

**实践课程报告**

**题目： Dragon 语言编译器实现**

**课程名称： 编译原理实践**

**专业班级： 　　计科 1409 班**

**学 号： U201414800**

**姓 名： 刘 一 龙**

**指导教师： 邵 志 远**

**报告日期： 2017年1月14日**

**计算机科学与技术学院**

**目 录**

[1选题背景 1](#_Toc779505207)

[1.1任务 1](#_Toc740849389)

[1.2目标 1](#_Toc1079335120)

[1.3 源语言定义 1](#_Toc1948302203)

[2实验一 词法分析和语法分析 5](#_Toc1028690505)

[2.1单词文法描述 5](#_Toc1095409141)

[2.1.1 常量规则 5](#_Toc221004509)

[2.1.2 注释规则 5](#_Toc1927791858)

[2.1.3 规则综述 6](#_Toc1995156923)

[2.2语言文法描述 6](#_Toc725174400)

[2.2.1 if-else 冲突 6](#_Toc921941510)

[2.2.2 expr 冲突 7](#_Toc572311180)

[2.3 词法分析器的设计 7](#_Toc696907566)

[2.3.1 调试信息 7](#_Toc733730239)

[2.3.2 设计综述 8](#_Toc782961066)

[2.4 语法分析器设计 8](#_Toc659465534)

[2.4.1 错误恢复 8](#_Toc2089013435)

[2.4.2 错误提示 9](#_Toc1581501185)

[2.4.3 生成语法树 10](#_Toc2142015688)

[2.5语法分析器实现结果展示 12](#_Toc74110273)

[2.5.1 语法报错功能 12](#_Toc133904309)

[2.5.2 语法树生成功能 15](#_Toc201484024)

[3语义分析 23](#_Toc696661945)

[3.1语义表示方法描述 23](#_Toc1148196713)

[3.2符号表结构定义 23](#_Toc863597475)

[3.3错误类型码定义 23](#_Toc65116027)

[3.4 语义分析实现技术 23](#_Toc1452271121)

[3.5语义分析结果展示 23](#_Toc1102201809)

[4中间代码生成 24](#_Toc1988159792)

[4.1中间代码格式定义 24](#_Toc282126473)

[4.2中间代码生成规则定义 24](#_Toc516286858)

[4.3 中间代码生成过程 24](#_Toc620181351)

[4.4代码优化 24](#_Toc1022975862)

[4.5 中间代码生成结果展示 24](#_Toc1595621978)

[5目标代码生成 25](#_Toc420999906)

[5.1指令集选择 25](#_Toc2051666367)

[5.2寄存器分配算法 25](#_Toc543547471)

[5.3 目标代码生成算法 25](#_Toc642004415)

[5.4 目标代码生成结果展示 25](#_Toc1831974578)

[5.5目标代码运行结果展示 25](#_Toc391220746)

[6结束语 26](#_Toc1367178815)

[6.1 实践课程小结 26](#_Toc606432440)

[6.2自己的亲身体会 26](#_Toc963531926)

[参考文献 27](#_Toc2064086381)

# 1选题背景

1.1任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

1.2目标

本次课程实践目标是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

1.3 源语言定义

Dragon 语言是基于 Decaf 语言的一门简单的支持面向对象范式的语言。其语言基本定义如下（仿巴库斯-瑙尔范式，并结合扩展正则表达式 + ?）：

program:

class\_defs

;

class\_defs:

(class\_def ‘,’)\*

;

class\_def:

CLASS IDENTIFIER '{' fields '}'

| CLASS IDENTIFIER EXTENDS IDENTIFIER '{' fields '}'

;

fields:

(field ‘,’)\*

;

field:

var\_def

| func\_def

;

var\_def:

var ';'

;

var:

type IDENTIFIER

;

type:

INT

| BOOL

| STRING

| VOID

| CLASS IDENTIFIER

| type '[' ']'

;

func\_def:

type IDENTIFIER '=' '(' formals ')' OP\_ARROW stmt\_block

;

formals:

(var ‘,’)\*

;

stmt\_block:

‘{‘ stmt ‘}’

;

stmts:

stmt+

;

stmt:

var\_def

| simple\_stmt ';'

| if\_stmt

| while\_stmt

| for\_stmt

| return\_stmt ';'

| print\_stmt ';'

| stmt\_block

;

simple\_stmt:

left\_val '=' right\_val

| call

;

left\_val:

receiver '.' IDENTIFIER

| expr '[' expr ']'

;

if\_stmt:

IF '(' expr ')' stmt

| IF '(' expr ')' stmt ELSE stmt

;

while\_stmt:

WHILE '(' bool\_expr ')' stmt

;

for\_stmt:

FOR '(' simple\_stmt ';' expr ';' simple\_stmt ')' stmt

;

return\_stmt:

RETURN (expr)?

