目录

# 1. 需求分析：

## 1.1 功能需求

**词法分析**：读取源文件，通过词法分析模块 生成 token 序列，用 C++的 vector 容器存储。

**语法分析** ：对 SYsy 文法的 EBNF 表达式进行递归下降，在递归下降的同时建立符号表，根据语义规则进行语法分析，同时生成中间代码。

**局部中间代码优化**：基本块划分 代数恒等式的优化 删除冗余临时变量 删除公共子表达式 局部常量传播。

**代码生成**：生成 mips 汇编，没有采用特殊的寄存器分配策略，使用 $t0-$t7,$s0-$s8 进行分配，如果使用到的中间变量多余全部可用的寄存器，则在栈中存取。

**错误处理**： 在其它模块中进行错误处理分析,在 error.cpp 中打印并提示错误信息

# 2.相关说明

## 2.1 开发环境和语言说明：

开发工具：visual studio 2019 IDE
开发环境：windows10
开发语言：C++
文法：Sysy
汇编代码：MIPS

## 2.2 使用说明：

在 visual studio 2019 下编译运行 sysy 工程文件，输出编译过程中的语法错误类型和行数。

```
Microsoft Visual Studio 调试控制台
The source file has benn successfully opened!

File name:      mips/0_global.txt

--------Lexial Analysis-------
The lexial analysis has done!
Total token unit:63

---------Grammatical analysis---------

That is the end of Grammatical Analysis!
Total error:0

---------print midcode----------
```

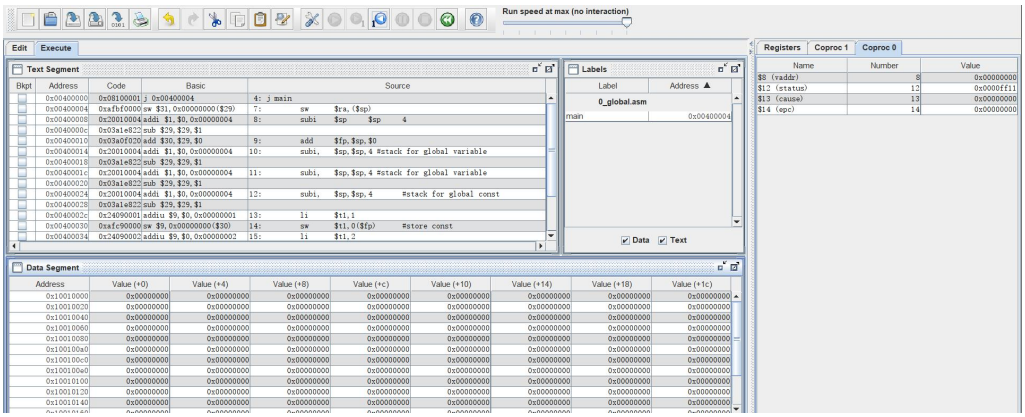
打印符号表信息：

```
---------function Symbol table---------
function name:      return type         parameter number    total offset

main                1                   0                   7


---------variable Symbol table---------
variable name:      field               address offest      size                value               kind

a                   main                0                   0                   -858993460          var
b                   main                1                   0                   -858993460          var
d                   main                2                   0                   -858993460          var
e                   main                3                   0                   -858993460          var
f                   main                4                   0                   -858993460          var
g                   main                5                   0                   -858993460          var
c                   main                6                   0                   1                   const
$1                  main                6                   0                   -858993460          temp
```

优化中间代码：

```
-------after-------

1  : func           int                         main
2  : var            int                         a
3  : var            int                         b
4  : var            int                         d
5  : var            int                         e
6  : var            int                         f
7  : var            int                         g
8  : const          int            1            c
9  : =              1                           a
10 : =              2                           b
11 : =              3                           d
12 : =              4                           e
13 : =              5                           f
14 : =              6                           g
15 : =              3                           c
16 : ret                                        0
17 : end_func
```

