2019《编译器课程设计》申优文章

170613 班 17373356 号 杨昶

目录

一、 需求说明

1. 文法说明

＜加法运算符＞ ::= + | -

＜乘法运算符＞ ::= * | /

＜关系运算符＞ ::= ＜ | <= | ＞ | >= | != | ==

＜字母＞ ::= _ | a | . . . | z | A | . . . | Z

＜数字＞ ::= 0 | ＜非零数字＞

＜非零数字＞ ::= 1 | . . . | 9

＜字符＞ ::= '＜加法运算符＞' | '＜乘法运算符＞' | '＜字母＞' | '＜数字＞'

＜字符串＞ ::= "｛十进制编码为 32,33,35-126 的 ASCII 字符｝"

＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}

| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞} 存在赋值 int char

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝| 0

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝ //标识符和保留字都区分大小写

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞'['＜无符号整数＞']'){,(＜标识符＞|＜标识符＞'['＜无符号整数＞']')}

//＜无符号整数＞表示数组元素的个数，其值需大于 0

//变量没有初始化的情况下没有初值

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞'('＜参数表＞')' '{'＜复合语句＞'}'

＜无返回值函数定义＞ ::= void＜标识符＞'('＜参数表＞')"{'＜复合语句＞'}'

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数表＞　　::=　＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

＜主函数＞　　::= void main‘(’‘)’ ‘{’＜复合语句＞‘}’

＜表达式＞　　::= [＋｜－]＜项＞{＜加法运算符＞＜项＞}　//[+|-]只作用于第一个<项>

＜项＞　　::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞　　::= ＜标识符＞｜＜标识符＞'['＜表达式＞']'|'('＜表达式＞')'｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞　　//char 类型的变量或常量，用字符的 ASCII 码对应的整数参加运算

//＜标识符＞'['＜表达式＞']'中的＜表达式＞只能是整型，下标从 0 开始

//单个＜标识符＞不包括数组名，即数组不能整体参加运算，数组元素可以参加运算

＜语句＞　　::= ＜条件语句＞｜＜循环语句＞| '{'＜语句列＞'}'| ＜有返回值函数调用语句＞;

|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜返回语句＞;

＜赋值语句＞　::=　＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞　存在赋值

//＜标识符＞＝＜表达式＞中的<标识符>不能为常量名和数组名

＜条件语句＞　::= if '('＜条件＞')'＜语句＞［else＜语句＞］

＜条件＞　　::=　＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式需均为整数类型才能进行比较，第二个侯选式中表达式为 0 条件为假，否则为真

＜循环语句＞　::=　while '('＜条件＞')'＜语句＞| do＜语句＞while '('＜条件＞')' |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞　　//for 语句先进行条件判断，符合条件再进入循环体　存在赋值

＜步长＞::= ＜无符号整数＞

＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

＜值参数表＞　::= ＜表达式＞{,＜表达式＞}｜＜空＞　存在赋值

//实参的表达式不能是数组名，可以是数组元素

//实参的计算顺序,要求生成的目标码运行结果与 Clang8.0.0 编译器运行的结果一致

＜语句列＞　::= ｛＜语句＞｝

＜读语句＞　　::=　scanf '('＜标识符＞{,＜标识符＞}')'　　存在赋值

//从标准输入获取<标识符>的值，该标识符不能是常量名和数组名

//生成 PCODE 代码的情况：需要处理为一个 scanf 语句中，若有多个<标识符>，无论标识符的类型是 char 还是 int，每输入一项均需回车

//生成 MIPS 汇编的情况：按照 syscall 指令的用法使用即可

＜写语句＞　　::= printf '('　＜字符串＞,＜表达式＞　')'| printf '('＜字符串＞　')'| printf '('＜表达式＞')'

//printf '('　＜字符串＞,＜表达式＞　')'

//printf '('　＜字符串＞,＜表达式＞　')'输出时，先输出字符串的内容，再输出表达式的值，两者之间无空格

//表达式为字符型时，输出字符；为整型时输出整数

//＜字符串＞原样输出（不存在转义）

//每个 printf 语句的内容输出到一行，按结尾有换行符\n 处理

＜返回语句＞　　::=　return['('＜表达式＞')']

//无返回值的函数中可以没有 return 语句，也可以有形如 return;的语句

//有返回值的函数只要出现一条带返回值的 return 语句即可，不用检查每个分支是否有带返回值的 return 语句

另：关于类型和类型转换的约定：

1. 表达式类型为 char 型有以下三种情况：

1）表达式由<标识符>或＜标识符＞'['＜表达式＞']构成，且<标识符>的类型为 char，即 char 类型的常量和变量、char 类型的数组元素。

2）表达式仅由一个<字符>构成，即字符字面量。

3）表达式仅由一个有返回值的函数调用构成，且该被调用的函数返回值为 char 型

除此之外的所有情况，<表达式>的类型都是 int

2. 只在表达式计算中有类型转换，字符型一旦参与运算则转换成整型，包括小括号括起来的字符型，也算参与了运算，例如(‘c’)的结果是整型。

3. 其他情况，例如赋值、函数传参、if/while 条件语句中关系比较要求类型完全匹配，并且＜条件＞中的关系比较只能是整型之间比，不能是字符型，if ‘(’＜条件＞‘)’和 while ‘(’＜条件＞‘)’里边，如果<条件>是单个表达式，则必须是整型。

