**编译原理实验一实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| **组号: 21** | **学号** | **姓名** | **联系方式** |
| **组长** | **161220096** | **欧阳鸿荣** | **895254752@qq.com** |
| **组员** | **161220097** | **戚赟** | [**986300572@qq.com**](mailto:986300572@qq.com) |

**一、实验目标和完成的功能**

**1.1实验要求：**

程序要能查出 C--源代码中可能包含的下述几类错误:

1)**词法错误(错误类型A)**: 出现C--语法中未定义的字符，以及任何不符合C--词法单元定义的字符;

2)**语法错误(错误类型B)**。

除此之外，还需实现**要求1.3**：识别’//’和”/\*...\*/”形式的注释。如果包含符合定义的注释，程序需要能够滤出这样的注释；若输入文件中包含不符合定义的注释，程序需要给出不符合定义引发的提示信息。

**1.2 完成功能：**

我们完成了**所有的基本功能**（词法分析和语法分析），能正确报出错误的类别和行号，同时**实现额外要求1.3**,对于注释也能进行识别和报错。

**二、实验实现方式**

**2.0 文件结构:**

**额外的文件:**

“common.h”: 包含共同的头文件和公共变量

“GramTree.h”: 语法树的头文件，包含节点的定义和功能函数

“GramTree.c”: 功能函数的实现

**2.1 编译测试:**

由于实验要求不进行flex识别的输出，但是如果debug的话我们需要这些输出，于是定义了一个伪目标debug模式，来进行打印输出。

debug:

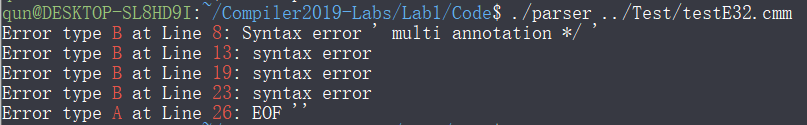
    flex lexical.l

    bison -d -v -t syntax.y

    gcc syntax.tab.c GramTree.c main.c -lfl -ly -o parser -D YY\_DEBUG

输入**make debug**来生成parser目标程序，由于 YY\_DEBUG 的定义，parser执行分析的时候就能进行输出flex词法单元，从而能更好进行调试。除了”debug”模式，我们还定义了**gramTest**等伪目标方便语法树的测试和自己定义额外的测试文件，所以实验具有方便的debug模式和较多的用例测试。

在打印的输出，我们还进行了**语法高亮**，使得输出错误更加明显。



**2.2 词法分析:**

**2.2.1** **基本实验:**由于flex部分进行词法分析，主要的内容是利用正则表达式匹配语法单元，针对C--定义，我们给出了单元的正则表达式。典型的几个如下:

1.匹配十进制数字

digit\_10    [0-9]

Dec         [1-9]{digit\_10}\*|0

说明十进制数要么是0，要么是不能以0开头的数字串。

2.匹配float

float       [+-]?({digit\_10}\*[.]{digit\_10}+|{digit\_10}+[.])

参考书上写法，进行分情况整合。

3.匹配ID

letter      [\_A-Za-z]

ID          {letter}({letter}|{digit\_10})\*

表明开头不能是数字，由下斜杆、字母和数字组成的符号串。

4. 特别注意由于正则表达式的特殊性，某些符号前面需要加’\’来代表原意符号，比如：”|”前需要加”\”进行转义。

5. 在书写正则表达式的时候，由于flex是从上往下依次匹配的，所以如果前面都无法匹配会被识别成错误的符号。虽然flex会进行提醒，但我们同时也要注意书写顺序，否则书写顺序靠后的规则将无法识别。