生成 mips 汇编，在 mars 软件中仿真运行。



## 2.3 代码规模

**总代码量：**

3946 行(无注释和空行)

5709 行(有注释和空行)

**模块代码量：**

| 模块 | 代码量 |
| --- | --- |
| main.cpp | 60 |
| lexial.cpp | 469 |
| grammer.cpp | 1812 |
| opt.cpp | 571 |
| tomips.cpp | 1633 |

## 2.4 优化和寄存器分配策略

优化：基本块划分 代数恒等式的优化 删除冗余临时变量 删除公共子表达式 局部常量传播。

寄存器分配：没有采用特殊的寄存器分配策略，使用$t0-$t7,$s0-$s8 进行分配，如果使用到的中间变量多余全部可用的寄存器，则在栈中存取。

# 3.总体设计

给出了编译器各个需求及其模块划分，数据传递和模块调用情况。
模块的具体实现将在详细设计中进行说明。
因为所有的模块都要调用错误处理模块，因此在错误处理模块中详细说明编译各个阶段的错误信息，而不是在其他模块内进行说明。

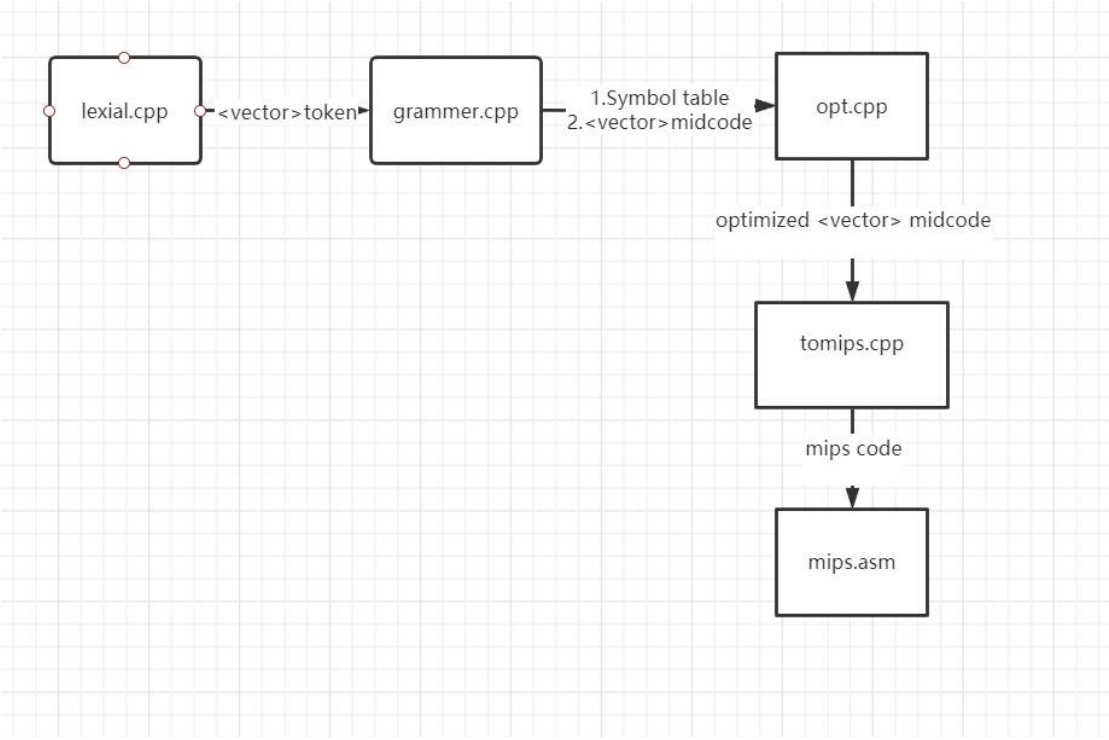
## 3.1 模块划分：

| 模块 | 功能需求 |
| --- | --- |
| main.cpp | 顶层模块 |
| lexial.cpp | 词法分析 |
| grammer.cpp | 语法分析，符号表，语义检查，中间代码生成 |
| opt.cpp | 全局优化，局部优化 |
| tomips.cpp | 生产中间代码。寄存器分配优化 |
| error.cpp | 错误处理 |