2. 我对文法的改写

因为课程网站公布的文法某些非终结符的右侧 FIRST 集合有重叠，所以，为了避免在分析中无法判断下一步“动作”的情况，我对文法进行了修改：

对于＜程序＞　　::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞，默认进行对于＜变量说明＞的检测，如果在＜变量说明＞的第一个＜常量定义＞中读取到“<标识符><（>”，则说明误将＜有返回值函数定义＞当做了＜变量说明＞中的＜常量定义＞，立刻重置相关标记，依次从＜常量定义＞＜变量说明＞中退出，进入＜有返回值函数定义＞中。

对于＜常量说明＞ ::= const＜如果常量定义＞;{ const＜常量定义＞;}，每当检测完一整个“const＜如果常量定义＞;”部分，如果可以检测到“const”，则继续进行“const＜如果常量定义＞;”的检测，直到检测不到“const＜如果常量定义＞;”为止，结束对＜常量说明＞的检测。

对于＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}，每当检测完一整个“＜变量定义＞;”部分，如果可以检测到“int”或“char”，则继续进行“＜变量定义＞;”的检测，直到检测不到“int”或“char”为止，结束对＜变量说明＞的检测。

对于＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞'['＜无符号整数＞']'){,(＜标识符＞|＜标识符＞'['＜无符号整数＞']')}，为使右侧 FIRST 集合彼此不重合，修改此条文法为：＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞['['＜无符号整数＞']']{,＜标识符＞['['＜无符号整数＞']']}。

对于＜参数表＞　　::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞，进入＜参数表＞函数后，先判断是否遇到‘)’，以此判断这个参数表是否为空，如果遇到‘)’，则说明参数表为空，应该从＜参数表＞函数退出，否则，正常进行对于“＜类型标识符＞＜标识符＞”的判断，如果下一个符号是‘,’，就继续判断新的一组“＜类型标识符＞＜标识符＞”，直到下一个符号不是‘,’为止。

对于＜因子＞　　::= ＜标识符＞｜＜标识符＞'['＜表达式＞']'|'('＜表达式＞')'｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞，为使右侧 FIRST 集合彼此不重合，修改此条文法为：＜因子＞　　::= ＜标识符＞['['＜表达式＞'], ＜有返回值函数调用语句＞除函数头的其他部分]|'('＜表达式＞')'｜＜整数＞|＜字符＞。

对于＜语句＞　　::= ＜条件语句＞｜＜循环语句＞| '{'＜语句列＞'}'| ＜有返回值

函数调用语句＞; |＜无返回值函数调用语句＞;丨＜赋值语句＞;丨＜读语句＞;丨＜写语句＞;丨＜空＞;|＜返回语句＞; ，为使右侧 FIRST 集合彼此不重合，修改此条文法为：＜语句＞　::= ＜条件语句＞丨＜循环语句＞| '{'＜语句列＞'}'| <标识符>（＜有返回值函数调用语句除首部标识符部分＞|＜无返回值函数调用语句除首部标识符部分＞丨＜赋值语句除首部标识符部分＞）;丨＜读语句＞;丨＜写语句＞;丨＜空＞;|＜返回语句＞;。

对于＜赋值语句＞　::= ＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞，为使右侧 FIRST 集合彼此不重合，修改此条文法为：＜赋值语句＞　::= ＜标识符＞['['＜表达式＞']']＝＜表达式＞。

对于＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'，＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'，根据＜标识符＞，即被调用的函数的函数名，和函数定义时存储在符号栈中的信息，判断该函数是有返回值的，还是无返回值的，再调用对应的＜有返回值函数调用语句＞或＜无返回值函数调用语句＞。

对于＜值参数表＞　::= ＜表达式＞{,＜表达式＞}丨＜空＞，进入＜值参数表＞函数后，先判断是否遇到‘)’，以此判断这个参数表是否为空，如果遇到‘)’，则说明这个值参数表为空，应该从＜值参数表＞函数退出，否则，正常进行对于＜表达式＞的判断，如果下一个符号是‘,’，就继续判断新的＜表达式＞，直到下一个符号不是‘,’为止。

对于＜语句列＞　::= ｛＜语句＞｝，因为＜语句列＞一定以“'{'＜语句列＞'}'”的形式出现，所以进入＜语句列＞函数后，先判断是否遇到‘}’，以此判断这个语句列是否为空，如果遇到‘}’，则说明这个语句列为空，应该从<语句列>函数退出，否则，正常进行对于＜语句＞的判断，直到下一个符号是‘}’为止。

修改后的文法避免了 FIRST 集冲突的情况，因此可以用递归下降方法进行语法分析。

3. 目标代码说明

代码生成作业，要求将符合上述 C0 文法的 C 语言程序，转化为符合 mips 语法要求的程序。关于 mips 汇编语言的相关知识，详请参考《计算机组成原理》的有关内容。在这里，我给出我在将 c 程序转化为 mips 程序的过程中用到的 mips 指令，并给出简单的使用例子和解释。

指令	使用例子	解释

add	add $t0, $t1, $t2	寄存器加运算
addi	addi $t0, $t1, 100	寄存器和立即数加运算
sub	sub $t0, $t1, $t2	寄存器减运算
li	li $t0, 100	加载立即数到寄存器
sw	sw $t0, 0($t1)	寄存器值保存到内存中
lw	lw $t0, 0($t1)	内存上的值加载到寄存器中
beq, bne, bgez, bgtz, blez, bltz	beq, bne, bgez, bgtz, blez, bltz	条件跳转
j jal jr	j jal jr	跳转
mult	mult $t0, $t1	乘法
div	div $t0, $t1 mflo $t2	除法
mflo	mflo $t0	从低位寄存器中取值到寄存器$t0
syscall	syscall	系统调用
la	la $a0, 0x20000000	加载地址

