**注：各部分详细设计参见各部分的文档。**

# 目录

[1 目录 1](#_Toc54295059)

[2 课程设计目标和要求 2](#_Toc54295060)

[2.1 题目 2](#_Toc54295061)

[2.2 目标 2](#_Toc54295062)

[2.3 要求 2](#_Toc54295063)

[3 修改和扩展 2](#_Toc54295064)

[4 开发环境 3](#_Toc54295065)

[5 编译器模块图 3](#_Toc54295066)

[6 各模块对外接口 4](#_Toc54295067)

[6.1 词法分析 4](#_Toc54295068)

[6.1.1 词法分析的输入 4](#_Toc54295069)

[6.1.2 词法分析的输出 4](#_Toc54295070)

[6.2 词法分析和语法分析互联接口 4](#_Toc54295071)

[6.3 语法分析 4](#_Toc54295072)

[6.3.1 语法分析的输入 4](#_Toc54295073)

[6.3.2 语法分析的输出 5](#_Toc54295074)

[6.4 语法分析和语义分析的互联接口 5](#_Toc54295075)

[6.5 语义分析 5](#_Toc54295076)

[6.5.1 语义分析的输入 5](#_Toc54295077)

[6.5.2 语义分析的输出 5](#_Toc54295078)

[6.6 语义分析和代码生成之间的接口 5](#_Toc54295079)

[6.7 代码生成 5](#_Toc54295080)

[6.7.1 代码生成的输入 5](#_Toc54295081)

[6.7.2 代码生成的输出 5](#_Toc54295082)

[7 命令行接口详细设计 5](#_Toc54295083)

[7.1 命令行调用参数说明 5](#_Toc54295084)

[7.2 相关函数设计 6](#_Toc54295085)

[7.2.1 参数帮助文档初始化 6](#_Toc54295086)

[7.2.2 输出参数帮助文档 7](#_Toc54295087)

[7.2.3 将char\*指向的字符串保存到string中 8](#_Toc54295088)

[7.2.4 将char\*指向的字符串转化为整型 8](#_Toc54295089)

[7.2.5 获取命令行参数 9](#_Toc54295090)

[7.2.6 输出错误 11](#_Toc54295091)

[7.2.7 统计和检测错误的宏 12](#_Toc54295092)

[8 运行逻辑 12](#_Toc54295093)

[9 主函数流程图 13](#_Toc54295094)

[10 实验总结 13](#_Toc54295095)

[10.1 实验中遇到的问题和解决方案 13](#_Toc54295096)

[10.1.1 重命名恢复策略 13](#_Toc54295097)

[10.1.2 抽象语法树 14](#_Toc54295098)

[10.1.3 词法分析缓存下一行 14](#_Toc54295099)

[10.1.4 位置信息 14](#_Toc54295100)

[10.2 存在的问题/待改进的地方 14](#_Toc54295101)

# 课程设计目标和要求

## 题目

Pascal-S语言编译为C语言的程序的设计与实现

## 目标

按照所给Pascal-S语言的语法，参考Pascal语言的语义，设计并实现Pascal-S语言的编译为C语言的程序

## 要求

要求给出各阶段的设计成果，如：

* 需求分析报告
* 总体设计报告（软件功能描述、功能模块划分、软件结构图、符号表结构设计、模块间接口定义等）
* 详细设计报告（模块功能、输入/输出、处理逻辑等）
* 编码实现（源程序、可执行程序）
* 测试报告（测试计划、测试用例、测试结果及分析等）

