**《编译原理》课程实验报告**

专业年级 2019级信息安全

姓　　名 鄢锦琪

学　　号 2019302180149

姓　　名 \

学　　号 \

填写时间 2021 年 5 月 30 日

目录

[1.文法与语义规则描述 3](#_Toc73383302)

[1.1 文法描述 3](#_Toc73383303)

[1.2语义规则描述: 4](#_Toc73383304)

[2.设计与实现 6](#_Toc73383305)

[2.1变量增加int类型和char类型: 6](#_Toc73383306)

[2.2优化tiny中的分号机制: 12](#_Toc73383307)

[2.3变量名的限定 14](#_Toc73383308)

[2.4数字的限定 14](#_Toc73383309)

[2.6增加布尔表达式运算 17](#_Toc73383310)

[3.测试案例与测试结果 21](#_Toc73383311)

[3.1char和int类型不同的读写和运算。 21](#_Toc73383312)

[3.2变量名和数字的限制 22](#_Toc73383313)

[3.3布尔运算 23](#_Toc73383314)

[4.提交内容说明 24](#_Toc73383315)

[4.1 源程序运行环境说明 24](#_Toc73383316)

[4.2 程序运行操作说明 25](#_Toc73383317)

[5.实验小结 25](#_Toc73383318)

[5.1实验分工： 25](#_Toc73383319)

[5.2实验收获： 25](#_Toc73383320)

[5.3实验心得: 26](#_Toc73383321)

# 1.文法与语义规则描述

## 1.1 文法描述

Tiny文法为

program → stmt-sequence

stmt-sequence → stmt-sequence; statement | statement

statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if-stmt → if (exp) then stmt-sequence end | if (exp) then stmt-sequence else stmt-sequence end

repeat-stmt → repeat stmt-sequence until exp

assign-stmt → identifier := exp

read-stmt → read identifier

write-stmt → write exp

exp → simple-exp comparson-op simple-exp | simple-exp

comparison → < | =

simple-exp → simple-exp addop term | term

addop → + | -

term → term mulop factor | factor

mulop → \* | /

factor → (exp) | number | identifier

number → [1-9][0-9]\* | 0

identifier → [a-zA-Z]?

本次实验中实现的tiny+是在tiny的基础上进行修改和优化:

* 变量类型支持char 和 int两种类型
* 优化了tiny中的分号机制
* 变量名中允许出现数字,但需以字母开头
* 数字支持正负且不能以0开头(除0外)
* if判断语句中增加了大于号
* 实现if判断语句中的布尔表达式的运算,

包括与和或,并且与的优先级高于或,且支持括号

根据以上修改和优化得到tiny+的文法如下

program → declaration-list; stmt-sequence

declaration-list → declaration-list declaration; | declaration;

declaration → type-specifier identifier

type-specifier → int | char

stmt-sequence → stmt-sequence statement; | statement;

statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if-stmt → if (exp) then stmt-sequence end | if (exp) then stmt-sequence else stmt-sequence end

repeat-stmt → repeat stmt-sequence until exp

assign-stmt → identifier := exp

read-stmt → read identifier

write-stmt → write exp

exp → simple-exp comparson-op simple-exp | simple-exp

comparison → < | = | >

simple-exp → simple-exp addop term | term

addop → + | -

term → term mulop factor | factor

mulop → \* | /

factor → (exp) | number | identifier

number → (+|-)?([1-9][0-9]\* | 0)

identifier → [a-zA-Z]([0-9]| [a-zA-Z])\*

bool-first-exp → exp | ( bool-exp )

and-exp → and-exp and bool-first-exp | bool-first-exp

or-exp → or-exp or and-exp | and-exp

bool-exp → or-exp

## 1.2语义规则描述:

Tiny+的语义规则如下(因版面问题与上方文法一一对应)

Tiny+语义规则

assign-stmt → identifier := exp

id.val = exp.val

exp → simple-exp comparson-op simple-exp

exp.val = sim1.val (<,>,=) sim2.val

simple-exp → simple-exp addop term

sim1.val = sim2.val + term.val

term → term mulop factor

term1.val = term2.val \* factor.val

write-stmt → write exp

print(exp.val)

declaration -> type-specifier identifier

id.type = type-specifier

exp -> simple-exp comparison simple-exp

exp.type = Boolean

simple-exp1 -> simple-exp2 addop term

sim1.type = 一个为Integer, 一个为Character ----> 报错

sim2.type==Integer||term.type==Integer ----> Integer

sim2.type==Character||term.type==Character ----> Character

都为Constant ----> Constant

term1 -> term2 mulop factor

term1.type = 一个为Integer, 一个为Character ----> 报错

term2.type==Integer||factor.type==Integer ----> Integer

term2.type==Character||factor.type==Character ----> Character

都为Constant ----> Constant

factor -> (exp)

