> right = vector<symbol>();
 string semanticAction;
 int actionType = -1;
public:
 void print();
};
```

上述为扩展问题二中的产生式数据结构 `production`，在其中添加两个新的变量 `semanticAction` 以及 `actionType`。其中 `semanticAction` 表示语义动作的字符串，即没条产生式括号中的语义动作。此外，`actionType` 表示 `action` 的类型，在本表达式属性文法中 `action` 有如下几种类型，使用枚举变量进行表示，数据结构如下所示。

```
enum SEMANTIC_ACTION_TYPE{
 constant_assign=0,//常量初始化类型的赋值 比如 t1 := 10
 parameter_assign=1,//变量之间的赋值 比如 t2 := t1
 computing_assign_l_r=2,//计算形式的赋值，二元运算 比如 t3 := t1 * t2，并且左操作数在左
 computing_assign_r_l=3,//计算形式的赋值，二元运算 比如 t3 := t2 * t1，并且右操作数在左
 print=4//输出语义值，也就是表达式的计算结果
};
```

在该枚举类型数据结构中包括了如下四种类型的语义：

- ✓ 常量初始化类型的赋值，即将 d 字面值赋值给 d，如 t1:=10；
- ✓ 变量之间的赋值，如 t2:=t1 形式的赋值。
- ✓ 二元运算赋值，产生是右侧左边的操作符号在运算左侧，如 T 在左且 F 在右，则运算计算 T.val operator F.val。
- ✓ 二元运算赋值，产生是右侧左边的操作符号在运算符号右侧，如 T 在左且 F 在右，则运算计算 F.val operator T.val。

## 4.3 表达式语义分析中算法

### 4.3.1 属性文法读入算法

本程序中属性文法的读入算法如下所示：

1. 读入一行产生式。分割，得出产生式的左部以及右部。
2. 分割语义动作，确定语义动作的类型。将其归结为上述数据结构中四种类型中的一类。

### 4.3.2 LR(1) 分析算法以及四元式构造算法

数据结构定义：

变量 current\_index：当前四元式中变量的数量。

状态栈：stateStack（用于 LR(1) 分析中的状态）

符号栈：symbolStack（用于 LR(1) 分析中保存已经规约的符号）

语义栈：semanticStack（用于保存当前符号栈中每一个符号的语义）

额外栈：extraStack（用于保存当前符号栈中每一个符号对应于四元式中的变量）

注：其中栈的实现使用 C++vector，由于需要打印栈中的内容，需要随机访问栈中的元素，故采用 vector 数据结构。

LR(1) 分析算法的执行流程如下所示：

1. 初始化状态栈，其中有唯一的状态 0，初始化符号栈，其中有唯一的程序边界符号#，初始化语义栈，其中有内容-1，输出时显示\_表示该符号没有语义。  
初始化符号栈，内容为\_。

2. 按照问题二中 LR(1) 分析方法进行预测分析，每个步骤中检查是否是利用产生式进行规约，检查规约的语义动作，查看属于哪一类，分析方式如下：
- 对于产生式的类型为 `constant_assign` 的，将当前语义值放入栈中
  - 对于产生式类型是 `parameter_assign` 的，计算应当进行赋值的新变量的变量名称，额外栈中栈顶元素即为被赋值变量。
  - 对于产生式类型为 `computing_assign_l_r` 或者 `computing_assign_r_l` 的，从额外栈中获得被赋值的两个变量的变量名称，计算新变量的变量名称，构造赋值等式。
  - 上述三个步骤均在当前更新语义栈和额外栈。并将新变量压入额外栈中，将计算结果压入语义栈中。

3. 计算完毕之后，语义栈中的内容应为表达式的(假设表达式的计算中间结果以及最终结果不超过 INT\_MAX 的范围)。

通过该产生式集合构建的 LR (1) ACTION 和 GOTO 表如下所示，其中-1 表示不合法状态。

ACTION+GOTO 表：

|    | ( ) | *   | +   | -   | [scientific-format] | #   | E   | F  | T  |    |
|----|-----|-----|-----|-----|---------------------|-----|-----|----|----|----|
| 0  | S4  | -1  | -1  | -1  | -1                  | S5  | -1  | 1  | 3  | 2  |
| 1  | -1  | -1  | S7  | S6  | S8                  | -1  | acc | -1 | -1 | -1 |
| 2  | -1  | -1  | S9  | r4  | r4                  | -1  | r4  | -1 | -1 | -1 |
| 3  | -1  | -1  | r6  | r6  | r6                  | -1  | r6  | -1 | -1 | -1 |
| 4  | S13 | -1  | -1  | -1  | -1                  | S14 | -1  | 10 | 12 | 11 |
| 5  | -1  | -1  | r8  | r8  | r8                  | -1  | r8  | -1 | -1 | -1 |
