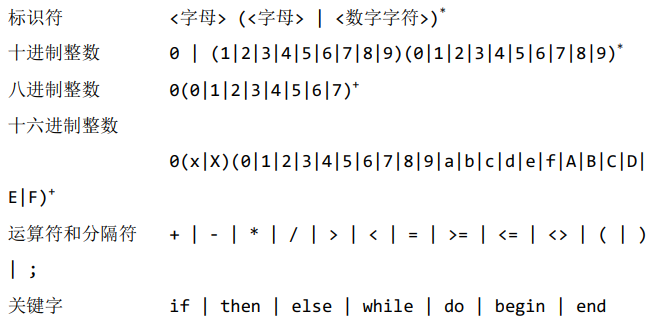
编译原理实验报告

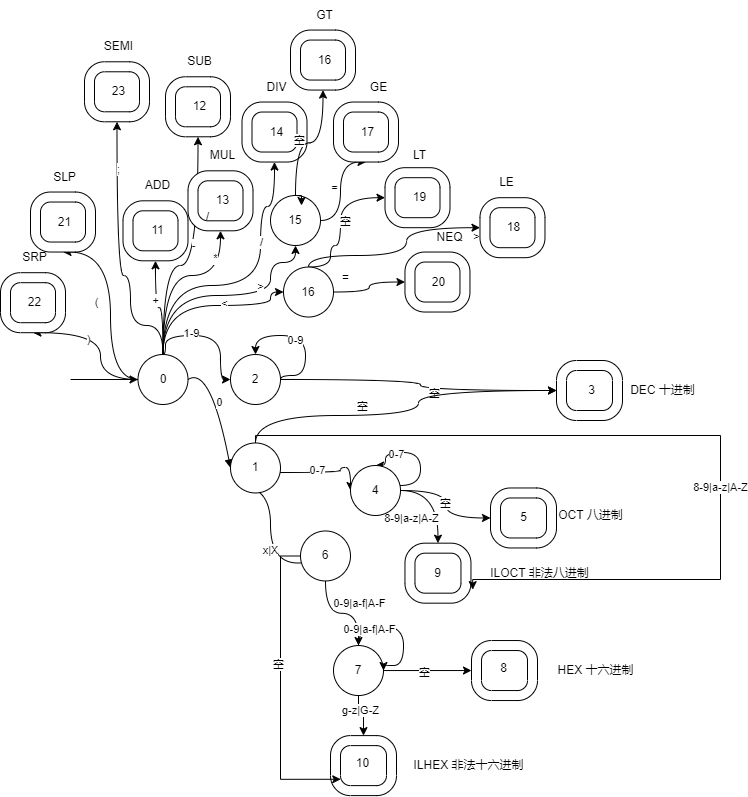
21074113张嘉茵

21074114王林诺

1. 词法分析子系统
2. 词法的正规式描述



1. 状态图



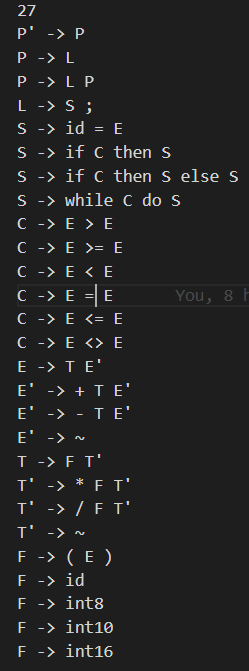
其中双线状态表示终状态，对应种别名在状态边标出。

1. 词法分析程序的主要数据结构与算法
   1. 数据结构
      1. class symbolTableNode 存储语法单元基本属性
         1. string type 存储种别值
         2. int typeCode 以数字形式存储种别值
         3. string props 存储字符串属性值
         4. int prop 存储数值类型属性值
      2. 全局变量
         1. 与tellType函数配合进行字符类型判断
            1. IS\_LETTER 0
            2. IS\_DIGIT 1
            3. IS\_SYMBOL 2
         2. enum TOKEN枚举类型用于配合typeCode以数字形式存储种别值
   2. 算法
      1. queue<string> sep\_words ( string in )
         1. 输入：从用户端直接获取到的字符串
         2. 输出：分割好的序列
         3. 功能：将从用户端直接获取到的字符串进行分割，按顺序存放在队列里返回给上层函数
         4. 实现方式简述：逐字符遍历字符串，调用tellType函数判断当前字符的类型，通过标志位（初值为输入字符串第一个字符的类型）记录上一次类型发生变化后的类型。当类型不发生改变且当前类型不为symbol时，在临时字符串t尾部附加当前字符；当类型发生改变时，记录变化后的类型值，将当前的t字符串push到队列里。  
            有几个特例进行特殊处理：
            1. 为16进制数字设置标志位，当前一个字符为0，当前字符为x或X时，设置flag为1，当flag为1时，进入特殊处理方式，不再按上文处理方式进行处理，持续读入字符直到分隔符或symbol类型截止并压栈。
            2. 当前一个输入字符为<或>时，判断当前字符是否为可以与前一符号构成整体的=或>，若可以构成整体，将当前字母附加到字符串尾部。
      2. symbolTableNode scan\_digit ( std::string in, int\* error )
         1. 输入：从sep\_words获取的判断为数字类型的字符串，int型错误标识符
