编译设计文档

参考编译器介绍

选择的编译器是PL/0-Compiler，PL/0是Pascal语言的一个子集，其编译程序采用以语法分析为核心，一遍扫描的编译方法。词法分析和代码生成是独立的子程序供语法分析程序调用。在进行语法分析的同时，提供了出错报告和出错恢复的功能。在源程序没有错误编译通过的情况下，调用解释程序解释执行生成的类PCODE代码。

词法分析程序

词法分析子程序名为getsym，功能是从源程序中读出一个单词符号（token），把它的信息放入全局变量sym、id和num中,法分析器需要获得单词时，直接从这三个变量中获得
具体流程如注释所说。

代码如下：

```
procedure getsym;    {词法分析程序}
var i,j,k : integer;    {声明计数变量}
procedure getch;
    begin
      if cc = ll  { get character to end of line }    {如果读完了一行（行指针与该行长度相等）}
      then begin { read next line }    {开始读取下一行}
             if eof(fin)    {如果到达文件末尾}
             then begin
                    writeln('program incomplete');    {报错}
                    close(fin);    {关闭文件}
                    exit;    {退出}
                  end;
             ll := 0;    {将行长度重置}
             cc := 0;    {将行指针重置}
             write(cx:4,' ');  { print code address }    {输出代码地址，宽度为4}
             while not eoln(fin) do    {当没有到行末时}
               begin
                 ll := ll+1;    {将行缓冲区的长度+1}
                 read(fin,ch);    {从文件中读取一个字符到ch中}
                 write(ch);    {控制台输出ch}
                 line[ll] := ch    {把这个字符放到当前行末尾}
               end;
             writeln;    {换行}
             readln(fin);    {源文件读取从下一行开始}
             ll := ll+1;    {行长度计数加一}
             line[ll] := ' ' { process end-line }    {行数组最后一个元素为空格}
           end;
      cc := cc+1;    {行指针+1}
      ch := line[cc]    {读取下一个字符，将字符放进全局变量ch}
    end; { getch }
  begin { procedure getsym;   }    {标识符识别开始}
    while ch = ' ' do    {去除空字符}
      getch;    {调用上面的getch过程}
    if ch in ['a'..'z']    {如果识别到字母，那么有可能是保留字或标识符}
    then begin  { identifier of reserved word }    {开始识别}
           k := 0;    {标识符指针置零，这个量用来统计标识符长度}
           repeat    {循环}
             if k < al    {如果k的大小小于标识符的最大长度}
             then begin
                    k := k+1;    {k++}
                    a[k] := ch    {将ch写入标识符暂存变量a}
                  end;
             getch    {获取下一个字符}
           until not( ch in ['a'..'z','0'..'9']);    {直到读出的不是数字或字母的时候，标识符结束}
           if k >= kk        { kk : last identifier length }    {若k比kk大}
           then kk := k    {kk记录当前标识符的长度k}
           else repeat    {循环}
                  a[kk] := ' ';        {标识符最后一位为空格}
                  kk := kk-1    {k--}
                until kk = k;    {直到kk等于当前标识符的长度，这样做的意义是防止上一个标识符存在a中的内
```

```
        id := a;    {id保存标识符名}
        i := 1;    {i指向第一个保留字}
        j := norw;   { binary search reserved word table }    {二分查找保留字表，将j设为保留字的
        repeat
          k := (i+j) div 2;    {再次用到k，但这里只是作为二分查找的中间变量}
          if id <= word[k]    {若当前标识符小于或等于保留字表中的第k个，这里的判断依据的是字典序，
          then j := k-1;        {j = k-1}
          if id >= word[k]    {若当前标识符大于或等于保留字表中的第k个}
          then i := k+1         {i = k+1}
        until i > j;        {查找结束条件}
        if i-1 > j    {找到了}
        then sym := wsym[k]    {将找到的保留字类型赋给sym}
        else sym := ident    {未找到则把sym置为ident类型，表示是标识符}
      end
else if ch in ['0'..'9']    {如果字符是数字}
      then begin  { number }
             k := 0;    {这里的k用来记录数字的位数}
             num := 0;    {num保存数字}
             sym := number;    {将标识符设置为数字}
             repeat    {循环开始}
               num := 10*num+(ord(ch)-ord('0'));    {将数字字符转换为数字并拼接起来赋给num}
               k := k+1;    {k++}
               getch    {继续读字符}
             until not( ch in ['0'..'9']);    {直到输入的不再是数字}
             if k > nmax    {如果数字的位数超过了数字允许的最大长度}
             then error(30)    {报错}
           end
else if ch = ':'    {当字符不是数字或字母，而是':'时}
      then begin
             getch;    {读下一个字符}
             if ch = '='    {如果下一个字符是'='}
             then begin
                    sym := becomes;    {将标识符sym设置为becomes，表示复制}
                    getch    {读下一个字符}
                  end
             else sym := nul {否则，将标识符设置为nul，表示非法}
           end
else if ch = '<'    {当读到的字符是'<'时}
        then begin
               getch;    {读下一个字符}
               if ch = '='    {若读到的字符是'='}
               then begin
                      sym := leq;    {则sym为leq,表示小于等于}
                      getch    {读下一个字符}
                    end
               else if ch = '>'    {若读到的字符是'>'}
                    then begin
                           sym := neq;    {则sym为neq,表示不等于}
                           getch    {读下一个字符}
                         end
               else sym := lss    {否则,sym设为lss,表示小于}
```

```
            end
else if ch = '>'    {若读到的是'>'}
        then begin
               getch;    {读下一个字符}
               if ch = '='    {若读到的是'='}
               then begin
                      sym := geq;    {sym设为geq,表示大于等于}
                      getch    {读下一个字符}
                    end
               else sym := gtr    {否则,sym设为gtr,表示大于}
             end
else begin    {若非上述几种符号}
       sym := ssym[ch];    {从ssym表中查到此字符对应的类型,赋给sym}
       getch    {读下一个字符}
     end
end; { getsym }
```

语法分析程序

其采用了自顶向下的递归子程序法。语法分析同时也根据程序的语意生成相应的代码，并提供了出错处理的机制。语法分析主要由分程序分析代码块（block）、常量定义分析（constdeclaration）、变量定义分析（vardeclaration）、语句分析（statement）、表达式处理（expression）、项处理（term）、因子处理（factor）和条件处理（condition）构成。除此之外，还有错误处理、代码生成、符号表的维护与使用等相关接口来辅助语法生成

代码块分析

首先调用代码块（block）处理分程序。过程入口参数置为：0层、符号表位置0、出错恢复单词集合为句号、声明符或语句开始符号。

进入block过程后，进行了内存相关的指针分配。然后用tx0记录下当前符号表位置并产生一条jmp指令，准备跳转到主程序的开始位置，同时在符号表的当前位置记录下这个jmp指令在代码段中的位置。在判断了嵌套层数没有超过规定的层数后，开始分析源程序。

首先判断是否遇到了常量声明与变量声明，并维护符号表，。变量定义过程中会用dx变量记录下局部数据段分配的空间个数。接着处理过程（procedure）声明和定义，把过程的名字和所在的层次记入符号表。

随后进入语句的处理，这时的代码分配指针cx的值正好指向语句的开始位置，这个位置正是前面的jmp指令需要跳转到的位置,并在符号表中记录下当前的代码段分配地址和局部数据段要分配的大小（dx的值）。生成一条int指令，分配dx个空间，作为这个分程序段的第一条指令。接下来就调用语句处理(statement)。分析完成后，生成操作数为0的opr指令，用于从分程序返回。具体处理见注释

```
begin  {   procedure block( lev,tx : integer; fsys : symset );
    var  dx : integer;  /* data allocation index */
    tx0: integer;  /*initial table index */
    cx0: integer;  /* initial code index */              }    {分程序处理过程开始}
    dx := 3;    {记录运行栈空间的栈顶位置,设置为3是因为需要预留SL,DL,RA的空间}
    tx0 := tx;    {记录当前符号表的栈顶位置}
    table[tx].adr := cx;    {符号表当前位置的偏移地址记录下一条生成代码开始的位置}
    gen(jmp,0,0); { jump from declaration part to statement part }    {产生一条jmp类型的无条件跳转
    if lev > levmax    {当前过程所处的层次大于允许的最大嵌套层次}
    then error(32);    {报32号错误}

    repeat    {循环开始}
      if sym = constsym    {如果符号类型是const保留字}
      then begin
             getsym;    {获取下一个sym类型}
             repeat    {循环开始}
               constdeclaration;    {处理常量声明}
               while sym = comma do    {如果声明常量后接的是逗号,说明常量声明没有结束,进入下一循环}
                 begin
                   getsym;    {获取下一个sym类型}
                   constdeclaration    {处理常量声明}
                 end;
               if sym = semicolon    {如果读到了分号,说明常量声明已经结束了}
               then getsym    {获取下一个sym类型}
               else error(5)    {如果没有分号,报5号错误}
             until sym <> ident    {循环直到遇到下一个标志符}
           end;
      if sym = varsym    {如果读到的是var保留字}
      then begin
             getsym;    {获取下一个sym类型}
             repeat        {循环开始}
               vardeclaration;    {处理变量声明}
               while sym = comma do    {如果读到了逗号,说明声明未结束,进入循环}
                 begin
                   getsym;    {获取下一个sym类型}
                   vardeclaration    {处理变量声明}
                 end;
               if sym = semicolon    {如果读到了分号,说明所有声明已经结束}
               then getsym    {获取下一个sym类型}
               else error(5)    {如果未读到分号,则报5号错误}
             until sym <> ident;    {循环直到读到下一个标识符为止}
           end;
      while sym = procsym do    {如果读到proc关键字}
        begin
          getsym;    {获取下一个sym类型}
          if sym = ident    {第一个符号应该是标识符类型}
          then begin
                 enter(prosedure);    {将该符号录入符号表,类型为过程,因为跟在proc后面的一定是过程名}
                 getsym    {获取下一个sym类型}
               end
```

```
            else error(4);    {如果第一个符号不是标识符类型,报4号错误}
            if sym = semicolon    {如果读到了分号,说明proc声明结束}
            then getsym    {获取下一个sym类型}
            else error(5);    {如果声明过程之后没有跟分号,报5号错误}
            block(lev+1,tx,[semicolon]+fsys);    {执行分程序的分析过程}
            if sym = semicolon    {递归调用返回后应该接分号}
            then begin    {如果接的是分号}
                   getsym;    {获取下一个sym类型}
                   test( statbegsys+[ident,procsym],fsys,6)    {测试当前的sym是否合法}
                 end
            else error(5)    {如果接的不是分号,报5号错误}
          end;
        test( statbegsys+[ident],declbegsys,7)    {测试当前的sym是否合法}
      until not ( sym in declbegsys );    {一直循环到sym不在声明符号集中为止}
      code[table[tx0].adr].a := cx;  { back enter statement code's start adr. }    {将之前生成无条件
      with table[tx0] do    {对符号表新加记录}
        begin
          adr := cx; { code's start address }    {记录当前代码的分配为止}
        end;
      cx0 := cx;    {记录当前代码分配的地址}
      gen(int,0,dx); { topstack point to operation area }    {生成int指令,分配dx个空间}
      statement( [semicolon,endsym]+fsys);    {调用语法分析程序}
      gen(opr,0,0); { return }    {生成0号gen程序,完成返回操作}
      test( fsys, [],8 );    {测试当前状态是否合法,有问题报8号错误}
      listcode;    {列出该block所生成的PCODE}
    end { block };
```

