摘 要

本文是关于LALR(1)语法分析生成器马修的设计与实现，本文首先介绍了什么是语法分析器，语法分析器有什么作用以及马修命名的由来，随后对课题研究的目的及意义进行了简单的概述。目前国内外关于语法分析生成器的研究已经非常成熟，国内外都有许多著名的LALR(1)语法分生成析器，各分析生成器提供给开发者的书写规格也略显差别，关于生成语法分析驱动程序的方式各有千秋。研究此课题的目的在于深入了解LALR(1)语法分析生成器的实现，研究内部实现的算法，扩展分析生成器的功能，还有增强分析器后端输出的灵活性等。随后分析了国内外的研究现状，并举出了几个国内外的研究成果，比如著名的VPGE和YACC还有LEMON等。接着就是系统设计的总体概述，在系统设计的总体概述中首先是对需求进行了分析，在需求分析中对已有的分析生成器进行了分析，并对改进后的分析生成器进行了分析。随后分别以性能、功能、灵活度等角度列举出了系统的设计目标。接着是开发工具的介绍以及文法书写的一些规则的介绍。最后给出了系统结构图。系统设计的总体概述之后就是具体实现。在具体实现的内容中首先是对消息处理的方式进行了说明。接下来关于其它模块功能的说明都是根据输入流经过各个阶段的形式进行的。比如词法分析过后就是语法分析，语法分析过后到分析状态的建立。中间掺插着对各个阶段的处理说明和使用的算法说明。在算法说明的过程中，某些特殊的算法还附带了图片或表格。具体实现过后就是测试，测试对几个主要的功能模块进行。分别对命令行，预处理，错误提示等模块进行测试，并给出了测试结果的简要说明。

关键词：LALR(1)；语法分析生成器；YACC;

**Abstract**

This article is about the LALR (1) grammar analysis of the design and realization of the generator to Mathew, this paper first introduces what is the syntax analyzer, what role does the parser and Mathew the origin of the name, then the research purpose and meaning of brief overview.Study of syntax analysis generator at home and abroad has been very mature, there are many famous at home and abroad LALR (1) grammar points generated parser, the analysis of the generator for developers writing specification is slightly different, on generative grammar analysis of the driver come in different ways.Research aimed at understanding the subject LALR (1) grammar analysis of the realization of the generator, the internal implementation of the algorithm, extended analysis the function of the generator, and strengthen the analyzer back-end output flexibility, etc.Then analyzes the research status at home and abroad, and several domestic and foreign research results, such as the famous VPGE and YACC and LEMON.Followed by general overview of the system design, the overall overview of the system design is first analyzed the demand, the demand analysis in the analysis of the existing generator are analyzed, and the analysis of the improved generator are analyzed.Then on performance, functionality and flexibility lists the design goals of the system.Followed by the introduction of development tools and the introduction of some of the rules for writing grammar.In the end, the system structure was presented.After the general overview of the system design is the specific implementation.In the context of a specific implementation first is the way of message processing.The following instructions regarding other modules are based on the input stream through various stages of the form.Is grammar analysis after such as lexical analysis, syntax analysis to the analysis of the status after the build.Intermediate mixed with the various stages of processing instructions and the use of the algorithm.In the process of algorithm shows, some special algorithm also comes with a picture or form.Concrete implementation is after the test, the test of several major functional modules.Respectively on the command line, pretreatment, error message module, such as testing, and presents a brief description of the test result.

**Key words**：LALR(1); Syntax analysis generator; YACC;

目 录

[1 绪论 1](#_Toc420830057)

[1.1 课题的目的背景及意义 1](#_Toc420830058)

[1.2 国内外的研究现状 1](#_Toc420830059)

[1.2.1国内的研究现状 1](#_Toc420830060)

[1.2.2国外的研究现状 1](#_Toc420830061)

[2 设计的总体概述 2](#_Toc420830064)

[2.1 需求分析 2](#_Toc420830065)

[2.2设计的目标 3](#_Toc420830067)

[2.3开发工具简介 3](#_Toc420830067)

[2.4语法文件介绍 4](#_Toc420830067)

[2.4.1文法书写规则 4](#_Toc420830078)

[2.4.2声明部份 4](#_Toc420830078)

[2.4.3规则部份 6](#_Toc420830078)

[2.5系统结构图 6](#_Toc420830067)

[3 系统具体实现 7](#_Toc420830064)

[3.1消息处理 7](#_Toc420830067)

[3.2命令行 7](#_Toc420830067)

[3.3符号表 8](#_Toc420830067)

[3.4预处理 8](#_Toc420830067)

[3.5词法分析 8](#_Toc420830067)

[3.6语法分析 9](#_Toc420830067)

[3.7插入动作 10](#_Toc420830067)

[3.8开始符号的确定 10](#_Toc420830067)

[3.9计算FIRST集 10](#_Toc420830067)

[3.10分析状态的建立 11](#_Toc420830067)

[3.11计算FOLLOW集 11](#_Toc420830067)

[3.12装配动作链表 12](#_Toc420830067)

[3.13生成LALR(1)语法分析驱动程序 12](#_Toc420830067)

[3.14二维数组的线性化压缩 13](#_Toc420830067)

[3.15特殊标识 15](#_Toc420830067)

[4 系统测试 16](#_Toc420830096)

[4.1 系统测试简述 16](#_Toc420830097)

[4.2 系统测试内容 16](#_Toc420830099)

[4.2.1命令行输入测试 16](#_Toc420830100)

[4.2.2预处理功能测试 16](#_Toc420830101)

[4.2.3语法文件分析流测试 17](#_Toc420830102)

[4.2.4语法文件分析出错测试 17](#_Toc420830103)

[4.2.5各阶段显示功能测试 17](#_Toc420830104)

[4.2.6状态转换信息测试 18](#_Toc420830105)

[4.2.7语法分析驱动程序测试 19](#_Toc420830106)

[4.2.8切换平台测试 20](#_Toc420830106)

[5 结论 20](#_Toc420830107)

[谢辞 22](#_Toc420830108)

[参考文献 23](#_Toc420830109)

