|  |  |  |
| --- | --- | --- |
| 南农大  **编译原理课程设计**  XH2 | | |
|  | 题 目: | 词法、语法、语义分析 |
|  | 姓 名: | 曾凡 |
|  | 学 号: | 19218228 |
|  | 学 院: | 信息科学技术学院 |
|  | 专 业： | 计算机科学与技术 |
|  | 班 级: | 计科182 |
|  | 指导教师: | 黄 芬 |
|  | 成绩评定 | : |
| 2020年12月26日 | | |

目录

[**1** 目的 4](#_Toc59933686)

[**2** 设计思路与实现 4](#_Toc59933687)

[**2.1** 编译过程概述 4](#_Toc59933688)

[**2.2** Flex与Bison工具的简介 5](#_Toc59933689)

[**2.2.1** Flex文件内容结构 5](#_Toc59933690)

[**2.2.2** Bison文件内容结构 6](#_Toc59933691)

[**2.2.3** Flex-Bison协作关系 6](#_Toc59933692)

[**2.2.4** 结构体含义 7](#_Toc59933693)

[**2.3** 文法的设计 7](#_Toc59933694)

[**3** 程序实现 8](#_Toc59933695)

[**3.1** 程序总体思想 8](#_Toc59933696)

[**3.2** 词法分析设计 8](#_Toc59933697)

[**3.3** 语法分析设计 9](#_Toc59933698)

[**3.4** 抽象语法树的生成 12](#_Toc59933699)

[**3.5** AST图片的设计 13](#_Toc59933700)

[**3.6** MakeFile文件的设计 14](#_Toc59933701)

[**4** 测试用例以及实验结果 15](#_Toc59933702)

[**4.1** 文件测试用例 15](#_Toc59933703)

[**4.2** 使用说明 15](#_Toc59933704)

[**4.3** 文件测试 15](#_Toc59933705)

[**4.3.1** 文件1测试 15](#_Toc59933706)

[**4.3.2** 文件2测试 18](#_Toc59933707)

[**4.3.3** 文件3测试 20](#_Toc59933708)

[**5** 技术问题以及解决方案 22](#_Toc59933709)

[**5.1** 技术问题 22](#_Toc59933710)

[**5.2** 心得体会 23](#_Toc59933711)

[**6** 备注 23](#_Toc59933712)

编译原理设计报告

曾凡

南京农业大学

日期:2020年12月26日

# 目的

采用词法分析工具Flex自动构建分析工具，对输入流的程序进行分析，开发了文法描述语言的词法分析程序，实现词法分析过程中对单个单词的分析，加深对课本词法分析内容的理解，提高自身的实践能力。

采用了SLR(1)语法分析方法，利用Bison工具实现对语法分析。利用给的文法开发如下语言描述分析程序，实现对单个词语序列的语法分析和结构分析。

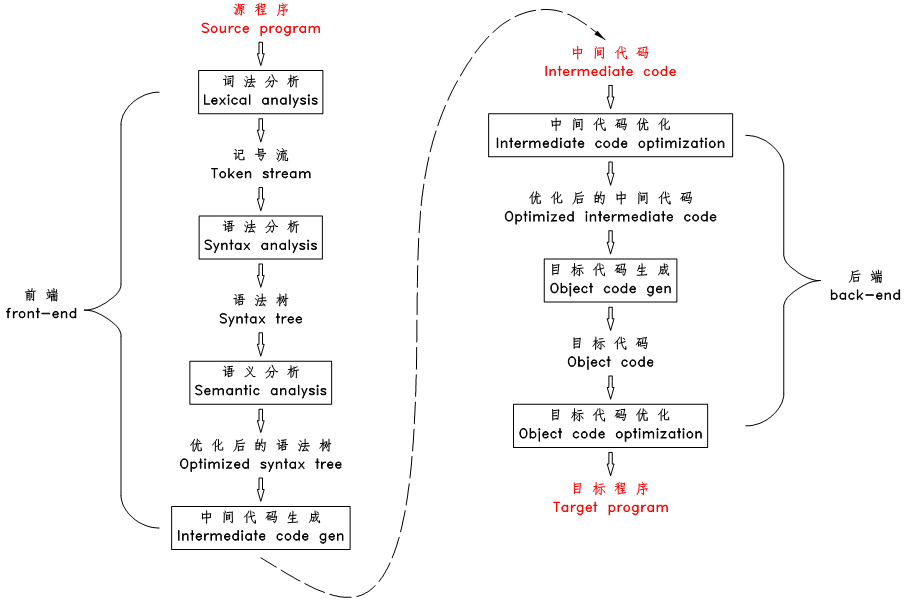
编译原理课程中学到的知识，学习程序设计语言编译程序的一般原理、基本设计方法、主要实现技术，加深对词法分析、语法分析的理解和认识，通过编程给出具体的实现，进而掌握词法分析和语法分析的基本思想。

# 设计思路与实现

## 编译过程概述

编译程序完成从源程序到目标程序的翻译工作，是一个复杂的整体的过程。从概念上来讲，一个编译程序的整个工作过程是划分成阶段进行的，每个阶段将源程序的一种表示形式转换成另一种表示形式，各个阶段进行的操作在逻辑上是紧密连接在一起的。一般一个编译过程划分成词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成六个阶段。