常量定义分析

通过循环，反复获得标识符和对应的值，存入符号表。符号表中记录下标识符的名字和它对应的值。

```
procedure constdeclaration;     {处理常量声明的过程}
    begin
      if sym = ident    {如果sym是ident说明是标识符}
      then begin
             getsym;    {获取下一个sym类型}
             if sym in [eql,becomes]    {如果sym是等号或者赋值符号}
             then begin
                    if sym = becomes    {若是赋值符号}
                    then error(1);    {报一号错误,因为声明应该使用等号}
                    getsym;  {获取下一个sym类型}
                    if sym = number    {如果读到的是数字}
                    then begin
                           enter(constant);    {将该常量入表}
                           getsym    {获取下一个sym类型}
                         end
                    else error(2)    {如果等号后面不是数字,报2号错误}
                  end
             else error(3)    {如果常量标识符后面接的不是等号或赋值符号,报三号错误}
           end
      else error(4)    {如果常量声明第一个符号不是标识符,报4号错误}
    end; { constdeclaration }    {常量声明结束}
```

变量定义分析

与常量类似

```
  procedure vardeclaration;     {变量声明过程}
    begin
      if sym = ident    {变量声明要求第一个sym为标识符}
      then begin
             enter(variable);    {将该变量入表}
             getsym    {获取下一个sym类型}
           end
      else error(4)    {如果第一个sym不是标识符,抛出4号错误}
    end; { vardeclaration }
```

语句处理

是一个递归过程。通过调用表达式处理、项处理、因子处理等过程及递归调用自己来实现对语句的分析。当遇到begin/end语句时，就递归调用自己来分析。分析的同时生成相应的类PCODE指令和错误处理指令。

```
begin { procedure statement( fsys : symset );
     var i,cx1,cx2: integer; }    {声明处理过程}
     if sym = ident    {如果以标识符开始}
     then begin
            i := position(id);    {i记录该标识符在符号表中的位置}
            if i= 0    {如果返回0则是没找到}
            then error(11)    {抛出11号错误}
            else if table[i].kind <> variable    {如果在符号表中找到了该符号,但该符号的类型不是变量
                 then begin { giving value to non-variation }    {那么现在的操作属于给非变量赋值}
                        error(12);    {报12号错误}
                        i := 0    {将符号表标号置零}
                      end;
            getsym;    {获取下一个sym类型}
            if sym = becomes    {如果读到的是赋值符号}
            then getsym    {获取下一个sym类型}
            else error(13);    {如果读到的不是赋值符号,报13号错误}
            expression(fsys);    {赋值符号的后面可以跟表达式,因此调用表达式处理子程序}
            if i <> 0    {如果符号表中找到了合法的符号}
            then
              with table[i] do    {使用该表项的内容来进行操作}
                 gen(sto,lev-level,adr)    {生成一条sto指令用来将表达式的值写入到相应变量的地址}
         end
     else if sym = callsym    {如果读到的符号是call关键字}
     then begin
            getsym;    {获取下一个sym类型}
            if sym <> ident    {如果call后面跟的不是标识符}
            then error(14)    {报14号错误}
            else begin    {如果没有报错}
                   i := position(id);    {记录当前符号在符号表中的位置}
                   if i = 0    {如果没有找到}
                   then error(11)    {报11号错误}
                   else    {如果找到了}
                     with table[i] do    {对第i个表项做如下操作}
                       if kind = prosedure    {如果该表项的种类为过程}
                       then gen(cal,lev-level,adr)    {生成cal代码用来实现call操作}
                       else error(15);    {如果种类不为过程类型,报15号错误}
                   getsym    {获取下一个sym类型}
                 end
          end
     else if sym = ifsym    {如果读到的符号是if关键字}
          then begin
                 getsym;    {获取下一个sym类型}
                 condition([thensym,dosym]+fsys);    {if后面跟的应该是条件语句,调用条件分析过程}
                 if sym = thensym    {如果条件语句后面跟的是then关键字的话}
                 then getsym    {获取下一个sym类型}
                 else error(16);    {如果条件后面接的不是then,报16号错误}
                 cx1 := cx;    {记录当前的生成代码位置}
                 gen(jpc,0,0);    {生成条件跳转指令,跳转位置暂填0}
                 statement(fsys);    {分析then语句后面的语句}
                 code[cx1].a := cx    {将之前记录的代码的位移地址改写到现在的生成代码位置(参考instru
```

```
      end
else if sym = beginsym    {如果读到了begin关键字}
      then begin
             getsym;    {获取下一个sym类型}
             statement([semicolon,endsym]+fsys); {begin后面默认接语句,递归下降分析}
             while sym in ([semicolon]+statbegsys) do    {在分析的过程中}
               begin
                 if sym = semicolon    {如果当前的符号是分好}
                 then getsym    {获取下一个sym类型}
                 else error(10);    {否则报10号错误}
                 statement([semicolon,endsym]+fsys)    {继续分析}
               end;
             if sym = endsym    {如果读到了end关键字}
             then getsym    {获取下一个sym类型}
             else error(17)    {报17号错误}
           end
      else if sym = whilesym    {如果读到了while关键字}
           then begin
                  cx1 := cx;    {记录当前生成代码的行数指针}
                  getsym;    {获取下一个sym类型}
                  condition([dosym]+fsys);    {因为while后需要添加循环条件,因此调用条件语
                  cx2 := cx;    {记录在分析完条件之后的生成代码的位置,也是do开始的位置}
                  gen(jpc,0,0);    {生成一个条件跳转指令,但是跳转位置(a)置零}
                  if sym = dosym    {条件后应该接do关键字}
                  then getsym    {获取下一个sym类型}
                  else error(18);    {如果没接do,报18号错误}
                  statement(fsys);    {分析处理循环节中的语句}
                  gen(jmp,0,cx1);        {生成跳转到cx1的地址,既是重新判断一遍当前条件是否
                  code[cx2].a := cx    {给之前生成的跳转指令设定跳转的位置为当前位置}
                end
       else if sym = readsym    {如果读到的符号是read关键字}
            then begin
                   getsym;    {获取下一个sym类型}
                   if sym = lparen    {read的后面应该接左括号}
                   then
                     repeat    {循环开始}
                       getsym;    {获取下一个sym类型}
                       if sym = ident    {如果第一个sym标识符}
                       then begin
                              i := position(id);    {记录当前符号在符号表中的位置}
                              if i = 0    {如果i为0,说明符号表中没有找到id对应的符号}
                              then error(11)    {报11号错误}
                              else if table[i].kind <> variable {如果找到了,但该符号的类
                                   then begin
                                          error(12);    {报12号错误,不能像常量和过程赋值
                                          i := 0    {将i置零}
                                        end
                                   else with table[i] do    {如果是变量类型}
                                         gen(red,lev-level,adr)    {生成一条red指令,读取
                            end
                       else error(4);    {如果左括号后面跟的不是标识符,报4号错误}
```

```
                        getsym;     {获取下一个sym类型}
                      until sym <> comma     {知道现在的符号不是都好,循环结束}
                    else error(40);     {如果read后面跟的不是左括号,报40号错误}
                    if sym <> rparen     {如果上述内容之后接的不是右括号}
                    then error(22);     {报22号错误}
                    getsym     {获取下一个sym类型}
                  end
            else if sym = writesym     {如果读到的符号是write关键字}
                 then begin
                        getsym;     {获取下一个sym类型}
                      if sym = lparen     {默认write右边应该加一个左括号}
                      then begin
                             repeat     {循环开始}
                               getsym;     {获取下一个sym类型}
                               expression([rparen,comma]+fsys);     {分析括号中的表达式}
                               gen(wrt,0,0);     {生成一个wrt海曙，用来输出内容}
                             until sym <> comma;     {知道读取到的sym不是逗号}
                             if sym <> rparen     {如果内容结束没有右括号}
                             then error(22);     {报22号错误}
                             getsym     {获取下一个sym类型}
                           end
                      else error(40)     {如果write后面没有跟左括号}
                    end;
  test(fsys,[],19)     {测试当前字符是否合法,如果没有出现在fsys中,报19号错}
end; { statement }
```

