

**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。

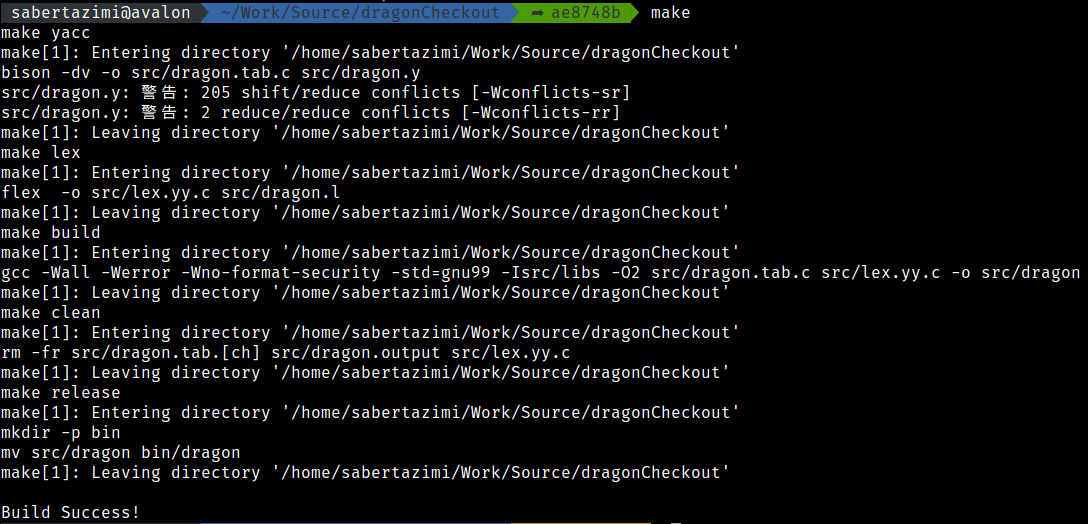


图 2-1 文法冲突结果

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。

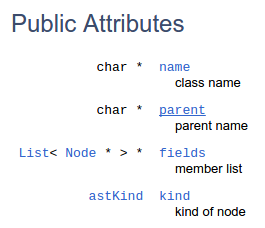


图 2-3 ClassDef 数据结构基础定义

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。

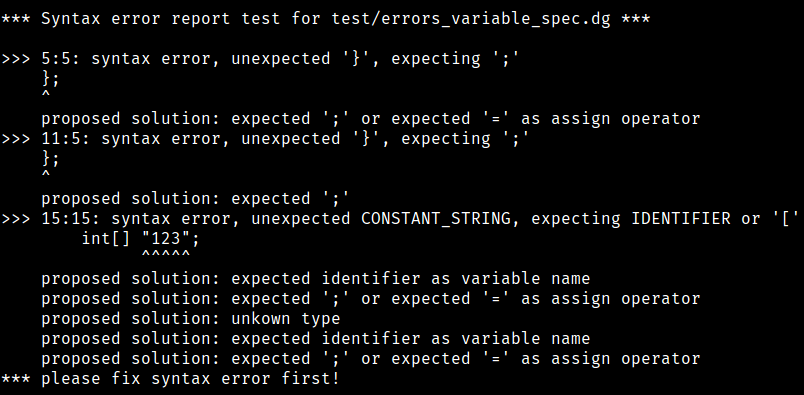


图 2-5 变量定义相关语法报错

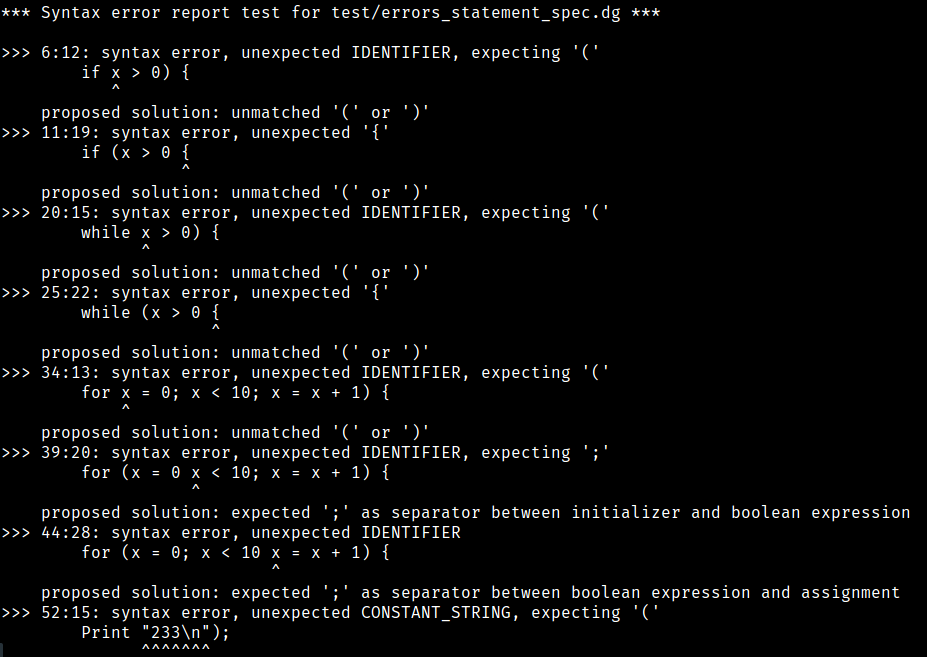


图 2-6 语句相关语法报错（1）

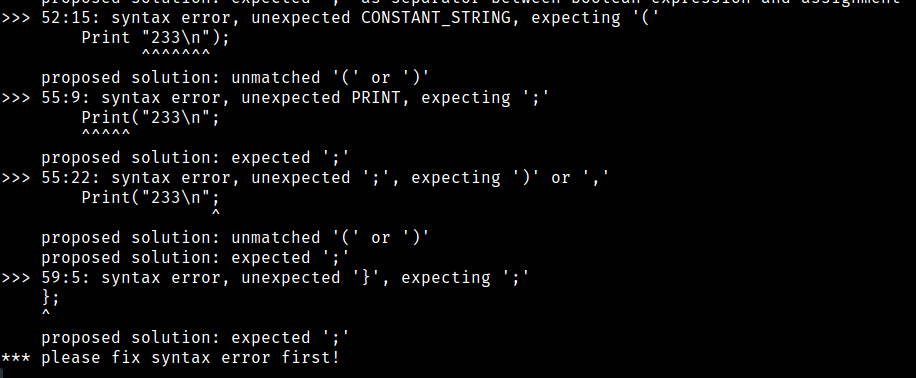


图 2-7 语句相关语法报错（2）

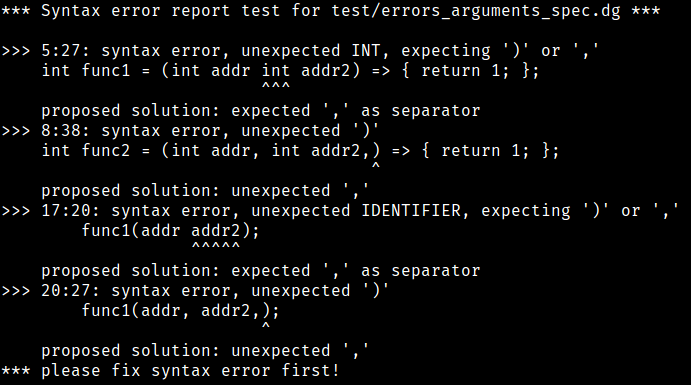


图 2-8 参数相关语法报错

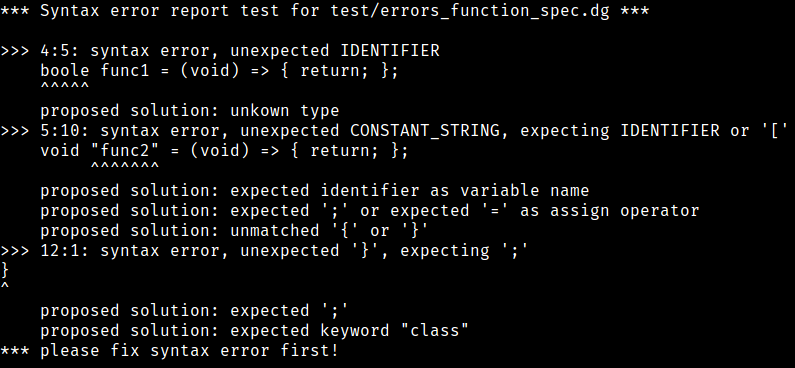


图 2-9 函数定义相关语法报错

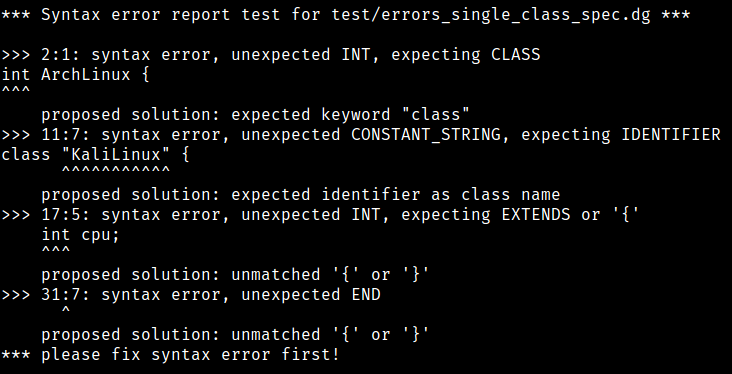


图 2-10 类定义相关语法报错

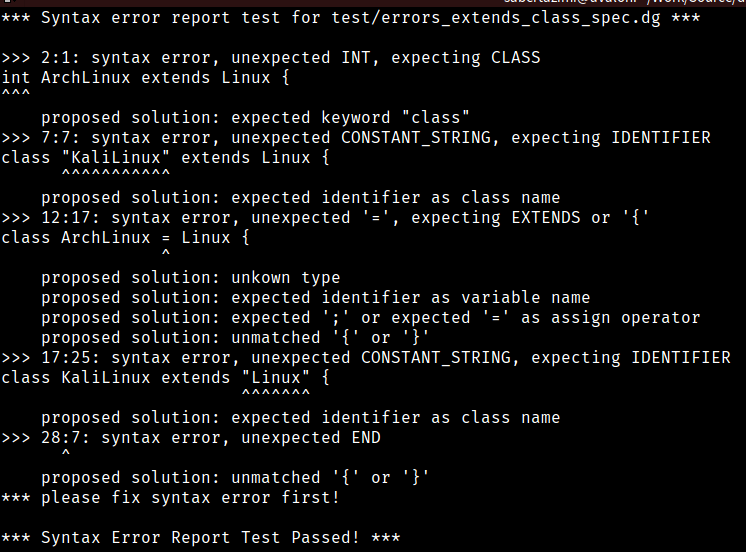


图 2-11 类继承相关语法报错

### 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）

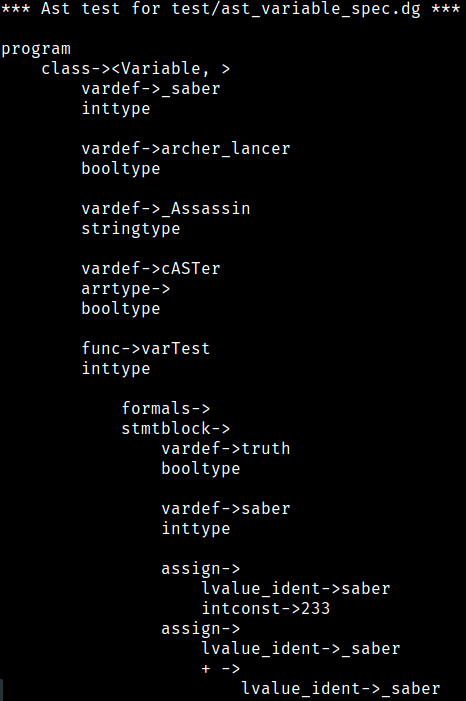


图 2-13 生成抽象语法树（2）

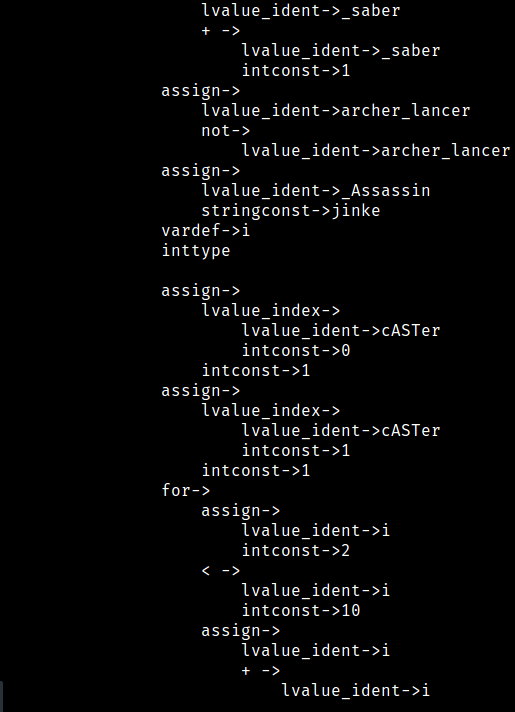


图 2-14 生成抽象语法树（3）

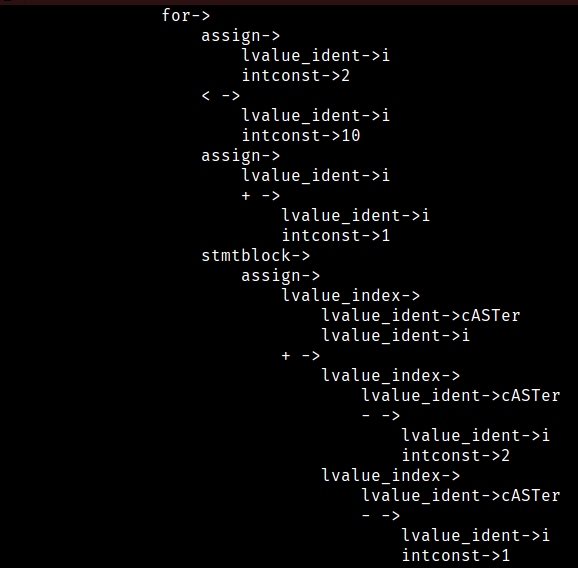


图 2-15 生成抽象语法树（4）

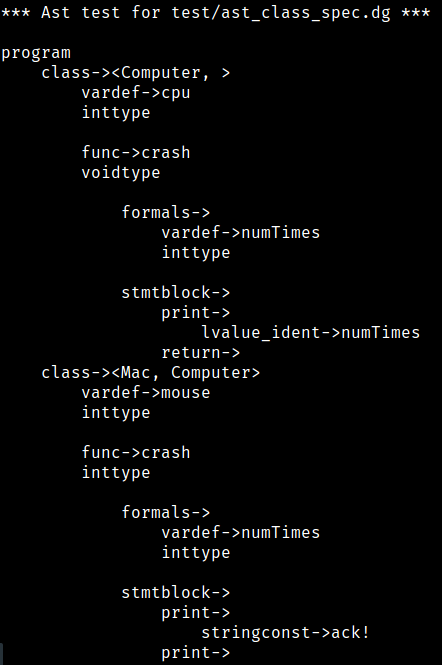


图 2-16 生成抽象语法树（5）

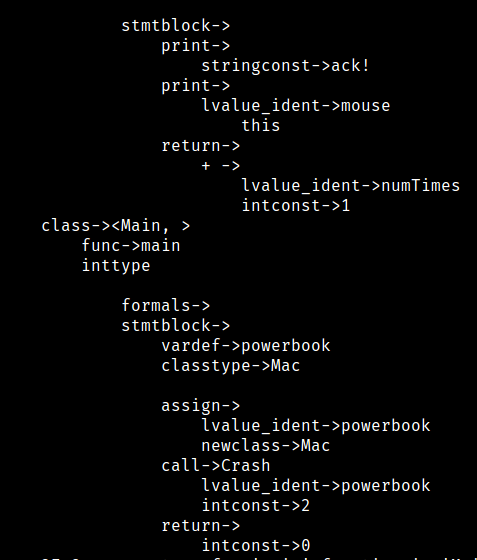


图 2-17 生成抽象语法树（6）

