浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	许子钧 薛司悦 赵欣岚
学　　院：	计算机科学与技术
系：	
专　　业：	
学　　号：	3160100056 3160104527 3160104685
指导教师：	李莹老师

Contents

1 序言

1.1 编译器概述

此编译器为基于 Java 语言开发的针对 C-minus 语言的简易编译器。我们的报告会针对词法分析、语法分析、语义分析、中间代码生成、目标代码生成、错误修复这六个部分而展开。

1.2 文件说明

- Main.java: 编译器入口
- 词法分析：Word.java(定义 Word 类)，Lexer.java(词法分析实现)
- 语法分析：ParserTreeNode.java(定义 ParserTreeNode 类)，Pareser.java(实现语法分析)
- 代码生成：Quadruple.java(定义四元组类)，具体实现代码在 Parser.java 中
- 错误处理：Error.java(定义错误类)，具体实现在 Lexer.java 与 Parser.java 中

1.3 小组分工

- 薛司悦：词法分析、错误处理
- 许子骏：中间代码生成、目标代码生成、代码整体框架、测试
- 赵欣岚：语法分析、语义分析

2 词法分析

2.1 预定义

为了使词法分析部分更简洁，我们首先抽象出两个类，Word 和 Error。其中 Word 类主要实现的是对于一些运算符（'+', '-', '*', '/'）的定义，以及关键字、标识符的判断等功能。Error 类则是对于出现错误的部分进行一个标记和存储。

2.1.1 Word

首先，Word 类中的字段类型一共有以下几种：

Keyword, Operator, Integer constant, Character constant, Boolean Constant, Identifier, Boundary sign, End sign, Undefined.

Operator 包含以下符号，将其储存在一个 ArrayList 中：

'+', '-', '*', '/', '++', '–', '>', '<', '=', '>=', '<=', '==', '!=', '&&', '||', '!', ':', '?', '|', '&'

其中还可细分出 Arithmetic Operator：'+', '-', '*', '/'

和 Boolean Operator：'>', '<', '=', '>=', '<=', '==', '!=', '&&', '||', '!'
Boundary Sign 包含以下符号，将其储存在一个 ArrayList 中：
"(", ")", "", "", ";", ","
Keyword 包含以下符号，将其储存在一个 ArrayList 中：
"void", "main", "int", "char", "if", "else", "while", "for", "scanf", "printf"
我们定义了以下几个功能函数在 Word 类中：

- isKeyword：通过查询 Keyword List 来判断当前 word 是否是 Keyword
- isOperator：通过查询 Operator List 来判断当前 word 是否是 Operator
- isBoundarySign：通过查询 BoundarySign List 来判断当前 word 是否是 Boundary Sign
- isArithmeticOp：判断当前 word 是否是 Arithmetic Operator
- isBooleanOp：判断当前 word 是否是 Boolean Operator

同时，Word 类还有以下几个私有属性（即需要通过函数来访问或者修改）：

- id：该 word 的序号
- line：所处行
- value：word 的内容
- type：类型

2.1.2 Error

Error 类比较简单，因为我们只需要在词法分析中锁定 Error 的位置，所以 Error 类中只定义了关于 error word 的一些私有属性和处理属性的相关函数

- id：该 error 的序号
- info：错误信息
- line：错误所在行
- word：error word
- 其中每个私有属性对应两个函数，一个用于设定该属性的值，一个用于访问该属性的值。

2.2 词法分析

在词法分析中我们要主要做的是对词的类型进行区分和判断，判断的范围有以下几个：标识符/关键字/整数/字符/运算符/注释/错误等，以及最后的一个综合输出。

首先我们定义一个主函数 Lexer，在该函数中我们创建两个 ArrayList，一个用于储存 word，一个用于储存 error。然后将传入的文件按行分开，分别对每一行调用 Analyze 函数进行分析。