         2. 输出：一个正确赋值好属性、类型的数字symbolTableNode 对象
         3. 功能： 根据状态转移图构建状态机，处理被判断为数字类型的字符串。
         4. 实现方式简述: 使用一个int类型变量state储存当前状态，按照状态图进行状态机构造。使用一个int类型变量i用于逐字符遍历输入字符串，将字符串中的字符按位输入，匹配当前状态。成功匹配后，i值加一，匹配下一位，直到字符串结束。
         5. 错误处理：
            1. 在匹配时，使用error标志位对状态机判断出的错误八进制和十六进制情况进行记录，并在匹配结束时输出报错。
            2. 返回到上层函数时，若error位为1，则不把当前Node加入存储Node序列的deque中，完成去除错误数字节点的错误处理。
      3. symbolTableNode scan\_letter ( std::string in )
         1. 输入：从sep\_words获取的判断为字符类型的字符串
         2. 输出：一个正确赋值好属性、类型的字符串symbolTableNode 对象
         3. 功能：处理被判断为字符（即IDN型）类型的字符串
         4. 实现方式简述：对输入字符串进行匹配，若匹配到if,then,else等关键字，记录Type为对应值(if then else)，匹配不上的记录为普通字符串类型，props设置为字符串内容，生成symbolTableNode节点并输出至上层函数。
      4. symbolTableNode scan\_else ( std::string in, int\* error )
         1. 输入：从sep\_words获取的字符串，int型错误标识符
         2. 输出：一个正确赋值好属性、类型的symbolTableNode 对象
         3. 功能：处理被判断为符号类型的字符串
         4. 实现方式简述：对输入字符串进行匹配，若匹配到+,=,\*等符号，记录Type为各类符号类型（ADD,GE,MUL），生成symbolTableNode节点并输出至上层函数。
         5. 特殊情况处理：当输入字符串检测到>或<时，可能为大于小于一个字符的符号，也可能是<=,>=,＜＞两个字符的符号，故要读取下一个字符，进行匹配，优先匹配两个字符的符号。
         6. 错误检测：
            1. 当没有匹配到符号、两个字符的符号匹配后字符串没有结束时，将error置为1，并输出错误信息。
            2. 返回到上层函数时，若error位为1，则不把当前Node加入存储Node序列的deque中，完成去除错误数字节点的错误处理。
2. 语法分析子系统
3. LR(1)分析法

