

**课 程 设 计 报 告**

**题目： SC语言编译器的设计与实现**

**课程名称： 编 译 原 理**

**专业班级： 计算机1012班**

**学 号： U201014592**

**姓 名： 苏 宙 行**

**指导教师： 徐 丽 萍**

**报告日期： 2013年11月13日**

**计算机科学与技术学院**

目 录

1 课程设计任务书 3

1.1 主要内容 3

1.2 任务要求 3

1.3 难度规定 3

1.4 基本要求 3

1.5 参考文献 4

2 问题描述 5

2.1 选题背景 5

2.2 课题概述 5

3 系统设计 6

3.1 方案的基本原理 6

3.2 整体设计思路 6

3.3 系统的模块划分 6

3.4 工作计划 7

3.4.1 第一阶段：搭建开发环境与代码兼容性调整 7

3.4.2 第二阶段：主要数据结构和接口的定义 7

3.4.3 第三阶段：符号表的生成与管理 7

3.4.4 第四阶段：中间代码的生成 7

3.4.5 第五阶段：中间代码的优化 7

3.4.6 第六阶段：目标代码的生成 7

3.4.7 第七阶段：附加功能的实现 8

4 系统实现 9

4.1 数据结构和算法设计 9

4.2 中间代码的设计 9

4.2.1 中间代码的指令集 9

4.2.2 中间代码到目标代码的映射关系 9

4.3 执行效率分析 9

4.3.1 复杂度分析 9

4.3.2 实际运行测试 9

4.4 C语言程序实现的简要说明 9

4.4.1 开发环境 9

4.4.2 函数原型与功能及调用关系 10

4.4.3 源程序清单 10

5 使用说明 11

5.1 概述 11

5.1.1 功能概述 11

5.1.2 依赖软件包 11

5.1.3 运行环境 11

5.2 操作说明 11

5.2.1 软件安装与卸载 11

5.2.2 执行本程序 11

5.3 维护与扩展 12

5.3.1 目录组织 12

5.3.2 调试方法 12

6 总结 13

6.1 系统评价 13

6.1.1 程序测试结果分析 13

6.1.2 本程序的特色 13

6.1.3 本程序的不足 13

6.2 课程设计心得体会 13

参考文献 14

附录1：源代码 15

附录2：英文缩写词 16

# 课程设计任务书

## 主要内容

采用C语言(或C++语言或C#语言或JAVA语言)或程序器lex和yacc，构造一个SC语言的编译程序或集成开发环境（IDE）。

## 任务要求

收集与阅读相关文献资料，确定采用的技术线路，设计系统方案，完成系统实现；提交标准格式打印的《课程设计报告》和包括《课程设计报告》、编译器源程序、编译器目标程序和测试模拟数据文件之光盘。

## 难度规定

参照下列完成难度系数进行评定分数。难度系数值大小表示实现的系统功能的难度大小。

⑴完成词法分析、语法分析、语义分析、中间代码生成、优化和目标代码生成难度系数区间为[1.0]

⑵完成词法分析、语法分析、语义分析和目标代码生成难度系数区间为[0.9,1.0]

⑶完成词法分析、语法分析、语义分析和中间代码生成难度系数区间为[0.8,0.9)

⑷完成词法分析和语法分析难度系数区间为[0.6,0.8)

## 基本要求

⑴SC语言是指C语言的子语言，其要求至少包含的语言成分如下，具体语言范围自行确定；

* 数据类型至少包括char类型、int类型和float类型
* 基本运算至少包括算术、比较、自增自减和复合赋值运算
* 控制语句至少包括if语句和while语句

⑵如果仅完成中间代码生成，编译器的输出为中间代码形式的文本格式；

⑶语义分析至少包括数据类型属性分析。

⑷《课程设计报告》封面和报告内容目录采用统一格式，各小节具体划分自己根据实际情况确定。

## 参考文献

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[2] 张昱等. 编译原理实验教程. 北京：高等教育出版社，2009

[3] Andrew.W.Appel.赵克佳(译). 现代编译原理C语言描述. 北京：人民邮电出版社，2006

[4] 陈火旺等. 程序设计语言编译原理(第三版). 北京：国防工业出版社,2000

[5] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[6] 冯雁等. 编译原理课程设计. 杭州：浙江大学出版社，2007