因为 Analyze 的体量较大，所以我们额外定义了如下的功能函数：

- isInteger：该函数用于判断传入的 word 是否是整数
- isIdentifier：该函数用于判断传入的函数是否是标识符
- isLexError：该函数用于判断当前是否出错

2.2.1 Anaylze 函数

1. Identifier

   首字符为字母或者'_' 的时候进入该分支，然后扫描到整个句子末端或者扫描到 Tab 键等类似的符号作为终止，提取出来一个 word 进行判断。

   首先在 Keyword List 里面进行查找，如果找到，则将该 word 归为 Keyword；

   如果找不到继续在 Identifier List 里面查找，如果找到，则将该 word 归为 Identifier；

   以上两个都不满足的情况下，则判为 error，录入 error 信息，并将该词存在 error list 中。

2. Integer

   判断首字符是否为数字，如果是继续扫描到和上面分支一样的终止条件，然后提取 word。

   根据功能函数 isInteger 判断是否为 Integer，是则归为 Integer，否则归为 Error

3. Character

   判断首字符是否为" ' "，如果是，在不超过该行长度，以及扫描到的字符确实在 ASCII 中的情况下，继续扫描到" ' "。提取该 word。

   判断扫描是否溢出（即超过该行长度），如果没有溢出，则归为 Character，否则计入 error。

4. Operator：('+', '-') 等

   遇到'+', '-' 等可能不止一个符号的 operator 的情况，我们会直接向后扫描，看后面是否还有其他可能匹配的字符，例如'+' 后面可能再跟一个'+'，如果有的话就将两个字符打包为一个 word 存起来，否则只存一位。

5. Comment

   判断首字符是否为'/'，是的话继续扫描，如果扫到第二个'/' 就终止扫描，因为说明后面的部分均为注释内容；如果扫到'*'，就说明是多行注释，下一行要注意。如果只是单个字符，则归为 Operator 中的'/'。

6. Others

   此部分主要是对一些 Tab 键、终止符进行归类，同时 default 为 error。

2.2.2 Output 函数

此部分直接利用 Java 自带的库函数进行分别输出即可

3 语法分析

3.1 调整 C Minus 的 BNF 语法

本次编译器采用 LL(k) 进行语法分析，需要对附录提供的 BNF 语法进行消除左递归以及提取左公因子的处理。以下列出需要处理的部分产生式以及处理结果：

1. 消除左递归

```
local-declarations -> local-declarations var-declaration|empty
local-declarations -> var-declaration local-declarations|empty

statement-list -> statement-list statement|empty
statement-list -> statement statement-list|empty

additive-expression -> additive-expression addop term|term
additive-expression -> term add-exp
add-exp -> addop term add-exp | empty

term -> factor term'
term' -> mulop factor term | empty
```

2. 提取左公因子

```
selection-stmt -> if(expression) statement
| if (expression) statement else statement
selection-stmt -> if (expression) statement else-part
else-part -> else statement | empty

simple-expression -> additive-expression relop additive-expression
```

```
| additive−expression
simple−expression −> additive−expression add−exp−follower
add−exp−follower −> relop additive−expression | empty
```

3.2 对 C Minus 语法的简单扩充

1. 增加 statement-block 取代 selection-stmt 与 while-stmt 中的 statement
   在 C-Minus 中 if 语句后面仅能执行单独的一条语句 statement，加入的 statement-block 是用划分的语句集合，在产生式上表示为：

```
statement−block −> {statement−list}
```

2. 增加循环结构 for-stmt

```
statement −> fot−stmt
for−stmt −> for(expresison;simple−stmt;expression) statement−block
```

3.3 LL(K) 分析的具体实现

3.3.1 数据结构说明

本次实验中的语法分析基本上依赖 LL(1) 进行，但也存在向前多读了一个 toekn 的情况。使用 LL(K) 分析语句时，需要三个数据结构作为支撑：分析栈 (AnalyzeStack)，语义栈 (semanticStack)，以及用于存储输入程序的字符序列 (wordList)。在语法分析与语义分析中，为了便于计算输出，包括类型、变量名、常量值等都以字符串形式存储。