表达式处理

表达式应该是由正负号或无符号开头、由若干个项以加减号连接而成。而项是由若干个因子以乘除号连接而成，因子则可能是一个标识符或一个数字，或是一个以括号括起来的子表达式。根据这样的结构，构造出相应的过程，递归调用就完成了表达式的处理。

```
procedure expression( fsys: symset);    {处理表达式的过程}
      var addop : symbol;    {定义参数}
        procedure term( fsys : symset);  {处理项的过程}
          var mulop: symbol ;    {定义参数}
          procedure factor( fsys : symset );    {处理因子的处理程序}
            var i : integer;    {定义参数}
            begin
              test( facbegsys, fsys, 24 );    {测试单词的合法性,判别当前sym是否在facbegsys中,后者在
              while sym in facbegsys do    {循环处理因子}
                begin
                  if sym = ident    {如果识别到标识符}
                  then begin
                         i := position(id);    {查表,记录其在符号表中的位置,保存至i}
                         if i= 0    {如果i为0,表示没查到}
                         then error(11)    {报11号错误}
                         else
                           with table[i] do    {对第i个表项的内容}
                             case kind of        {按照表项的类型执行不同的操作}
                               constant : gen(lit,0,val);    {如果是常量类型,生成lit指令,操作数为0
                               variable : gen(lod,lev-level,adr);    {如果是变量类型,生成lod指令,
                               prosedure: error(21)    {如果因子处理中识别到了过程标识符,报21号错误
                             end;
                       getsym    {获取下一个sym类型}
                     end
                  else if sym = number    {如果识别到数字}
                       then begin
                            if num > amax    {判别数字是否超过规定上限}
                            then begin
                                   error(30);    {超过上限,报30号错误}
                                   num := 0    {将数字重置为0}
                                 end;
                            gen(lit,0,num);    {生成lit指令,将num的值放到栈顶}
                            getsym    {获取下一个sym类型}
                            end
                       else if sym = lparen    {如果识别到左括号}
                            then begin
                                 getsym;    {获取下一个sym类型}
                                 expression([rparen]+fsys);    {调用表达式的过程来处理,递归下降子程
                                 if sym = rparen    {如果识别到右括号}
                                 then getsym    {获取下一个sym类型}
                                 else error(22)    {报22号错误}
                               end;
                  test(fsys,[lparen],23)    {测试结合是否在fsys中,若不是,抛出23号错误}
                end
          end; { factor }
        begin { procedure term( fsys : symset);
                var mulop: symbol ;    }    {项的分析过程开始}
          factor( fsys+[times,slash]);    {项的第一个符号应该是因子,调用因子分析程序}
          while sym in [times,slash] do    {如果因子后面是乘/除号}
            begin
```

```
        mulop := sym;    {使用mulop保存当前的运算符}
        getsym;    {获取下一个sym类型}
        factor( fsys+[times,slash] );    {调用因子分析程序分析运算符后的因子}
        if mulop = times    {如果运算符是称号}
        then gen( opr,0,4 )    {生成opr指令,乘法指令}
        else gen( opr,0,5)    {生成opr指令,除法指令}
      end
  end; { term }
begin { procedure expression( fsys: symset);
        var addop : symbol; }    {表达式的分析过程开始}
  if sym in [plus, minus]    {如果表达式的第一个符号是+/-符号}
  then begin
         addop := sym;    {保存当前符号}
         getsym;    {获取下一个sym类型}
         term( fsys+[plus,minus]);    {正负号后面接项,调用项的分析过程}
         if addop = minus    {如果符号开头}
         then gen(opr,0,1)    {生成opr指令,完成取反运算}
       end
  else term( fsys+[plus,minus]);    {如果不是符号开头,直接调用项的分析过程}
  while sym in [plus,minus] do    {向后面可以接若干个term,使用操作符+-相连,因此此处用while}
    begin
      addop := sym;    {记录运算符类型}
      getsym;    {获取下一个sym类型}
      term( fsys+[plus,minus] );    {调用项的分析过程}
      if addop = plus    {如果是加号}
      then gen( opr,0,2)    {生成opr指令,完成加法运算}
      else gen( opr,0,3)    {否则生成减法指令}
    end
end; { expression }
```

条件处理

首先判断是否为一元逻辑表达式：判奇偶。如果是，则通过调用表达式处理过程分析计算表达式的值，然后生成判奇指令。如果不是，则肯定是二元逻辑运算符，通过调用表达式处理过程依次分析运算符左右两部分的值，放在栈顶的两个空间中，然后依不同的逻辑运算符，生成相应的逻辑判断指令，直接生成代码。

```
procedure condition( fsys : symset );     {条件处理过程}
      var relop : symbol;    {临时变量}
      begin
        if sym = oddsym    {如果当天符号是odd运算符}
        then begin
               getsym;    {获取下一个sym类型}
               expression(fsys);    {调用表达式分析过程}
               gen(opr,0,6)    {生成opr6号指令,完成奇偶判断运算}
             end
        else begin
             expression( [eql,neq,lss,gtr,leq,geq]+fsys);    {调用表达式分析过程对表达式进行计算}
             if not( sym in [eql,neq,lss,leq,gtr,geq])    {如果存在集合之外的符号}
               then error(20)    {报20号错误}
               else begin
                      relop := sym;    {记录当前符号类型}
                      getsym;    {获取下一个sym类型}
                      expression(fsys);    {调用表达式分析过程对表达式进行分析}
                      case relop of    {根据当前符号类型不同完成不同的操作}
                        eql : gen(opr,0,8);    {如果是等号,生成opr8号指令,判断是否相等}
                        neq : gen(opr,0,9);    {如果是不等号,生成opr9号指令,判断是否不等}
                        lss : gen(opr,0,10);    {如果是小于号,生成opr10号指令,判断是否小于}
                        geq : gen(opr,0,11);    {如果是大于等于号,生成opr11号指令,判断是否大于等于}
                        gtr : gen(opr,0,12);    {如果是大于号,生成opr12号指令,判断是否大于}
                        leq : gen(opr,0,13);    {如果是小于等于号,生成opr13号指令,判断是否小于等于}
                      end
                    end
             end
      end; { condition }
```

错误处理

分为判断单词合法性和直接输出错误信息。判断单词合法性则遇到不合法的单词会选择跳过，直到读到合法的单词

```
procedure error( n : integer );  {错误处理程序}
  begin
    writeln( '****', ' ':cc-1, '^', n:2 );     {报错提示信息，'^'指向出错位置，并提示错误类型}
    err := err+1 {错误次数+1}
  end; { error }

procedure test( s1,s2 :symset; n: integer );     {测试当前字符合法性过程,用于错误语法处理,若不合法则跳
  begin
    if not ( sym in s1 )    {如果当前符号不在s1中}
    then begin
           error(n);    {报n号错误}
           s1 := s1+s2;    {将s1赋值为s1和s2的集合}
           while not( sym in s1) do    {这个while的本质是pass掉所有不合法的符号,以恢复语法分析工作}
             getsym    {获得下一个标识符}
           end
  end; { test }
```

目标代码生成

Pcode代码形式为命令+两个操作数，同时需要统计一下行数信息

```
procedure gen( x: fct; y,z : integer );    {目标代码生成过程,x表示PCODE指令,y,z是指令的两个操作数}
  begin
    if cx > cxmax    {如果当前生成代码的行数cx大于允许的最大长度cxmax}
    then begin
           writeln('program too long');    {输出报错信息}
           close(fin);    {关闭文件}
           exit    {退出程序}
         end;
    with code[cx] do    {如果没有超出,对目标代码cx}
      begin
        f := x;    {令其f为x}
        l := y;    {令其l为y}
        a := z    {令其a为z}    {这三句对应着code身为instruction类型的三个属性}
      end;
    cx := cx+1    {将当前代码行数之计数加一}
  end; { gen }
```

符号表管理

对于传入的符号记录了其名字、类型（过程、变量、常数）。对于变量会记录当前层次以及对应的偏移量，对于过程记录乘此，常数会做正确性检验

```
procedure enter( k : objecttyp );     {将对象插入到符号表中}
    begin  { enter object into table }
      tx := tx+1;    {符号表序号加一,指向一个空表项}
      with table[tx] do    {改变tx序号对应表的内容}
        begin
          name := id;    {name记录object k的id,从getsym获得}
          kind := k;    {kind记录k的类型,为传入参数}
          case k of    {根据类型不同会进行不同的操作}
            constant : begin    {对常量}
                       if num > amax    {如果常量的数值大于约定的最大值}
                       then begin
                             error(30);    {报30号错误}
                             num := 0    {将常量置零}
                            end;
                       val := num    {val保存该常量的值,结合上句可以看出,如果超过限制则保存0}
                     end;
            variable : begin    {对变量}
                       level := lev;    {记录所属层次}
                       adr := dx;    {记录变量在当前层中的偏移量}
                       dx := dx+1    {偏移量+1,位下一次插入做准备}
                     end;
            prosedure: level := lev;    {对过程,记录所属层次}
          end
        end
    end; { enter }
```

编译器总体设计

本人设计的编译器中应该包含词法分析、语法分析、语义分析、中间代码生成、代码优化、错误处理等模块。通过对源程序的分析，生成目标代码，然后通过目标代码生成可执行程序。其中词法分析和语法分析过程中包括着错误处理和符号表的维护。随后得到结果进行语义分析并生成中间代码。利用生成的中间代码实现了代码优化（实现的代码优化为乘法优化，除法优化，临时寄存器分配，常量扩散），最后生成可用的目标代码