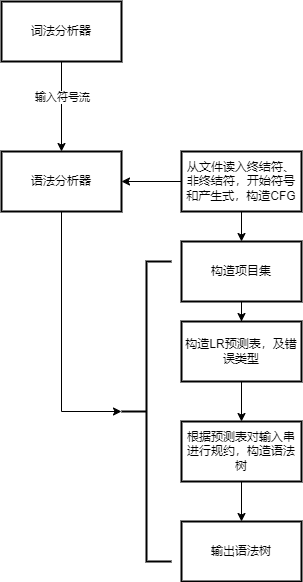
完成SLR(1)后，发现冲突过多之后选择完成LR(1)。

完成消除左递归，并添加拓广。

处理后的产生式如下：



1. 语法分析子系统结构



1. 语法分析子系统的主要数据结构与算法
   1. 数据结构
      1. Struct TreeNode 语法树节点
         1. string places

存储属性值，标识符存储字符串，数字存储string化的属性值

* + - 1. int proId

存储生成该节点的产生式ID

* + - 1. int time

记录当前节点被遍历的次数

* + - 1. string syntaxType

存储语法类型

* + - 1. vector<TreeNode\*> children

存储当前节点的子节点

* + - 1. TreeNode \*father

存储当前节点的父节点

* + - 1. int place

存储数字属性值

* + - 1. string code

存储用于三地址代码生成的代码段

* + - 1. int next

存储下一条要执行语句的label

* + - 1. int begin

while循环时，存储循环开始执行语句的label

* + - 1. int T

if判断时，存储条件为真时下一条要执行语句的label

* + - 1. int F

if判断时，存储条件为假时下一条要执行语句的label

* + 1. struct Production 产生式
       1. string left产生式左侧符号
       2. vector<string> right产生式右侧符号
    2. Struct item 项目
       1. Production pro 产生式
       2. int dot 点的位置,-1表示到达最后
       3. set<string> symbol 展望串
    3. Struct Goto goto表相关数据结构
       1. int first 现在状态
       2. string sign 接收符号
       3. int next 下一个状态
    4. Struct Action action表相关数据结构
       1. ActionState state 存储处理过程中的状态
       2. int num = -1 下一个状态，初始值为-1
    5. Class CFG\_LR1 CFG文法
       1. unordered\_set<string> VT 终结符集
       2. unordered\_set<string> VN 非终结符集
       3. string startSymbol 开始符号
       4. map<int, Production> production 产生式集
    6. Class PredictTable 预测表
       1. map<int, Production> production; 产生式集
    7. enum ActionState
       1. 定义处理当前序列（生成树和三地址代码）时的几个状态：
          1. STATE 移进
          2. REVERSE 规约
          3. ACCEPT 接收
          4. EMPTY 空
       2. 处理时产生错误时，转到对应State，显示错误类型，报告处理出错
          1. ERROR1 未等到比较符号
          2. ERROR2 只有)
          3. ERROR3 未等到赋值或语法结构
          4. ERROR4 未等到运算数或标识符
          5. ERROR5 未等到运算符 + -等
          6. ERROR6 未等到赋值=
          7. ERROR7 while 或if语法不全
          8. ERROR8 缺少;
          9. ERROR9 缺少）
  1. 算法
     1. 构造项目集
        1. createItem
           1. 计算First和Follow集合