factor.type = exp.type

factor -> number

factor.type = Constant

factor -> identifier

factor.type = id.type

and\_exp -> and\_exp and bool-first-exp

and\_exp.val = and\_exp1.val and bool-first-exp.val

and\_exp -> bool\_first\_exp

and\_exp.val -> bool\_first\_exp.val

or\_exp -> or\_exp or and\_exp

or\_exp.val = or\_exp2.val or and\_exp.val

# 2.设计与实现

## 2.1变量增加int类型和char类型:

(1)词法分析(在globals.h和scan.c中和util.c中):

增加int和char类型首先需要在scan中能够扫描得到token,首先在globals.h中修改最大保留字个数,也将其加进TokenType的枚举体中,同时也需要在ArgumentKind和ExpType中增加Char类型。涉及改动的代码如下。

#define MAXRESERVED 10

**typedef** enum

/\* book-keeping tokens \*/

**{**

/\* argument type \*/

CHAR**,**INT**,**

**}** TokenType**;**

**typedef** enum **{**CHAR**,**INT**}** ArgumentKind**;**

**typedef** enum **{**Void**,**Integer**,**Boolean**,**Char**}** ExpType**;**

而在scan.c中同样需要修改reservedWord表中的对应关系

static struct

**{** char**\*** str**;**

TokenType tok**;**

**}** reservedWords**[**MAXRESERVED**]**

**=** **{{**"if"**,**IF**},{**"then"**,**THEN**},{**"else"**,**ELSE**},{**"end"**,**END**},**

**{**"repeat"**,**REPEAT**},{**"until"**,**UNTIL**},{**"read"**,**READ**},**

**{**"write"**,**WRITE**},{**"int"**,**INT**},{**"char"**,**CHAR**}};**

在util.c中的printTree和printToken函数中也要添加int和char相关的内容,以便输出正确的语法树和token

// in printToken (TokenType token, const char\* tokenString )

**case** CHAR**:**

**case** INT**:**

**case** WRITE**:** fprintf**(**listing**,** "reserved word: %s\n"**,**tokenString**);** **break;**

// in printTree(TreeNode \* tree )

**case** CharK**:**

fprintf**(**listing**,**"Char: %s\n"**,**tree**->**attr**.**name**);**

**break;**

**case** IntK**:**

fprintf**(**listing**,**"Int: %s\n"**,**tree**->**attr**.**name**);**

**break;**

(2)语法分析(在parse.c中)

因为分为char和int两种类型,在parse中生成结点时应该有所区别。并且因为declaration是作为一种和statement类似的语句出现在代码中，所以按照文法需要增加declaration函数和declaration\_list函数，这两个函数用于生成声明变量的语法树结点，并且将其连接。仿照statement和statement\_sequence函数即可。

TreeNode **\***declaration**(**void**){**

TreeNode **\*** t **=** **NULL;**

**switch(**token**){**

**case** INT**:**

t **=** newStmtNode**(**IntK**)** **;**

match**(**INT**);**

t**->**attr**.**name**=** copyString**(**tokenString**);**

match**(**ID**);**

**break;**

**case** CHAR**:**

t **=** newStmtNode**(**CharK**);**

match**(**CHAR**);**

t**->**attr**.**name**=** copyString**(**tokenString**);**

match**(**ID**);**

**break;**

**default:**

**break;**

**}**

**return** t**;**

**}**

TreeNode **\***declaration\_list**(**void**){**

TreeNode **\*** t **=** declaration**();**

TreeNode **\*** p **=** t**;**

**while(**token **==** SEMI**){**

TreeNode **\***q**;**

match**(**SEMI**);**

q **=** declaration**();**

**if(**q**!=NULL){**

**if(**t **==** **NULL)** **{**

t **=** p **=** q**;**

**}** **else** **{**

p**->**sibling **=** q**;**

p **=** q**;**

**}**

**}**

**}**

**return** t**;**

**}**

因为增加了declaration\_list函数,此处的parse函数原本是相当于program,所以此处修改,parse的内容.将program\_sequence中的program以sibling的形式连接到declaration-list的后面.因为两者的执行形式是一样的,赋值语句实际上也就是一个statement