# 修改和扩展

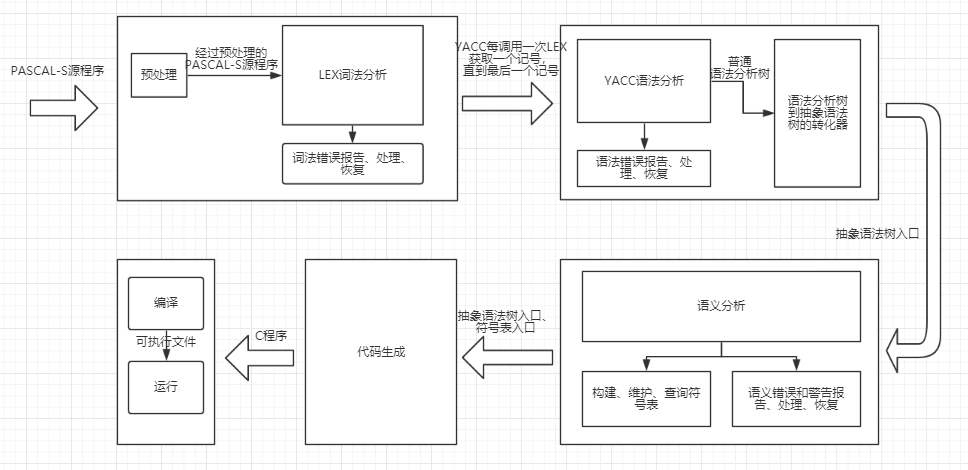
* 字符常量的检错与识别（包括周围单引号）放在了词法分析中
* 词法分析中，数值型常量设置了整数和浮点数两种，及整数和浮点数分别作为两种不同的终结符，UINUM和UFNUM，为unsigned integer number和unsigned float number的缩写
* 修改了常量的文法，为const\_value ---> + IDENTIFIER、const\_value ---> - INDENTIFIER、const\_value ---> IDENTIFERI、 const\_value ---> + UNINUM、const\_value ---> - UNINUM、const\_value ---> UINUM、const\_value ---> + UFNUM、const\_value ---> -UFNUM、const\_value ---> UFNUM、const\_value ---> CHAR，其中CHAR由词法分析器识别
* 修改了factor的文法产生式，将factor ---> num拆成了factor ---> UNINUM和factor ---> UFNUM两种，并增加了字符常量，即factor ---> CHAR
* 函数返回值语句支持赋值形式和exit调用两种方法
* 库程序支持read、write、writeln、exit的调用，read要求实参个数大于等于1，write要求实参个数大于等于1，writeln实参个数无要求，exit在过程中出现时，要求没有实参，exit在函数中出现时，要求实参个数为1。且read的实参只能为普通变量，即能提供可供修改的存储单元地址的简单表达式。
* 除了for循环，还支持while和repeat-until循环
* 语义分析非常的详尽，除了常规的语义分析，还考虑了如下内容
  + 规定了所有标识符都不能和库程序名，主程序名，主程序参数名同名，且这三者之间不能同名，并进行了检查
  + 支持整型常量表达式的计算
  + 支持由整型常量表达式指定的数组下标的越界检查
  + 支持/、div、mod由整型常量表达式指定的操作数的除0检查
  + 检查了引用形式参数对应的形参种类，必须是普通变量，即能提供地址的、且可修改的最简单的表达式，包括read库程序调用的实参。而不能是常量、常量标识符、复杂表达式等。
  + 用一种递归的方法，检查了函数定义中，返回值语句的存在性，将其作为**警告信息**输出，具体方法见语义分析详细设计。
  + 一般情况下支持integer到real的隐式类型转换；传引用参数时，则不支持隐式类型转换，类型必须强一致。
* 代码生成中，深入分析了引用参数和指针之间的转化，将引用参数以较好的方式转化为了指针。例如可以判断当前变量是否为指针（原PASCAL-S程序中的引用参数），如果是指针，则**在scanf中出现该变量**时，不需要任何前缀，如果是普通变量出现在scanf中，则需要加取地址符。
* 原PASCAL-S语法中，可能出现6--3这种表达式，注意该表达式并没有错误！更好的风格应为6-(-3)。在代码生成中，我们深入考虑了运算符之间的优先级，通过添加括号的方法解决了上述的表达式风格问题，编写实现时提供了较好的可扩展性。
* 支持boolean型表达式的输出，由于C不支持直接输出，所以我们将其转化为了if分支语句，如果原PASCAL-S程序的write调用中，其它类型表达式穿插着boolean类型表达式，可以自动拆分成若干句进行转化。

# 开发环境

* 词法分析器利用LEX编写，所以需要安装flex
* 语法分析器利用YACC编写，所以需要安装bison
* LEX和YACC整合为一个工程，支持跨平台开发
* 整个工程的语言用的是C++，所以需要将LEX和YACC生成的C文件改为CPP文件
* 整个工程在windows上利用visual studio community 2017搭建

