华中科技大学

编译原理实验报告

课程名称：编译原理

专业班级：ACM1501

学　　号：U201514593

姓　　名：辛　杰

指导教师：刘　铭

报告日期：2018/7/1

计算机科学与技术学院

目录

选题背景

1.1 任务

实现 Decaf 语言编译器：按词法分析、语法分析、语义分析、中间代码生成、目标代码生成五部分完成。最终效果：输入 Decaf 代码，编译得到 MIPS 汇编，在 MIPS 模拟器中运行。

1.2 目标

实验的每一部分的输出是后一部分的输入，词法分析的输入是源代码，语法分析的输入就是词法分析的输出结果，具有如下：

词法分析目标：输入源程序，输出词法序列；

语法分析目标：输入词法序列，输出语法树；

语义分析目标：输入语法树，检测类型错误、语义错误，生成符号表；

中间代码生成：根据语法树以及符号表，生成中间代码；

目标代码生成：做寄存器分配，将中间代码翻译为 MIPS 汇编。

1.3 源语言定义

使用 Stanford CS143 课程的 Decaf 语言语法描述：

Program	::=	Decl+
Decl	::=	VariableDecl | FunctionDecl | ClassDecl | InterfaceDecl
VariableDecl	::=	Variable ;
Variable	::=	Type ident
Type	::=	int | double | bool | string | ident | Type []
FunctionDecl	::=	Type ident (Formals)StmtBlock |
		Void ident (Formals) StmtBlock
Formals	::=	Variable+, | ε
ClassDecl	::=	classident⟨extendsident⟩⟨implementsident+,⟩{ Field* }
Field	::=	VariableDecl | FunctionDecl
InterfaceDecl	::=	interfaceident { Prototype* }
Prototype	::=	Type ident(Formals) ; | voidident(Formals) ;

StmtBlock	::=	{ VariableDecl* Stmt* }
Stmt	::=	⟨Expr⟩; | IfStmt | WhileStmt | ForStmt |
		BreakStmt | ReturnStmt | PrintStmt | StmtBlock
IfStmt	::=	if (Expr) Stmt ⟨else Stmt⟩
WhileStmt	::=	while(Expr) Stmt
ForStmt	::=	for (⟨Expr⟩; Expr ; ⟨Expr⟩) Stmt
ReturnStmt	::=	return⟨Expr⟩;
BreakStmt	::=	break ;
PrintStmt	::=	Print (Expr+,) ;
Expr	::=	LValue = 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 ident | NewArray(Expr , Type)
LValue	::=	ident | Expr .ident | Expr [Expr]
Call	::=	ident(Actuals) | Expr .ident(Actuals)
Actuals	::=	Expr+,| ε
Constant	::=	intConstant | doubleConstant | boolConstant | stringConstant | null

实验一 词法分析和语法分析

2.1 单词文法描述

词法分析使用 GNU 的开源工具 flex。

词法分析的主要工作是分解出源语言中的每个词法单元，Decaf 语言的内置关键字有：void、int、double、bool、string、class、null、while、for、if、else、return、break、extends、this、implements、interface、new、NewArray、Print、ReadInteger、ReadLine、true、false，共 24 个。

除了关键字外，应该由词法分析解析出的词法单元还有：注释块、标识符、常量（整形、实数、字符串），以及还有各种运算符。

关键字和运算符的识别直接使用全字来匹配，例如 void 就使用“void”进行识别，而注释等的识别使用正则表达式进行匹配，具体如下：

首先构建基本字符集的正则表达式描述：

D	[0-9]	数字
L	[a-zA-Z_]	字母
H	[a-fA-F0-9]	十六进制字符
E	([Ee][+-]?{D}+)	科学记数法表示

标识符：[a-zA-Z]({L}|{D})*

整形常量十六进制描述：0[xX]{H}+

整形常量十进制描述：[0-9]+

实数常量描述：{D}+{E}

实数常量科学记数表示：{D}+"."{D}*{E}?