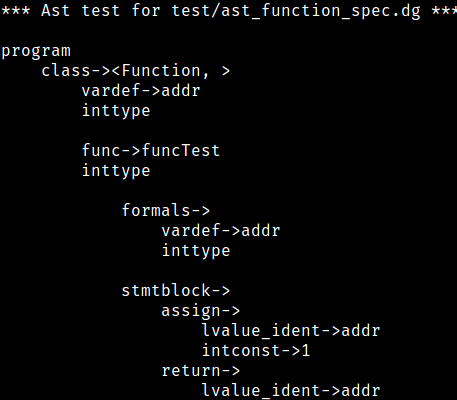


图 2-18 生成抽象语法树（7）

# 3语义分析

3.1语义表示方法描述

语义分析阶段主要有 3 大任务。第一，建立符号表，并进行相关语义检查（如重复定义、使用前未定义等）；第二，进行作用域检查；第三，进行类型检查。除此之外，还有一系列有关面向对象的语义检查任务。

### 3.1.1 符号表与作用域

可以看出，此阶段的基础便是设计符号表与作用域。为不同的符号建立相应的数据结构，描述相关符号的信息。为不同的作用域建立相应的数据结构，描述相关作用域的信息。

由于 Dragon 编译器采用的是多趟编译方式，并且利用 Visitor 类实现了较为简洁的多趟遍历语法树的实现，故此阶段所需的信息可以绑定在语法树的相关结点上，以使得后续阶段进行语法树遍历时能够轻松地取得符号信息与作用域信息。符号主要有 3 类，变量（Variable）、函数（Function）与类（Class），如图 3-1 所示。作用域主要有 4 类，形参作用域（FormalScope）、局部作用域（LocalScope）、类作用域（ClassScope）与全局作用域（GlobalScope），如图 3-2 所示。

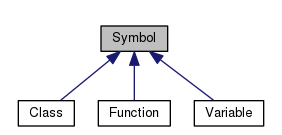


图 3-1 Symbol 继承关系图

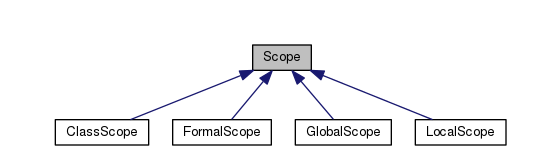


图 3-2 Scope 继承关系图

首先，Program、ClassDef、FuncDef、Block这几个语法树结点是拥有作用域的，于是在 Program、Block 类内部添加作用域成员，如图 3-3 、3-4 所示，在 ClassDef、FuncDef 类内部添加符号成员，如图 3-5、3-6 所示，使得进行相应语义分析时能够快速获取作用域信息。其次，对于 Class、Function类，需要在其上绑定作用域成员，如图 3-7、3-8 所示；对于 Scope 类，需要在其上绑定符号成员，如图 3-9、3-10、3-11 所示，从而达到 Symbol 与 Scope 之间的双向绑定。除此之外，在 VarDef、LValue 结点上绑定符号成员，如图 3-9、3-10 所示，可以在表达式解析、语句解析时快速地获取符号信息。如此以来，只要结点上拥有 Symbol 或 Scope 其一作为成员，即可获取相关的所有符号信息与作用域信息，使得进行多趟分析时能够快速地获取有效信息。