上述编译过程的六个阶段的任务，可以分别由六个模块完成，它们称作词法分析程序、语法分析程序、语义分析程序、中间代码生成程序、代码优化程序和目标代码生成程序。



编译的6个阶段

## Flex与Bison工具的简介

Flex和bison是两个在Linuxx下用来生成程序的工具，它们生成的程序分别叫做词法分析器和语法分析器，工作中主要是用来生成SQL语句的词法和语法分析器。

### Flex文件内容结构

/\* P1: declarations(定义段) \*/

%{

%}

%%

/\* P2: translation rules(规则段) \*/

%%

/\* P3: auxiliary functions(用户辅助程序段，c函数)\*/

### Bison文件内容结构

/\*P1: declarations 定义段\*/

%{

%}

%%

/\*P2: grammar rules 规则段(rule-action)\*/

A: a1 { 语义动作1 }

| a2 { 语义动作2 }

| …

| an { 语义动作n }

| b //没有{…},则使用缺省的语义动作

; //产生式结束标记

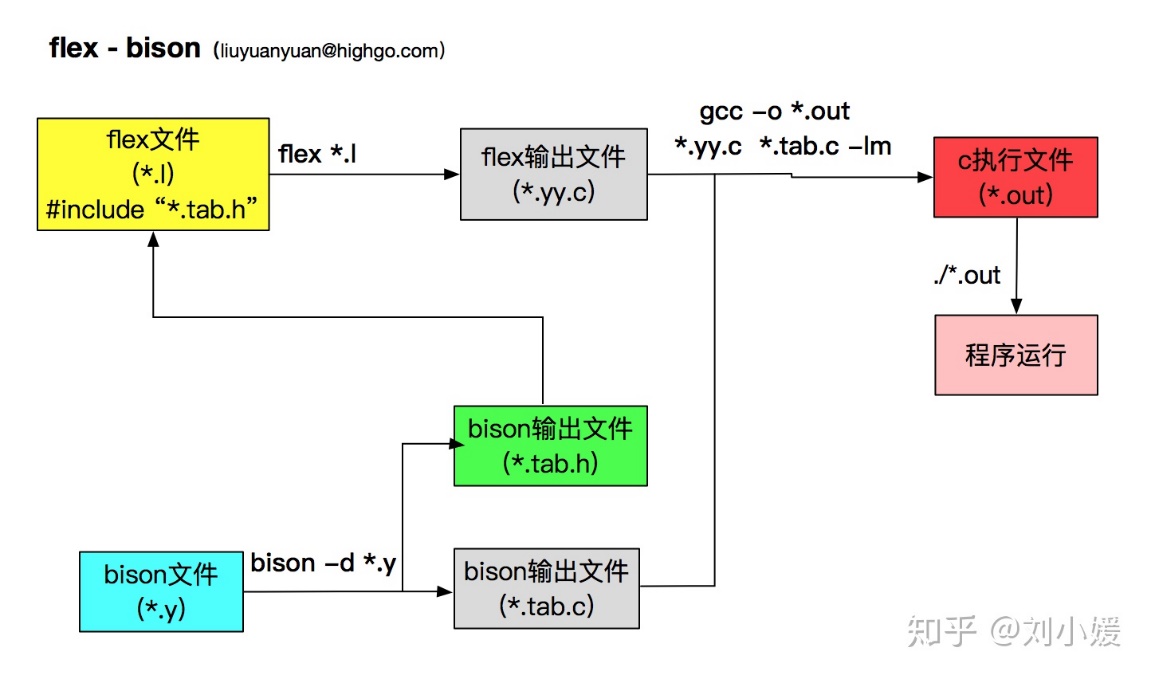
//语义动作一般是在产生式右部分析完，归约动作进行前执行。

A→ a1 | a2 | … | an | b

%%

/\* P3: supporting C routines 用户辅助程序段(C函数) \*/

### Flex-Bison协作关系



### 结构体含义

定义段：包括文字块、定义、内部声明等。C语言的头文件、函数和变量的声明等一般就放在%{…%}之间，这一部分的内容会被直接复制到生成.c文件的开头部分。

规则段： %%...%%之间部分，为一系列匹配模式(正则表达式)和动作(C代码)。当flex扫描程序运行时，它把输入与规则段的模式进行匹配，每次发现一个匹配（被匹配的输入称为标记（token））时就执行与那种模式相关的C代码。

用户辅助程序段： 为C代码，会被原样复制到c文件中，一般这里定义一些辅助函数等。

## 文法的设计

根据课程设计要求设计如下文法。加入了除法和减法，后续的 and、or、not 修改为了类似 C 语言的 &&、||、! 符号。rop 运算包含了 <=、>=、。

S ′ → S E → − E

S → C S E → ( E )

S → begin L end E → id

S → A B → B or B

C → i f B then B → B and B

L → S B → not B

L → K S B → ( B )

K → L B → E rop E

A → id := E B → true

E → E + E B → f alse

E → E ∗ E

# 程序实现

## 程序总体思想

首先应当对使用指定文法的源文件进行词法分析，将整个文本分解为一个一个词素，然后 为每个终结符建立一个语法节点。将上述过程看作是一个解析函数，每次调用就生成语法节点 并返回词素的类型。

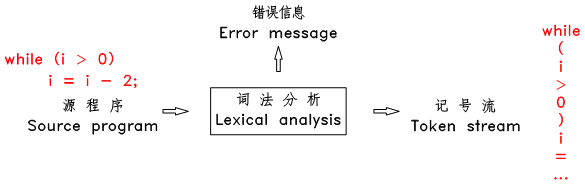
接着需要对其进行语法分析（事实上词法分析和语法分析是同时进行的）。使用 LR 分析技 术，按照产生式规则生成项集，然后构造项集的 goto 函数。此过程事实上完成了一个可以识别 此文法的 DFA，根据此 DFA 构造此文法的语法分析表，接着根据上一步词法分析器返回的终结 符类型来选择执行移入或规约的动作，在规约的同时生成新的语法节点并使其与子语法节点建 立连接关系。重复以上直到状态栈中的状态达到 Acc 也就是接收状态，说明输入字符串被此文 法接收。

上述动作执行完毕后，即构造出一棵完整的抽象语法树。

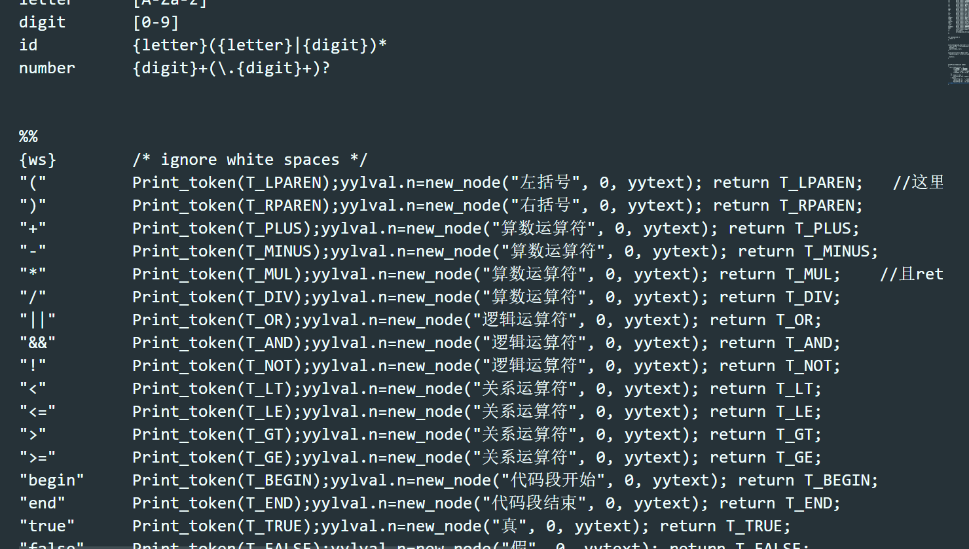
## 词法分析设计

词法分析过程借助GNU Flex工具使用正则表达式描述每一个单词的组成，即Flex会为每一个单词构建相应的DFA,其作用是构造出相应的yylex()函数，返回下一个输入流的单词类型，同时yyparse()函数每一次的语法分析，都需要调用相应的yylex()函数。利用Flex工具会生成lex.yy.c文件，该文件中包含了对于词法分析的相应的函数等。