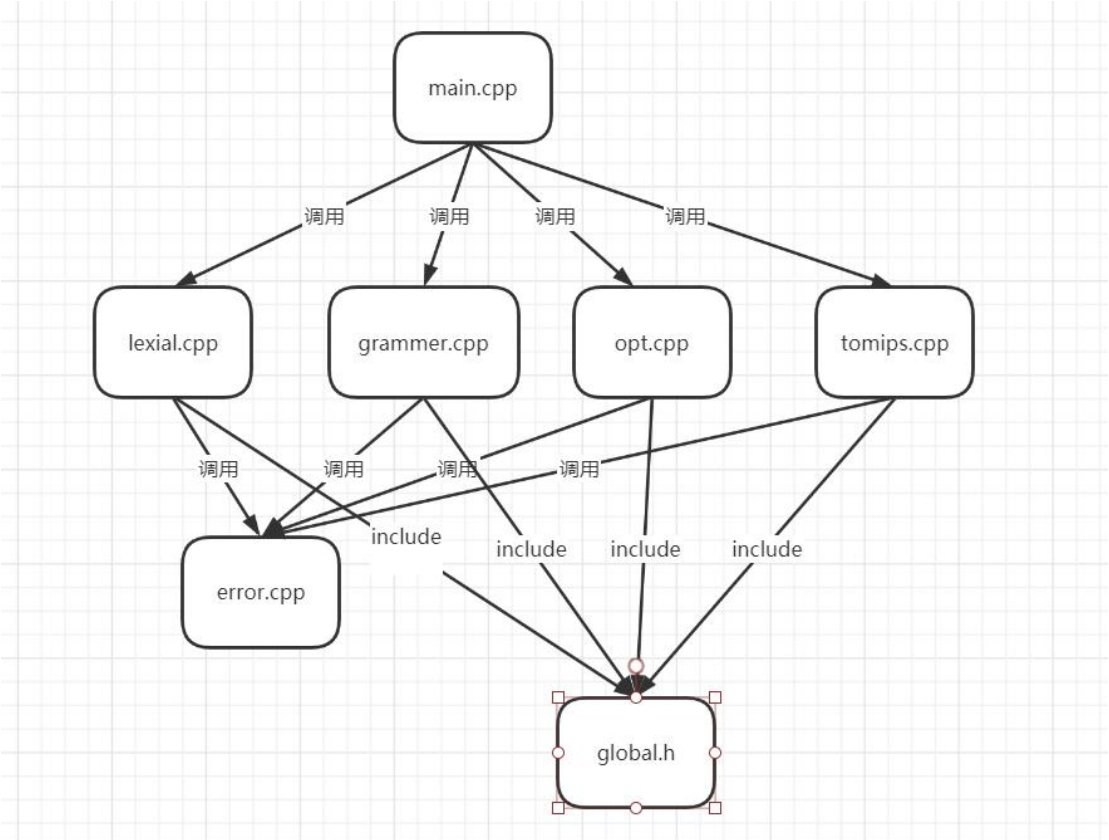
## 3.2 模块间传递的数据：

| 模块 | 传递的数据 | 说明 |
| --- | --- | --- |
| main.cpp | String filename | 在 main 函数中输入要编译的源文件 |
| lexial.cpp | <vector> token_vector | 将词法分析的 token 序列保存在 vector 中 |
| grammer.cpp | <vector> midcode_vector<br>Symboltable table | 在递归下降过程中进行语法分析<br>符号表信息保存在 table 中<br>中间代码序列保存在 vector 中 |
| opt.cpp | <vector> midcode_vector | 优化后的中间代码序列<br>在原先的基础上进行更新和修改 |
| tomips.cpp | mips.asm | 生成的 mips 汇编代码 |

