**编译原理 实验报告**

|  |  |  |
| --- | --- | --- |
| **评 语** | **成绩** |  |
| 教 师： 邓岳  年 月 日 | | |

**教学班级： 01**

**学生学号： 19030500024**

**学生姓名： 马子豪**

**实验日期： 2021.12.18**

# 一、实验目的

通过实验实现一个简短绘图语言解释器，加深对编译器构造原理和方法的理解，巩固所学知识。

1. 会用正规式设计简单语言的词法；
2. 会用产生式设计简单语言的语法；
3. 会用flex和bison编写语言的解释器。

# 二、实验环境

**硬件环境：**

* Intel i5-9300H CPU 2.40GHz 4核
* 8GB 内存
* 1T 硬盘

**软件环境：**

* **主机操作系统**：Windows 10
* **实验平台**：Windows Subsystem for Linux 2(WSL2): Ubuntu20.04
* **工具**：flex bison
* **程序设计语言**：C++
* **绘图库**：GitHub开源项目 matplotlib-cpp
  + **依赖：**Python3.6.0以上，matplotlib3.5.1以上

# 三、实验内容

**项目目录树,**摘录重要部分：

.

├── CMakeLists.txt 将src中的源文件与matplotlib编译链接

├── Makefile CMakeLists.txt 生成的文件

├── bin 生成的可执行文件目录

│ └── funcdraw make之后生成的可执行文件（绘图解释器）

├── matplotlibcpp.h matplotlibcpp头文件

└── src 用户源文件目录

├── Makefile 用于执行简单命令，将lex和yacc文件编译为C++文件

├── lex.yy.cpp scanner.l编译生成的C++文件

├── main.hpp scanner.l, parser.y和semantics.cpp共用的头文件

├── parser.y 语法分析yacc源文件

├── scanner.l 词法分析lex源文件

├── semantics.cpp 语义分析C++源文件

├── y.tab.cpp parser.y生成的C++文件

└── y.tab.hpp parser.y生成的符号表头文件

## 3.1 词法分析

词法分析部分使用工具flex完成。

首先，对Token的种类进行划分：

|  |  |
| --- | --- |
| **Token种类** | **Token** |
| 常数 | CONST\_ID |
| 参数 | T |
| 函数 | FUNC |
| 字符串 | STR |
| 保留字 | ORIGIN, SCALE, ROT, IS, FOR, FROM, TO, STEP, DRAW,  TITLE, COMMENT, CLEAR, NEXT, COLOR, LINE等 |
| 运算符 | POWER, PLUS, MINUS, MUL, DIV |
| 分隔符 | SEMICO, L\_BRACKET, R\_BRACKET, COMMA, |

表格1 Token种类

这些Token在yacc源文件的符号定义区定义后，只需要在lex源文件中引入y.tab.hpp即可使用Token。

由于我使用的是词法分析器生成工具flex，在进行词法分析时，需要先在程序的**正规定义**处，定义如下单词的正规式：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| number | whitespace | newline | comments | str |
| [0-9]+(\.[0-9]\*)? | (" "|\t|\r|\f|\v)+ | \n | "//"|"--"[^\n]\* | \<.\*\> |

表格2 正规定义

然后，在程序的**词法规则**处写入识别出不同Token的**正规式**和**动作**即可进行词法分析。语法分析程序会调用lex生成的yylex词法分析函数，每次识别到一个Token并返回给语法分析程序。

在词法分析阶段，在识别到种别为参数、保留字、运算符和分隔符的Token时，只会返回对应的Token；而在识别到种类为常数、字符串和函数的Token之后，我们需要保留其对应的Token属性，并返回语法分析器。下面是定义的全局变量，Token属性结构体：

|  |  |
| --- | --- |
| **1** | **struct Token{**  **string lexeme; //** 记号的字面值  **int type; //** 记号的种别  **double value; //** 记号的属性值  **double (\*FuncPtr)(double); //** 函数地址  }; |
| **2** |
| **3** |
| **4** |
| **5** |
| **6** |