1. 分析栈
   分析栈调用 Java 中的 Stack 类，其中存储元素的类型为自定义的 ParseTreeNode 类。后者主要记录的内容如下：

   - Node type: private String type;
   - Node name: private String name;
   - Node value: private String value;

   由 type 区分的 ParseTreeNode 分为三类：terminal, nonterminal, actionSignal。其中前两者对应于编译原理教学内容中的终结符与非终结符；而 actionSignal 服务于语义分析。

2. 语义栈：存储当前计算涉及的变量，以 String 形式存储在 Stack 中

3. 字符序列

   在 LL(1) 分析过程中，对于经过词法分析处理的待编译程序内容进行顺序访问。用于存储的数据结构为 ArrayList。

3.3.2 Parser 模块重要函数说明

语法分析是基于词法分析的结果以及 C Minus 语法的 parsing table 的进一步操作。对于当前的分析栈栈顶内容、下一个将要被处理的字符，依据 parsing table，做出相应的 match 或 generate 操作。相关的数据结构与函数都包含在类 Parser 中。以下是重要函数及其功能、思想说明。

1. grammarAnalyze

   - 获取当前分析栈栈顶元素、输入中的下一个 token。
   - 判断栈顶与获取的 token 是否为'#'(人为规定的终止标记)。如果两者都为"#"，语法分析结束。
   - 若尚未匹配到"#"，依据当前栈顶元素决策：若为 nonterminal，则参考输入中的下一个 token 与 parsing table 执行 generate 操作；若为 actionSignal，对语义栈进行操作，生成相关四元组并存储；若为 token，进行 match。

2. terminalOp
   当分析栈栈顶元素类型与输入的下一个 token 相匹配时，进行 match 操作：分析栈弹出栈顶元素，wordList 删去首位的 word。若出现不匹配的情况，报错 Grammar error。

3. nonTerminalOp

   当分析栈栈顶元素类型为非终结符时，需要依据输入的下一个 token 选择合适的产生式进行 generate 操作。由于语法分析中非终结符、终结符类型相对较多，实现过程中并没有列出 parsing table，而是仅仅列出各个非终结符的 first set 与 first set 中含有 empty 项的非终结符的 follow set。这两个表格参见附录。

   在实现过程中，并非每一个非终结符都有特定表示：比如 expression-stmt，没有以非终结符的状态出现在分析栈中，而是以 expression; 和; 的形式入栈；而 statement -> expression-stmt|selection-stmt|while-stmt|for-stmt 在实现过程中直接以 expression-stmt 等的对应形式入栈。

   为了辅助后续的语义分析与中间代码的四元组建立，在各个状态入栈的同时，会将部分 actionSignal 压入分析栈：比如对于 selection-stmt 中 if 语句截止与 else-part 的起止处会标记对应的 actionSignal，当分析栈栈顶出现这些符号时，结合语义栈生成三地址码的四元组。

   下面以分析栈栈顶为 statement 的情况为例，说明分析栈的分析操作。

   在 parser 中，以 D 表示 statement，E 表示 else-part，G 表示 statement-block，H 表示 expression。当下一个输入的 token 为 if 时，statement 出栈，产生式中各个部分倒序入栈。if 语句之后的 statement-block 前后分别入栈了 IF_FJ 与 IF_BACKPATH_FJ，用于标记这一部分的起止，指导三地址码的生成。如果输入的下一个 token 不属于 statement 的 first set，考虑到 statement 的 first set 中包含 id，认为可能是输入了非法的 identifier，进行报错。

```
case 'D':
  //statement -> expression-stmt|selection-stmt|while-stmt|for-stmt
  if (firstWord.getWordValue().equals("if")){
    // selection-stmt -> if (expression) statement-block else-part
    AnalyzeStack.pop();
    AnalyzeStack.push(IF_BACKPATCH_RJ);
    AnalyzeStack.push(E);
    AnalyzeStack.push(IF_RJ);
    AnalyzeStack.push(IF_BACKPATCH_FJ);
    AnalyzeStack.push(G);
    AnalyzeStack.push(IF_FJ);
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, ")
        ", null));
    AnalyzeStack.push(H);
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, "(
        ", null));
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, "
        if", null));
  }
  ------------other possibilities are not mentioned here
      ----------
break;
```