# 绪论

## 课题的目的及研究意义

语法分析生成器是将拥有语法规则的文件作为输入，输出特定语法分析器的分析器生成工具。语法分析器通常是以编译器或解释器的组件的形式出现，它的作用是根据输入单词的序列进行语法检查、并建立语法分析树、抽象语法树(AST)等层次化的数据结构。语法分析作为《编译原理》中比较重要的一部份，目前最流行的分析器就是lalr(1)型语法分析器，很多关于编译器的书籍都只是将语法分析的算法提及了一下，并没有去深入剖析语法分析生成器或者是给出语法分析生成器的具体实现。本人首先研究了著名的yacc，发现yacc比较庞大。后来本人接触了一个叫做lemon的lalr(1)型语法分析生成器，这是一个短小精悍的分析器生成器，由美国计算机专家Richard Hipp开发。本人在阅读其源代码时发现lemon有些地方的设计很精妙，但lemon还是比较粗糙的，并且作者提出了一些思想还未在源代码中实现。因此本人就去计划实现一个更强大更完善的lalr(1)语法分成器，本人将其命名为马修(Mathew)。本人为何要将它命名为马修(Mathew)。马修这一名字的由来源于一部漫画《魔力女管家》。马修也被设定为其中的重要角色“星神”，她是创造机器人V1046-R Mahoro的人工智能信息结合体，这与yacc(Yet Another Compiler Compiler)与编译器的关系相似。一个好的语法分析生成器应该要确保生成的正确性，如果生成的分析器不正确那么将无法使用。生成的语法分析器的执行效率也是非常重要的，这关系到编译器和解释器的执行效率，生成速度同时也是衡量一个语法分析生成器好坏的标准。生成后端程序的灵活性是衡量一个语法分析生成器是否强大的标准，能够生成适应越多平台的语法分驱动程序，就证明这个语法分析生成器是非常强大的。正如[斯坦福大学](http://baike.baidu.com/item/%E6%96%AF%E5%9D%A6%E7%A6%8F%E5%A4%A7%E5%AD%A6)计算机科学系的荣誉退休教授，算法和[程序设计技术](http://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E6%8A%80%E6%9C%AF)的先驱者Donald E. Knuth所言“一个好的程序应该如一部文学作品一般。”马修的设计和实现正如打造一部文学一般追求更加完美。从马修的源代码中可以看出本人是花费了大量的心思去制作的，每一个细节本人都尽力做到更完美。

## 国内外的研究现状

### 国内的研究现状

国内也有不少优秀的LALR(1)语法分析生成器，比如VPGE。VPGE(Visual Parser Generation Environment)是一个可视化语法分析器集成开发环境,除了具有良好的界面和强大的调试功能,其生成速度达到并超过公认的分析器Bison，所能处理的文法规模也优于Bison。

### 国外的研究现状

著名的LR(k)文法是Donald E. Knuth在对自左至右、自底向上的移进一归约分析进行了深刻剖析的基础上，经过高度概括和集中以后发明的，它表示具有从左(L)到右(R)的分析而向前看k个符号。而LALR（1）简化了LR(1)，采用合并同心集方法，大大缩小了LR(1)的自动机状态规模，功能介于SLR(1)与LR(1)之间，但应用于目前的语言，已经绰绰有余。在此本人在网上查找到了几个国外著名的语法分析器生成，其中也包括LL(1)型的语法分析器。

ANTLR—A，其前身是PCCTS，它为包括Java，C++，C#在内的语言提供了一个通过语法描述来自动构造自定义语言的识别器（recognizer），编译器（parser）和解释器（translator）的框架。ANTLR可以通过断言（Predicate）解决识别冲突；支持动作（Action）和返回值（Return Value）。

LEMON是一个LALR(1)语法分析器生成器。与GNU Bison和Yacc不同。为了减少编写代码的错误，它使用了一种不同的语法。Lemon使用了一种更为高级的分析引擎（LALR的好处就是产生的状态表比较小），运行速度快，并且该引擎是可重入的和线程安全的。更进一步的，Lemon实现了能够消除资源泄漏的特性，适合于要求长时间稳定运行的程序。

YACC(Yet Another Compiler Compiler)，是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。Yacc生成的编译器主要是运行于C语言平台的语法解析器（Parser），需与词法解析器Lex一起使用，再把两部份产生出来的C程序一并编译。yacc起初只在Unix系统上才有，但现在已普遍移植于Windows及其他平台。Yacc采用的也是LALR(1)语法分析方法。Yacc最初由AT&T的Steven C. Johnson为Unix操作系统开发，后来一些兼容的程序如Berkeley yacc，GNU bison，MKS yacc和[Abraxas](http://www.baike.com/sowiki/Abraxas?prd=content_doc_search) yacc陆续出现。其中的Berkeley Yacc 一般认为是目前最好的yacc变种。与bison相比，避免了对特定编译器的依赖。

SJPT是一个分析工具包支持包括自顶向下(LL(1))和自底向上(LR(0), SLR(1), LR(1) and LALR(1))。该工具包同时支持为所有自底向上的分析法生成Java剖析器

CUP是用于构建在java上的LALR(1)分析器，它是由 [C. Scott Ananian](http://cscott.net/), Frank Flannery, Dan Wang,[Andrew W. Appel](http://www.cs.princeton.edu/~appel/)和[Michael Petter](http://www2.in.tum.de/hp/Main?nid=27)开发，与词法扫描器JFlex联用特别适合，JFlex也可以用于BYacc或ANTLR，也可以作为一个独立的工具使用。

GOLD 是一个自由的分析系统，您可以用它来开发自己的编程语言,脚本语言和解释器。GOLD可以用于许多编程语言和在多个平台上。

# 设计的总体概述

## 需求分析

笔者对一些lalr(1)语法分析生成器展开了研究，其中大多数语法分析生成器生成后端的灵活性都较低，有的仅仅只生成一种平台的语言，大家熟知的yacc也只能生成在c语言平台上运行的语法分析驱动程序。因此本人引入了透明化输出。只需要修改模板文件的一些输出规范就可以生成其它语言平台的语法分析驱动程序。这种方式让不同平台的编写者拥有更灵活的生成方式，本人提供给开发者们更多的想象空间去构建适应自己语言平台的语法分析器驱动器。开发者不仅需要注意的是如何编写语法文件，同时还应该注意怎样去定制模板文件中的输出规范。通过编写出不同生成规则的模板文件，就可以让马修生成拥有自己特色的语法驱动程序。对于一些特殊语言，就看开发者的发挥了，如果无法直接生成特殊语言平台的完整程序，如何利用模板文件的生成规则去减少工作量也是种捷径。本人在研究lalr(1)分析器lemon的源代码时发现，lemon的设计思路很好，但是存在许多可以升级改进的地方。因此本人从性能的角度改进并且替换了许多算法，将yacc中与lemon的特性合并，加入透明化输出等特性去打造一个更优秀的lalr(1)语法分析生成器。