词法分析设计

需求分析

- 对于读入的字符串需要保留原样以便输出。并且设计方便打开和关闭词法分析的输出的开关。
- 需要为之后的语法分析和错误处理预留接口
- 记录单词的类别和单词的值，方便后续阶段调用。

设计

将词法分析封装为Lexer类，并类别码定义为enum，方便后续调用。
在Lexer构造函数中，并且初始化map容器，将读入的字符串单词名称对应到相应的枚举类型类别码。
读入时，每次读入单个字符，利用switch-case对其进行判断，识别出之后直接将其输出。

组织架构

```
.
|- include
|  |- lexer.h
|  |- PRELOAD.h
|-lexer.cpp
|-main.cpp
```

具体设计

预先加载类

Preload.h文件中的内容，对于一些常量的定义，全局需要使用的变量或结构体，后续需要用作优化宏定义的开关。

```
static const int identifierNum = 40;

    using namespace std;

    enum identifierType {
        IDENFR, INTCON, STRCON, MAINTK, CONSTTK, INTTK, BREAKTK, CONTINUETK, IFTK, ELSETK,
        NOT, AND, OR, WHILETK, GETINTTK, PRINTFTK, RETURNTK, PLUS, MINU, VOIDTK,
        MULT, DIV, MOD, LSS, LEQ, GRE, GEQ, EQL, NEQ,
        ASSIGN, SEMICN, COMMA, LPARENT, RPARENT, LBRACK, RBRACK, LBRACE, RBRACE
    };

    static const char *pairName[identifierNum] = {
            "IDENFR", "INTCON", "STRCON", "MAINTK", "CONSTTK", "INTTK", "BREAKTK", "CONTINUETK",
            "NOT", "AND", "OR", "WHILETK", "GETINTTK", "PRINTFTK", "RETURNTK", "PLUS", "MINU", "
            "MULT", "DIV", "MOD", "LSS", "LEQ", "GRE", "GEQ", "EQL", "NEQ",
            "ASSIGN", "SEMICN", "COMMA", "LPARENT", "RPARENT", "LBRACK", "RBRACK", "LBRACE", "RE
    };
```

词法分析类

Lexer类中的内容，对于词法分析的具体实现，包括对于单词的识别，对于单词的输出，对于单词的类别码和单词的值的记录。此外将整个类包在了一个命名空间里面了，有利于封装。

```
class Lexer {
    private:
        map<string, identifierType> identifierMap;
        fstream input, output;
    public:
        Lexer(const char* inputFile, const char* outputFile);
        ~Lexer();
        void analysis();
    };

    bool isLetter(char ch); //判断是否为字母

    bool isSymbol(char ch);//判断是否为可识别的符号

    bool isDigit(char ch);//判断是否为数字

    bool isBlank(char ch);//判断是否为空白字符
```

后续设计修改

- 后续我采用的设计将翻译得到成分全部放进语法分析和语义分析。因此用一个结构体将输出结果封装了起来，方便后续调用，在使用的时候注意 `extern` 的用法，若用 `static` 修饰，会导致只能本文件访问，其他文件不能访问同一个值。。

```
struct Identifier{
    string name;
    enum identifierType type;
};

extern vector<Identifier> identifierList;
extern map<string, identifierType> identifierMap;
```

- 为了满足后续预先读入的需求，设计了专门的读取函数。

语法分析设计

需求分析

- 基于词法分析器所识别单词，识别各类语法成分。
- 输出对应的语法成分名字，形如。

编码前设计

- 利用词法分析得到全部单词之后再放入语法分析器。
- 获取每语法的First集进行判断，利用swich-case+递归子程序的方式进行语法分析。，对于某些文法预读的方式进行语法分析。
- 建立好语法树，方便后续的错误处理以及代码生成。

组织架构

```
.
├── Lexer.cpp
├── Parser.cpp
├── include
│   ├── AST.h
│   ├── Lexer.h
│   ├── Parser.h
│   ├── PRELOAD.h
└── main.cpp
```

语法分析设计

```
namespace parser{
    using namespace std;
    using namespace AST;
    class Parser{
    private:
        fstream output; //output stream
        int index; // index of identifierList
        string currentIdentifier;
        identifierType currentType;
        int prefetchIndex; // index of identifierList prefetched
        identifierType prefetchType;
        shared_ptr<CompUnitAST> globalAST;
    public:
        explicit Parser(const char* outputFile);
        ~Parser();
        void analysis();
        int prefetch();
        int readNext();
        void parseProgram();
        shared_ptr<CompUnitAST> getGlobalAST();
        shared_ptr<CompUnitAST> parseCompUnit();
        shared_ptr<DeclAST> parseDecl();
        shared_ptr<FuncDefAST> parseFuncDef();
        shared_ptr<MainFuncDefAST> parseMainFuncDef();
        shared_ptr<ConstDeclAST> parseConstDecl();
        shared_ptr<VarDeclAST> parseVarDecl();
        shared_ptr<ConstDefAST> parseConstDef();
        shared_ptr<ConstInitValAST> parseConstInitVal();
        shared_ptr<ConstExpAST> parseConstExp();
        shared_ptr<BtypeAST> parseBtype();
        shared_ptr<VarDefAST> parseVarDef();
        shared_ptr<InitValAST> parseInitVal();
        shared_ptr<FuncTypeAST> parseFuncType();
        shared_ptr<FuncFParamsAST> parseFuncFParams();
        shared_ptr<BlockAST> parseBlock();
        shared_ptr<BlockItemAST> parseBlockItem();
        shared_ptr<FuncParamAST> parseFuncParam();
        shared_ptr<StmtAST> parseStmt();
        shared_ptr<ExpAST> parseExp();
        shared_ptr<CondAST> parseCond();
        shared_ptr<LValAST> parseLVal();
        shared_ptr<PrimaryExpAST> parsePrimaryExp();
        shared_ptr<NumberAST> parseNumber();
        shared_ptr<UnaryExpAST> parseUnaryExp();
        shared_ptr<UnaryOpAST> parseUnaryOp();
        shared_ptr<FuncRParamsAST> parseFuncRParams();
        shared_ptr<MulExpAST> parseMulExp();
        shared_ptr<AddExpAST> parseAddExp();
```

```
        shared_ptr<RelExpAST> parseRelExp();
        shared_ptr<EqExpAST> parseEqExp();
        shared_ptr<LAndExpAST> parseLAndExp();
        shared_ptr<LOrExpAST> parseLOrExp();
        friend void dealMulExp(vector<shared_ptr<UnaryExpAST> > &t, Parser& p);
        friend void dealAddExp(vector<shared_ptr<MulExpAST> > &t, Parser& p);
    };
}

#endif //COMPILER_PARSER_H
```

AST是待构建的语法树，Parser是语法分析器，Parser中包含了一个CompUnitAST的指针，用于构建语法树。readNext是用来读取词法分析得到的单词串，prefetch是预读，用来判断当前语法成分的类型，从而进行相应的语法分析。parseXXX是用来进行语法分析的具体函数，每个函数都会返回一个AST的指针，用于构建语法树。
指针采用的C++的智能指针中的 `shared_ptr` ，可以共享，仅仅增加引用数量的值。并且不用主动去释放该指针，仅当引用为0时会自动释放，有利于维护内存安全。

语法树构建

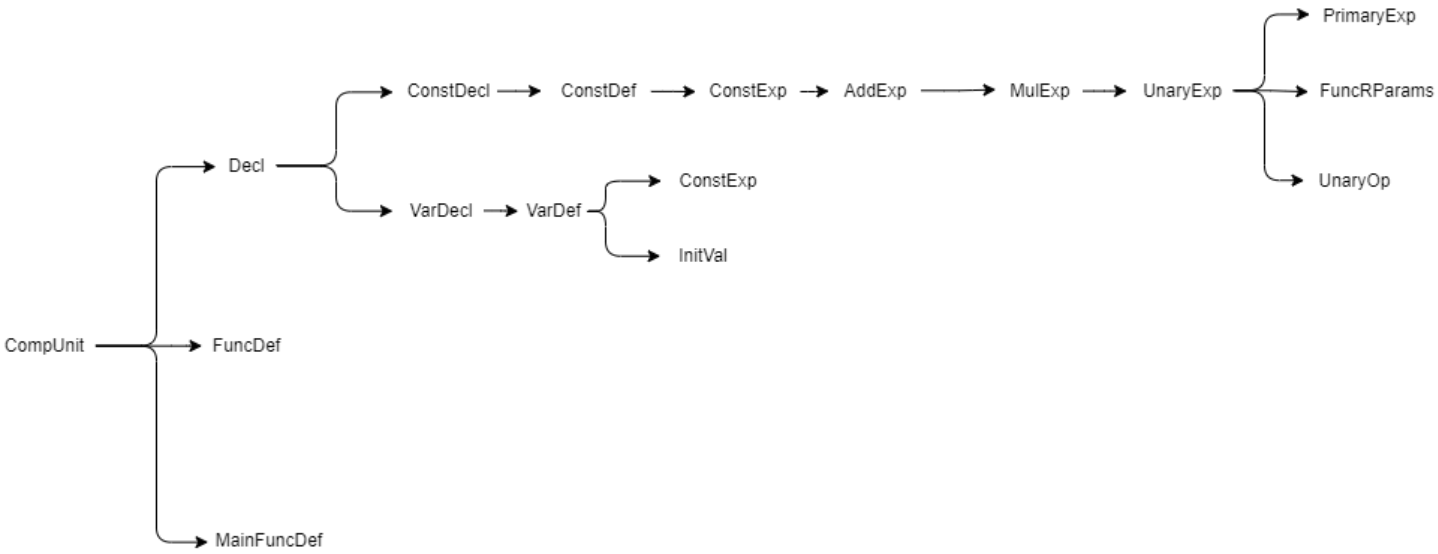
利用继承的面向特性来构建，在AST.h中存放了每个语法成分语法树的派生类，都继承自AST基类。均使用shared_ptr共享指针存放每个属性。当属性多于一个时，使用vector容器存储。同时在构造函数中，对每个传入的共享参数使用move方法来转接指针使用权。

```
//程序的主语法树
class CompUnitAST : public AST {
  private:
      vector<shared_ptr<DeclAST> > decls;
      vector<shared_ptr<FuncDefAST> > funcDefs;
      shared_ptr<MainFuncDefAST> mainfunc;
  public:
      CompUnitAST(vector<shared_ptr<DeclAST>> decls, vector<shared_ptr<FuncDefAST>> funcDefs,
                  shared_ptr<MainFuncDefAST> mainfunc) :
              decls(std::move(decls)), funcDefs(std::move(funcDefs)), mainfunc(std::move(mainfur

      vector<shared_ptr<DeclAST> > &getDecls() { return decls; }

      vector<shared_ptr<FuncDefAST> > &getFuncDefs() { return funcDefs; }

      shared_ptr<MainFuncDefAST> &getMainFuncDef() { return mainfunc; }
  };
```