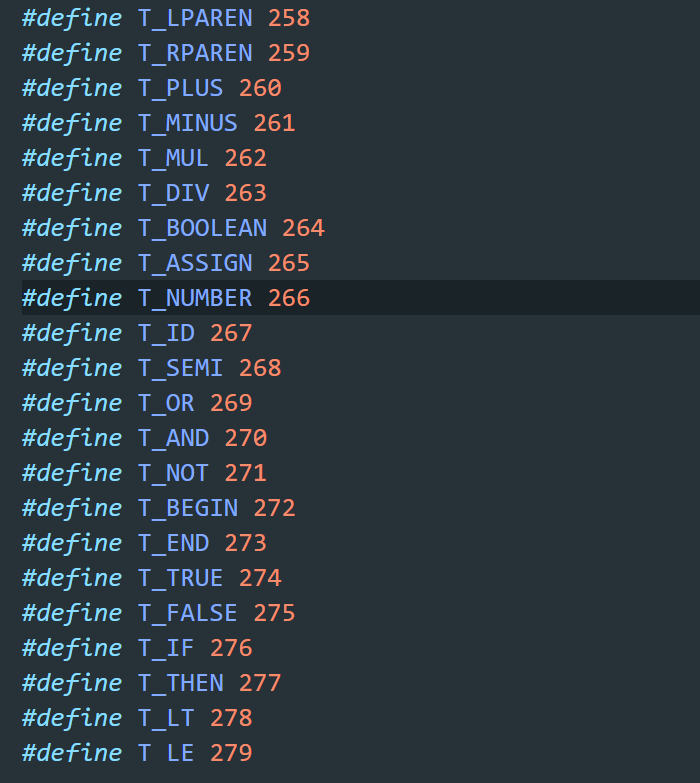
词法分析过程：



Scanner.l文件：



把识别的标识设计相应的数字：



y.tab.h 文件

## 语法分析设计

语法分析是借助Bison工具来构造相应的LALR自动机，同时会根据相应的文法构造出相应的语法分析表。在yyparse()函数调用下，对输入流的终结符序列进行语法分析，根据终结符的类型进行跳转，在合适的时机进行相应的移进与规约操作。

下图是得到的相应的自动机：

下面是根据Bison得到的状态,共55个状态：

State 0 0 $accept: . Q $end

State 1 9 A: "identifier" . ":=" E

State 2 3 S: "begin" . K "end"

State 4 0 $accept: Q . $end

State 5 7 L: S

State 6 2 S: C . S

State 7 1 Q: L . 8 K: L . ";"

State 8 6 L: K . S

State 9 4 S: A

State 10 9 A: "identifier" ":=" . E

State 11 8 K: L . ";"

State 12 3 S: "begin" K . "end

State 13 15 E: "(" . E ")" 21 B: "(" . B ")"

State 14 14 E: "-" . E

State 15 17 E: "number" .

State 16 16 E: "identifier" .

State 17 20 B: "!" . B

State 18 26 B: "true" .

State 19 27 B: "false" .

State 20 10 E: E . "+" E 11 | E . "-" E 12 | E . "\*" E 13 | E . "/" E 22 B: E . "<" E 23 | E . "<=" E 24 | E . ">" E 25 | E . ">=" E

State 21 5 C: "if" B . "then" 18 B: B . "||" B 19 | B . "&&" B

State 22 0 $accept: Q $end .

State 23 2 S: C S .

State 24 8 K: L ";" .

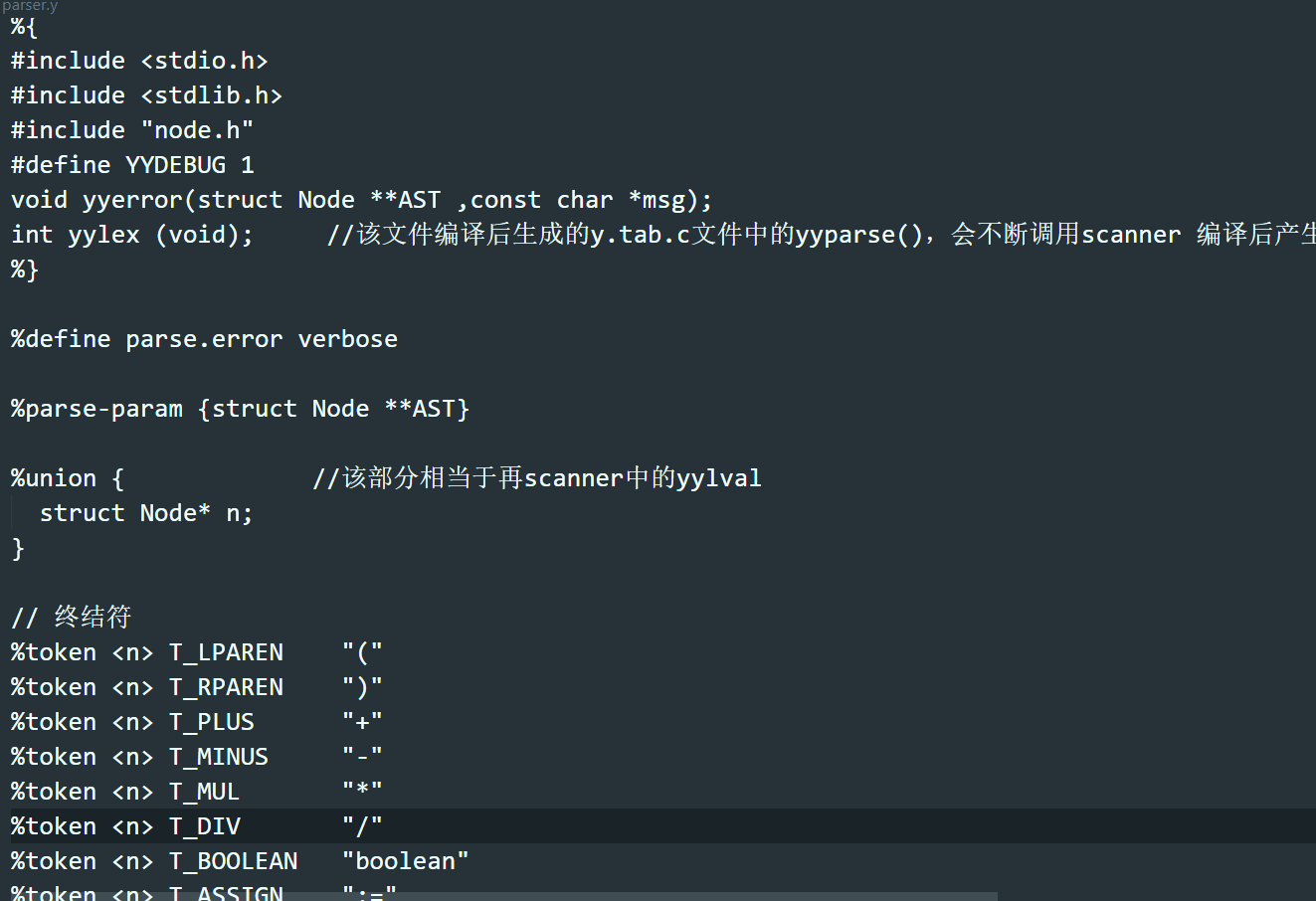
State 25 6 L: K S .

