

华中科技大学

实践课程报告

题目： 面向对象的 Decaf 语言的编译器设计

课程名称： 编译原理实践

专业班级： 计卓 1401

学 号： U201410081

姓 名： 李冠宇

指导教师： 徐丽萍

报告日期： 2016 年 12 月 26 日

计算机科学与技术学院

目录

1 选题背景	1
1.1 任务.....	1
1.2 目标.....	1
1.3 源语言定义.....	1
2 实验一 词法分析和语法分析	2
2.1 单词文法描述.....	2
2.2 语言文法描述.....	3
2.3 词法分析器的设计.....	6
2.4 语法分析器设计.....	7
2.5 语法分析器实现结果展示.....	11
3 语义分析	16
3.1 语义表示方法描述.....	16
3.2 符号表结构定义.....	17
3.3 错误类型码定义.....	18
3.4 语义分析实现技术.....	21
3.5 语义分析结果展示.....	23
4 中间代码生成	28
4.1 中间代码格式定义.....	28
4.2 中间代码生成规则定义.....	30
4.3 中间代码生成过程.....	33
4.4 代码优化.....	34
4.5 中间代码生成结果展示.....	34
5 目标代码生成	41
5.1 指令集选择.....	41
5.2 寄存器分配算法.....	42
5.3 目标代码生成算法.....	44
5.4 目标代码生成结果展示（综合 4 次实验）	47
5.5 目标代码运行结果展示（综合组原 CPU）	52
6 结束语	55
6.1 实践课程小结.....	55
6.2 自己的亲身体会.....	56
参考文献	58

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

在本次编译原理实践中，选择面向对象的 Decaf 作为源语言，该语言在风格上与 Java 和 C++相似。

2 实验一 词法分析和语法分析

2.1 单词文法描述

单词是 Decaf 语言中具有独立意义的最小单位，可分为 5 大类：关键字（保留字）、运算符、界符、常量和标识符。

Decaf 中的关键字（它们都是保留字）包括：基本类型关键字 `void`、`int`、`float`、`bool` 和 `string` 的，定义类时用到的 `class`、`new`、`extends`、`this` 和 `instanceof`，分支与循环语句涉及的 `if`、`else`、`for`、`while`、`break`，类型修饰关键字 `static`，返回语句 `return`，还有输入输出语句关键字 `Print`、`ReadInteger` 和 `ReadLine`。

Decaf 中的运算符包括：“+ - * / < <= > >= == != && || ! [] ()”，涉及到了算术运算和逻辑运算，值得注意的是，在具体的设计时，小括号 () 和中括号 [] 被当作了运算符而非界符，因为它们在函数调用、数组访问时都有特殊意义，不能当作单纯的分隔符看待。

Decaf 中界符包括：“; , . { }”，小括号 () 和中括号 [] 并不算界符（原因已经解释）。

Decaf 中的常量包括：空指针 (`null`)、布尔常量、浮点数常量、指数形式的浮点数常量、整型常量（十进制）、整型常量（十六进制）以及字符串常量。其中，空指针是一个指针常量表示该指针并不指向任何类型的变量；布尔常量是 `true` 或者 `false`，对应布尔代数中的真或假；浮点数常量可以用一般的小数形式表示，例如“6.66666、8888.8”等，注意，在本词法设计中，小数“.0001”与“1.”都算合法小数；浮点数常量也可以用指数形式即科学记数法表示，例如“1.05e-4”实际上等于用一般的小数形式表示为“0.000105”；一个整型常量可以是十进制整数，一个十进制整数是一个十进制数字（0-9）的序列，例如“68686”；一个整型常量也可以是十六进制整数，十六进制整数必须以“0X”或者“0x”开头（是零，而不是字母 O），后面跟着一个十六进制数字的序列。十六进制数字包括了十进制数字和从 a 到 f 的六个字母（大小写均可）。例如“0x0”和“0X12aE”；一个字符串常量是被一对双引号包围的可打印 ASCII 字符序列，字符串常量中不可以包含换行符，也不可以分成若干行，例如：

```
"this is not a  
valid string constant"
```

Decaf 中的标识符是以字母或下划线开头的字母、数字和下划线的序列。Decaf 是大小写敏感的，例如 `if` 是一个关键字，但是 `IF` 却是一个标识符，`binky`

和 Binky 是两个不同的标识符。

除此之外，在词法分析阶段，为了后续程序设计以及错误提示的需要，本词法分析程序还会识别注释、空白字符、换行符以及错误的标识符、字符串。注释包括单行注释和多行注释，单行注释是以//开头直到该行的结尾，多行注释是用“/*”和“*/”包含的所有字符（自身除外）；空白字符包括空格、制表符；错误的标识符是以数字或者下划线开始的由数字、字母、下划线组成的字符序列；错误的字符串是包含了换行符的“字符串”。

2.2 语言文法描述

文法描述以 EBNF 的扩展形式给出，用到的元符号如表 2-1 所示。

表 2-1 元符号说明表

x (粗体)	表示 x 是一个终结符（即“单词”）。除个别关键字以外，本节中的终结符名字均为小写字母。
y (常规)	表示 y 是一个非终结符。非终结符的名字均为首字母大写。
< x >	表示 0 或 1 个 x 的出现，也就是说， x 是可选的。
x *	表示 0、1 或多个 x 的出现。
x +,	表示一个或多个以逗号分隔的 x
	表示并列关系
ε	表示没有，即不存在任何符号

为了可读性起见，用操作符的词法形式表示它们，例如用 != 而不是 NOT_EQUAL 等由词法分析器返回的代表码来表示不等号。

代码规范如下所示：

```
Program      ::= ClassDef +
VariableDef  ::= Variable ;
Variable     ::= Type identifier
Type         ::= int | bool | string |
               void | class identifier | Type [ ]
Formals      ::= Variable+, | ε
FunctionDef  ::= <static>Type identifier ( Formals ) StmtBlock
ClassDef     ::= class identifier <extends identifier> { Field* }
Field        ::= VariableDef | FunctionDef
StmtBlock    ::= { Stmt* }
Stmt         ::= VariableDef | SimpleStmt ; | IfStmt |
               WhileStmt | ForStmt | BreakStmt ; | ReturnStmt ; |
               PrintStmt ; | StmtBlock
SimpleStmt   ::= LValue = Expr | Call | ε
LValue       ::= <Expr.> identifier | Expr [ Expr ]
Call         ::= <Expr.> identifier ( Actuals )
Actuals      ::= Expr+, | ε
ForStmt      ::= for ( SimpleStmt ; BoolExpr ; SimpleStmt ) Stmt
WhileStmt    ::= while ( BoolExpr ) Stmt
IfStmt       ::= if ( BoolExpr ) Stmt <else Stmt>
```

```

ReturnStmt      ::= return | return Expr
BreakStmt       ::= break
PrintStmt       ::= Print ( Expr+, )
BoolExpr        ::= Expr
Expr             ::= Constant | LValue | this | Call | ( Expr ) |
                  Expr + Expr | Expr - Expr | Expr * Expr |
                  Expr / Expr | Expr % Expr | - Expr |
                  Expr < Expr | Expr <= Expr | Expr > Expr |
                  Expr >= Expr | Expr == Expr | Expr != Expr |
                  Expr && Expr | Expr || Expr | ! Expr |
                  ReadInteger ( ) | ReadLine ( ) |
                  new identifier ( ) | new Type [ Expr ] |
                  instanceof ( Expr , identifier ) |
                  ( class identifier ) Expr
Constant        ::= intConstant | boolConstant |
                  stringConstant | null

```

接下来具体地介绍 Decaf 中有关程序结构、作用域、类型、变量、函数、类和对象的内容。

程序结构:

一个 Decaf 程序是一个类定义的序列，其中每个类定义包含该类的完整描述（也就是说 Decaf 中没有像 C++ 中的前视声明，实际上，Decaf 根本不需要这样的前视声明）。

一个 Decaf 程序应当包含一个名为“Main”主类，主类中应包含一个名为 main、不带任何参数且返回类型为 void 的 static 方法。注意，从父类继承的 main 方法在这里不起作用。

作用域:

Decaf 支持多种层次的作用域。最高层是全局作用域，其中只包含类定义。每个类定义有自己的类作用域。每个函数有一个用于声明参数表的参数作用域和存放函数体的局部作用域。局部作用域中一对大括号建立了一个嵌套的局部作用域。内层作用域屏蔽外层作用域。需要注意以下 5 点：

1. 类，函数和类成员变量可以在声明之前使用，唯一的条件是该符号是在引用处是可访问的。
2. 局部作用域中的变量必须先声明后使用。
3. 同一个作用域中的标识符是唯一的（Decaf 不支持函数重载）。
4. 在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，但不允许在局部作用域中声明与外层的局部作用域或参数作用域中的变量同名的变量。另外，类名不会被任何非全局标识符屏蔽。
5. 不可访问在一个已经关闭的作用域中声明的标识符。

类型:

预定义好的基本类型有 `int`, `bool`, `string` 和 `void`。Decaf 允许类类型。数组类型可以通过任何非 `void` 的元素类型建立起来。

变量：

变量的类型可以是除 `void` 以外的基本类型、数组类型或者类类型之一。在一个类定义内部、但是不在任何函数内声明的变量具有类作用域。在函数参数表中声明的变量具有参数作用域，而在函数体中声明的变量具有局部作用域。一定被声明，则该变量保持可见直到该作用域关闭。需要注意的是：局部变量可以在语句序列的任意地方声明，而且在声明点到该声明所在的作用域末之间的区域可访问。

函数：

函数的定义用于建立函数名字以及与这个名字相关联的类型签名，类型签名包括函数是否是静态的、返回值类型、形参表的大小以及各形参的类型。函数的定义提供类型签名以及组成函数体的语句。

函数必须定义在一个类作用域中，函数之间不允许嵌套。函数可以有零或者多个形参。形参的类型可以是除 `void` 以外的基本类型、数组类型或者类类型。用在形参表中的标识符必须唯一（即形参不能重名）。函数的形参是声明在函数体关联的局部作用域之外的另一个作用域中。函数的返回类型可以是任何的基本类型、数组类型或者类类型。`void` 类型用于指出函数没有返回值。一个函数只能被定义一次。不支持函数的重载（`overload`），函数重载是指使用名字相同但类型签名不同的函数。如果一个函数具有不是 `void` 的返回值类型，则其中的任何 `return` 语句必须返回一个与该返回类型兼容的值。如果一个函数的返回值类型为 `void`，则它能够使用不带参数的 `return` 语句。

函数调用包括从调用方到被调用方传递参数值、执行被调用方的函数体、并返回到调用方（很可能带有返回值）的过程。当一个函数被调用的时候，要传递给他的实参将会被求值并且与对应形参进行绑定。Decaf 中所有的参数和返回值都通过传值的方式进行传递。

所调用的函数必须是有定义的，无论其定义是否出现在调用处之前。函数调用中实参的个数必须与函数所需形参的个数相匹配。函数调用中每个实参的类型必须与对应形参的类型相匹配。函数调用时实参的求值顺序是从左至右。函数调用过程中执行到一个 `return` 语句或者到达函数在源程序中的结尾时把控制权交还给调用方。函数调用结果的类型是函数声明时候的返回值类型。

类：

Decaf 程序中定义一个类的时候将会创建一个新的类型名称以及一个类作用域。一个类定义是一个成员域的列表，每一个成员域要么是一个变量，要么是

一个函数——这些变量有的时候被称为实例变量、成员数据或者属性;这些函数被称为方法或者成员函数。

Decaf 通过一种简单的机制来强制对象的封装:所有的变量都是私有的（访问范围限于定义它的类及其子类，C++中称这种访问级别为 `protected`），所有的方法都是公开的（到处都可以访问）。因此，访问一个对象的状态的唯一手段是通过它的成员函数。所有的类定义都是全局的，也就是说，类定义不能出现在函数中。所有的类必须拥有唯一的名字。一个成员域的名字在同一个类作用域中只能使用一次（即不允许方法同名、变量同名或者变量和方法之间同名）。成员域可以先使用后声明。实例变量的类型可以是除 `void` 以外的基本类型、数组类型或者类类型。在非静态方法中访问同一个类的成员域的时候 “`this.`”的使用是可选的。

对象：

一个变量如果其类型为类类型的话则称为对象，或者该类的实例。对象的访问以引用方式实现。

所有的对象都是使用内置的 `new` 操作符在堆中动态分配的。声明一个对象变量的时候所使用的类名必须有定义。`new` 参数中的类名必须是有定义的。操作符.用于访问一个对象的成员域（变量或者方法）。对于形如 `expr.method()` 这样的方法调用，`expr` 的类型必须是某个类 `T`，`method` 必须是 `T` 的成员方法的名字。对于静态方法，`expr` 可以是一个类名，对于非静态方法，`expr` 必须是一个对象。对于形如 `expr.var` 这样的变量访问，`expr` 的类型必须是某类 `T`，`var` 必须是 `T` 的成员变量的名字，而且这样的访问只能出现在类 `T` 或者其子类的作用域中。对上一条的补充:在类作用域中，你可以通过该类或其子类的任何实例访问该类的私有变量，但不可以访问与该类无关的其他类实例的变量。对象的赋值是浅拷贝（也就是说对象的赋值仅仅复制引用）。对象可以作为函数的参数或者返回值进行传递。对象本身是通过传值的方式进行传递的，但是它通过引用的方式进行访问，因此对其成员函数的更改会反映到调用方那里。

2.3 词法分析器的设计

词法分析器采用的工具是自动化生成工具 GNU Flex，该工具要求词法规则以正则表达式（正规式）给出，并根据给定的词法规则生成相应的词法分析程序。Flex 的原理是有穷自动机，即 Flex 会将用正则表达式表示的词法规则等价转化为相应的有穷自动机 FA，生成对应的词法分析程序。所以，设计词法分析器的关键便是设计能准确识别各类单词的正则表达式。

根据 2.1 的分析，合法单词包括关键字、运算符、界符、常量和标识符，以

及其他一些辅助“单词”。

关键字的正则表达式十分简单，例如：对于关键字 `void` 而言，其正则表达式就是“`void`”（包括引号）。以此类推，不难得到所有关键字的正则表达式。

运算符与界符的正则表达式与关键字的正则表达式类似，都是用引号括起自身即可，于是不再累述。例如 `+` 的正则表达式是“`+`”，`{` 的正则表达式是“`{`”。

常量的表达式相对复杂。对于空指针和布尔表达式，由于本质上它们属于关键字，所以正则表达式是引号加自身；对于一般小数形式的浮点数常量，因为要考虑“`36.`”和“`.36`”这样的特殊形式，所以形式比较复杂，要分情况讨论，并用“`|`”将规则相或，最后设计出的正则表达式为“`[+-]?([0-9]*\.[0-9]+|[0-9]+\.)`”

（不包括引号，下同）；对于指数形式的浮点数常量，指数用整型常量即可，小数部分则跟小数形式的浮点数常量一样考虑即可，不过要排除一些特殊情况，最后设计得到的正则表达式为“`[+-]?[0-9]+\.[0-9]*([eE]{CONSTANTINTD})?`”；对于十进制的整型常量，实际上就是 `0-9` 的序列再加上正负号，所以正则表达式相对简单为“`[+-]?[0-9]+`”；对于十六进制的整型常量，由正负号、`0X` 或 `0x`、`0-9` 或 `a-f` 或 `A-F` 的序列组成，依次可以设计出十六进制的整型变量的正则表达式为“`[+-]?0[xX][0-9a-fA-F]+`”；对于字符串常量，其为由两个双引号扩起来的不包含换行符的字符序列，由此设计其正则表达式为“`\"[^\n]*\"`”（注意双引号要转义）。

根据标识符的定义，设计其正则表达式时需要对开头第一个字符作限制，即第一个字符只能是 `a-z` 或 `A-Z`，由此得到其正则表达式为“`\"[^\n]*\"`”。不过，这里需要注意的是，对标识符的识别规则应放到关键字的识别规则的后面，否则会将所有的关键字当作标识符处理。

空白字符和换行符的正则表达式十分简单，同关键字方法相同。

错误的标识符应该是以数字 `0-9` 或下划线 `_` 开头，所以其正则表达式为“`[0-9_][a-zA-Z0-9_]*`”；错误的字符串这里定义为缺少右引号的字符序列，所以其正则表达式设计为“`\"[^\n]*$`”。

为了能在词法分析和语法分析报错时提供错误的详细位置信息，运用了 Flex 的部分高级特性，例如：开启 `yylineno` 选项，从而全局变量 `yylineno` 会记录当前正在分析的词法单元在源程序中的行号，并由 Flex 自行维护（初值设为 1）。

2.4 语法分析器设计

语法分析器的实现采用的是自动化生成工具 GNU Bison，Bison 可以根据给定的语法规则，自动化生成对应的语法分析程序。但是，语法分析的目的不仅仅是判断源程序的语句是否符合语法规则，还应该（如果符合语法规则）构造源程

序对应的语法分析树，用于编译的后续阶段。Bison 和 Flex 可以无缝对接，即将 Flex 进行词法分析后得到的单词序列作为 Bison 的输入，从而用来进行语法分析。为了实现这一点，需要按照实验指导书上的步骤进行修改和编译，这里不赘述。

设计语法分析器的第一步，便是设计相应的语法规则。语法规则在 2.2 中的 Decaf 语法规范中已经详细给出，这里需要做的便是将语法规则按照 Bison 的标准写成相应的生成式。在具体的转化过程中，我遇到了两个比较大的问题：第一，是上标 “+” 和 “*” 的翻译，根据 EBNF 的规定，上标 “+” 表示对应的元符号出现 1 次或更多次，而上标 “*” 表示对应的元符号出现 0 次或 1 次或更多次；第二，由 Decaf 语法规范直接转化来的生成式存在大量的移进-规约冲突或规约-规约冲突，需要通过一定的方法来消除冲突。