读取和预读的处理

注意用变量来表示应该预读的下标，而不是直接使用 `readNext` ，因为 `readNext` 会改变下标，而预读的下标应该是不变的。注意在预读结束后应及时与当前下标同步，防止下一次预读出错。

```
int Parser::readNext() {
    /*
      * Step 1: Get the next token
      * Step 2: Print the related information
      * Step 3: Update the currentType and currentIdentifier
      */
    if (index < identifierList.size()) {
        currentIdentifier = identifierList[index].name;
        prefetchIndex = index + 1;
        if (index > 0) {
            output << pairName[identifierList[index - 1].type] << " " << identifierList[index -
        }
        currentType = identifierList[index++].type;
        return currentType;
    } else { // take attention to the end
        if (index > 0) {
            output << pairName[identifierList[index - 1].type] << " " << identifierList[index -
        }
    }
    currentType = END;
    return END;
}

int Parser::prefetch() {
    if (prefetchIndex < identifierList.size()) {
        prefetchType = identifierList[prefetchIndex++].type;
        return prefetchType;
    } else
        return END;
}
```

语法分析流程

采用递归下降法，顺着语法树往下找，直到叶子结点为止。对于First集有重合的非终结符采用预读的方式确定分支。将一个结点的子树遍历好之后，输出该结点对应的成分

```
shared_ptr<CompUnitAST> Parser::parseCompUnit() {
    vector<shared_ptr<DeclAST> > declList;
    vector<shared_ptr<FuncDefAST> > funcDefList;
    shared_ptr<MainFuncDefAST> mainFuncDef;
    while (true) {
        switch (currentType) {
            case END:
                break;
            case CONSTTK: {
                // const type
                shared_ptr<DeclAST> decl = parseDecl();
                declList.push_back(decl);
                break;
            }
            case VOIDTK: {
                // func type
                shared_ptr<FuncDefAST> funcDef = parseFuncDef();
                output << "<FuncDef>" << endl;
                funcDefList.push_back(funcDef);
                break;
            }
            case INTTK: {
                // var or func or main
                prefetch();
                switch (prefetchType) {
                    case MAINTK: {
                        // main func
                        mainFuncDef = parseMainFuncDef();
                        output << "<MainFuncDef>" << endl;
                        break;
                    }
                    case IDENFR :{
                        // fun or var
                        prefetch();
                        switch (prefetchType) {
                            case LPARENT: {
                                // func
                                shared_ptr<FuncDefAST> funcDef = parseFuncDef();
                                output << "<FuncDef>" << endl;
                                funcDefList.push_back(funcDef);
                                break;
                            }
                            case ASSIGN:
                            case LBRACK:
                            case COMMA:
                            case SEMICN: {
                                // var
                                shared_ptr<DeclAST> decl = parseDecl();
                                declList.push_back(decl);
                                break;
```

```
                        }
                        default:
                            break;
                    }
                }
                default:
                    break;
            }
        }
        default:
            break;
    }
    if (currentType == END) break;
    // cout << "In compUnit" << endl;
}
return make_shared<CompUnitAST>(std::move(declList), std::move(funcDefList), std::move(mainF
}
```

编码后修改

- 对应的语法分析过程建立了相应的符号表
- 对读取过程进行了一些小调整，方便错误处理的判断和跳过，因为题目要求能将全部错误都输出，所以不能直接退出
- 把一些重复的过程封装成友元函数，方便调用
- 记录单词信息的时候加入了行号，为错误处理准备
- 增加了控制词法分析和语法分析结果输出

错误处理设计

需求

- 对于代码能识别出给定的所有的语法和语义错误
- 输出错误行号和错误的类别码
- 控制词法分析和语法分析的错误。

编码前设计

- 语法错误进行词法分析时完成，如 ; 的缺失， [] , () 不匹配
- 在进行语义分析时，要维护语法树（AST），对于部分语义错误也可以在分析时输出，如使用了未定义的变量等
- 需要一个符号表，用于记录变量的定义，函数的定义，以及变量的作用域等。
- 需要实现一个错误类，用于记录错误信息，包括错误行号，错误类型，错误信息等

组织架构

```
.
├── Lexer.cpp
├── Parser.cpp
|── SymbolTable.cpp
|── ErrorProcesser.cpp
├── include
│   ├── AST.h
│   ├── Lexer.h
│   ├── Parser.h
│   ├── PRELOAD.h
│   ├── SymbolTable.h
│   └── ErrorProcesser.h
└── main.cpp
```

错误处理类设计

```
//
// Created by Leo on 2022/10/11.
//

namespace error {
    using namespace std;
    using namespace ast;

    extern SymbolTable symbolTable;
    extern vector <pair <int , string> >errors; // (line, type)

    class ErrorProcesser {
    private:
        fstream err;
        int currentDomain;
        string currentFunction;
        stack<identifierType>currentType;
        identifierType returnType;
        stack<bool> isLoop;
        shared_ptr<CompUnitAST> &compUnitAST;
    public:
        ErrorProcesser(const char *err, shared_ptr<CompUnitAST> &compUnitAST);
        ~ErrorProcesser();
        void analysis();
        void print();
        void processDecl(shared_ptr<DeclAST> &declAST);
        void processConstDecl(shared_ptr<ConstDeclAST> &constDeclAST);
        void processConstDef(shared_ptr<ConstDefAST> &constDefAST);
        void processVarDecl(shared_ptr<VarDeclAST> &varDeclAST);
        void processVarDef(shared_ptr<VarDefAST> &varDefAST);
        void processFuncDef(shared_ptr<FuncDefAST> &funcDefAST);
        void processFuncFParam(shared_ptr<FuncParamAST> &funcParamAST, vector<VarSymbol> &params
        void processBlock(shared_ptr<BlockAST> &blockAST, bool isFunction);
        void processBlockItem(shared_ptr<BlockItemAST> &blockItem);
        void processStmt(shared_ptr<StmtAST> &stmtAST);
        int processExp(shared_ptr<ExpAST> &expAST);
        void processMainDef(shared_ptr<MainFuncDefAST> &mainFuncDefAST);
        int processLVal(shared_ptr<LValAST> &lValAST, bool &con);
        bool processPrint(string &formmat, int num);
        void processCond(shared_ptr<CondAST> &condAST);
        void processConstExp(shared_ptr<ConstExpAST> &constExpAST);
        void processConstInitVal(shared_ptr<ConstInitValAST> &constInitValAST);
        void processInitVal(shared_ptr<InitValAST> &initValAST);
        void processRelExp(shared_ptr<RelExpAST> &relExpAST);
        int processAddExp(shared_ptr<AddExpAST> &addExpAST);
        int processMulExp(shared_ptr<MulExpAST> &mulExpAST);
        int processUnaryExp(shared_ptr<UnaryExpAST> &unaryExpAST);
        int processPrimaryExp(shared_ptr<PrimaryExpAST> &primaryExpAST);
```

```
        void processEqExp(shared_ptr<EqExpAST> &eqExpAST);
        void processFuncRParams(shared_ptr<FuncRParamsAST> &funcRParamsAST, vector<int>& dimensi
        void processLAndExp(shared_ptr<LAndExpAST> &lAndExpAST);
        void processLOrExp(shared_ptr<LOrExpAST> &lOrpExpAST);
    };

}

#endif //COMPILER_ERRORPROCESSER_H
```

currentDomain表示当前的作用域，currentFunction表示当前的函数，currentType表示当前的类型,，isLoop表示当前是否在循环中，compUnitAST表示语法树的根节点，errors表示错误信息的集合方便输出。其余函数为各类错误的具体处理。

符号表类设计

```
class Symbol {
private:
    string name;
    identifierType type;
public:
    Symbol(string name, identifierType type) : name(std::move(name)), type(type) {}

    string getName() const { return name; }

    identifierType getType() const { return type; }
};

class VarSymbol : public Symbol {
private:
    int dimension; //  0 for scalar, 1 for array, 2 for matrix
    int domain;
public:
    VarSymbol(string name, identifierType type, int dimension, int domain) : Symbol(std::move(na
                                                                                    dimer
                                                                                    domai

    int getDimension() const { return dimension; }

    int getDomain() const { return domain; }
};

class ConstSymbol : public Symbol {
private:
    int domain;
    int dimension;
public:
    ConstSymbol(string name, identifierType type, int domain, int dimension) : Symbol(std::move(

                                                                                    dom
                                                                                    dim

    int getDomain() const { return domain; }

    int getDimension() const { return dimension; }
};

class FuncSymbol : public Symbol {
private:
    vector<VarSymbol> parameters;
public:
    FuncSymbol(string name, identifierType type, vector<VarSymbol> parameters) : Symbol(std::mov

                                                                                    p
```

```
    int getParameterNum() const { return (int) parameters.size(); }

    vector<VarSymbol> &getParameters() { return parameters; }
};

class SymbolTable {
private:
    vector<VarSymbol> varTable;
    vector<ConstSymbol> constTable;
    vector<FuncSymbol> funcTable;
public:
    vector<VarSymbol> &getVarTable() { return varTable; }

    vector<ConstSymbol> &getConstTable() { return constTable; }

    vector<FuncSymbol> &getFuncTable() { return funcTable; }
};
```