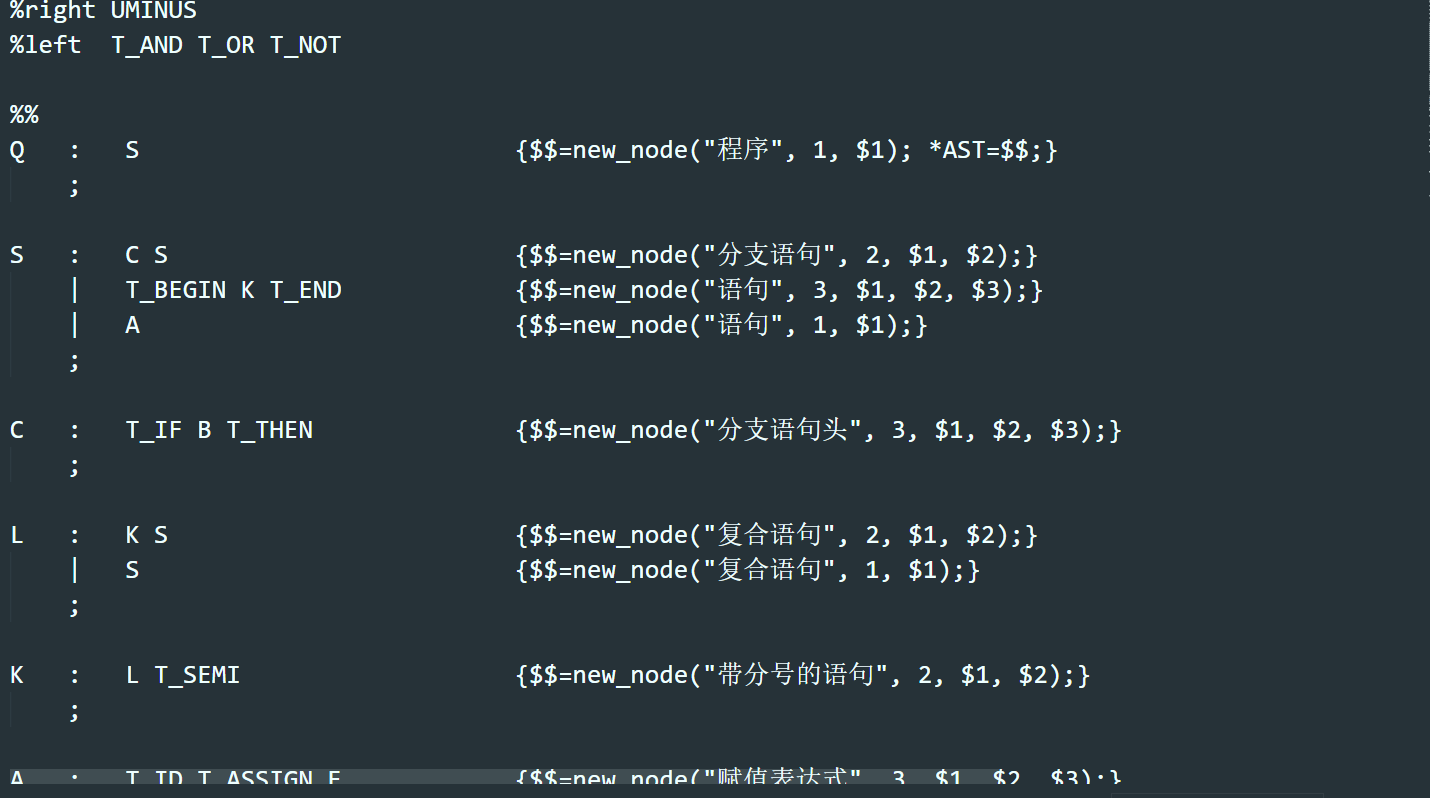
State 26 15 E: "(" . E ")"

…..

在利用Bison对于parser.y文件进行操作会生成y.tab.c 、y.tab.h、y.output三个文件,其中y.output文件中包含了生成的所有的状态集。且在在y.tab.c中包含了对于语法分析的各类函数结构体的定义以及函数、相关栈的构建等。

下图为parser.y函数：





在课本与实际编程过程中，对于文法的语法过程中会遇到移进与规约冲突，于是分析yy.output查看冲突位置可知，是由于加减乘除运算符的二义性问题，于是决定在parser.l文件中规定相应的运算符优先级来解决移进规约冲突问题，以此来解决二义性。

// 算符优先级

%left T\_PLUS T\_MINUS

%left T\_MUL T\_DIV

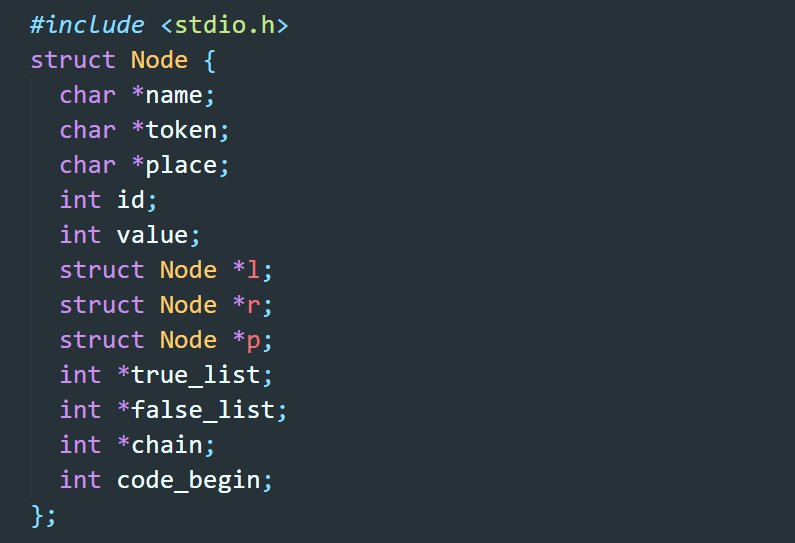
%right UMINUS

%left T\_AND T\_OR T\_NOT

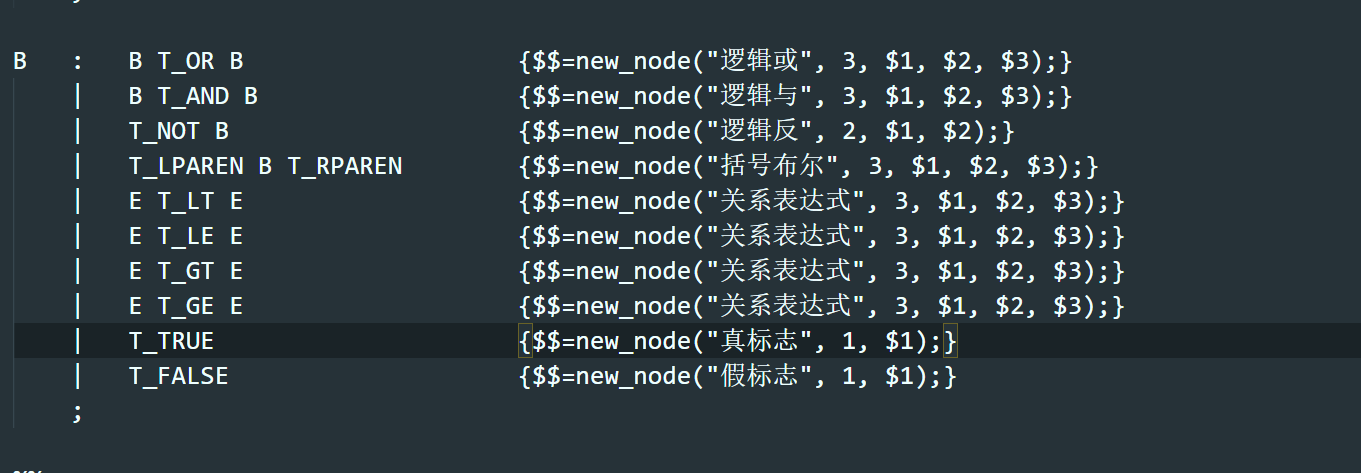
## 抽象语法树的生成

在语法分析的过程中在对输入流的关键字规约的同时建立起抽象的语法树，在构建语法树的前提下，首先需要了解语法树的结构体。其中name属性表示节点类型，token属性只有终结符才有。l 指针指向其第一个子节点，r 指针指向 其第一个兄弟节点，p 指针指向其父节点。true\_list、false\_lit、code\_begin、chain 均为后续对分 支语句语义分析时所用。由于使用的C语言编程，因此在数据结构体方面有些地方显得有些复杂。