## 设计的目标

(1) 拥有更高的生成效率，马修进行了大量的算法优化，在生成语法分析驱动程序时能够在短暂的时间内生成目标程序。

(2) 功能齐全，各阶段清晰。马修在生成语法分析驱动程序的时候提供了许多功能选项，这些选项控制着各个处理阶段的一些产物。开发者可以根据这些选项订制需要获取的内容，比如显示阶段产物的信息和生成哪些文件或不生成哪些文件。

(3) 马修具有错误跟踪功能和错误发生后的重同步功能。马修对于一些语法规则的出错部份给出的做法是先显示错误的提示信息，然后在出错处进行重同步并且继续对语法文件进行分析。

(4) 马修继承了一些语法分析生成器的优秀功能，马修不仅继承了LEMON的许多特性，还继承了YACC的一些特性，比如访问符号值时既可以以LEMON的别名方式访问也可以以YACC的$number的方式访问。马修引入了LEMON没有的，而YACC有的插入动作。马修也支持YACC提供的继承属性，并在其中进行了扩展。

(5) 马修具有输出透明化的性质，订制不同的模板规则文件可以生成不同平台的语法分析驱动程序。

## 开发工具简介

1. Qt Creator 4.0.3 (Community)

Qt Creator是跨平台的轻量级[集成开发环境](http://baike.so.com/doc/5602996-5815603.html)(IDE)。此 IDE 能够跨平台运行，支持的系统包括 Linux(32 位及 64 位)、Mac OS X 以及 Windows。

1. GCC 4.9.2

GCC（GNU Compiler Collection，GNU编译器集合），是一套由 GNU 开发的编程语言编译器。[gcc](http://p8.qhmsg.com/t016f339e7bf8f48668.png)它是一套GNU编译器套装以GPL许可证所发行的自由软件，也是 GNU计划的关键部分。GCC原本作为GNU操作系统的官方编译器，现已被大多数类Unix操作系统（如Linux、BSD、Mac OS X等）采纳为标准的编译器，GCC在微软的Windows下的移植版本叫[MinGW](http://baike.so.com/doc/5464973.html)。

1. GUN Make

GUN Make 是一个管理项目生成计划的工具。

## 语法文件介绍

### 文法书写规则

语法文件主要由两大部份构成，声明部份和规则部份。声明部份与规则部份的书写可以相互间插进行。关于文法的书写规则如下。

终结符：

必须以大写字母开头后接“\_”或字母和数字。

非终结符：

必须以小写字母开头后接“\_”或字母和数字。

别名技术：

必须用括号扩起且以$开头后面接以字母开头的标识符。

### 声明部份

马修声明部份的书写方式是以%号开头后面接关键字的形式出现的，其余的形式将被识别为产生式。

特殊声明符

%accept %code % destructor(NT)

%default\_code % fallback % include

%left %nonassoc %nonterminal\_type

%nonterminal\_destructor %parse\_name %parse\_failed

%right %start %syntax\_error

%type(NT) %token\_type %token\_destructor

%token\_prefix

对上面的特殊声明符的使用说明如下：

%accept {…}

表示分析进入接受状态时执行的动作。

%code {…}

在语法驱动程序最后附加的代码，可以将分析驱动程序入口函数写在这里，提供外部调用。

%destructor {...} nt\_1[ nt\_2].

设置非终结符nt\_1，nt2的销毁动作。

%default\_code {…}

给那些没有动作的产生式赋予的默认动作。

%fallbak T\_F T\_1 T\_2.

将终结符T\_1，T\_2的回退属性设置为T\_F，当终结符T\_1，T\_2不被接受时可以回退成终结符T\_F继续分析。

%include {…}

在语法驱动程序最前面附加的代码，可以再次包含一些头文件。

%left T\_1 T\_2.

优先级具有左结合性的非终结符举例，同一高度声明的终结符具有相同优先级，越往下优先级越低。

%nonassoc

设置终结符不具备结合性。

%nonterminal\_type <type>

设置非终结符的默认类型。

%nonterminal\_destructor {…}

设置默认非终结符的销毁动作。

%parse\_name @Parse

设置驱动文件的分析入口函数名。

%parse\_failed {…}

分析失败执行的动作。

%right T\_1 T\_2.

优先级具有右结合性的非终结符举例，同一高度声明的终结符具有相同优先级，越往下优先级越低。

%start nt\_1

设置开始符号。

%syntax\_error {…}

语法错误时执行的动作。

%type <type> nt\_1 nt\_2.

将非终结符nt\_1 和 nt2的值的数据类型设置为type。

%token\_type <type>

设置终结符的默认类型。

%token\_destructor {…}

终结符的默认析构函数。

%token\_prefix @TK\_

为终结符设置前缀。

非终结符需要统一的数据类型否则分析入口函数Parse就要多个，分别适应不同参数，这就需要使用重载来实现，但这太局限了。于是本人给出的提议是对不同类型的非终结符进行析构的方法是建立一个数据结构，成员type代表具体的数据类型。析构接口中再利用type进行类型判别调用相应的析构函数。马修对于终结符与非终结符的默认析构函数是分开的，即使类型相同也不不会交叉调用相同的默认析构函数。

### 规则部份

马修对于规则部份可以归结为：

非终结符 –> [非终结符|终结符|{插入动作}][(别名)] . [{动作}]

产生式：

expr -> expr PLUS expr.

expr -> expr LPAREN expr RPAREN.

expr -> VALUE.

加入动作的产生式:

expr($A) -> expr($B) PLUS ($plus){

printf(“just enter the addition symbol.”);

}expr($C).{$<int>A=$B+$C;}

马修对于规则部份的书写方式还是挺容易理解的，产生式左边为以小写字母开头的非终结符，如果有别名的话需要用括号括起来这点参考了LEMON里的别名书写方式。马修为了更突显别名与普通符号的区别，就规定要为符号设置别名不仅要用括号括起来还需要以“$”特殊字符打头。在同一个产生式中可以为符号设置多个别名，但马修只识别最新定义的别名。马修关于文法的书写规则与大多数语法分析生成器和教科书的文法书写方式都非常相似，这样也便于开发者理解。

## 系统结构图

本系统的系统结构图，如图2-1。

命令行

预处理

消息处理

词法分析

语法分析

建立状态

动作表

目标代码

图2-1 系统结构图

# 系统具体实现

## 消息处理

消息提示为系统必不可少的部份，马修为消息处理提供了适应不同地区语言的处理方案。马修将输出语句的提示集中在了mathew .c error.h output.h三个文件中，只需将相应提示字符串替换为相应地区的语言并重构马修，就可以生成适应相应地区语言的目标程序。这样处理消息的方式是本人参考了crowbar语言提供的消息处理机制，本人认为将消息集中翻译是种很不错的方法，于是本人就引进了这种方式。本人曾想过对消息的提示集中存储在文件中，启动程序时再动态载入，这样更方便平台转化。但本人并没有这样做，因为这样做是降低执行效率的行为，每次启动时动态载入区域语言对于类似于游戏程序这种生存周期较长的程序来说是可行的，但对于像马修这种生存周期较短的程序来说这是不可行的。