代码1 Token属性结构体

* 识别出CONST\_ID，token.value=atof(yytext)获得常数值，并返回；
* 识别出STR，使用token.lexeme=yytext获得字符串值，并返回；
* 识别出FUNC，使用token.FuncPtr=某个函数名，获得函数地址，返回；

其中，yytext是lex的全局变量，是词法分析器当前读到的单词字面值。

## 3.2 语法分析

语法分析部分使用工具bison完成。

首先，我们需要在**符号定义区**定义所有在词法分析中使用到的Token。

然后，在**语法定义区**填写文法。由于bison生成的语法分析器是LALR分析器，是一种移进-规约分析器，能处理的文法比递归下降分析器多，并且因为左递归使得语法分析效率提高，所以填入的文法可以是二义文法。

但是，为了解决移进-规约分析中的移进-规约和规约-规约冲突，需要指明部分运算符的结合性和优先级：

|  |  |
| --- | --- |
| **1:** | **%left PLUS MINUS**  **%left MUL DIV**  **%right UNSUB**  **%right POWER** |
| **2:** |
| **3:** |
| **4:** |

代码2 运算符优先级和结合性

其中，优先级从上到下降低，结合性由%left和%right来规定。需要注意的是，Token为MINUS的符号“-”，可能是二元运算符“减”或者一元运算符“负”，这两者的结合性和优先级都不同。因此在第三行定义了一个%right UNSUB，只需要在相应候选式中使用%prec UNSUB，即可指定某个运算符拥有与UNSUB相同的优先级和结合性，如产生式Expression的倒数第二和第三个候选式所示：

|  |
| --- |
| **Program**: Program Statement SEMICO  |  ;  **Statement**: FOR T FROM Expression TO Expression STEP Expression DRAW L\_BRACKET Expression COMMA Expression R\_BRACKET  | ORIGIN IS L\_BRACKET Expression COMMA Expression R\_BRACKET  | SCALE IS L\_BRACKET Expression COMMA Expression R\_BRACKET  | ROT IS Expression  | TITLE IS STR  | COMMENT IS L\_BRACKET Expression COMMA Expression COMMA STR R\_BRACKET  | COLOR IS ColorChoice  | LINE IS LineChoice  | CLEAR  | NEXT  ;  **Expression**: ERRTOKEN  | T  | CONST\_ID  | Expression PLUS Expression  | Expression MINUS Expression  | Expression MUL Expression  | Expression DIV Expression  | Expression POWER Expression  | L\_BRACKET Expression R\_BRACKET  | PLUS Expression %prec UNSUB  | MINUS Expression %prec UNSUB  | FUNC L\_BRACKET Expression R\_BRACKET  ;  **ColorChoice**: RED  | GREEN  | BLUE  | BLACK  ;  **LineChoice**: SOLID  | DASHED  | DOTTED  ; |

代码3 二异文法

## 3.3 语义分析

### 3.3.1 抽象语法树

为了构建以Expression为根节点的抽象语法树AST，需要在语法分析每次规约为Expression时，添加语义动作——为Expression构造一个节点，该节点的数据结构会根据产生式右部变化而变化，如下图所示：

|  |  |
| --- | --- |
| **1** | **struct ExprNode{**  **int OpCode;** //Token：PLUS, MINUS, MUL, DIV, POWER, FUNC, CONSTID, T  **union {**  **//** 操作符节点是有两个孩子的内部节点  **struct {**  **struct ExprNode \*Left, \*Right;**  **}CaseOperator;**  **//** 函数调用是有一个孩子的内部节点  **struct {**  **struct ExprNode \*Child;**  **FuncPtr MathFuncPtr;**  **}CaseFunc;**  **//**常数和参数是叶子节点  **double CaseConst; //**绑定右值  **double\* CaseParm; //**绑定左值  **}Content;**  **};** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20** |

代码4 抽象语法树节点

该数据结构是针对非终结符Expression和终结符类别为操作符、函数、常数和参数的，在bison源文件中需要使用#define YYSTYPE struct ExprNode \*来定义语义变量类型为树节点指针，同时方便构造Expression的抽象语法树。