;

print\_stmt:

PRINT '(' exprs ')'

;

call:

receiver' IDENTIFIER '(' actuals ')'

;

receiver:

(expr ‘.’)?

;

actuals:

` (exprs )?

;

exprs:

(expr ‘,’)+

expr:

constant

| left\_val

| THIS

| call

| '(' expr ')'

| expr '+' expr

| expr '-' expr

| expr '\*' expr

| expr '/' expr

| expr '%' expr

| '-' expr

| expr '<' expr

| expr OP\_LE expr

| expr '>' expr

| expr OP\_GE expr

| expr OP\_EQ expr

| expr OP\_NE expr

| expr OP\_AND expr

| expr OP\_OR expr

| '!' expr

| READINTEGER '(' ')'

| READLINE '(' ')'

| NEW IDENTIFIER '(' actuals ')'

| NEW type '[' expr ']'

;

constant:

CONSTANT\_INT

| CONSTANT\_BOOL

| CONSTANT\_STRING

| NIL

;

# **2实验一 词法分析和语法分析**

2.1单词文法描述

词法分析阶段所需要做的工作为将输入串按照一定的词法规则解析成 token 序列，提供给下一编译阶段（语法分析）使用。在 Dragon 语言中，token 大致分为四类，关键字、标识符、常量、运算符（包括分隔符）。由于 Flex 的最长匹配与最先匹配原则，应将关键字列在词法规则描述的最前部，才可避免将关键字识别成标识符的错误。

确定好 token 的基本分类与解析优先级后，便可以开始编写词法规则描述。其中，需要特别注意各个常量的词法规则与注释规则的解析。

### 2.1.1 常量规则

10进制整型常量：0 或 非 0 开始的数字串，得到正则表达式为 0|[1-9]{0-9}；

16进制整型常量：0x 或 0X 开始的数字串（包括a-f、A-F），得到正则表达式为0[xX]{a-fA-F0-9}+；

字符串常量：以 “ “ 为开头与结尾的任意串，得到正则表达式为 \"(\\.|[^\\"\n])\*\"。值得一提的是，为了编译器实现逻辑上的统一整洁，并没有编写真正的输出函数，而是直接利用 linux C 中的标准格式化输出函数（printf），从而实现了一定程度上（可以说十分完备的）转义字符支持。

e.g string s; s = “\nreturn”; Print(s);

上述程序会将 “\n” 转义为换行符进行输出。

### 2.1.2 注释规则

行注释：由 // 开始直至换行符即可，得到正则表达式为 "//"[^\n]\*；

块注释：块注释的识别借用了函数，当解析到 /\* 后调用识别块注释的函数，如下所示：

while ((ch = yyinput()) != '\0' && ch != EOF) {

if (ch == '\n') {

srcbuf\_append("\n");

}

if (prev == '\*' && ch == '/') {

return;

} else {

prev = ch;

}

}

上述代码实现了一个简单的状态机，用于识别是否遇到了 \*/，即可完成块注释的识别。

### 2.1.3 规则综述

解决了常量识别与注释识别后，其余规则仅是简单地罗列关键字与符号。综合上述几步，即可完成词法规则描述的实现。利用 Flex 将相关的描述文件（scanner.ll）进行编译得到 C 文件，其便是 Flex 为我们生成的一个可识别 Dragon 语言中所有合法 token 的确定性有限状态自动机（DFA）。利用这个文件便可完成所有输入串的识别与 token 序列的生成。

2.2语言文法描述

由于 Bison 默认进行 LALR（1）分析，所以需要将第1章所提到的文法改写为不含任何移进-归约冲突或归约-归约冲突的 LALR（1）文法。一方面，需要将 \*/+/? 转化为上下文无关文法，另一方面要消除所有可能的冲突。

简单地将 \*/+/? 等正则表达式规则转化为标准 Bison 文法规则描述后，发现存在 205 个移进-归约冲突以及 2 个归约-归约冲突，如图 2-1 所示。