[7] John E. Hopcroft,Jefferey D Ullman. Formal langualges and their relation to automata, Addison-Wesley Publishing Company,1969

# 问题描述

## 选题背景

学习C语言的语法，能让人了解这门程序设计语言；能够用C语言编写简单的程序，说明自己已经熟悉了C语言；而掌握了C语言的编译过程，则是向“精通C语言”迈出了一大步。设计简单的C语言（C语言的子集）编译器的过程，能让我对C语言甚至其他程序设计语言有更加深刻的理解。与此同时，也可以让这个“编译器”利用编译过程中的语法和语义分析时得出的结果生成一些辅助信息，作为日后供自己在阅读和分析别人的代码时使用的小工具。

由此可见，该题目对我日后的学习和工作将有很大的帮助，所以我选择了该题目，利用暑假时间完成本题的设计与实现。

## 课题概述

SC语言为C语的子集，具体包含多少C语言的语法成分由设计者自行决定，但是数据类型至少包括char类型、int类型和float类型，基本运算至少包括算术运算、比较运算、自增自减运算和复合赋值运算，控制语句至少包括if语句和while语句。我的实现程度见第三章“[系统设计](#_系统设计)”。

对C语言源代码进行了词法分析和语法分析之后，生成中间代码和目标代码。其中中间代码的格式由设计者自行确定，目标代码为x86汇编代码。

另外，对于中间代码，可以使用代码优化技术，进行一定的优化，提高目标代码的生成效率和最终生成的可执行文件的执行效率。

# 系统设计

## 方案的基本原理

程序的整体执行流程为先使用flex对输入文件进行词法分析，然后由bison对解析得到的词法元素序列进行语法分析，每得到一条匹配的语法规则就对该语句进行语义分析，更新符号表并生成相关的中间代码。生成中间代码之后再对其进行一定的优化，最后生成目标代码。

## 整体设计思路

本程序的主要实现语言为C++，以lex和yacc的源代码为辅助。由于符号表中每一项都有一些共有的信息比如名字，而不同符号又有各自独特的信息，所以采用具有多态特性的面向对象程序设计语言在实现上有一定的优势。在本程序中，所有符号的基类为Symbol类，变量、类型、函数等具体项目由基类派生，各自具有与其特性相关的属性。

与符号表相关的主要数据结构一共有三级——符号表栈、符号表、符号。

每个作用域有一个符号表，进入作用域时将符号表压入符号表栈，在作用域中对符号表进行更新和读取操作，离开作用域时将其弹出，回到上层作用域。这样可以通过每次都从栈顶往栈底依次搜索符号表来实现作用域的特性（实际搜索过程将根据搜索对象的不同而使用辅助数据结构实现）。由于实际上所有符号再起作用域之外都是不可见的，所以个人认为没有必要使用树的形式将每个作用域和它的子作用域都保存下来，而是直接使用栈，不可见的作用域直接弹出，不会一直保留到编译程序扫描完整个程序。

每扫描到一个符号，就将它放入符号表中，同时每种符号都有一个辅助的数据结构来保存它们的位置，这样在需要查找特定类型的符号时不用搜索整个符号表并且对每个符号进行类型判断。关于符号的具体内容，参见“[数据结构和算法设计](#_数据结构和算法设计)”。

## 系统的模块划分

## 工作计划

### 第一阶段：搭建开发环境与代码兼容性调整

之前编译原理实验时写的代码全部都在lex和yacc源文件中，而且完全使用C语言编写。由于本次课程设计的工作量远远超过实验的水平，所以我选择了Visual Studio 2012作为开发环境，以便对项目中的大量源文件进行集中管理。同时，我还使用了git作为版本控制的工具（并使用了TortoiseGit的图形化界面）。另外，我还修改了实验时的源代码，使其即可以在Linux中使用gcc编译并运行，又可以在Windows中使用Visual Studio中编译并运行。