# 编译器模块图

我们的编译器在具体实现时，基本符合需求分析和概要设计中的描述，这里给出一个更加详细的模块图。



**由于篇幅限制，更加清晰地版本见附件《编译器模块图.pdf》。**

# 各模块对外接口

## 词法分析

### 词法分析的输入

由用户指定输入文件名inName，默认为PascalProgram.pas，里面保存了PASCAL-S源程序。

### 词法分析的输出

记号序列，以及词法错误信息（如果有词法错误的话）。

词法错误信息保存在vector<string> lexicalErrorInformation中。

## 词法分析和语法分析互联接口

记号序列，以yylex的返回值表示记号编号，属性保存在YYSTYPE类型的结构体指针yylval指向的结构中。

|  |  |
| --- | --- |
| 接口 | 描述 |
| int yylex(); | 语法分析程序每调用一次该程序，可以获得一个记号编号 |
| YYSTYPE yylval; | 语法分析程序每调用一次yylex，获得的记号属性保存在yylval指向的结构体中 |

## 语法分析

### 语法分析的输入

由词法分析提供的记号序列。

### 语法分析的输出

抽象语法树，语法错误信息。

语法错误信息保存在vector<string> syntaxErrorInformation中。

## 语法分析和语义分析的互联接口

|  |  |
| --- | --- |
| 接口 | 描述 |
| \_Program \*ASTRoot | 抽象语法树的根节点指针 |

## 语义分析

### 语义分析的输入

语法分析提供的抽象语法树。

### 语义分析的输出

经过注释的抽象语法树和保存了完整信息的符号表，语义错误信息，语义警告信息。

语义错误信息保存在vector<string> semanticErrorInformation中，语义警告信息保存在vector<string> semanticWarningInformation中。

## 语义分析和代码生成之间的接口

|  |  |
| --- | --- |
| 接口 | 描述 |
| \_Program \*ASTRoot | 经过注释的抽象语法树的根节点指针 |
| \_SymbolTable \*mainSymbolTable | 主符号表指针 |
| \_SymbolRecord\* findSymbolRecord(\_SymbolTable\* currentSymbolTable, string id, int mode=0); | 从指定符号表找到指定标识符的记录，mode=0表示内层符号表未找到时，需要到外层符号表查找，mode!=0表示不需要 |

## 代码生成

### 代码生成的输入

经过语义分析注释的抽象语法树，符号表入口。

### 代码生成的输出

C程序。

# 命令行接口详细设计

## 命令行调用参数说明

我们最终生成的编译器采用命令行调用方式。调用时，可选的参数及其功能如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数接口 | 参数 | 参数功能 | 参数默认值 |
| -inname | [file name] | 指定输入文件名 | PascalProgram.pas |
| -outname | [file name] | 指定输出文件名 | CProgram.c |
| -compiler | [compiler name] | 指定c编译器名,并将c程序编译成可执行文件 | gcc |
| -exename | [exe name] | 指定可执行文件名，自动编译 | CProcess.exe |
| -execute | 无 | 自动执行生成的可执行文件，如果未出现-e、-exename参数，则均按照默认方式进行操作 |  |
| -errorbound | [n] | 指定错误上限，即编译器发现了指定个数的错误后，立即停止运行 | INF |
| -developer | 无 | 输出开发者信息 |  |
| -version | 无 | 输出版本信息 |  |
| -help | 无 | 输出所有命令行参数的帮助信息 |  |

注：中括号表示该参数可有可无

如果出现了-developer –version –help 或者 其它非法参数，则不运行编译器，只显示对应信息。

## 相关函数设计

### 参数帮助文档初始化

首先需要引入一个存储帮助信息的map结构：

1. map<string, string> argumentsExplanation;

是参数名字符串到其解释信息字符串的映射。

* 函数接口

1. **void** argumentsExplanationInit();

* 返回值

无

* 参数列表

无

* 伪代码

1. **void** argumentsExplanationInit() {
2. 下面的映射全部添加到map argumentsExplanation中;
3. 添加"-inname" 到 "-inname [file name]:\t\tdesignate the name of input pascal program, default is \"PascalProgram.pas\"." 的映射;
4. 添加"-outname" 到 "-outname [file name]:\t\tdesignate the name of output C program, default is \"CProgram.c\"." 的映射;
5. 添加"-compiler" 到 "-complier [complier name]:\tdesignate the name of C compiler, default is \"gcc\"." 的映射;
6. 添加"-exename" 到 "-exename [exe name]:\t\tdesignate the name of exe file, default is \"CProcess.exe\"." 的映射;
7. 添加"-execute" 到 "-execute:\t\t\tautomatically run the exe file." 的映射;
8. 添加"-errorbound" 到 "-errorbound [n]:\t\tdesignate the up bound of error number as n, if the pascal2c compiler finds n errors, the compile process will abort, default is INF." 的映射;
9. 添加"-developer" 到 "-developer:\t\t\tinformation about developers." 的映射;
10. 添加"-version" 到 "-version:\t\t\tinformation about version of pascal2c compiler." 的映射;
11. 添加"-help" 到 "-help:\t\t\t\toutput all the explanation abount command line arguments." 的映射;
12. }