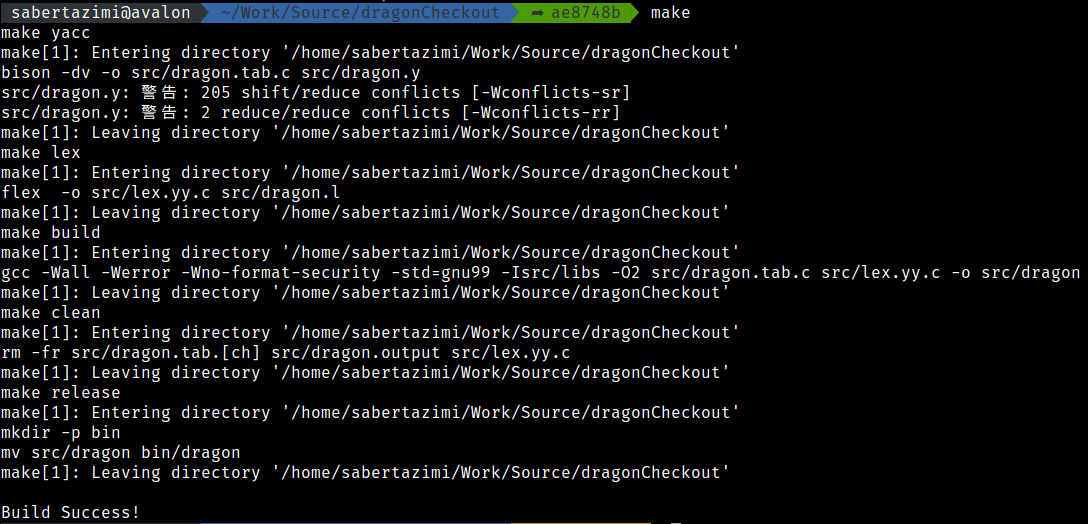


图 2-1 文法冲突结果

利用 bison -v 标志来编译文法规则描述文件（parser.yy），可以产生分析表（.output），观察分析表文件可以得到冲突主要集中在 expr 文法定义中，以及 if-else 匹配冲突。利用 Bison 内置的优先级与结合性定义，可以在不对文法做过大的改动的情况下，解决大部分冲突。

### 2.2.1 if-else 冲突

利用优先级，强制 Bison 优先匹配带分支的语句，即可解决此移进-归约冲突。

%nonassoc ')' NOELSE

%nonassoc ELSE

在 .yy 文件第一部分定义优先级。值得注意的是，定义在下方的符号优先级要高于定义在上方。在修改原 if 语句文法至如下所示：

if\_stmt

: IF '(' expr ')' stmt %prec NOELSE

{

@$ = @1;

$$ = new If($3, $5, 0, locdup(&@1));

}

| IF '(' expr ')' stmt ELSE stmt

{

@$ = @1;

$$ = new If($3, $5, $7, locdup(&@1));

}

；

### 2.2.2 expr 冲突

同样的道理，尽可能给所有表达式文法产生式给定一个优先级别，从而达到消除冲突的目的。添加如下的优先级与结合性定义：

%left OP\_OR

%left OP\_AND

%nonassoc OP\_EQ OP\_NE

%nonassoc OP\_LE OP\_GE '<' '>'

%left '+' '-'

%left '\*' '/' '%'

%nonassoc UNEG '!'

%nonassoc '[' '.'

即可消除表达式文法中的冲突。

2.3 词法分析器的设计

### 2.3.1 调试信息

为了 Dragon 编译器能够有良好的开发体验与用户体验，需要提供 2 种调试信息。一类为面向开发者的 token 序列识别结果信息，另一类为面向用户的 token 位置信息（用于词法/语法/语义错误提示）。

为实现面向开发者的调试信息，仅需编写简单的输出函数，并将其加入至词法规则中的动作中去，便可完成相关调试信息的输出。为此，编写了一个名为 DRAGON\_DEBUG 的宏，如下所示：

/// \macro DRAGON\_DEBUG

/// \brief print debug info for lexical analysis

#ifdef LEX\_DEBUG

#undef DRAGON\_DEBUG

#define DRAGON\_DEBUG(type, value) do { dragon\_debug(type, value); } while (0)

#else

#undef DRAGON\_DEBUG

#define DRAGON\_DEBUG(type, value) do { } while (0)

#endif

可以看到，当开发者在源码（scanner.ll）中加入 LEX\_DEBUG 宏定义，即可打开词法分析调试规则，上述宏会自动调用声明于 src/errors/utils.h 中的调试信息输出函数来完成打印调试信息功能。

为实现面向用户的调试信息，利用 Flex 内置的位置信息宏与位置信息结构体，即可实现位置信息的获取。按照实验指导书上所给的内置宏定义，并加以修改，得到下列宏定义，以获取当前 token 的位置信息：

#define YY\_USER\_INIT \

yy\_push\_state(SRCBUF); \

current\_line = current\_column = 1; \

yylloc.first\_line = yylloc.first\_column = 1; \

srcbuf\_init();

#define YY\_USER\_ACTION \

yylloc.first\_line = yylloc.last\_line = yylineno; \

yylloc.first\_column = current\_column; \

yylloc.last\_column = current\_column + yyleng - 1; \

current\_column += yyleng;