4. actionSignOp（服务于语义分析）
   语法分析过程中对于出现的 identifier 要记录在语义栈中，还有常量。每一个算符出现后都需要在语义栈中进行更新（pop 操作数，push 结果值，结果值是临时变量）。标记 if，for，while 等的各个模块起止位置，出现的时候要结合语义栈写出跳转指令（RJ 是无条件跳转，FJ 是 ARG1 为 0 时调到 RES 去）。模块的起止位置存储在 if、while、for 各自的小栈里面，是当前四元组的位置。

3.4 错误处理

当分析栈顶为非终结符，且遇到的下一个输入 token 不属于其 first set 或在非终结符 first set 包含 empty 而 token 不属于其 follow set 时，根据上述集合内容进行报错：

- 当非终结符的 first set 中包含 identifier 时，报错为非法变量名。
- 当非终结符的 first set 为各种算符时，报错为未定义的算符。

当分析栈栈顶为终结符而下一个 token 与之不匹配时，报错为 Grammar，并打印当前字符。

4 语义分析

中间代码的生成可以视作属性文法的计算。

4.1 identifier 以及 constant 的属性值记录

将这两者以 String 的形式存储在当前如 expression 等拥有数值意义的节点的 name 中，便于后续值的传递。由于 C Minus 文法考虑的计算式为中缀表达式，下一个算符涉及的属性值必然属于最近出现的变量或常量，故将值压入语义栈留待后续属性值计算。

下面以 factor 处的属性操作为例进行说明。

- 当下一个 token 为 ID 时，弹出 factor，入栈一个标记为 id 的终结符用于这个 token 的 match 操作，再入栈一个 ASS_M 操作，修改 M 的属性值为当前 token 属性值，并且将之压入语义栈。
- 下一个 token 为 NUM 时，操作同上。

```
case 'M'://factor -> (expression)|ID|NUM
  if(firstWord.getWordType().equals(Word.IDENTIFIER)){
    AnalyzeStack.pop();
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, "id",
        null));
    AnalyzeStack.push(ASS_M);
  }else if(firstWord.getWordType().equals(Word.INT_CONST)){
    AnalyzeStack.pop();
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, "num",
        null));
    AnalyzeStack.push(ASS_M);
  }
```

4.2 表达式中的属性值计算

在 C Minus 文法中涉及属性值计算（即产生式中包含算符）有如下几处：（除此之外，赋值表达式也涉及属性计算，在此不做列举）。

```
add-exp-follower -> relop additive-expression
add-exp -> addop term add-exp
term' -> mulop factor term
```

这里类似于 P-code 中的思想：取得算符与最近入语义栈的两个操作数，计算结果后结果入栈。并将结果值赋给当前非终结符作为属性值。最后将动作记号由分析栈弹出。

```
if (stackTop.getNodeName().equals("@DIV_MUL")) {
  if (OP != null && (OP.equals("*") || OP.equals("/"))) {
    //获取操作数
    ARG2 = semanticStack.pop();
    ARG1 = semanticStack.pop();
    //建立结果临时变量
    RES = newTemporary();
```

```
        //生成对应四元组
        Quadruple quadruple = new Quadruple(++quadrupleCount, OP, ARG1,
            ARG2, RES);
        //存储四元组
        quadrupleList.add(quadruple);

        L.setNodeValue(RES);//L : term
        //语义栈存入结果值
        semanticStack.push(L.getNodeValue());

        OP = null;
    }
    //动作记号出栈
    AnalyzeStack.pop();
}
```