字符常量：\"(\\.|[^\\"\n])*\"

按照以上描述的正则表达式规则即可写出相应的 flex 代码。

2.2 语言文法描述

使用 bison 实现语法分析。

Decaf 语言的语法在上文中已经叙述过了，不再赘述。

2.3 词法分析器的设计

词法分析只需要输出词法单元，按照上文中的正则表达式设计规则 flex 语句即可。

2.4 语法分析器设计

语法分析需要输出语法树，首先根据语法规则设计语法树。

参考上文中给出的语法规则，将每个非终结符当作一个树节点；对于类似 Decl+这样的

闭包规则，将它看作新的节点，例如将 Decl+当作 DeclList 节点。由此提出整个语法树节点结构。

Decl 类型

VarDecl	变量声明节点
ClassDecl	类声明节点
InterfaceDecl	接口声明节点
FnDecl	函数声明节点

算术表达式类型

ArithmeticExpr	算术运算节点
RelationalExpr	关系运算节点
EqualityExpr	等式运算节点
LogicalExpr	逻辑运算节点

常量类型

IntConstant	整数常量节点
DoubleConstant	实数常量节点
BoolConstant	Bool 常量节点
StringConstant	字符串常量节点
NullConstant	常量 Null 节点

其他表达式节点

AssignExpr	赋值表达式节点
This	This 表达式节点
ArrayAccess	访问数组节点
Call	函数调用节点
NewExpr	New 运算节点
NewArrayExpr	New 数组节点

语句块节点

StmtBlock	一般语句块节点
ConditionalStmt	条检测试块节点
ForStmt	For 语句块节点
WhileStmt	While 语句块节点
IfStmt	If 语句块节点
BreakStmt	Break 语句块节点
ReturnStmt	函数返回节点

变量类型节点

AstNamedType	类类型节点
AstArrayType	数组类型节点
AstType	一般类型节点

语法树节点的类型如上表所示，每个节点类型作为一个类，树的根节点为 Program，并依照面向对象的设计思想，所有的节点类全部继承 Node 类。定义虚函数 PrintAST，输出正确的 AST 树。对于每个虚函数的实现，先输出节点所在源程序的行号，接着输出节点的内容。

在编写 bison 程序时，将上述节点定义为 type，按照编写 bison 程序的一般规则完成程序编写。

2.5 语法分析器实现结果展示

对于源程序：

```
class Matrix {
    void Init() {}
    void Set(int x, int y, int value) {}
    int Get(int x, int y) {}
    void PrintMatrix() {
        int i;
        int j;
        for (i = 0; i < 10; i = i + 1) {
            for (j = 0; j < 10; j = j + 1)
                Print(Get(i,j), "\t");
```

```
        Print("\n");
      }
    }
    void SeedMatrix() {
      int i;
      int j;
      for (i = 0; i < 5; i = i + 1)
        for (j = 0; j < 5; j = j + 1)
          Set(i,j, i+j);
      Set(2,3,4);
      Set(4,6,2);
      Set(2,3,5);
      Set(0,0,1);
      Set(1,6,3);
      Set(7,7,7);
    }
}
```