### 2.3.2 设计综述

结合词法规则描述、词法动作集，完成 token 序列识别以及调试信息的获取与输出，即可完成词法分析器的设计。看上去编写一个词法分析器，似乎是只要编写好不同 token 的识别规则即可，这是因为 Flex 做了大量的工作，它负责读取撰写好的词法描述规则，将割裂的各个正则表达式转化为一个大的非确定性有限状态自动机（NFA），再经过一系列变换，得到最简化的确定性有限状态自动机，其可以识别 Dragon 语言源文件的任意合法 token。

2.4 语法分析器设计

除了移进-归约冲突与归约-归约冲突的消除，使得语法分析器能够进行正常的语法检查外，还需实现错误恢复、错误提示、生成语法树等功能。

### 2.4.1 错误恢复

当遇到用户程序出现语法错误时，若不进行错误恢复，那么每次用户运行完编译器后都只能找到一个语法错误。显然，这样的编译器用户体验足够地差。为了大幅度改善用户体验，使得 Dragon 编译器能够一次性提示多条语法错误，而不至于一遇到语法错误便退出编译，就必须进行必要的错误恢复。

错误恢复指的便是利用 Bison 内置的 error 标志，编写额外的产生式，用于归约错误的文法规则，从而达到将错误的文法规则转化为正常的文法规则的效果，使得编译器能够正常地继续执行下去。

以变量定义为例：

var

: type IDENTIFIER

{

@$ = @2;

$$ = new VarDef($2, $1, locdup(&@2));

}

/\* error recovery \*/

| type error

{

@$ = @2;

$$ = 0;

proposed\_solution("expected identifier as variable name");

}

;

产生式 var -> type error，使得当用户编写了 int 3; 错误程序时，编译器利用语法动作（即大括号内部的程序），先输出错误信息，再输出解决建议，并将 int 3; 视为正常规则进行归约，使得 LALR（1）分析栈中可能产生错误的 token 全部出栈，并从错误语句的下一 token 开始进行分析，从而达到消除错误状态的目的，最终规避了 Bison 的中途停止。

类似地，逐条分析每一个非终结符的产生式，在合适的位置加入 error 产生式，尽可能多地进行错误恢复，使得编译器在尽可能少的调用次数里，将所有可能的语法错误都展示给用户，使得用户获得一个良好的体验。

### 2.4.2 错误提示

上述错误恢复中，提到了所谓的语义动作，这是 Bison 提供的属性文法的实现方式，是用来实现复杂语法分析器的基础。在每条产生式的后面，可以用大括号将一段标准 C/C++ 程序包裹。每当语法分析器要进行归约时，都会先执行大括号内部的程序，再进行归约。一般将大括号内部程序称为语义动作集。其与基本的文法一道，构成了基本的属性文法实现。

利用语义动作，可以实现基本的错误提示功能。首先回到前一个阶段，若想实现类似 gcc 那样报告语法错误的行数，并打印相关源代码，就必须在词法阶段完成以下 2 项工作，保存 token 的位置信息以及保存源代码。在之前的词法分析阶段，已经利用 Flex 内置宏获取了位置信息，现在只用实现源代码的保存。

关于源代码的保存，基本的思路如下图 2-2 所示。每读取一个非换行字符，都将其保存至缓冲区；当读取到换行符时，将缓冲区中所有字符连接成句子，保存至源代码链表中，并清空缓冲区，准备读入下一行源代码。利用 Flex 内置的状态机规则，以及 yyless 函数，可以大幅度简化实现。利用 Flex 内置的状态机规则，可以避免将每个词法规则的后面都添加额外的字符入缓冲动作。利用 yyless 函数，可以一次性读入整句，将其加入源代码列表，然后调用 yyless(0)，会将之前读入的句子归还给输入串，重新分析；在进行第二次分析的时候， Flex 会跳过调用了 yyless 函数的那条词法规则，再结合内置的状态机，可以实现第一次读入用于保存源代码，第二次读入用于词法分析与 token 生成，大幅度简化实现。

**读入非换行字符**

**读入非换行字符**

**读入换行符**

**读入换行符**

**字符加入缓冲**

**保存句子，清缓冲**

图 2-2 保存源代码状态机

### 2.4.3 生成语法树

语法分析阶段为后续阶段提供的最重要的数据结构便是由开始符号起而生成的一个完整的抽象语法树结构。只有当生成了完备（足以完成后续阶段的实现）而正确（正确反映源代码语法结构）的抽象语法树时，才是真正地完成了一个具有一定功能的语法分析器。