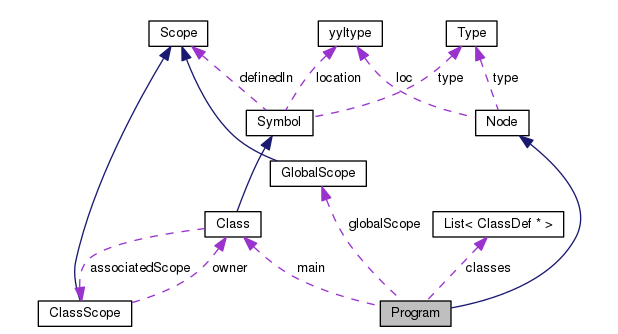


图 3-3 Program 作用域绑定图

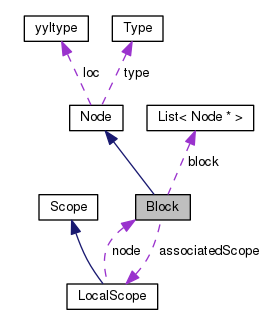


图 3-4 Block 作用域绑定图

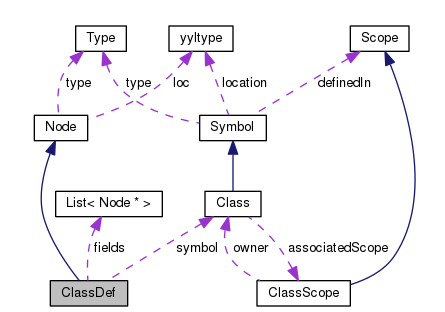


图 3-5 ClassDef 符号绑定图

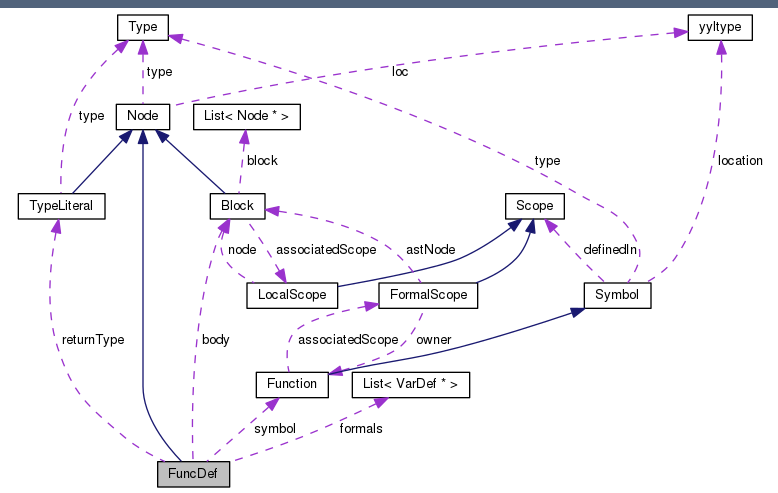


图 3-6 FuncDef 符号绑定图

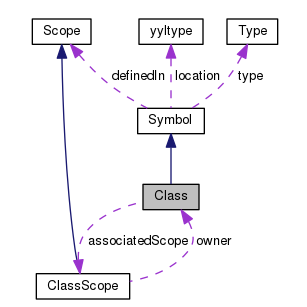


图 3-7 Class 作用域绑定图

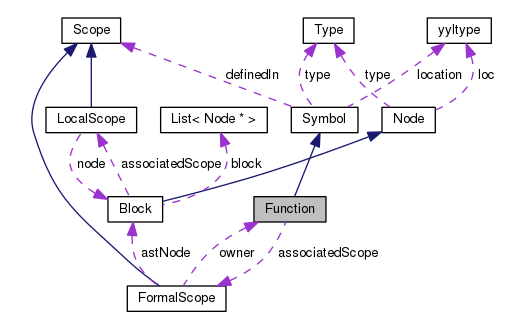


图 3-8 Function 作用域绑定图

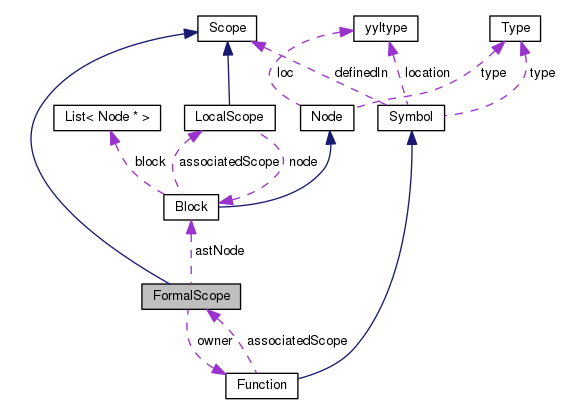


图 3-9 FormalScope 符号绑定图

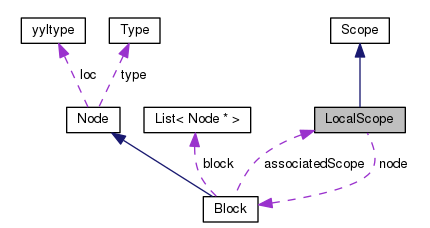


图 3-10 LocalScope 符号绑定图

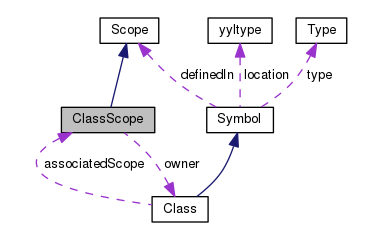


图 3-11 ClassScope 符号绑定图

### 3.1.2 简单类型系统

上述工作完成后，只实现了语义分析中符号检查与作用域检查 2 部分相关数据结构。要正确地完成类型检查，必须建立一个简单的类型系统，并为不同的表达式或语句指定不同的类型。类型主要有4类，基本类型（BaseType）、数组类型（ArrayType）、函数类型（FuncType）、类类型（ClassType），如图 3-12 所示。

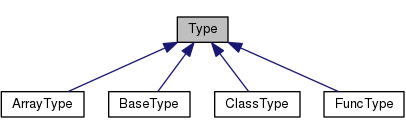


图 3-12 类型系统

BaseType 包括 INT、BOOL、STRING、VOID、NIL、ERROR，如图 3-13 所示，其中 ERROR 类型用于报错，当检查到某个表达式或语句的类型为 ERROR 时，可以在必要的地方进行语义报错。这样便可以将下级的语义分析结果通过 ERROR 向上传递。ArrayType 包括元素类型与，是对 BaseType的一层封装，如图 3-14 所示。FuncType是一个复合类型，其类型定义为 f = (t1 \* t2 \* ... \* tn） -> tn+1，其中括号内部为形参类型，最后一个为返回值类型。通过此种定义，可以将函数类型的判断切分成普通类型的组合判定，从而简化实现，如图 3-15 所示。ClassType （如图 3-16 所示）中包含了相关符号成员的绑定以及父类成员的类型。上述每个类型都有 2 个公共的接口，toString 与 compatible，前者用于打印类型名，后者用于类型兼容性的判定，也是类型检查的关键。

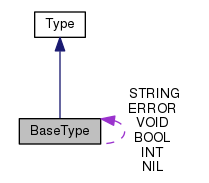


图 3-13 BaseType

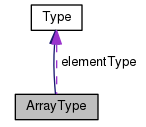


图 3-14 ArrayType

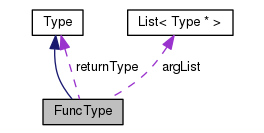


图 3-15 FuncType

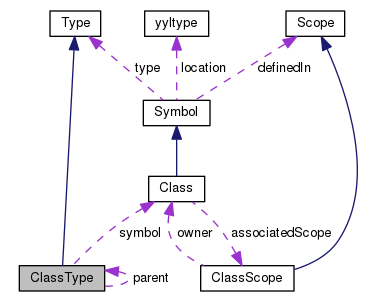


图 3-16 ClassType

BaseType 与 ArrayType 进行类型一致性判定时，只需简单地比较指针引用，因为在编译器全局中只存在唯一的 INT、BOOL等基本类型实例。同理，FuncType 进行类型一致性判定时，仅仅是将形参与返回值类型判定进行与逻辑，即可得到 FuncType的类型一致性。ClassType进行判定时，需要结合继承树规定父类兼容子类，再利用 ClassType 上绑定的 Class Symbol 进行类的判定。

在图 3-3 至 图 3-11 中可以看到，无论是语法树结点还是符号结点，都拥有一个 Type 成员，用于表示相关结点的类型。Type 成员便是实现类型检查的关键。利用多趟递归下降遍历得到的 Symbol 与 Scope，再次对语法树进行递归下降遍历，对必要的表达式或语句进行类型分析与类型匹配等相关检查，即可完成比较完备的类型检查。

### 3.1.3 类型检查

类型检查包括空检查（基本变量、数据元素、类实例不可为 void）、条件表达式必须为 BOOL 型检查、函数形参与实参匹配检查、函数返回类型与返回语句类型匹配检查、赋值表达式类型匹配检查、数组元素访问下标类型检查、数组访问类型检查、二元运算类型匹配检查、一元运算类型匹配检查等。

### 3.1.4 面向对象范式检查

由于 Dragon 是一门支持面向对象范式的语言，所以除了上述常规检查外，还需进行难度最大的面向对象相关语义检查。第一，进行 Main 类以及 main 函数合法性的检查，使得程序拥有唯一且正确的执行入口；第二，继承关系的检查，包括父类存在性检查、继承链合法性检查；第三，进行重载检查，包括变量的重载与函数的重载；第四，进行成员合法性检查，包括成员存在性检查、成员访问控制权限检查。

第一，要实现 Main 类与 main 函数的检查，只需在遍历 Program 结点时，额外地对类名为 Main 的类进行 main 函数合法性的检查即可，main 函数必须存在且无参无返回值。