针对第一个问题，以语法规范中的 “Program ::= ClassDef +” 为例进行说明解决方法。解决方法就是添加一个新的非终结符 “ClassDefs”，用 “ClassDefs” 的生成式产生 “ClassDef +”。具体的生成式如下所示：

```
Program: ClassDefs
      ;
ClassDefs: ClassDef
        | ClassDefs ClassDef
      ;
```

对于上标 “*” 的解决方法类似，不过要借助于空规则。

针对第二个问题，即二义性与冲突处理，则是通过显示规定优先级和结合性来解决。经过排查移进-规约冲突和规约-规约冲突的来源，不难发现大部分的冲突来自于运算符，造成的原因便是分析器不知道运算符的优先级和结合性。例如，对于算数表达式 “1+2-3”，分析器并不知道是先算 “1+2” 还是 “2-3”。当然，如果告诉分析器 “+” 和 “-” 都是左结合，那么分析器自然知道是要先计算 “1+2” 而不是 “2-3”。除了算数表达式外，IF-ELSE 语句的语法也会带来移进-规约冲突，因为对于生成式 “IfStmt: KEYIF OLEFTTPRNT BoolExpr ORIGHTTPRNT Stmt” 和 “KEYIF OLEFTTPRNT BoolExpr ORIGHTTPRNT Stmt KEYELSE Stmt”，在分析到 “ORIGHTTPRNT” 时，分析器既可以选择第一条生成式进行规约，也可以选择第二条生成式进行移近，显然根据语法规则和编程常识，应该优先选择第二条生成式也就是移进，所以需要借助 Bison 的特性赋予第二条生成式更高的优先级。为了解决二义性与冲突，在 Bison 中定义的结合性和优先级如下所示。

```
/******结合性*****/
%right OPASSIGN
%left OPOR
%left OPAND
```

```

%left OPLIGHT OPLIGHTEQ OPGREAT OPGREATEQ OPEQUAL
OPNOTEQUAL
%left OPPLUS OPMINUS
%left OPMULTIPLY OPDIVIDE
%right OPNOT
%left SPDOT
%left OPRIGHTBRACKET
%right OPLEFTBRACKET
%left OPRIGHTPRNT
%right OPLEFTPRNT
/*****优先级*****/
%nonassoc LOWER_THAN_ELSE
%nonassoc KEYELSE

```

设计语法分析器的第二步（应首先完成第一步的语法分析部分，即语法的生成式已经无移进-规约冲突且符合语法规则）便是构造语法分析树。根据编译原理课程上所学的知识，为了实现在语法分析的同时构造语法树，应该为语法的每条产生式添加一定的语义动作来完成叶节点的生成和添加到已有语法树。这就要求，终结符和非终结符应该都有相关的“属性”作为语义动作的对象。再结合最终目的是构造语法分析树，不难想到这里的“属性”应该选择语法分析树的节点指针。考虑到语法分析树是一棵各节点度都不相等的树，所以决定采用“孩子兄弟法”表示语法分析树，这样可以通过二叉树的相关操作（自己比较熟悉）来实现语法分析树的相关操作。

为此，设计并定义语法分析树的节点结构体（C 语言），如图 2-1 所示。

```

typedef struct GrammarTreeNode
{
    int line;           // the number of its line
    char* name;         // the name of this grammar unit
    struct GrammarTreeNode* lchild;
    struct GrammarTreeNode* rchild;
    union               // the value of this grammar unit
    {
        char* string_value;
        int int_value;
        float float_value;
    };
} GrammarTreeNode;

```

图 2-1 语法分析树节点结构体

如图 2-1 所示，该节点结构体包括语法单元的行号（错误定位时使用）line、语法单元的名称 name、左右孩子节点和一个用来记录语法单元（常量）“值”的局部联合。

定义了语法分析树节点结合体，还需设计并实现语法分析树的生成函数和遍历函数，其中，语法分析树生成函数是生成一颗以生成式左部语法单元（非终结符）为根节点、生成式右部所有语法单元为子节点的语法树并返回根节点指针。两个函数的函数原型如图 2-2 所示，具体的实现参考附录中的源代码。生成函数的设计思想是借助 C 语言的可变参数（因为不确定产生式的右部到底有多少个语法单元），将生成式右部所有语法单元对应的语法树指针作为可变参数传入。

```
/* Create GrammarTree Using Chile-Brother representation
 * name: the name of the grammar unit
 * num: the number of grammar unit in the variable parameter list
 */
GrammarTree CreateGrammarTree(char* name, int num, ...);

/* Traverse GrammarTree Using Pre-Order
 * tree: the grammar tree
 * level: the number of the level
 */
void TraverseGrammerTree(GrammarTree gmtree, int level);
```

图 2-2 语法分析树生成函数和遍历函数的函数原型

完成了语法分析树相关结构体和函数的定义和实现后，需要将其与 Bison 中的生成式代码相结合。为此，需要完成两项工作：

一是将所有终结符和非终结符的属性值类型声明为语法树节点的指针类型，具体的做法如图 2-3 所示（只给出了部分声明）；

```
%union {
    GrammarTree grammar_tree;
}

%token <grammar_tree> SPSEMICOLON SPCOMMA SPDOT SPLEFTBRACE SPRIGHTBRACE
%token <grammar_tree> OPLEFTPRNT OPRIGHTPRNT OPLEFTBRACKET OPRIGHTBRACKET
```

图 2-3 声明语法单元（部分）的属性值类型

二是为每条生成式添加语义动作，用来根据生成式构造语法树。需要注意的是，对于最顶层的生成式“Program: ClassDefs”的语义动作，不仅要实现构造语法树的功能，还应根据语法分析的结果打印最终的语法分析树。对于一般生成式的语义动作如图 2-4 所示，对于顶层生成式“Program: ClassDefs”的语义动作如图 2-5 所示。

```
SimpleStmt: { $$ = CreateGrammarTree("SimpleStmt", 0, -1); }
           | LValue OPASSIGN Expr { $$ = CreateGrammarTree("SimpleStmt", 3, $1, $2, $3); }
           | Call { $$ = CreateGrammarTree("SimpleStmt", 1, $1); }
           ;
```

图 2-4 简单语句生成式对应的语义动作

```

Program: ClassDefs {
    $$ = CreateGrammarTree("Program", 1, $1);
    if (two_tuples_trigger)
    {
        printf("_____\n\n");
        printf("The two-tuples of \"Lexical Analyzing\" are printed!\n");
        printf("_____\n");
    }
    if (!gmerror) {
        printf("\nNow print the grammar-tree of \"Grammar Analyzing\":\n");
        printf("_____\n\n");
        TraverseGrammarTree($$, 0);
        printf("_____\n\n");
        printf("The grammar-tree of \"Grammar Analyzing\" is printed!\n\n");
    }
}
;

```

图 2-5 顶层生成式对应的语义动作

2.5 语法分析器实现结果展示

在展示词法分析和语法分析的测试结果前，首先给出所用的测试用例，是一份合法的 Decaf 代码，如图 2-6 所示。

```

1 class test
2 {
3     int m;
4     int a;
5     void display(int a, int b)
6     {
7         int x; //test
8         float f;
9         bool b, d, e;
10        string s;
11        x = 211;
12        x = -0x1a;
13        f = 0.001;
14        f = -1.2E-1;
15        b = true;
16        s = "just test";
17        s = null;
18        x = a + b;
19        if (a == b)
20        {
21            if (c == d)
22                Print("ifif");
23            else
24                Print("ifelse");
25        }
26        else
27        {
28            if (c == d)
29                Print("elseif");
30            else
31                Print("elseelse");
32        }
33    }
34 }

```

图 2-6 实验 1 测试用例代码

在这份测试用例代码中，测试了词法分析器对各种类型常量的支持，并通过 if-else 嵌套分支语句、类型定义语句、函数定义语句等测试了语法分析器的语法

分析功能。在接下来的测试结果中，会针对测试的不同方面对测试用例稍作修改。

词法分析功能测试：

词法分析器的输出为二元组，具体的形式为“(词法单元名称，词法类型)”。词法分析程序将源程序中的每个词法单元按照二元组的形式打印出来，上述测试用例的二元组输出结果如图 2-7、2-8 和 2-9 所示。

```
[dracula@localhost]~/Documents/IT_Study/compiler/lab1
$ ./parser test.decaf

Now print the two-tuples of "Lexical Analyzing":

(class, KEYCLASS)
(test, IDENTIFIER)
(\n, EOL)
({, SPLEFTBRACE)
(\n, EOL)
(int, TYPEINTEGER)
(m, IDENTIFIER)
(;;, SPSEMICOLON)
(\n, EOL)
(int, TYPEINTEGER)
(a, IDENTIFIER)
(;;, SPSEMICOLON)
(\n, EOL)
(void, TYPEVOID)
(display, IDENTIFIER)
((, OPLEFTPRNT)
(int, TYPEINTEGER)
(a, IDENTIFIER)
(,, SPCOMMA)
(int, TYPEINTEGER)
(b, IDENTIFIER)
(,), OPRIGHTPRNT)
(\n, EOL)
({, SPLEFTBRACE)
(\n, EOL)
(int, TYPEINTEGER)
(x, IDENTIFIER)
(;;, SPSEMICOLON)
(//test, COMMENT)
(\n, EOL)
(float, TYPEFLOAT)
(f, IDENTIFIER)
(;;, SPSEMICOLON)
(\n, EOL)
(bool, TYPEBOOL)
(b, IDENTIFIER)
(,, SPCOMMA)
(d, IDENTIFIER)
(,, SPCOMMA)

(e, IDENTIFIER)
(;;, SPSEMICOLON)
(\n, EOL)
(string, TYPESTRING)
(s, IDENTIFIER)
(;;, SPSEMICOLON)
(\n, EOL)
(x, IDENTIFIER)
(=, OPASSIGN)
(211, CONSTANTINTD)
(;;, SPSEMICOLON)
(\n, EOL)
(x, IDENTIFIER)
(=, OPASSIGN)
(-0x1a, CONSTANTINTH)
(;;, SPSEMICOLON)
(\n, EOL)
(f, IDENTIFIER)
(=, OPASSIGN)
(0.001, CONSTANTFLOAT)
(;;, SPSEMICOLON)
(\n, EOL)
(f, IDENTIFIER)
(=, OPASSIGN)
(-1.2e-1, CONSTANTFLOATSC)
(;;, SPSEMICOLON)
(\n, EOL)
(b, IDENTIFIER)
(=, OPASSIGN)
(true, CONSTANTBOOL)
(;;, SPSEMICOLON)
(\n, EOL)
(s, IDENTIFIER)
(=, OPASSIGN)
("just test", CONSTANTSTRING)
(;;, SPSEMICOLON)
(\n, EOL)
(s, IDENTIFIER)
(=, OPASSIGN)
(null, CONSTANTNULL)
(;;, SPSEMICOLON)
(\n, EOL)
(x, IDENTIFIER)
(=, OPASSIGN)
(a, IDENTIFIER)
```

图 2-7 词法分析结果（1）

从图 2-7 中的右面一栏可以发现，词法分析器成功地识别并记录了十进制整型常量、十六进制整型常量、一般小数形式浮点数常量、指数形式浮点数常量、布尔常量以及字符串常量等，分别对应源程序中的 211、-0x1a、0.001、-1.2e-1、true、“just test”等。从图 2-7、2-8 和 2-9 中也不难得出，词法分析器可以准确识别出前述讨论的 5 大类词法单元（关键字、运算符、界符、常量和标识符）。

需要注意的是，词法分析器也可以识别出空白字符，但是因为源程序中的空白字符太多，不方便打印和展示，因此在打印的二元组中，并无空白字符（空格和制表符）。


```

("ifif", CONSTANTSTRING)      (+, OPPLUS)
(), OPRIGHTPRNT                (b, IDENTIFIER)
(;, SPSEMICOLON)               (;, SPSEMICOLON)
\n, EOL                        \n, EOL)
(else, KEYELSE)                (if, KEYIF)
\n, EOL                        ((, OPLEFTPRNT)
(Print, KEYPRINT)              (a, IDENTIFIER)
((, OPLEFTPRNT)                (==, OPEQUAL)
("ifelse", CONSTANTSTRING)    (b, IDENTIFIER)
(), OPRIGHTPRNT                (), OPRIGHTPRNT)
(;, SPSEMICOLON)              \n, EOL)
\n, EOL                        ({, SPLEFTBRACE)
}, SPRIGHTBRACE)              \n, EOL)
\n, EOL                        (if, KEYIF)
(else, KEYELSE)                ((, OPLEFTPRNT)
\n, EOL                        (c, IDENTIFIER)
({, SPLEFTBRACE)              (==, OPEQUAL)
\n, EOL                        (d, IDENTIFIER)
(if, KEYIF)                    (), OPRIGHTPRNT)
((, OPLEFTPRNT)               \n, EOL)
(c, IDENTIFIER)               (Print, KEYPRINT)
(==, OPEQUAL)                 ((, OPLEFTPRNT)
(d, IDENTIFIER)               ("ifif", CONSTANTSTRING)

```

图 2-8 词法分析结果 (2)

```

(), OPRIGHTPRNT)
\n, EOL)
(Print, KEYPRINT)
((, OPLEFTPRNT)
("elseif", CONSTANTSTRING)
(), OPRIGHTPRNT)
(;, SPSEMICOLON)
\n, EOL)
(else, KEYELSE)
\n, EOL)
(Print, KEYPRINT)
((, OPLEFTPRNT)
("elseelse", CONSTANTSTRING)
(), OPRIGHTPRNT)
(;, SPSEMICOLON)
\n, EOL)
}, SPRIGHTBRACE)
\n, EOL)
}, SPRIGHTBRACE)
\n, EOL)
}, SPRIGHTBRACE)
\n, EOL)
}, SPRIGHTBRACE)
\n, EOL)

The two-tuples of "Lexical Analyzing" are printed!

```

图 2-9 词法分析结果 (3)

词法分析器支持三种词法错误的识别和报错：错误的标识符、少一个引号的字符串以及不能识别的词法单元。如果将测试用例中的“int x;”修改为“int 1x”，即错误的标识符，则词法分析器会给出错误提示如图 2-10 所示。

```
Error type A at Line 7 column 13: Wrong format of identifier.
```

图 2-10 错误的标识符错误提示

如果将测试用例中的字符串删除一个引号，即将 16 行改为“s = “just test;”，

即错误的字符串，则词法分析器会给出错误提示如图 2-11 所示。

```
Error type A at Line 16 column 23: Missing "".
```

图 2-11 错误的字符串错误提示

通过图 2-10 和图 2-11 不难看出，词法分析器不仅支持错误提示，还能准确定位错误的词法单元的位置信息，可以给出其在源程序中的行数和列数，从而方便程序员的修改。

词法分析功能测试：

语法分析的输出为语法树。语法树是一颗多叉树，但在设计时采用了孩子兄弟法，因此其在程序中的存储形式为二叉树。打印语法树时，需要对其进行深度优先遍历，显示所有的节点（语法单元：非终结符或者终结符）。为了方便打印和展示，将测试用例中的声明语句部分用多行注释的方法注释掉。此时，在词法分析打印的二元组中会存在这样一个二元组：

```
(/*  
    int x; //test  
    float f;  
    bool b, d, e;  
    string s;  
    x = 211;  
    x = -0x1a;  
    f = 0.001;  
    f = -1.2e-1;  
    b = true;  
    s = "just test";  
    s = null;  
    x = a + b;  
    */, COMMENT)
```

表明声明语句部分已被识别成注释。

由于语法分析树的节点数较多，因此极其不方便截图展示，所以图 2-12 和图 2-13 只给出了语法树的开始和结束部分。

```
Now print the grammar-tree of "Grammar Analyzing":  
  
Program (1)  
  ClassDefs (1)  
    ClassDef (1)  
      KEYCLASS (1)  
      IDENTIFIER: test  
      SPLEFTBRACE (2)  
      Fields (3)  
        Field (3)  
          VariableDef (3)  
            Variable (3)  
              Type (3)  
                TYPEINTEGER (3)  
                IDENTIFIER: m  
                SPSEMICOLON (3)  
          Fields (4)  
            Field (4)  
              VariableDef (4)  
                Variable (4)  
                  Type (4)  
                    TYPEINTEGER (4)  
                    IDENTIFIER: a  
                    SPSEMICOLON (4)
```

图 2-12 语法树开始部分

```

      PRINTSTMT (33)
      KEYPRINT (33)
      OPLEFTPRNT (33)
      Exprs (33)
      Expr (33)
      Constant (33)
      CONSTANTSTRING: "elseelse"
      OPRIGHTPRNT (33)
      SPSEMICOLON (33)
      Stmt (Epsilon)
      SPRIGHTBRACE (34)
      Stmt (Epsilon)
      SPRIGHTBRACE (35)
      Fields (Epsilon)
      SPRIGHTBRACE (36)

```

The grammar-tree of "Grammar Analyzing" is printed!