二、 具体设计

1. 整体结构分析

我认为，从整体上，需要完成这几个方面的工作：1）遍历 c 程序，进行词法分析和语法分析，进行错误处理；2）建立符号变，分配各参数、变量、数组对应的相对与$sp 的偏移量，记录各个常量的值，记录各个函数存储位置的相关信息；3）生成中间代码并保存；4）对中间代码进行处理和优化；5）将优化后的中间代码转换成 mips。其中，任务 1 2 3 都在遍历 c 程序的过程中完成，且是进行任务 4 5 的必备条件。

2. 符号栈设计说明

我们符号栈设计分为 3 个部分：

首先，stack1 如下图所示：

```
struct node {
    char name[100];
    char nameType[10];
    char parameters[20][10]; // 横行为参数类型int/char 不需要在这里储存参数名字
    int parameterNum = 0;
    bool canChange = true;

    int value;  // 针对 常量 -- 第一次出现时就有值 要保留这个值
    int offset = 1; // 针对 变量 参数 数组 -- 要分配一个相对于 $sp 的偏移量 用于储存值
    int inStack3List = -1; // 针对 func  -- 对应的是第几个自定义函数 这个函数的 参数+变量+数组 的 offset
                           // -1 不是函数   >= 0 是函数

    int offset_begin; // offset_end < ___ <= offset_begin
    int offset_end;

} stack1[2000]; // a name chat f1 ...
int stacktop1 = 0;
```

这个符号栈用于在“遍历 c 程序的某个瞬间”存储“在这个瞬间可以使用的所有常量、变量、函数、参数、数组——即所有标识符”。

name 用于存储标识符的名字

nameType 用于存储这个标识符对应主体的种类，可能的取值包括“int”“char”“f_int”“f_char”“f_void”。

parameters，如果这个标识符对应一个函数，那么，parameters 顺序存储这个函数的所有参数的类型（int 或 char）。

parameterNum，如果这个标识符对应一个函数，那么，parameterNum 顺序存储这个函数的参数的个数。

canChange，如果这个标识符对应一个常量，那么，canChange 取值为 false，否则，取值为 true。

value，如果这个标识符对应一个常量，那么，value 存储常量的值，否则其值无意义。

offset，如果这个标识符对应一个变量或数组或参数，那么，offset 存储这个标识符被分配的、相对于$sp 的偏移量，否则其值无意义。

inStack3List，如果这个标识符对应一个函数，那么，inStack3List 表示这个函数对应 stack3List 中的哪一“块”；否则，其值无意义，置为-1。

offset_begin，如果这个标识符对应一个函数，那么，offset_begin 表示这个函数被调用时需要缓存的内存段的起始地址。

offset_end，如果这个标识符对应一个函数，那么，offset_end 表示这个函数被调用时需要缓存的内存段的终止地址。

stack2 主要用于配合 stack1 进行函数定义结束后的“覆盖”，保证 stack1 对应的是“遍历 c 程序的某个瞬间”存储“在这个瞬间可以使用的所有常量、变量、函数、参数、数组—

—即所有标识符”。

```
int stack2[20]; // 1 4 12 ...
int stacktop2 = 0;
```

在即将进入一个函数定义模块时，进行这样的操作：

```
stack2[stacktop2] = stacktop1;
stacktop2++;
```

在即将离开一个函数定义模块时，进行这样的操作：

```
stacktop1 = stack2[stacktop2 - 1];
stacktop2--;
```

Stack3List 主要用于保存在 stack1 中被覆盖的、各个自定义函数内部的变量、参数、数组对应的偏移量。

```
struct stack3 {
    char name[50][100];
    int offset[50];
    int top = 0;
};

stack3 stack3List[30];
int stack3Listtop = 0;
```

3. 中间代码设计说明

symbol	iden1	iden2	iden3	解释
+	a	b	c	a=b+c
-	a	b	c	a=b-c
*	b	b	c	a=b*c
/	c	b	c	a=b/c
=[]	a	b	c	a=b[c]
[]=	a	b	c	a[b]=c
f_begin	f$begin:			函数开始的标签

f_end	f_end			函数结束的标签
s_envi				进行函数调用前保存环境
goto	$label2			跳转到某标签
exit				主函数结束
main:				主函数开始的标签
>,>=,<,<=,==,!=				< 条件 > 中的跳转语句
label	$label3			生成一个标签，例如：$label3
printf	str	2		输出第二个字符串，即，字符串$Message2
printf	int	3		输出 int 型值 2
printf	char	97		输出 char 型值 97，即‘a’
scanf	-80	3		将值 3 存储到 -80($sp) 的位置
call	f$begin			调用函数 f，即，跳转到标签“f$begin：”
push	-80	3		将值 3 存储到 -80($sp) 的位置

4. 寄存器的使用说明

我使用了两种寄存器：

1）真寄存器

```
char registerStack[15][10] = { "$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$t9",
        "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7"};
int registerStackTop = 0;
```