第二，要实现继承关系的检查，必须为一颗继承树上的各个类标明继承顺序，防止循环继承、逆序继承的发生。实时计算子类在继承树上的层级，并与父类的层级进行比较，若子类的层级高于父类的层级（1 高于 2），则此种继承为非法的。计算继承层级的功能有函数 BuildSymbol::calcOrder 实现。

第三，关于重载检查，需要进行以下一些检查：变量不能重载；不能用变量重载函数或用函数重载变量；重载函数返回值必须相同。

第四，个人认为成员访问是语义检查一环中最为复杂的一个部分。成员访问有 2 种，一种为不加 this 的访问，一种为 this.xx 的访问。其逻辑集中于 TypeCheack::visitIdent 中。为了简化实现，为所有不加 this 的访问添加上 this 绑定，并且对其进行存在性检查。

### 3.1.5 语义表示方法

在建立好上述数据结构及其相互绑定之后，相关语义可分为 3 部分。第一，符号定义相关语义；第二，作用域相关语义；第三，类型相关语义；第四，面向对象相关语义。完成上述语义检查，即可实现一个基本的语义分析器。

3.2符号表结构定义

在 Dragon 编译器中，采用的是多符号表结构，每个 Scope 包含一个键值对集 <name, Symbol>（如图 3-17 所示），用于保存在此 Scope 中所定义的所有符号信息，并在 Scope 上定义 2 个公共借口， declare与lookup，分别用于插入符号与查找符号。具体符号数据结构在 3.1.1 中进行了详细说明，见图 3-1、3-7、3-8。

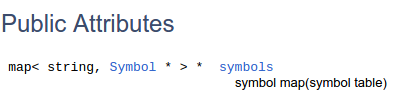


图 3-17 符号表

3.3错误类型码定义

综合 3.1 节中所述的 4 大语义检查，可以得出如下错误类型，如表 3-1 所示：

表 3-1 语义错误综述表

| 错误号 | 错误类型 | 错误描述 |
| --- | --- | --- |
| 1 | 符号 | 重定义类 |
| 2 | 符号 | 重定义方法 |
| 3 | 符号 | 重定义变量 |
| 4 | 作用域 | 未定义类 |
| 5 | 作用域 | 未定义方法 |
| 6 | 作用域 | 未定义变量 |
| 7 | 类型 | 变量为 void 型 |
| 8 | 类型 | 数组元素为 void 型 |
| 9 | 类型 | 数组操作不可对非数组进行 |
| 10 | 类型 | 数组下标不为整数 |
| 11 | 类型 | 条件控制处必须为 bool 型表达式 |
| 12 | 类型 | 函数参数类型不匹配 |
| 13 | 类型 | 函数返回类型不匹配 |
| 14 | 类型 | 赋值类型不匹配 |
| 15 | 类型 | 二元运算类型不匹配 |
| 16 | 类型 | 一元运算类型不匹配 |
| 17 | 面向对象 | 缺少 Main 类或 main 函数 |
| 18 | 面向对象 | 错误的 main 函数原型 |
| 19 | 面向对象 | 错误的重载：变量不可重载 |
| 20 | 面向对象 | 错误的重载：变量与函数不可互相重载 |
| 21 | 面向对象 | 错误的重载：重载函数返回值必须保持一致 |
| 22 | 面向对象 | 未定义父类 |
| 23 | 面向对象 | 对 Main 类的无效继承 |
| 24 | 面向对象 | 错误的继承：循环继承或逆序继承 |
| 25 | 面向对象 | 所访问成员不是方法 |
| 26 | 面向对象 | 访问权限错 |
| 27 | 面向对象 | 访问未定义域 |

3.4 语义分析实现技术

按照上述思路，进行 4 大方面的语义检查，利用 Visitor 类实例对语法树进行 2 次递归下降遍历。第一次遍历，完成符号表与作用域的建立，并进行相关的语义检查；第二次遍历，利用前次遍历的符号信息与作用域信息，进行类型检查与面向对象语义检查，便可实现一个完整的语义分析器。

3.5语义分析结果展示

修改 dragon.cc，开启 SEMA\_DEBUG 宏定义，利用一些不含语法错误的用户代码（test/sema\_\*.dg）进行测试，可得到语义报错以及符号表、作用域信息如下所示：