输出的 AST 树为：

```
    Program:
  4     ClassDecl:
  4         Identifier: Matrix
  5         FnDecl:
                (return type) Type: void
  5             Identifier: Init
                (body) StmtBlock:
  6         FnDecl:
                (return type) Type: void
  6             Identifier: Set
  6             (formals) VarDecl:
                    Type: int
  6                 Identifier: x
  6             (formals) VarDecl:
                    Type: int
  6                 Identifier: y
  6             (formals) VarDecl:
                    Type: int
  6                 Identifier: value
                (body) StmtBlock:
  7         FnDecl:
                (return type) Type: int
  7             Identifier: Get
  7             (formals) VarDecl:
                    Type: int
  7                 Identifier: x
```

```
 7          (formals) VarDecl:
                Type: int
 7              Identifier: y
            (body) StmtBlock:
 8      FnDecl:
            (return type) Type: void
 8          Identifier: PrintMatrix
            (body) StmtBlock:
 9              VarDecl:
                    Type: int
 9                  Identifier: i
10              VarDecl:
                    Type: int
10                  Identifier: j
                ForStmt:
11                  (init) AssignExpr:
11                      FieldAccess:
11                          Identifier: i
11                      Operator: =
11                      IntConstant: 0
11                  (test) RelationalExpr:
11                      FieldAccess:
11                          Identifier: i
11                      Operator: <
11                      IntConstant: 10
11                  (step) AssignExpr:
11                      FieldAccess:
11                          Identifier: i
11                      Operator: =
11                      ArithmeticExpr:
11                          FieldAccess:
11                              Identifier: i
11                          Operator: +
11                          IntConstant: 1
                    (body) StmtBlock:
                        ForStmt:
12                          (init) AssignExpr:
12                              FieldAccess:
12                                  Identifier: j
12                              Operator: =
12                              IntConstant: 0
12                          (test) RelationalExpr:
12                              FieldAccess:
12                                  Identifier: j
```

```
12                              Operator: <
12                              IntConstant: 10
12                          (step) AssignExpr:
12                              FieldAccess:
12                                  Identifier: j
12                              Operator: =
12                              ArithmeticExpr:
12                                  FieldAccess:
12                                      Identifier: j
12                                  Operator: +
12                                  IntConstant: 1
                            (body) PrintStmt:
13                              (args) Call:
13                                  Identifier: Get
13                                  (actuals) FieldAccess:
13                                      Identifier: i
13                                  (actuals) FieldAccess:
13                                      Identifier: j
13                              (args) StringConstant: "\t"
                        PrintStmt:
14                          (args) StringConstant: "\n"
17      FnDecl:
            (return type) Type: void
17          Identifier: SeedMatrix
            (body) StmtBlock:
18              VarDecl:
                    Type: int
18                  Identifier: i
19              VarDecl:
                    Type: int
19                  Identifier: j
                ForStmt:
20                  (init) AssignExpr:
20                      FieldAccess:
20                          Identifier: i
20                      Operator: =
20                      IntConstant: 0
20                  (test) RelationalExpr:
20                      FieldAccess:
20                          Identifier: i
20                      Operator: <
20                      IntConstant: 5
20                  (step) AssignExpr:
20                      FieldAccess:
```

```
20                      Identifier: i
20                  Operator: =
20                  ArithmeticExpr:
20                      FieldAccess:
20                          Identifier: i
20                      Operator: +
20                      IntConstant: 1
                (body) ForStmt:
21                  (init) AssignExpr:
21                      FieldAccess:
21                          Identifier: j
21                      Operator: =
21                      IntConstant: 0
21                  (test) RelationalExpr:
21                      FieldAccess:
21                          Identifier: j
21                      Operator: <
21                      IntConstant: 5
21                  (step) AssignExpr:
21                      FieldAccess:
21                          Identifier: j
21                      Operator: =
21                      ArithmeticExpr:
21                          FieldAccess:
21                              Identifier: j
21                          Operator: +
21                          IntConstant: 1
22                  (body) Call:
22                      Identifier: Set
22                      (actuals) FieldAccess:
22                          Identifier: i
22                      (actuals) FieldAccess:
22                          Identifier: j
22                      (actuals) ArithmeticExpr:
22                          FieldAccess:
22                              Identifier: i
22                          Operator: +
22                          FieldAccess:
22                              Identifier: j
23          Call:
23              Identifier: Set
23              (actuals) IntConstant: 2
23              (actuals) IntConstant: 3
23              (actuals) IntConstant: 4
```

```
24          Call:
24              Identifier: Set
24              (actuals) IntConstant: 4
24              (actuals) IntConstant: 6
24              (actuals) IntConstant: 2
25          Call:
25              Identifier: Set
25              (actuals) IntConstant: 2
25              (actuals) IntConstant: 3
25              (actuals) IntConstant: 5
26          Call:
26              Identifier: Set
26              (actuals) IntConstant: 0
26              (actuals) IntConstant: 0
26              (actuals) IntConstant: 1
27          Call:
27              Identifier: Set
27              (actuals) IntConstant: 1
27              (actuals) IntConstant: 6
27              (actuals) IntConstant: 3
28          Call:
28              Identifier: Set
28              (actuals) IntConstant: 7
28              (actuals) IntConstant: 7
28              (actuals) IntConstant: 7
```

实验二 语义分析

3.1 语义分析方法描述

语义分析的主要任务是输出符号表，检测类型错误，语义错误。由于此语言是类 Java 语言，根据语言规则， 语义分析无法在第一趟遍历完成，例如：函数可以先使用再声明。所以采用多趟编译。