2）假寄存器

```
int fackRegisterOffset[200];
int fackRegisterOffsetTop = 0;
```

真寄存器的使用，只需要从前到后将第一个未被使用的真寄存器“弹栈”，然后在生成中间代码时使用即可；当所有真寄存器都被使用完了，可以生成对应一小块内存空间的假寄存器，形式上是将值存储在假寄存器中，实际上，是将值放在各个假寄存器对应的小块内存中。具体使用例子如下图所示：

```
if (registerStackTop >= 15) {
    char s0[100];
    strcpy(s0, "$");
    char s1[100];
    transferNumToStr(fackRegisterOffsetTop, s1);
    strcat(s0, s1);
    fackRegisterOffset[fackRegisterOffsetTop] = offsetToBeUsed;
    offsetToBeUsed -= 4;
    fackRegisterOffsetTop++;
    strcpy(midStack[midStacktop].iden1, s0);
    strcpy(item1Iden, s0);
}
else {
    strcpy(midStack[midStacktop].iden1, registerStack[registerStackTop]);
    strcpy(item1Iden, registerStack[registerStackTop]);
    registerStackTop++;

}
```

三、 代码优化

1. 常数合并与传播

在这一部分，我的目标是，当一个表达式的整体或者部分可以直接用数字（或数值）来表示的时候，在遍历输入的 testfile.txt 生成中间代码的时候，直接生成最终的数字运算结果，比如，testfile.txt 中 c 代码为

c = (2 + 3) * (4 + 5);

我的程序会将这个<语句>分析为<标识符> = <表达式>，其中，<标识符> = c，<表达式> = (2 + 3) * (4 + 5)。通过<表达式><项><因子>的嵌套调用，最终给出<表达式> = 54 的结果。这一部分我主要通过对各个<表达式><项><因子>的类型的分析（这个<表达式><项><因子>究竟是不知道具体值的“名称”，还是已经可以对应给出具体数值的“值”），并进行相应的运算或生成中间代码来完成。这一部分主要在我的 Program()遍历 testfile.txt 部分进行，在<表达式><项><因子>的检测中完成。

```
void expression(char expressionIden[100], int* expressionchooseWhich, int* expressionNum) { ... }
void item(char itemIden[100], int* itemchooseWhich, int* itemNum) { ... }
void factor(char factorIden[100], int* factorchooseWhich, int* factorNum) { ... }
```

比如，expr = item1 + item2。如果 item1、item2 中有一个或两个应当对应无法给出具体值的 itemIden（表示运算的中间变量的真、假寄存器，或不确定值的变量、数组元素、参数、有返回值函数调用的返回值），那么，应当生成中间代码；如果 item1、item2 都是具体的值（直接是数字、数值，或者是已定义常量），那么，立即对这两个值进行对应的计算，并向 expr 返回最终运算结果。

相关代码局部截图如下所示。

```
judgeType();

char item2Iden[100];
int item2chooseWhich;
int item2Num;
item(item2Iden, &item2chooseWhich, &item2Num);

if (item1chooseWhich == 1 && item2chooseWhich == 1) { ... }
else if (item1chooseWhich == 1 && item2chooseWhich == 2) { ... }
else if (item1chooseWhich == 2 && item2chooseWhich == 1) { ... }
else if (item1chooseWhich == 2 && item2chooseWhich == 2) {
    // 1                                    1
    // item1Iden = "#" item1chooseWhich = 2  itemuom  item1Num = 1 +/- 1
    strcpy(item1Iden, "#");
    item1chooseWhich = 2;
    if (with_ == true) {
        // -
        item1Num = item1Num - item2Num;
    }
    else {
        // +
        item1Num = item1Num + item2Num;
    }
}

expressionType = 1;
```

2. 循环语句生成中间代码的修改

原本，秉承着“直接翻译，减少错误”的思想，我对循环语句的中间代码进行了这样的翻译：

```
while (a < b) {
    c = 1;
}
// 翻译为
// $label0:
// if a >= b then goto $label1
// c = 1
// goto $label0
// $label1:

do {
    c = 1;
} while (a < b);
// 翻译为
// $label0:
// c = 1
// if a >= b then goto $label1
// goto $label0
// $label1:


for (i = 0; i < 10; i = i + 1) {
    c = 1;
}
// 翻译为
// i = 0 + 0
// $label0:
// if i >= 10 then goto $label1
// goto $label2
// $label3:
// i = i + 1
// goto $label0
// $label2:
// c = 1
// goto $label3
// $label1:
```

后来，考虑到竞速排名的要求，我对循环语句部分中间代码的设计进行了这样的修改：

```
while (a < b) {
    c = 1;
}
// 翻译为
// if a >= b then goto $label0
// $label1:
// c = 1
// if a < b then goto $label1
// $label0:

do {
    c = 1;
} while (a < b);
// 翻译为
// $label0:
// c = 1
// if a < b then goto $label0
```

```
for (i = 0; i < 10; i = i + 1) {
    c = 1;
}
// 翻译为
// i = 0 + 0
// if i >= 10 then goto $label0
// $label1:
// c = 1
// i = i + 1
// if i < 10 then goto $label1
// $label0:
```

这样修改，使得中间代码从原来的，每循环一次，至少要经过 1 条 branch 语句和 1 条 jump 语句（原 for 循环需要经过更多的语句），改变为只需要经过 1 条 jump 语句，在很大程度上降低了生成的 mips 汇编的运行 FinalCycle 值。

3. 死代码删除

在我的程序中，死代码删除，指的是删除掉明显不会进入的循环快或 if-else 语句块，比如，在下述 while 循环中：

```
while (a < b) {
    c = 1;
}
// 翻译为
// if a >= b then goto $label0
// $label1:
// c = 1
// if a < b then goto $label1
// $label0:
```