模块与模块之间传递数据结构，达到了标记耦合，模块设计的独立性较好


lexial.cpp
<vector>token
grammer.cpp
1.Symbol table
2.<vector>midcode
opt.cpp
optimized <vector> midcode
tomips.cpp
mips code
mips.asm

## 3.3 模块调用关系：


main.cpp
调用
调用
调用
调用
lexial.cpp
grammer.cpp
opt.cpp
tomips.cpp
调用
调用
调用
调用
include
include
include
include
error.cpp
global.h

# 4.词法分析：

## 4.1 数据结构：

**Token**

| 数据 | 说明 |
| --- | --- |
| Tokentype | 每个 token 序列的类型 |
| row | 当前 token 序列所在的行数 |
| word | 当前 token 序列的词法单元值 |

说明：

使用如下结构定义每个 token 的类型，row 用于在错误处理输出错误词法的代码行数，用 C++风格的字符串定义，在扫描的时候存储每个 token 的所有字符。Type 是每个 token 的种类。

**TokenType（使用枚举类型表示）**

```
typedef enum
{
    CONST, INT, KWCHAR, VOID, MAIN, KWSTRING,
    WHILE, IF, ELSE, BREAK, CONTINUE, RETURN,
    SCANF, PRINTF,
    ID, NUM,
    // CHAR, STRING,
    ARTHMATIC,
    ASSIGN,
    NONE,//用于no_get_next
}TokenType;
```

有关键字，标识符，赋值符号和运算符号。

**token_vector:**

将得到的 token 序列存储到 C++的 Vector 容器中，得到词法分析的结果。

## 4.3 函数说明：

| 函数 | 说明 |
| --- | --- |
| void read_file(string filename); | 读取源文件 |
| void lexial_test(); | 词法分析模块的单元测试 |
| void read_file(string filename); | 读取源文件 |
| int is_valid_character(); | 是否为合法字符 |
| int is_unerline(char c); | 判断当前字符是否为下划线 |
| char no_get_next_char(); | 查看下一字符序列，但不消耗 |
| char get_next_char(); | 获得下一个字符序列，消耗一个序列 |
| int is_letter(char c); | 判断当前 char 是否为字母 |

## 4.4 使用方法：

具体测试结果见测试报告。

## 4.5 设计思路和问题解决方法：

词法分析模块的实现比较简单，功能需求也很好理解，就是分析源程序代码中的字符流，根据读入的字符流，组成 token 序列送入下一阶段的语法分析器。

实现的方法参考了 tiny 的编译器，整体思路就是通过读取源文件的函数读取并分析每一个字符序列直到文件 EOF 标识，其中要注意'\n'的处理以及记录代码行数（用于错误处理的定位）。分析字符流，通过向前看的操作和获取下一字符进行分析。

在识别 token 类别的时候，当语法分析器已经分析到 token 序列末尾的时候，会返回一个空的 token，因此,TokenType 枚举类型还要增加 NONE 的类型来适应这种情况。

词法分析的模块实现起来比较简单，主要参考的就是 tiny 的词法分析实现，通过循环读取字符的方式进行词法单元的识别，没有用到 DFA，即可对词法单元做出有效的识别。

具体的测试方法和结果见测试报告。

# 5 语法分析：

对 SYsy 文法的 EBNF 表达式进行递归下降，在递归下降的同时建立符号表，根据语义规则进行语法分析，同时生成中间代码。这也是整个项目中复杂度最高的一个模块。