TreeNode **\*** parse**(**void**)**

**{** TreeNode **\*** t**;**

token **=** getToken**(**token**);**

t **=** declaration\_list**();**

**if(**t**!=NULL){** // here regard declaration stmt and stmt-sequence as the same process

TreeNode**\*** p **=** t**;**

**while(**p**->**sibling**!=NULL)** p **=** p**->**sibling**;**

p**->**sibling **=** stmt\_sequence**();**

**}** **else** **{**

t **=** stmt\_sequence**();**

**}**

（3）语义分析（在analyze.c中和symtab.c中）

需要根据parse得到的树结构进行结点type分析和插入结点等东西和构建符号表，那么就需要在analyze中的insertNode函数中添加生成intk和chark结点的代码。而因为加入了int和char类型，需要对生成的结点进行类型上的分别，所以修改了st\_insert函数，新增了添加类型的操作

void st\_insert**(** char **\*** name**,** int lineno**,** int loc**,** TokenType type**)**

**{** int h **=** hash**(**name**);**

BucketList l **=** hashTable**[**h**];**

**while** **((**l **!=** **NULL)** **&&** **(**strcmp**(**name**,**l**->**name**)** **!=** 0**))**

l**->**lines **=** **(**LineList**)** malloc**(sizeof(**struct LineListRec**));**

l**->**lines**->**lineno **=** lineno**;**

l**->**memloc **=** loc**;**

l**->**type **=** type**;**

l**->**lines**->**next **=** **NULL;**

l**->**next **=** hashTable**[**h**];**

hashTable**[**h**]** **=** l**;** **}**

在symtab中也模拟st\_lookup写一个get\_type函数用于获取某个变量名的类型（获取是char还是int）。

int get\_type **(**char **\*** name**)**

**{** int h **=** hash**(**name**);**

BucketList l **=** hashTable**[**h**];**

**while** **((**l **!=** **NULL)** **&&** **(**strcmp**(**name**,**l**->**name**)** **!=** 0**))**

l **=** l**->**next**;**

**if** **(**l **==** **NULL)** **return** **-**1**;**

**else**

**return** l**->**type**;**

**}**

然后再insertNode中添加char和int相关的类型,并且因为修改了st\_insert中的参数,所以需要修改所有出现的传参.

static void insertNode**(** TreeNode **\*** t**)**

**{**

**switch** **(**t**->**nodekind**)**

**{** **case** StmtK**:**

**switch** **(**t**->**kind**.**stmt**)**

**{**

**case** IntK**:**

**if(**st\_lookup**(**t**->**attr**.**name**)** **==** **-**1**)**

st\_insert**(**t**->**attr**.**name**,**t**->**lineno**,**location**++,**INT**);**

**else**

nameError**(**t**,**"redeclaration of identifier"**);**

**break;**

**case** CharK**:**

**if(**st\_lookup**(**t**->**attr**.**name**)** **==** **-**1**)**

st\_insert**(**t**->**attr**.**name**,**t**->**lineno**,**location**++,**CHAR**);**

**else**

nameError**(**t**,**"redeclaration of identifier"**);**

**break;**

………

(4)中间代码生成(cgen.c中 )

因为有了char类型和int 类型,所以两者的读入和输出需要有所区别,int类型仍然使用之前的读入读出,而char类型使用新的inc和outc来进行读入读出。所以需要在cgen.c中生成inc指令和outc指令。

对case readk和case writek进行修改，让他们能区分int和char。

**case** ReadK**:**

temp **=** get\_type**(**tree**->**attr**.**name**);**

**if(** temp **==** INT**){**

emitRO**(**"IN"**,**ac**,**0**,**0**,**"read integer value"**);**

**}** **else** **if(**temp**==**CHAR**){**

emitRO**(**"INC"**,**ac**,**0**,**0**,**"read integer value"**);**

**}**

**case** WriteK**:**

/\* generate code for expression to write \*/

cGen**(**tree**->**child**[**0**]);**

temp **=** get\_type**(**tree**->**child**[**0**]->**attr**.**name**);**

/\* now output it \*/

**if(**temp **==** INT**){**

emitRO**(**"OUT"**,**ac**,**0**,**0**,**"write ac"**);**

**}** **else{**

emitRO**(**"OUTC"**,**ac**,**0**,**0**,**"write ac"**);**

**}**

**break;**

（5）运行环境（在tm.c中）

为了能够识别inc和outc，需要在声明处增加opinc和opoutc，，opcodetab中也增加inc和outc。读取到相关的指令时需要能够执行，故在steptm函数中需要增加对case inc和case outc的操作。