一个符号表的基类和变量符号表、常量符号表和函数符号表的派生类。函数FuncSym类是记录了函数的名字、返回值、参数个数、参数变量信息。常量和变量类都记录了名字、类型、作用域、数组维度（不是数组，记为0）。在处理符号表时，每进入一个块，作用域层次加一，在退出该块时，会将该作用域的变量常量全部删除，作用域层次减一。

错误处理流程

```
void ErrorProcesser::analysis() {
      vector<shared_ptr<DeclAST> > &decls = compUnitAST->getDecls();
      for (auto &decl: decls) {
          processDecl(decl);
      }
      vector<shared_ptr<FuncDefAST> > &funcDefs = compUnitAST->getFuncDefs();
      for (auto &funcDef: funcDefs) {
          processFuncDef(funcDef);
      }
      shared_ptr<MainFuncDefAST> &mainFuncDef = compUnitAST->getMainFuncDef();
      processMainDef(mainFuncDef);
  }
```

同样地递归下降法，顺着语法树一层层的往下处理，查看是否有不符合的错误。

编码后修改

- 先前对于表达式和赋值语句的判定是依靠；来区分的，但现在有可能出现；缺失的情况，因此更改读取方式为更改记录当前的index值，先调用函数进行读取标识符，然后判断之后的单词是否是等号，然后将index恢复到原来的位置。再接着执行和之前同样的内容即可。
- 将符号表的操作提取出来封装在符号表类中，方便调用

代码生成设计

需求分析

- 对于代码生成中间代码
- 依靠中间代码生成目标代码

编码前设计

- 一个中间代码生成模块，采用的是四元式形式，方便后续优化。
- 一个中间代码生成器模块，通过语法树生成中间代码，因此对应的各种语句的升级机制都要完善
- 代码生成模块，原始版本是不用任何寄存器的指派和控制，即对于每一个操作数把其从内存中取出，得到结果再存回内存当中。

符号表的修改

代码生成中之前错误处理所使用的符号表过于简陋，为了避免直接修改造成之前一些不容易出现的bug，就新设计了一个专门用于中间代码生成的符号表。

```
class VarIRCode {
private:
    string name;
    SymbolType type;
    int dimension;
    int domain;
    vector<int> spectrum;
    //single var
    shared_ptr<Object> value;
    //array
    vector<shared_ptr<Object> > values;
    int offset;
    int space;
    bool isArrayPara;
    bool isConst;
    bool isUse;

public:
    VarIRCode(string name, int dimension, bool isConst, vector<int> &spectrum, SymbolType type,

    VarIRCode(string name, int dimension, bool isConst, shared_ptr<Object> &value, SymbolType ty

    VarIRCode(string name, int dimension, bool isConst, SymbolType type, int domain);

    void setValue(int _value);

    void setValue(shared_ptr<Object> _value);

    void setSpaceInfo(int _offset, int _space);

    int getDimension() const {return  dimension;}

    vector<shared_ptr<Object> >& getValues() {return values;}

    vector<int>& getSpectrum() {return spectrum;}

    string getName() const {return name;}

    shared_ptr<Object>& getValue() {return value;}

    bool getIsConst() const {return isConst;}

    void setDefinedValue(bool defined);

    void setOffset(int _offset);

    int getDomain() const {return domain;}

    int getOffset() const {return offset;}
```

```
    void setIsArrayPara(bool arrayPara);

    bool getIsArrayPara() const {return isArrayPara;}

    void setIsUse(bool use);

    bool getIsUse() const {return isUse;}


    bool operator < (const VarIRCode &varIRCode) const {
      return this->offset < varIRCode.offset;
    };
};
```

变量表增加了很多描述变量性质的属性，以及对应的get和set方法。offset表示变量的偏移量，方便储存在内存当中；space表示变量所占的空间大小，isConst表示是否是常量，isUse表示是否被使用，isArrayPara用来描述，isArrayPara表示是否是数组参数，spectrum表示数组的维度，values表示数组的值，value表示单个变量的值。

```
class FuncIRCode {
private:
    string name;
    SymbolType returnType;
    vector<shared_ptr<VarIRCode> > exps;
    int offset;
public:
    FuncIRCode(string name, SymbolType returnType, vector<shared_ptr<VarIRCode> > &exps) :
            name(std::move(name)), returnType(returnType), exps(exps) {}

    FuncIRCode(string name, SymbolType returnType) : name(std::move(name)), returnType(returnTyp

    void setOffset(int offset_t) { this->offset = offset_t; }

    string getName() const { return name; }

    SymbolType getReturnType() const { return returnType; }

    int getOffset() const { return offset; }
    vector<shared_ptr<VarIRCode> > &getExps() { return exps; }
};
```

函数表也类似，需要用offset记录偏移量，exps存放其对于的参数，方便后续的调用。

Object类

因为四元式的任何一项可能可以是任何类别，因此用类似Java中Object的概念来统一表示。

```
class Object {
    // generic class for all kinds of objects
private:
    /*
      * 0 for null
      * 1 for type:int, void
      * 2 for name of label
      * 3 for array: name[index] (index can be anything)
      * 4 for array, also but index is int32
      * 5 for num
      * 6 or long long num
      */
    int category;
    string name;
    int numericIndex;
    int num;
    long long longNum;
    shared_ptr<Object> index;
    SymbolType type;

    shared_ptr<VarIRCode> var;
    shared_ptr<FuncIRCode> func;

public:
    Object() : category(0) {}

    explicit Object(SymbolType type) : category(1), type(type) {}

    explicit Object(string name) : category(2), name(std::move(name)) {}

    Object(string name, shared_ptr<VarIRCode> &var) : category(2), name(std::move(name)), var(va

    Object(string name, shared_ptr<FuncIRCode> &func) : category(2), name(std::move(name)), func

    Object(string name, shared_ptr<Object> index) : category(3), name(std::move(name)), index(st

    Object(string name, int numericIndex) : category(4), name(std::move(name)), numericIndex(num

    Object(string name, shared_ptr<Object> &obj, shared_ptr<VarIRCode> &var) {}

    Object(string name, int numericIndex, shared_ptr<VarIRCode> &var) : category(4), name(std::m

    explicit Object(int num) : category(5), num(num) {}

    explicit Object(long long longnum, string _) : category(6), longNum(longnum) {}

    Object(Object &t) {
        this->category = t.category;
        this->name = t.name;
        this->numericIndex = t.numericIndex;
```

```
        this->num = t.num;
        this->index = t.index;
        this->type = t.type;
        this->var = t.var;
        this->func = t.func;
    }

    string Print();

    shared_ptr<FuncIRCode> &getFunc() { return func; }

    shared_ptr<VarIRCode> &getVar() { return var; }

    string getName() const { return name; }

    int getNumericIndex() const {return numericIndex; }

    int getNum() const {return num; }

    int getCategory() const {return category; }

    long long getLongNum() const {return longNum; }

    void setNum(int number);

    void setCategory(int _category);

    void setVar(shared_ptr<VarIRCode> &_var);

    shared_ptr<Object> &getIndex() { return index; }
};
```

用识别码进行区分，方便后续的处理。其中，category

- 为0表示空，
- 为1表示类型，
- 为2表示标签，
- 为3表示数组，
- 为4表示数组，但是索引为整数，
- 为5表示整数，
- 为6表示长整数。

name表示名字，numericIndex表示数组的索引，num表示整数，longNum表示长整数，index表示数组的索引，type表示类型，var表示变量，func表示函数。

最终的四元式可以以Object[3]来，Object[0]为左值，Object[1]为操作符，Object[2]表示操作数1，Object[3]表示操作数2。

中间代码生成过程

大致和语法过程类似，按照推荐的中间代码格式从语法树的树根开始逐层分析，生成中间代码。对于变量，在变量表中找到对应的变量，取出并生成相应的Object类来生成四元式

注意的区别遇到表达式生成临时变量和while以及if生成标签上

```
shared_ptr<Object> IRCodeGenerator::generateTempVarT() {
    static int count = 0;
    ++count;
    string name = "tmpVar_" + to_string(count);
    shared_ptr<Object> object = make_shared<Object>(name);
    return object;
}
string IRCodeGenerator::generateNewLabel(SymbolType type) {
    int count = 0;
    static int labelWhile = 0;
    static int labelIf = 0;
    if (type == WHILEST) {
        count = labelWhile++;
    } else if (type == IFST) {
        count = labelIf++;
    }
    string label = symbolName[type] + to_string(count);
    return label;
}
```