图 2-13 语法树结束部分

语法单元的缩进表示语法单元所在的层次，语法单元后面的括号中给出的是语法单元的行号，将其和测试用例相比对，可以证明打印出的语法树正确无误。注意的是，对于常量，会给出常量的具体值。例如在图 2-13 中，语法单元“CONSTANTSTRING”后面紧跟的就是字符串常量的值，即“elseelse”。

语法分析阶段还可以对源程序的语法正确性检测，并根据检测结果给出相应的报错提示。例如，如果不小心漏打了语句后面的分号，语法分析程序会给出提示，提示缺少分号（语法单元名称为“SPSEMICOLON”），如图 2-14 所示。

```

[dracula@Lenovo-PC]~/Documents/IT_Study/compiler/lab1
$ ./parser test.decaf
Error type B at Line 25 Column 13: syntax error, unexpected KEYELSE, expecting SPSEMICOLON.
Error type B at Line 32 Column 13: syntax error, unexpected KEYELSE, expecting SPSEMICOLON.
Error type B at Line 37 Column 2: syntax error, unexpected $end.
[dracula@Lenovo-PC]~/Documents/IT_Study/compiler/lab1
$

```

图 2-14 语法错误报错提示

通过图 2-14 也可以看出，语法分析阶段不仅支持报错，还支持对错误位置的准确定位，并有一定的错误恢复能力。错误恢复能力是指语法分析在检测到语法错误时，可以记录该错误并恢复，进而分析剩余的程序找出其余语法错误。对于图 2-14，其显示源程序中有 3 处语法错误，分别对应缺少相应的分号。

3 语义分析

3.1 语义表示方法描述

进行语义分析的理论工具是属性文法，属性值可以分成不相交的两类：综合属性和继承属性。在语法树中，一个节点的综合属性值是从其子节点的属性值计算而来的，而一个结点的继承属性则是由该节点的父节点和兄弟结点的属性值计算而来的。为了能够自底向上地进行语义分析，所以在节点的属性方面，主要选择的是综合属性。相对于继承属性，由于一个节点的综合属性的值只依赖于其子节点的属性值，与自底向上的分析方向完全相同，因此更适合自底向上的语义分析。

从编程实现的角度来看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的词法分析模块或者后面的中间代码生成模块。不过，由于这样做其牵扯到的内容较多而且较为复杂，所以还是决定将语义分析单独作为一个模块。在完成了实验一的词法分析和语法分析后，将会得到源程序的语法分析树，自底向上的语义分析完全可以通过对语法树进行深度优先遍历来实现。这样，语义分析便可以作为独立模块而单独进行，而不用在语法分析时进行。实际上，遍历语法树进行语义分析和在语法分析时进行语义分析完全是等价的。

为了进行语义分析，除了需要语法分析阶段得到的语法分析树，还需要借助符号表进行类型检查等分析。

符号表的构建时机也是一个需要考虑的问题。符号表可以在语法分析时与语法分析树一同构建，也可以在语法分析树得到后对语法树进行一次或多次遍历来构建。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表。

除了借助符号表进行类型检查外，语义分析阶段还有一个重要职责是符号作用域合法性的检查。对符号作用域合法性的检查，仅仅只靠符号表是不够的，需要通过一个作用域栈来维持当前对程序可见的作用域。因此，符号表也需要按照作用域来进行分类，具体的介绍见 3.2 节。

3.2 符号表结构定义

3.1 节中已经对符号表的分类动机进行了简略介绍，即为了实现符号作用域合法性的检查，需要将符号表按作用域进行分类。在 Decaf 中，共有 4 种类型的作用域，即全局（Global）作用域、类（Class）作用域、形参（Formal）作用域以及局部（Local）作用域。各个作用域对应符号表的连接关系图 3-1 所示（该符号表对应的源程序为 3.5 节中的测试用例）。

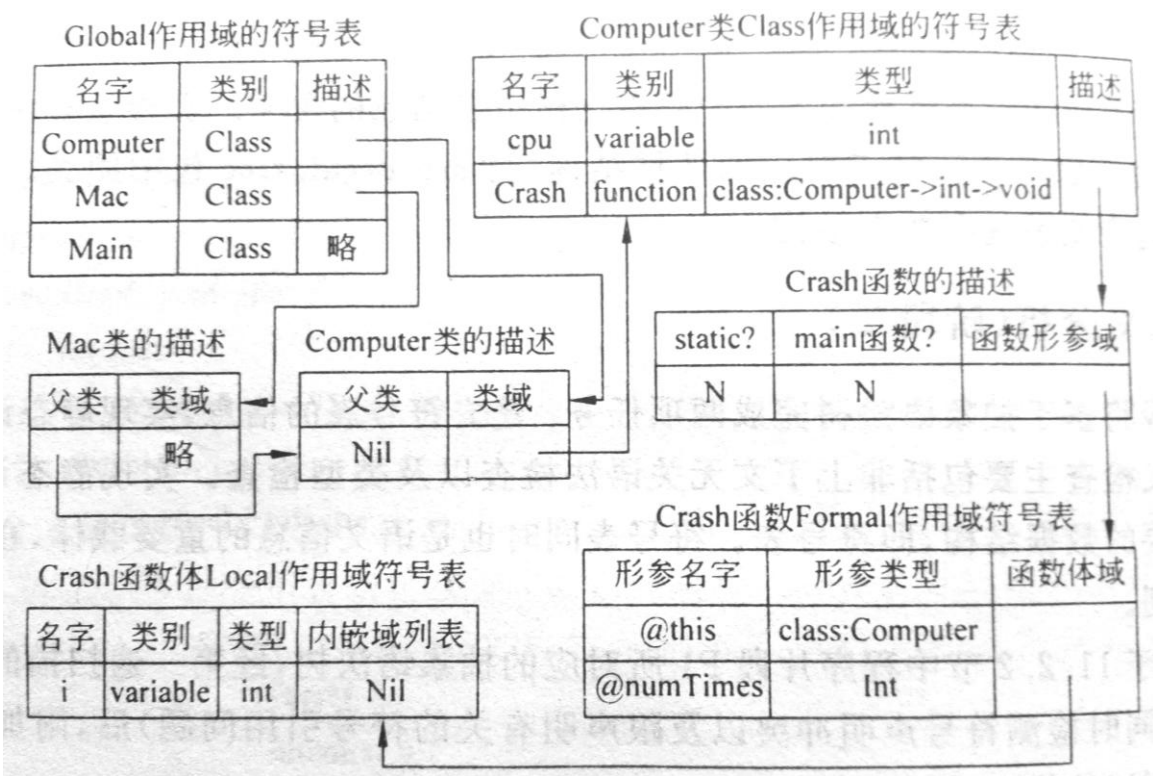


图 3-1 符号表结构示意图

对于全局作用域的符号表，通过图 3-1 可知，全局作用域符号表的每条表项应该包括符号的名字（类名）、符号类别（类）、描述（这里是指类的描述，具体实现时，用一个包含类的描述和函数的描述的联合实现）。若采用链表组织符号表，则全局作用域的符号表的结构体可设计为图 3-2 所示。

```
typedef struct GlobalScopeEntry
{
    char* name;
    DecafCategory category;
    Depictor depictor;
    struct GlobalScopeEntry* next;
} GlobalScopeEntry;

typedef GlobalScopeEntry* GlobalScope;
```

图 3-2 全局作用域符号表结构体定义

对于类作用域的符号表，通过图 3-1 可知，类作用域符号表的每条表项应该包括符号的名字（变量名或函数名）、符号类别（变量或函数）、符号类型、描述（对于变量而言该项为空，对于函数该项指向函数的描述）。若采用链表组织符号表，则类作用域的符号表的结构体可设计为图 3-3 所示。

```
typedef struct ClassScopeEntry
{
    char* name;
    DecafCategory category;
    char* type;
    Depictor depictor;
    struct ClassScopeEntry* next;
} ClassScopeEntry;

typedef ClassScopeEntry* ClassScope;
```

图 3-3 类作用域符号表结构体定义

对于形参作用域的符号表，通过图 3-1 可知，形参作用域符号表的每条表项应该包括符号的名字（形参名）、符号类型（形参类型）、函数体域（指向函数体局部作用域符号表的指针）。若采用链表组织符号表，则形参作用域的符号表的结构体可设计为图 3-4 所示。

```
typedef struct FormalScopeEntry
{
    char* name;
    char* type;
    LocalScope functionscope;
    struct FormalScopeEntry* next;
} FormalScopeEntry;

typedef FormalScopeEntry* FormalScope;
```

图 3-4 形参作用域符号表结构体定义

对于局部作用域的符号表，通过图 3-1 可知，局部作用域符号表的每条表项应该包括符号的名字（变量名或语句块临时名）、符号类别（变量或语句块）、符号类型、内嵌域列表（指向语句块所在的局部作用域）。若采用链表组织符号表，则局部作用域的符号表的结构体可设计为图 3-5 所示。

```
typedef struct LocalScopeEntry
{
    char* name;
    DecafCategory category;
    char* type;
    struct LocalScopeEntry* embededscope;
    struct LocalScopeEntry* next;
} LocalScopeEntry;

typedef struct LocalScopeEntry* LocalScope;
```

图 3-5 局部作用域符号表结构体定义

除了为各个作用域的符号表定义结构外，在 3.1 节中也提到了，需要通过一个作用域栈来维持当前对程序可见的作用域，进而实现符号作用域合法性的检

查。当处理到程序的某一位置时，可以访问的作用域称为开作用域，否则为闭作用域。需要建立一个栈来管理整个程序的作用域：每打开一个作用域，就把该作用域压入栈中；每关闭一个作用域，就从栈顶弹出该作用域。这样，这个作用域栈中就记录着当前所有打开的作用域的信息，栈顶元素就是当前最内层的作用域。查找一个变量时，按照自栈顶向下的顺序查找栈中各作用域的符号表，最先找到的就是最靠近内层的变量。作用域栈的结构示意图如图 3-6 所示。

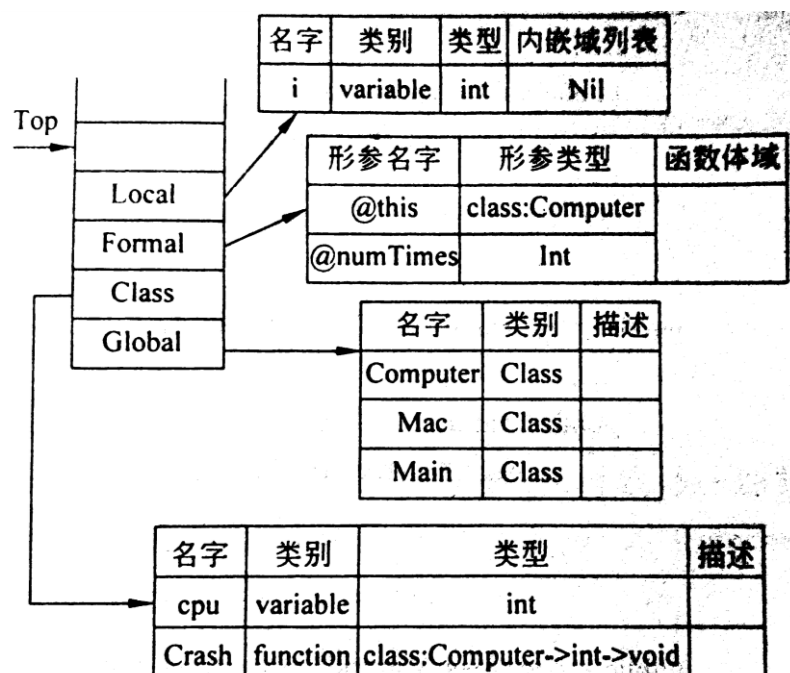


图 3-6 作用域栈的结构示意图

注意到，作用域是分 4 个类别的，对应 4 种不同的结构体，但是在 C 语言里并没有像 C++ 模板这样的特性，所以为了能让栈的元素可以是任意一种作用域，需要再定义一个结构体 Scope，该结构包含 4 种不同类型的作用域，具体的定义如图 3-7 所示。

```
typedef struct Scope
{
    ScopeType type;
    union
    {
        LocalScope localscope;
        FormalScope formalscope;
        ClassScope classscope;
        GlobalScope globalscope;
    };
} Scope;
```

图 3-7 汇总作用域结构体定义

这样便可以将 Scope 类型的变量作为作用域栈的元素。对于作用域栈结构体的定义仿照之前数据结构课程里整型栈的定义方式，栈结构体有 3 个成员：栈底指针、栈顶指针、栈的大小，因此得到作用域栈结构体的定义如图 3-8 所示。


```
typedef struct ScopeStack
{
    Scope* base;
    Scope* top;
    int stacksize;
} ScopeStack;
```

图 3-8 作用域结构体定义

3.3 错误类型码定义

本编译器的语义分析阶段可识别的错误类型如下：

- (1) 错误类型 1：类未定义（整个源程序中）就实例化对象。
- (2) 错误类型 2：类体中有重名的成员（对象或函数）。
- (3) 错误类型 3：定义了重名类。
- (4) 错误类型 4：子类所要继承的父类没有被定义（整个程序中）。
- (5) 错误类型 5：变量未定义便使用。
- (6) 错误类型 6：类被当作左值。
- (7) 错误类型 7：赋值操作等号两边类型不匹配。
- (8) 错误类型 8：对象调用了类中没定义的属性（数据成员）。
- (9) 错误类型 9：非对象变量使用运算符“.”。
- (10) 错误类型 10：函数未定义便使用。
- (11) 错误类型 11：将函数名作为形参使用（作用域问题）。
- (12) 错误类型 12：将函数名作为类名或变量名使用（作用域问题）。
- (13) 错误类型 13：对象调用了类中没定义的方法（函数成员）。

需要注意的是，虽然上述仅列出了 13 个错误类型，但实际上错误类型数远大于 13。因为对于错误类型 7 而言，可进一步划分，例如函数返回值与变量类型不匹配、父类对象赋给子类等。

除此之外，本编译器的语义分析阶段所作的定义和声明都是全局范围的，也就是说，变量和函数可以在未声明或定义前便可使用，但是必须要有声明或定义。同样的道理，子类所继承的父类可以在子类前定义，也可以在子类后定义，但一定要定义父类，否则，语义分析会出错。

语义分析阶段发现的语义错误也会被准确定位，编译器会给出错误的具体位置，包括行号和列号。编译器还可以针对具体的错误类型，给出相关提示。例如如果赋值运算等号两边的表达式类型不匹配，编译器会给出提示，告之等号两边的表达式的具体类型是什么。

总之，错误类型和错误提示会帮助程序员准确地找到错误位置和发现错误原因，从而快速修正。

3.4 语义分析实现技术

虽然在 3.1 节中对语义分析的过程作了简要的概述，但不够详细，接下来将详细阐述语义分析阶段的具体实现和各个过程。

语义分析的第一步便是构建符号表。构建符号表的重要意义在 3.1 与 3.2 节中已经说明，这里不再累述。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表操作，因此没有必要单独进行一次语法树的遍历来构造符号表。

综上，符号表的建立被放在了语法分析阶段与语法分析树的构造同时进行。对于使用了 **Bison** 的语法分析，其过程是自底向上的规约，所以在语法分析阶段，当规约说明语句（包括类的定义、变量定义、函数定义等）时，便为相应的符号构造一个对应作用域符号表的表项（指向该表项的指针被放到了语法分析树的对应节点中）；当说明语句作为生成式的右部作为进一步规约时，根据生成式的语义，将说明语句所带的符号表表项传递给左部语法单元或者归并左边的所有说明语句的符号表表项，将归并后的符号表传递给左部语法单元。以此类推，各符号表便可以从底层逐层传递和累积，形成最终的符号表。这里有个特殊情况，那就是继承关系中的父类指向（指针），由于是自底向上的分析无法在分析时便能确定父类的位置（有可能先引用，后定义），所以需要一个小遍历确定。

以类作用域符号表的建立来说明这一过程，关于类的定义的生成式和对应的语义动作如图 3-9 所示。

```
ClassDef: KEYCLASS IDENTIFIER SPLEFTBRACE Fields SPRIGHTBRACE {
    $$ = CreateGrammarTree("ClassDef", 5, $1, $2, $3, $4, $5);
    AddThisType($4->classscope, $2->string_value);
    $$->depictor = CreateClassDepictor(NULL, $4->classscope);
}
| KEYCLASS IDENTIFIER KEYEXTENDS IDENTIFIER SPLEFTBRACE Fields SPRIGHTBRACE {
    $$ = CreateGrammarTree("ClassDef", 7, $1, $2, $3, $4, $5, $6, $7);
    AddThisType($6->classscope, $2->string_value);
    $$->depictor = CreateClassDepictor($4->string_value, $6->classscope);
}
| error SPRIGHTBRACE { $$ = CreateGrammarTree("ClassDef", 1, $2); gerror += 1; }
;

Fields: { $$ = CreateGrammarTree("Fields", 0, -1); $$->classscope = NULL; }
| Field Fields {
    $$ = CreateGrammarTree("Fields", 2, $1, $2);
    $$->classscope = MergeClassScope($1->classscope, $2->classscope);
}
;

Field: VariableDef {
    $$ = CreateGrammarTree("Field", 1, $1);
    Depictor depictor;
    $$->classscope = AddIntoClass($$->classscope, $1->lchild->lchild->rchild->string_value, Variable, $1->lchild->lchild->string_value, depictor);
}
| FunctionDef {
    $$ = CreateGrammarTree("Field", 1, $1);
    $$->classscope = $1->classscope;
}
;
```

图 3-9 类定义相关生成式及语义动作

对于左部语法单元为 Field 的生成式而言，对于变量定义是创建一个符号表项然后传递给 Field，对于函数定义则是将右部语法单元的符号表表项传递给 Field。对于左部语法单元为 Fields 的生成式而言，是将右部语法单元所带的符号表表项归并后传递给 Fields。

语义分析的第二步便是静态语义错误检查。本编译器在语义分析阶段的静态语义检查可以分为两大类，第一类是符号作用域相关检查，第二类是类型检查，这两类检查可以在一次语法树遍历过程内完成。当完成语法分析后，语法分析树与符号表便会均构建完毕，可以随时调用。本编译器将静态语义错误检查单独写成了一个模块，在语法分析阶段规约到最顶层生成式时，会调用该模块进行语义检查。语义检查整体上就是一次语法树的遍历，自底向上进行语义分析，但是由于类型检查和作用域分析都要用到符号表，所以在遍历语法树的同时需要维护一个作用域栈，遍历的同时进行出栈和入栈操作。

