编译器设计文档

目录

[一、 文法解读 2](#_Toc27476)

[二、词法分析 3](#_Toc31595)

[三、语法分析 4](#_Toc25861)

[四、语义分析：词法语法语义的错误处理 5](#_Toc17159)

[五、中间代码四元式及目标代码MIPS生成 7](#_Toc24381)

[六、代码优化处理 11](#_Toc5084)

1. **文法解读**

首先构建语法树

确保能够覆盖给出的所有文法，测试程序以<程序>为识别入口，主函数void main(){}

＜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＞' 这里的字符包括下划线

＜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＞} 这里字符只有加减乘除和字母数字

＜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＞｝

＜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

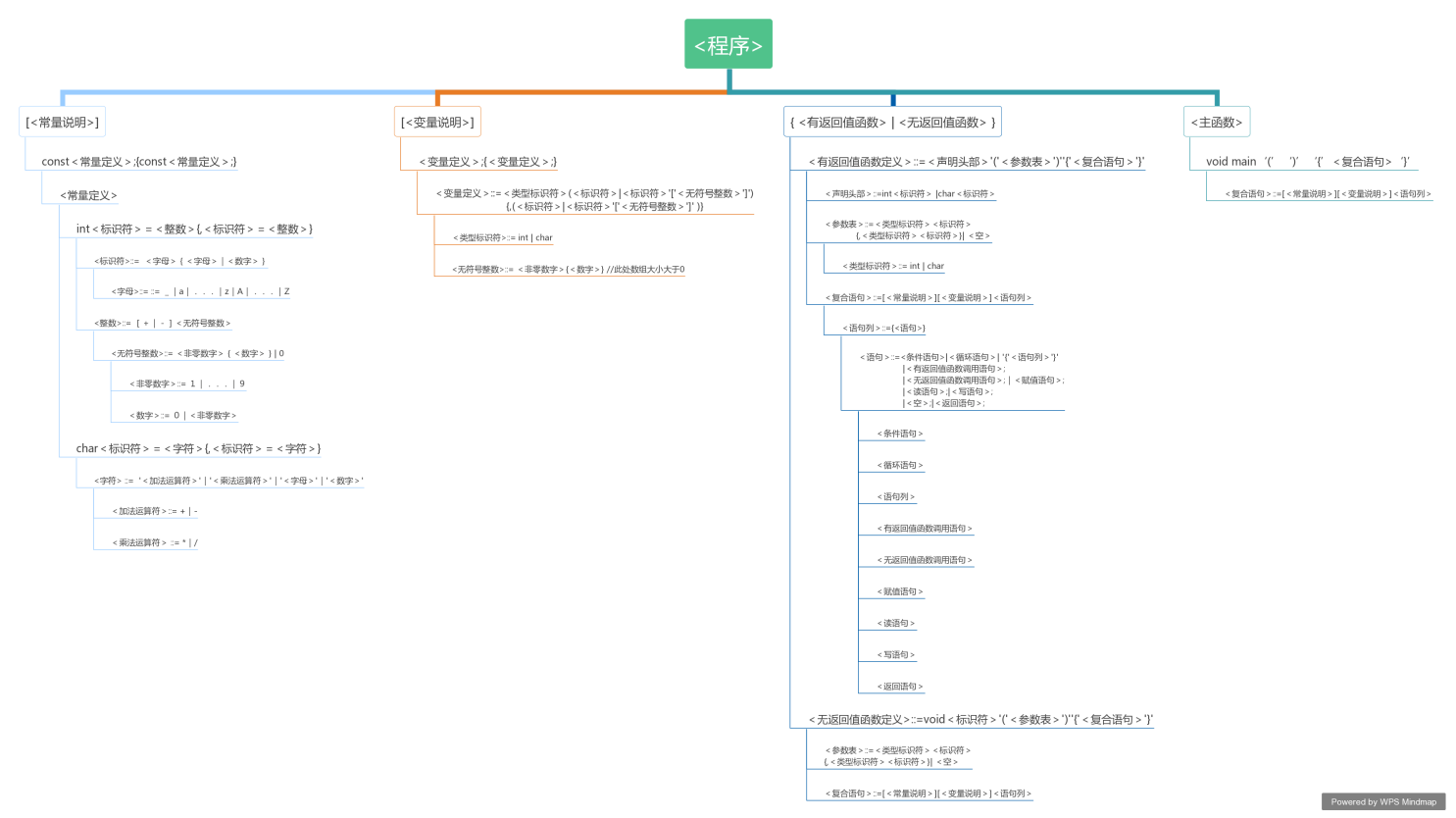
＜类型标识符＞      ::=  int | char

＜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＞           
＜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＞  
＜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＞  
＜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＞')' 需要用符号表区分，和赋值语句区分  
＜值参数表＞   ::= ＜表达式＞{,＜表达式＞}｜＜空＞  
＜语句列＞   ::= ｛＜语句＞｝  
＜读语句＞    ::=  scanf '('＜标识符＞{,＜标识符＞}')'  
＜写语句＞    ::= printf '(' ＜字符串＞,＜表达式＞ ')'| printf '('＜字符串＞ ')'| printf '('＜表达式＞')'  
＜返回语句＞   ::=  return['('＜表达式＞')']



从文法来看，函数内参数不能为指针或者数组，没有switch case情况语句，字符串不考虑换行回车符

**二、词法分析**



// 关键字枚举

const char\* KeyWordsArr[KEY\_NUM] =

{