**case** opINC **:**

printf**(**"Enter value for INC instruction: "**)** **;**

fflush **(**stdin**);**

fflush **(**stdout**);**

temp **=** getchar**();**

reg**[**r**]** **=** **(**int**)** temp**;**

**break;**

**case** opOUTC **:**

printf **(**"OUT instruction prints(char): %c\n"**,** reg**[**r**])** **;**

**break;**

## 2.2优化tiny中的分号机制:

在原先的tiny文法中，下面的文法导致一个stmt-sequence中，最后一个句子是不能有分号的，如果存在分号则会在词法分析时出现错误。

program → stmt-sequence

stmt-sequence → stmt-sequence; statement | statement

这就导致了在制作测试数据和debug时存在很大不方便，因为在一个代码块（stmt-sequence）中，只有一条语句时需要分号，两条语句以上时，最后一个不能有分号等等，此规则非常不合理，于是经过思考将其修改为每个语句都需要有分号。将文法按照如下进行修改。

program → stmt-sequence

stmt-sequence → stmt-sequence statement； | statement；

修改后每个句子就都需要有分号了

根据文法的改变，因为tiny中使用的是递归下降子程序法，所以去parse.c中修改结点的构建。修改过程如下，只需修改SEMI（分号）的匹配位置即可。



declaration和declaration-list中采用和statement与statement-sequence相同的操作即可。

修改后的代码如下

TreeNode **\*** stmt\_sequence**(**void**)**

**{** TreeNode **\*** t **=** statement**();**

match**(**SEMI**);**

TreeNode **\*** p **=** t**;**

**while** **((**token**!=**ENDFILE**)** **&&** **(**token**!=**END**)** **&&**

**(**token**!=**ELSE**)** **&&** **(**token**!=**UNTIL**))**

**{** TreeNode **\*** q**;**

q **=** statement**();**

**if** **(**q**!=NULL)** **{**

match**(**SEMI**);**

**if** **(**t**==NULL)** t **=** p **=** q**;**

**else** /\* now p cannot be NULL either \*/

**{** p**->**sibling **=** q**;**

p **=** q**;**

**}**

**}**

**}**

**return** t**;**

**}**

## 2.3变量名的限定

新的文法中限定了变量名中可以出现数字，并且需要以字母开头。显然这部分操作需要修改 词法分析器，在scan.c中只需修改INID的退出条件即可。

**case** INID**:**

**if** **(!**isalpha**(**c**)&&!**isdigit**(**c**))**

**{** /\* backup in the input \*/

ungetNextChar**();**

save **=** FALSE**;**

因为在词法分析器中匹配到字母以后会进入到INID状态，而源代码中通过不断获取下一个char，判断如果不是字母（isalpha()判断）就表示这个ID名称结束。而此时只需要修改此处的出口条件，让它匹配到数字时不退出，非字母非数字时再退出即可。

## 2.4数字的限定

（1）数字不能以0开头（除0外）

因为词法分析的过程和结果都记录在tokenstring中，所以如果tokenstring[0]是

并且tokenStringIndex是2时，说明此时已经在读入其他数字，那么就应当报错

代码修改后如下

**if(**tokenString**[**0**]** **==** '0' **&&** tokenStringIndex **>** 1**){**

ungetNextChar**();**

save **=** FALSE**;**

currentToken **=** ERROR**;**

**}**

**break;**

（2）数字前有正负号

判断一个数字前的符号是正号还是加号，是减号还是负号，需要读前面的token，而在原来的词法分析器中并没有记录前一个token的功能，所以为其新增一个参数preToken,此时若preToken是ID或者是NUM，那么这个+号或者-号就是运算符号，否则就是表示正负。

所以在词法分析器读取到+号和-号时添加一个判断，如果前一个是NUM或者ID，那么就进入到PLUS或者MINUS状态，否则是进入INNUM状态。

修改后的代码如下。

**case** '+'**:**

**if(**preToken**==**NUM **||** preToken**==**ID**){**

currentToken **=** PLUS**;**

**}** **else{**

state **=** INNUM**;**

**}**

**break;**

**case** '-'**:**

**if(**preToken**==**NUM **||** preToken**==**ID**){**

currentToken **=** MINUS**;**

**}** **else{**

state **=** INNUM**;**

**}**

**break;**

2.5增加大于号

增加大于号相对比较简单，步骤如同增加char类型。参照原代码中对小于号的操作，在其后追加相应的大于号相关代码即可。

首先在globals.h中增加表示大于号的token,此处用GT

在scan.c中添加对大于号的识别.

**case** '>'**:**

currentToken **=** GT**;**

**break;**

在parse.c中增加对GT类型的结点的生成,也就是在原小于号判断的后面增加一个大于号判断,下面加红处.