出错消息处理主要依赖文件为error.h 和 error.c，error.h分类枚举出了在分析过程中所有的错误提示。

ERROR(ERROR\_HEAD = 0, "dummy")

ERROR(OUT\_OF\_MEMORY, "内存不足\n")

…

ERROR(ERROR\_TAIL, "dummy")

错误由错误头和错误尾包括，头尾是为了验证消息格式是否正确。error.c 提供了验证错误消息格式和输出错误提示等功能，并提供了四个接口函数：

void SetErrorStream(FILE \*stream); /\* 错误输出流的订制 \*/

FILE \*GetErrorStream(void); /\* 错误输出流的获取 \*/

void PrintFailureNote(int sp\_count); /\* 错误位置标注 \*/

void Error\_fprintf(Error id, ...); /\* 错误输出提示接口函数 \*/

输出错误只需要填写相应的错误ID和一些附加内容就可以输出错误消息提示，例如读取文件出错：

Error\_fprintf(CANNOT\_READING\_FILE,psp->filename);

输出消息的处理基本和错误消息处理的方式一致只不过订制的输出流不同而已。

## 命令行

命令行输入形式为:

mathew 语法文件 [控制选项 | 宏定义 | 变量赋值 | 扩展函数]

选项列表：

mathew gram.y -?

-a 输出所有内容

-b 只打印核心项目

-c 不压缩动作表

-d 不生成驱动文件

-e 输出被使用的产生式中能推出空(epsilon)的符号

-f 输出非终结符的首符集

-r 重新打印语法文件(.y)且不打印动作

-m 输出附加头文件

-n 输出未使用的产生式

-q 不生成gram.out文件

-s 输出分析表的状态数目

-u 输出被使用的产生式

-v 输出版本信息

tpl=<string> 指定模板文件名

macroinc=<integer> 宏数组每次预留空间大小

echo(<string>) 原样输出指定参数

<macro> 定义%ifdef宏变量

例如：

mathew gram.y –v –r Win32 –m tpl=java\_tpl.java echo(\“asda”) echo(123)

表示输出版本信息，输出非终结符首符集，定义宏变量Win32 , 输出附加头文件例如gram.h, 将模板文件设定为java\_tpl.java，原样输出asda”和123，echo函数调用接口的演示，提供了对未来的扩展。

## 符号表

对于符号表的处理，马修与LEMON一样使用的是哈希表。但马修改进了LEMON对于符号表的处理，马修删除了符号表的冗余代码，将共性代码保留提高了代码的可重用性，马修在table.c文件里建立了用于存储字符串、符号、状态、项目的符号表静态变量，它们使用统一的基底函数进行插入和查找。

## 预处理

词法分析之前药对语法文件进行预处理，预处理是利用命令行选项提供的宏定义对语法文件进行修饰。马修提供的预处理关键字为%ifdef ，%ifndef 并且以%endif 结束。马修规定预处理命令后必须跟一个空格，预处理除了出现在注释和字符串常量之外都将会被处理。

## 词法分析

对于词法分析方面马修使用的是手动编写的词法分析程序，马修并没有沿用LEMON的的词法分析程序，因为LEMON的词法分析程序和很多教科书给的词法分析一样是比较粗糙的，马修参考了UCC中给出的词法分析程序，这种词法分析程序的设计规范而且效率高，UCC给出的词法分析程序并非使用传统的分支判断语句，而是使用字符映射的方式进行单词获取。因此本人采用了UCC中给出的思想应用于马修的词法分析程序中。词法分析还引入了状态切换，词法分析有两个状态，结构如下:

/\* for lexer state switch \*/

typedef enum{

NORMAL = 0,

SPECIAL

}STATUS;

NORMAL状态下主要接受合法单词，SPECIAL状态利用在某些特殊场合，例如给终结符设置前缀或修改驱动器分析入口函数名时可以引入一些特殊的字符串，这些字符串的某些字符在常规状态下是不被接受的。

## 语法分析

词法分析活动于语法分析的过程中，语法分析阶段主要填充一些数据结构，建立起一些框架结构，比如为产生式申请空间，为符号申请空间等。语法分析时不断的的向词法程序获取单词的同时也在进行状态的切换，马修的语法分析是采用的是递归下降子程序法。在这里虽然没有列出马修的语法规则，但是这里给出了马修的语法状态转换表。

表3-1 语法状态转换表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 状态 | % | 小写字母 | 大写字母 | 别名 | number | { | [ | ] | -> | ( | ) | . | < | Keyword |
| 0 | INITIALIZE |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | WAITING\_FOR\_DECL\_OR\_RULE | 2 | 3 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | WAITING\_FOR\_DECL\_KEYWORD |  |  |  |  |  |  |  |  |  |  |  |  |  | 13 |
| 3 | WAITING\_FOR\_ARROW |  |  |  |  |  |  |  |  | 4 | 5 |  |  |  |  |
| 4 | IN\_RHS |  | 4 | 4 |  |  | 4 |  |  |  | 8 |  | 12 |  |  |
| 5 | LHS\_ALIAS\_1 |  |  |  | 7 |  |  |  |  |  |  |  |  |  |  |
| 6 | LHS\_ALIAS\_2 |  |  |  |  |  |  |  |  |  |  | 7 |  |  |  |
| 7 | LHS\_ALIAS\_3 |  |  |  |  |  |  |  |  | 4 |  |  |  |  |  |
| 8 | RHS\_ALIAS\_1 |  |  |  | 9 |  |  |  |  |  |  |  |  |  |  |
| 9 | RHS\_ALIAS\_2 |  |  |  |  |  |  |  |  |  |  | 4 |  |  |  |
| 10 | PRECEDENCE\_MAK\_1 |  |  | 11 |  |  |  |  |  |  |  |  |  |  |  |
| 11 | PRECEDENCE\_MAK\_2 |  |  |  |  |  |  |  | 12 |  |  |  |  |  |  |
| 12 | WAITING\_DECL\_CODE |  |  |  |  |  | 1 | 11 |  |  |  |  |  |  |  |
| 13 | WAITING\_DECL\_TYPE |  |  |  |  |  |  |  |  |  |  |  | 1 |  |  |
| 14 | WAITING\_IDENTIFIER |  | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |
| 15 | WAITING\_TERMINAL |  |  | 1 |  |  |  |  |  |  |  |  |  |  |  |
| 16 | WAITING\_NONTERMINAL |  | 1 |  |  |  |  |  |  |  |  |  |  |  |  |
| 17 | WAITING\_FALLBACK\_SYMBOLS |  |  | 17 |  |  |  |  |  |  |  |  | 1 |  |  |
| 18 | WAITING\_PRECEDENCE\_SYMBOLS |  |  | 18 |  |  |  |  |  |  |  |  | 1 |  |  |
| 19 | WAITING\_DESTRUCT\_SYMBOLS |  |  |  |  |  | 1 |  |  |  |  |  |  |  |  |
| 20 | WAITING\_TYPE\_SYMBOLS |  | 20 |  |  |  |  |  |  |  |  |  | 1 |  |  |
| 21 | WAITING\_RULE\_CODE |  |  |  |  |  | 1 |  |  |  |  |  |  |  |  |
| 22 | RESYNC\_AFTER\_ERROR | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |

任何输入不合法都将进入RESYNC\_AFTER\_ERROR状态，RESYNC\_AFTER\_ERROR状态将会重新定位分析处并且进入WAITING\_FOR\_DECL\_OR\_RULE状态。

## 插入动作

关于插入动作的处理，插入动作参考YACC对插入动作的处理，处理如下：

A->a {action1} b.{action2}

将被处理为：

A->a nt\_000001 b.{action2}

nt\_0000001->.{action1}

对于语法分析阶段的插入动作，本人是利用了环形队列来实现的。在产生式未结束前遇到插入动作，就将相关信息入队，产生式结束后出队并建立插入动作。

## 开始符号的确定

马修规定对文法分析前必须得有一个开始符号，且这个开始符号不能出现在产生式的右边。如果没有找到开始符号，马修会自动为其添加一个开始符号，名字为start\_00x或者将第一个出现的产生式的左边的非终结符作为开始符号。 如果开始符号出现在产生式的右边，则新建一个开始符号并将第一个产生式应用于这个开始符号。

## 计算FIRST集

在计算状态转换表之前，首先需要计算FIRST集，具体实现在ffsets.c文件中。构造FIRST集的算法：

对于G中的每一文法符号V,为求FIRST(X),可反复应用如下规则，直到所求的FIRST集不再增大为止。

1. 若X ∈ VT ， 则 FIRST(X) = {X}。
2. 若 X ∈ VN ， 且有产生式 X→a...，a ∈ VT 则令a ∈ FIRST(X)。
3. 若 X→Y1 Y2 Y3…Yk∈ P，且Y1 ∈Vn，则令

FIRST(Y1)-{} ⊆ FIRST(X)

而对所有的j(1≤j≤i-1)， Yj ∈ VN，且 Yj 则令

FIRST(Yj)- {} ⊆ FIRST(X) (1≤j≤i)

特别当∈FIRST(Yj) (1≤j≤k)时，令∈FIRST(X)。

马修求FIRST集的算法是先计算非终结符的epsilon再计算FIRST集。马修采用递归的方式求非终结符的epsilon和FIRST集，从开始符号开始逐层求解关联的非终结符的epsilon和FIRST集，这样求解的好处是可以剔除无用的产生式。为了节省存储空间，马修以位的方式存储first集。马修参考了bison源代码提供的位运算的一些方法，并将其应用到了马修中。

## 分析状态的建立

求出FIRST集后就是进入LALR(1)项目集族的建立，马修建立LALR(1)项目集族前要建立LR(0)的各个状态，步骤：

(1) 以开始符号出发找到第一状态的基本项目，然后将结束符“#”加入第一状态的这一个基本项目上。

(2) 对第一状态的基本项目集进行闭包运算，求得此状态中包括基本项目和派生的非基本项目的全体集合，这个体项目的集合就是第一状态的完整形式。

(3) 对第一状态内的所有项目上的某个符号进行一次移进操作，移进操作后就可以得到一系列的新项目，这些新项目会是另一个状态的基本项目，我们在这些基本项目的bplp域，挂接一个它的“母体”。这样新项目和原项目之间就有了明确的遗传关系。这是LALR(1),语法分析器分析先行符号传播所必须的，把移动该符号所得到的所有项目作为基本项目，我们再一次运用闭包运算，则会得到由这些基本项目派生出来的非基本项目。在进行闭包运算时，会在基本项目的fplp域，挂上新产生的非基本项目。这两类项目的全体可以确定一个新状态。如果在这个新状态的项目上进行某个符号的又一次移动操作，则有两种肯能，一种是重复上述过程再建立一个新状态，另一种是进入已有状态中，此时状态中的全体项目数量不会有变化，变化的仅仅是某些项目上可能出现的“母体”，把这些项目的bplp域挂上进来时的“母体”。这就是LALR(1)语法分析器的先行符号传播的过程。

构造基本项目集合I的closure(I)运算：

(1) 首先，将I中的每个基本项目都放入闭包集合closure(I)中。

(2) 若有项目 A→α\* Bβ在closure(I)中，且B→γ是产生式，那么，若B→\*γ不在closure(I)中，则将其加入closure(I)中。

(3) 运用规则2，直到closure(I)不再扩大为止。

在进行找出FOLLOW集之前，要进行项目传播链的颠倒，将所有项目的bplp域进行反转，在寻找FOLLOW集的过程中仅需要装有后继项目的顺向传播链表fplp。

## 计算FOLLOW集

根据FOLLOW集的定义，对文法中每一个文法符号A ∈ VN，计算FOLLOW(A):

1. 若S为文法的开始符号，把{#}加入FOLLOW(S)中。
2. 若存在产生式A→α\* Bβ，则将FIRST(β)中除外的元素加入FOLLOW(B)中。
3. 若FIRST(β)中包含,则FOLLOW(A)也加入FOLLOW(B)中。

反复使用规则(2)直到每个非终结符的FOLLOW集不再增大为止。

## 装配动作链表

装配动作实现的是将动作装配成链表的形式，并对动作进行冲突解决，压缩动作，输出状态装换信息等。

马修对于冲突的解决方案与大多数语法分析器的解决方案一样。马修对于冲突的解决，依赖于产生式的优先级。马修规定产生式的优先级是以产生式右侧符号倒数，最先存在优先级的符号确定。如果产生式所有的符号都不存在优先级，则产生式不设置优先级。