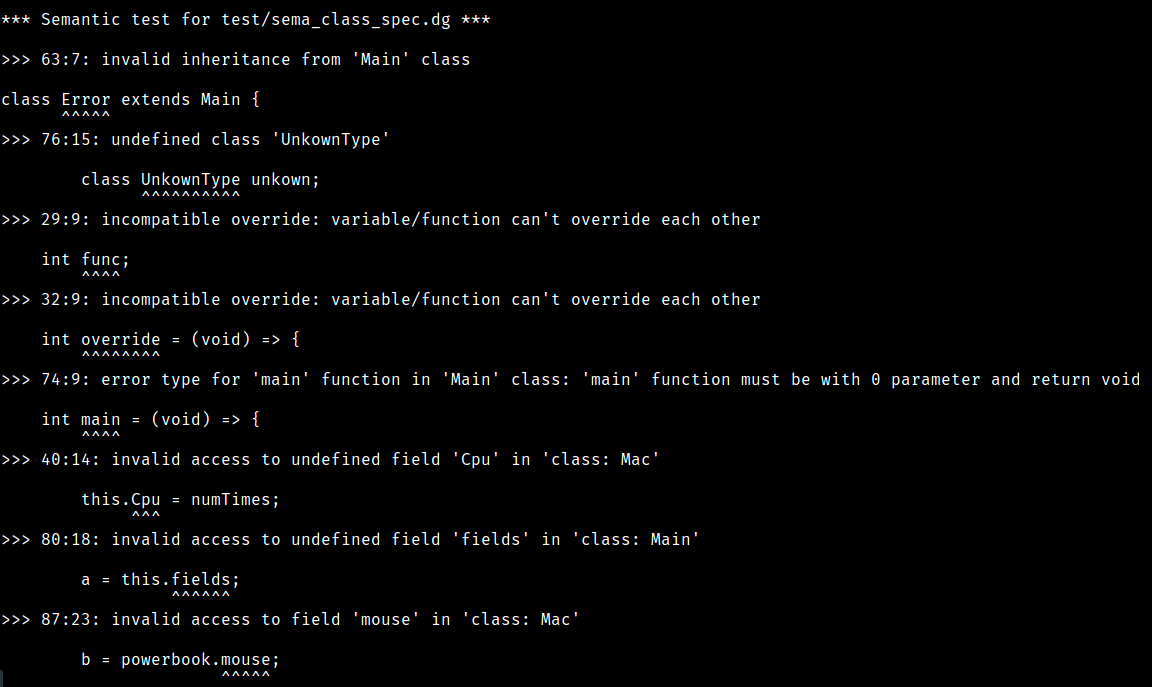


图 3-18 语义分析结果（1）

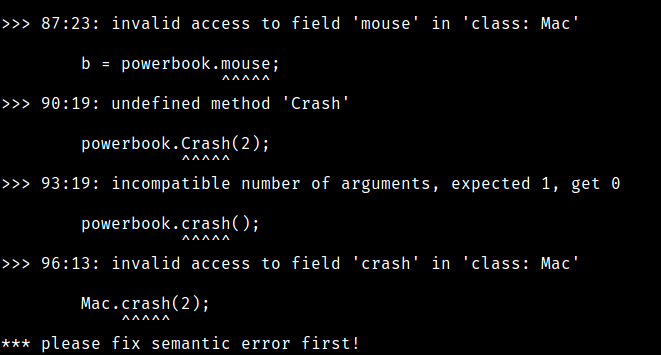


图 3-19 语义分析结果（2）

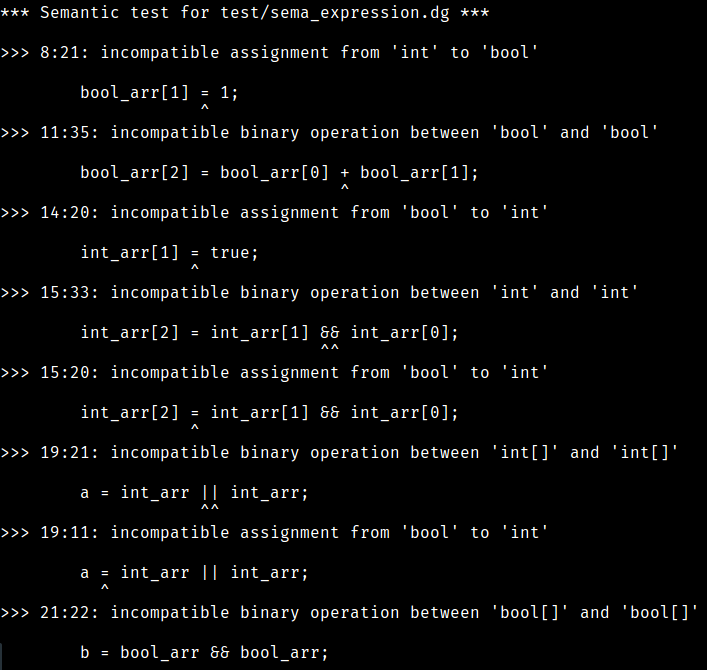


图 3-20 语义分析结果（3）

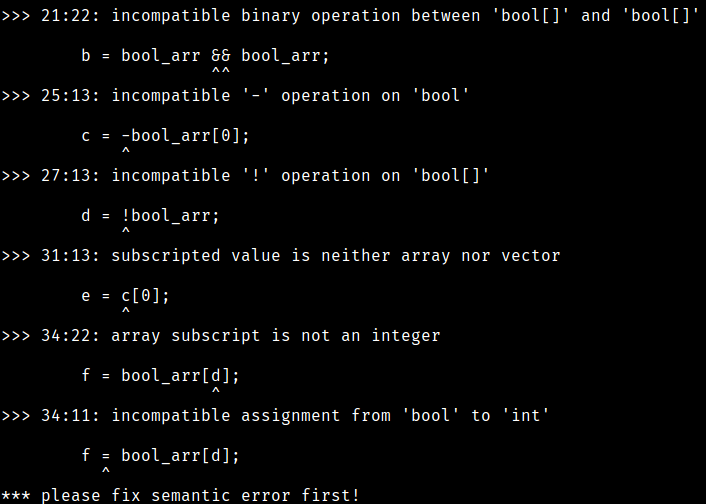


图 3-21 语义分析结果（4）

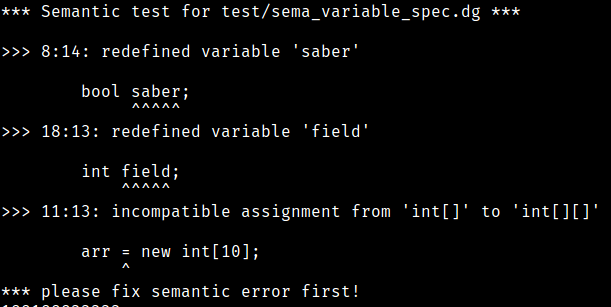


图 3-22 语义分析结果（5）

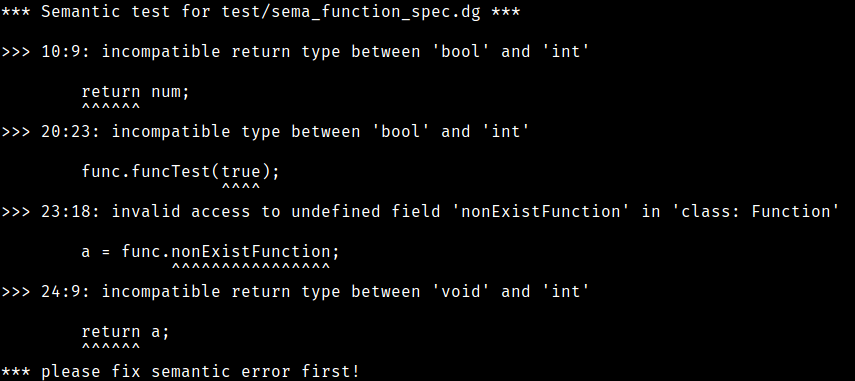


图 3-23 语义分析结果（6）

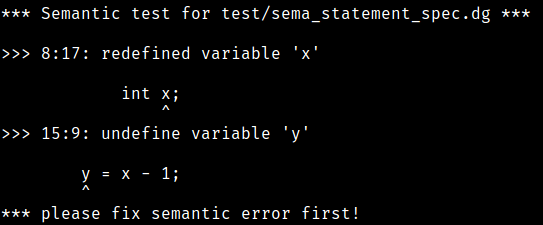


图 3-24 语义分析结果（7）

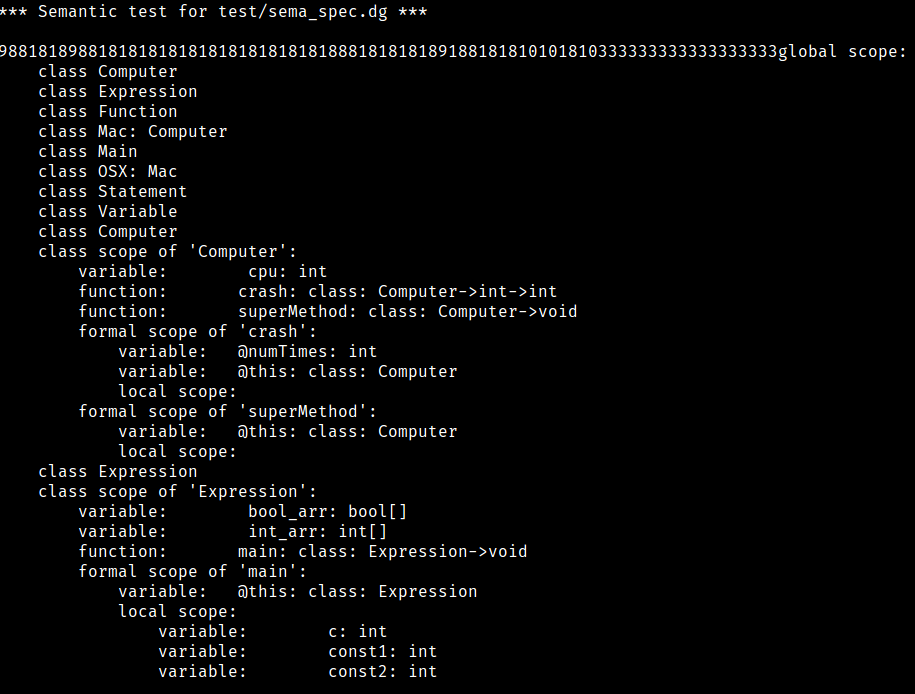


图 3-25 语义分析结果（8）

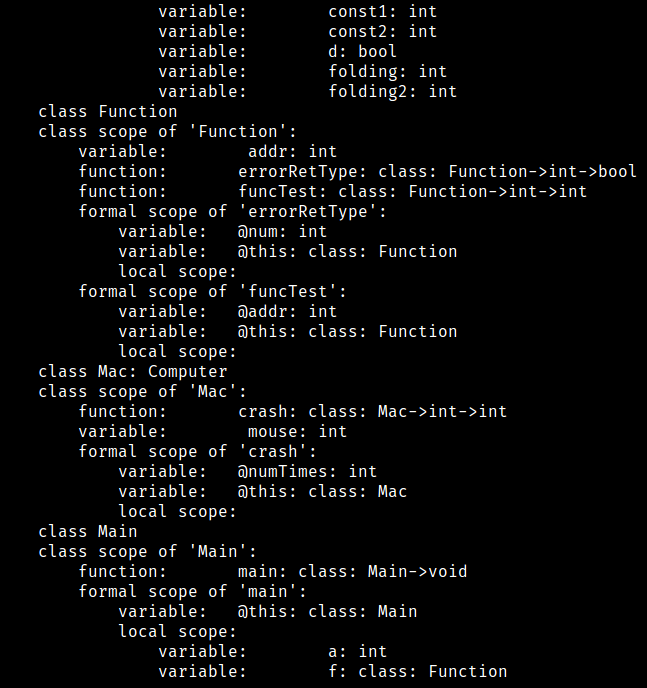


图 3-26 语义分析结果（9）

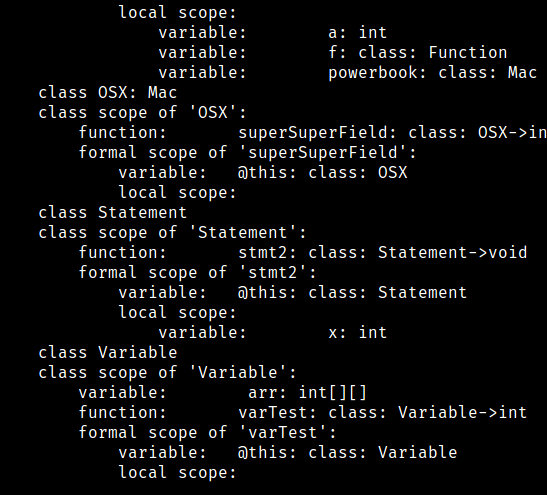


图 3-27 语义分析结果（10）

# 4中间代码生成

### 4.1中间代码格式定义

中间代码采用三地址码，其设计如表 4-1 所示：

表 4-1 Dragon 三地址码表

| 名字 | 形式 |
| --- | --- |
| TAC\_ADD | c := a + b |
| TAC\_SUB | c := a - b |
| TAC\_MUL | c := a \* b |
| TAC\_DIV | c := a / b |
| TAC\_MOD | c := a % b |
| TAC\_NEG | c := -a |
| TAC\_AND | c := a && b |
| TAC\_OR | c := a || b |
| TAC\_NOT | c := !a |
| TAC\_GT | c := a > b |
| TAC\_GE | c := a >= b |
| TAC\_EQ | c := a == b |
| TAC\_NE | c := a != b |
| TAC\_LE | c := a <= b |
| TAC\_LT | c := a < b |
| TAC\_ASSIGN | c := a |
| TAC\_LOAD\_VTBL | x := VTBL <Class> |
| TAC\_INDIRECT\_CALL | x := call a |
| TAC\_DIRECT\_CALL | x := call \_Print |
| TAC\_RETURN | return c |
| TAC\_JMP | jmp \_L1 |
| TAC\_BEQZ | if c == 0 jmp \_L1 |
| TAC\_BNEZ | if c != 0 jmp \_L1 |
| TAC\_LOAD | x := \*(y + 2) |
| TAC\_STORE | \*(x + 4) := y |
| TAC\_LOAD\_IMM4 | x := 233 |
| TAC\_LOAD\_STR\_CONST | x := “dragon” |
| TAC\_MARK | \_L1: |
| TAC\_PARM | parm a |

### 4.2中间代码生成规则定义

由于 Dragon 编译器采用的是多趟遍历语法树的方式，故除了语法树生成是与语法分析同时进行的，需要编写语法树语义动作，其余阶段如语义分析阶段、中间代码生成阶段都是独立于语法分析的，故无需显示定义所谓的中间代码生成规则。

同样地，与语义分析阶段一样，构造一个 Visitor 实例，重写其 vist接口，再次遍历抽象语法树，利用前一阶段得到的符号信息与作用域信息，进行本阶段的翻译工作。将本阶段的翻译工作分成 2 趟。