通过辅助函数makeFirst() 和 makeFollow()，计算每个符号的First和Follow集合。

* + - * 1. 初始化符号集合

通过调用 getVT() 获取所有终结符，并调用 getVN() 获取所有非终结符，合并到符号集合sign中

* + - * 1. 初始化项目集和队列

获取文法的开始符号，并遍历所有产生式。如果产生式的左部是开始符号，则创建一个初始项目，并将其符号集初始化为#

调用 Closure() 来计算初始项目集的闭包

如果初始项目集不在itemset中，则将其添加进去，并将其编号入队列deal

* + - * 1. 主循环处理项目集

主循环从队列中取出一个项目集，针对每个符号进行处理：如果它可以成为项目集的下一个符号，则创建新的项目，并将其添加到新的项目集中

调用 Closure() 计算新项目集的闭包，如果新项目集非空且未被处理过，则将其添加到itemset中，并将其编号入队列deal

更新Goto表 GOfuction，记录状态转换信息

最后输出一条信息，标识函数执行完毕

* + 1. 构造预测表
       1. PredictTable\_LR

利用LR(1)文法项集来生成动作表（action table）和Goto表（goto table），并处理各类可能的冲突和错误。

* + - * 1. 初始化和表头设置

检查输入文法是否为LR(1)文法

初始化布尔变量isLLR1

用于记录文法是否为LR(1)。初始值设为 true

初始化产生式

通过调用 setProduction(lr1.getProduction()) 函数，获取并初始化文法的产生式

初始化动作表头

动作表头（actionHeader）包含所有终结符，以及 `#` 符号

初始化Goto表头\*\*：

Goto表头（gotoHeader）包含所有非终结符，除了开始符号

* + - * 1. 初始化表格

每个状态初始化动作表和Goto表

遍历项目集

对于每个项目集状态，函数为动作表和Goto表初始化条目，默认状态为 EMPTY

* + - * 1. 设置接受状态

检查项目集中的项目\*\*：

如果某个项目的点位于末尾，并且产生式的左部是开始符号，则该项目表示可以接受输入。

设置接受状态

对于这样的项目，函数将对应的表项状态设置为 ACCEPT。如果该表项已被设置，则记录冲突，并将 isLLR1设为 false

* + - * 1. 设置移进项

函数根据项目集的转移函数（GOfuction）设置移进操作：

遍历转移函数

对于每个转移函数，函数设置移进操作，状态为 `STATE`，并记录目标状态

检查冲突

如果目标表项已被设置，记录冲突，并将 isLLR1`设为 false

* + - * 1. 设置归约项

函数为每个项目集中的归约项目设置归约操作：

遍历项目集

对于每个项目集中的项目，如果点位于末尾且产生式的左部不是开始符号，则该项目表示可以进行归约操作。

获取产生式编号

函数获取对应产生式的编号。

获取Follow集合

根据产生式的左部符号获取其Follow集合。

更新表项

对于每个Follow集合中的符号，如果表项为空，则设置为归约操作。如果表项已被设置，记录冲突，并将 isLLR1 设为 false

* + - * 1. 错误处理

遍历项目集

函数检查每个项目集中的项目，针对点未在末尾且符号不是非终结符的情况进行处理。

设置特定错误状态

根据不同的符号类型，设置不同的错误状态。例如，对于关系运算符，设置 `ERROR1`；对于条件语句的符号，设置 `ERROR3` 等。

* + 1. 分析
       1. analyse
          1. 将词法分析后的结果转化为语法分析器可以识别的终结符
          2. 开始进行语法分析：