利用 Bison 内置的语义动作，可以在进行语法分析归约产生式的同时，将结点构造与树的连接动作加入语义动作集，从而实现抽象语法树的生成。

既然利用的是语法分析的归约过程，那么相关抽象语法树的结点数据结构的设计，应当尽可能贴合文法结构，使得结点构造与树的连接实现变得尽可能简单。

以类定义为例：

class\_def

: CLASS IDENTIFIER '{' fields '}'

| CLASS IDENTIFIER EXTENDS IDENTIFIER '{' fields '}'

;

可以看到由 class 关键字、一个类标识符、一个父类标识符以及一个成员集可以归约成一个类定义，则将类定义的抽象语法树结点设计成拥有 4 个 成员的 C++ 类，分别为 kind 域标识其为类定义， name 域记录 identifier 的名字，parent 域记录父类的名字，fields 域为一个 field 链表。再加上其余必要的上下文成员，如 loc 域记录类定义的具体位置信息等此阶段或后续阶段可能用到的信息，即可完成类定义结点的设计。最终其相关数据结构定义如图 2-3 所示。

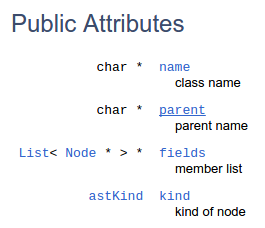


图 2-3 ClassDef 数据结构基础定义

当然，ClassDef 类远不止这些成员，若后续阶段需要再次遍历抽象语法树时，可能需要往抽象语法树上的结点添加更多的成员，以简化编译器的实现。

同样地，逐条分析语法规则，依次设计必要的抽象语法树结点数据结构，并编写相应的构造函数，并在对应的语法规则的语义动作中插入相应的抽象语法树构建函数的调用，即可在语法分析完成的同时，构造出一颗抽象语法树。如下所示，其中 @x 为对应符号的位置信息索引， $x 为对应符号的指针索引：

class\_def

: CLASS IDENTIFIER '{' fields '}'

{

@$ = @2;

$$ = new ClassDef($2, strdup(""), $4, locdup(&@2));

}

| CLASS IDENTIFIER EXTENDS IDENTIFIER '{' fields '}'

{

@$ = @2;

$$ = new ClassDef($2, $4, $6, locdup(&@2));

}

;

如图 2-4 所示，其为抽象语法树结点所需数据结构的列表。若想获取各个类的详细信息，可以参看 <https://sabertazimi.github.io/dragon/index.html>，此为 Dragon 编译器的 API 在线文档（由 Doxygen 构建）。



图 2-4 抽象语法树实现相关数据结构

2.5语法分析器实现结果展示

### 2.5.1 语法报错功能

修改 src/dragon.cc 开启 AST\_DEBUG 宏，利用几个错误用户代码（见test/error\_\*.dg）来测试报错功能。利用 Dragon 语法分析器分析后得到如图 2-5 至图 2-11 所示。