AST节点数据结构：

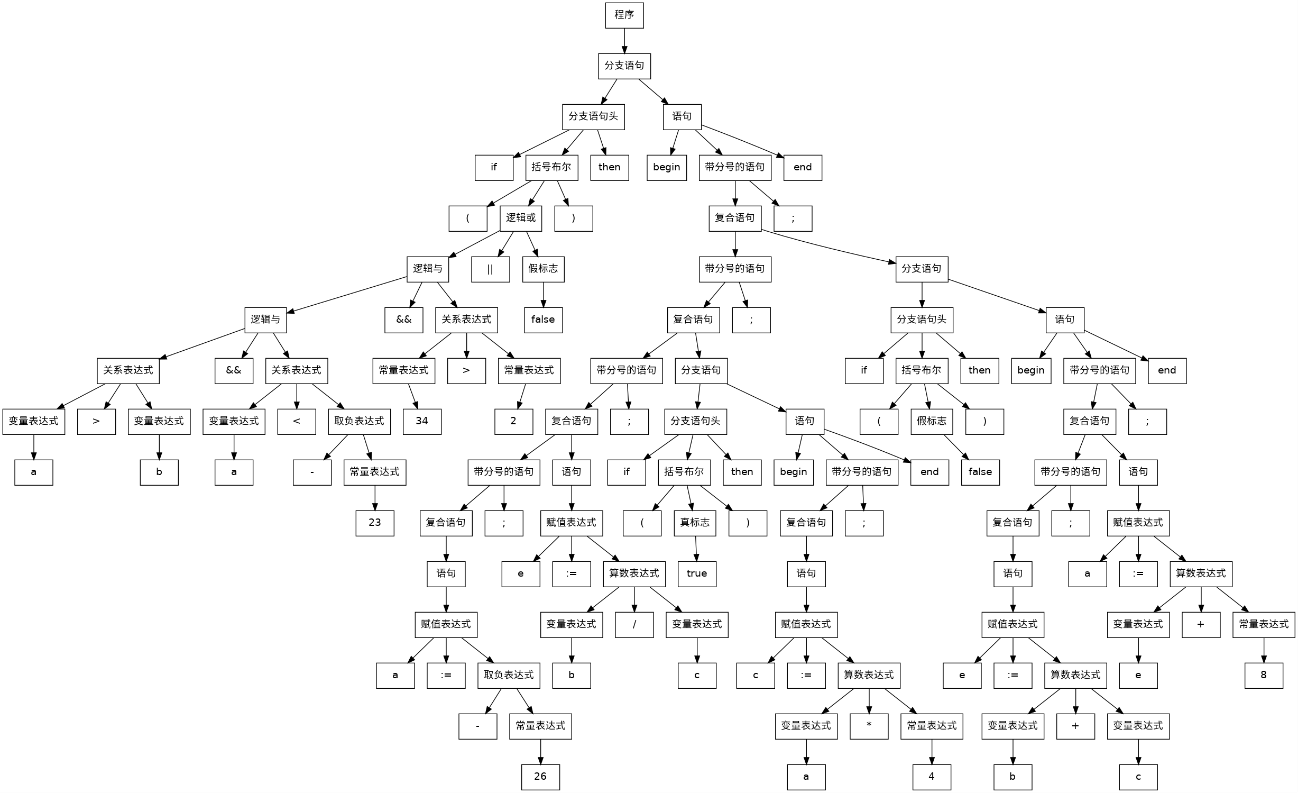


在规约的过沉中会伴随着其他状态的转换，即生成新的节点，需要一个对算术表达式构建语法树Bison的操作过程。



## AST图片的设计

由于本次实验实在Lniux上设计的编译原理课设，在上面得到了通过语法分析得到的AST抽象语法树，于是通过类似流程图样子展示出来，于是通过网上了解到Graphviz 绘图库，于是将相应的语法树以图片的形式展示出来。通过apt install graphviz 下载相应的包，通过./my\_compiler -t <1 >1.dot &&dot -Tpng 1.dot>1.png 即可得到相应的图。



## MakeFile文件的设计

通过在网上学习相关编译器构建过程，让我了解到Make的用法,也通过相关网站进行了相关用法的学习。由于在利用Flex与Bison工具过程需要手动的利用工具将相关.y与.l文件进行处理，得到相应的.c文件，然后再利用gcc编译器将众多的.c文件编译成一个可以执行的文件，就显得笨拙。于是利用Make，构建相关的MakeFile 文件，将相关操作在该文件中以目标任务的形式进行规定，然后只需要利用make 指令即可以按照要求的顺序顺利的执行上述的操作。下述为相关MakeFile文件中语法运用：

CC = gcc

OUT = my\_compiler

OBJ = lex.yy.o y.tab.o node.o main.o

SCANNER = scanner.l

PARSER = parser.y

build: $(OUT)

rm -f \*.o lex.yy.c y.tab.c y.tab.h y.output

clean:

rm -f \*.o lex.yy.c y.tab.c y.tab.h y.output y.dot $(OUT)

graph: $(PARSER)

bison -vdty --graph $< && rm -f \*.o y.tab.c y.tab.h y.output

$(OUT): $(OBJ)

$(CC) -o $(OUT) $(OBJ)

lex.yy.c: $(SCANNER) y.tab.c

flex $<

y.tab.c: $(PARSER)

bison -vdty $<

运用Make完成了整个编译软件的构建，也清除了在过程中产生的多余文件。

# 测试用例以及实验结果

## 文件测试用例



## 使用说明

本项目是在Ubuntu下开发的，因此需要gcc编译器来构建，且需要下载相应的库才可以实现整个编译器的运行。

整个项目需要在Ubuntu下运行，需要通过apt install flex 、apt install bison 、apt install make、apt install gcc 、apt install graphviz 下载相关的库。

由于我已经写了Makefile文件,以此在该目录下执行make指令即可以执行相关文件

可执行文件名是 my\_compiler，执行时有一个参数：

**-t 用来生成抽象语法树**

所有输入默认以标准输入流方式读入，在实际使用中可以使用重定向或者管道来将文件中 的字符串输入程序，输入默认也是向标准输出流输出，同样可以使用重定向来将输出结果保存 到文件。