### 输出参数帮助文档

* 函数接口

1. **void** outputArgumentsExplanation();

* 返回值

无

* 参数列表

无

* 伪代码

1. **void** outputArgumentsExplanation() {
2. 遍历map argumentExplanations {
3. 输出参数说明和换行;
4. }
5. }

### 将char\*指向的字符串保存到string中

* 函数接口

1. string char2str(**char**\* chs);

* 返回值

string，转化后的字符串

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| char\* chs | chs指向待转化的字符串 |

* 代码

1. string char2str(**char**\* chs) {
2. string res;
3. **for** (; \*chs != 0; chs++)
4. res += \*chs;
5. **return** res;
6. }

### 将char\*指向的字符串转化为整型

* 函数接口

1. **bool** chs2int(**char**\* chs, **int** &num);

* 返回值

bool，true表示转化成功，false表示转化失败

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| char\* chs | chs指向待转化的字符串 |
| int &num | num保存转化后的整型数字 |

* 代码

1. **bool** chs2int(**char**\* chs, **int** &num) {
2. 将num备份在tmp中;
3. num=0;//初始化为0
4. 遍历chs指向的字符串的字符 {
5. **if** (当前字符是数字) {
6. num \*= 10;
7. num += \*chs - '0';
8. }
9. **else** {
10. num=tmp;//转化失败则num不变;
11. **return** **false**;
12. }
13. }
14. **return** **true**;
15. }

### 获取命令行参数

* 函数接口

1. **void** getRunArguments(**int** argc, **char** \*\*argv, string &inName, string &outName, string &compilerName, string &exeName, **int** &errorBound, **bool** &willCompile, **bool** &willExecute);

* 返回值

无

* 参数列表

|  |  |
| --- | --- |
| 参数 | 描述 |
| int argc | 参数及其取值总个数 |
| char \*\*argv | 存储了所有的参数 |
| string &inName | 输入PASCAL-S源程序的文件名 |
| string &outName | 输出C程序的文件名 |
| string &compilerName | C程序编译器名 |
| string &exeName | 可执行文件名 |
| int &errorBound | 错误上限 |
| bool &willCompile | 是否编译C程序 |
| bool &willExecute | 是否运行可执行文件 |

* 伪代码

1. **void** getRunArguments(**int** argc, **char** \*\*argv, string &inName, string &outName, string &compilerName, string &exeName, **int** &errorBound, **bool** &willCompile, **bool** &willExecute){
2. **for** (**int** i = 1; i < argc; i++) {
3. **if** (当前串是"-inname") {
4. **if** (没有下一个串 || 下一个串仍是参数名)
5. **continue**;
6. inName = 调用char2str将第i+1个串转化为string;
7. **if** (不能找到或打开名为inName的文件) {
8. 输出报错信息;
9. 退出程序;
10. }
11. i++;
12. }
13. **else** **if** (当前串是"-outname") {
14. **if** (没有下一个串 || 下一个串仍是参数名)
15. **continue**;
16. outName = 调用char2str将第i+1个串转化为string;
17. i++;
18. }
19. **else** **if** (当前串是"-compiler") {
20. willCompile = **true**;
21. **if** (没有下一个串 || 下一个串仍是参数名)
22. **continue**;
23. compilerName = 调用char2str将第i+1个串转化为string;
24. i++;
25. }
26. **else** **if** (当前串是"-exename") {
27. **if** (没有下一个串 || 下一个串仍是参数名)
28. **continue**;
29. exeName = 调用char2str将第i+1个串转化为string;
30. i++;
31. willCompile = **true**;
32. }
33. **else** **if** (当前串是"-execute") {
34. willCompile = **true**;
35. willExecute = **true**;
36. }
37. **else** **if** (当前串是"-errorbound") {
38. **if** (没有下一个串 || 下一个串仍是参数名)
39. **continue**;
40. 调用chs2int将下一个串转化为数字，保存到errorBound中;
41. **if** (下一个串不能转化为数字){
42. 输出报错信息;
43. 退出程序;
44. }
45. i++;
46. }
47. **else** **if** (当前串是"-developer") {
48. cout << "MilesGO BUPT" << endl;
49. 退出程序;
50. }
51. **else** **if** (当前串是"-version") {
52. cout << "Version 1.0" << endl;
53. 退出程序;
54. }
55. **else** **if** (当前串是"-help") {
56. 调用outputArgumentsExplanation输出帮助文档;
57. 退出程序;
58. }
59. **else** {
60. 报非法参数错误;
61. 退出程序;
62. }
63. }
64. }