如果 a，b 对应的是已知值的常量或一个数字，且 a 的值大于等于 b 的值，那么，我们可以直接删除这个 while 循环语句对应的这几行中间代码，因为他们永远不会被真正使用了。

需要考虑进行死代码删除的，包括 if 语句，if-else 语句，while 语句，for 语句，do-while 语句，具体进行实在生成了所有中间代码后，对已经生成的中间代码进行遍历，找到可以删除的地方，删除。

```
if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "==") == 0
    || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "!=") == 0
    || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, ">=") == 0
    || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "<=") == 0
    || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, ">") == 0
    || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "<") == 0) {

    if (ifThisIdenIsNum(funcBlockList[r].basicBlockList[p].midList[q].iden1)
        && ifThisIdenIsNum(funcBlockList[r].basicBlockList[p].midList[q].iden2)) {
        int m = transferStrToNum(funcBlockList[r].basicBlockList[p].midList[q].iden1, 0);
        int n = transferStrToNum(funcBlockList[r].basicBlockList[p].midList[q].iden2, 0);

        if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "==") == 0 && m == n
            || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "!=") == 0 && m != n
            || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, ">=") == 0 && m >= n
            || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "<=") == 0 && m <= n
            || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, ">") == 0 && m > n
            || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "<") == 0 && m < n) {

            char target[100];
            strcpy(target, funcBlockList[r].basicBlockList[p].midList[q].iden3);
            strcat(target, ":");

            bool couldJump = false;
            int j = q + 1;
            for (j = q + 1; j < funcBlockList[r].basicBlockList[p].midListTop; j++) {
                if (strcmp(funcBlockList[r].basicBlockList[p].midList[j].symbol, "label") == 0
                    && strcmp(funcBlockList[r].basicBlockList[p].midList[j].iden1, target) == 0) {

                    couldJump = true;
                    break;
                }
            }

            if (couldJump) {

                int k = q;
                for (k = q, j = j + 1; j < funcBlockList[r].basicBlockList[p].midListTop; k++, j++) {
                    strcpy(funcBlockList[r].basicBlockList[p].midList[k].symbol,
                        funcBlockList[r].basicBlockList[p].midList[j].symbol);
                    strcpy(funcBlockList[r].basicBlockList[p].midList[k].iden1,
                        funcBlockList[r].basicBlockList[p].midList[j].iden1);
                    strcpy(funcBlockList[r].basicBlockList[p].midList[k].iden2,
                        funcBlockList[r].basicBlockList[p].midList[j].iden2);
                    strcpy(funcBlockList[r].basicBlockList[p].midList[k].iden2,
                        funcBlockList[r].basicBlockList[p].midList[j].iden2);
                }
                funcBlockList[r].basicBlockList[p].midListTop = k;
                q--;

            }

        }
```

4. 寄存器的复用策略

在原本的设计中，我的寄存器主要使用在：

① 中间代码生成 mips 汇编指令时，用于“中间暂时存储”，例如：

```
// n = a + b
// 对应mips指令可以是
// lw $t0, a_offset($sp)
// lw $t1, b_offset($sp)
// add $t2, $t0, $t1
// sw $t2, n_offset($sp)
```

② <表达式><项><因子>嵌套，生成中间代码时，例如：

```
// n = (a + 1) * (b + 2)
// 对应mips指令可以是
// $t0 = a + 1
// $t1 = b + 2
// mult $t0, $t1
// mflo $t2
// sw $t2, n_offset($sp)
```

③ <条件><返回语句><赋值语句><写语句>等调用<表达式>语句的地方，例如：

```
// printf((a + 1) * (b + 2));
// 对应mips指令可以是
// $t0 = a + 1
// $t1 = b + 2
// mult $t0, $t1
// mflo $t2
// printf int $t2
```

在这里，为了优化性能，我进行修改：

1，中间代码生成 mips 汇编指令时，只是使用$t0、$t1 两个寄存器，例如：

```
// n = (a + 1) * (b + 2));
// 对应mips指令 可以是
// lw $t0, a_offset($sp)
// addi $t0, $t0, 1
// lw $t1, b_offset($sp)
// addi $t1, $t1, 2
// mult $t0, $t1
// mflo $t0
// sw $t0, n_offset($sp)
```

2，<表达式><项><因子>嵌套生成中间代码，或<条件><返回语句><赋值语句><写语句>等调用<表达式>语句时，在调用或生成结束后，立刻恢复原本的寄存器栈使用标记，使得同一个寄存器多次使用，例如：

```
// n = (a + 1) * (b + 2));
// m = (a + 3) * (b + 4));
// 对应mips指令 原本可以是
// lw $t0, a_offset($sp)
// addi $t0, $t0, 1
// lw $t1, b_offset($sp)
// addi $t1, $t1, 2
// mult $t0, $t1
// mflo $t0
// sw $t0, n_offset($sp)
// lw $t2, a_offset($sp)
// addi $t2, $t2, 3
// lw $t2, b_offset($sp)
// addi $t3, $t3, 4
// mult $t2, $t3
// mflo $t2
// sw $t2, m_offset($sp)
```