## 文件测试

### 文件1测试

文件1源代码

f (a > b && a < -23 && 34 > 2 || false) then

begin

a := -26;

e := b / c;

if (true) then

begin

c := a \* 4;

end

;

if (false) then

begin

e := b + c;

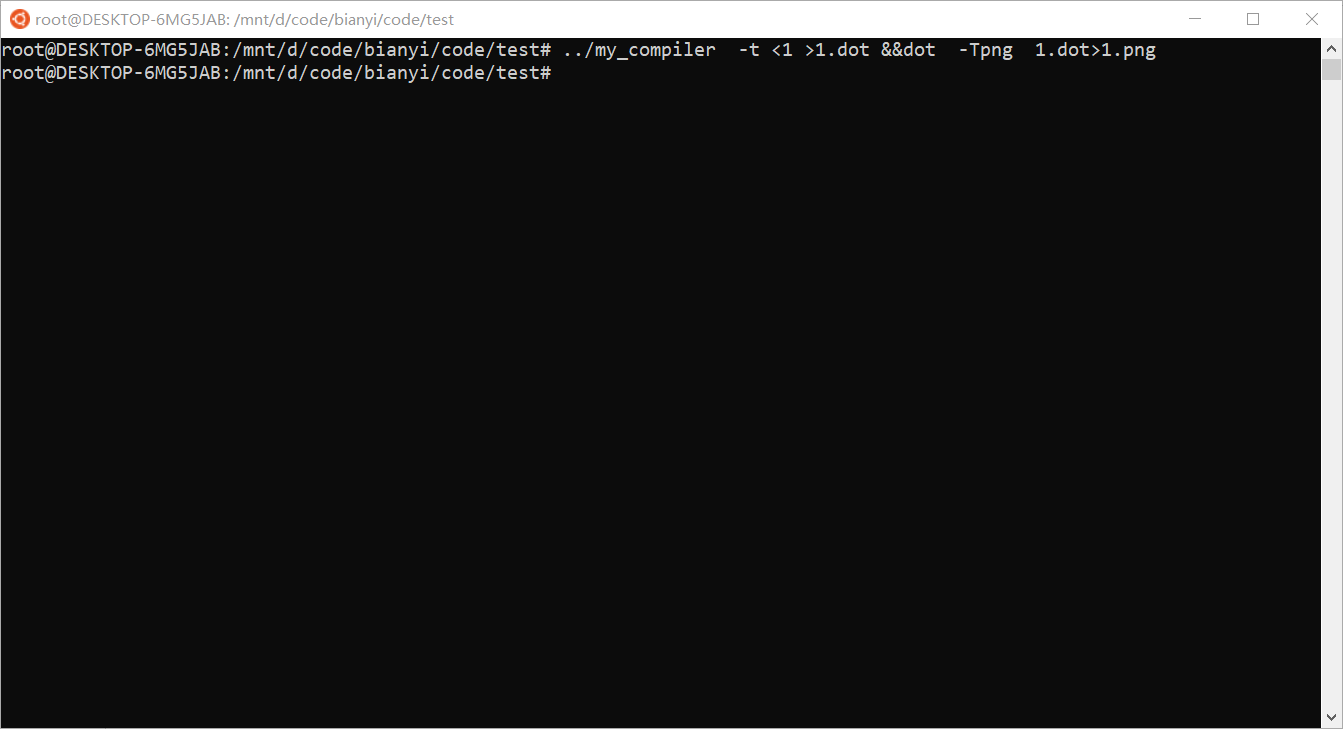
a := e + 8;

end

;

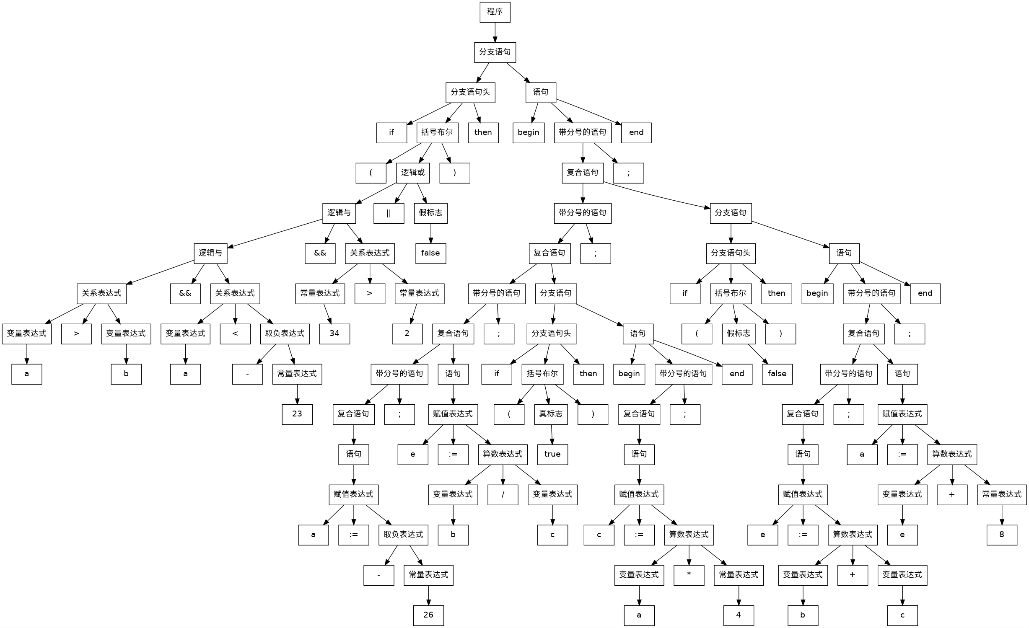
End

操作过程



输出结果

AST图



..........词法分析结果为.............

类型 Token 符号

T\_IF 276 if

T\_LPAREN 258 (

T\_ID 267 a

UMINUS 280 >

T\_ID 267 b

T\_AND 270 &&

T\_ID 267 a

T\_GT 278 <

T\_MINUS 261 -

T\_NUMBER 266 23

T\_AND 270 &&

T\_NUMBER 266 34

UMINUS 280 >

T\_NUMBER 266 2

T\_OR 269 ||

T\_FALSE 275 false

T\_RPAREN 259 )

T\_LE 277 then

T\_BEGIN 272 begin

T\_ID 267 a

T\_ASSIGN 265 :=

T\_MINUS 261 -

T\_NUMBER 266 26

T\_SEMI 268 ;

T\_ID 267 e

T\_ASSIGN 265 :=

T\_ID 267 b

T\_DIV 263 /

T\_ID 267 c

T\_SEMI 268 ;

T\_IF 276 if

T\_LPAREN 258 (

T\_TRUE 274 true

T\_RPAREN 259 )

T\_LE 277 then

T\_BEGIN 272 begin

T\_ID 267 c

T\_ASSIGN 265 :=

T\_ID 267 a

T\_MUL 262 \*

T\_NUMBER 266 4

T\_SEMI 268 ;

T\_END 273 end

T\_SEMI 268 ;

T\_IF 276 if

T\_LPAREN 258 (

T\_FALSE 275 false

T\_RPAREN 259 )