遍历语法分析得到的语法树，对每个 AST 节点进行对应的语义检查，顺便生成符号表。在语义检查过程中，作用域是一个重要的概念，要对语法节点生成正确的作用域信息。

3.2 符号表结构定义

符号表可以分为 Class 类型的符号表，Interface 类型的符号表，函数内部的符号表，内部语句块（while、for）的符号表以及全局符号表。

根据符号表的类型不同，定义的方法也有不同。而且符号表与定义域之间并没有完全明晰的界限，例如 Class，它既有拥有一个符号表，也是一个作用域，而且符号表需要输出的内容可以从作用域中得到，所以可以将作用域看作动态的符号表，语义分析结束之后就是符号表。先定义符号表和作用域的基本类型：

符号的定义：每个符号拥有一个名称、类型、所属作用域、所属节点；

基本作用域的定义：每个作用域中保存 hash 表，保存所拥有的符号；父作用域；本作用域下的内部语句块作用域；

有了符号的定义和基本作用域的定义，给出符号表设计的定义：

Class 类的符号表 ： Class 是一个基本符号，也是一个基本作用域，所以定义时 ClassSymbol 直接继承符号和作用域两个基本类；对于 Class 类型来说，拥有一个 VTable，父类的指针，接口类型链表。

函数类型的符号表 ：和 Class 一样，函数也即是一个函数，也是一个作用域；除此之外，还有返回类型与参数列表；参数列表的定义就是一个类型链表。

3.3 错误类型码定义

语义分析阶段检测的语义错误有以下：

1. 声明冲突，在同一作用域内定义同名符号；
2. 接口没有声明，类 J 继承了一个接口，但此接口并没有声明；
3. 接口没有实现，实现了接口 I 的类 J 并没有全部实现 I 中的函数；
4. 内置函数 New 的参数并不是类类型；
5. 错误的操作符使用，例如在类类型间使用了算术运算符；
6. This 语句在 Class 语句块外；
7. 数组的访问下表不是整数类型；
8. 内置函数 New 的参数不是整数类型；
9. 下表访问运算符作用到了非数组类型上；
10. 函数调用的实参数量与定义的形参数量不匹配；

11. 函数调用的实参类型与定义的实参类型不匹配；
12. 类成员访问运算符没有找到实际的成员，例如 A.b，而类 A 中并没有成员 b；
13. 测试语句块中语句的结果并不是 Bool 类型；
14. 函数 Return 语句返回的类型与定义不匹配；
15. Break 语句没有出现在循环语句块中；

除此之外，还有一些错误，例如 class 继承了自己等。

3.4 语义分析实现技术

对于每个 AST 节点，都有它独特的语义分析技巧，所以每个 AST 节点定义独自的语义分析处理函数，具体的处理方式如下：

Program 节点：

1. 设置全局作用域；
2. 遍历 Program 下的所有 Decl 节点，将其符号加入全局作用域中；
3. 对于每一个 Decl，调用它的语义分析处理函数。

ClassDecl 节点：

1. 从上层作用域中得到关于 Class 符号的描述（Class 既是符号也是作用域）；
2. 设置继承来的类；
3. 扫描类中的全部声明，将他们插入类作用域中；
4. 对于每一个声明，检查是否有语法错误；
5. 最后检查是否所有的接口都被实现。

FnDecl 节点（函数声明节点）：

1. 从上层作用域中得到关于此函数的符号；
2. 扫描参数列表，加入函数作用域；
3. 扫描 body，对每个语句块，生成每个块的作用域。

算术表达式节点：

1. 只有 double 类型和 int 类型可以使用算术运算符，目标是类型检查
2. 首先对运算符左边进行类型检查，得到类型 I（借用符号表）
3. 接着对运算符右边进行类型检查，得到类型 J（借用符号表）
4. 查看 I 与 J 是否都是 double 或者 int
5. 根据结果输出错误或者接受语句