### 第二阶段：主要数据结构和接口的定义

本阶段主要是设计并编写相关类的头文件，确定其数据成员与函数成员。同时可以适当编写部分核心的public方法的代码。具体内容可以参考“[整体设计思路](#_整体设计思路)”和“[数据结构和算法设计](#_数据结构和算法设计)”。

### 第三阶段：符号表的生成与管理

实现全部接口函数，并且在lex和yacc源文件中添加相关代码，实现符号表的建立与管理，以及基本的类型检查和未定义/重定义错误的检查。本阶段的工作主要集中在类型、变量、函数和标签（循环语句的第一条语句也算标签）四种语法成分的处理上。

### 第四阶段：中间代码的生成

第三阶段已经成功组织了符号表，但是并没有处理运算的语句。本阶段的主要工作是生成运算语句的中间代码，并且输出相关的类型信息。

### 第五阶段：中间代码的优化

使用DAG等对代码进行优化。（暂未实现）

### 第六阶段：目标代码的生成

将（优化后的）中间代码转化为x86汇编代码。

### 第七阶段：附加功能的实现

统计函数调用关系、全局变量引用情况等。

# 系统实现

## 数据结构和算法设计

符号表

## 中间代码的设计

该中间代码的词法和语法的设计参考了C语言、GCC的RTL层中间语言以及llvm采用的CIL。为了简化描述，我将在下面用IC（intermediate code）代指该语言。

### 整体结构布局

IC一共由五个部分组成，其分布顺序如下：

（由于暂时只支持函数和自定义类型的声明而不支持其使用，故本次课设生成的中间代码只有entry中的入口函数体中的代码）

1. // 文件：IR.def
2. types {
3. 类型定义
4. }
5. functions {
6. 函数声明
7. }
8. constants {
9. 常量定义
10. }
11. variables {
12. 全局变量定义
13. }
14. // 文件：IR.imp
15. entry {
16. 入口函数函数体
17. }
18. modules {
19. 其他函数体
20. }

* 类型定义的格式为：

1. 聚合度 类型名称 {
2. 成员列表
3. }

其中聚合度为struct或union，类型名称为标识符，成员列表的格式与全局变量的定义格式相同。

* 函数声明的格式为：

1. 返回类型 函数名 (参数1, 参数2,…)

其中返回类型和参数为“类型系统”中的类型，函数名为标识符。

* 常量和全局变量定义的格式为：

1. 变量名称:类型 = 值

其中名称为标识符，类型为“类型系统”中的类型，值为整数、实数、用单引号围住的字符、用双引号围住的字符串。

* 函数（包括入口函数）体定义的格式为：

1. 返回类型 函数名 (参数1, 参数2,…) {
2. types {
3. 类型定义
4. }
5. 指令序列
6. }

指令序列参见“指令集”，其他部分与之前提到过的相同。

### 类型系统

IC的所有原子类型如下：

c1; i8,i4,i2; u8,u4,u2; f8,f4; sN(N为正整数)

它们分别代表字符、8/4/2字节有符号整数、8/4/2字节无符号整数、8/4字节浮点数、N字节字符数组（N为将多维数组转化为一维数组后得到的计算结果）。

对于用户的自定义类型，全部转换为原子类型，结构体全部拆解成原子类型，产生的变量用“结构体名\_成员名\_变量名”作为变量名来表示。但是需要事先在types块中进行定义。

由于在语法分析阶段已经进行了相关检查，所以此处不再对数组维数、指针重数做匹配检查。

注意，本次课设中只支持原子类型的运算。

### 变量与常量表示

临时变量（无需定义）：%@name:type

局部变量（无需定义）：%name:type

常量（需先定义）： #value:type

其中，

name为标识符（其集合与C语言的标识符集相同）。

type为“[类型系统](#_类型系统_1)”中提到的类型。

value为立即数，如果为字符串，则为字符串在常量定义中定义的名字，整型和浮点型立即数可以直接出现在程序中。

由于在语法分析阶段已经进行了相关检查，所以此处不再区分常量和变量，static和volatile变量全部看成全局变量，不做优化处理。故此时限定符已经没有存在的必要。