对于作用域相关检查，最重要的便是在语法树遍历时维护作用域栈。当遇到类的定义、函数定义、对象声明、函数调用等需要开启一个新的作用域时，便将相应的作用域入栈，当遇到类似大括号这种语法单元时，就要将关闭当前的作用域关闭，即从作用域栈中弹出闭作用域。当遇到使用一个符号时，就从栈中从栈顶向栈底搜索，第一个搜索到的就是当前作用域最先访问的符号，没找到表明该符号没有声明过。

例如，在遇到类的定义时，需要将对应的类作用域符号表压入作用域栈中，如果该类有父类，同时还应将其父类的类作用域的符号表压入作用域栈，相关代码如图 3-10 所示。

```
else if (strcmp(tree->name, "ClassDef"))
{
    Scope result;
    ClassDepictor classdsp = tree->depictor.classdsp;
    scope.type = ScopeClass;
    result = TraverseScopeStack(*stack, tree->lchild->rchild->string_value);
    // Judge whether the class has been declared
    if (result.type != ScopeNone && result.globalscope->depictor.classdsp != tree->depictor.classdsp)
    {
        smerror += 1;
        printf("Semantic Error 3 at Line %d Column %d: The Class \"%s\" has been Declared!\n",
            tree->line, tree->lchild->rchild->column, tree->lchild->rchild->string_value);
    }
    else
    {
        SameNameTrial(classdsp->classscope, tree->lchild->rchild);
        // Judge if the class has parent class
        if (classdsp->parentname)
        {
            if (classdsp->parent)
            {
                scope.classscope = classdsp->parent->classscope;
                // Push parent class into scope stack
                PushScopeStack(stack, scope);
            }
            else
            {
                smerror += 1;
                printf("Semantic Error 4 at Line %d Column %d: The Parent Class \"%s\" is Not Declared!\n",
                    tree->line, tree->lchild->rchild->rchild->rchild->column, classdsp->parentname);
            }
        }
        scope.classscope = classdsp->classscope;
        PushScopeStack(stack, scope);
    }
}
```

图 3-10 类定义时的入栈操作

例如，再遇到右大括号时，表明类作用域或函数作用域或语句块作用域已经结束，需要从作用域栈中弹出相应的作用域符号表。如果当前作用域为类作用域，还需要判断其有没有父类，如果有父类，将父类作用域符号表也弹出；如果当前作用域为函数作用域，则还需要将形参作用域也弹出。相关代码如图 3-11 所示。

```
else if (!strcmp(tree->name, "SPRIGHTBRACE"))
{
    Scope top;
    PopScopeStack(stack, &top);
    // Judge if it has parent class, if it has, pop parent class as well
    if (top.type == ScopeClass && GetStackTop(stack)->type == ScopeClass)
        PopScopeStack(stack, &top);
    // Judge if it is function
    else if (top.type == ScopeLocal && GetStackTop(stack)->type == ScopeFormal)
        PopScopeStack(stack, &top);
    scope = *GetStackTop(stack);
}
```

图 3-11 右大括号对应的出栈操作

对于类型检查，最主要的便是检查赋值运算两边的表达式的类型是否匹配。为了实现检查表达式的类型，专门设计了一个独立的函数“GetExprType”来判断表达式的类型，该函数主要是递归，直到遇到能准确确定表达式的类型的语法单元为止。由于赋值运算出现的形式有很多种，所以其判断分支也相对分散，一个示例代码如图 3-12 所示。

```
else if (tree->rchild && !strcmp(tree->rchild->name, "OPASSIGN"))
{
    // If this is a assign statement, judge two sides of '=' whether are valid and compatible
    ExprType* exprtype = GetExprType(tree->rchild->rchild, stack);
    if (exprtype)
    {
        if (result.type == ScopeClass && strcmp(result.classscope->type, exprtype->type))
        {
            smerror += 1;
            printf("Semantic Error 7 at Line %d Column %d: The Two Sides of '=' have Conflicted Types! One is \"%s\", but the Other is \"%s\"!\n",
                tree->line, tree->rchild->column, result.classscope->type, exprtype->type);
        }
        else if (result.type == ScopeFormal && strcmp(result.formalscope->type, exprtype->type))
        {
            smerror += 1;
            printf("Semantic Error 7 at Line %d Column %d: The Two Sides of '=' have Conflicted Types! One is \"%s\", but the Other is \"%s\"!\n",
                tree->line, tree->rchild->column, result.formalscope->type, exprtype->type);
        }
        else if (result.type == ScopeLocal && strcmp(result.localscope->type, exprtype->type))
        {
            smerror += 1;
            printf("Semantic Error 7 at Line %d Column %d: The Two Sides of '=' have Conflicted Types! One is \"%s\", but the Other is \"%s\"!\n",
                tree->line, tree->rchild->column, result.localscope->type, exprtype->type);
        }
    }
}
```

图 3-12 赋值运算的类型检查

3.5 语义分析结果展示

结果展示分为两部分，第一部分为符号表信息展示，第二部分为静态语义错误检查展示。

首先，展示编译器生成的符号表。为了与 3.2 节中符号表的示例图相对应，在本测试中与其采用了相同的测试用例，该测试用例考虑了继承关系，例如子类“Mac”继承于父类“Computer”。测试用例的代码如图 3-13 所示。为了单独显示符号表，所以测试用例的代码是完全正确没有语法和语义错误的。

```

class Computer
{
    int cpu;
    void Crash(int numTimes)
    {
        int i;
        for (i = 0; i < numTimes; i = i + 1)
            Print("sad\n");
    }
}

class Mac extends Computer
{
    int mouse;
    void Crash(int numTimes)
    {
        Print("ack!");
    }
}

class Main
{
    static void main()
    {
        class Mac powerbook;
        powerbook = new Mac();
        powerbook.Crash(2);
    }
}

```

图 3-13 实验 2 测试用例代码

测试用例对应的全局作用域的符号表如图 3-14 所示。其中，Depictor 指向对应类的描述，类的描述和符号表将在下面给出。

```

[dracula@Lenovo-PC]~/Documents/IT_Study/compiler/lab2
$ ./parser test.decaf

Now print the symbol tables of "Semantic Analyzing":

```

Name	Category	Depictor
Main	Class	——> Main Class Depictor
Mac	Class	——> Mac Class Depictor
Computer	Class	——> Computer Class Depictor

图 3-14 测试用例全局作用域符号表

测试用例中类 Main、Mac、Computer 对应的类作用域的符号表分别如图 3-15、图 3-16、图 3-17 所示。其中，类“Depictor”的“ClassScope”指向对应类的类作用域符号表，类作用域符号表的“Depictor”指向对应的函数描述（下面给出）。

Main Class Depictor

ParentClass	ParentClassDepictor	ClassScope
Nil	Nil	——> Main Class Scope Symbol Table

Main Class Scope Symbol Table

Name	Category	Type	Depictor
main	Function	void	——> main Function Depictor

图 3-15 测试用例 main 类作用域符号表

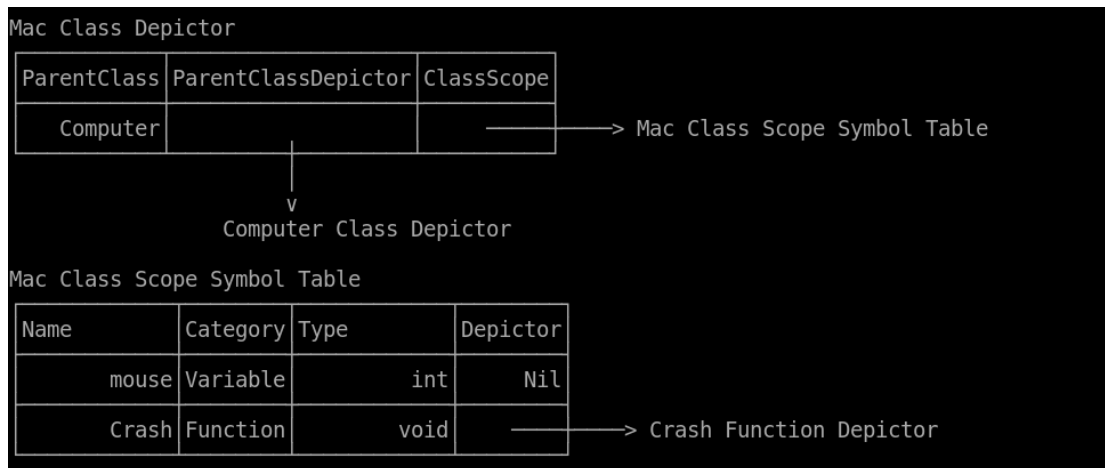


图 3-16 测试用例 main 类作用域符号表

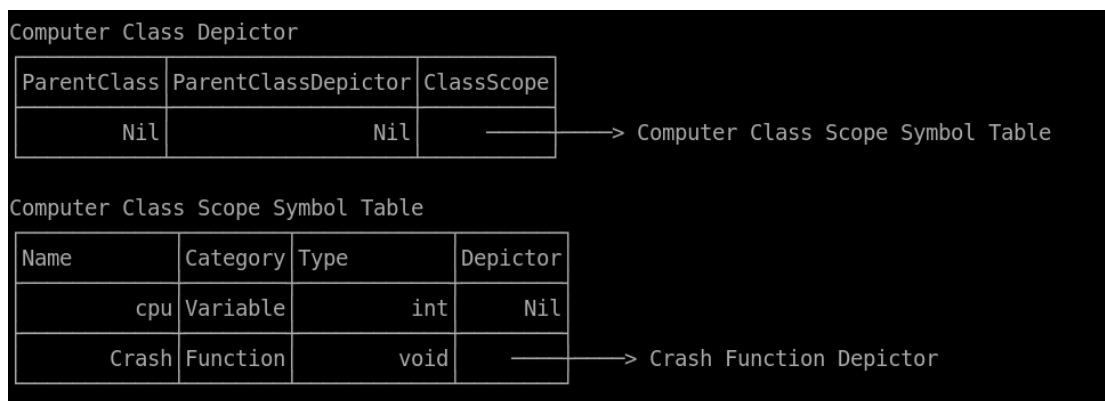


图 3-17 测试用例 computer 类作用域符号表

在测试用例中，类 Computer、Mac、Main 都有自己的成员函数，其中类 Mac 重载了类 Computer 的 Crash 函数，但相对简单，所以接下来展示只给出了 Computer 类中的 Crash 函数符号表（包括函数描述），包括一个形参作用域符号表（自动添加了类 Computer 的一个 this 形参）和一个内嵌局部作用域符号表，如图 3-18 所示。Main 类中 main 函数的符号表、形参符号表等如图 3-19 所示。

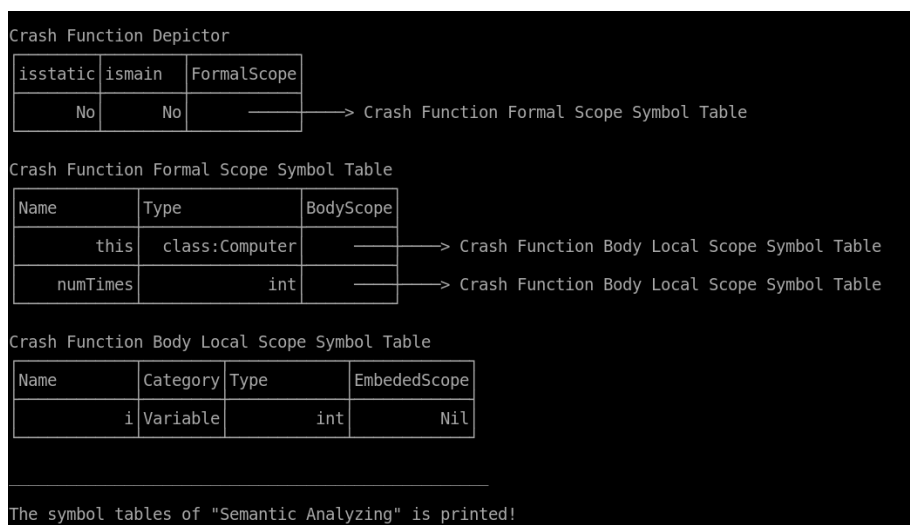


图 3-18 测试用例 Crash 函数包含的所有符号表

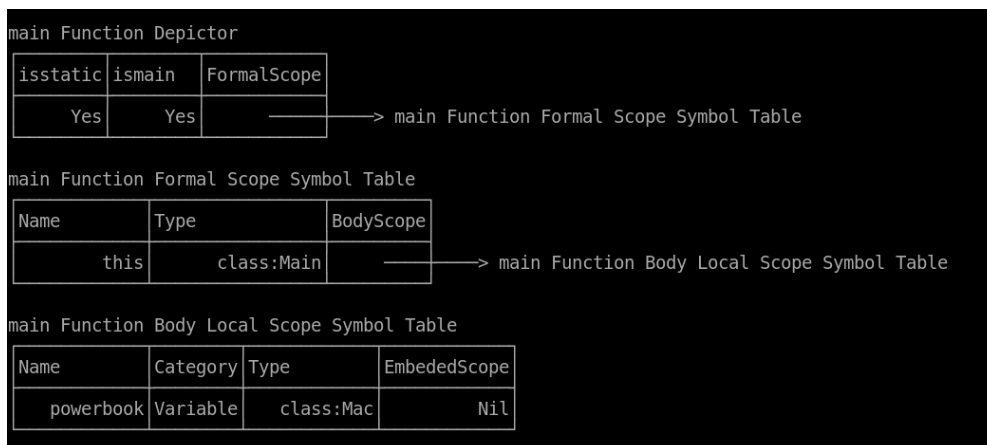


图 3-19 测试用例 main 函数包含的所有符号表

然后，展示编译器的静态语义错误检查能力。为了尽量多的打印出多种类型的错误，对测试用例进行了修改，使其包含了 1~10 和第 13 类错误。修改后的测试用例如图 3-20 所示。

```

1 class Computer {}
2
3 class Computer
4 {
5     int cpu;
6     void Crash(int numTimes)
7     {
8         int i;
9         class Pro laptop;
10        laptop = new Pro();
11        j = 1;
12        Computer = 1;
13        i = -0.1;
14        i.test();
15        test();
16        for (i = 0; i < numTimes; i = i + 1)
17            Print("sad\n");
18    }
19    void cpu()
20    {
21    }
22 }
23
24
25 class Mac extends Computers
26 {
27     int mouse;
28     void Crash(int numTimes)
29     {
30         Print("ack!");
31     }
32 }
33
34 class Main
35 {
36     static void main()
37     {
38         class Mac powerbook;
39         powerbook = new Mac();
40         powerbook.Crash(2);
41         powerbook.keybord = 1;
42         powerbook.PowerOn();
43     }
44 }

```

图 3-20 测试用例 error.decaf 代码

根据测试用例 `error.decaf` 的代码不难看出，其中的错误包括：定义了重名类 `Computer`，在类 `Computer` 的 `Crash` 函数中创建了一个没有定义过的类 `Pro` 的对象，在类 `Computer` 的 `Crash` 函数中未定义变量 `j` 便使用，在类 `Computer` 的 `Crash` 函数中将类 `Computer` 作为左值，在类 `Computer` 的 `Crash` 函数中将 `-0.1` 浮点数常量赋给整型变量 `i`，在类 `Computer` 的 `Crash` 函数中对非对象的整型变量 `i` 实施“.”运算，在类 `Computer` 的 `Crash` 函数中调用未定义过的函数 `test`，类 `Mac` 继承于一个没有定义过的父类 `Computers`（注意不是 `Computer`），在 `Main` 类的 `main` 函数中 `Mac` 类对象 `powerbook` 调用了类 `Mac` 没有定义的属性 `keybord` 和方法 `PowerOn`。语义分析的结果如图 3-21 所示。

```
[dracula@Lenovo-PC]~/Documents/IT_Study/compiler/lab2
$ ./parser error.decaf
Semantic Error 3 at Line 1 Column 7: The Class "Computer" has been Declared!
Semantic Error 2 at Line 3 Column 7: There are Members whose Name are all "cpu" in Class "Computer"!
Semantic Error 1 at Line 10 Column 22: The Class "Pro" is not Declared!
Semantic Error 5 at Line 11 Column 9: The Variable "j" is Not Declared Before Used!
Semantic Error 6 at Line 12 Column 9: The Class "Computer" can Not be Used as Left Value!
Semantic Error 7 at Line 13 Column 11: The Two Sides of "=" have Conflicted Types! One is "int", but the Other is "float"!
Semantic Error 9 at Line 14 Column 9: It is Not an Instance of Any Class!
Semantic Error 10 at Line 15 Column 9: The Function "test" is Not Declared Before Used!
Semantic Error 4 at Line 25 Column 19: The Parent Class "Computers" is Not Declared!
Semantic Error 8 at Line 41 Column 9: Class "Mac" has No Attribute Called "keybord"!
Semantic Error 13 at Line 42 Column 9: Class "Mac" has No Method Called "PowerOn"!
```

图 3-21 语义检查测试结果

从图 3-21 中可以看出，语义分析阶段对源程序中的所有语义错误都进行了准确地定位，告之程序员语义错误的具体位置，包括行号和列号。除此之外，编译器还能够给错相关的错误提示，包括错误类型和具体的错误原因。例如，在图 3-21 中有一行的内容为：

“Semantic Error 7 at Line 13 Column 11: The Two Sides of "=" have Conflicted Types! One is "int", but the Other is "float"!”