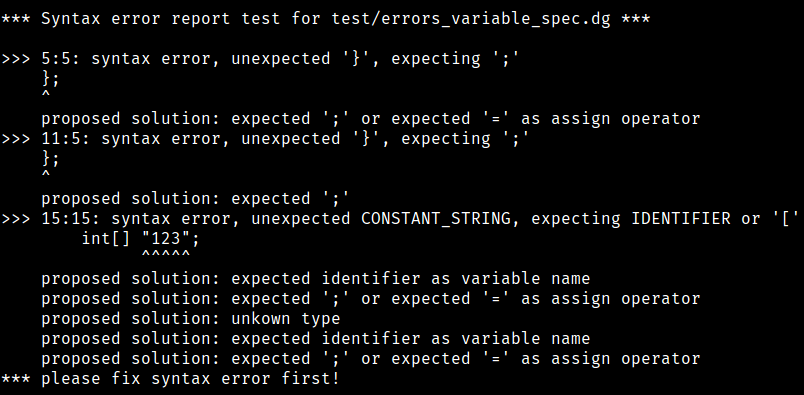


图 2-5 变量定义相关语法报错

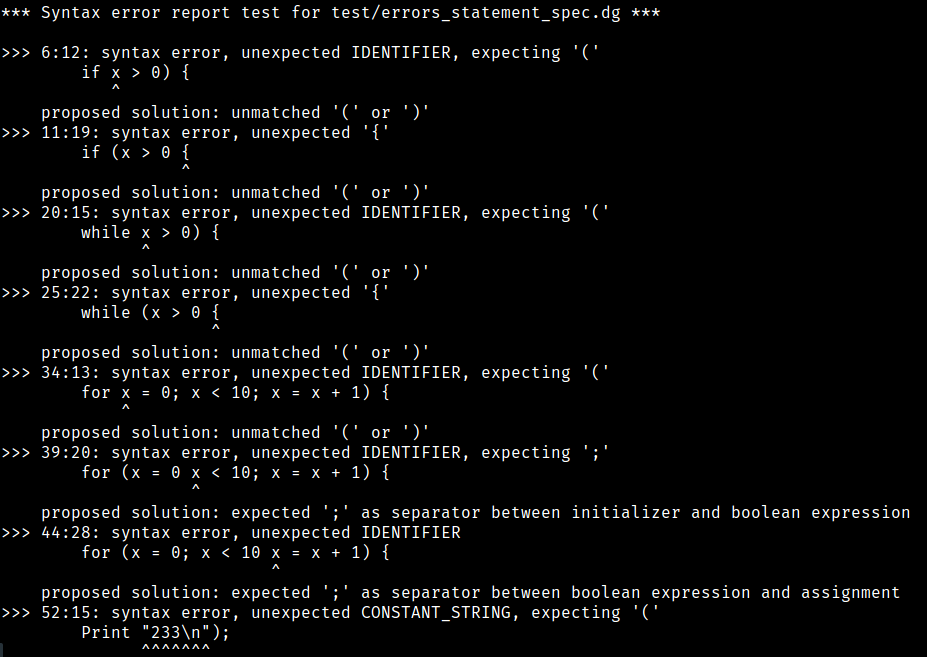


图 2-6 语句相关语法报错（1）

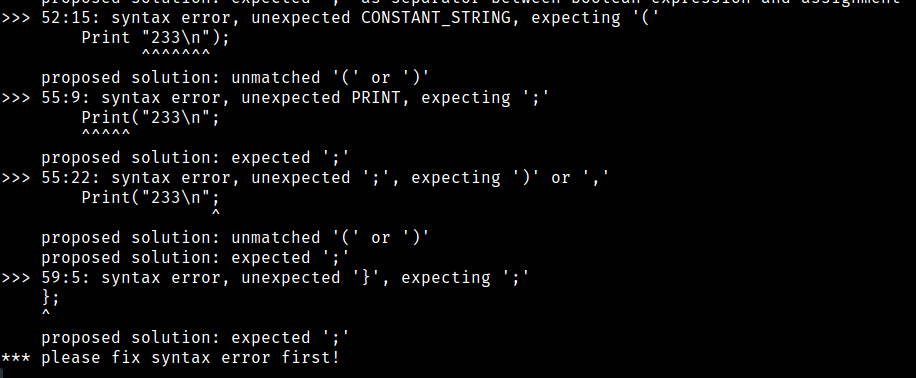


图 2-7 语句相关语法报错（2）

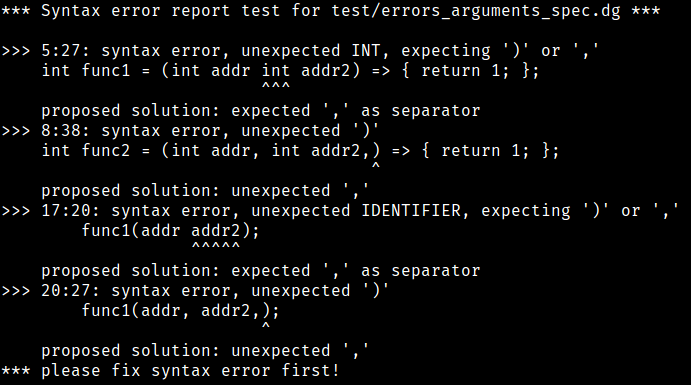


图 2-8 参数相关语法报错

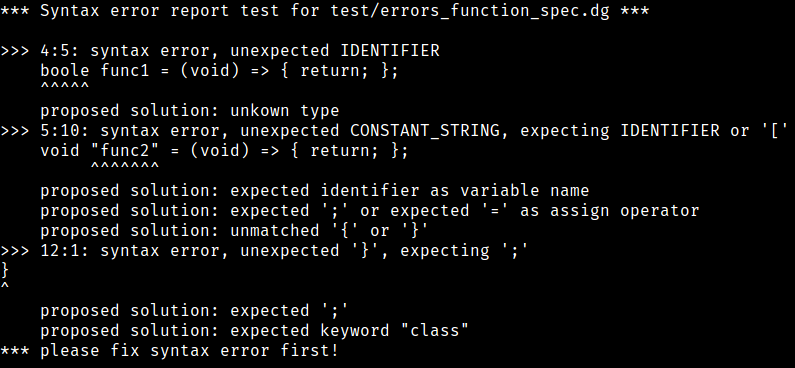