4.3 属性值的传递

依旧以 factor 为例，当需要采用产生式 factor -> (expression) 时，需要进行属性值的传递：由最近的 expression 将值传给 factor，动作指令为 TRAN_HM。这个指令需要最先入栈，因为值的传递需要在计算出 expression 的属性值之后进行。

```
if (firstWord.getWordValue().equals("(")){
    AnalyzeStack.pop();
    AnalyzeStack.push(TRAN_HM);//M:factor
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, ")", null)
        );
    AnalyzeStack.push(H);//expression
    AnalyzeStack.push(new ParseTreeNode(ParseTreeNode.TERMINAL, "(", null)
        );
}
```

5 优化考虑

5.1 语法分析优化

1. 将部分 nonterminal 直接用其产生式代替
   对 LL(k) 分析的代码实现中，考虑到过多的中间状态会增加栈操作的复杂性，并且重写 BNF 产生式会降低语法可读性，对于如 for-stmt 等只有一个产生式的 nonterminal，当其出现在产生式的右边时，入栈的是它对应的产生式。类似的还有 while-stmt，expression-stmt 等非终结符。因此在语法分析部分的例子 statement 的处理中，依据下一个输入的 token，statement 被直接转化为 epression; 或; 等入栈，而非原本 BNF 中的 expression-stmt。

2. 采用向前获取两个字符的方式避免为消除左公因子设置更多的 nonterminal
对于产生式

```
expression->var=expression|simple-expression
```

我们可以从 C Minus 的 BNF 语法中推知 simple-expression 与 var = expression 两者的 first set 都包含相同的成分：id。但是为此提取左公因子再进行实现，需要增加新的非终结符，产生额外开支。考虑到这个产生式本身并不复杂，添加新的非终结符也会影响文法的可读性。所以在这里我们不仅仅查看了输入列中的第一个 token，而是又多获取了一个 token。当且仅当 first-token.type == identifier 而且 second-token.value == "=" 时，采用前一个产生式进行 generate; 除此之外，从分析栈中弹出 expression，入栈 simple-expressino。

6 代码生成

6.1 中间代码生成

中间代码生成器根据语义分析器的输出生成中间代码。中间代码可以表示成许多种形式，像是 P-code 和三地址码（Three address code），共同特征是与具体机器无关。本项目中间代码以三地址码表示，三地址码的优点是便于阅读和优化，可以利用四元式（quadruple）实现。

四元组格式：（运算符，运算对象 1，运算对象 2，结果），通过 Quadruple 类储存四元组。

```
// Index.
int id;
// Operator.
String op;
// First argument.
String arg1;
// Second argument.
String arg2;
// Result.
String result;
```

在语法分析阶段，同时也保存所有的四元组。生成中间代码时，仅需遍历四元组列表输出到 output 文件夹下的 quadruple.txt 文件中。

6.2 目标代码生成

目标代码生成是编译器的最后一个阶段。在生成目标代码时要考虑许多问题，如计算机的系统结构、指令系统、寄存器的分配以及内存分配的设计等。编译器生成的目标程序代码可以有多种形式：汇编语言、可重定位二进制代码、内存形式。本项目目标代码以 x86 汇编语言（x86 assembly language）表示。

通过遍历四元组列表，分析各个运算符，执行对应指令及其所需参数，输出 output 文件夹下的 target_code.txt 文件中。

```
// Each quadruple.
Quadruple quadruple = quadrupleList.get(i);

// '='
if (quadruple.op.equals("=")) {
  outputString = "MOV " + quadruple.result + ", " + quadruple.arg1;
// '++'
} else if (quadruple.op.equals("++")) {
  outputString = "INC " + quadruple.arg1;
// '+'
} else if (quadruple.op.equals("+")) {
  outputString = "ADD " + quadruple.result + ", " + quadruple.arg1;
// '-'
} else if (quadruple.op.equals("-")) {
  outputString = "SUB " + quadruple.result + ", " + quadruple.arg1;
// '*'
} else if (quadruple.op.equals("*")) {
  outputString = "MUL " + quadruple.result + ", " + quadruple.arg1;
// '/'
} else if (quadruple.op.equals("/")) {
  outputString = "DIV " + quadruple.result + ", " + quadruple.arg1;
// 'RJ'
} else if (quadruple.op.equals("RJ")) {
  outputString = "JMP " + quadruple.result;
// 'FJ'
} else if (quadruple.op.equals("FJ")) {
  outputString = "JZ " + quadruple.result;
// '>'
} else if (quadruple.op.equals(">")) {
  outputString = "JG " + quadruple.result;
// '<'
} else if (quadruple.op.equals("<")) {
  outputString = "JL " + quadruple.result;
}
```