TreeNode **\*** exp**(**void**)**

**{** TreeNode **\*** t **=** simple\_exp**();**

**if** **((**token**==**LT**)||(**token**==**EQ**)** **||**(token == GT)**)** **{**

TreeNode **\*** p **=** newExpNode**(**OpK**);**

**if** **(**p**!=NULL)** **{**

p**->**child**[**0**]** **=** t**;**

p**->**attr**.**op **=** token**;**

t **=** p**;**

**}**

match**(**token**);**

**if** **(**t**!=NULL)**

t**->**child**[**1**]** **=** simple\_exp**();**

**}**

**return** t**;**

**}**

在analyze.c中的结点类型检测中增加识别,同时也为了能够让char类型进行比较,所以也需要修改一下checknode中对比较式的修改.

static void checkNode**(**TreeNode **\*** t**)**

**{** **switch** **(**t**->**nodekind**)**

**{** **case** ExpK**:**

**switch** **(**t**->**kind**.**exp**)**

**{** **case** OpK**:**

**if** **((**t**->**child**[**0**]->**type **==** Boolean**)** **||**

**(**t**->**child**[**1**]->**type **==** Boolean**))**

typeError**(**t**,**"Op applied to non-integer and non-char"**);**

**if** **((**t**->**attr**.**op **==** EQ**)** **||** **(**t**->**attr**.**op **==** LT**)** **||** **(**t**->**attr**.**op **==** GT**))**

t**->**type **=** Boolean**;**

**else**

t**->**type **=** Integer**;**

**break;**

在util.c中,打印token和打印树时需要

case GT: fprintf(listing,">\n"); break;

在Cgen.c中按照LT的操作,修改相应寄存器位置,即可实现大于

**case** GT**:**

emitRO**(**"SUB"**,**ac**,**ac**,**ac1**,**"op >"**)** **;**

emitRM**(**"JLT"**,**ac**,**2**,**pc**,**"br if true"**)** **;**

emitRM**(**"LDC"**,**ac**,**0**,**ac**,**"false case"**)** **;**

emitRM**(**"LDA"**,**pc**,**1**,**pc**,**"unconditional jmp"**)** **;**

emitRM**(**"LDC"**,**ac**,**1**,**ac**,**"true case"**)** **;**

**break;**

## 2.6增加布尔表达式运算

原本的tiny中的if语句的条件和repeat语句中until的条件,都只能是一个单语句,不能实现与和或的操作,所以想要实现一个条件语句中的布尔运算.

新增功能首先需要有token,选取and 作为与操作 ,or作为或操作.对应的Token分别是AND和OR

首先在globals.h中修改Token总数,并且在Token中添加OR和AND,算式类型EXPkind中也添加布尔类型Boolk

然后在scan.c中的reservedWord中添加and和or与其Token的绑定.

再是在util.c中添加对boolk语句的输出,和and 与or的token的输出.

以上内容都非常简单,并且已经在上面有了类似的代码展示,此处就不再贴代码.

因为布尔表达式的运算中,与和或的优先级是不同的,与的优先级高于或的优先级,并且有优先级的高低就需要考虑是否可以存在括号改变原有的运算顺序,所以根据这些需要写出了如下文法.

bool-first-exp → exp | ( bool-exp )

and-exp → and-exp and bool-first-exp | bool-first-exp

or-exp → or-exp or and-exp | and-exp

bool-exp → or-exp

根据这个文法则需要在parse.c中更改树的生成方式.

编写bool\_first\_exp,and\_exp(),bool\_exp(),模拟factor,term,simple\_exp的操作

TreeNode **\*** bool\_exp**(**void**){**

TreeNode **\***t **=** bool\_exp\_and**();**

**while(**token **==** OR**){**

TreeNode **\*** p **=** newExpNode**(**BoolK**);**

**if(**p**!=NULL){**

p**->**child**[**0**]** **=** t**;**

p**->**attr**.**op **=** token**;**

t **=** p**;**

match**(**token**);**

t**->**child**[**1**]** **=** bool\_exp\_and**();**

**}**

**}**

**return** t**;**

**}**

TreeNode **\*** bool\_exp\_and**(**void**){**

TreeNode **\*** t **=** bool\_exp\_first**();**

**while(**token **==** AND**){**

TreeNode **\***p **=** newExpNode**(**BoolK**);**

**if(**p**!=NULL){**

p**->**child**[**0**]** **=** t**;**

p**->**attr**.**op **=** token**;**

t **=** p**;**

match**(**token**);**

t**->**child**[**1**]** **=** bool\_exp\_first**();**

**}**

**}**

**return** t**;**

**}**

TreeNode **\*** bool\_exp\_first**(**void**){**

TreeNode **\*** t **=** **NULL;**

**if** **(**token **==** LPAREN**){**

match**(**LPAREN**);**

t **=** bool\_exp**();**

match**(**RPAREN**);**

**}** **else** **{**

t **=** exp**();**

**}**

**return** t**;**

**}**

因为修改了生成树的方式,所以需要考虑analyze.c中的对结点检查,因为遇到的结点可以是boolk类型的多条布尔语句,也可以是一条语句,所以需要修改相应的判断条件.