读取一个字符

查询预测表

根据表中内容进行状态转移

若表中状态是STATE，转移

若表中状态是EMPTY，转向当前状态为空对应的状态，若仍为空，报错，停止分析

若表中状态是REVERSE，进行规约

根据action结构体中记录的产生式序号生成新的父节点

根据产生式右侧符号的数量对应字符栈中的字符出栈，作为父节点的子节点，同时状态栈出栈，对应字符栈现有字符数量

同时对产生式中的综合属性（places）和部分产生式的code属性进行赋值、生成

产生式左侧符号进栈、对应goto表中状态进展、

若表中状态是ACCEPT，规约至开始符号，规约完毕

对生成的语法树进行深度优先遍历

向下遍历过程中，对继承属性进行赋值（next、begin等）

分支遍历完成后回到子树根节点时，生成根节点的code

打印生成语法树

1. 三地址代码生成器

• 语法制导定义

|  |  |
| --- | --- |
| **产生式** | **语法制导定义** |
| P' -> P | P’.Code = P.code / gen( P.places ) |
| P -> L | P.Code = L.code / gen( L.places )  P.begin已赋值  P.begin = L.begin |
| P -> L P | P1.Code = L.code / gen( L.places ) || P2.code / gen( P2.places )  P1.begin未赋值  L.next = newlabel();  P2.begin = newlabel()  P1.begin已赋值  P1.begin 归于初始化  L.begin = newlabel() |
| L -> S ; | L.next已赋值  S.next = L.next  L.code += gen( L.begin ”:++”)  S.code有值  L.code += S.code / gen( S.places ) |
| S -> id = E | S.code = gen( S.places “:=”) || E.code / gen( E.places ) |
| S -> if C then S | C.True = newlabel()  C.False = S1.next  S2.next = S1.next  S.code = C.code || gen( C.True “:”) || S2.code |
| S -> if C then S else S | C.True = newlabel()  C.False = newlabel()  C.next = S1.next  S2.next = C.next  S.code = C.code || gen( C.True “:” ) || S.code || gen( C.False “:” ) || S3.code |
| S -> while C do S | S1.begin = newlabel  C.True = S2.next  S1.code = gen( C.begin “:” ) || C.code || S2.Code || gen( “goto” S1.begin ) |
| C -> E > E | C.code = ’if’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| C -> E >= E | C.code = ’if’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| C -> E < E | C.code = ’If’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| C -> E = E | C.code = ’if’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| C -> E <= E | C.code = ’if’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| C -> E <> E | C.code = ’if’+ E1.code / gen( E1.places ) || E2.code / gen( E2.places )  C.code += gen( “goto” C.True ) || gen( “goto” C.False ) |
| E -> T E' | E.code = T.code / gen( T.places ) || E'.code / gen( E'.places ) |
| E' -> + T E' | E.code = T.code / gen( T.places ) || E'.code / gen( E'.places ) |
| E' -> - T E' | E.code = T.code / gen( T.places ) || E'.code / gen( E'.places ) |
| E' -> ~ | E’.code.clear() |
| T -> F T' | T.code = F.code / gen( F.places ) || T'.code / gen( T'.places |
| T' -> \* F T' | T.code = F.code / F.places + T'.code / T'.places |
| T' -> / F T' | T.code = F.code / F.places + T'.code / T'.places |
| T' -> ~ | T’.code.clear() |
| F -> ( E ) | E.code = (.code / (.places + E.code / E.places + ).code / ).places |
| F -> id | F.places = id.places |
| F -> int8 | F.places = int8.places |
| F -> int10 | F.places = int10.places |
| F -> int16 | F.places = int16.places |

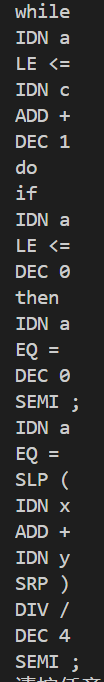
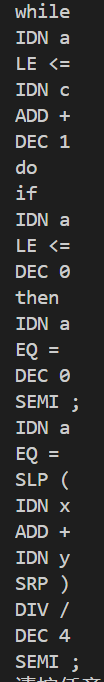
• 算法基本思想