5.1 文法说明

| | | | |
|---|---|---|---|
| 编译单元 | CompUnit | → | [ CompUnit ] ( Decl \| FuncDef ) |
| 声明 | Decl | → | ConstDecl \| VarDecl |
| 常量声明 | ConstDecl | → | **'const'** BType ConstDef { ',' ConstDef } ';' |
| 基本类型 | BType | → | **'int'** |
| 常数定义 | ConstDef | → | **Ident** { '[' ConstExp ']' } '=' ConstInitVal |
| 常量初值 | ConstInitVal | → | ConstExp |
| | | | \| '{' [ ConstInitVal { ',' ConstInitVal } ] '}' |
| 变量声明 | VarDecl | → | BType VarDef { ',' VarDef } ';' |
| 变量定义 | VarDef | → | **Ident** { '[' ConstExp ']' } |
| | | | \| **Ident** { '[' ConstExp ']' } '=' InitVal |
| 变量初值 | InitVal | → | Exp \| '{' [ InitVal { ',' InitVal } ] '}' |
| 函数定义 | FuncDef | → | FuncType **Ident** '(' [FuncFParams] ')' Block |
| 函数类型 | FuncType | → | **'void'** \| **'int'** |
| 函数形参表 | FuncFParams | → | FuncFParam { ',' FuncFParam } |
| 函数形参 | FuncFParam | → | BType **Ident** ['[' ']' { '[' Exp ']' }] |
| 语句块 | Block | → | '{' { BlockItem } '}' |
| 语句块项 | BlockItem | → | Decl \| Stmt |
| 语句 | Stmt | → | LVal '=' Exp ';' \| [Exp] ';' \| Block |
| | | | \| **'if'** '( Cond ')' Stmt [ **'else'** Stmt ] |
| | | | \| **'while'** '(' Cond ')' Stmt |
| | | | \| **'break'** ';' \| **'continue'** ';' |
| | | | \| **'return'** [Exp] ';' |
| 表达式 | Exp | → | AddExp　注：SysY 表达式是 int 型表达式 |
| 条件表达式 | Cond | → | LOrExp |
| 左值表达式 | LVal | → | **Ident** {'[' Exp ']'} |
| 基本表达式 | PrimaryExp | → | '(' Exp ')' \| LVal \| Number |
| 数值 | Number | → | **IntConst** |

| | | | |
|---|---|---|---|
| 一元表达式 | UnaryExp | → | PrimaryExp \| **Ident** '(' [FuncRParams] ')' |
| | | | \| UnaryOp UnaryExp |
| 单目运算符 | UnaryOp | → | '+' \| '−' \| '!'　注：'!'仅出现在条件表达式中 |
| 函数实参表 | FuncRParams | → | Exp { ',' Exp } |
| 乘除模表达式 | MulExp | → | UnaryExp \| MulExp ('*' \| '/' \| '%') UnaryExp |
| 加减表达式 | AddExp | → | MulExp \| AddExp ('+' \| '−') MulExp |
| 关系表达式 | RelExp | → | AddExp \| RelExp ('<' \| '>' \| '<=' \| '>=') AddExp |
| 相等性表达式 | EqExp | → | RelExp \| EqExp ('==' \| '!=') RelExp |
| 逻辑与表达式 | LAndExp | → | EqExp \| LAndExp '&&' EqExp |
| 逻辑或表达式 | LOrExp | → | LAndExp \| LOrExp '\|\|' LAndExp |
| 常量表达式 | ConstExp | → | AddExp　注：使用的 Ident 必须是常量 |