在变量生成后要对其属性进行设置，如偏移量、所需空间等，并将其插入到变量表中。这二者生成主要在于不要重复生成，如果已经生成过了，就不要再生成了。

代码生成相关

对于大部分指令，按照对应的中间代码生成相应的即可，主要是函数的处理需要多加设计。以及因为本次是直接利用内存沟通，所以需要load和save相关的设计

load&save

```
void MipsGenerator::load(shared_ptr<Object> &obj1, string reg) {
    if (obj1->getCategory() == 5) {
        output << "li " << reg << ", " << obj1->getNum() << endl;
        return;
    }
    shared_ptr<VarIRCode> var = obj1->getVar();
    if (var->getDimension() == 0) {
        int address = var->getOffset() * 4;
        string pointeer_reg = (var->getDomain() == 0) ? "$gp" : "$sp";
        output << "lw " << reg << ", " << address << "(" << pointeer_reg << ")" << endl;

    } else {
        int address = var->getOffset() * 4;
        string pointeer_reg = (var->getDomain() == 0) ? "$gp" : "$sp";
        if (obj1->getCategory() == 3) {
            // a[n]
            shared_ptr<Object> &index = obj1->getIndex();
            load(index, "$t2");
            // t2 for index
            if (var->getIsArrayPara()) {
                output << "lw $t1, " << (var->getOffset()) * 4 << "($sp)" << endl;
            } else {
                output << "addi $t1, " << pointeer_reg << ", " << address << endl;
            }
            output << "sll $t2, $t2, 2" << endl;
            output << "subu $t2, $t1, $t2" << endl;
            output << "lw " << reg << ", 0($t2)" << endl;
        } else if (obj1->getCategory() == 4) {
            // a[10]
            int numericIndex = obj1->getNumericIndex();
            if (var->getIsArrayPara()) {
                output << "lw $t1, " << (var->getOffset()) * 4 << "($sp)" << endl;
                output << "lw " << reg << ", " << -numericIndex * 4 << "($t1)" << endl;
            } else {
                address -= numericIndex * 4;
                output << "lw " << reg << ", " << address << "(" << pointeer_reg << ")" << endl;
            }
        }
    }
}

void MipsGenerator::save(shared_ptr<Object> &obj, string reg) {
    shared_ptr<VarIRCode> var = obj->getVar();
    if (var->getDimension() == 0) {
        int address = var->getOffset() * 4;
        string pointeer_reg = var->getDomain() == 0 ? "$gp" : "$sp";
        output << "sw " << reg << ", " << address << "(" << pointeer_reg << ")" << endl;
    } else {
```

```
        int address = var->getOffset() * 4;
        string pointeer_reg = (var->getDomain() == 0) ? "$gp" : "$sp";
        if (obj->getCategory() == 3) {
            // a[n]
            shared_ptr<Object> &index = obj->getIndex();
            load(index, "$t2");
            // t2 for index
            if (var->getIsArrayPara()) {
                output << "lw $t1, " << (var->getOffset()) * 4 << "($sp)" << endl;
            } else {
                output << "addi $t1, " << pointeer_reg << ", " << address << endl;
            }
            output << "sll $t2, $t2, 2" << endl;
            output << "subu $t2, $t1, $t2" << endl;
            output << "sw " << reg << ", 0($t2)" << endl;
        } else if (obj->getCategory() == 4) {
            // a[10]
            int numericIndex = obj->getNumericIndex();
            if (var->getIsArrayPara()) {
                output << "lw $t1, " << (var->getOffset()) * 4 << "($sp)" << endl;
                output << "sw " << reg << ", " << -numericIndex * 4 << "($t1)" << endl;
            } else {
                address -= numericIndex * 4;
                output << "sw " << reg << ", " << address << "(" << pointeer_reg << ")" << endl;
            }
        }
    }
}
```

二者本质上一样的，都找到对应的变量的偏移量，计算出所对应的空间即可。

函数调用

函数调用则用了栈帧，按照mips函数的调用法则，每次调用函数时，都会在栈上分配一个栈帧，用来存储函数的参数和局部变量。利用 `$sp` 来完成变量的相应调用

```
case CallOT: {
shared_ptr<Object> object = ircode->getObj(0);
shared_ptr<FuncIRCode> func = object->getFunc();
vector<shared_ptr<VarIRCode> > &exps = func->getExps();
int num = (int) exps.size();
int offset = (4 * func->getOffset() + 8);
string name = func->getName();
output << "addi $s0, $sp, " << -offset << endl;
while (num > 0) {
    shared_ptr<Object> param = funcParams.top();
    // param is real, exps is set
    funcParams.pop();
    shared_ptr<VarIRCode> &var = param->getVar();
    if (var != nullptr and var->getDimension() not_eq 0
        and (var->getDimension() == exps[num - 1]->getDimension()
              or (var->getDimension() == 2 and exps[num - 1]->getDimension() == 1))) {
        int domain = var->getDomain();
        string t_reg = domain == 0 ? "$gp" : "$sp";
        if (var->getIsArrayPara()) {
            // the parameter used as the other function's parameter
            if (var->getDimension() == exps[num - 1]->getDimension()) {
                output << "lw $t0, " << var->getOffset() * 4 << "($sp)" << endl;
                output << "sw $t0, " << num * 4 << "($s0)" << endl;
            } else {
                // a is 2D and use as 1D, int fun(int a[][3]) {func(a[0]);}
                output << "lw $t0, " << var->getOffset() * 4 << "($sp)" << endl;
                if (param->getCategory() == 4) {
                    int index = param->getNumericIndex() * var->getSpectrum()[1] * 4;
                    output << "subi $t0, $t0, " << index << endl;
                } else {
                    load(param->getIndex(), "$t1");
                    output << "li $t2, " << var->getSpectrum()[1] * 4 << endl;
                    output << "mult $t1, $t2" << endl;
                    output << "mflo $t2" << endl;
                    output << "subu $t0, $t0, $t2" << endl;
                }
                output << "sw $t0, " << num * 4 << "($s0)" << endl;
            }
        } else if (exps[num - 1]->getDimension() == var->getDimension()) {
            exps[num - 1]->setOffset(var->getOffset());
            output << "li $t0, " << var->getOffset() * 4 << endl;
            output << "addu $t0, " << t_reg <<", $t0" << endl;
            output << "sw $t0, " << 4*num << "($s0)" << endl;
        } else if (exps[num -1]->getDimension() == 1 and var->getDimension() == 2) {
            if (param->getCategory() == 4) {
                exps[num-1]->setOffset(var->getOffset() - param->getNumericIndex() * var->getSpe
                output << "li $t0, " << exps[num-1]->getOffset() * 4 << endl;
                output << "addu $t0, " << t_reg << ", $t0" << endl;
                output << "sw $t0, " << 4*num << "($s0)" << endl;
            } else {
```

```
                load(param->getIndex(), "$t0");
                output << "li $t1, " << var->getSpectrum()[1] * 4 << endl;
                output << "mult $t0, $t1" << endl;
                output << "mflo $t2" << endl;
                output << "li $t1, " << var->getOffset() * 4 << endl;
                output << "subu $t0, $t1, $t2" << endl;
                output << "addu $t0, $t0, " << t_reg << endl;
                output << "sw $t0, " << 4*num << "($s0)" << endl;
            }
        }
    } else {
        load(param, "$t0");
        output << "sw $t0, " << 4 * num << "($s0)" << endl;
    }
    --num;
}
output << "sw $ra, ($sp)" << endl;
output << "sw $fp, -4($sp)" << endl;
output << "addi $sp, $sp, " << -offset << endl;
output << "addi $fp, $sp," << offset << endl;
output << "jal function_" << name << endl;
output << "addi $sp, $sp, " << offset << endl;
output << "lw $fp, -4($sp)" << endl;
output << "lw $ra, ($sp)" << endl;
break;
}
```

编码后修改

- 有大量的重复语段，应该提取出来单独封装为一个方法，或者是用友元函数的方式进行调用。（未能实现）
- 为了后续寄存器的指派和分配，对于变量是否是临时变量进行区分
- 为了后续的优化，需要增加一些优化开关(define实现)
- 为了寻找变量的方便，在等式左边Object增加一个属性表示指向变量的引用
- 增加mips的位运算指令以便以后乘除优化
- 为了方便寄存器优化，应该要划分基本块。这个较为容易，找到导致代码不顺序运行的操作符，分割代码块即可

```
void GenerateBlock::run() {
    vector<int> cutDots;
    bool flag = false;
    for (int i = 0; i < IRCodeList.size(); i++) {
        if (!flag) {
            if (IRCodeList[i]->getOp() == MainOT) {
                flag = true;
                setStart(i);
                continue;
            } else {
                continue;
            }
        }
       if (IRCodeList[i]->getOp() == JumpOT) {
            cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == FuncOT) {
            cutDots.push_back(i);
       } else if (IRCodeList[i]->getOp() == CallOT) {
            cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == ReturnOT) {
            cutDots.push_back(i+1);
       } else if (IRCodeList[i]->getOp() == LabelOT) {
            cutDots.push_back(i);
       }
    }
    sort(cutDots.begin(), cutDots.end());
    cutDots.erase(unique(cutDots.begin(), cutDots.end()), cutDots.end());
    int cnt = 0;
    while (cnt < cutDots.size()) {
        shared_ptr<BasicBlock> basicBlock;
        for (int i = cutDots[cnt]; i < (cnt == cutDots.size() - 1 ? IRCodeList.size() : cutDots[
            if (i == cutDots[cnt]) {
                basicBlock = make_shared<BasicBlock>("B" + to_string(cnt));
            }
            basicBlock->addCode(IRCodeList[i]);
        }
        basicBlocks.push_back(basicBlock);
        ++cnt;
    }
}
```

代码优化设计

需求

- 相关的优化
- 优化开关

编码前设计

- 优化开关利用define，ifdef, endif等指令来实现
- 实现乘除法优化。通过将模转换为加减乘除的组合来阻止div的使用
- 寄存器指派和分配

乘法优化

只针对一些特殊的情况, 也就是可以转换为位运算的情况来进行, 当乘数为

-
-
-
-

其方法就是对于传入整数的乘数，判断是否与2的幂数相关

```
bool isPowerOfTwo(int n) {
    return n && (!(n & (n - 1)));
}

bool judge(int n, int &a, int &b) {
    for (int i = 1; i <= (int) sqrt(n); ++i) {
        if (n % i == 0) {
            if (isPowerOfTwo(i) and isPowerOfTwo(n / i - 1)) {
                a = (int) log2(i);
                b = (int) log2(n / i - 1);
                return true;
            } else if (isPowerOfTwo(n / i) and isPowerOfTwo(i - 1)) {
                a = (int) log2(n / i);
                b = (int) log2(i - 1);
                return true;
            }
        }
    }
    return false;
}
```