**2.2.2额外要求1.3:**

对于C-源代码中的注释有2种风格，对于不同风格的注释，采用不同的方法。

1. 首先定义几个正则表达式并为其取别名：

single      \/\/ #表示单行注释起始 //

multiStart \/\\* #表示多行注释开始 /\*

multiEnd    \\*\/ #表示多行注释结束 \*/

1. 使用双斜线“//”进行单行注释，这种情况下我们采用Flex的库函数来进行过滤。在输入文件中发现双斜线“//”，后，将从当前字符开始一直到行尾的字符全部丢弃。

{single} {

    char c = input();

    while (c != '\n') c = input();

}

对于多行注释，主要利用Flex提供的开始开始条件机制来进行识别与过滤。首先声明一个互斥的开始条件CONTENT，则可以定义以下规则：

%x CONTENT

首先，是定义CONTENT的激活条件以及定义回到开始状态的规则

{multiStart} {  BEGIN CONTENT; }

//…

<CONTENT>"\*"+"/" {  BEGIN INITIAL;}

表示在匹配到多行注释开始条件后，激活CONTENT，并在CONTENT遇到终止条件后，对一个多行注释匹配完成。

同时，在上述代码中省略的部分，还需增加如下条件过滤注释内容：

<CONTENT>[^\*\n]\* { //过滤不是\*的行 }

<CONTENT>"\*"+[^\*/\n]\* { //过滤\*后不是接着/的行 }

<CONTENT>\n { //保证行数正常计数 }

通过开始条件的使用，也可以对嵌套的多重注释等错误进行识别：

<CONTENT><<EOF>> { BEGIN INITIAL; //...并记录错误}

{multiEnd} { //..记录错误}

**2.3 语法分析**

**2.3.1 文法分析和语法树构建:**

对于语法树的构建，我们在实验中定义了如下数据结构：

typedef *struct* TreeNode{

*int* lineNo;         //TempLineNumber

*int* nChild;         //Number of children

*char* tag[32];

*union* { //value

*int* a;

*float* b;

*char* str[32];

    } val;

*struct* TreeNode \*child[MAX\_CHILD];

}GramTree;

通过定义一个全局变量 GramTree \* treeRoot 后，我们在Bison代码中的定义部分，将词法单元的YYTYPE都定义成GramTree \*的类型，即可在规则部分中通过我们所书写的C语言文法定义，自上而下地构建输入文件的语法树。通过如下函数构建：

GramTree \* newTreeNode(*char*\* *tag*,*int* *n*, ...);

其中tag是词法单元的名称，n是非终结符的产生式右侧的单元个数，最后是根据n所确定的参数个数，也即该树节点的子节点。通过如下定义，则可如下构建语法树：

Program: ExtDefList { $$ = treeRoot = newTreeNode("Program",1,$1); };

Args: Exp COMMA Args  { $$ = newTreeNode("Args", 3, $1, $2, $3);}

    | Exp    { $$ = newTreeNode("Args", 1, $1); };

每次产生新节点时给节点加上对应的词法单元名，同时在为词法单元赋值并返回

{STRUCT} {  yylval.node\_type = newTreeNode("STRUCT",0,yylineno);

return STRUCT; }

则通过上述操作，我们可以自上而下地进行语义分析，同时成功构建语法树。

**2.3.2 错误恢复**

根据书本上的错误恢复方式，我们自定义错误规则，书写包含error的产生式，从而然后对于错误进行匹配识别，从而达到要求。举例来说，**Compst-> error RC**能够进行括号”{”, “}”的匹配。

我们自定义了十几条包含error的产生式来匹配规则，基本满足测试用例的要求，具体规则请见 **syntax.y** 文件。

**三、实验编译方式**

**3.1实验的编译方式如下:**

① 进入 **Lab1/Code**目录下

② 执行命令**make clean**，再执行**make parser** 命令（先执行make clean 防止先前文件定义冲突）

③ 生成目标程序**parser**，执行命令 “**./parser test.cmm**”来启动parser对于test.cmm的分析。或者改变 makefile中make test的.cmm文件位置然后执行’make test’操作即可。其中提供测试文件放在 **Lab1/Test**目录下。