采用递归下降的方式进行词法分析，根据文法定义递归表达式

## 5.2 数据结构：

| 数据结构 | 说明 |
| --- | --- |
| token_vector | 词法分析得到的 token 序列用于语法分析，进行递归下降 |

## 5.3 函数：

| 函数 | 说明 |
| --- | --- |
| void compile_unit(); | 编译单元，语法分析模块的入口 |
| void get_next_token(); | 获取下一个 token 单元 |
| bool not_the_end(); | 判断 token 序列是否分析结束 |
| Token no_get_next_token(int i); | 向前看 i 个词法单元并返回 |
| bool match(string s); | 匹配关键字 |
| bool match_type(TokenType t); | 匹配变量名或者数字 |
| bool match_true(string s); | 匹配关键字，若不匹配则调用错误处理 |
| bool match_true_type(TokenType t); | 匹配变量名或数字，若不匹配则调用错误处理 |
| void assign_statement(); | 赋值语句 |
| void if_statement(); | 条件判断语句 |
| void statement(); | 语句 |
| void while_statement(); | 循环语句 |
| void break_statement(); | break 语句 |
| void continue_statement(); | continue 语句 |
| void return_statement(); | return 语句 |
| void block(); | block,sysy 的语句块 |
| void blockitem(); | 语句块内的内容 可进行声明，执行语句 |
| void call_function(); | 函数调用头 |
| void function_def(); | 函数定义头 |
| void function_parameters(); | 函数形参列表 |
| void function_parameter(); | 函数形参 |
| void int_function_def(); | int 返回类型函数定义 |
| void void_function_def(); | void 返回类型函数定义 |
| void main_func(); | main 函数定义 |
| void const_declare(); | 常量声明 |
| void variable_declare(); | 变量声明 |
| void varaible_def(); | 变量定义 |

| | |
|---|---|
| void const_def(); | 常量定义 |
| void expression(); | 表达式 |
| void term(); | 项 |
| void factor(); | 因子 |
| void condition(string ); | 条件判断（传递条件判断结果，放在临时变量中） |

## 5.4 表达式及优先级处理:

Sysy 的表达式优先级：

| 优先级 | 表达式符号 |
|---|---|
| 1 | () |
| 2 | *,/,% |
| 3 | +,- |
| 4 | <,>,<=,<= |
| 5 | ==,!+ |
| 6 | && |
| 7 | \|\| |

通过 expression,term 和 factor 三个递归下降函数进行优先级处理

根据 sysy 文法的语义规则：

一元表达式 UnaryExp → PrimaryExp | **Ident** '(' [FuncRParams] ')'
| UnaryOp UnaryExp
单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中
函数实参表 FuncRParams → Exp { ',' Exp }
乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp

从 expression-->term-->factor 的表达式优先级依次提高。

expression：处理 + ，-运算。

term: 处理 *,/,%运算。

factor：处理一元表达式和单目运算，包括简单类型变量，数组，函数调用，数字类型 token 的处理和一元运算。

expression,term,factor 三者在递归下降中的处理反映了运算的优先级关系，将每一步运算的结果保存在 exp_value 变量中，必要时新建临时变量用于存放运算结果。用于之后的中间代码生成。

```
: -          0          1          $3
: *          $3         8          $4
: +          $2         $4         $5
```

## 5.5 设计思路和问题解决方法：

在我的实现中，递归下降过程要解决的两个主要问题分别是：

1. 怎么对当前相同的 token 词法单元进行预测分析。
2. 如何进行错误处理

对于第一个问题，这里没有采用消除二义性和左递归进行文法修改，在原本的 Sysy 文法上，通过”向前看”的操作进行预测，实现的方法是通过函数 Token no_get_next_token(int i)，对后 k 个词法单元进行预测分析。

对于第二个问题，我对 match 函数做了四种不同的变形设计。

1. void match(string s);
2. void match_true(string s);
3. void match_type(TokenType t);
4. void match_type_true(TokenType t);

void match(string s)就是参考的 tiny 编译器的设计，对当前的 token 序列进行匹配。
void match_true(string s)涉及到错误处理，即如果无法匹配到下一个 token 序列，就调用错误处理模块。
而 match_type 是对符号的匹配，因为对于数字和标识符，无法预测到 token.word，只能通过 token 的 type 进行匹配，同时也要设计对应的错误处理方法 match_type_true(TokenType t)。

## 5.6.使用方法：

通过 main 函数的 compile_unit()调用语法分析模块，对于语法分析阶段出现的错误，输出错误的行数和类型。当编译错误为 0 的时候，根据递归下降过程中生成的中间代码输出对应的 MIPS 汇编。具体的测试方法和结果见测试报告。