这段报错信息，不仅给出了错误的类型为语义错误第 7 种错误，还准确定位了错误所在位置为第 13 行第 11 列。除此之外，还给出了详细的错误原因和提示，即“=”两边表达式类型不匹配，一边是整型“int”而另一边是“float”。

4 中间代码生成

4.1 中间代码格式定义

中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式。一方面，中间代码将编译器自然地分为前段和后端两个部分；另一方面，在采用中间代码有利于进行机器无关的优化。

从中间代码所体现出的细节上，可以将中间代码分为高层次中间代码、中层次中间代码、低层次中间代码，而从表现形式上来看，中间代码又可以分为图形中间代码、线形中间代码和混合型中间代码。

在本编译器中，中间代码选择了线形中间代码的“四元式”，也叫三地址码。这种结构最大的优点是表示简单、处理高效，而缺点就是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得很复杂。

在做本实验时，由于时间关系，因此一开始就没有考虑要进行太复杂的代码优化。基于这样考虑，线性中间代码的缺点对本实验的影响并不大，再考虑到线性中间代码表现直观、实现简单的优点，因此，便选择了线形中间代码作为本编译器中间代码的形式。

对于线形中间代码，本编译器选择了“四元式”，具体的格式为：

(OP, DST, SRC1, SRC2)

其中，OP 为操作名称，例如加 Add、减 Sub、乘 Mul 等等；DST、SRC1、SRC2 分别为目的操作数、源操作数 1、源操作数 2 的地址，因此，这种表示方式也称为“三地址码 (TAC)”。

实现上述三地址码的方式也有很多种，可以采用不同的数据结构来实现，例如静态数组、单链表、双向链表等。本编译器采用了双向链表的形式实现三地址码，因为虽然双向链表增加了一点的实现复杂度，但却也换来了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且不存在代码最大行数的限制。

在具体实现过程中，并没有直接将 TAC 就定义成双向链表的形式，而是将其作为链表节点的一个属性（数据成员），之所以这样实现是为了方便后面的扩展（如果有可能）。除此之外，为了实现双向链表，每一个链表结点中还要有指向前置节点和后置节点的指针。关于 TAC 的定义，实现的形式如前面介绍的其具体格式相同。实现三地址码 (TAC) 和三地址码双向链表 (TACCode) 的具体代码如图 4-1 所示。


```

typedef struct TAC
{
    TACOpKind opkind;
    TACOperand dest;
    TACOperand firstsrc;
    TACOperand secondsrc;
} TAC;

/*
 *double orientation link list
 *Notice: the first element's "prev" points to the last element
 */
typedef struct TACCode
{
    int line;
    TAC code;
    struct TACCode* prev;
    struct TACCode* next;
} TACCode;

```

图 4-1 三地址码和其双向链表结构体

尽管中间代码在编译器的内部存储形式为三地址码（线形中间代码），但是打印出来的中间代码并不是完全按照“(OP, DST, SRC1, SRC2)”的格式，而是一种更为接近汇编语言或者是伪代码的形式，如表 4-1 所示。

表 4-1 中间代码的打印形式和对应的四元式

语法	描述	OP	Dst	Src1	Src2
LABEL x:	定义标号 x	LABEL	x		
FUNCTION f:	定义函数 f	FUNC	f		
x := y	赋值操作	ASSIGN	x	y	
x := y + z	加法操作	ADD	x	y	z
x := y - z	减法操作	SUB	x	y	z
x := y * z	乘法操作	MUL	x	y	z
x := y / z	除法操作	DIV	x	y	z
x := &y	取 y 得地址赋给 x	ASSIGN	x	&y	
x := *y	取以 y 值为地址的内存单元的内容赋给 x	ASSIGN	x	*y	
*x := y	取 y 值赋给以 x 值为地址的内存单元	ASSIGN	*x	y	
GOTO x	无条件跳转至标号 x	GOTO	x		
IF x [relop] y GOTO z	如果 x 与 y 满足[relop]则跳转至标号 z	IFREGOTO	z	x	y
RETURN x	退出当前函数并返回 x 值	RETUR	x		
DEC x [size]	内存空间申请，大小为 4 的倍数	DEC	x	size	
ARG x	传实参 x	ARG	x		
x := CALL	调用函数，并将返回值赋给 x	CALL	x		
PARAM x	函数参数声明	PARAM	x		
READ x	从控制台读取 x 的值	READ	x		
WRITE x	向控制台打印 x 的值	WRITE	x		

4.2 中间代码生成规则定义

表 4-2 列出了与表达式相关的一些结构的生成规则，假设有函数为 `translate_Exp()`，它接受三个参数：语法树的节点 `Exp`、符号表 `sym_table` 以及一个变量名 `place`，并返回一段语法树当前节点及其子孙节点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。这里需要注意的是，根据语法单元 `Exp` 所采用的产生式的不同，将会产生不同的中间代码。

表 4-2 基本表达式的生成规则

Expr 的产生式	生成中间代码的生成规则
INT	<code>value = get_value(INT)</code> <code>return [place := #value]</code>
ID	<code>variable = lookup(sym_table, ID)</code> <code>return [place := variable.name]</code>
Exp1 ASSIGNOP Exp2 (Exp1 \rightarrow ID)	<code>variable = lookup(sym_table, EXP1.ID)</code> <code>t1 = new_temp()</code> <code>code1 = translate_Exp(Exp2, sym_table, t1)</code> <code>code2 = [variable.name := t1] + [place := variable.name]</code> <code>return code1 + code2</code>
Expr1 PLUS Expr2	<code>t1 = new_temp()</code> <code>t2 = new_temp()</code> <code>code1 = translate_Exp(Exp2, sym_table, t1)</code> <code>code2 = translate_Exp(Exp2, sym_table, t1)</code> <code>code3 = [place := t1 + t2]</code> <code>return code1 + code2 + code3</code>
MINUS Expr1	<code>t1 = new_temp()</code> <code>code1 = translate_Exp(Exp2, sym_table, t1)</code> <code>code2 = [place := #0 - t1]</code> <code>return code1 + code2</code>
Expr1 RELOP Expr2	<code>label1 = new_label()</code> <code>label2 = new_label()</code> <code>code0 = [place := #0]</code> <code>code1 = translate_Cond(Exp, label1, label2, sym_table)</code> <code>code2 = [LABEL label1] + [place := #1]</code> <code>return code0 + code1 + code2 + [LABEL label2]</code>

说明：（1）用方括号扩起来的内容表示新建一条具体的中间代码。

（2）`Expr` 的下标只是用来区分产生式 `Exp \rightarrow Exp ASSIGNOP Exp` 中多次出现的 `Expr`。

（3）加号相当于连接运算，表示将两段代码连接成一段。

(4) 实际代码中的函数名或者生成式与上表可能不同，但核心思想是一样的。

Decaf 的语句包括表达式语句、符合语句、返回语句、跳转语句和循环语句，它们的生成规则如表 4-3 所示。

表 4-3 语句的生成规则

Stmt 的产生式	生成中间代码的生成规则
Expr	return translate_Expr(Expr, sym_table, NULL)
RETURN Expr	t1 = new_temp() code1 = translate_expr(Expr, sym_table, t1) code2 = [Return t1] return code1 + code2
IF LP Expr RP Stmt	label1 = new_label() label2 = new_label() code1 = translate_Cond(Expr, label1, label2, sys_table) code2 = translate_Stmt(Stmt, sym_table) return code1+[LABEL label1]+code2+[LABEL label2]
IF LP Expr RP Stmt1 Else Stmt2	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Expr, label1, label2, sys_table) code2 = translate_Stmt(Stmt1, sym_table) code3 = translate_Smt(Stmt2, sym_table) return code1+[LABEL label1]+code2+[GOTO label3]+[LABEL label2]+code3+[LABEL label3]
WHILE LP Expr RP Stmt	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Expr, label2, label3, sys_table) code2 = translate_Stmt(Stmt1, sym_table) return [LABEL label1]+code1+[LABEL label2]+code2+[GOTO label1]+[LABEL label3]

在上述表格中，出现了函数 `translate_Cond`，实际上，是在生成翻译表达式的同时生成这些跳转语句，`translate_Cond` 函数负责对条件表达式进行翻译，其生成规则如表 4-4 所示。需要注意的是，表 4-4 中并没有与回填相关的任何内容。原因很简单：将跳转的两个目标 `label_true` 和 `label_false` 作为继承属性（函数参数）进行处理，在这种情况下，每当在条件表达式内部需要跳转到页面时，跳转目标都已经从父节点那里通过参数得到了，直接填上即可。所谓回填，只用于将 `label_true` 和 `label_false` 作为综合属性处理的情况。因此，在本编译器中没有采用“拉链回填”的处理技术。

表 4-4 条件表达式的生成规则

Exp 的产生式	生成中间代码的生成规则
Exp1 RELOP Exp2	<pre> t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp1, sym_table, t1) code2 = translate_Exp(Exp2, sym_table, t2) op = get_relop(RELOP) code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false] </pre>
Not Exp1	<pre> return translate_Exp(Exp1, label_false, label_true, sym_table) </pre>
Exp1 AND Exp2	<pre> label1 = new_label() code1 = translate_Exp(Exp1, label1, label_false, sym_table) code2 = translate_Exp(Exp2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2 </pre>
Exp1 OR Exp2	<pre> label2 = new_label() code1 = translate_Exp(Exp1, label_true, label1, sym_table) code2 = translate_Exp(Exp2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2 </pre>
(other cases)	<pre> t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false] </pre>

本编译器对函数调用与函数定义相关的中间代码的生成作了一些特殊处理。当遇到函数定义的说明语句时，先创造一条“**FUNCTION** 函数名”的中间代码，然后通过寻找符号表找到该函数对应的符号表项，然后从符号表中获取该函数的形参域符号表，进而获取该函数的所有形参，然后为每一个形参创造一条“**PARAM** 形参名”的中间代码，最后，返回上述中间代码的合并即可；当遇到函数调用的语句时，处理方法类型，不过方向正好相反，也就是说，对于函数调用而言，先获得实参列表，然后为每一个实参创造一条“**ARG** 实参名”的中间代码，最后创造一条“**CALL** 函数名”的中间代码，最后返回中间这些代码的合并即可。

除此之外，在生成中间代码时还需考虑 Decaf 面向对象的特性。例如，对于类的函数成员，若考虑继承，函数名在父类和子类中应该有所不同；对于类的数据成员，应该是在定义一个对象并用“new”为其分配内存时，将该对象对应类的所有数据成员（加对象名）添加到对应作用域的符号表中。实际上，本编译器为了支持 Decaf 的面向对象的特性，作了很多细微的工作，这里限于篇幅不能详细给出，可以参见本编译器的源代码。

4.3 中间代码生成过程

中间代码的生成过程与语义分析阶段中静态语义检查的过程相类似，都是通过遍历语法分析树并同时维护作用域栈来实现的，区别只在于一个是通过语法单元判断是否存在语义错误，另一个则是通过语法单元判断如何生成中间代码。所以，生成中间代码的程序框架借用了实验 2 中静态语义错误检查的程序框架，采用自低向上的方式，递归实现语法分析树的遍历。

在遍历语法分析树的过程中，需要与实验 2 一样同时维护作用域栈，以保证当需要查找符号表时，可以准确定位符号的相关信息。具体的方法同实验 2 一样，所以这里不在累述。

在遍历语法树的过程中，有一步十分重要，那就是合并不同分支产生的中间代码。所为合并中间代码，实际上是合并两个分支的中间代码对应的双向链表。双向链表的合并算法在数据结构课程中已经学习过，比单项链表的合并复杂一些，相关的代码如图 4-2 所示。

```
TACCode* MergeTACCode(int num, ...)
{
    int i = 1;
    va_list codelist;
    TACCode* entrance;
    TACCode* tmp;
    va_start(codelist, num); // Init the variable parameter list
    while ((entrance = va_arg(codelist, TACCode*)) == NULL && i < num)
        i += 1;
    while (i < num)
    {
        // First, previous linklist's last element's "next" points to new linklist's first element
        if ((entrance->prev->next = va_arg(codelist, TACCode*)) == NULL)
        {
            /*printf("3 test i:%d num:%d\n", i, num);*/
            i += 1;
        }
        else
        {
            // Then, save the pointer to new linklist's last element
            tmp = entrance->prev->next->prev;
            // Next, new linklist's first element's "prev" points to previous linklist's last element
            entrance->prev->next->prev = entrance->prev;
            // Finally, previous linklist's first element's "prev" points to new linklist's last element
            entrance->prev = tmp;
            i += 1;
        }
    }
    return entrance;
}
```

图 4-2 合并中间代码双向链表

如图 4-2 所示，由于在合并各路分支的中间代码时，并不能确定分支的个数，因此需要通过一个可变参数列表实现参数个数不确定的函数。合并双向链表时需要借助一个 tmp 来储存因为改变链表结点导致的有效节点的失踪(无指针指向)，同时还需要注意的一点是双向链表的“prev”指针也是有含义的，需要连接，这是其与单向链表最大的不同。

4.4 代码优化

由于时间有限，本编译器在代码优化方面所做的工作较少，只对标号和常量进行了处理。

在标号方面，当生成完中间代码后，编译器会在遍历中间代码的过程中，删除多余标号。所谓多余标号，是指在中间代码中紧挨着的两条中间代码对应的都是“LABEL labelx”语句，显然，这两条中间代码有一条是多余的，可以直接删除而不影响程序的正确性。多余标号的产生是可能的，因为在翻译表达式和判断语句的生成规则中并没有考虑多余标号，仅保证了生成规则生成的中间代码的逻辑正确性。

在常量方面，对应常量而言，有些情况下是没有必要专门为一个常量产生中间代码的。

4.5 中间代码生成结果展示

首先给出的是本次实验的第一个测试用例的 Decaf 代码，如图 4-3 所示。

```
1 class fruit
2 {
3     int kind;
4     int taste;
5     int value(int m, int n)
6     {
7         int result;
8         kind = 1;
9         if (m > 0)
10        {
11            if (m == 1)
12                result = n * taste + kind;
13            else if (m == 2)
14                result = n * taste;
15        }
16        else
17            result = n * taste - kind;
18        return result;
19    }
20 }
21
22 class Main
23 {
24     static void main()
25     {
26         int x;
27         int y;
28         class fruit apple;
29         apple = new fruit();
30         apple.kind = 2;
31         apple.taste = 6;
32         apple.value(x, y);
33     }
34 }
```

图 4-3 实验 3 第一个测试用例代码

在该测试用例中，共定义了两个类：fruit 和 Main。其中 fruit 包含一个成员函数 value，value 函数中包含一个嵌套的 if-else 语句，而其中每个基本语句又都是一个复杂算数表达式。所以，本测试用例，主要测试编译器将分支语句和表达式转换成中间代码的正确性。

将实验 3 对应的代码编译执行后，对于该测试用例生成的中间代码如图 4-4 和 4-5 所示。

```
[dracula@localhost]--[~/Documents/IT_Study/compiler/lab3]
$ ./parser test.decaf

Now print the intermediate code of "InterCodeGenerate":
```

No	Source Position	TACCode
0	(at line 0)	CALL main
1	(at line 5)	FUNCTION value :
2	(at line 5)	PARAM m
3	(at line 5)	PARAM n
4	(at line 8)	kind := #1
5	(at line 9)	_t2 := m
6	(at line 9)	_t3 := #0
7	(at line 9)	IF _t2 > _t3 GOTO label0
8	(at line 9)	GOTO label1
9	(at line 9)	LABEL label0 :
10	(at line 11)	_t4 := m
11	(at line 11)	_t5 := #1
12	(at line 11)	IF _t4 = _t5 GOTO label2
13	(at line 11)	GOTO label3
14	(at line 11)	LABEL label2 :
15	(at line 12)	_t8 := n
16	(at line 12)	_t9 := taste
17	(at line 12)	_t6 := _t8 * _t9
18	(at line 12)	_t7 := kind
19	(at line 12)	result := _t6 + _t7
20	(at line 11)	GOTO label4
21	(at line 11)	LABEL label3 :
22	(at line 13)	_t10 := m
23	(at line 13)	_t11 := #2
24	(at line 13)	IF _t10 = _t11 GOTO label5
25	(at line 13)	GOTO label6
26	(at line 13)	LABEL label5 :
27	(at line 14)	_t12 := n
28	(at line 14)	_t13 := taste
29	(at line 14)	result := _t12 * _t13
30	(at line 13)	LABEL label6 :
31	(at line 11)	LABEL label4 :
32	(at line 9)	GOTO label7
33	(at line 9)	LABEL label1 :
34	(at line 17)	_t16 := n
35	(at line 17)	_t17 := taste

图 4-4 实验 3 第一个测试用例中间代码 1

```
36 (at line 17)      _t14 := _t16 * _t17
37 (at line 17)      _t15 := kind
38 (at line 17)      result := _t14 - _t15
39 (at line 9)       LABEL label7 :
40 (at line 18)      _t18 := result
41 (at line 18)      RETURN _t18

42 (at line 24)      FUNCTION main :
43 (at line 30)      apple.kind := #2
44 (at line 31)      apple.taste := #6
45 (at line 32)      _t1 := x
46 (at line 32)      _t0 := y
47 (at line 32)      ARG _t1
48 (at line 32)      ARG _t0
49 (at line 32)      CALL value

The intermediate code of "InterCodeGenerate" is printed!

[dracula@localhost]--[~/Documents/IT_Study/compiler/lab3]
$
```