若相关就转换为位运算和加减

除法运算

利用"Magic Number"的理论进行除法优化。

设代码a/b = c　　b是常量

设b*k = 2^n　　2的n次直接用移位运算即可

a *2^n/ (b*2 ^n) = c

a * b *k/b = c* 2^n

a *k >> n = c

因此对于每个除数需要获得其"Magic Number"

```
long long getMagicNumber(const BigInteger &divImm, int &leftBit, int &shPosition) {
      leftBit = 0;
      BigInteger twoBase(2);
      while (divImm > (twoBase ^ leftBit)) {
          leftBit++;
      }
      shPosition = leftBit;
//        cout << "leftBit: " << leftBit << endl;
      long long lowValue = ((twoBase ^ (leftBit + 32)) / divImm).toLongLong();
      long long highValue = (((twoBase ^ (leftBit + 32)) + (twoBase ^ (leftBit + 1))) / divImm).
//        cout << lowValue << " " << highValue << endl;
      while ((highValue >> 1) > (lowValue >> 1) and shPosition > 0) {
          highValue >>= 1;
          lowValue >>= 1;
          shPosition--;
      }
      return highValue;
  }
```

待寻找到对应的MagicNumber之后，观察x86的编译优化

```
mov eax,2E8BA2E9
imul ecx   //ecx是我们的a
sar edx,1
mov ecx,edx
shr ecx,1F
add edx,ecx
```

分析其过程可以知道

首先eax是由编译器生成的一个数，然后　　,得到的积低32位存储在eax中，高32位存储在edx中，如果积大于32位的话。

```
sar edx,1
```

则是直接取出高32位来看，相当于直接舍去了低32位，也就相当于先右移了32位。再右移1位，相当于我们得到的积总共向右移了33位

等 ， 也就是所谓的倒数了。

第5行的右移31位是为了得到结果的符号位，这里主要是看符号位是否为1，也就是是否为负数。如果是负数的话，符号位为1，最后的结果+1。
之所以要+1，这是因为负数算术右移跟负数除以2最后的结果其实是不同的，尽管常常认为右移1位就是除以2。
负数算术右移是向下取整的，换句话说负数, shr 的结果 + 1 = 负数除以2 的 结果 负数shr的结果+1 = 负数除以2的结果负数shr的结果+1=负数除以2的结果。所以最后要加1。

```
 if(objects[1]->getCategory() != 5 and objects[2]->getCategory() == 5) {
int dividend = objects[2]->getNum();
int result, shPosition, leftBit;
shared_ptr<IRCode> ircodeMagic, ircode1;
long long magicNumber = getMagicNumber(BigInteger(abs(objects[2]->getNum())), leftBit, shPositic
if (abs(objects[2]->getNum()) == 1) {
    if (objects[2]->getNum() == 1) {
        ircode = make_shared<IRCode>(AssignOT, objects);
    } else {
        shared_ptr<Object> objects1[3];
        objects1[0] = objects[0];
        objects1[2] = objects[1];
        objects1[1] = make_shared<Object>(0);
        ircode = make_shared<IRCode>(MINUSOT, objects1);
    }
}
  else {
    // objects1 for positive or negative
    shared_ptr<Object> objectsMagic[3], objects1[3];
    objectsMagic[0] = objects[0];
    objectsMagic[1] = make_shared<Object>(magicNumber, "LONG");
    objectsMagic[2] = objects[1];
    ircodeMagic = make_shared<IRCode>(MULTHOT, std::move(objectsMagic));
    IRCodeList.push_back(std::move(ircodeMagic));
    if (magicNumber >= INF) {
        shared_ptr<Object> objects2[3];
        objects2[0] = generateTempVarT();
        objects2[1] = objects[1];
        objects2[2] = make_shared<Object>(1LL << 32, "LONG");
        shared_ptr<VarIRCode> var_temp = make_shared<VarIRCode>(objects2[0]->getName(), 0, false
                                                                INTST,
                                                                currentDomain);
        varIndex += 1;
        if (currentDomain == 0) {
            global_offset += 1;
        }
        var_temp->setIsTemp(true);
        var_temp->setSpaceInfo(varIndex, 1);
        objects2[0]->setVar(var_temp);
        ircode1 = make_shared<IRCode>(MULTOT, objects2);
        IRCodeList.push_back(std::move(ircode1));
        objectsMagic[0] = objects[0];
        objectsMagic[1] = objects[0];
        objectsMagic[2] = objects2[0];
        ircodeMagic = make_shared<IRCode>(MINUSOT, std::move(objectsMagic));
        IRCodeList.push_back(std::move(ircodeMagic));
    }
    if (magicNumber >= INF) {
        objectsMagic[0] = objects[0];
        objectsMagic[1] = objects[0];
```

```
        objectsMagic[2] = before;
        ircodeMagic = make_shared<IRCode>(PLUSOT, std::move(objectsMagic));
        IRCodeList.push_back(std::move(ircodeMagic));
    }
    // >> shPosition
    objectsMagic[0] = objects[0];
    objectsMagic[2] = make_shared<Object>(shPosition);
    objectsMagic[1] = objects[0];
    ircodeMagic = make_shared<IRCode>(SRAOT, objectsMagic);
    IRCodeList.push_back(ircodeMagic);
    // get sign bit
    objects1[0] = generateTempVarT();
    objects1[1] = before;
    objects1[2] = make_shared<Object>(31);
    shared_ptr<VarIRCode> var_temp = make_shared<VarIRCode>(objects1[0]->getName(), 0, false,
                                                            INTST,
                                                            currentDomain);
    varIndex += 1;
    if (currentDomain == 0) {
        global_offset += 1;
    }
    var_temp->setIsTemp(true);
    var_temp->setSpaceInfo(varIndex, 1);
    objects1[0]->setVar(var_temp);
    ircode1 = make_shared<IRCode>(SRLOT, objects1);
    IRCodeList.push_back(ircode1);
    objects[1] = objects[0];
    objects[2] = objects1[0];
    ircode = make_shared<IRCode>(PLUSOT, objects);
    if (dividend < 0) {
        cout << "deal neg" << endl;
        objectsNeg[0] = objects[0];
        objectsNeg[1] = make_shared<Object>(0);
        objectsNeg[2] = objects[0];
        ircodeNeg = make_shared<IRCode>(MINUSOT, objectsNeg);
    }
}
```

寄存器分配

全局寄存器分配的图着色算法未能完成，只实现了临时变量的寄存器分配。其过程和书中的一致。建立一个临时寄存器池，检测到临时变量时，从池中取出一个寄存器，如果池中没有寄存器，采用的是FIFO中，将其放入全局寄存器池中，然后再从全局寄存器池中取出一个寄存器。在基本块结束时，将所有临时寄存器中的变量放回内存中。换回内存时，利用之前所写的load&save函数，将变量放回内存中。

采用FIFO中，用一个变量来标记现在进行替换的寄存器是哪一个，如果该寄存器对应的变量在等式的右值，就顺延到下一个去。

设计的寄存器类如下：

```
class Register {
private:
    bool isFree;
    shared_ptr<Object> object;
    string id;
public:
     Register(string id) : id(std::move(id)) {
        isFree = true;
    }

    void setFree(bool free) {
        isFree = free;
    }

    bool getFree() {
        return isFree;
    }

    void setObject(shared_ptr<Object> &obj) {
        object = obj;
    }

    shared_ptr<Object>& getObject() {
        return object;
    }

    string getId() {
        return id;
    }

    void rollBack(MipsGenerator &mipsGenerator);

    void storeValue(MipsGenerator &mipsGenerator, shared_ptr<Object> &obj);

    static void moveTo(shared_ptr<Register> &reg,  shared_ptr<Object> &obj);

};
```

rollback用于将寄存器储存的变量返回内存中，storeValue用于将变量放入寄存器中，moveTo用于将一个变量放入另一个寄存器中。

编码后修改

- 在计算MagicNumber的时候，运算过程中似乎有爆longlong的风险，于是写了一个高精度类，负责运算。

```
class BigInteger {
    typedef bool Sign;

    friend ostream &operator<<(ostream &, const BigInteger &num);

    friend istream &operator<<(istream &, const BigInteger &num);

public:
    BigInteger() {
        numBits.clear();
    };

    BigInteger(const string &num);

    BigInteger(long long num);

    BigInteger(const BigInteger &num);

    BigInteger &operator=(const BigInteger &num);

    BigInteger operator-(const BigInteger &num) const;

    BigInteger operator+(const BigInteger &num) const;

    BigInteger operator*(const BigInteger &num) const;

    BigInteger operator/(const BigInteger &num) const;

    BigInteger operator^(const int num) const;

    long long toLongLong() const;

    bool operator==(const BigInteger &num) const;

    bool operator!=(const BigInteger &num) const;

    bool operator>(const BigInteger &num) const;

    bool operator<(const BigInteger &num) const;

    bool operator>=(const BigInteger &num) const;

    bool operator<=(const BigInteger &num) const;

private:
    static const int BASE = 10000;
    static const int BASE_LEN = 4;
    vector<int> numBits;
```

```
    static void fotmatting(BigInteger &num);
};
```

- 在mips获得乘法的高32位很简单，直接 `mfhi` 即可，不用像x86那样运算
- 临时寄存器分配不够完善，临时寄存器检查的标准仅为是否是在运算时generate出来的，其真正的分配应该为是否跨越了基本块，导致效率提升有限。