T\_LE 277 then

T\_BEGIN 272 begin

T\_ID 267 e

T\_ASSIGN 265 :=

T\_ID 267 b

T\_PLUS 260 +

T\_ID 267 c

T\_SEMI 268 ;

T\_ID 267 a

T\_ASSIGN 265 :=

T\_ID 267 e

T\_PLUS 260 +

T\_NUMBER 266 8

T\_SEMI 268 ;

T\_END 273 end

T\_SEMI 268 ;

T\_END 273 end

### 文件2测试

文件2源代码

begin

a := 3;

b := 4;

if (a > 2) then

begin

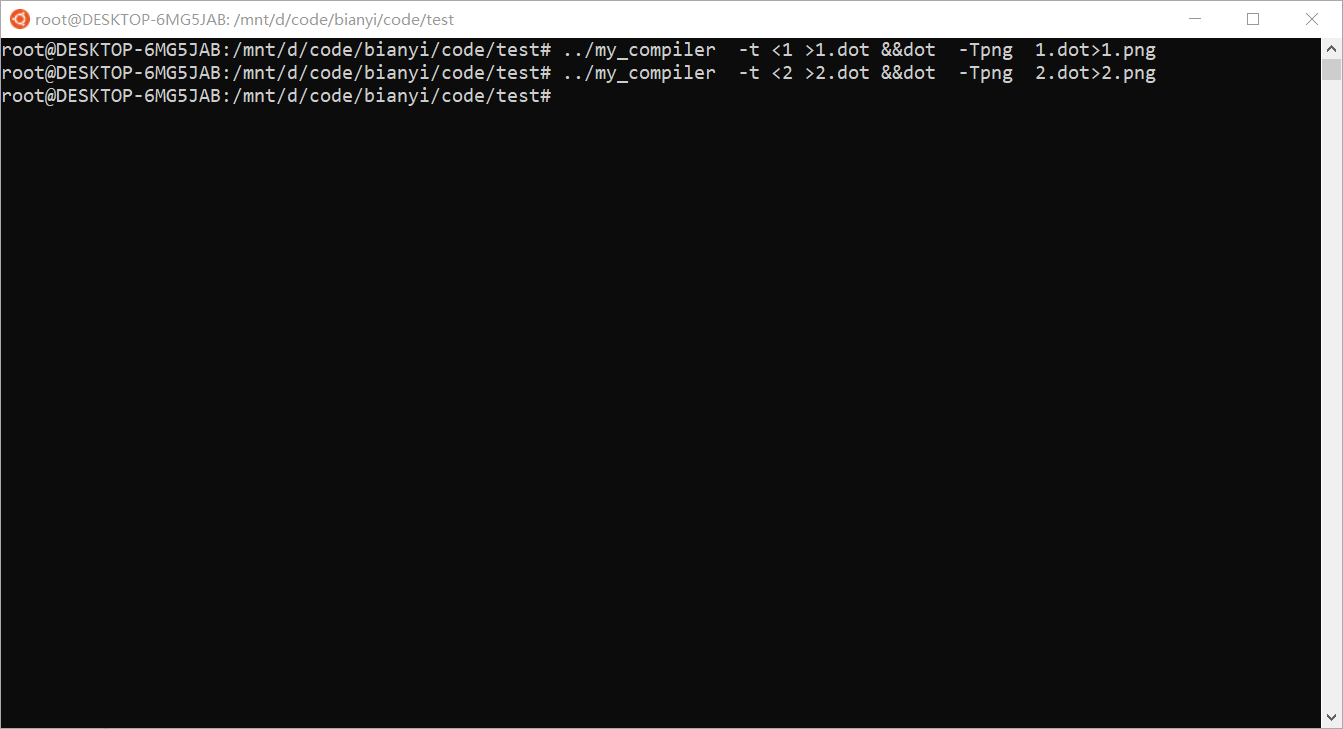
a := b + 1;

end

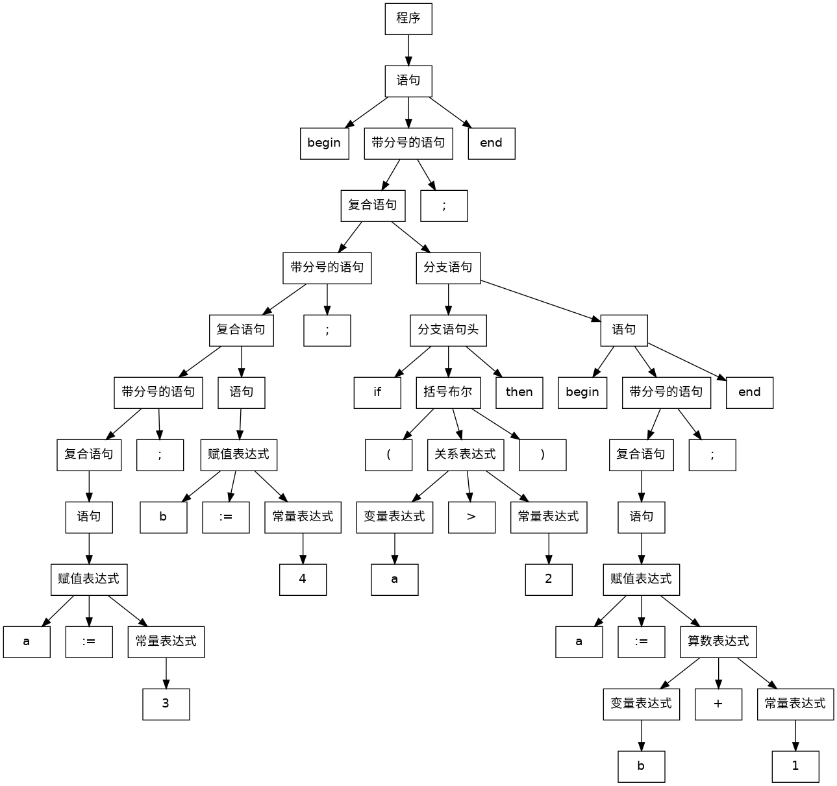
;

End

操作过程：



AST图：



..........词法分析结果为.............

类型 Token 符号

T\_BEGIN 272 begin

T\_ID 267 a

T\_ASSIGN 265 :=

T\_NUMBER 266 3

T\_SEMI 268 ;

T\_ID 267 b

T\_ASSIGN 265 :=

T\_NUMBER 266 4

T\_SEMI 268 ;

T\_IF 276 if

T\_LPAREN 258 (

T\_ID 267 a

UMINUS 280 >

T\_NUMBER 266 2

T\_RPAREN 259 )

T\_LE 277 then

T\_BEGIN 272 begin

T\_ID 267 a

T\_ASSIGN 265 :=

T\_ID 267 b

T\_PLUS 260 +

T\_NUMBER 266 1

T\_SEMI 268 ;

T\_END 273 end

T\_SEMI 268 ;

T\_END 273 end

### 文件3测试

文件3源代码

begin

a := (2 - 1) \* 3;

b := -3

if (false && a > b) || a > b then

begin

c := 1 + 2;

d := a / b;