图 2-9 函数定义相关语法报错

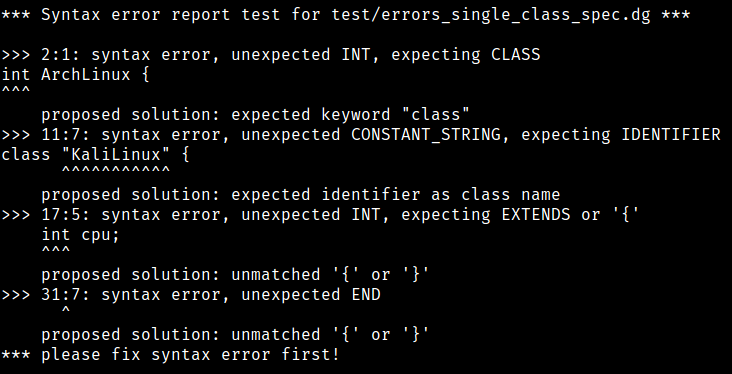


图 2-10 类定义相关语法报错

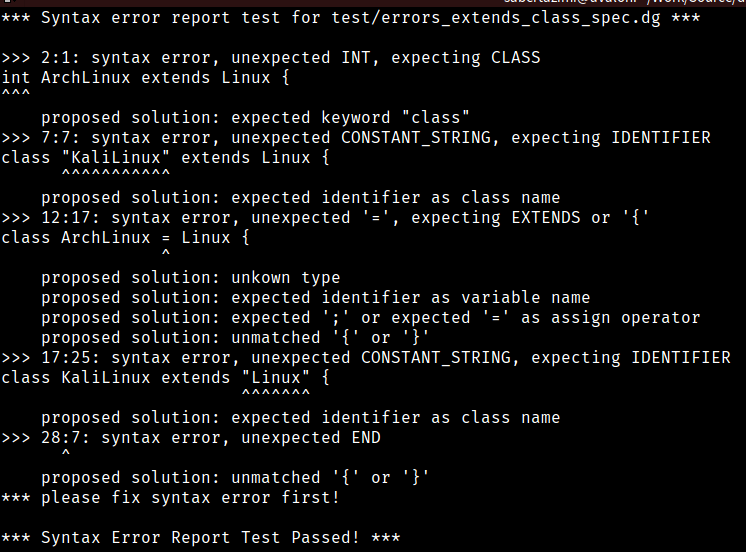


图 2-11 类继承相关语法报错

### 2.5.2 语法树生成功能

同样地，修改 src/dragon.cc，开启 AST\_SYNTAX 宏，用 Dragon 语法解析器解析正确的用户代码（见 test/ast\_\*.dg），结果如图 2-12 至图 2-18 所示：