**case** StmtK**:**

**switch** **(**t**->**kind**.**stmt**)**

**{** **case** IfK**:**

**if** **(**t**->**child**[**0**]->**kind**.**exp **!=** BoolK **&&** t**->**child**[**0**]->**type **!=** Boolean**)**

typeError**(**t**->**child**[**0**],**"if test is not Boolean"**);**

**break;**

因为新建的树结点类型是boolk类型,所以在生成中间代码时,读取到此处时要能够正确生成中间代码.原本的genexp中是没有对boolk的类型的修改的.所以需要添加case BoolK,并且or和and 本质上是将原先的if语句中的exp的结果进行某些操作得到新的判断是否跳转的中间值.所以要尽可能的利用原本的对exp的运算.

在原本的ifk中,是递归的处理三个子结点,然后再opk中的LT,GT,EQ中进行布尔的运算,拿出一个LT的进行分析

**case** LT **:**

emitRO**(**"SUB"**,**ac**,**ac1**,**ac**,**"op <"**)** **;**

emitRM**(**"JLT"**,**ac**,**2**,**pc**,**"br if true"**)** **;**

emitRM**(**"LDC"**,**ac**,**0**,**ac**,**"false case"**)** **;**

emitRM**(**"LDA"**,**pc**,**1**,**pc**,**"unconditional jmp"**)** **;**

emitRM**(**"LDC"**,**ac**,**1**,**ac**,**"true case"**)** **;**

**break;**

分析汇编代码以后可以知道,其实最终就是满足条件时,将reg[ac]置为1,不满足条件时,将reg[ac]置为0

而在ifk中会有相关的语句判断,如果是1则不跳转,是0则跳转

我们要实现布尔表达式的运算,其实就是对两个或多个语句生成的reg[ac]进行处理

所以我新选用一个寄存器,reg[ac2]用于暂时存储临时结果,并且通过汇编语句,当是AND时,当两个都为1时将reg[ac]置1,否则为0,当是OR时,当两个都为0时置0,否则置1

最终实现的汇编代码如下

**case** BoolK**:**

**if(**TraceCode**)** emitComment**(**"-> Bool"**);**

p1 **=** tree**->**child**[**0**];**

p2 **=** tree**->**child**[**1**];**

cGen**(**p1**);**

emitRM**(**"LDA"**,**ac2**,**0**,**ac**,**"store the first bool value to reg[ac2]"**);**

cGen**(**p2**);**

emitRO**(**"ADD"**,**ac2**,**ac**,**ac2**,**"add two bool results to reg[ac2]"**);**

**switch(**tree**->**attr**.**op**){**

**case** OR**:**

emitRM**(**"LDA"**,**ac**,**0**,**ac2**,**"put reg[ac2] into reg[ac]"**);**

**break;**

**case** AND**:**

emitRM**(**"LDC"**,**ac**,**2**,**ac**,**"set reg[ac] to 2 to sub reg[ac2] "**);**

emitRO**(**"SUB"**,**ac**,**ac2**,**ac**,**"ac2 from 2 to 0 then and is true"**);**

emitRM**(**"JNE"**,**ac**,**2**,**pc**,**"if ac is zero ,then make it not zero"**);**

emitRM**(**"LDC"**,**ac**,**1**,**0**,**"make reg[ac] non-zero"**);**

emitRM**(**"LDA"**,**pc**,**1**,**pc**,**"unconditional jmp"**)** **;**

emitRM**(**"LDC"**,**ac**,**0**,**0**,**"make reg[ac] zero"**);**

**break;**

**default:**printf**(**"op is error"**);**

**}**

因为在建树的时候是递归调用,在此处ifk语句和boolk语句也都是递归调用,所以表面上是处理了两个布尔表达式的操作,其实是多个布尔表达式的操作都能实现了

# 3.测试案例与测试结果

测试样例已经全部存储在了TestCases文件夹中

## 3.1char和int类型不同的读写和运算。

{ 这个文件用于测试 char和 int两种类型的读写}

char x;

int y;

read x;

read y;