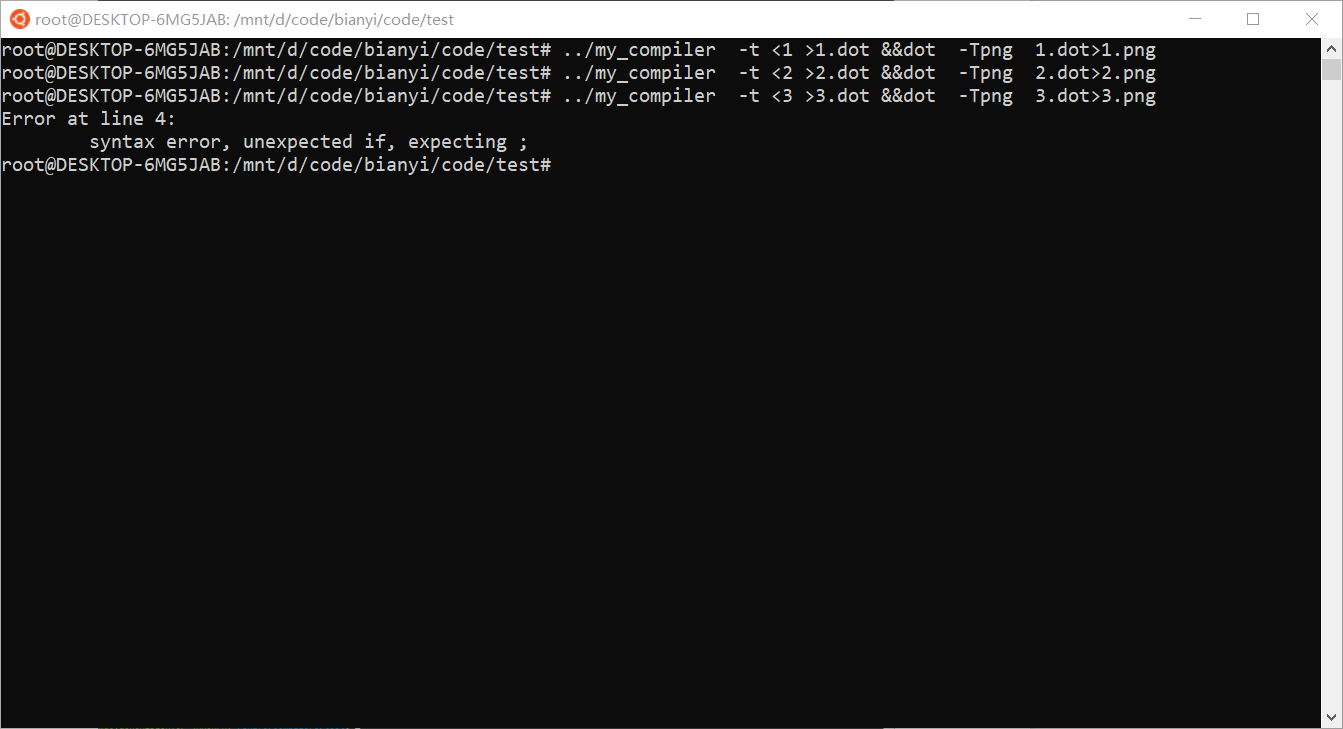
end

;

a := a + 1;

end

操作过程



错误信息：

Error at line 4:

syntax error, unexpected if, expecting ;

..........词法分析结果为.............

类型 Token 符号

T\_BEGIN 272 begin

T\_ID 267 a

T\_ASSIGN 265 :=

T\_LPAREN 258 (

T\_NUMBER 266 2

T\_MINUS 261 -

T\_NUMBER 266 1

T\_RPAREN 259 )

T\_MUL 262 \*

T\_NUMBER 266 3

T\_SEMI 268 ;

T\_ID 267 b

T\_ASSIGN 265 :=

T\_MINUS 261 -

T\_NUMBER 266 3

T\_IF 276 if

# 技术问题以及解决方案

## 技术问题

1.由于这次实验实在Flex与Bison工具下做的词法与语法分析，遇到的问题在于开始时，不知道从何下手，在使用 Bison/Flex 工具的时候由于中文资料比较少，所以主要通过 GNU 的官方文档来学习 这两种工具的用法，同时也熟悉了 LR 分析的流程。

2.在使用Flex与Bison时，不清楚二者工具之间是如何工作的，通过大量学习网上的例子，了解了二者的语法与规则以及二者之间需要通过相应的函数，才能实现词法与语法的分析，且通过这次实验，让我明白了词法与语法的分析是同时进行的。

3.在做之前了解到Flex与Bison与Linux有着更好的交互与联系，于是选择在Ubuntu上做，且整个文件是通过管道，数据流的形式输入的，最后才明白整个程序的执行过程：

1).首先在控制台通过标准输入流输入某个文本的文件内容

2).yyparse接受当前的输入流(bision编译的y.tab.c文件中的)在接受时调用flex()

3).在输入文件时调用flex生成的(yylex()函数),将文件的输入流解析为一个个单词，还有返回单词的属性(返回的是y.tab.h中已经定义好的宏定义)（返回到yypars）

4).yylval时bision全局变量，yytext是flelx当前解析的单词，给yylval赋值(union结构体),让yyparse函数得到当前解析到单词的值。

5).此时yyparse函数开始进行语法分析，(得到的终结符进行移进,再利用yylex函数调入下一个单词，单词是否为规约还是移进。

6).根据语法分析的过程生成相应的AST，在构建出相应的AST图。

4. 在语法分析的时候会发现使用默认的文法会产生移进/规约冲突，虽然不影响正常的词法分 析但是会影响效率，在重新观察了文法之后发现是由于运算符的优先级导致了文法的二义性，所 以我人为的制定了运算符的优先级和结核性来消除文法的二义性来解决移入/规约的冲突。

5. 在程序构建时学会了 Makefile 的编写使得程序构建的过程更简单，构建流程也变得自动化。

## 心得体会

通过这次编译原理课设，让我对于编译器的工作原理有了更加深刻的理解，不再是简单针对于课本上简单的词法、语法分析，这次实践让我更加通透的去理解整个编译的过程。不过令人可惜的是没有时间来的及去做语义分析，以及整个过程是借助工具去实现的，希望下次有机会可以自己实现手动完成词法、语法、语义的分析。特别是对于 LR 方法的理解上，也通过代码让我切身的体会到编译原理的精妙之处。

这次课设更令我感触深的是对于在Linux系统上完整的执行一个程序，对于Linux下库的下载、引用以及管道的使用有了更深的认知，以及对于Flex、Bison、make、gcc四个工具的使用，让我对于Linux这个开源世界有了更深的向往，我相信对于我以后的学习一定有很大的帮助。

# 备注

/code 根目录

test 测试文件

main.c 主程序

makefile 构建文件

my\_comilper 可执行文件

my\_compiler 可执行程序

node.c 抽象语法树文件

node.h 抽象语法文件

parser.y GNU bison 源文件

scanner.l GNU flex 源文件

参考文献：

[1] <https://pandolia.net/tinyc/index.html>

[2] <https://www.gnu.org/>

[3] <https://www.gnu.org/software/make/manual/make.html>

[4] <https://graphviz.org/>