移进—规约冲突的解决方案：

1. 如果一个终结符未指定优先级，当产生移进规约冲突时，则采用移进的解决方式，并且给出移进—规约的语法冲突信息。
2. 如果一个将移进的终结符的优先级小于一个将被规约的产生式的优先级，则采用规约的解决方式。
3. 如果一个将移进的终结符的优先级大于一个将被规约的产生式的优先级，则采用移进的解决方法。
4. 如果一个将移进的终结符的优先级等于一个将被规约的产生式的优先级，此时当这个终结符具有左结合性，则采用规约的方式解决。
5. 如果一个将移进的终结符的优先级等于一个将被规约的产生式的优先级，此时当这个终结符具有右结合性，则采用移进的方式解决。
6. 除了以上情况外，解决移进—规约冲突的方式都为移进操作，并且给出移进——规约的冲突信息。

规约—规约冲突的解决方案：

1. 如果两个产生式都有指定优先级， 但优先级不同，则采用具有更高优先级的产生式进行规约。
2. 如果两个产生式其中一个没有指定优先级，则采用指定了优先级的产生式进行规约，并且给出规约—规约语法冲突信息。
3. 其余的情况都采用先来先得的规则，排列在前面的产生式进行规约，并且给出规约—规约语法冲突信息。

压缩动作链表的目的在于缩小最后建立的分析表数组的大小，压缩方法是把一个状态中最频繁出现的规约动作作为这个状态的默认动作。压缩后的分析表情形就相当于解决了移入—规约冲突和规约规—约冲突的LR(0)分析器表。

## 生成LALR(1)语法分析驱动程序

生成LALR(1)语法分析驱动程序作为马修的最后一执行模块，马修在这一模块中引入了透明化输出这一概念，对于驱动文件的生成的输出格式是透明的，可提供给开发者进行修改的空间，只需过订制不同的输出格式的模板文件即可。通过这项特性，不需要修改马修源代码就可以生成适应不同语言平台的的语法分析驱动程序，即使无法直接生成开发者想要的语法分析驱动程序，也可以为开发者减少工作量。模板文件结构：

输出格式的Map映射部分

%%

驱动文件头

%%

驱动文件模板内容部份

…

%%

驱动文件尾

驱动模板文件Map映射部份采用红黑树进行存储。红黑树是一种性能较好的自平衡二叉树，因此本人将红黑树用于驱动模板文件的映射部份。关于利用红黑树如何实现map的具体实现，可以参考map.c这个文件。

## 二维数组的线性化压缩

LALR(1)分析器最后会生成一张二维的动作表，如果输入的内容较多，那么这个二维数组将会非常庞大，于是对于动作表这个庞大说的二维数组需要线性化压缩，也就是将其压缩为一维的数组。

数组的压缩算马修参考了LEMON使用的梳子算法。

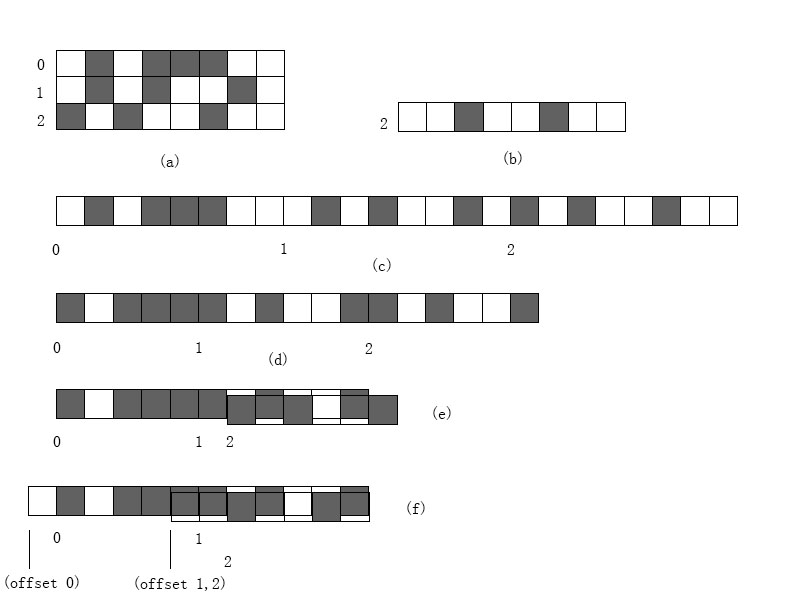


图 3-1数组的压缩

假设图3-1(a)给出了动作的二维数组，如果只进行线性化而不进行压缩那么这会使数组变得非常庞大，程序语言静态申请的数组空间的大小往往是受到限制的，庞大的数组也不利于对空间的利用。图3-1(c)中可以发现，数组中有许多空白处是没有装入动作的，因此可以利用这些空白处填入其它状态的动作，将放置动作数组的内存空间利用起来。图3-1(d)显示的是每一个状态去掉前后空白后对数组的初步压缩，图3-1(e)显示的是梳子算法对数组的再次压缩，梳子算法利用插梳子的方法，将锯齿数组看成梳子插入到另一把梳子中，这也是我们为什么称其为梳子算法的原因。

在进行插梳时马修首先使用的方法参考了LEMON给出的插梳算法，但马修并没有对各状态的动作数组进行排序，然后本人发现，程序出现语法错误后错误提示并未被正确的显示出来。本人发现，LEMON对于动作的查找算法是先查找出状态在动作数组中的offset，再与向前查看符号ilookahead相加得到的值作为下标i,再用下标i去向前查看符号数组中提取数组里的值，如果提取的值与向前查看符号相等，则说明在动作数组中。但这存在一个问题，假如我们将图3-1(a)的状态2的动作数组替换为图3-1(b)的样子，那结果如图3-1(f)所视，图中状态1与状态2拥有同样的offset。那么就会出现下面这种情况：

状态1只接受符号A的移入动作，状态2只接受符号B的移入动作。假如当前处于状态1，当此时输入的向前查看符号为B时，因为这两个状态拥有同样的offset，所以在向前查看符号数组里提取出的值通过了验证。这样状态1就执行了状态2中对于B的动作去了，错误提示就会因此被延期。