图 2-12 生成抽象语法树（1）

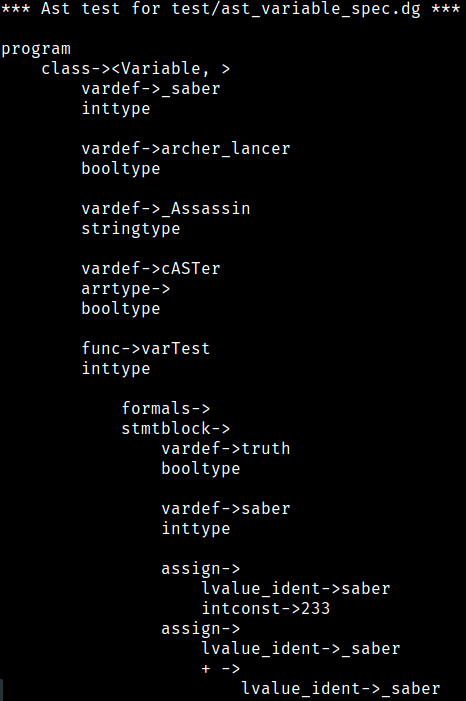


图 2-13 生成抽象语法树（2）

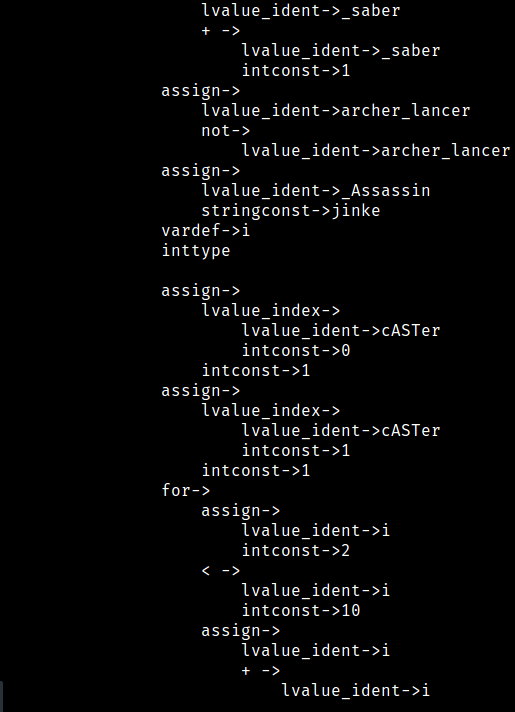


图 2-14 生成抽象语法树（3）

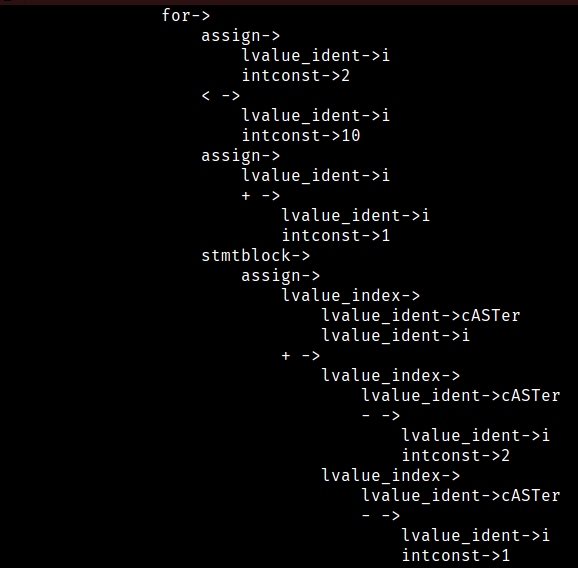


图 2-15 生成抽象语法树（4）

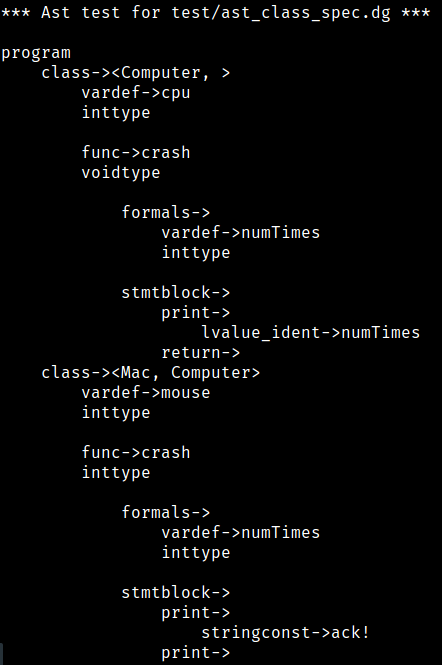


图 2-16 生成抽象语法树（5）

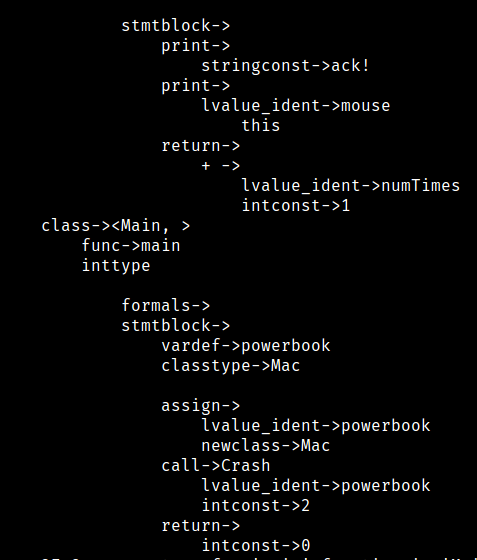


图 2-17 生成抽象语法树（6）

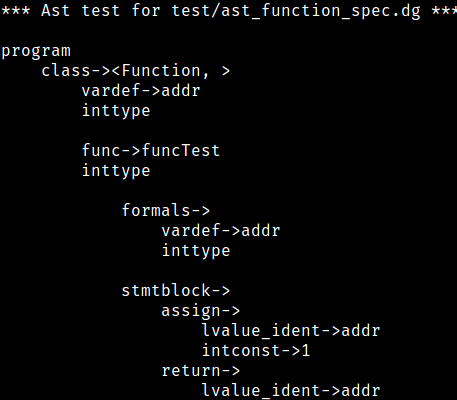


图 2-18 生成抽象语法树（7）

# 3语义分析

3.1语义表示方法描述

3.2符号表结构定义

3.3错误类型码定义

3.4 语义分析实现技术

3.5语义分析结果展示

# 4中间代码生成

### 4.1中间代码格式定义

### 4.2中间代码生成规则定义

### 4.3 中间代码生成过程

### 4.4代码优化

### 4.5 中间代码生成结果展示

# 5目标代码生成

### 5.1指令集选择

### 5.2寄存器分配算法

### 5.3 目标代码生成算法

### 5.4 目标代码生成结果展示

### 5.5目标代码运行结果展示

# 6结束语

6.1 实践课程小结

6.2自己的亲身体会

**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] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008