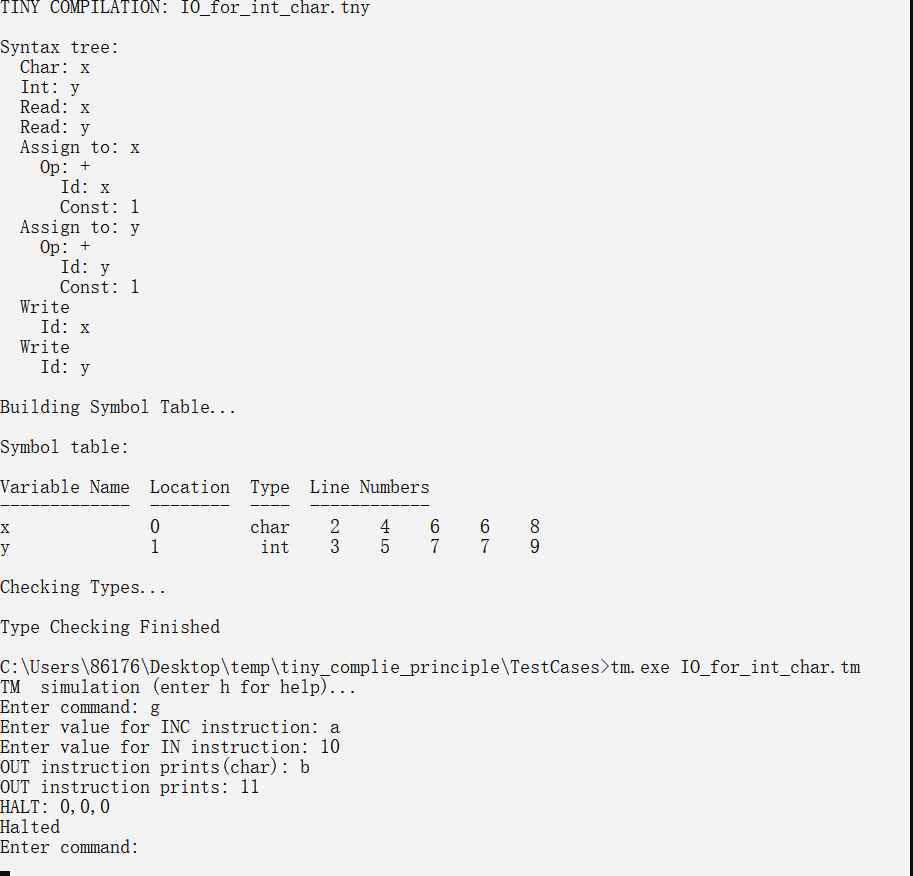
x := x+1;

y := y+1;

write x;

write y;

运行结果如下



## 3.2变量名和数字的限制

{这个文件用于测试变量名可以带有数字, 变量赋值数字可以正负}

int x1;

x1 := -1;

write x1;

x1 := +1;

write x1;

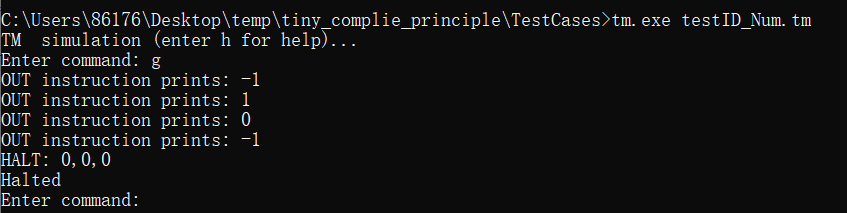
x1 := 0;

write x1;

x1 :=x1+-1;

write x1;

此处已经说明变量名可以含数字，以及赋值可以是0

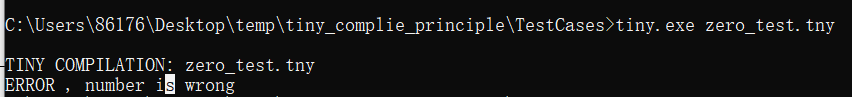


{ 测试可以是0但是不能是以0开头的其他数字 }

int x1;

x1 := 011231;

write x1;



遇到0开头的数字时会进行报错

## 3.3布尔运算

{测试if和until后的布尔运算}

int x1;

int x2;

char x3;

x1 := 0;

x2 := 1;

x3 := 67;

if (x1>0 or x2>0) and x1<10 then

x3 := x3+1;

write x3;

else

x3 := x3+2;

write x3;

end;

x1 := 10;

x2 := 0;