7 测试案例

1. 赋值，比较，for 回圈，嵌套 for 回圈，加运算

   - 测试内容

```
void main()
{
  int sum = 0, a = 0;
  int limit = 10;
  for (int i = 1; i < limit; i++)
  {
    sum = sum + a;
    for (int i = 1; i < 11; i++)
    {
      a = a + i;
    }
  }
}
#
```

- 中间代码

```
Quadruples:
--------------------------------
No.     OP      ARG1    ARG2    RES
1       =       0       /       sum
2       =       0       /       a
3       =       10      /       limit
4       =       1       /       i
5       <       i       limit   T1
6       FJ      T1      /       18
7       +       sum     a       T2
8       =       T2      /       sum
9       =       1       /       i
10      <       i       11      T3
11      FJ      T3      /       16
12      +       a       i       T4
13      =       T4      /       a
14      ++      i       /       i
15      RJ      /       /       10
16      ++      i       /       i
17      RJ      /       /       5
```

前四项皆是赋值语句，操作数是'='，第一个参数是被赋的值，结果是变量名。
第五项是比较语句，操作数是'<'，第一个参数是左值，第二个参数是右值，结果储存在结果中。
第六项是跳转语句，操作数是 FJ，指出如果第一个参数是零就跳转到结果的语句。
第七项是加运算语句，操作数是'+'，将第一个参数和第二个参数相加并储存在

结果中。
第十四项是 ++ 运算语句，操作数是'++'，将第一个参数加一并储存在结果。
第十七项是跳转语句，操作数是 RJ，跳转到结果指出的语句。

- 目标代码

```
1  MOV sum, 0
2  MOV a, 0
3  MOV limit, 10
4  MOV i, 1
5  JL T1
6  JZ 18
7  ADD T2, sum
8  MOV sum, T2
9  MOV i, 1
10 JL T3
11 JZ 16
12 ADD T4, a
13 MOV a, T4
14 INC i
15 JMP 10
16 INC i
17 JMP 5
```

前四项皆是 MOV 指令，为赋值语句，将第二个参数赋值给第一个参数。
第五项是 JL 指令，为跳转语句，小于时跳转。
第六项是 JZ 指令，为跳转语句，如果前一句语句为零，就跳转到第一个参数的地址。
第七项是 ADD 指令，为算术语句，将第二个参数加到第一个参数。
第十四项是 INC 指令，为算术语句，将第一个参数增加一。
第十五项是 JMP 指令，为跳转语句，跳转到第一个参数的地址。

2. while 回圈

- 测试内容

```
1  void main()
2  {
3    int i = 0, limit = 10, sum = 0;
4
5    while (i < limit)
6    {
7      i = i + 1;
8      sum = sum + i;
9    }
```

```
}
#
```

- 中间代码

```
Quadruples:
--------------------------------------
No.     OP      ARG1    ARG2    RES
1       =       0       /       i
2       =       10      /       limit
3       =       0       /       sum
4       <       i       limit   T1
5       FJ      T1      /       11
6       +       i       1       T2
7       =       T2      /       i
8       +       sum     i       T3
9       =       T3      /       sum
10      RJ      /       /       4
```