其他的如关系表达式，常量表达式与算术表达式的处理方式类似，不再赘述。

类访问节点：

1. 类访问由 base.exp 构成
2. 首先检查 base 的类型是否是类，得到他的符号表
3. 其次在此符号表中查询类中是否有对应的符号

函数调用节点：

1. 函数调用的表达类似 fun 或者 base.fn，只列举前一种，后一种容易类推
2. 从符号表中找到该函数，对每个参数进行类型匹配查询

剩余的节点类型检查也容易写出，语义分析的关键就是符号表的构建与类型匹配的检查。

3.5 语义分析结果展示

1. 对于正确的源代码，语义分析将给出符号表的描述，下面是一份正确的源代码：

```
class QueueItem {
    int data;
    QueueItem next;
    QueueItem prev;

    void Init(int data, QueueItem next, QueueItem prev) {
        this.data = data;
        this.next = next;
        next.prev = this;
        this.prev = prev;
        prev.next = this;
    }
    int GetData() {
        return this.data;
    }
    QueueItem GetNext() { return next;}
    QueueItem GetPrev() { return prev;}
    void SetNext(QueueItem n) { next = n;}
    void SetPrev(QueueItem p) { prev = p;}
}
class Queue {
    QueueItem head;
    void Init() {
        this.head = new QueueItem;
        this.head.Init(0, this.head, this.head);
    }
    void EnQueue(int i) {
        QueueItem temp;
        temp = new QueueItem;
        temp.Init(i, this.head.GetNext(), this.head);
    }
    int DeQueue() {
        int val;
        if (this.head.GetPrev() == this.head) {
            Print("Queue Is Empty");
            return 0;
        } else {
            QueueItem temp;
            temp = this.head.GetPrev();
            val = temp.GetData();
```

```
        temp.GetPrev().SetNext(temp.GetNext());
        temp.GetNext().SetPrev(temp.GetPrev());
    }
    return val;
  }
}

void main() {
  Queue q;
}
```

符号表输出：

全局符号表

```
----------------------------------------
|               Global                 |
----------------------------------------
+----------+----------+----------------+
|Line Number|Identifier|Type            |
+----------+----------+----------------+
|3         |QueueItem |QueueItem(class)|
+----------+----------+----------------+
|25        |Queue     |Queue(class)    |
+----------+----------+----------------+
|53        |main      |(*void)()       |
+----------+----------+----------------+
```

函数 init 符号表

```
----------------------------------------
|       Function QueueItem::Init       |
----------------------------------------
+----------+----------+----------------+
|Line Number|Identifier|Type            |
+----------+----------+----------------+
|8         |data      |int             |
+----------+----------+----------------+
|8         |next      |QueueItem(class)|
+----------+----------+----------------+
|8         |prev      |QueueItem(class)|
+----------+----------+----------------+
```

类 QueueItem 符号表

```
------------------------------------------------------------------------------
|                            Class QueueItem                                 |
------------------------------------------------------------------------------
+----------+----------+-------------------------------------------------+----+
|Line Number|Identifier|Type                                             |Base|
+----------+----------+-------------------------------------------------+----+
|4         |data      |int                                              |this|
+----------+----------+-------------------------------------------------+----+
|5         |next      |QueueItem(class)                                 |this|
+----------+----------+-------------------------------------------------+----+
|6         |prev      |QueueItem(class)                                 |this|
+----------+----------+-------------------------------------------------+----+
|8         |Init      |(*void)(int,QueueItem(class),QueueItem(class))|this|
+----------+----------+-------------------------------------------------+----+
|15        |GetData   |(*int)()                                         |this|
+----------+----------+-------------------------------------------------+----+
|18        |GetNext   |(*QueueItem(class))()                            |this|
+----------+----------+-------------------------------------------------+----+
|19        |GetPrev   |(*QueueItem(class))()                            |this|
+----------+----------+-------------------------------------------------+----+
|20        |SetNext   |(*void)(QueueItem(class))                        |this|
+----------+----------+-------------------------------------------------+----+
|21        |SetPrev   |(*void)(QueueItem(class))                        |this|
+----------+----------+-------------------------------------------------+----+
```

可以看到给出的符号表较好的展示了

2. 对于错误的源代码，语义分析可以给出错误的原因与位置，以下是一份具有语义错误的源代码：

```
void test(int a) {}
class Cow {}
void main() {
    int[] a;
    test(3.0 * 5.6 - 1.0);
    NewArray(true && false, bool);
    if (NewArray(5, int)) 1;
    if (ReadInteger()) 1;
    if (ReadLine()) 1;
    if (new Cow) 1;
    if (a[3+5]) 1;
    a[true && false];
    a[test(4)];
}
```