### 指令集

中间代码IC的指令集及指令功能如表1所示。

表 1中间代码指令集-指令功能-汇编指令集映射关系

|  |  |  |
| --- | --- | --- |
| IC指令 | 功能（类C语言表示） | x86汇编指令 |
| MOV r, a | r = a | MOV EDX, a  MOV r, EDX |
| ADD r, a, b | r = a + b | MOV EDX, a  ADD EDX, b  MOV r, EDX |
| SUB r, a, b | r = a - b | MOV EDX, a  SUB EDX, b  MOV r, EDX |
| MUL r, a, b | r = a \* b | MOV EAX, a  IMUL b  MOV r, EAX |
| DIV r, a, b | r = a / b | MOV EDX, 0  MOV EAX, a  IDIV b  MOV r, EAX |
| MOD r, a, b | r = a % b | MOV EDX, 0  MOV EAX, a  IDIV b  MOV r, EDX |
| INC a | ++a | INC a |
| DEC a | --a | DEC a |
| AND r, a, b | r = a & b | MOV EDX, a  AND EDX, b  MOV r, EDX |
| OR r, a, b | r = a | b | MOV EDX, a  OR EDX, b  MOV r, EDX |
| XOR r, a, b | r = a ^ b | MOV EDX, a  XOR EDX, b  MOV r, EDX |
| NEG r, a | r = ~a | MOV EDX, a  NEG EDX  MOV r, EDX |
| LSHL r, a, b | r = a 逻辑左移 b位 | MOV ECX, b  MOV EAX, a  SHL EAX, CL  MOV r, EAX |
| LSHR r, a, b | r = a 逻辑右移 b位 | MOV ECX, b  MOV EAX, a  SHR EAX, CL  MOV r, EAX |
| ASHL r, a, b | r = a 算术左移 b位 | MOV ECX, b  MOV EAX, a  SAL EAX, CL  MOV r, EAX |
| ASHR r, a, b | r = a 算术右移 b位 | MOV ECX, b  MOV EAX, a  SAR EAX, CL  MOV r, EAX |
| EQ r, a, b | r = a == b | CMP a, b  JE EQUAL  MOV r, 0  JMP ENDCMP  EQUAL:  MOV r, 1  ENDCMP: |
| NE r, a, b | r = a != b | CMP a, b  JE EQUAL  MOV r, 1  JMP ENDCMP  EQUAL:  MOV r, 0  ENDCMP: |
| GT r, a, b | r = a > b | CMP a, b  JG GREAT  MOV r, 0  JMP ENDCMP  GREAT:  MOV r, 1  ENDCMP: |
| LT r, a, b | r = a < b | CMP a, b  JL LESS  MOV r, 0  JMP ENDCMP  LESS:  MOV r, 1  ENDCMP: |
| GE r, a, b | r = a >= b | CMP a, b  JL LESS  MOV r, 1  JMP ENDCMP  LESS:  MOV r, 0  ENDCMP: |
| LE r, a, b | r = a <= b | CMP a, b  JG GREAT  MOV r, 1  JMP ENDCMP  GREAT:  MOV r, 0  ENDCMP: |
| LEA r, a | r = &a | LEA ESI, a  MOV r, ESI |
| DREF r, a | r = \*a | MOV EDX, [a] |
| CALL r, f, a1, a2, ... | r = f (a1, a2, ...) | PUSH r  PUSH a1  PUSH a2  ...  CALL f  MOV EDX, [SP]  MOV r, EDX |
| INVOKE f, a1, a2, ... | f (a1, a2, ...) | PUSH a1  PUSH a2  ...  CALL f |
| RET | return | RET |
| RET r | return r | MOV EDX, r  MOV ARGSIZE[SP], EDX  RET ARGSIZE  (注：ARGSIZE为所有参数的大小之和) |
| JCT t, c | if (c==true) PC = t; | CMP c, 0  JNE t |
| JCF t, c | if (c==false) PC = t | CMP c, 0  JE t |
| JMP t | PC = t | JMP t |