图 4-5 实验 3 第一个测试用例中间代码 2

可以发现，在中间代码中，由于未对中间代码进行优化，导致生成的中间代码中存在大量的临时变量“_tx”（为了区分临时变量和用户定义的标识符变量，特地将临时变量写为“_tx”的形式，因为“_tx”是一个不合法的标识符，不可能与用户定义的变量同名）。这些临时变量是被添加到相应的作用域的符号表中的，以方便生成目标代码时使用。为了进一步显示这一点，可以打印出源程序的符号表进行观察。这里，只给出 main 函数的函数体局部作用域符号表如图 4-6 所示，和 value 函数的函数体局部作用域符号表和内含语句块的局部作用域符号表如图 4-7 所示。

main Function Body Local Scope Symbol Table			
Name	Category	Type	EmbededScope
_t1	Variable	int	Nil
_t0	Variable	int	Nil
apple.taste	Variable	int	Nil
apple.kind	Variable	int	Nil
apple	Variable	class:fruit	Nil
y	Variable	int	Nil
x	Variable	int	Nil

图 4-6 main 函数体局部作用域符号表

Name	Category	Type	EmbeddedScope
_t18	Variable	int	Nil
_t17	Variable	int	Nil
_t16	Variable	int	Nil
_t15	Variable	int	Nil
_t14	Variable	int	Nil
_t3	Variable	int	Nil
_t2	Variable	int	Nil
0	Block	Block	—————> Block 0 Local Scope Symbol Table
result	Variable	int	Nil

Name	Category	Type	EmbeddedScope
_t13	Variable	int	Nil
_t12	Variable	int	Nil
_t11	Variable	int	Nil
_t10	Variable	int	Nil
_t9	Variable	int	Nil
_t8	Variable	int	Nil
_t7	Variable	int	Nil
_t6	Variable	int	Nil
_t5	Variable	int	Nil
_t4	Variable	int	Nil

图 4-7 value 函数体及内嵌局部作用域符号表

通过比对图 4-6、4-7 和对应的源程序以及中间代码，可以知道临时变量所加的作用域是正确的。

综上，对于测试用例 1，因为无法对其进行实际运行测试，只能通过肉眼观察，暂时认定中间代码生成阶段已经可以正确运行。为了印证这一观点，再进行一个测试用例。在该测试用例中，同样定义了两个类：**Sum** 和 **Main**。其中 **Sum** 类包含函数成员 **getSum**，该函数的主体便是一个 **for** 循环，所以，该测试用例主要测试的是编译器是否可以成功将 **for** 函数转化成对应的中间代码。该测试用例的 Decaf 代码如图 4-8 所示。

```

1 class Sum
2 {
3     int getSum(int n)
4     {
5         int i;
6         int sum;
7         sum = 0;
8         for (i = 1; i <= n; i = i + 1)
9             sum = sum + i;
10        return sum;
11    }
12 }
13
14 class Main
15 {
16     int main()
17     {
18         int n;
19         int result;
20         class Sum sum;
21         sum = new Sum();
22         n = 10;
23         result = sum.getSum(n);
24         return 0;
25     }
26 }

```

图 4-8 实验 3 第二个测试用例代码

对于测试用例 2，生成的中间代码如图 4-9 所示。

Now print the intermediate code of "InterCodeGenerate":

No	Source Position	TACCode
0	(at line 0)	CALL main
1	(at line 3)	FUNCTION getSum :
2	(at line 3)	PARAM n
3	(at line 7)	sum := #0
4	(at line 8)	i := #1
5	(at line 8)	LABEL label0 :
6	(at line 8)	_t2 := i
7	(at line 8)	_t3 := n
8	(at line 8)	IF _t2 <= _t3 GOTO label1
9	(at line 8)	GOTO label2
10	(at line 8)	LABEL label1 :
11	(at line 9)	_t4 := sum
12	(at line 9)	_t5 := i
13	(at line 9)	sum := _t4 + _t5
14	(at line 8)	_t6 := i
15	(at line 8)	_t7 := #1
16	(at line 8)	i := _t6 + _t7
17	(at line 8)	GOTO label0
18	(at line 8)	LABEL label2 :
19	(at line 10)	_t8 := sum
20	(at line 10)	RETURN _t8
21	(at line 16)	FUNCTION main :
22	(at line 22)	n := #10
23	(at line 23)	_t0 := n
24	(at line 23)	ARG _t0
25	(at line 23)	result := CALL getSum
26	(at line 24)	_t1 := #0
27	(at line 24)	RETURN _t1

The intermediate code of "InterCodeGenerate" is printed!

图 4-8 实验 3 第二个测试用例中间代码

手工验证编译器生成的中间代码，发现该中间代码对应的逻辑执行顺序与源程序测试用例代码执行顺序相同，即编译器可以成功地将 for 函数转化为相应的中间代码。

同样的，打印出源程序对应的符号表，这里只给出 main 函数体的局部作用域符号表如图 4-9 所示和 getSum 函数体的局部作用域符号表如图 4-10 所示。

main Function Body Local Scope Symbol Table

Name	Category	Type	EmbeddedScope
_t1	Variable	int	Nil
_t0	Variable	int	Nil
sum	Variable	class:Sum	Nil
result	Variable	int	Nil
n	Variable	int	Nil

图 4-9 main 函数体局部作用域符号表

getSum Function Body Local Scope Symbol Table

Name	Category	Type	EmbeddedScope
_t8	Variable	int	Nil
_t7	Variable	int	Nil
_t6	Variable	int	Nil
_t5	Variable	int	Nil
_t4	Variable	int	Nil
_t3	Variable	int	Nil
_t2	Variable	int	Nil
sum	Variable	int	Nil
i	Variable	int	Nil

图 4-10 getSum 函数体局部作用域符号表

通过观察源程序对应的符号表，可以验证临时变量均被正确地加入到对应的

作用域的符号表中。

实际上，对于实验 3 也就是编译器的第三阶段——中间代码生成阶段，所生成的中间代码的正确性是不容易直观测试的，只能通过手动运算然后肉眼比对来判断正误，显然，这样的验证方法是不严谨的，带有一定主观性的。

这一段的正确性只能在后一阶段中进行验证，事实上，当做到第 4 次实验时，发现了不少该阶段的错误然后又回来改正的。例如，由于生成中间代码这一阶段一直没用到指向前一项的指针“prev”，所以总认为其应该是没有问题的，但到了第 4 阶段向前搜索中间代码时程序总是出错，仔细分析，步步回溯搜索，才发现错误是在合并中间代码时对前置指针“prev”的操作有点问题。

这一阶段最大的遗憾便是没有时间作代码优化相关的工作，如果有机会，一定要做一下相关的工作。

5 目标代码生成

5.1 指令集选择

本编译器的最终目标是使得编译出来的汇编代码经过汇编器汇编后可以直接运行在组成原理实验完成的 CPU 上。而组成原理实验所完成的 CPU 仅支持 MIPS 指令集的一个子集，为了实现最终目标，本编译器选择的也为 MIPS 指令集，且只支持其一个子集，如表 5-1 所示。

表 5-1 编译器与组原 CPU 支持的指令集

#	指令	格式	备注
1	Add	add \$rd, \$rs, \$rt	指令功能及 指令格式 参考 MIPS32 指令集
2	Add Immediate	addi \$rt, \$rs, immediate	
3	Add Immediate Unsigned	addiu \$rt, \$rs, immediate	
4	Add Unsigned	addu \$rd, \$rs, \$rt	
5	And	and \$rd, \$rs, \$rt	
6	And Immediate	andi \$rt, \$rs, immediate	
7	Shift Left Logical	sll \$rd, \$rt, shamt	
8	Shift Right Arithmetic	sra \$rd, \$rt, shamt	
9	Shift Right Logical	srl \$rd, \$rt, shamt	
10	Sub	sub \$rd, \$rs, \$rt	
11	Or	or \$rd, \$rs, \$rt	
12	Or Immediate	ori \$rt, \$rs, immediate	
13	Nor	nor \$rd, \$rs, \$rt	
14	Load Word	lw \$rt, offset(\$rs)	
15	Store Word	sw \$rt, offset(\$rs)	
16	Branch on Euqal	beq \$rs, \$rt, label	
17	Branch Not on Euqal	bne \$rs, \$rt, label	
18	Set Less Than	slt \$rd, \$rs, \$rt	
19	Set Less Than Immediate	slti \$rt, \$rs, immediate	
20	Set Less Than Unsigned	sltu \$rd, \$rs, \$rt	
21	Jump	j label	
22	Jump and Link	jal label	
23	Jump Register	jr \$rs	
24	syscall	syscall	

这里需要说明下的是，syscall 指令只支持 4 类：print_int、read_int、print_char 和 exit，分别是打印整数，读入整数，打印字符和退出。

5.2 寄存器分配算法

对于寄存器的分配算法，本编译器并没有完全按照书上的几种寄存器分配算法设计，例如朴素寄存器分配算法、局部寄存器分配算法、图染色算法等，主要的原因有：朴素寄存器分配算法过于简单，寄存器的使用率极低，只有少数几个寄存器一直在使用；局部寄存器分配算法与图染色寄存器分配算法都需要将中间代码划分为基本块，然后基于活跃变量分析分配寄存器，显然，为了实现这两个算法，还要设计 DAG（有向图）的数据结构来组织基本块，然后设计活跃变量分析的相关算法和函数，图染色还要学习并实现图染色算法，代价都过高。

基于上述考虑，本编译器设计了自己的寄存器分配算法，从原理上讲，实际上是一种简化过的局部寄存器分配算法，但又不划分数据块；也是一种改进版的朴素寄存器分配算法，考虑了 LRU 算法，极大的提高了各个寄存器的使用率。

首先明确本编译器可分配的寄存器有哪些。对于编译器而言，有些寄存器是有特殊用途的，比如“\$sp”存放栈顶指针、“\$ra”存放函数返回值等，所以这些寄存器是不能拿来随便分配的。在本编译器的寄存器分配算法中，可分配的寄存器一共有 18 个，包括\$t0~\$t7、\$s0~\$s7 以及\$t8~\$t9。

然后考虑如何设计寄存器分配算法。对于编译器而言，一共就有 18 个寄存器可以分配，但是源程序的变量（包括临时变量）数量往往大于 18，所以必须设计一个较为高效的算法来保证个寄存器的使用率。显然，对于已经分配寄存器的变量，直接它占用的寄存器返回给它就可以；对于没有分配寄存器的变量，则应首选一个空寄存器，如果没有，再选择一个最久未使用的寄存器；最坏情况下，随机分配一个寄存器即可。

本编译器的分配算法依次考虑如下策略选择寄存器：

- （1）判断申请寄存器的变量是否已经占用了某个寄存器，如果占用，将其占用的寄存器返回；
- （2）从可分配寄存器中，选择一个编号最低的未被占用的寄存器分配给申请寄存器的变量；
- （3）从可分配的寄存器中，利用最久未使用算法（LRU）和寄存器标记值“used”选择一个编号最低的最久未使用的寄存器；
- （4）选择\$t0 寄存器并返回。

当策略（1）不满足时考虑策略（2），当策略（2）不满足时考虑策略（3），当策略（3）不满足时考虑策略（4）。但需要注意的是，对于从策略（3）和策略（4）中选择的寄存器，由于原寄存器已经被某个变量占用了，所以必须先将寄存器中原有的变量写会寄存，即执行依次“sw”操作才能进行“lw”操作。上述寄存器分配算法的流程图如图 5-1 所示。

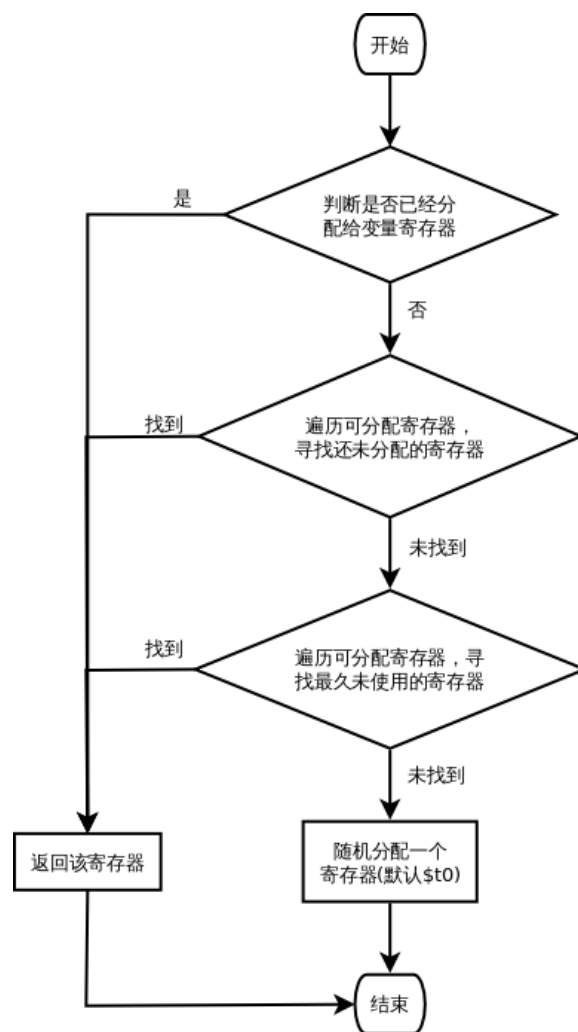


图 5-1 寄存器分配算法

基于上述算法，用 C 语言可以实现相关的寄存器分配函数。对于可分配寄存器 \$t0~\$t7、\$s0~\$s7 以及 \$t8~\$t9，它们的寄存器编号正好连续，从 8 到 25，所以遍历可寄存器时，只需循环访问寄存器组的 8 到 25 号寄存器即可。在实现寄存器分配算法之前，先给出寄存器的相关结构体定义，如图 5-2 所示。

```

typedef enum Register
{
    Reg_zero,
    Reg_at,
    Reg_v0, Reg_v1,
    Reg_a0, Reg_a1, Reg_a2, Reg_a3,
    Reg_t0, Reg_t1, Reg_t2, Reg_t3, Reg_t4, Reg_t5, Reg_t6, Reg_t7,
    Reg_s0, Reg_s1, Reg_s2, Reg_s3, Reg_s4, Reg_s5, Reg_s6, Reg_s7,
    Reg_t8, Reg_t9,
    Reg_k0, Reg_k1,
    Reg_gp, Reg_sp, Reg_fp, Reg_ra
} Register;

typedef struct RegDepictorEntry
{
    int filled;           // Imply if the register is allocated
    int used;             // Tag for LRU algorithm
    Scope variable;       // The variable in register if there is
} RegDepictorEntry;
  
```

图 5-2 寄存器相关结构体定义

基于给定寄存器分配算法实现的寄存器分配函数如图 5-3 所示。

```
int GetReg(FILE* fp, Scope variable)
{
    int i;
    // If the variable is formal
    if (variable.type == ScopeFormal)
    {
        if (variable.formalscope->reg != 0)
        {
            // The variable has been allocated register
            RegDepictor[variable.formalscope->reg].used += 1;
            return variable.formalscope->reg;
        }
        else
        {
            for (i = 8; i <= 25; i++)
            {
                // Judge if there is still empty register left
                if (RegDepictor[i].filled == 0)
                {
                    RegDepictor[i].filled = 1;
                    RegDepictor[i].used = 0;
                    RegDepictor[i].variable = variable;
                    variable.formalscope->reg = i;
                    fprintf(fp, "lw %s, %d($fp)\n", RegName[i], variable.formalscope->offset);
                    return i;
                }
            }
            for (i = 8; i <= 25; i++)
            {
                if (RegDepictor[i].used == 0)
                {
                    // Find the least used register
                    fprintf(fp, "sw %s, %d($fp)\n", RegName[i], RegDepictor[i].variable.formalscope->offset);
                    RegDepictor[i].variable = variable;
                    variable.formalscope->reg = i;
                    fprintf(fp, "lw %s, %d($fp)\n", RegName[i], variable.formalscope->offset);
                    RegDepictor[i].used = 0;
                    return i;
                }
            }
            // If tactics above all fails, select register t0
            fprintf(fp, "sw %s, %d($fp)\n", RegName[8], RegDepictor[8].variable.formalscope->offset);
            fprintf(fp, "lw %s, %d($fp)\n", RegName[8], variable.formalscope->offset);
            RegDepictor[8].used = 0;
            return 8;
        }
    }
}
```

图 5-3 寄存器分配函数（部分）

5.3 目标代码生成算法

目标代码的生成，实际上就是将中间代码转化为汇编指令的过程。所以生成目标代码，可以扫描一遍中间代码，然后逐条将中间代码翻译成对应的汇编指令，写入目标文件即可。

由于在中间代码生成阶段，本编译器选择的中间代码形式为三地址码（“四元式”），这种形式的中间代码和汇编代码有较好的对应性，大部分情况下每个三地址码用 1 或 2 条汇编指令便可完成。所以，对于大部分中间代码而言，可以很容易地将其翻译成最终的汇编指令，而不需要做复杂的变换。

正式基于上述考虑，下面将以表格的形式给出部分可以直接转化成 MIPS 汇编指令的中间代码。需要注意的是，列出的中间代码远远不是全部的中间代码。简单中间代码与 MIPS 汇编指令的对应表如表 5-2 所示。

表 5-2 简单中间代码与 MIPS 汇编指令的对应关系

中间代码	MIPS32 指令
LABEL x:	x:
x := #k	ori reg(x), \$0, k
x := y	addu reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k, \$0
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	addi reg(x), reg(y), -k
x := y - z	sub reg(x), reg(y), reg(z)
x := y * z	mul reg(x), reg(y), reg(z)
x := y / z	div reg(y), reg(z) mfo reg(x)
GOTO x	j x
IF x == y GOTO z	beq reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z