马修之前未对动作数组进行排序就出现了这种情况，LEMON对动作数组按动作的多少进行排序，这样撞击问题就不出现了。但本人认为还是有撞击的可能的，于是本人修改了关于动作数组的查找算法。本人不再沿用LEMON的向前查看符号数组，将向前查看数组转换为状态识别数组，也就是状态偏移值加上向前查看符号的值作为搜寻下标，取得的值如果与当前状态相等则认为此动作存在。这样做就避免了offset的撞击问题，每一个状态识别数组的元素都仅仅表示某个状态独有这个动作。但这样做有可能会使数组看起来比LEMON生成的数组要大，因为LEMON里给出的算法中，拥有相同动作数组的状态可被合并，但马修不允许合并。虽然如此但本人认为马修生成的动作数组不一定会比旧的算法要大，因为插梳算法对于梳子的形状和梳子插入的顺序是相关的，梳子插入顺序的变动会最终会影响生成动作数组的大小。目前本人还没找到一种能够找出最优顺序的插入算法，暴力运算是不实在的，以目前的计算机情况来看无法在短时间内列出所有插入情况，并从中选出最优情况。因此马修并没有对动作数组进行排序，这样就能减少马修的工作量，提高马修生成驱动程序的速度。

## 特殊标识

马修在某些特殊的动作代码中引入了特殊标识，对处于不同的动作代码的特殊标识，它们有着不同的含义。

1. 在析构动作的代码中“$$”代表要析构的目标。
2. 在语法出错执行的动作中“@@”代表正在输入的向前看符号，“$$”代表号符号对应的值。
3. 在语法规约动作中“$[<type>]$”代表产生式左边的非终结符的值。“$[<type>]number”中如果number为大于0零时和别名具有相同的用法，如果number为负数则代表分析栈距离栈顶的符号的值。“@number” 中number的范围如同$number的一样，但其表示的是距离栈顶number距离的符号，而不是符号的值。这些对分析栈的隐式操作是不安全的，YACC里曾经这样提到过—继承属性有时是非常有用的，但它们也是很难发现错误来源的，使用继承属性的动作必须考虑语法中使用规则的每个地方，使用合成属性更安全。
4. 对于拥有解析特殊标识的动作，如果在模板规则中开启了的转义符号，转义符号的作用是为了原样输出后面紧跟着的那个字符。所以在编辑以上类型的动作时要注意假如将感叹号设置为特殊转义符号，输出感叹号时要使用双感叹号。

# 系统测试

## 系统测试简述

系统测试的主要目的是验证程序是否达到目标要求，程序是否存在bug。本人将会就任务书的要求对马修的各个功能部份进行测试，验证马修是否达到预期目标要求。马修将会举出几个测试实例，并根据数据的流向进行分析。

## 系统测试内容

### 命令行输入测试

命令行作为马修执行分析控制和识别输入文件的重要部份，命令行识别出错时不仅要在出错位置提供准确的错误描述，还应给出命令行的正确参数形式。例如给出如下的输入命令：

mathew gram.y -k -v Win32 echo() echo=3 macroinc()

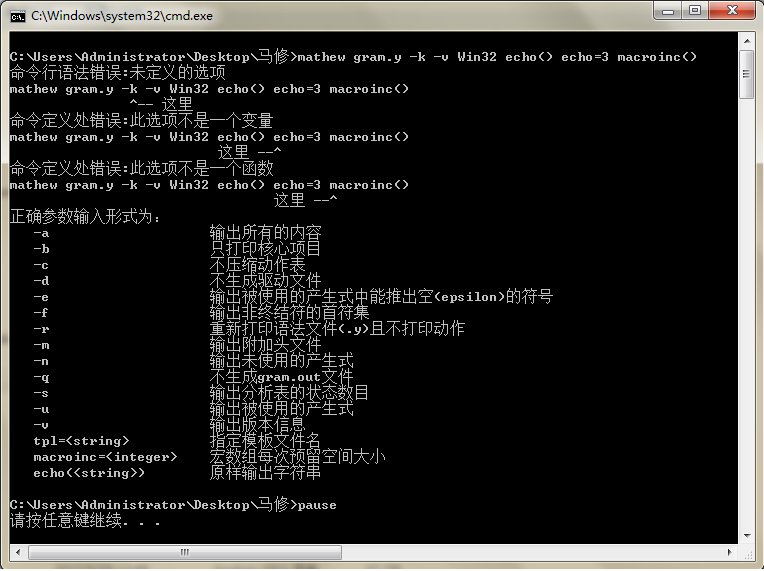


图4-1 命令行提示

结果如图4-1所示，当命令的选项不存在时将给出选项未定义的提示，同时还指出了出错的位置。

### 预处理功能测试

预处理主要用于对宏指令%ifdef,%ifndef和%endif的测试。为了方便观察结果本人以文件形式输出被预处理过了的语法文件。

测试结果表明马修利用图4-1给出的Win32宏成功的剔除或保留了需要处理的代码。

### 语法文件分析流测试

为了跟踪马修的词法部份是如何将输入流拆分成一个个单词，语法部份又如何根据输入单词进行一步步分析的。本人在每个状态的执行函数中引入了STATE宏，STATE宏用于内部测试使用，假如马修的源文件中定义了TEST宏，那么马修将会根据STATE宏里定义的内容输出当前分析的情况信息。

### 语法文件分析出错测试

出错提示做为程序非常重要的部份，显示错误的内容，指出出错位置在哪里，这些都是提供给开发者检查语法文件书写问题的有效手段。

假如在书写别名时忘加“$”符号：

program -> expr(A).{

printf("Result = %d\n", $A);

}

那么马修就会给出如下提示：

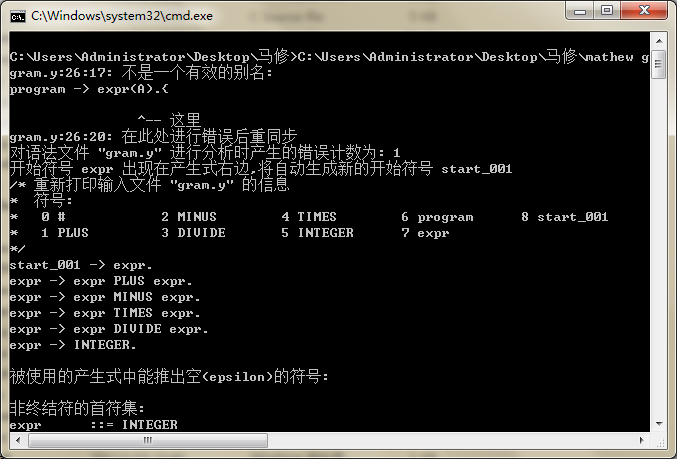


图4-2 错误提示处理

图4-2显示的测试结果表明马修不仅对错误的内容给出了正确的提示，还指出了错误所在的位置。马修的错误重同步功能也在这里体现了出来，图4-2中也体现出了马修在没有确定的开始符号的情况下会自动创建开始符号。