构造语法树时，根据输入的Token来区别该使用ExprNode中的哪个数据结构，构造语法树函数MakeExprNode如下，

|  |  |
| --- | --- |
| **1** | **struct ExprNode \* MakeExprNode(int opcode, ...){**  **//** 构造以操作符、函数、常数、参数构成的树  **va\_list ArgPtr;**  **struct ExprNode \*ExprPtr = new struct ExprNode;**  **ExprPtr->OpCode = opcode;**  **va\_start(ArgPtr, opcode);**  **switch(opcode){**  **case CONST\_ID:**  **ExprPtr->Content.CaseConst = (double)va\_arg(ArgPtr,double);**  **break;**  **case FUNC:**  **ExprPtr->Content.CaseFunc.Child = (struct ExprNode \*)va\_arg(ArgPtr,struct ExprNode \*);**  **ExprPtr->Content.CaseFunc.MathFuncPtr = (MathFuncPtr)va\_arg(ArgPtr,FuncPtr);**  **break;**  **case T:**  // 使得在绘图时能获得不断变化的t  **ExprPtr->Content.CaseParm = &Parameter;**  **break;**  **default:**  **ExprPtr->Content.CaseOperator.Left = (struct ExprNode \*)va\_arg(ArgPtr,struct ExprNode \*);**  **ExprPtr->Content.CaseOperator.Right = (struct ExprNode \*)va\_arg(ArgPtr,struct ExprNode \*);**  **}**  **return ExprPtr;**  **}** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30** |

代码5 构造语法树节点

在构造完一个节点后，需要将值赋给bison中的特殊变量$$，其本质是非终结符Expression的变量，类型是由上述YYSTYPE定义。打印语法树算法使用递归实现深度优先遍历语法树，仍然需要根据节点中Token来决定如何打印：

|  |  |
| --- | --- |
| **1** | **void TravelTree(struct ExprNode \* root, int indent){**  // 输出Expr的语法树  **indent += 4;** // 节点前缀空格个数  **switch(root->OpCode){**  **case CONST\_ID:**  **printf("%\*s%f\n",indent," ",root->Content.CaseConst);**  **break;**  **case FUNC:**  **printf("%\*s%p\n",indent," ",root->Content.CaseFunc.MathFuncPtr);**  **TravelTree(root->Content.CaseFunc.Child, indent);**  **break;**  **case T:**  **printf("%\*sT\n",indent," ");**  **break;**  **default:**  **printf("%\*s",indent," ");**  **cout<<GetTokenStr(root->OpCode)<<endl;**  **TravelTree(root->Content.CaseOperator.Left, indent);**  **TravelTree(root->Content.CaseOperator.Right, indent);**  **break;**  **}**  **}** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27** |

代码6 遍历语法树

为了获得Expression的值，需要递归遍历语法树中的节点，仍然需要根据节点的Token类型来求值：

|  |  |
| --- | --- |
| **1** | **double GetExprValue(struct ExprNode \* expr){**  // 获得Expr的值  **switch(expr->OpCode){**  **case CONST\_ID:**  **return expr->Content.CaseConst;**  **break;**  **case T:**  **return \*(expr->Content.CaseParm);**  **break;**  **case PLUS:**  **return GetExprValue(expr->Content.CaseOperator.Left) + GetExprValue(expr->Content.CaseOperator.Right);**  **break;**  **case MINUS:**  **return GetExprValue(expr->Content.CaseOperator.Left) - GetExprValue(expr->Content.CaseOperator.Right);**  **break;**  **case MUL:**  **return GetExprValue(expr->Content.CaseOperator.Left) \* GetExprValue(expr->Content.CaseOperator.Right);**  **break;**  **case DIV:**  **return GetExprValue(expr->Content.CaseOperator.Left) / GetExprValue(expr->Content.CaseOperator.Right);**  **break;**  **case POWER:**  **return pow(GetExprValue(expr->Content.CaseOperator.Left), GetExprValue(expr->Content.CaseOperator.Right));**  **break;**  **case FUNC:**  **return (\* expr->Content.CaseFunc.MathFuncPtr)(GetExprValue(expr->Content.CaseFunc.Child));**  **break;**  **default:**  **return 0.0;**  **}**  **}** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33** |