### 输出错误

该函数涉及到四个存储错误、警告信息的列表：

1. **extern** vector<string> lexicalErrorInformation;
2. **extern** vector<string> syntaxErrorInformation;
3. **extern** vector<string> semanticErrorInformation;
4. **extern** vector<string> semanticWarningInformation;

* 函数接口

1. **void** outputErrors();

* 返回值

无

* 参数列表

无

* 伪代码

1. **void** outputErrors() {
2. **if** (有词法错误)
3. 输出词法错误;
4. **if** (有语法错误)
5. 输出语法错误;
6. **if** (有语义警告)
7. 输出语义警告;
8. **if** (有语义错误)
9. 输出语义错误;
10. }

### 统计和检测错误的宏

该宏用于所有错误总个数的统计，需要在词法分析添加错误、语法分析添加错误、语义分析添加错误的函数尾部调用该宏。

宏定义如下：

#define CHECK\_ERROR\_BOUND errorCount++;\

if(errorCount>=errorBound){\

cout << "There have been more than " << errorBound << " errors, compiler abort." << endl;\

outputErrors();\

exit(0);\

}

涉及的函数和变量解释如下：

|  |  |
| --- | --- |
| 设计的函数或变量 | 描述 |
| errorCount | 忽略错误类型，统计错误个数的变量，初始化为0, |
| errorBound | 通过命令行参数指定的错误上限 |
| outputErrors() | 4.2.6中介绍的输出错误的函数 |
| exit(0) | 编译器终止运行 |

即每次新增错误时，调用该宏，错误个数+1，然后判断是否打到了errorBound规定的上限，如果达到了，则输出当前已经发现的错误，并终止编译器的运行。

# 运行逻辑

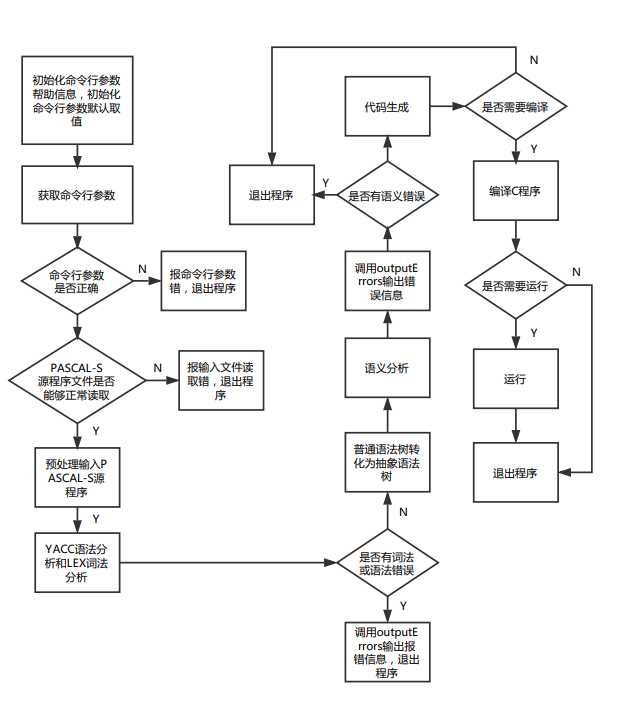
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 源程序的错误情况 | | | 编译器的运行情况 | | | |
| 词法情况 | 语法情况 | 语义情况 | 词法分析 | 语法分析 | 语义分析 | 代码生成 |
| × | \ | \ | √ | √ | × | × |
| √ | × | \ | √ | √ | × | × |
| √ | √ | × | √ | √ | √ | × |
| √ | √ | √ | √ | √ | √ | √ |
| ×：源程序包含对应类型的错误  √：源程序不包含对应类型的错误  \：源程序包含或不包含对应类型的错误 | | | ×：没有运行对应模块  √：运行了对应模块 | | | |