需要注意的是，虽然在表中给出的中间代码“ $x := y * z$ ”对应的 MIPS 指令为“mul reg(x), reg(y), reg(z)”，但这只是对于汇编模拟器而言的。如果用户指定了生成的目标代码的运行平台为组原 CPU，则需对乘法作进一步处理，因为组原的 CPU 中是没有“mul”指令的，而只有表 5-1 所列出的指令。所以，若运行平台为组原 CPU，则需要将乘法以一个汇编子程序的形式实现，实现乘法的算法为 booth 乘法，其具体的汇编程序如图 5-4 所示。

```
fprintf(fp, "\nbooth_mul:\n");
fprintf(fp, "srl $a0, $a0, 16\n");
fprintf(fp, "addi $a2, $0, 16\n");
fprintf(fp, "add $v0, $0, $a0\n");
fprintf(fp, "addi $a3, $0, 0\n");
fprintf(fp, "booth_loop:\n");
fprintf(fp, "andi $t2, $a0, 1\n");
fprintf(fp, "beq $a3, $t2, booth_l1\n");
fprintf(fp, "slt $t3, $t2, $a3\n");
fprintf(fp, "beq $t3, $0, booth_l2\n");
fprintf(fp, "add $v0, $v0, $a1\n");
fprintf(fp, "j booth_l1\n");
fprintf(fp, "booth_l2:\n");
fprintf(fp, "sub $v0, $v0, $a1\n");
fprintf(fp, "booth_l1:\n");
fprintf(fp, "sra $v0, $v0, 1\n");
fprintf(fp, "add $a3, $t2, $0\n");
fprintf(fp, "sra $a0, $a0, 1\n");
fprintf(fp, "addi $a2, $a2, -1\n");
fprintf(fp, "bne $a2, $0, booth_loop\n");
fprintf(fp, "add $s0, $v0, $0\n");
fprintf(fp, "jr $ra\n");
```

图 5-4 booth 乘法的汇编子程序

当检测到中间代码“ $x := y * z$ ”时，可以通过子程序调用的方式，代用 booth 乘法子程序，以实现在组原 CPU 上运行乘法运算。

同样的道理，组原的 CPU 也不支持比较跳转指令 bgt、blt、bge 和 ble，所以对于形如“IF x >/<(=) y GOTO z”的中间代码，需要多条 MIPS 汇编指令翻译。对应的翻译方法和程序代码如图 5-5 所示。

```

else if (tmp->code.opkind == IFLTGOTO)
{
    sprintf(cache, "label%d", tmp->code.dest.labelvalue);
    fprintf(fp, "slt $1, %s, %s\n", RegName[GetReg(fp, tmp->code.firstsrc.variable)], RegName[GetReg(fp, tmp->code.secondsrc.variable)]);
    fprintf(fp, "addi, $1, $1, -1\n");
    fprintf(fp, "beq $1, $0, %s\n", cache);
}
else if (tmp->code.opkind == IFLEGOTO)
{
    sprintf(cache, "label%d", tmp->code.dest.labelvalue);
    fprintf(fp, "beq %s, %s, %s\n", RegName[GetReg(fp, tmp->code.firstsrc.variable)], RegName[GetReg(fp, tmp->code.secondsrc.variable)], cache);
    fprintf(fp, "slt $1, %s, %s\n", RegName[GetReg(fp, tmp->code.firstsrc.variable)], RegName[GetReg(fp, tmp->code.secondsrc.variable)]);
    fprintf(fp, "addi, $1, $1, -1\n");
    fprintf(fp, "beq $1, $0, %s\n", cache);
}
else if (tmp->code.opkind == IFGTGOTO)
{
    sprintf(cache, "label%d", tmp->code.dest.labelvalue);
    fprintf(fp, "slt $1, %s, %s\n", RegName[GetReg(fp, tmp->code.secondsrc.variable)], RegName[GetReg(fp, tmp->code.firstsrc.variable)]);
    fprintf(fp, "addi, $1, $1, -1\n");
    fprintf(fp, "beq $1, $0, %s\n", cache);
}
else if (tmp->code.opkind == IFGEGOTO)
{
    sprintf(cache, "label%d", tmp->code.dest.labelvalue);
    fprintf(fp, "beq %s, %s, %s\n", RegName[GetReg(fp, tmp->code.secondsrc.variable)], RegName[GetReg(fp, tmp->code.firstsrc.variable)], cache);
    fprintf(fp, "slt $1, %s, %s\n", RegName[GetReg(fp, tmp->code.secondsrc.variable)], RegName[GetReg(fp, tmp->code.firstsrc.variable)]);
    fprintf(fp, "addi, $1, $1, -1\n");
    fprintf(fp, "beq $1, $0, %s\n", cache);
}

```

图 5-5 比较跳转类中间代码的翻译方法

考虑完所有上述中间代码，已经可以生成大部分目标代码，但对于最终目标来说，仍有一大步没有完成。那就是函数定义和函数调用相关中间代码的翻译。对于函数调用而言，最重要就是栈帧的维护和调用过程的设计。接下来将详细介绍本编译器这两方面的实现方法。

对于栈帧，本编译器在栈帧中存放的内容有：局部变量、上一函数的返回地址、上一函数的栈帧指针\$fp 以及多余的 4 个形参。

对于调用过程，本编译器没有采用传统的调用者和被调用者各自保存寄存器的方法，而是统一由调用者保存寄存器。注意，调用者保存的寄存器并没有保存到栈中，而是直接将寄存器中的变量写回内存。除此之外，剩余的过程与标准的 MIPS 子程序调用没有太多区别，这里就不再累述。

除此之外，由于 Decaf 默认从 Main 类的 main 函数开始执行，所以在目标代码的头部应加上相应的跳转信息以及参数初始化（若要运行在组原 CPU 上）。

下面将总的介绍下为了能够让组成原理实验实现的 CPU 上运行编译器编译出的汇编代码所做的一些工作。

首先，简单说明编译器所做的特殊处理。第一，是关于一些条件跳转指令的处理，由于此次组原实验支持的条件跳转指令只有“beq”和“bne”，即相等跳转和不相等，故需要编译器时对其余条件跳转指令作一些转化，例如对于小于等于跳转指令“ble \$rs, \$rt, label”，利用四条等价 MIPS 指令（已经实现了的）和 1 号编译器使用寄存器去翻译，使用“beq \$rs, \$rt, label”、“slt \$1, \$rs, \$rt”、“addi \$1, \$1, -1”、“beq \$1, \$0, label” 四条指令便可等效地实现小于等于跳转的功能；第

二，是关于乘除法的处理，由于此次组原实验并不支持乘法指令和除法指令，因此编译器需要在生成汇编代码时，将乘法利用 booth-乘法算法（该算法只用到移位、加法等已有指令）实现，至于除法，没有找到合适的算法实现；第三，由于 CPU 中用到的数据存储器 RAM 的大小仅有 1024 个字节，地址是从 0~1023，因此为了避免可能的错误，编译器生成的汇编代码在开始处便将“\$sp”置为 1023（数据存储器尾）；第四，编译器在编译打印字符串时，针对组原 CPU 用到的“哑终端”，生成汇编代码时作了一定处理。

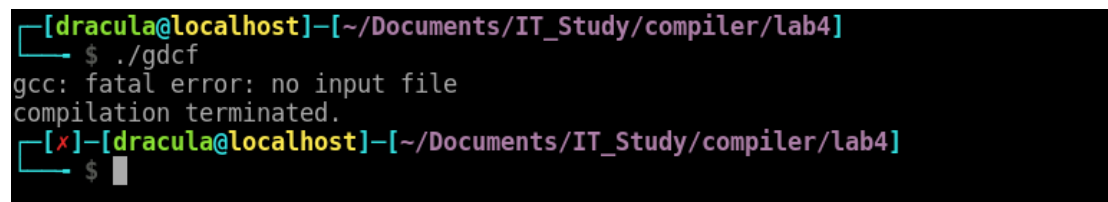
接下来考虑组原方面的 CPU 所做的改进。第一，增加了“输入一个整数的” syscall 功能，该 syscall 对应的 \$v0 值为 5，且会将读入的整数存到 \$v0 寄存器中，为此需要做的修改有：添加一个输入端以及一个输入确认按钮（当调用该 syscall 时，CPU 会处在停机状态等待用户输入，用户输入完毕后按下输入确认按钮程序才会继续执行），为 RegFile 的 RW 端、Write_data 端（多路选择器）添加新的数据来源，并在控制器中进行适当修改；第二，增加了“在哑终端输出一个字符”的 syscall 功能，该 syscall 对应的 \$v0 值为 11，会将 \$a0 中 ascii 码对应的字符打印到哑终端中，为此需要添加一个哑终端，并对控制器作适当修改和改进；第三，对原有的打印 \$a0 到显像管的 syscall 作了修改，原 CPU 下只要 \$v0 != 10 即是调用该功能，现在规定其 syscall 对应的 \$v0 值为 1，同样需要对控制器作适当修改和改进。

5.4 目标代码生成结果展示（综合四次试验）

在第四次实验时也就是本次实验后，对四次实验进行了汇总和封装。封装的具体方式是通过命令行的参数指定显示四次实验的结果，且命令行参数的位置和顺序无所谓，但格式有所要求，接下来将详细介绍。

首先，在实验目录下输入“make”后编译源代码并得到一个名为“gdcf”（GNU-Decaf）的可执行程序，这就是最终的编译器。

若不加任何参数执行“./gdcf”会有报错信息如图 5-6 所示。



```
[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ./gdcf
gcc: fatal error: no input file
compilation terminated.
[x]-[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$
```

图 5-6 不加参数执行 gdcf 报错信息

报错信息提示产生了致命错误：没有输入原程序文件，所以编译器不知道要编译的源程序是哪一个，因此编译只能终止。（实际上，该报错信息与 gcc 不加参数执行的报错信息相同）。

对于大部分的 Linux 程序而言，参数“-h”都是“打印程序使用的帮助信息”。对于编译器 gdcf 而言，也是如此，加“-h”参数后执行的输出如图 5-7 所示。

```
[*]-[dracula@localhost]--[~/Documents/IT_Study/compiler/lab4]
$ ./gdcf -h
Usage: gdcf [options] file
Compile Decaf source file to asm format. Notice: gdcf only support a subset of Decaf

Options:
-d, --debug      Add debug information(source file line) to asm file.
-g, --gmtree     Display the grammar tree of the target.
-h, --help      Display this information.
-i, --intercode  Display the intermediate code of the target.
-s, --symtable  Display the symbol table of the target.
-m, --mars      Generate the code which can run on Mars MIPS Simulator. Otherwise the code can only run on special cpu.
-o, <file>      Place the output into <file>.
[dracula@localhost]--[~/Documents/IT_Study/compiler/lab4]
$
```

图 5-7 gdcf 加参数-h 输出的帮助信息

通过图 5-7 不难看出，本编译器的帮助信息十分规范，不仅给出了程序的运行格式为“gdcf [options] file”，也给出了 options 的具体类型。并且，每个参数都可以通过缩写和全程的方式调用，例如“gdcf -h”和“gdcf -help”具有同样的功能。各参数具体的含义和功能为：

- d, --debug: 向生成的目标代码中添加一些用于 Debug 的信息
- g, --gmtree: 打印源程序的语法分析树
- h, --help: 打印帮助信息
- i, --intercode: 打印编译阶段生成的中间代码
- s, --symtable: 打印编译过程中生成的符号表
- m, --mars: 目标代码运行在 Mars 汇编模拟器上，否则运行在组原 CPU 上
- o, <file>: 指定目标代码文件，默认为“result.asm”

各参数中，-g、-i、-s 都是输出编译器前三个阶段的中间结果，可以随意搭配，例如，执行“./gdcf -s -i testfile/calculate.decaf”，将会编译源程序“calculate.decaf”，同时，打印编译器编译过程中生成的符号表和中间代码（由于内容过多，不方便截图展示，因此只作陈述）。

参数-d 用于向目标代码中添加一些 Debug 信息，比如生成的每条汇编指令对应源程序中的哪一行、一些提示信息等等。默认情况下，这些 Debug 信息是不会添加到目标代码中的。不加-d 参数得到的目标代码如图 5-8 所示。

```
1 ori $sp, $0, 1023
2 ori $fp, $0, 1023
3 jal main
4 ori $v0, $0, 10
5 syscall
6
7 getSum:
8 sw $ra, 0($sp)
9 sw $fp, -4($sp)
10 addu $fp, $sp, $0
11 ori $1, $0, 48
12 sw $1, -44($sp)
13 sub $sp, $sp, $1
14 lw $t0, -36($fp)
15 ori $t0, $0, 0
16 sw $t0, -36($fp)
```

图 5-8 gdcf 不加参数-d 生成的目标代码

加上-d 参数后，生成的目标代码如图 5-9 所示。比较图 5-8 和图 5-9 可知，所添加的 Debug 信息有解释说明信息例如“Execute from main”和汇编指令对应的源程序中的代码行数。

```

1 ori $sp, $0, 1023
2 ori $fp, $0, 1023
3 jal main
4 ori $v0, $0, 10
5 syscall
6 ##### Execute from "main" #####
7
8 getSum:
9 sw $ra, 0($sp)
10 sw $fp, -4($sp)
11 addu $fp, $sp, $0
12 ori $l, $0, 48
13 sw $l, -44($sp)
14 sub $sp, $sp, $l
15 ##### Correspond to source file line 3 #####
16 ##### Correspond to source file line 3 #####
17 lw $t0, -36($fp)
18 ori $t0, $0, 0
19 sw $t0, -36($fp)
20 ##### Correspond to source file line 7 #####
21 lw $t1, -40($fp)
22 ori $t1, $0, 1
23 sw $t1, -40($fp)
24 ##### Correspond to source file line 8 #####
25 label6:
26 ##### Correspond to source file line 8 #####
27 lw $t2, -32($fp)
28 addu $t2, $t1, $0
29 sw $t2, -32($fp)
30 ##### Correspond to source file line 8 #####

```

图 5-9 gdcf 加参数-d 生成的目标代码

参数-m 用于指定目标代码的运行平台，加参数-m 表明运行平台为 Mars 汇编模拟器，不加参数-m 表明运行平台为组原 CPU。之所以这样设计，是因为组原的 CPU 中数据存储器 ROM 只有 1024B 的大小，所以需要默认使\$sp 指向最后一个字节作为栈底，但是 Mars 中\$sp 赋值 1023 有问题，除此之外，还有其他组原 CPU 与 Mars 的不兼容特性，所以需要用该参数指定运行平台。

参数-o 用于指定目标代码文件，其后一个参数必须跟目标代码文件的名称。参数-o 使得可以生成任何指定合法名字的目标代码文件，而不一直是默认的“result.asm”。加-o 参数和不加-o 参数生成的目标代码文件分别如图 5-10 和图 5-11 所示。

```

[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ./gdcf testfile/calculate.decaf
[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ls
cpu      Decaf.tab.c  Decaf.y      GrammarTree.c  IntermediateCode.c  lex.yy.c      main.h      Mars.jar      ObjectCode.h  SemanticAnalysis.c  SymbolTable.c  testfile
Decaf.l  Decaf.tab.h  gdcf         GrammarTree.h  IntermediateCode.h  main.c        Makefile    ObjectCode.c  result.asm    SemanticAnalysis.h  SymbolTable.h

```

图 5-10 gdcf 不加参数-o 生成的目标代码

```

[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ./gdcf testfile/calculate.decaf -o calculate.decaf
[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ls
calculate.decaf  Decaf.l      Decaf.tab.h  gdcf         GrammarTree.h  IntermediateCode.h  main.c        Makefile    ObjectCode.c  SemanticAnalysis.c  SymbolTable.c  testfile
cpu             Decaf.tab.c  Decaf.y      GrammarTree.c  IntermediateCode.c  lex.yy.c        main.h        Mars.jar      ObjectCode.h  SemanticAnalysis.h  SymbolTable.h

```

图 5-11 gdcf 加参数-o 生成的目标代码

接下来，将主要测试生成的目标代码的正确性。本次实验所用的测试用例代码如图 5-12 所示。

```
class Claculate
{
    int getSum(int n)
    {
        int i;
        int sum;
        sum = 0;
        for (i = 1; i <= n; i = i + 1)
            sum = sum + i;
        return sum;
    }

    int getMul(int n)
    {
        int i;
        int mul;
        mul = 1;
        for (i = 1; i <= n; i = i + 1)
            mul = mul * i;
        return mul;
    }

    int getMulSum(int n)
    {
        int i;
        int mul;
        int sum;
        mul = 1;
        for (i = 1; i <= n; i = i + 1)
        {
            mul = mul * i;
            sum = sum + mul;
        }
        return sum;
    }
}

class Main
{
    int main()
    {
        int n;
        int result1;
        int result2;
        int result3;
        class Claculate calc;
        calc = new Claculate();
        Print("Input n now");
        n = ReadInteger();
        Print("n is on digits now");
        result1 = calc.getMulSum(n);
        Print(result1);
        Print("The total factorial is on digits");
        result2 = calc.getSum(n);
        Print(result2);
        Print("The sum is on digits");
        result3 = calc.getMul(n);
        Print(result3);
        Print("The factorial is on digits");
        return 0;
    }
}
```

图 5-12 实验 4 测试用例代码

在测试用例中，定义了两个类：Main 和 Calculate，其中类 Calculate 包含 3 个函数成员 getSum、getMul 和 GetMulSum，分别用来计算整数 n 的累加和、阶乘和阶乘和。在类 Main 的 main 函数中，定义了类 Calculate 的一个对象 calc，然后输入一整数 n，接下来将依次调用 calc 的 getMulSum、getSum 和 getMul 方法，分别计算并打印 n 的阶乘和、累加和和阶乘。