表中r、a、b均为变量，t为目标地址，即标签，c为条件。

按照r和a、b的大小和类型分别采用DL/DX/EDX/ST(0)存储中间结果，然后如果a、b为变量且位数比r少，则使用MOVSX进行移动，如果a、b为常量则仍然使用MOV。如果a、b为浮点数，则使用F(I)LD和F(I)ST进行从内存读或写入内存。目前在不做优化的情况下不用考虑EAX,ECX,EDX的内容是否会被覆盖，如果后期需要对寄存器分配做相关优化（比如使用一个寄存器替换某些计数变量）则可能要加入在是杨倩将寄存器的值入栈使用后将其弹出的操作。

## 源代码代码到中间代码的映射

中间代码中直接使用变量名如t1，且除了全局变量之外均无需事先定义，可以直接使用，但是每次使用其后都要有类型标注。

用户自定义标签名称之前加“U\_”，系统会在循环、分支等语句中自动生成标签，此类标签全部命名为“AUTO\_LABEL[N]”，其中[N]为数字。

用户定义的struct全部使用下划线代替“.”，把结构体拆解成独立的变量。（暂未实现）

其他详细映射规则参见中间代码的设计、表1和表2。

表 2逻辑运算的IC实现方式

|  |  |  |
| --- | --- | --- |
| IC指令 | 功能（类C语言表示） | x86汇编指令 |
| NEQ r, a, 0  JCF FALSE, r  NEQ r, b, 0  FALSE: | r = a && b | 参考NEQ和JCF |
| NEQ r, a, 0  JCT TRUE, r  NEQ r, b, 0  TRUE: | r = a || b | 参考NEG和JCT |
| EQ r, a, 0 | r = !a | 参考EQ |

## 中间代码到目标代码的映射

中间代码中的局部变量全部用堆栈栈顶指针SP加上偏移量来表示。在作用域内第一次出现时使用PUSH命令压入栈，离开作用域时使用POP命令弹出。

全局变量使用堆中的空间静态分配。

对于中间代码中的基本类型，均按大小对应成寄存器的AL、AX或EAX。

详细内容参见表1和表2。

# 使用说明

## 概述

### 功能概述

输入一个或多个C语言源程序，然后经过词法和语法分析，生成中间代码。然后再根据中间代码生成以x86汇编为目标代码的源程序。

### 依赖软件包

GNU Flex ：词法分析工具LEX的开源版本

GNU Bison ：语法分析工具YACC的开源版本

GNU g++ ：UNIX系统中C++语言编译器的开源版本

Visual Studio 2012 ：Windows下的集成开发环境

（其中g++和VS2012二选一）

### 运行环境

取决于编译环境，经过本人测试的系统有CentOS-6.4-x86\_64和Windows-7-Ultimate-x86\_64。

## 操作说明

### 软件安装与卸载

对于编译好的可执行文件，可以直接使用。

对于源程序，在获取了[依赖软件包](#_依赖软件包)之后，执行build.sh脚本（g++）或者在VS2012中编译并运行本程序。

卸载本程序只需将程序所在目录删除即可。

### 执行本程序

在shell中执行br.sh或run.sh脚本，或者自行执行编译好的可执行文件。注意，程序至少需要传入两个命令行参数，第一个是输出文件的目录，从第二个开始是待编译的文件的路径。