语义分析结果输出：

```
*** Error line 8.
   test(3.0 * 5.6 - 1.0);
        ^^^^^^^^^^^^^^^^
*** Incompatible argument 1: double given, int expected


*** Error line 9.
   NewArray(true && false, bool);
            ^^^^^^^^^^^^^
*** Size for NewArray must be an integer


*** Error line 10.
   if (NewArray(5, int)) 1;
       ^^^^^^^^^^^^^^^^
*** Test expression must have boolean type


*** Error line 11.
   if (ReadInteger()) 1;
       ^^^^^^^^^^^^^
*** Test expression must have boolean type


*** Error line 12.
   if (ReadLine()) 1;
       ^^^^^^^^^^
*** Test expression must have boolean type


*** Error line 13.
   if (new Cow) 1;
       ^^^^^^^
*** Test expression must have boolean type
```

4 中间代码生成

4.1 中间代码格式定义

生成的中间代码命令格式有以下几种：

LoadConstant	%s = %d	加载常数	LoadString	%s = %s	加载字符
LoadLabel	%s = %s	加载标志	Assign	%s = %s	赋值
Load	%s = *(%s)	读内存	Store	*(%s)= %s	存入内存
BinaryOp	%s=A op B	运算	PopParam	PopParams %d	出栈
Goto	Goto %s	Goto 指令	IfZ	IfZ%sGoto %s	条件跳转
BeginFunc	BeginFunc %d	函数开始	EndFunc	EndFunc	函数结束
Return	Return %s	函数返回	PushParam	PushParam %s	入栈
ACall	ACall %s	函数调用	VTable	VTableforclass %s	Vtable

4.2 中间代码生成过程

中间代码的生成流程与语义分析很类似，都是对每个 AST 节点进行检查。

Program 节点：

1. 遍历每个 decl，依次生成每个 decl 的代码

ClassDecl 节点：

1. 得到此 class 的符号表
2. 得到 class 的 VTable 描述（从符号表中很容易构建）
3. 对每个方法，生成他的代码

FnDecl 节点：

1. 得到此函数的符号表
2. 依照函数名生成一个唯一的 Label
3. 对于每一个形参，确定其相对于栈帧的偏移，保存起来
4. 生成函数体的代码；
5. 退出函数标记。

算术运算节点：

1. 加载右操作符（加载操作下文叙述）；
2. 加载左操作符；
3. 生成赋值操作。

函数调用节点：

1. 对每个实参，首先加载值，然后对它 push
2. 生成 Call 语句
3. 如果有返回值，加载

数组访问的加载操作：

1. 取得偏移的 label，在 Array 的 label 上 Load 相对偏移的值

普通值的加载操作：

取得该 label 即可。

4.5 中间代码生成结果展示

源代码使用语义分析的样例一中的前部分。

中间代码结果：

```
VTable QueueItem =
        _QueueItem.GetData,
        _QueueItem.GetNext,
        _QueueItem.GetPrev,
        _QueueItem.Init,
        _QueueItem.SetNext,
        _QueueItem.SetPrev,
_QueueItem.Init:
        BeginFunc 40
        _L0 = 4
        _L1 = this + _L0
        *(_L1) = data
        _L2 = 8
        _L3 = this + _L2
        *(_L3) = next
        _L4 = 12
        _L5 = next + _L4
        *(_L5) = this
        _L6 = 12
        _L7 = this + _L6
        *(_L7) = prev
        _L8 = 8
        _L9 = prev + _L8
        *(_L9) = this
        EndFunc
_QueueItem.GetData:
        BeginFunc 12
        _L10 = 4
        _L11 = this + _L10
        _L12 = *(_L11)
        Return _L12
        EndFunc
_QueueItem.GetNext:
        BeginFunc 12
        _L13 = 8
        _L14 = this + _L13
        _L15 = *(_L14)
```