执行“./gdcf testfile/calculate.decaf -d -o calculate.asm”将得到目标代码（部分）如图 5-13 和图 5-14 所示。

```

1 ori $sp, $0, 1023
2 ori $fp, $0, 1023
3 jal main
4 ori $v0, $0, 10
5 syscall
6
7 getSum:
8 sw $ra, 0($sp)
9 sw $fp, -4($sp)
10 addu $fp, $sp, $0
11 ori $l, $0, 48
12 sw $l, -44($sp)
13 sub $sp, $sp, $l
14 lw $t0, -36($fp)
15 ori $t0, $0, 0
16 sw $t0, -36($fp)

```

图 5-13 实验 4 测试用例目标代码文件头部

```

798 lw $fp, -4($sp)
799 addu $v0, $t2, $0
800 jr $ra
801 ##### Correspond to source file line 60 #####
802
803 booth_mul:
804 srl $a0, $a0, 16
805 addi $a2, $0, 16
806 add $v0, $0, $a0
807 addi $a3, $0, 0
808 booth_loop:
809 andi $t2, $a0, 1
810 beq $a3, $t2, booth_l1
811 slt $t3, $t2, $a3
812 beq $t3, $0, booth_l2
813 add $v0, $v0, $a1
814 j booth_l1
815 booth_l2:
816 sub $v0, $v0, $a1
817 booth_l1:
818 sra $v0, $v0, 1
819 add $a3, $t2, $0
820 sra $a0, $a0, 1
821 addi $a2, $a2, -1
822 bne $a2, $0, booth_loop
823 add $s0, $v0, $0
824 jr $ra
825 ##### Use booth multiply to replace instruction "mul" #####

```

图 5-14 实验 4 测试用例目标代码文件尾部

注意到，在目标代码文件的头部，编译器添加了初始化代码；在目标文件代码的尾部，编译器添加了 booth 乘法的子程序。

5.5 目标代码运行结果展示（综合组原 CPU）

首先测试生成的目标代码（加-m 参数）在 Mars 汇编模拟器上运行的正确性，测试过程和测试结果如图 5-15 所示。

```
[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ ./gdcf testfile/calculate.decaf -m -d -o calculate.asm
[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$ java -jar Mars.jar calculate.asm
MARS 4.5 Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Input n now
4
n is on digits now
33The total factorial is on digits
10The sum is on digits
24The factorial is on digits

[dracula@localhost]~/Documents/IT_Study/compiler/lab4
$
```

图 5-14 实验 4 测试用例 Mars 运行结果

通过 Mars 的输出结果可以看出，输入的 n 为 4，计算的 $n(4)$ 的阶乘和为 $1+2!+3!+4!=1+2+6+24=33$ ，计算的 $n(4)$ 的累加和为 $1+2+3+4=10$ ，计算的 $n(4)$ 的阶乘为 $4!=24$ 。可见，程序在 Mars 汇编模拟器运行的过程和结果均正确无误。

接下来测试生成的目标代码（不加-m 参数）在组成原理实验完成的 CPU 上运行的正确性。通过编译得到汇编代码后，还需要借助汇编器 Mars 将汇编代码转化机器代码并以十六进制形式导出到文件中，然后将在该文件的文件头处添加“v2.0 raw”便可以将其加载进 logisim 的程序存储器 ROM 中。开始时，CPU 截图如图 5-15 所示。

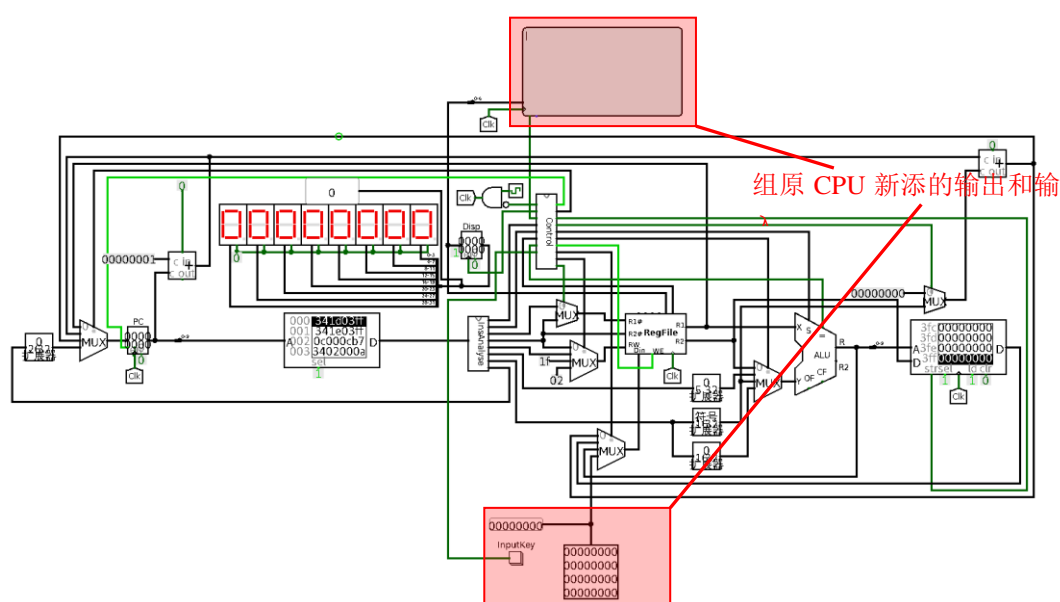


图 5-15 实验 4 测试用例加载进组原 CPU 中

开启时钟模拟，程序将会开始执行，并在需要输入 n 时等待用户输入。等待输入时的运行截图如图 5-16 所示。

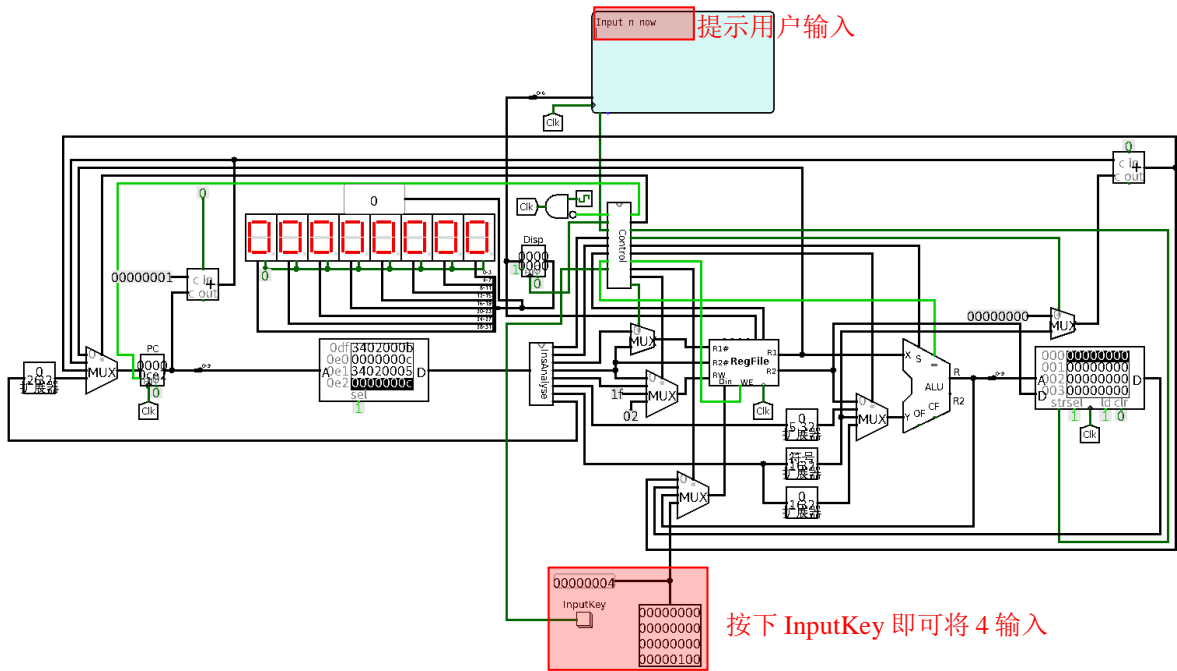


图 5-16 CPU 等待输入 n

在 CPU 运行过程中，可暂停时钟模拟，观察 CPU 所处的状态和当前的计算结果，观察完毕后，可重新启动时钟模拟使 CPU 继续运行。

计算并打印出阶乘和的运行截图如图 5-17 所示，因为版面原因，只截出显像管和哑终端。（显像管显示的是 16 进制，10 进制数值由探测器给出）

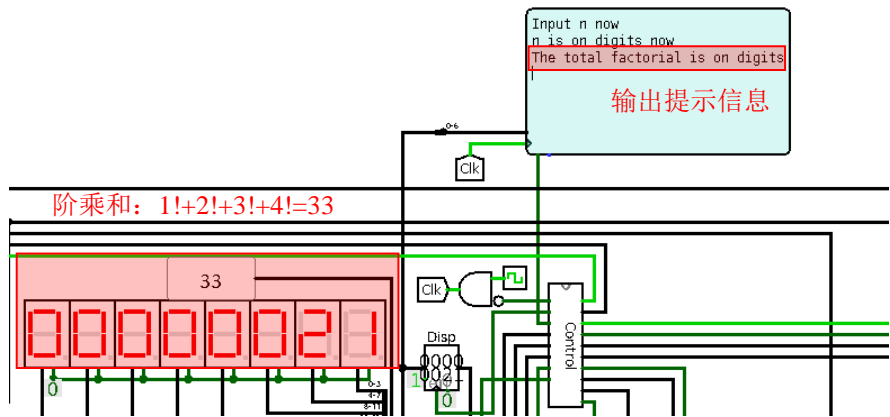


图 5-17 CPU 计算并打印阶乘和

计算并打印出类加和的运行截图如图 5-18 所示，因为版面原因，只截出显像管和哑终端。（显像管显示的是 16 进制，10 进制数值由探测器给出）

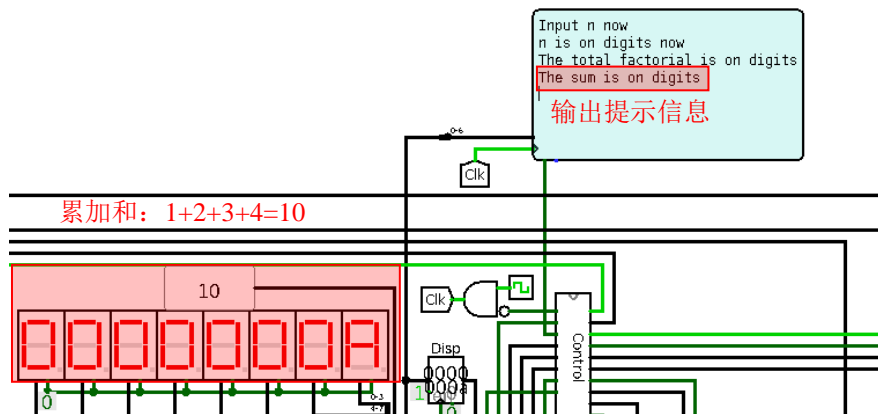


图 5-18 CPU 计算并打印累加和

计算并打印阶乘的运行截图如图 5-19 所示。

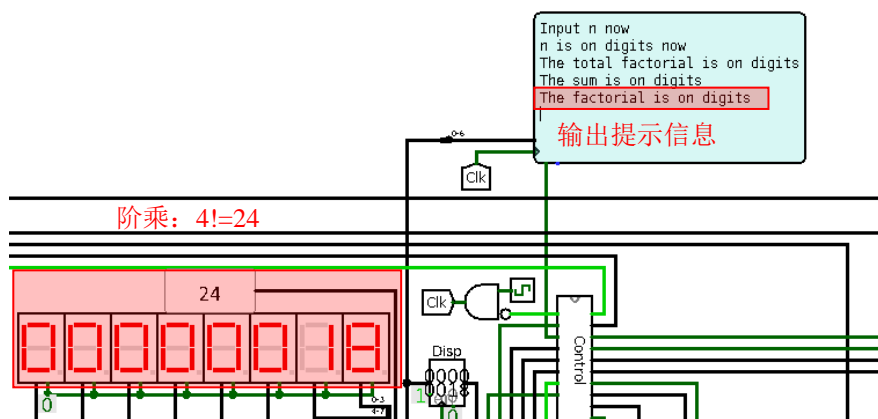


图 5-19 CPU 计算并打印阶乘

通过上述运行测试，可以证明测试用例在组成原理实验完成的 CPU 上正确运行且运行结果完全正确。

6 结束语

6.1 实践课程小结

总体来说，非常满意自己这学期在编译原理实验中的一些成果和心得。

在这个总共费时超过 100 个小时的实验中，我学会了如何编写词法分析器的正规式（正规表达式）和使用自动化生成工具 **Flex** 得到词法分析程序；学会了如何使用自动化生成工具 **Bison** 得到语法分析程序，以及如何在 **Bison** 环境中使用属性文法、语义动作增强文法的表达能力；学会了如何利用源程序的语法分析树，对其进行语义分析；学会了如何对源程序进行分析而能得到其符号表，以及如何维护和检测源程序的作用域栈；学会了中间代码的常用形式三地址码，以及如何将源代码转化为中间代码；学会了如何将中间代码翻译成目标代码，以及如何理解函数调用过程包括栈帧维护、现场保存等。

我所完成的 **Decaf** 编译器主要有以下几个亮点：

1、能够以形式化、结构化、表格化的形式直观美观地打印出编译阶段所用到的符号表，通过打印的符号表，可以清晰地观察各作用域之间的联系（有指针），方便程序员 **Debug** 和了解编译器的翻译方式。

2、对几次实验进行了封装和整合，以命令行参数的方式提供各个阶段数据结构的访问接口，可以打印第一阶段词法和语法分析阶段的语法分析树，可以打印第二阶段语义分析阶段的符号表，可以打印第三阶段中间代码生成阶段的中间代码，可以指定是否需要在目标代码中添加 **Debug** 信息，可以指定目标代码的运行平台，还可以像 **gcc** 那样指定目标代码文件的文件名。

3、创新式地整合了组成原理实验和编译原理实验，使它们互相支持，最终使得组原实验完成的 **CPU** 可以运行编译实验完成的编译器编译出来的程序。为了达到这一点，对组原 **CPU** 作了很多非常巧妙的设计，例如输入的处理和输出的处理；也对编译的编译器作了非常漂亮的设计，例如用 **booth** 乘法子程序代替 **mul** 指令，再例如对条件转移指令的等价处理等等。

当然，这次编译实验除了让我在编译原理方面收获颇丰外，还让我回顾和巩固了数据结构课程上所学的各种数据结构，例如语法分析树用到的孩子兄弟表示法，便是利用二叉树去表示多叉树；再例如语义分析用到的作用域栈，中间代码生成用到的双向链表等，这些曾经没足够重视的数据结构都在编译原理实验中扮演着极其重要的角色。

6.2 自己的亲身体会

这次编译原理实验对我来说，真的是一个巨大的挑战和升华，从一开始的毫无头绪，到后来的得心应手，从一开始的循规蹈矩到后来的不断创新，从一开始的满心恐惧到后来的成就感爆棚，鬼知道我经历了什么？

自己从头到尾实现一个简易的编译器，真的十分有助于自己理解整个编译过程。原来在课堂上，只是单纯的听老师讲这个地方应该如何设计，那个阶段又应该怎么规划，但从来没考虑过这样设计和规划的原因。只有当自己亲身去实践去体会时，才能理解编译器这样设计的原因和初衷，才能感受到编译史上一些前人的思想是多么天才。原来的自己，一直觉得用汇编写代码的人很酷，但是直到做了编译器，我才意识编译器到底有着怎样的地位，正是有了编译器，才有了上层更接近自然语言的高级程序语言。

纵观这学期所学的课程，编译是一个连接软件和硬件的过程，从最顶层的源程序，逐步分析和深入，直到汇编代码级；组成原理则是将汇编代码翻译成的机器指令转化为具体的硬件电路，使得在硬件上可以实现软件的功能，说白了，就是做一个 CPU。不难发现，编译原理和组成原理的中间，只差了一个汇编器，这也是我整合组成原理实验和编译原理实验的一个灵感来源。操作系统则是涵盖范围更广的一门课程，编译原理是其基础，但有时操作系统又为编译原理提供了理论基础，最典型的例子就是函数调用，编译原理中所讲的活动记录 AR，与操作系统所讲的栈帧概念上就十分相似。目前为止，我还未想到如何将组成原理、编译原理以及操作系统整合在一起，虽然感觉上是可行的，但又感觉实现起来难度太大，而且没什么思路。

这次实验也激起了我对开发编译器的热情和激情，最近我就有了一个自己觉得还不错的想法，那就是开发一门基于 51 单片机的面向对象的编程语言。虽然随着 Arm、Stm32 这些单片机的出现，51 已经没有那么火，但其无论在工业界还是学术界，都还扮演着极其重要的角色。然而，目前主流的开发 51 的语言就是 C，虽然 C 语言更确实更适合底层开发，但是其天生缺少一种模块化的概念。现在的单片机，其实完全可以看成是一个个模块相互连接而成，如果针对每个嵌入式模块（例如 wifi 模块、红外线模块、Led 屏模块）都能定义一个类，并规定其接口，那么在实际开发时，只需要设计好电路，分析出所用的模块和参数，就可以在编程时，根据需要定义对象，然后规定对象之间的通信方式，便可以很容易地完成单片机程序的编写。所以，一门面向对象的单片机编程语言将会使得嵌入式编程更方便、开发周期更短。当然，这也只是我一个构想，还需要寒假时进行一定的尝试和实验。

虽然这次编译实验花费的时间较长，也耗费了不少的精力和心思，还和其他

课程的实验有所冲突,但我还是认为这次编译实验让我收获颇丰,不仅仅有知识,还有自信、兴趣和眼界。

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