"const" , "int" , "char" , "void",

"main" , "if" , "else" , "switch",

"do", "while", "for", "scanf",

"printf", "return"

};

**三、语法分析**

沿用词法分析的程序和输出，将词法分析的结果保存至字符串数组中。增加预读和回退函数，输出当前单词函数等功能。增加部分符号表和错误处理。

采用自顶向下的递归方式进行语法翻译，部分递归课改为迭代。

/\*-------宏定义------\*/

#define SYNTAX\_ELE\_NUM 40

#define SYNTAX\_ERROR\_NUM 40

int printWordIndex = 0; // 打印单词时使用的下标

int syntaxWordNo = 0; // 分析的单词个数

SYM nextSym; // 下一个单词的类别码

Token\_Sym token\_sym[SYNTAX\_WORD]; // 存储识别完成的词法

map <KEY, SYMBOLTABLE> symbolTable; // 定义符号表

const string syntaxWord[SYNTAX\_ELE\_NUM] = { // 语法成分

"字符串", "程序", "常量说明", "常量定义", "变量说明", "变量定义", "声明头部",6

"无符号整数", "整数", "有返回值函数定义", "无返回值函数定义", "复合语句",11

"参数表", "主函数", "表达式", "项", "因子", "语句", "赋值语句", "条件语句",19

"条件", "循环语句", "步长", "有返回值函数调用语句", "无返回值函数调用语句",24

"值参数表", "语句列", "读语句", "写语句", "返回语句"；29

};

const string syntaxError[SYNTAX\_ERROR\_NUM] = { // 语法错误标识码

"非法字符串", "非法程序", "非法常量说明", "非法常量定义", "非法变量说明", "非法变量定义", "非法声明头部",

"非法无符号整数", "非法整数", "非法有返回值函数定义", "非法无返回值函数定义", "非法复合语句",

"非法参数表", "非法主函数", "非法表达式", "非法项", "非法因子", "非法语句", "非法赋值语句", "非法条件语句",

"非法条件", "非法循环语句", "非法步长", "非法有返回值函数调用语句", "非法无返回值函数调用语句",

"非法值参数表", "非法语句列", "非法读语句", "非法写语句", "非法返回语句"

};