5 目标代码生成

5.1 指令集选择

选用 MIPS 指令集。

5.2 寄存器分配算法

首先建立寄存器数据结构，每个寄存器有一个它目前所表示的中间代码变量名，目前是否在使用，以及里面的数据是否为脏数据（写过之后变为脏数据）；

寄存器分配选用较为简单的算法，维护一个未使用寄存器队列和已使用寄存器队列。

寄存器分配函数将以 label（中间代码的变量）和两个要避免分配的寄存器名为参数，返回一个分配给 label 的寄存器。

1. 如果这个 label 已经在寄存器中，直接返回；
2. 如果存在未使用的寄存器直接返回此寄存器；
3. 寻找一个不是 dirty 的寄存器，将新 label 代替旧 label
4. 没有非 dirty 的寄存器，则将寄存器值写回内存，然后代替。

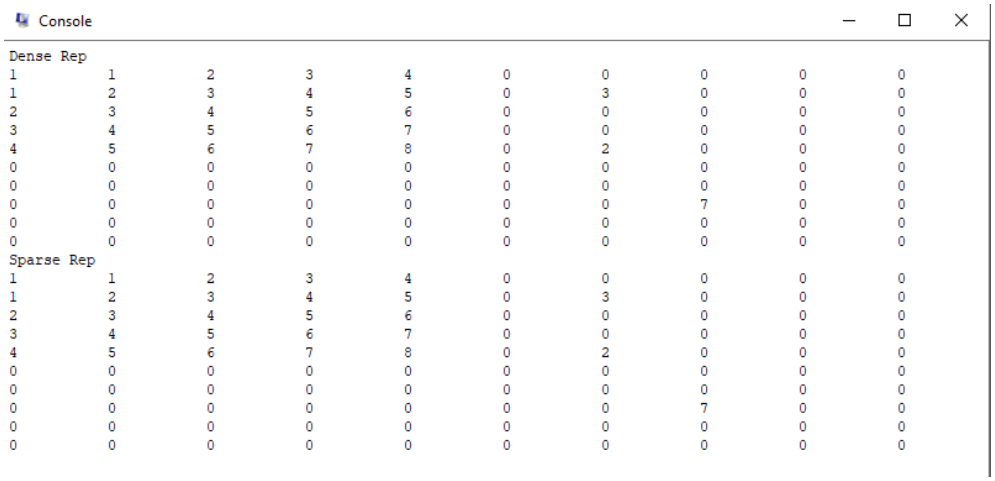
5.3 目标代码生成算法

对于每一条中间代码，生成合适的目标代码，中间代码的格式和 MIPS 汇编已经有了很大的相似程度，但是中间代码在生成时假设的是无限的寄存器，在生成目标代码时，只需按寄存器分配算法分配寄存器即可。

5.4 目标代码生成结果展示

目标代码十分冗长，只展示运行结果。

源程序是一个关于矩阵的赋值以及输出程序，他的控制台输出结果：



6 结束语

6.1 实验小结

基本完成了每个实验要求完成的工作，从编译程序代码的命令：

```
g++ -std=gnu++11 -g -Wall -Wno-unused -Wno-sign-compare ^
main.cc parser/lex.yy.c parser/y.tab.c ^
ast/ast_decl.cc ast/ast_expr.cc ast/ast_stmt.cc ast/ast_type.cc ast/ast.cc ^
error/errors.cc tools/utility.cc tools/windows.cc tools/TextTable.cc ^
semantic/semantic.cc semantic/semantic_decl.cc semantic/semantic_type.cc semantic/semantic_stmt.cc semantic/semantic_e
symbol/Scope.cc symbol/Generate.cc symbol/SymbolType.cc ^
symbol/InternalType.cc symbol/FunctionSymbol.cc symbol/ClassSymbol.cc symbol/InterfaceSymbol.cc ^
generate/codegen.cc generate/mips.cc generate/rw.cc ^
generate/tac.cc generate/tac_decl.cc generate/tac_expr.cc generate/tac_stmt.cc generate/tac_type.cc ^
```

中可以看到工程的复杂性，每个实验涉及的领域其实都各有不同，完成实验给了很大的成就感。

6.2 自己的心得体会

实验的代码量巨大，工作量也就很大，前前后后花费了大量时间，几百个小时锻炼了代码能力以及熟悉了编译器的设计过程。

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