前三项皆是赋值语句，操作数是'='，第一个参数是被赋的值，结果是变量名。
第四项是比较语句，操作数是'<'，第一个参数是左值，第二个参数是右值，结果储存在结果中。
第五项是跳转语句，操作数是 FJ，指出如果第一个参数是零就跳转到结果的语句。
第六项是加运算语句，操作数是'+'，将第一个参数和第二个参数相加并储存在结果中。
第十项是跳转语句，操作数是 RJ，跳转到结果指出的语句。

- 目标代码

```
MOV i, 0
MOV limit, 10
MOV sum, 0
JL T1
JZ 11
ADD T2, i
MOV i, T2
ADD T3, sum
MOV sum, T3
JMP 4
```

前三项皆是 MOV 指令，为赋值语句，将第二个参数赋值给第一个参数。
第五项是 JZ 指令，为跳转语句，如果前一句语句为零，就跳转到第一个参数的地址。
第六项是 ADD 指令，为算术语句，将第二个参数加到第一个参数。

第十项是 JMP 指令，为跳转语句，跳转到第一个参数的地址。

3. if-else 语句，嵌套 if-else 语句

- 测试内容

```
void main()
{
  int a = 3, b = 1;
  if (a > b)
  {
    c = 7;
    if (a > c)
    {
      c = a;
    }
    else
    {
      c = a + b;
    }
  }
  else
  {
    c = 1;
  }
}
#
```

- 中间代码

```
Quadruples:
--------------------------------------
No.     OP      ARG1    ARG2    RES
1       =       3       /       a
2       =       1       /       b
3       >       a       b       T1
4       FJ      T1      /       13
5       =       7       /       c
6       >       a       c       T2
7       FJ      T2      /       10
8       =       a       /       c
9       RJ      /       /       12
10      +       a       b       T3
11      =       T3      /       c
12      RJ      /       /       14
13      =       1       /       c
```

前两项皆是赋值语句，操作数是'='，第一个参数是被赋的值，结果是变量名。第三项是比较语句，操作数是'>'，第一个参数是左值，第二个参数是右值，结果储存在结果中。第四项是跳转语句，操作数是 FJ，指出如果第一个参数是零就跳转到结果的语句。第九项是跳转语句，操作数是 RJ，跳转到结果指出的语句。第十项是加运算语句，操作数是'+'，将第一个参数和第二个参数相加并储存在结果中。

- 目标代码

```
1  MOV a, 3
2  MOV b, 1
3  JG T1
4  JZ 13
5  MOV c, 7
6  JG T2
7  JZ 10
8  MOV c, a
9  JMP 12
10 ADD T3, a
11 MOV c, T3
12 JMP 14
13 MOV c, 1
```

前两项皆是 MOV 指令，为赋值语句，将第二个参数赋值给第一个参数。

第三项是 JG 指令，为跳转语句，大于时跳转。

第四项是 JZ 指令，为跳转语句，如果前一句语句为零，就跳转到第一个参数的地址。

第九项是 JMP 指令，为跳转语句，跳转到第一个参数的地址。

第十项是 ADD 指令，为算术语句，将第二个参数加到第一个参数。

A C Minus 语法部分 nonterminal 的 first set

nonterminalfist	set
program	void
local-declarations	int, char, bool, empty
var-declaration	int, char, bool
statement-list	ID, if, while, empty
type-specifier	int, char, bool
statement	ID, if, while, for
expression	ID
else-part	else, empty
statement-block	{, ID, if, while, for
simple-expression	(, ID, NUM
additive-expression	(, ID, NUM
add-exp-follower	<=, ==, >=, <, >, !=, ==, empty
add-exp	+, -, empty
relop	<=, ==, >=, <, >, !=, ==
term	(, ID, NUM
factor	(, ID, NUM

B C Minus 语法部分 nonterminal 的 follow set

nonterminal	follow
setlocal-declarations	}, ID, if, while
else-part	ID, if, while
statement-list	}
add-exp-follower	;,)
add-exp	relop, ;,)
term	+, -, ;,)