这样使用寄存器（不恢复），仅这一段，就使用了$t0$t1$t2$t3　4 个寄存器，且随着代码的增长会不断继续使用剩下的寄存器，寄存器“损耗”得太快，且一个寄存器只用于一句 c 代码的翻译，效率很低，且不利于后续“为变量分配寄存器”的优化（很容易没有寄存器剩下，即没有寄存器可用了）。因此，我更改了我使用寄存器的方法，每在一句话的翻译结束后，就恢复寄存器栈状态带这句话开始时的状态，相当于多次重用寄存器，例如：

```
// n = (a + 1) * (b + 2));
// m = (a + 3) * (b + 4));
// 对应mips指令 现在可以是
// lw $t0, a_offset($sp)
// addi $t0, $t0, 1
// lw $t1, b_offset($sp)
// addi $t1, $t1, 2
// mult $t0, $t1
// mflo $t0
// sw $t0, n_offset($sp)
// lw $t0, a_offset($sp)
// addi $t0, $t0, 3
// lw $t1, b_offset($sp)
// addi $t1, $t1, 4
// mult $t0, $t1
// mflo $t0
// sw $t0, m_offset($sp)
```

5. 中间代码生成 mips 指令的细致讨论

一开始，由中间代码生成 mips 指令时，我采取的是“直接分别取出”“进行运算”“放回结果”，将大多数中间代码转化为 3~4 条 mips 指令，例如：

```
// a = b + c
// lw $t0, b_offset($sp)
// lw $t1, c_offset($sp)
// add $t0, $t0, $t1
// sw $t0, a_offset($sp)
```

```
// $a1 = b + 1
// lw $t0, b_offset($sp)
// li $t1, 1
// add $t0, $t0, $t1
// addi $a1, $t0, 0
```

这样设计，在完成“将中间代码转化为 mips 指令”的函数时，无疑是比较方便快捷的：只需要关注 a、b、c 各自都时如何存储的，b、c 之间有怎样的运算关系，就可以了。不必细致地讨论各种可能存在的情况。然而，这种情况在很多情况下非常冗长，例如，上例 2 可以修改为：

```
// $a1 = b + 1
// lw $t0, b_offset($sp)
// addi $a1, $t0, 1
```

根据这个思路，我在生成 mips 时进行了较为细致的讨论，例如：

```
if (strcmp(midStack2[j].symbol, "+") == 0
    || strcmp(midStack2[j].symbol, "-") == 0
    || strcmp(midStack2[j].symbol, "*") == 0
    || strcmp(midStack2[j].symbol, "/") == 0) {
    // iden1 = iden2 +-*/ iden3

    // 数字  ---  数字
    if (ifThisIdenIsNum(midStack2[j].iden2) && ifThisIdenIsNum(midStack2[j].iden3)) { ... }
    // 数字  ---  真寄存器$v0
    else if (ifThisIdenIsNum(midStack2[j].iden2) && ifThisIdenIsRealRegister(midStack2[j].iden3)) { ... }
    // 数字  ---  假寄存器/标识符 对应offset
    else if (ifThisIdenIsNum(midStack2[j].iden2)
        && (ifThisIdenIsFackRegister(midStack2[j].iden3) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden3))) { ... }
    // 数字  ---  标识符 对应寄存器
    else if (ifThisIdenIsNum(midStack2[j].iden2) && ifThisIdenIsIdentifierWithRegister(midStack2[j].iden3)) { ... }

    // 真寄存器$v0  ---  数字
    else if (ifThisIdenIsRealRegister(midStack2[j].iden2) && ifThisIdenIsNum(midStack2[j].iden3)) { ... }
    // 真寄存器$v0  ---  真寄存器$v0  (已经排除 $v0 $v0 的情况)
    else if (ifThisIdenIsRealRegister(midStack2[j].iden2) && ifThisIdenIsRealRegister(midStack2[j].iden3)) { ... }
    // 真寄存器$v0  ---  假寄存器/标识符 对应offset
    else if (ifThisIdenIsRealRegister(midStack2[j].iden2)
        && (ifThisIdenIsFackRegister(midStack2[j].iden3) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden3))) { ... }
    // 真寄存器$v0  ---  标识符 对应寄存器
    else if (ifThisIdenIsRealRegister(midStack2[j].iden2) && ifThisIdenIsIdentifierWithRegister(midStack2[j].iden3)) { ... }

    // 标识符 对应寄存器  ---  数字
    else if (ifThisIdenIsIdentifierWithRegister(midStack2[j].iden2) && ifThisIdenIsNum(midStack2[j].iden3)) { ... }
    // 标识符 对应寄存器  ---  真寄存器$v0
    else if (ifThisIdenIsIdentifierWithRegister(midStack2[j].iden2) && ifThisIdenIsRealRegister(midStack2[j].iden3)) { ... }
    // 标识符 对应寄存器  ---  假寄存器/标识符 对应offset
    else if (ifThisIdenIsIdentifierWithRegister(midStack2[j].iden2)
        && (ifThisIdenIsFackRegister(midStack2[j].iden3) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden3))) { ... }
    // 标识符 对应寄存器  ---  标识符 对应寄存器
    else if (ifThisIdenIsIdentifierWithRegister(midStack2[j].iden2) && ifThisIdenIsIdentifierWithRegister(midStack2[j].iden3)) { ... }

    // 假寄存器/标识符 对应offset  ---  数字
    else if ((ifThisIdenIsFackRegister(midStack2[j].iden2) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden2))
        && ifThisIdenIsNum(midStack2[j].iden3)) { ... }
    // 假寄存器/标识符 对应offset  ---  真寄存器$v0
    else if ((ifThisIdenIsFackRegister(midStack2[j].iden2) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden2))
        && ifThisIdenIsRealRegister(midStack2[j].iden3)) { ... }
    // 假寄存器/标识符 对应offset  ---  假寄存器/标识符 对应offset
    else if ((ifThisIdenIsFackRegister(midStack2[j].iden2) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden2))
        && (ifThisIdenIsFackRegister(midStack2[j].iden3) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden3))) { ... }
    // 假寄存器/标识符 对应offset  ---  标识符 对应寄存器
    else if ((ifThisIdenIsFackRegister(midStack2[j].iden2) || ifThisIdenIsIdentifierWithOffset(midStack2[j].iden2))
        && ifThisIdenIsIdentifierWithRegister(midStack2[j].iden3)) { ... }
}
```