### 各阶段显示功能测试

马修在分析的各阶段会出现一些产物，例如非终结符FIRST集，被使用的产生式的显示和无用产生式的显示等，这些阶段产物可以用于教学上，用于验证手动计算是否与机器吻合。测试各阶段产物的输出也是检测马修是否达到目标要求的途径之一。

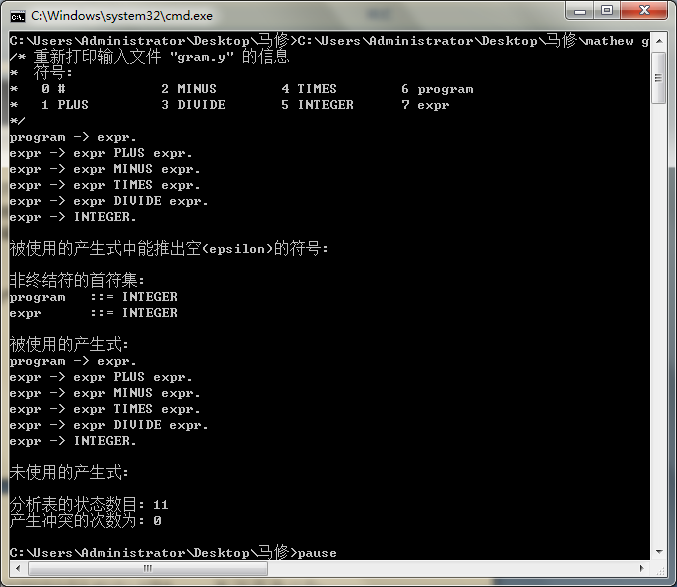


图4-3 各阶段输出结果

图4-3正确的显示出了各阶段产物的输出结果。

### 状态转换信息测试

状态转换信息将会以文件形式输出，输出文件与语法文件同名且后缀为“.out”。本次测试使用的是简易整数计算器，状态转装换表如图4-4。

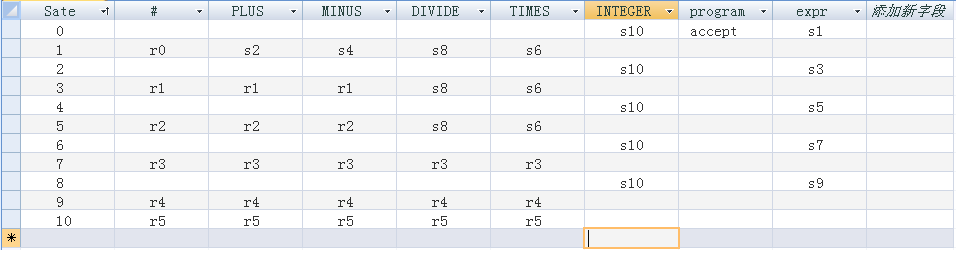


图4-4 状态转换表

State 0:

program -> \* expr .

expr -> \* expr PLUS expr .

expr -> \* expr MINUS expr .

expr -> \* expr TIMES expr .

expr -> \* expr DIVIDE expr .

expr -> \* INTEGER .

INTEGER shift 10

program accept

expr shift 1

截取.out文件中的一个状态，再对照生成的语法驱动程序中的动作素组的前三个元素。

/\* action table \*/

static const unsigned char yy\_action[] = {

10, 17, 1, 11, 2, 4, 8, 6,

10, 3, 8, 6, 10, 5, 8, 6,

10, 7, 10, 9,

};

可以发现动作数组的前三个元素10，17，1对应上面状态0的shift 10，accept，shift 1。

### 语法分析驱动程序测试

本次测试通过利用一个简易的整数计算器进行测试。本次测试平台为C语言平台，最终生成的简易整数计算器是以C语言的形式给出的。首先将驱动模板文件替换为C语言的驱动模板文件，再根据”.y”语法文件的规则生成语法驱动文件yy\_table.c，将此文件编译观察运行结果。

测试的计算内容如下：

Parse(pParser,INTEGER,5);

Parse(pParser,PLUS,0);

Parse(pParser,INTEGER,6);

Parse(pParser,TIMES,0);

Parse(pParser,INTEGER,3);

Parse(pParser,0,0);

以上的内容是计算5+6\*3,结果正好如图4-5所示。

正确.png

图4-5 正确结果

当把数字6的输入注释掉时：

Parse(pParser,INTEGER,5);

Parse(pParser,PLUS,0);

/\* Parse(pParser,INTEGER,6); \*/

Parse(pParser,TIMES,0);

Parse(pParser,INTEGER,3);

Parse(pParser,0,0);

也就是连续输入了两个运算符号，就会出现图4-6的情况。

错误.png

图4-6 错误结果

马修正确地显示出了语法出错的信息，并且给出了出错原因是输入了乘法符号TIMES(4),测试成功。

### 切换平台测试

马修的最初设计目标包含了生成适应不同平台的语法驱动程序这一项。本次测试的平台将在java中进行。本次测试使用相同产生式的简易整数计算器，修改驱动模板为java驱动模板文件。

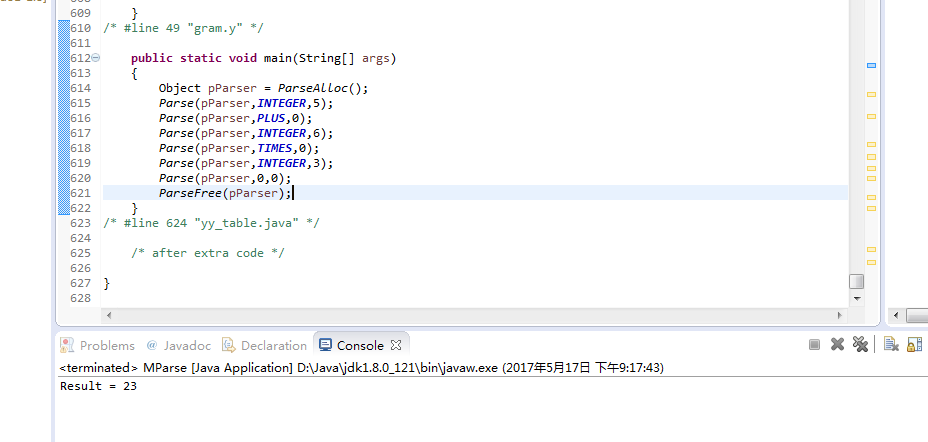


图4-7 平台转换结果

从图4-7中可以看出运行结果与C语言平台的运行结果一样。测试结果表明在不修改马修源代码的情况下可以生成不同平台的语法驱动程序。