## 维护与扩展

本程序完全开源，如果在使用过程中遇到bug可以根据源代码自行修改完善，或者联系作者（邮箱suzhouxing@qq.com）。源代码见“附录1：源代码”。

### 目录组织与文件清单

**-src目录（软件源代码）**

---main.cpp（入口函数和初始化函数所在的文件）

---lex.l（词法分析源文件）

---yacc.y（语法分析源文件）

---Symbol.h（符号类头文件）

---Symbol.cpp（符号类实现文件）

---SymbolTable.h（符号表类头文件）

---SymbolTable.cpp（符号表实现文件）

---CodeGenerator.h（代码生成器头文件）

---CodeGenerator.cpp（代码生成器实现文件）

**-VisualStudio2012project目录（VS2012项目配置文件）**

---VisualStudio2012project.sln（解决方案文件）

---ClassDiagram.cd（类图文件）

**-doc目录（文档）**

---程序基本流程.vsdx（基本流程图）

---编译原理课程设计报告.docx（项目说明及新的体会）

**-bin目录（可执行文件）**

---CCompiler（linux下的可执行文件）

---CCompiler.exe（windows下的可执行文件）

**-TestCase目录（测试样例和测试样例的输出目录）**

---in.txt（输入样例）

---IR.imp（中间代码函数实现文件）

---IR.def（中间代码类型、变量、函数定义文件）

---target.asm（x86汇编目标代码）

### 调试方法

使用Flex和Bison生成相应的源文件和头文件之后，可以通过使用‘VisualStudio2012project’目录下的VS2012解决方案进行编译与调试。默认情况下，已经在项目属性中指定了‘../TestCase/in.txt’为命令行参数用于自动输入测试文件；并在“配置属性 -> C/C++ -> 预处理器 -> 预处理器定义”中定义了\_CRT\_SECURE\_NO\_DEPRECATE宏，以便阻止废弃部分POSIX函数的错误提示。请根据自己的需要对相关设置进行修改。

另外，本程序也可以在Linux下使用g++进行编译，然后用GNU的调试工具或者向stderr或日志文件输出调试信息的方式进行调试。

# 总结

## 系统评价

### 程序测试结果分析

### 本程序的特色

能够识别大多数C语言语法，并且输出对应语句是什么语法。同时声称的中间代码是由我借鉴GCC与LLVM的中间表示自行设计的语言。加入函数引用关系统计、变量引用统计等附加功能（暂未实现）。

在遇到语法错误时，有比较详细的错误信息提示。

### 本程序的不足

没有充分利用C++的标准库。符号表的组织基本上是使用最基本的vector类型实现的，然后自己实现标识符的重复/冲突检测。实际上这些特性在标准库的set模板中已经被实现了，它本身所有元素就是无重复的。另外，采用键值对的方式的map，可能比我目前使用的四种符号继承自Symbol的方式更方便且高效，因为实际上目前Symbol里面也只有名字（标识符）信息和作用域信息。实际上，把记录各类符号的vector变成一个（二维的）像符号表栈一样的数据结构，栈中的元素类型为map，则作用域信息完全可以省略掉。

在我所使用的C语言编译器中（很有可能也是标准C中），子作用域的名称可以覆盖更大作用域的名称，也就是说，上级作用于中可能有一个名为a的变量或类型或goto标签甚至函数，则当前作用域仍然可以有用a命名的变量或类型或goto标签，而在我目前的设计当中，由于在词法分析时不确定标识符具体是哪种类型的话，会导致语法分析阶段出现规约/规约冲突（这是这个项目的“历史遗留问题”，参考doc目录下的“编译原理实验报告”文档）。于是，我不得不在词法分析阶段扫描符号表，一旦发现有同名的标识符即按对应类型返回给yacc。这种过早地确定标识符类型导致语法分析阶段遇到的token不是期望的ID，而且其他的如DEFINED\_TYPE，于是在定义同名变量（或类型、标签）时会报语法错误。结果，虽然不同作用域中的相同ID标识的符号并不会互相影响，却会被本程序报告为错误。

如果出现表达式“4+5+a+1+2+3”，本编译器不能将其变成“a+15”，也不能变成“9+a+6”，而是变成5个加运算。之所以没有实现相关优化是因为不同类型进行运算太难处理，而且不知道有什么办法能实现遇到常量就优先移进，遇到变量就优先规约（或者说左变量右常量时移进，否则规约）。在多种优先级的运算混合出现时情况更加复杂。

没有进行代码优化，而且生成的中间代码和目标代码之间存在一定程度的冗余，导致程序中会有多余的跳转。同时，临时变量的分配也可能存在冗余，比如进行计算之后马上进行赋值，本来可以直接将结果放入结果的变量，但是中间代码中需要经过一个中间变量。同时寄存器变量使用前无法保证是否在其他地方被使用，所以每条中间代码都需要进行寄存器的压栈和弹出操作。