**四、语义分析：词法语法语义的错误处理**



自己添加的错误类别码

|  |  |
| --- | --- |
| 应为右大括号’}’ |  |
| 数组下标越界 |  |
| 赋值语句左右类型不匹配 |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

const string errorSym\_to\_o[SYNTAX\_ERROR\_NUM] = { // 错误类型对应类别码输出

"非法符号或不符合词法", "名字重定义", "未定义的名字", "函数参数个数不匹配", "函数参数类型不匹配", "条件判断中出现不合法的类型", "无返回值的函数存在不匹配的return语句",

"有返回值的函数缺少return语句或存在不匹配的return语句", "数组元素的下标只能是整型表达式", "不能改变常量的值", "应为分号", "应为右小括号)",

"应为右中括号]", "do-while应为语句中缺少while", "常量定义中=后面只能是整型或字符型常量"

};

enum KIND { // 变量，常量，参数，函数等种类

CON\_K, VAR\_K, PARA\_K, FUNC\_K

};

enum TYPE { // 值类型或者返回类型，RE\_T在语法分析的时候笼统代表有返回值函数

INT\_T, CHAR\_T, STRING\_T, INTARR\_T, CHARARR\_T, VOID\_T, RE\_T

};

enum ERRORSYM { // 错误类别码

a\_ErrorS, b\_ErrorS, c\_ErrorS, d\_ErrorS, e\_ErrorS, f\_ErrorS, g\_ErrorS, h\_ErrorS,

i\_ErrorS, j\_ErrorS, k\_ErrorS, l\_ErrorS, m\_ErrorS, n\_ErrorS, o\_ErrorS, my\_ErrorS

};

只有类型a为词法分析的错误处理，其余为语法、语义分析的错误处理。

1. 对单词类别结构体建立单词所在行数信息；
2. 对词法分析进行行数记录，保存到单词类别结构体，错误处理为a类型，此时在语法分析预读的时候，预读函数判断是否为合法的词法；存在问题：多次预读又退回，会导致多次输出
3. 语法上的错误；应该为哪个符号；语法结构缺省……
4. 构造符号表项类，建立全局和局部符号表，建立符号表管理类函数，语义错误分析，已定义、未定义，返回值类型参数类型对应……
5. 变量常量定义只出现全局和函数内局部，不考虑if嵌套中定义。
6. 条件语句中条件的值只能是0或1或非0，字符类型和字符串其他类型不可以。因此需要处理条件表达式返回类型。条件只有一个表达式的时候必须是整数类型，并且＜条件＞中的关系比较只能是整型之间比，不能是字符型。
7. int a; a=’c’;  // 错误  
      if((‘c’))  // 是合理的情况  
      if (‘a’>’b’)  // 不合理  
      int func()  
      {  
        return (‘c’);  这里需要报错  
      }

**五、中间代码四元式及目标代码MIPS生成**

sprintf将输出到字符数组buffer中，但字符数组没有大小限制，有可能溢出

int sprintf（ char \*buffer， const char \*format ［， argument，。。。］）;

可采用to\_string(int)将Int转化为string，stoi(string)将string转化为int，正负、0、有前导零都可以；

测试递归程序，如全排列，return语句在中间的那种，输出字符串不存在转义字符

**1.中间代码-四元式形式**

符号 操作数1 操作数2 结果

语义分析：生成相应的功能语句，进行计算

声明语句：填符号表并检查、根据声明类型分配存储

执行语句：按照功能生成代码

四元式代码格式要求：

1. 函数声明

源码形如：

int foo( int a, int b, int c, int d)

中间代码：

int foo()

para int a // 参数定义

para int b

para int c

para int d

1. 函数调用

源码形如：

i = tar(x,y)

中间代码： // 按照参数顺序计算

push x

push y

call tar

i = RET + 0

1. 函数返回

源码形如：

return (x); return;

中间代码：

ret x; ret;

1. 变量声明

源码形如：

int i, j;