第一趟，创建 VTable 与 Functy，并将所有成员变量、形参变量、局部变量与一个特定 ID 的 Temp 相互绑定。其形式上，与 Symbol 与 Scope 的相互绑定类似。相关数据结构的相互绑定关系如下所示：

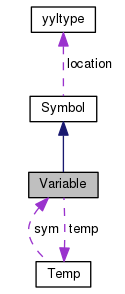


图 4-1 Variable 中绑定 Temp

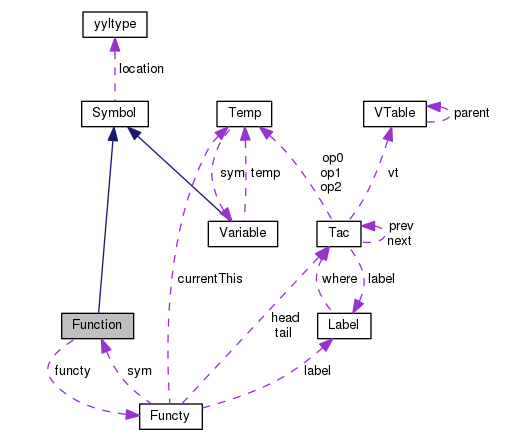


图 4-2 Function 中绑定 Functy

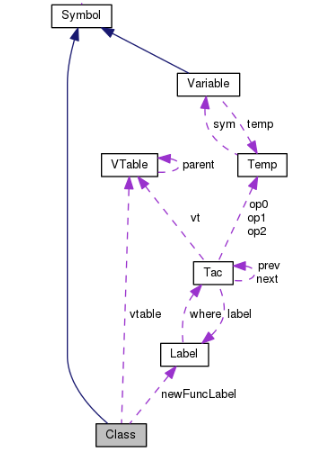


图 4-3 Class 中绑定 VTable

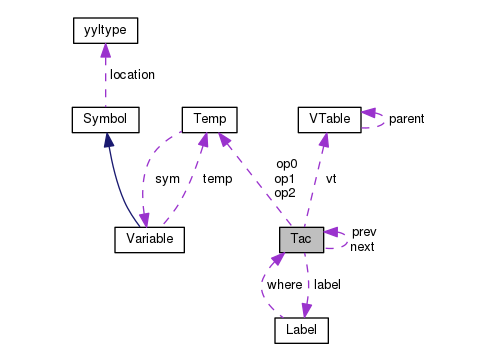


图 4-4

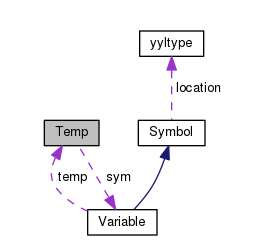


图 4-5 Temp 中绑定 Variable

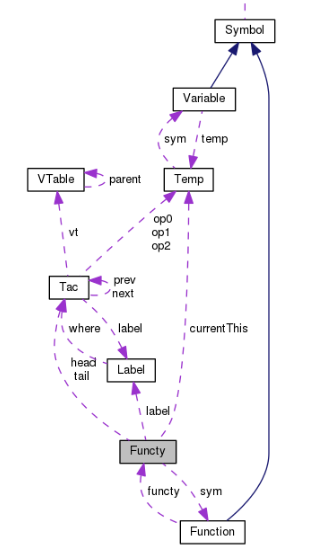


图 4-6 Fcunty 中绑定 Function

第二趟，利用绑定在语法树结点上的 Symbol 与 Scope，得到符号信息、作用域信息以及第一趟翻译中产生的部分三地址码信息，开始正式的中间代码生成，将具体的每个方法中的表达式与语句转成三地址码。

综上所述，对于不同的语法树结点，在 2 趟遍历中所需完成的工作由如下构成：

对于类，第一，创建 VTable，并将父类与本类方法填充至 VTable（完成重载）；第二，为每个类生成一个静态构造函数；第三，计算各个成员的继承顺序，确定每个成员在当前类中的偏移量。

对于方法，第一，创建一个空的 Functy 容器，并将形参对应的 Temp 送入 Functy,局部变量对应的 Temp 送入 Functy；第二，递归下降分析每条语句，在生成新的三地址码的同时，将新的三地址码加入 Functy 中的三地址码双向链表中。

对于语句，由于采用递归下降分析，故无需使用拉链回填，只需按照将 if/while/for 等语句翻译成 goto 形式，然后将其转为对应的三地址码即可。

对于表达式，第一，翻译二元运算/一元运算时，直接翻译成对应的多元三地址码即可；第二，翻译数组元素访问时，需要对地址进行计算，再利用 load/store 访存三地址码进行访问；第三，翻译函数调用时，需要区分直接调用与间接调用，对于静态函数或库函数，采用直接调用函数名翻译成直接调用三地址码，对于方法，需要通过虚地址表计算方法偏移，得到方法地址，再翻译成间接调用三地址码。其中，参数由右至左入栈；第四，对于赋值语句，若左值为数组元素，必须先计算地址，再利用 store 三地址码赋值，若左值为成员，必须显得到成员偏移，再利用 store 三地址码赋值，若左值为形参或局部变量，则直接利用 assign 将值赋给对应 Temp。