代码7 求Expression值

### 3.3.2 绘图

获得的Expression值需要付给绘图需要的全局变量，然后在用户输入for draw语句时，将其**规约**为Statement并加上**语义动作**绘图函数。

绘图函数为DrawLoop，通过函数CalCoord获得变换后的实际坐标向量，然后绘图函数中调用matplotlib-cpp头文件中的绘图方法plt::plot，将坐标向量中的每个点绘制出来。

## 3.4 测试结果

|  |  |
| --- | --- |
| **1** | **--test3:**  **--交叉圆**  **origin is (350, 200); -- 设置原点的偏移量**  **rot is pi/6; -- 设置旋转角度**  **scale is (2, 1); -- 设置横、纵坐标比例**  **color is red;**  **for t from -100 to 100 step 1 draw (t, 0); -- 横坐标**  **color is green;**  **for t from -100 to 100 step 1 draw (0, t); -- 纵坐标**  **scale is (200, 100); -- 设置横、纵坐标比例**  **color is blue;**  **for t from 0 to 2\*pi step pi/50 draw (cos(t),sin(t));** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6**  **7**  **8**  **9**  **10**  **11**  **12** |

代码8 test3

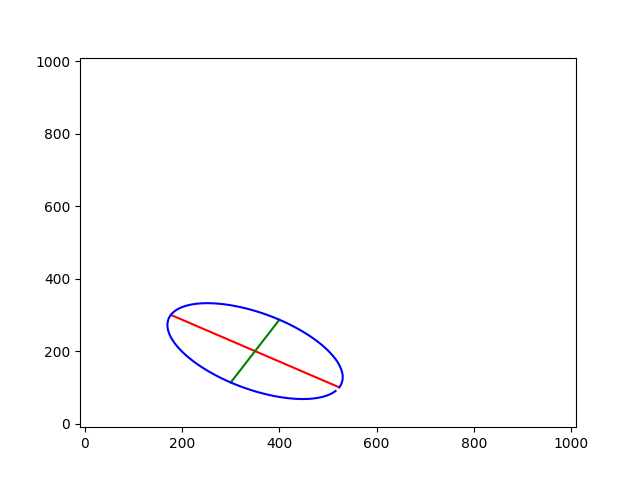


图1 test3结果图

# 四、心得体会

## 4.1 问题及解决方法

1. 将flex和bison源码编译为C++源码，然后再使用g++编译连接时会报出链接错误：
   * 原因：C++在编译时，为了解决多态问题，会生成一个由函数名和参数联合的中间函数名，而flex和bison生成的词法分析函数lex和yyparse是C函数，就找不到这两个函数。
   * 解决：用extern “C”对C函数进行链接指定即可。
2. 语法分析时，符号“-”不止是二元减，还是一元负号，两者优先级和结合性不同：
   * 解决：在bison源码的符号定义区，定义一个优先级更高并且有右结合的符号UNSUB，在语法规则区的相应候选式中使用%prec UNSUB指定该处的“-”的优先级和结合性与UNSUB一致即可

## 4.2 优点

1. 使用C++调用Python的matplotlib进行绘图，使得绘图简单、易扩充功能且图像清晰；
2. **附加功能：**
   1. 对逻辑坐标系进行了重新定义，与习惯上的坐标系一致；
   2. 扩充语句类型，允许用户使用clear语句清空当前画布，使用next语句使用下一个画布；
   3. 允许用户使用color is ColorChoice语句指定线条颜色；
   4. 允许用户使用line is LineChoice语句指定线条的样式；
   5. 允许用户使用title is <str>语句指定绘制图像的标题；
   6. 允许用户使用comment is (x\_coord,y\_coord,<str>)语句插入注释并指定注释的位置；

## 4.3 缺点

1. 指定标题和插入注释语句，并没有实现自动换行，因此输入的字符串过长时会使得字符串超出画布边缘；
2. 错误处理较为简陋，只能区别出哪一行有词法错误或者语法错误，然后继续分析，不能明确指出哪一列和错误的具体信息；