## 课程设计心得体会

3.4.5 Providing a Structured Semantic Value Type

3.4.7 Data Types of Values in Actions

3.5 Tracking Locations

3.6 Named References

3.7.6 Freeing Discarded Symbols

5.7 Mysterious Conflicts

%token <type> terminal

%type <type> non-terminal

g++的枚举类型中的枚举成员居然没有作用域的限制而是直接像宏定义一样使用，后来才知道需要添加-std=c++0x选项[5]。

本来以为在下一次调用yylex()之前，yytext会一直保持当前的状态，没想到在对应规则的action结束之后，yytext会发生变化（增加了一些接在匹配串之后的字符），导致直接将yytext赋值给yylval会导致之后引用该字符串时出现错误！所以，一般这种情况都要将字符串另外保存，但是又带来了内存泄露问题[6]。

Yacc源文件中有很多地方使用了动态分配的内存，如果正常执行，这些内存会在不需要时被释放掉。可是，如果分析过程中出现了语法错误，将有可能会导致之后负责释放该内存的代码没有被执行（例如，由于错误恢复机制清空了其后的一段输入缓冲区，所以没有进入对应规则的action）。一开始想使用%destructor，但是由于规则之间传递数据都是直接使用指针的，并没有进行深拷贝（出于效率考虑），所以前面的规则用完了的内存区域实际上之后还在被其他规则使用，而且此方法可能会有重复释放的问题。暂时的解决方案是不做处理，遇到语法错误就终止扫描，这样相当于一发生内存泄露程序就终止了。

开始写各种符号的析构函数时，各种各样的运行时错误就开始冒出来了，总是有各种各样的符号在还在被别的符号引用的时候就被析构了，导致引用了无效的内存空间，或者已经被析构过一次结果又被其他引用它的符号重复析构。我采用了很多种方法解决这些问题，最后有两个方案都可以修复这个问题。于是，在这里，我第一次使用了git的分支功能。

在暑假时我完成了本次课设的前三阶段（见3.4工作计划）。但是开学之后因为忙于保研等事情，课设的进度停滞了7个星期，第8周重新回复工作进度的时候，因为对flex和bison的文档已经遗忘了很对，所以又开始遇到了一些莫名其妙的错误。

由于暑假结束之前最后一次修改没有在git中确认并填写相关的改动信息记录，导致我在又做了一些修改之后运行程序输出了大量出错提示。通过tortoise git的代码对比功能，以及我在程序中编写的调试信息输出，我发现是常量无法被识别。原来，我修改了词法分析中跟常量相关的部分代码，而且忘记用花括号将表示常量的名称括起来。加上花括号马上恢复了正常。

在开始编写将表达式翻译成中间代码的部分时，yacc文件在编译时出现了很多警告：“type clash on default action”，但是我在对应行甚至连规则都没写，问什么会有类型冲突呢？后来经过一番搜索，发现出错的原因跟警告中提到的默认操作有关[10]——每一条语法规则都会默认执行“$$=$1”的操作，如果$$或$1中有一个在之前声明的时候就规定了其类型或者两者规定的类型不相同的话，就会发生类型冲突。

参考文献

1. <http://stackoverflow.com/questions/14386/fopen-deprecated-warning>
2. <http://flex.sourceforge.net/manual/Code_002dLevel-And-API-Options.html>
3. <http://flex.sourceforge.net/manual/Options-Affecting-Scanner-Behavior.html>
4. <http://www.gnu.org/software/bison/manual/bison.html>
5. <http://stackoverflow.com/questions/5188554/my-enum-is-not-a-class-or-namespace>
6. <http://flex.sourceforge.net/manual/A-Note-About-yytext-And-Memory.html>
7. <http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/>
8. <http://llvm.org/docs/LangRef.html>
9. <http://www.cse.iitb.ac.in/grc/gcc-workshop-09/downloads/gccw09-gimple.pdf>
10. <http://stackoverflow.com/questions/4605990/what-is-wrong-with-this-yacc-file>
11. <http://gcc.gnu.org/onlinedocs/gccint/RTL.html>

附录1：缩略词对照表

|  |  |
| --- | --- |
| 缩写词 | 全称 |
| AST | Abstract Syntax Tree |
| CIL | Common Intermediate Language |
| LLVM | Low Level Virtual Machine |
| IR | Intermediate Representation |
| PC | Program Counter |

附录2：源代码

1. //Filename:HuffmanCompression.c
2. //