6. 为变量分配寄存器

在遍历 testfile.txt 构建符号表、生成中间代码的过程中，我为每个变量、数组、参数分配了一个各不相同的 offset。

```
// 自定义函数 内部 变量 数组
char n[100];
transferNumToStr(offsetToBeUsed, n);
strcpy(stack1[stacktop1 - 1].offset, n);
//stack1[stacktop1 - 1].offset = offsetToBeUsed;
if (funcInStack3List != -1) {
    strcpy(stack3List[funcInStack3List].name[stack3List[funcInStack3List].top], wordStore);

    char s[10];
    transferNumToStr(offsetToBeUsed, s);
    strcpy(stack3List[funcInStack3List].offset[stack3List[funcInStack3List].top], s);
    //stack3List[funcInStack3List].offset[stack3List[funcInStack3List].top] = offsetToBeUsed;

    strcpy(stack3List[funcInStack3List].type[stack3List[funcInStack3List].top], "var");

    stack3List[funcInStack3List].top++;
}
offsetToBeUsed -= 4;
```

在生成了所有的中间代码后，在由中间代码转 mips 之前，为了优化性能，我为变量分配寄存器，要求是：对于各个函数，已使用过的寄存器不可被分派；分配寄存器只针对没分配过寄存器的变量（不包括数组和参数），同一函数内分配的寄存器不可重复，且按各个变量被“引用”（取值或赋值）的次数，由引用次数多、到引用次数少进行分配。

① 划分函数块和基本块，各个定义函数和 main 函数分别是不同的函数块；“能通过跳转到达的语句是基本块起始”，我们将 if 语句、if-else 语句、循环语句划分为基本块，例如：

```
while (a < b) {
    c = 1;
}
// ------------------------------
// if a >= b then goto $label0
// $label1:
// c = 1
// if a < b then goto $label1
// $label0:
// ------------------------------

do {
    c = 1;
} while (a < b);
// ------------------------------
// $label0:
// c = 1
// if a < b then goto $label0
// ------------------------------
```

```
for (i = 0; i < 10; i = i + 1) {
    c = 1;
}
// i = 0 + 0
// ------------------------------
// if i >= 10 then goto $label0
// $label1:
// c = 1
// i = i + 1
// if i < 10 then goto $label1
// $label0:
// ------------------------------
```

```
if (a > b) {
    c = 1;
}
// ------------------------------
// if a > b then goto $label0
// c = 1
// $label0:
// ------------------------------
```

```
if (a > b) {
    c = 1;
}
else {
    c = 2;
}
// ------------------------------
// if a > b then goto $label0
// c = 1
// goto $label1
// ------------------------------
// $label0:
// c = 2
// $label1:
// ------------------------------
```

② 统计各个基本块，各个函数块最“多”使用到了哪个寄存器。

```
for (int r = 0; r < funcBlockTop; r++) {
    for (int p = 0; p < funcBlockList[r].basicBlockTop; p++) {
        // t2 t3 t4 t5 t6 t7 t8 t9
        int use[8] = { 0,0,0,0,0,0,0,0 };
        for (int q = 0; q < funcBlockList[r].basicBlockList[p].midListTop; q++) {
            if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "+") == 0
                || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "-") == 0
                || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "*") == 0
                || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "/") == 0
                || strcmp(funcBlockList[r].basicBlockList[p].midList[q].symbol, "=[]") == 0) {
                if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t2") == 0) {
                    use[0] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t3") == 0) {
                    use[1] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t4") == 0) {
                    use[2] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t5") == 0) {
                    use[3] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t6") == 0) {
                    use[4] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t7") == 0) {
                    use[5] = 1;
                }
```

```
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t8") == 0) {
                    use[6] = 1;
                }
                else if (strcmp(funcBlockList[r].basicBlockList[p].midList[q].iden1, "$t9") == 0) {
                    use[7] = 1;
                }
            }
        }
        funcBlockList[r].basicBlockList[p].nextTempRegisterCouldUse = 0;
        for (int q = 7; q >= 0; q--) {
            if (use[q] == 1) {

                funcBlockList[r].basicBlockList[p].nextTempRegisterCouldUse = q + 1; // 是栈的下标 不是$tn

                break;
            }
        }
    }
    int ttop = 0;
    for (int p = 0; p < funcBlockList[r].basicBlockTop; p++) {
        if (funcBlockList[r].basicBlockList[p].nextTempRegisterCouldUse > ttop) {
            ttop = funcBlockList[r].basicBlockList[p].nextTempRegisterCouldUse;
        }
    }
    funcBlockList[r].registerTop = ttop;
    for (int p = 0; p < stacktop1; p++) {
        if (strcmp(stack1[p].name, funcBlockList[r].name) == 0) {
            stack1[p].register_begin = 0;
            stack1[p].register_end = ttop;
            break;

        }
    }
}
```