此部分代码包含在analyze函数中。在规约的同时产生新的父节点，同时对父子节点的综合属性值如place按照上表进行处理，最终可以实现语句跳转部分代码正确生成。

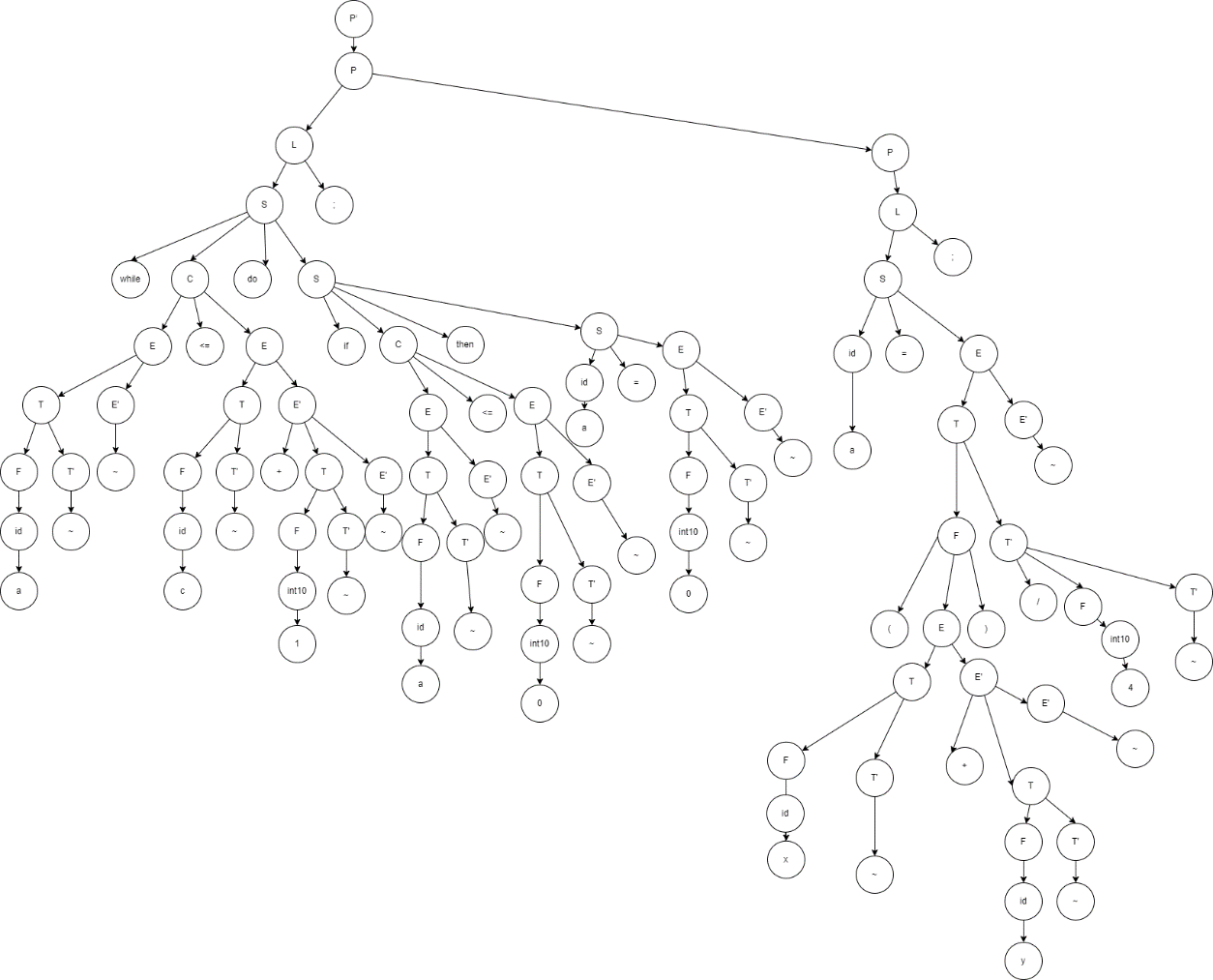
1. 处理实例
   1. 处理字符串

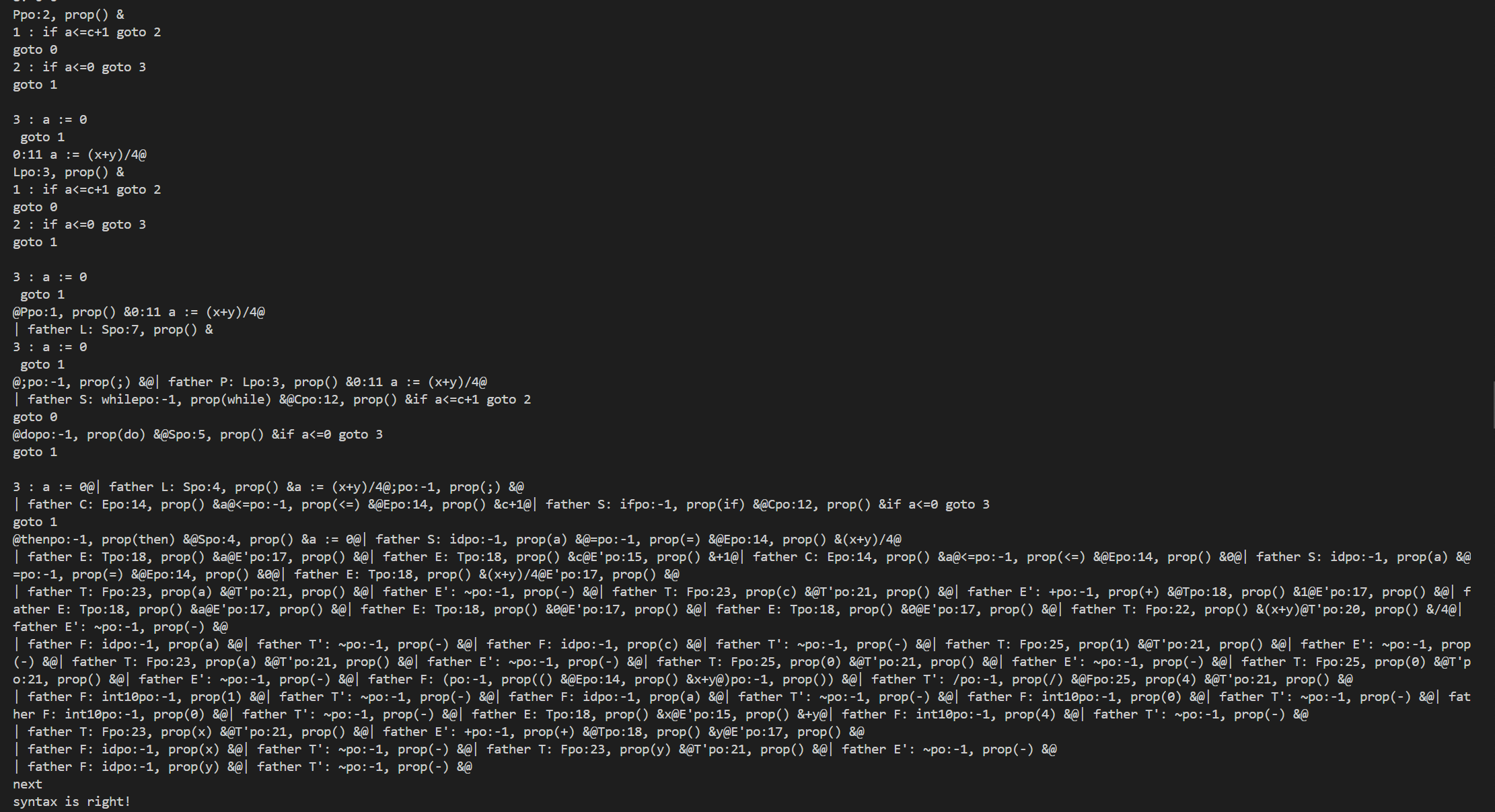


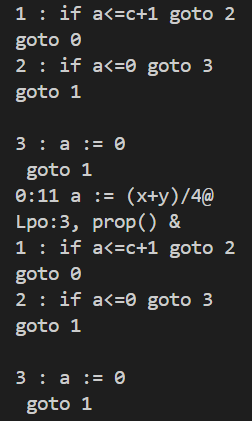
* 1. 词法分析结果



* 1. 语法树
     1. 预测结果



* + 1. 实际生成结果
  1. 三地址代码



1. 实验体会

通过这次实验，我们更深刻的理解了编译器各个部分的作用，也勇敢地应用了之前并不熟悉的git、makefile等工具进行合作开发，在深入研习课堂知识的同时，掌握了很多新技能、新技巧，收获颇丰，为以后的合作开发打下了基础。