对于常量，翻译成 load imm 三地址码即可。

通过 2 趟的递归下降分析，即可完成中间代码生成。最终得到如图 4-7 所示的三地址码结构：

Global

VTables

List <Label>

Functys

形参链表

局部变量链表

三地址码双向链表

图 4-7 三地址码组织结构

### 4.3 中间代码生成过程

利用 4.2 节中所述的 2 趟递归下降遍历，即可完成中间代码的生成

### 4.4代码优化

常量合并优化：a := 1+2 => a := 3，在 visitBinary 时，若两个操作树均为常量，则直接计算出结果并将其符给左边符号所绑定的 Temp。

### 4.5 中间代码生成结果展示

修改 dragon.cc，开启 IR\_DEBUG 宏定义，利用 test/ir\_spec.dg，进行中间代码生成的测试，结果如图 4-8至 4-19所示。

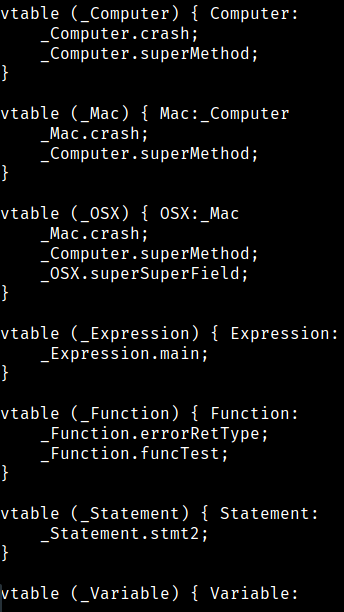


图 4-8 中间代码结果（1）

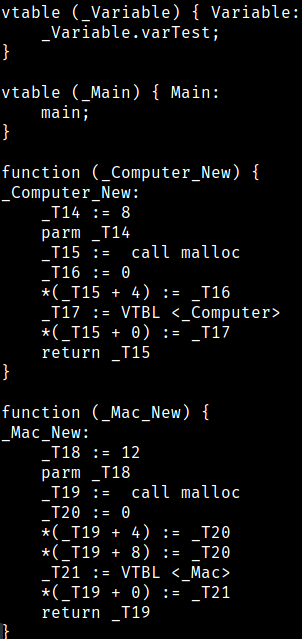


图 4-9 中间代码结果（2）

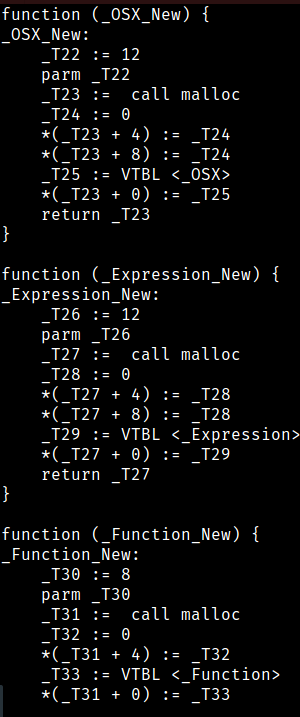


图 4-10 中间代码结果（3）

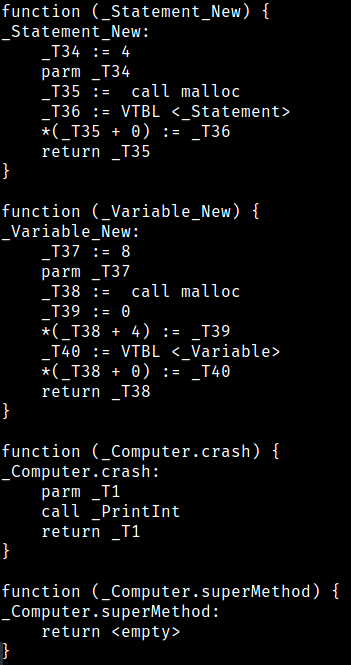


图 4-11 中间代码结果（4）

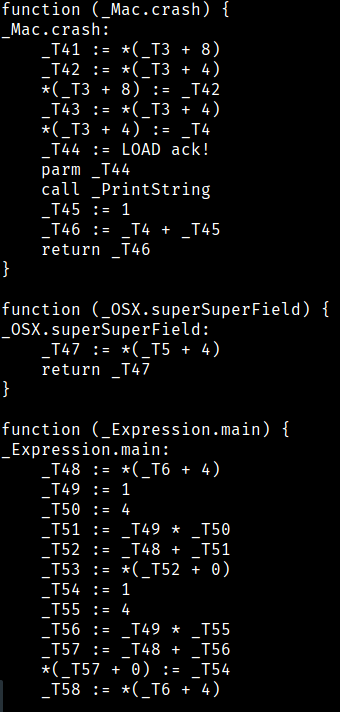


图 4-12 中间代码结果（5）

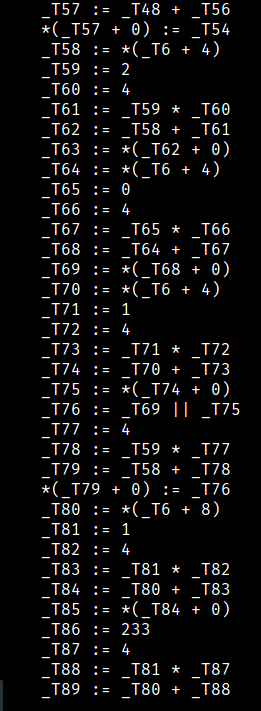


图 4-13 中间代码结果（6）

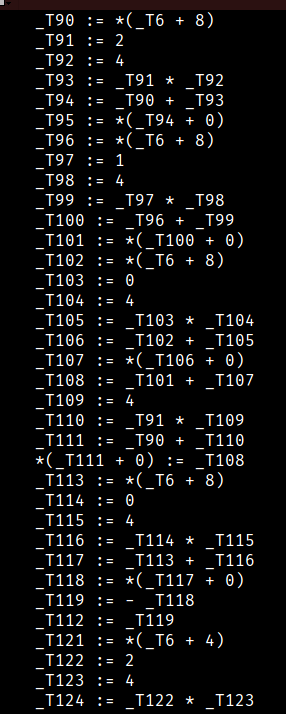


图 4-14 中间代码结果（7）

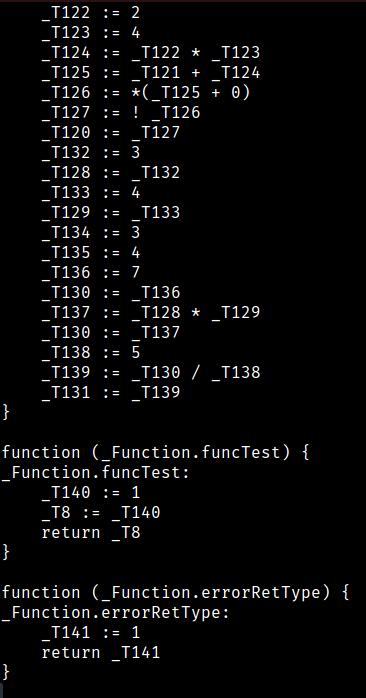


图 4-15 中间代码结果（8）

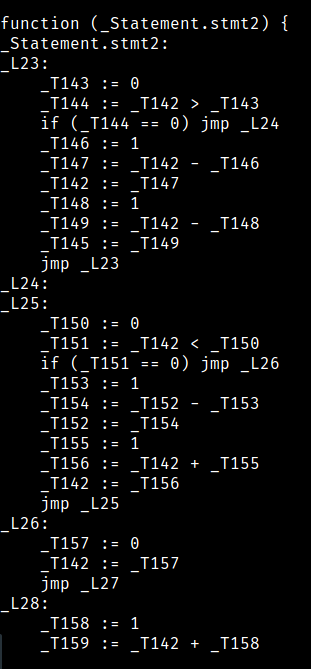


图 4-16 中间代码结果（9）

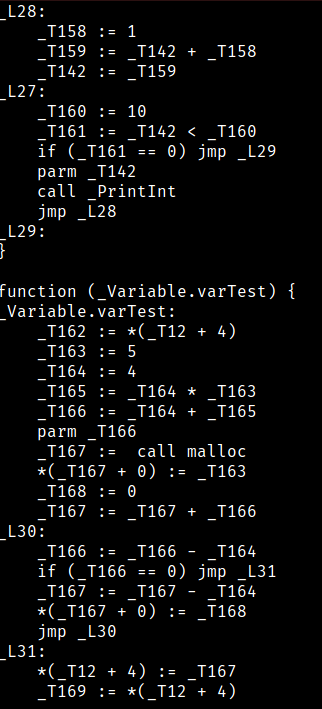


图 4-17 中间代码结果（10）

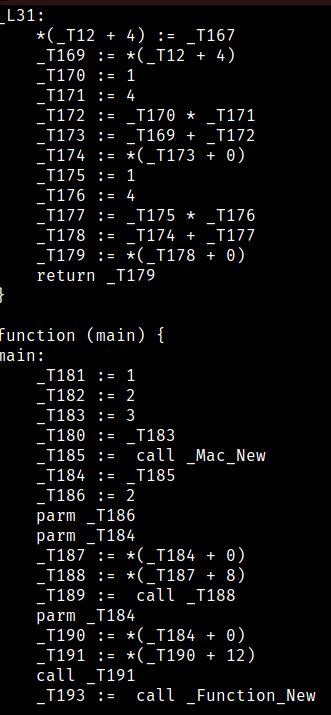


图 4-18 中间代码结果（11）

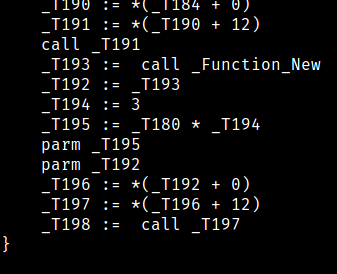


图 4-19 中间代码结果（12）

# 5目标代码生成

### 5.1指令集选择

仔细观察三地址码，可以选择如下指令进行翻译：

表 5-1 Dragon指令选择表

| 名字 | 形式 | 汇编 |
| --- | --- | --- |
| TAC\_ADD | c := a + b | addl |
| TAC\_SUB | c := a - b | subl |
| TAC\_MUL | c := a \* b | imull |
| TAC\_DIV | c := a / b | divl |
| TAC\_MOD | c := a % b | divl |
| TAC\_NEG | c := -a | negl |
| TAC\_AND | c := a && b | andl |
| TAC\_OR | c := a || b | orl |
| TAC\_NOT | c := !a | notl |
| TAC\_GT | c := a > b | cmpl setg |
| TAC\_GE | c := a >= b | cmpl setge |
| TAC\_EQ | c := a == b | cmpl sete |
| TAC\_NE | c := a != b | cmpl setne |
| TAC\_LE | c := a <= b | cmpl setle |
| TAC\_LT | c := a < b | cmpl setl |
| TAC\_ASSIGN | c := a | movl |
| TAC\_LOAD\_VTBL | x := VTBL <Class> | leal |
| TAC\_INDIRECT\_CALL | x := call a | call \*%edi |
| TAC\_DIRECT\_CALL | x := call \_Print | call \_label |
| TAC\_RETURN | return c | movl |
| TAC\_JMP | jmp \_L1 | jmp |
| TAC\_BEQZ | if c == 0 jmp \_L1 | testl je |
| TAC\_BNEZ | if c != 0 jmp \_L1 | testl jne |
| TAC\_LOAD | x := \*(y + 2) | movl |
| TAC\_STORE | \*(x + 4) := y | movl |
| TAC\_LOAD\_IMM4 | x := 233 | movl |
| TAC\_LOAD\_STR\_CONST | x := “dragon” | leal |
| TAC\_MARK | \_L1: | \_L1: |
| TAC\_PARM | parm a | pushl |

### 5.2寄存器分配算法

在生成的汇编代码一开始硬编码如下一段代码：

其开辟了若干虚拟寄存器，每个寄存器32 bit 长，直接将对应编号的 Temp 绑定到对应编号的虚拟寄存器上，即可完成虚拟寄存器分配。

### 5.3 目标代码生成算法

利用 5.1 节所选择指令， 5.2 节所分配寄存器，进行目标代码生成。

第一，为 VTables 中每一个 VTable 生成对应的数据声明。利用 .data、.align、.long，并结合方法名生成函数跳转表。

第二，为 Funcs 中每一个 Functy 生成对应的汇编代码。每个 Functy 翻译含 3 部分，开始阶段、实际代码、结束阶段。开始阶段与结束阶段是公有的保护现场与恢复现场。对于实际代码部分，遍历 Functy 中的每一句三地址码，将其翻译成汇编。

第三，翻译三地址码。通过 5.1 节的分析，可以逐一地将所有三地址码翻译成对应的汇编代码。其中，仅用到 4 个实际寄存器。%esi存放虚拟寄存器组首址，%edi用于存放方法跳转地址或虚拟寄存器编号，%eax与%ebx一般用于计算。特别的，对于访问栈中实参，必须即时写回。