③ 统计各个函数块中，各个变量的使用次数。

```
for (int r = 0; r < funcBlockTop; r++) {
    for (int p = 0; p < funcBlockList[r].basicBlockTop; p++) {
        for (int q = 0; q < funcBlockList[r].basicBlockList[p].midListTop; q++) {
            mid midNode = funcBlockList[r].basicBlockList[p].midList[q];
            if (strcmp(midNode.symbol, "+") == 0 || strcmp(midNode.symbol, "-") == 0
                || strcmp(midNode.symbol, "*") == 0 || strcmp(midNode.symbol, "/") == 0) {
                meetOneIden(midNode.iden1, r, funcBlockList[r].basicBlockList[p].inLoop);
                meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
                meetOneIden(midNode.iden3, r, funcBlockList[r].basicBlockList[p].inLoop);
            }
            else if (strcmp(midNode.symbol, "=[]") == 0) {
                meetOneIden(midNode.iden1, r, funcBlockList[r].basicBlockList[p].inLoop);
                meetOneIden(midNode.iden3, r, funcBlockList[r].basicBlockList[p].inLoop);
            }
            else if (strcmp(midNode.symbol, "[]=") == 0) {
                meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
                meetOneIden(midNode.iden3, r, funcBlockList[r].basicBlockList[p].inLoop);
            }
            else if (strcmp(midNode.symbol, ">") == 0 || strcmp(midNode.symbol, ">=") == 0 || strcmp(midNode.symbol, "<") == 0
                || strcmp(midNode.symbol, "<=") == 0 || strcmp(midNode.symbol, "==") == 0 || strcmp(midNode.symbol, "!=") == 0) {
                meetOneIden(midNode.iden1, r, funcBlockList[r].basicBlockList[p].inLoop);
                meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
            }
            else if (strcmp(midNode.symbol, "printf") == 0) {
                if (midNode.iden2[0] >= 'a' && midNode.iden2[0] <= 'z'
                    || midNode.iden2[0] >= 'A' && midNode.iden2[0] <= 'Z' || midNode.iden2[0] == '_') {
                    meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
                }
            }

            else if (strcmp(midNode.symbol, "scanf") == 0) {
                meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
            }
            else if (strcmp(midNode.symbol, "push") == 0) {
                if (midNode.iden2[0] >= 'a' && midNode.iden2[0] <= 'z'
                    || midNode.iden2[0] >= 'A' && midNode.iden2[0] <= 'Z' || midNode.iden2[0] == '_') {
                    meetOneIden(midNode.iden2, r, funcBlockList[r].basicBlockList[p].inLoop);
                }
            }
        }
    }
}
```

④ 按使用次数从多到少的顺序，为变量分配寄存器，函数内部不可重复，不同函数内

可以重复，并更新每个函数对应的环境信息。

```
for (int r = 0; r < stack3Listtop; r++) {
    while (funcBlockList[r].registerTop <= 15) {
        int meetMostTimes = 0;
        int whichOne = -1;
        for (int p = 0; p < stack3List[r].top; p++) {
            if (stack3List[r].meetNum[p] >= meetMostTimes && strcmp(stack3List[r].type[p], "var") == 0
                && stack3List[r].offset[p][0] != '$') {
                meetMostTimes = stack3List[r].meetNum[p];
                whichOne = p;
            }
        }
        if (whichOne == -1) {
            break;
        }
        stack3List[r].offsetUsedBefore[stack3List[r].offsetUsedBeforeTop] = transferStrToNum(stack3List[r].offset[whichOne], 0);
        stack3List[r].offsetUsedBeforeTop++;
        stack3List[r].meetNum[whichOne] = -1;
        strcpy(stack3List[r].offset[whichOne], registerStack[funcBlockList[r].registerTop]);
        funcBlockList[r].registerTop++;
    }
}

while (funcBlockList[funcBlockTop - 1].registerTop <= 17) {

    int meetMostTimes = 0;
    int whichOne = -1;

    for (int p = stacktop1MainBegin; p < stacktop1; p++) {
        if (stack1[p].meetNum >= meetMostTimes && stack1[p].isArray == false && stack1[p].offset[0] != '$') {
            meetMostTimes = stack1[p].meetNum;
            whichOne = p;
        }
    }

    if (whichOne == -1) {
        break;
    }

    stack1[whichOne].meetNum = -1;
    strcpy(stack1[whichOne].offset, registerStack[funcBlockList[funcBlockTop - 1].registerTop]);
    funcBlockList[funcBlockTop - 1].registerTop++;
}
```

7. 保存现场、恢复现场的修改策略

原本，在保存与恢复现场时，我是将一遍移动$fp（addi $fp, $fp, 4 或 addi $fp, $fp, -4），一遍进行保存现场或恢复现场的操作。这样无疑生成了更多的指令。现在更改为，用 int move 的加减 4 表示$fp 的移动，最后再生成 addi $fp, $fp, move 或 addi $fp, $fp, -move，例如。

```
if (judgeFuncBInFuncA(midStack2[j].iden1, midStack2[j].iden1) == true) {
    for (p = offset_para; p > offset_end; p -= 4) {

        bool usedBefore = false;

        for (int q = 0; q < stack3List[whichOne].offsetUsedBeforeTop; q++) {
            if (stack3List[whichOne].offsetUsedBefore[q] == p) {
                usedBefore = true;
                break;
            }
        }

        if (usedBefore == false) {
            fprintf(filewrite, "lw $t1, %d($sp)\n", p);
            fprintf(filewrite, "sw $t1, %d($fp)\n", move);
            move = move - 4;
        }

    }
}

fprintf(filewrite, "addi $fp, $fp, %d\n", move);
```