| 6  | S4  | -1  | -1  | -1  | -1                  | S5  | -1  | -1 | 3  | 15 |
| 7  | S4  | -1  | -1  | -1  | -1                  | S5  | -1  | -1 | 3  | 16 |
| 8  | S4  | -1  | -1  | -1  | -1                  | S5  | -1  | -1 | 3  | 17 |
| 9  | S4  | -1  | -1  | -1  | -1                  | S5  | -1  | -1 | 18 | -1 |
| 10 | -1  | S19 | S21 | S20 | S22                 | -1  | -1  | -1 | -1 | -1 |
| 11 | -1  | r4  | S23 | r4  | r4                  | -1  | -1  | -1 | -1 | -1 |
| 12 | -1  | r6  | r6  | r6  | r6                  | -1  | -1  | -1 | -1 | -1 |
| 13 | S13 | -1  | -1  | -1  | -1                  | S14 | -1  | 24 | 12 | 11 |
| 14 | -1  | r8  | r8  | r8  | r8                  | -1  | -1  | -1 | -1 | -1 |
| 15 | -1  | -1  | S9  | r1  | r1                  | -1  | r1  | -1 | -1 | -1 |

|    |     |     |     |     |     |     |    |    |    |    |
|----|-----|-----|-----|-----|-----|-----|----|----|----|----|
| 16 | -1  | -1  | S9  | r2  | r2  | -1  | r2 | -1 | -1 | -1 |
| 17 | -1  | -1  | S9  | r3  | r3  | -1  | r3 | -1 | -1 | -1 |
| 18 | -1  | -1  | r5  | r5  | r5  | -1  | r5 | -1 | -1 | -1 |
| 19 | -1  | -1  | r7  | r7  | r7  | -1  | r7 | -1 | -1 | -1 |
| 20 | S13 | -1  | -1  | -1  | -1  | S14 | -1 | -1 | 12 | 25 |
| 21 | S13 | -1  | -1  | -1  | -1  | S14 | -1 | -1 | 12 | 26 |
| 22 | S13 | -1  | -1  | -1  | -1  | S14 | -1 | -1 | 12 | 27 |
| 23 | S13 | -1  | -1  | -1  | -1  | S14 | -1 | -1 | 28 | -1 |
| 24 | -1  | S29 | S21 | S20 | S22 | -1  | -1 | -1 | -1 | -1 |
| 25 | -1  | r1  | S23 | r1  | r1  | -1  | -1 | -1 | -1 | -1 |
| 26 | -1  | r2  | S23 | r2  | r2  | -1  | -1 | -1 | -1 | -1 |
| 27 | -1  | r3  | S23 | r3  | r3  | -1  | -1 | -1 | -1 | -1 |
| 28 | -1  | r5  | r5  | r5  | r5  | -1  | -1 | -1 | -1 | -1 |
| 29 | -1  | r7  | r7  | r7  | r7  | -1  | -1 | -1 | -1 | -1 |

### (1) 案例 1

通过如下表达式进行测试:

**(432+489-4\*2+(3+12-2))**

得到的 TOKEN 表为 (按照从左到右, 从上到下形式排列):

|                                                                        |
|------------------------------------------------------------------------|
| ( 1 , 限定符 , '(' )( 1 , 常量 , '432' )( 1 , 运算符 , '+' )( 1 , 常量 , '489' ) |
| ( 1 , 运算符 , '-' )( 1 , 常量 , '4' )( 1 , 运算符 , '*' )( 1 , 常量 , '2' )     |
| ( 1 , 运算符 , '+' )( 1 , 限定符 , '(' )( 1 , 常量 , '3' )( 1 , 运算符 , '+' )    |
| ( 1 , 常量 , '12' )( 1 , 运算符 , '-' )( 1 , 常量 , '2' )( 1 , 限定符 , ')' )    |
| ( 1 , 限定符 , ')' )                                                      |

得到的四元式序列的第一种表达方式为:

```
t1:=432
t2:=t1
t3:=t2
t4:=489
t5:=t4
t7:=4
t8:=t7
```

t9:=2  
t12:=3  
t13:=t12  
t14:=t13  
t15:=12  
t16:=t15  
t18:=2  
t19:=t18  
t21:=t20  
t22:=t21  
t24:=t23  
t25:=t24  
t26:=t25

四元式表达:

( := ,432 ,\_ , t1 )  
( := ,t1 ,\_ , t2 )  
( := ,t2 ,\_ , t3 )  
( := ,489 ,\_ , t4 )  
( := ,t4 ,\_ , t5 )  
( + , t3 , t5 , t6 )  
( := ,4 ,\_ , t7 )  
( := ,t7 ,\_ , t8 )  
( := ,2 ,\_ , t9 )  
( \* , t8 , t9 , t10 )  
( - , t6 , t10 , t11 )  
( := ,3 ,\_ , t12 )  
( := ,t12 ,\_ , t13 )  
( := ,t13 ,\_ , t14 )  
( := ,12 ,\_ , t15 )  
( := ,t15 ,\_ , t16 )  
( + , t14 , t16 , t17 )  
( := ,2 ,\_ , t18 )  
( := ,t18 ,\_ , t19 )

```

(- , t17 , t19 , t20)
(:= , t20 , _ , t21)
(:= , t21 , _ , t22)
(+ , t11 , t22 , t23)
(:= , t23 , _ , t24)
(:= , t24 , _ , t25)
(:= , t25 , _ , t26)

```

其中包含三种类型的四元式:

- ✓  $( := , 432 , _ , t1 )$  表示将字面值 432 赋值给变量  $t1$ ;
- ✓  $( := , t1 , _ , t2 )$  表示将变量  $t1$  赋值给  $t2$ ;
- ✓  $( op, arg1, arg2, result )$  表示计算  $arg1 \ op \ arg2$ , 并将结果赋值给  $result$  变量;

经过上述计算以及 LR(1)分析结果, 最终语义栈中的结果为 926, 计算结果与实际相符。

## (2) 案例 2

通过计算表达式 **1+2** 观察 LR(1)分析结果:

| 1 + 2 的 LR(1) 识别过程     |                              |                 |             |            |                                |           |
|------------------------|------------------------------|-----------------|-------------|------------|--------------------------------|-----------|
| 步骤                     | 状态栈                          | { 符号栈 语义栈 额外栈 } | 输入字符串       | ACTION     | GOTO                           | 语义动作      |
| 1 : { 0 }              | {# _ }                       | 1 + 2 #         |             | S5(action) |                                |           |
| 2 : { 0 , 5 }          | {# _ }{1 1 _}                | + 2 #           | r8(action)  | 3(goto)    | "F".val:='d'.lexval(语义动作)      | t1:=1     |
| 3 : { 0 , 3 }          | {# _ }{F 1 t1}               | + 2 #           | r6(action)  | 2(goto)    | "T".val:="F".val(语义动作)         | t2:=t1    |
| 4 : { 0 , 2 }          | {# _ }{T 1 t2}               | + 2 #           | r4(action)  | 1(goto)    | "E".val:="T".val(语义动作)         | t3:=t2    |
| 5 : { 0 , 1 }          | {# _ }{E 1 t3}               | + 2 #           | S6(action)  |            |                                |           |
| 6 : { 0 , 1 , 6 }      | {# _ }{E 1 t3}{+ _ }         | 2 #             | S5(action)  |            |                                |           |
| 7 : { 0 , 1 , 6 , 5 }  | {# _ }{E 1 t3}{+ _ }{2 2 _}  | #               | r8(action)  | 3(goto)    | "F".val:='d'.lexval(语义动作)      | t4:=2     |
| 8 : { 0 , 1 , 6 , 3 }  | {# _ }{E 1 t3}{+ _ }{F 2 t4} | #               | r6(action)  | 15(goto)   | "T".val:="F".val(语义动作)         | t5:=t4    |
| 9 : { 0 , 1 , 6 , 15 } | {# _ }{E 1 t3}{+ _ }{T 2 t5} | #               | r1(action)  | 1(goto)    | "E".val:="E".val+"T".val(语义动作) | t6:=t3+t5 |
| 10 : { 0 , 1 }         | {# _ }{E 3 t6}               | #               | acc(action) |            |                                |           |

得出的四元式序列两种形式分别如下所示:

```

t1:=1
t2:=t1
t3:=t2
t4:=2
t5:=t4
t6:=t3+t5

```

```

(:= , 1 , _ , t1)
(:= , t1 , _ , t2)
(:= , t2 , _ , t3)

```

```
(:= ,2 , _ , t4)
(:= ,t4 , _ , t5)
(+ , t3 , t5 , t6)
```

## 五 推广与改进

### 改进方向之一：

问题三中优化四元式，使用代码优化技术，因为该问题中有众多的无用规约操作，故会产生众多无效变量，比如在  $1+2$  运算中会产生  $t1:=1; t2:=t1; t3:=t2$ ，其中有两项赋值是无效赋值。通过代码优化技术可以大大减少无效变量的数量。其次，制定合适的产生式同样可以使得有效规约的比例更高。

### 改进方向之二：

我制定的语言（称为 X 语言）尚且可以修改语法分析中的产生式规则，从而让语言更加完整。

### 改进方向只三：

在构建文法的 LR(1) 项目集规范族时，我发现有众多同心集，实际上如果问题二有更好的要求可以构建 LALR(1) 分析从而大量减少状态的数量，这不仅会减少存储空间，同样会使得符号串预测分析的复杂度下降。

## 六 总结与心得

通过本次课程设计，我对编译原理中词法分析、语法分析、初步语义分析等都有了一定的了解。通过程序实现也极好的加深了对于编译原理中各种算法的理解。

在本次编译原理中，在处理问题一时，我在如何构建自动机方面陷入了一段时间的思索之中，随后试想了通过组合四个 DFA 的方式构建完整的识别 DFA，这个过程我认为充满了思想实验，对我理解问题，寻找问题建模的方法等都有一定的提升。在处理过程中如何制定合适的数据结构又是十分重要的一个基础性条件。在我选择合适的数据结构之后，实现变得非常方便。

对于问题二，我认为相对于问题一来说实现更加快速一些，因为在《编译原理》书上都有各种算法的严格求解步骤，按照这样的步骤层层递进，即能快速实

现 LR(1) 分析方法。在解决问题二时，我发现，对于问题的分阶段、分层次解决是非常重要的，比如说对于问题二 LR(1) 分析方法的构建过程中，首先从文法符号能否推出  $\epsilon$  开始，紧接着求所有文法符号的 FIRST 集合，然后构建 LR 项目集规范族，即识别活前缀的自动机，随后构建 ACTIONGOTO 表，然后构建待识别符号序列的预测分析表。在这个过程中，每一步都依赖于上一步的算法，所以在进行当前步骤时，必须首先保证下层算法的正确性，在这样的自底向上进行程序设计的过程中，我明白了分步测试的重要性，这也是为什么在我的文件代码中有很多测试文件存在的原因。在问题一中对于每一类自动机我都写了一些串进行测试，在问题二中，我几乎将书上能够测试的例子全都用来做了测试，一共有六个测试文件。并且在过程中也发现过一下小 bug，对其进行解决之后，发现程序能够完整高效地实现预定的要求，这个过程非常令人欣喜。

对于问题三，我首先也是参考了《编译原理》书中的 167–168 页的 S-属性文法，对 168 页分析方法的观察让我得出可以分析表达式文法的算法，从而能够对符合规则的可计算式进行计算和产生四元式。

总之，在三个问题的解决中，《编译原理》这本书提供了大量的算法基础以及案例，对我实现上述三个问题可以说提供了最大的帮助，同时，在实现的过程中，我利用好了 C++ 语言的特性，由于我对 C++ 以及其各种数据结构都非常了解，从而能够快速选择合适的数据结构以及合适的实现算法，在这个过程中我对数据结构的运用也更加熟练，对工程问题的实现也不再是入门阶段，对问题的思考、分解更是得到了不小的提升，可谓受益匪浅。