中间代码（符号表信息输出，程序中可不生成真正的中间代码）：

var int i

var int j

1. 常数声明

源码形如：

const int c = 10

中间代码（符号表信息输出，程序中可不生成真正的中间代码）：

const int c = 10

1. 表达式、计算、赋值语句

源码形如：

x = a \* (b + c)

中间代码（可优化）：

t1 = b + c

t2 = a \* t1

x = t2 + 0

1. 条件判断

源码形如：

x == y

中间代码：

x == y

1. 条件或无条件跳转

中间代码：在这里所有的跳转都转化为与0的比较，把两个变量比较转化为一个变量left

GOTO LABEL1 //无条件跳转到LABEL1，用Jump直接跳转代替

BNZ LABEL1 //不满足条件跳转到LABEL1，实际条件判断为==，因为满足条件直接执行下一条语句

BZ LABEL1 //满足条件跳转到LABEL1，实际条件判断为！=

1. 带标号语句

中间代码：

Label\_1 :

x = a + b

1. 数组赋值或取值

源码形如：

a[i] = b \* c[j]

中间代码：

t1 = c[j] // 取值访问内存，需要单独一行

t2 = b \* t1 // 寄存器、临时变量运算

a[i] = t2 // 存入内存

1. 其他本文档未涉及到的语法现象，或者自行定义的四元式操作，原则上均按照“x = y op z”形式的中缀表达式进行表达。

四元式结构设计：

语句类型

/\* 1.值参传入 push x, push y

\* 2.调用函数 call add

\* 3.赋值语句 i = ret, i = t1 + 1, a[i] = b[i] \* j

\* 4.条件判断 x == y, x<=y

\* 5.纯标签Label1:

\* 6.跳转语句 jump label1, bnz label2 ...

\* 7.函数返回 ret x, ret

\* 8.函数声明 int x

\* 9.参数表 para x

\* 10.print "stringaaa", print 'c', print 123, print a

\* 11.read int a, read char c

\*/

enum ItmCodeType { // 中间代码类型

ValueParamDeliver, // 值参传递

FunctionCall,

AssignState,

Label,

FunctionDef,

ParamDef,

Jump,

BEZ,

BNZ,

BLZ,

BLEZ,

BGZ,

BGEZ,

ReadChar,

ReadInt,

PrintStr,

PrintChar,

PrintInt,

PrintId,

ReturnInt,

ReturnChar,

ReturnId,

ReturnEmpty,

EndProcedure

};

// 四元式结构体

struct FourYuanItem {

ItmCodeType type; // 项类型

VALUE\_TYPE valueType; // 参数值类型,print语句表达式输出时用

FUNC\_TYPE funcType; // 如果是函数，函数类型

string target; // 目标结果标识符Id

bool isTargetArr; // 目标结果是数组

bool isRightArr; // 等号右边是数组，数组取值

bool isPrintCharId; // 判断打印的是不是char类型的id

string left; // 左边操作数或者id，当是数组取值时存放数组id

string index1; // 数组1下标

string right; // 右边操作数或者id

string index2; // 数组2下标

char op; // 运算符

};

// 中缀表达式-->逆波兰(后缀表达式),栈中的项

struct PostfixItem {

VALUE\_TYPE type;

string str; // 字符串

int number; // 整数或者字符对应的ascii码

bool isCharVar; // 是否是char变量或char型数组的元素

bool isOperator; // 如果是char常量,判断是不是运算符

};

**动作程序**：

训环语法：

for (i = 0; i <= 5; i = i + 1) {

statement;

}

分析

string label1 = generateLabel(),labelEnd= generateLabel();

// label1为条件判断前，labelEnd为循环结尾，这里将步长中间代码保存，等到循环语句分析结束再放入中间代码中

i = 0

Label1:

T1 = i - 5

BGZ T1 LabelEnd

statement

i = i + 1

Jump Label1

labelEnd:

end loop

(优化后，将步长的中间代码放到循环结尾执行)