**注：编译器运行情况中，第一种情况，语法分析的运行情况为√。语法分析和词法分析的关系为，语法分析调用词法分析程序，每次返回一个记号序列。即语法和词法分析可以说是同时进行的，如果词法分析中遇到的错误可恢复，那么语法分析程序就可以正确运行，此时词法错误对语法分析程序是透明的；如果遇到词法错误不可恢复的情况，那么语法分析程序和词法分析程序同时停止运行。**

**另外，各部分的错误处理实时的统计错误个数，如果错误个数超过指定的errorBound，则编译器也要终止运行。**

**更详细的运行逻辑可以参考主函数流程图**

# 主函数流程图



**由于篇幅限制，更清晰的流程图见《主函数流程图.pdf》。**

# 实验总结

## 实验中遇到的问题和解决方案

### 重命名恢复策略

一般情况下，出现重命名时，可以直接采用忽略的方法，但是存在一些特殊情况。例如程序的参数定义发生了重命名，此时若不将其加入符号表，则后续如果出现了该程序的调用，则无法检查形参和实参的一致性。

一种可行的方案是进行两遍语义分析，第一遍做定义域检查，第二遍做类型检查，只要第一遍中发现错误，就不再进行第二遍。

还有一种方案就是进行恢复，即对参数进行适当的修改再加入符号表中，考虑到PASCAL-S的标识符中不包含下划线，所以我们可以在参数发生重命名时，在其前面加上一个或若干个（如果重命名多次）下划线，再加入符号表，这样后续调用的时候就可以正常比较形参和实参的一致性了。

另外，主程序名、主程序参数名在发生重命名时，也可以采用这样的策略。

### 抽象语法树

普通语法树要设计翻译方案十分繁杂，所以我们语法分析需输出抽象语法树，删去各种冗余信息，并表示成一棵便于语义分析和代码生成的树形结构。为了减少YACC代码编写的复杂度，我们将AST的生成单独抽离出来，即YACC只生成普通语法分析树（没什么工作量），再编写程序将普通语法分析树转化为抽象语法树。

### 词法分析缓存下一行

词法分析在报错时，应能输出错误所在行，并用^在该行下部指出错误所在位置。这样就需要提前缓存下一行，所以引入了\n.\*的词法规则匹配下一行，并利用yyless(1)退回下一行的所有内容，以便后续再进行具体的词法分析，同时需要在整个PASCAL-S源程序的最前面加上一个换行符，同时也要注意这样预处理后对于行号的影响，即当前行号比初始的实际行号大1。

### 位置信息

如果要想在语法分析中提供更加详细的位置信息，就需要当前语法成分在某一行的具体位置（开始位置和结束位置）。一开始我们自己设置了一个变量tokenPos，自己在词法分析中控制其清零和增长，但是精度不高，且不支持更高层的语法成分的位置信息，例如idlist的开始位置和结束位置。

此时我们查阅相关文档，发现了YACC提供了YYLTYPE结构，支持开始行号、开始列号、结束行号、结束列号四个值，并且能够自动计算各高层语法成分的这些取值，极大的方便了报错提供位置信息，只需要我们在LEX中添加终结符号位置信息的初始化代码即可。

## 存在的问题/待改进的地方

* 可以添加更多的警告信息，以帮助PASCAL-S程序员更好的检查出程序中比较隐蔽的问题，例如当for语句的初值表达式和终值表达式均为整型常量表达式时，可以比较两者的大小，若后者小于前者，则给出警告信息，而非报错信息。
* 应将true和false设置为布尔常量关键字，使得能够定义布尔型常量，且能够对布尔型变量直接赋值为true或者false
* 没有专门处理浮点数字不完整的错误，例如3.和.3的错误，在我们编译器的词法分析中，将返回给语法整数3和符号点，语法中将能检测出错误，但是并不会专门提示浮点数字不完整的错误。
* 语法错误恢复与处理，由于YACC的天生缺陷，很容易出现error burst的雪崩情况。