最后，将所有字符串常量翻译为数据段，将内置库函数硬编码为代码段。

至此，完成了汇编代码的生成。

### 5.4 目标代码生成结果展示

修改 dragon.cc，开启 ASM\_DEBUG 宏定义，利用 asm\_\*.dg 进行测试，结果如下：

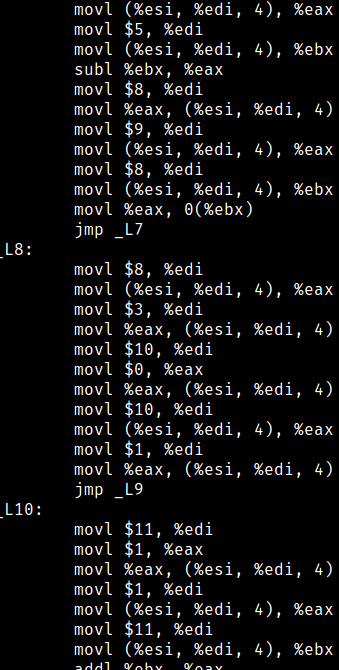


图 5-1 目标代码生成结果（1）

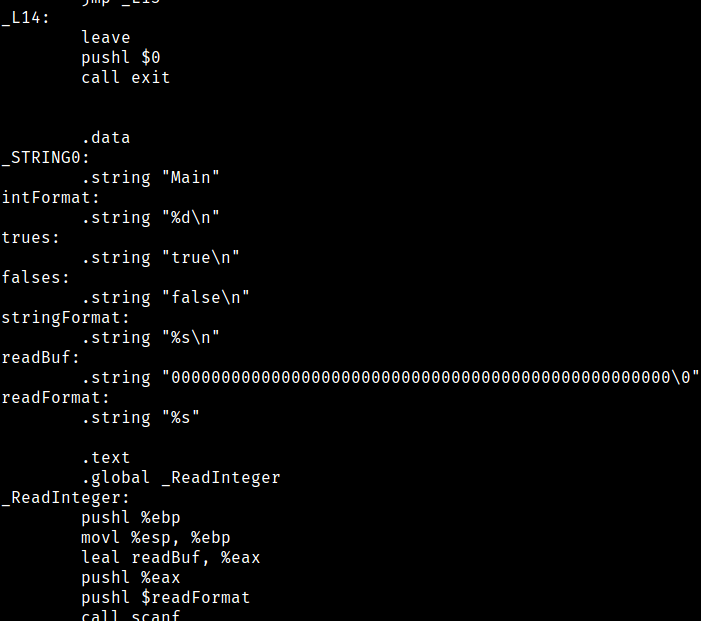


图 5-2 目标代码生成结果（2）

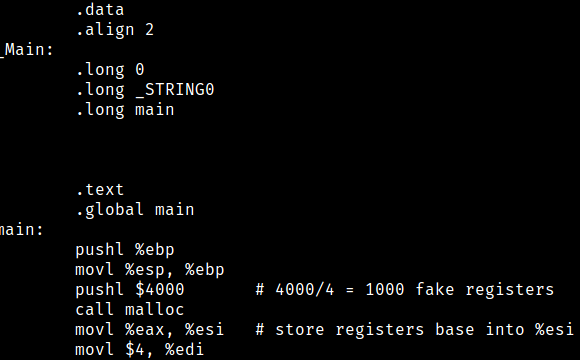


图 5-3 目标代码生成结果（3）



图 5-4 目标代码生成结果（4）

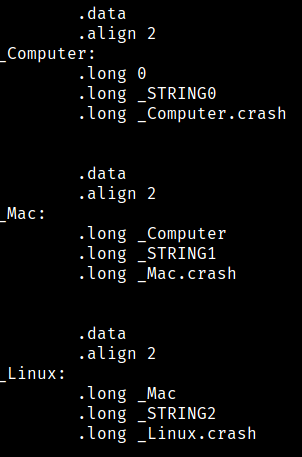


图 5-5 目标代码生成结果（5）

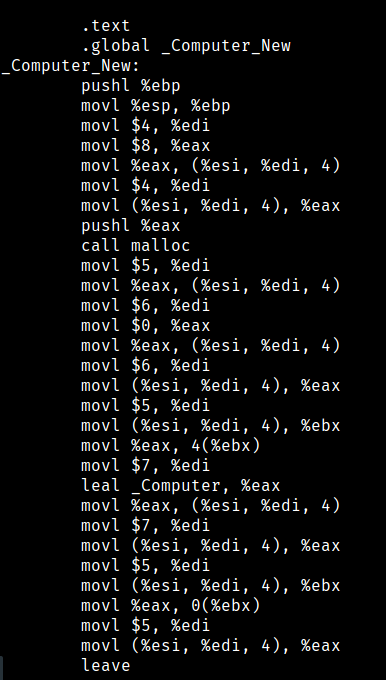


图 5-6 目标代码生成结果（6）

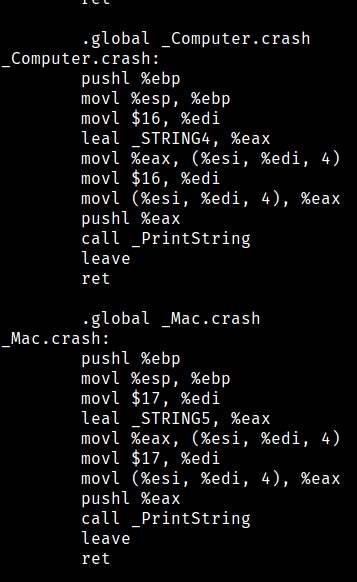


图 5-7 目标代码生成结果（7）

### 5.5目标代码运行结果展示

同样利用上述测试源代码生存二进制文件，执行结果如下：

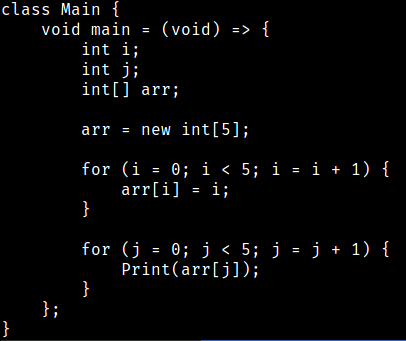


图 5-8 数组程序代码



图 5-9 数组程序结果

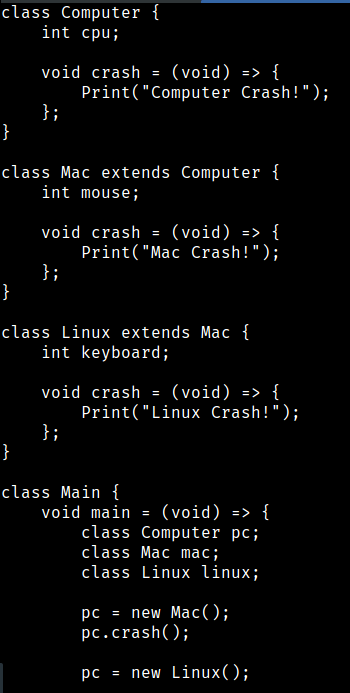


图 5-10 虚函数程序代码

asmclass2

图 5-11 虚函数程序结果

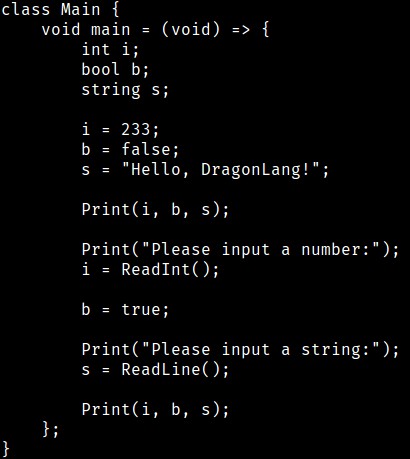


图 5-12 输入输出程序代码

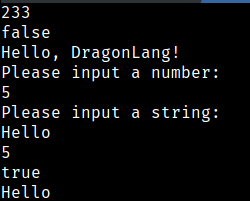


图 5-13 输入输出程序结果

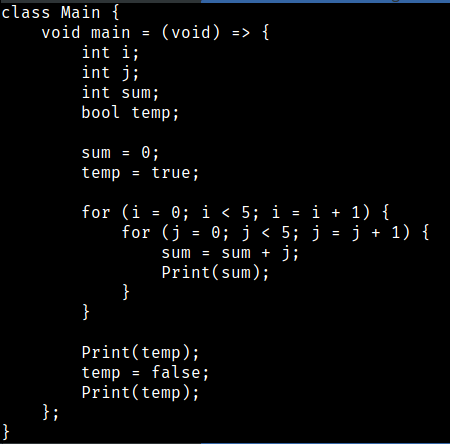


图 5-14 循环程序代码

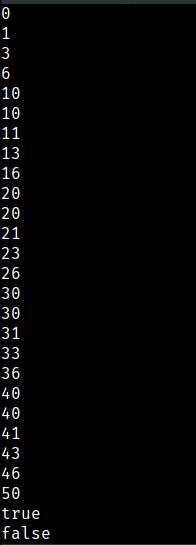


图 5-15 循环程序结果

# 6结束语

6.1 实践课程小结

1. 通过本次编译器实践实验，加深了对正则表达式以及 LALR（1）的理解，加深了对自底向上归约分析的理解；
2. 通过本次编译器实践实验，掌握了 Flex 与 Bison 这两大强有力工具的基本使用，并学会了如何利用其来构建较为复杂的词法分析器与语法分析器；
3. 通过本次编译器实践实验，掌握了基本的抽象语法树构建，加深了对属性文法的理解，并掌握了访问者模式这一高效的递归下降遍历设计模式；
4. 通过本次编译器实践实验，掌握了符号表与作用域的设计与实现，加深了对动态栈以及键值映射表的理解；
5. 通过本次编译器实践实验，掌握了基本语义分析的设计与实现，并理解了一些基本的面向对象语义检查的方法与技巧；
6. 通过本次编译器实践实验，掌握了基本三地址码的实现与完善，并理解了虚地址表在实现面向对象多态（动态绑定）时的重要作用；掌握了抽象语法树到三地址码的转化方法；
7. 通过本次编译器实践实验，掌握了三地址码到汇编指令的转化方法，遗憾的是没有进行基本块划分、控制流图构建与寄存器分配优化。

6.2自己的亲身体会

本次实践是一次难度极大的实验。从一开始的毫无方向，到后来的兀自摸索，直至最后编译器的完成，都耗费了大量精力与时间。利用数千行代码构建起一个编译器，不仅加深了对基本数据结构、基本算法与基本设计模式的理解，也加深了编译原理理论知识的掌握。通过这次实验，不仅对一个玩具编译器的实现有了很好的认识，更极大地提升了自己的编程能力。在不断碰壁，不断尝试，不断摸索的过程中，不仅掌握了编程技法，更掌握了重要的 Debug 技法。在埋头编写编译器的上百个小时里，发现了资料的重要性：有时想破头皮的一个问题，其实所需要的仅仅是一个小小的提示，而通过搜索引擎与代码分享平台，可以找到无穷多的灵感源泉。比如此次实验的三地址码设计，便是借鉴了清华 Mind 语言的三地址码设计。有了这次实验的经历，感觉以后遇到什么难度的实验都不会再放弃了。

**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