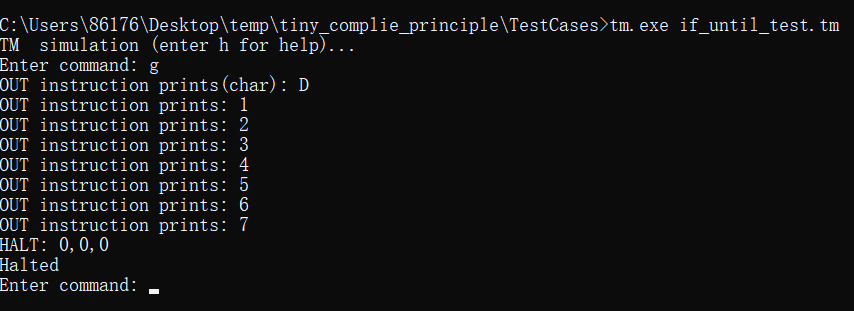
repeat

x1 := x1-1;

x2 := x2+1;

write x2;

until x1<5 and x2 >6;



上面的代码测试了在if语句的条件和repeat语句的条件中的布尔运算。

# 4.提交内容说明

## 4.1 源程序运行环境说明

运行环境：windows 10专业版 ， dev c++;

运行说明：

文件中已经包含了dev的项目文件和编译所需makefile，也同时已经包含了编译得到的tm.exe和tiny.exe文件。

如需重新编译则需要使用dev c++打开两个.dev文件，分别编译即可得到相应exe文件

也可以在代码中修改原有的编译开关获得不同的效果。

## 4.2 程序运行操作说明

在cmd中载入相关文件（exe文件）所在位置

首先使用tiny.exe 后接tny文件， 如tiny.exe SAMPLES.tny

然后使用tm.exe后接tm文件，如tm.exe SAMPLES.tm 即可进入tm机

# 5.实验小结

## 5.1实验分工：

本次实验内容均由2019302180149鄢锦琪一人独立完成。

## 5.2实验收获：

(1)巩固了理论知识，理论实践相结合。在学习理论课时，只是知道编译一个语言需要先词法分析，再语法分析，语义分析等等，但是并不能够将这些所学的想关联起来，知识体系相对比较分割，所以不能正确理解编译器的真正工作原理，而在本次作业中，因为需要向原有的语言中增加新的特性，那就需要从头到尾让一个编译器去适配这个语言，就需要切实了解编译器各部分的操作以及相互作用。因为有理论知识的欠缺，所以需要重新回顾课本上的知识点等等，切实的巩固

了理论知识，将理论和实践相结合也加深了对理论知识的理解。

(2)提高了动手能力和阅读源代码的能力。刚拿到这份作业时，其实是几乎没有头绪的，而编译原理与实践的书本中内容又太过丰富，没有时间去详细阅读，只能硬着头皮直接看源代码， 而源代码中的注释内容并不详细，需要不断的回想知识点来理解代码。代码的体系比较庞大，非常多的文件，各种函数之间的相互调用等等，都需要较强的阅读源代码的能力。

## 5.3实验心得:

在本次实验中，不得不说，遇到了非常多的问题，最开始，单单是想要重新编译出tiny.exe和tm.exe就花费了很大力气，对原先makefile的修改，遇到各种奇怪的错误，在linux中缺少make所需的库，在codeblock中编译出现错误等等。后来发现其实是因为没有熟悉项目的结构，tm.c和其他文件应该在两个不同的项目中，所以直接文件放错了位置，也就导致了相关的编译不能成功。最终明白了项目结构以后成功编译出了tiny.exe和tm.exe，走出了第一步。

初看代码时只能说是一脸懵逼，对于庞大的体系完全不知道该如何下手，后来反复的阅读推敲才逐渐渐入佳境。花费时间最久的两个地方分别是让数字能够正负两种和增加布尔表达式的运算。听起来似乎前者没有什么难度，但是因为对scan.c中的getToken函数的理解不够准以为它是在循环中反复获取token的，其实不是，而是在parse中调用getToken函数。因为以为是循环反复获取token，那么currentToken在修改之前就应该是上一个token，于是用于判断加号到底是正号还是运算符号，可是在词法分析中没有一点问题，但是在打开了parse的开关以后，currentToken竟然有时会达到40000多，后来得知其实这块内存已经被释放，所以可能被其他程序更改了，此处debug非常多时间，后来才得知是对程序的运行机理不够熟悉。而在增加布尔表达式时，因为没有细看汇编代码，所以对cGen中的代码非常不熟悉，也一度十分抗拒，但是即使一开始完全没看明白，反复看，反复揣摩，量变引起质变，最终成功添加了布尔表达式的运算。

学到两点，遇到不会的东西不要怕，硬着头皮上去，时间堆的多了，就容易恍然大悟。再就是程序出现的错误有时其实并不是浮现在程序表面的，需要从一个更大的格局去看待。