语法：

while (i >= 0) {

statement;

}

分析：

Label1:

BLZ i Label2

statement

Jump Laeb1

Label2:

end loop

语法：

do {

statement;

}while (i >= 0)

Label1:

statement

BGEZ i Label // 注意这里相反，大于等于0跳转到循环开头继续，否则往下退出循环

条件语法：

if (n == 0) return (0);

else if (n <= 1) return (1);

else if (n < 6) return (n);

else {

return (-1);

}

针对一个if语句讨论，一个else 讨论。

bne n 0 label2

statement

Jump label1

Label2:

else <>

label1:

**2.目标代码MIPS生成**

需要用到的MIPS指令序列：

#define MIPS\_INSTR\_NUM 40

const string reg[] = { // 用到的寄存器

t4~t9, sp, fp, a0, v0,v1

}

MIPSCODE {

/\* 加载指令 \*/ LB, LW,

/\* 存储指令 \*/ SB, SW,

/\* 运算指令 \*/ ADD, ADDU, ADDI, ADDIU, SUB, SUBU, MULT, MULTU, DIV, DIVU, AND, OR, XOR, NOR,

/\* 分支指令 \*/ BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ,

/\* 跳转指令 \*/ J, JAL, JALR, JR,

/\* 传输指令，寄存器间值的传递、读HI寄存器、写HI、读LO、写LO \*/ MOVE, MFHI, MFLO, MTHI, MTLO,

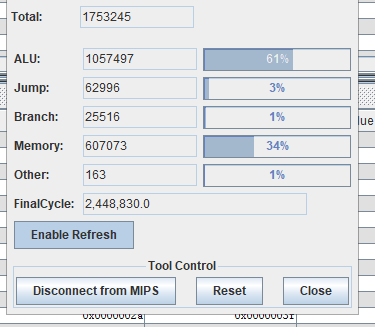
/\* 特权指令，异常返回、读CP0寄存器、写 CP0\*/ ERET, MFC0, MTC0,

/\* 陷阱指令，系统调用、断点异常 \*/ SYSCALL, BREAK

}

**六、代码优化处理**

优化前：



基本优化：对赋值语句，减少+0代码，可以减少ALU计算。

主要操作：在中间代码中找到连续的例如

T1 = a + 0

T2 = T1 + b类型的

删除T1语句直接让T2 = a + b

缩减表达式计算：

T= -1 + 2

直接计算用li指令代替

如果相连的后文用到了该寄存器，那就别保存在内存里，直接接着使用

减少跳转分支，减少if,for循环的跳转标签和跳转指令。

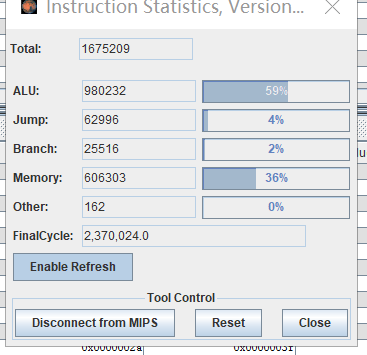
优化后的if语句：(竞速没有else语句)

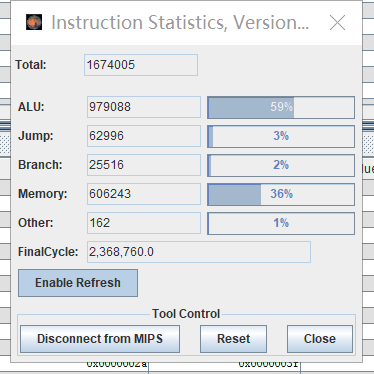
优化后的for循环语句：

mips寄存器的优化：

寄存器的使用，减少lw, sw访问内存的指令

优化后：

优化1：解决了多余+0情况

优化2：缩减表达式计算，T1=1+2

优化3：减少j指令，改写for循环分析的label