# 6 符号表和语义分析：

## 6.1 数据结构：

**variable_symbol：符号项**

| 数据 | 说明 |
| --- | --- |
| string function | 符号所在的函数名称 |
| string name | 符号名称 |
| string kind | 符号类别，分别是<br>const 常量<br>var 变量<br>para 参数<br>temp 临时变量 |
| int value | 常量的数值，如果其他类型则是一个随机值 |
| int addr | 在函数内部的偏移，用于在运行时栈开辟大小 |
| int size | 普通类型的 size=0，若是数组类型，size 为数组的大小 |

**function_symbol:函数项**

| 数据 | 说明 |
| --- | --- |
| string name | 函数名 |
| int depth | 嵌套深度 |
| bool is_return | 返回类型<br>0 代表 void 函数<br>1 代表 int 函数 |
| int parameter_num | 函数参数个数 |
| int total_offset | 函数全部的偏移，用于在运行时栈开辟大小。 |

**symbol table:**

| 数据 | 说明 |
| --- | --- |
| std::map<std::string, int> index_var | 获取符号的索引 |
| std::vector<function_symbol> All_function | 函数表 |
| std::vector<variable_symbol> All_variable; | 变量表 |

## 6.2 函数：

| 函数名 | 说明 |
| --- | --- |
| void init_table(); | 初始化符号表 |
| void check_and_insert_function(function_symbol symblo); | 向符号表插入一个函数符号 |
| void check_and_insert_variable(variable_symbol symbol); | 向符号表插入一个变量 |
| bool contain_function(string name); | 按名检索符号表是否包含函数 |
| function_symbol get_function(string name); | 返回一个函数符号 |
| bool contain_variable(string name); | 按名检索符号表是否包含一个变量 |
| variable_symbol check_and_get_variable(string name); | 返回一个函数。 |
| void Symbol_table_test(); | 单元测试，打印符号表 |

## 6.3 设计方法：

通过 C++的 map 容器，再每个符号中，通过：
“function_name”+var_name 的方法进行映射，得到一个符号的下标，以便在使用的时候查找和插入。
向量 all_function 存储了所有的函数名
向量 all_variable 存储了所有的变量名。
查找时使用函数名＋变量名的方式对符号进行定位查找，通过这种方式来实现作用域的定位。
如果是全局变量，则对应的符号的 key 的函数名称为 global。

使用方法：
void Symbol_table_test()函数调用，打印符号表。
函数表

```
---------function Symbol table---------
function name:      return type         parameter number    total offset

returntest          1                   2                   105

rename              0                   1                   4

c                   0                   2                   16

ccc                 1                   2                   107

test123             0                   2                   10
```

打印出所有的函数，返回类型，参数数量，以及总偏移大小

变量表：

```
--------variable Symbol table---------
variable name:      field               address offest      size            value           kind

x1                  returntest          0                   0               -858993460      para
x2                  returntest          1                   0               -858993460      para
a                   returntest          2                   100             -858993460      var
b                   returntest          102                 0               -858993460      var
```

打印变量名，作用域（函数），地址偏移，大小，初始值（只有常量有初值），类型

## 6.4 语义分析检查点：

| 检查点 | 是否通过 |
| --- | --- |
| main 函数：无参，返回类型为 int,程序入口点,输出返回结果。 | 通过 |
| 常量定义后必须初始化 | 通过 |
| 数组维度在编译时可以求到非负整数 | 通过 |
| 变量定义后可以初始化<br>变量定义后可以不初始化 | 通过 |
| sysy 运算优先级 | 通过（递归下降函数实现） |
| 非条件表达式中不可出现单目运算！ | 通过 |
| 全局变量和全局函数无法同名 | 通过 |
| 函数形参实参类型是否匹配 | 通过 |

## 6.5 问题和解决方法：

符号表的设计也是整个项目中的一个重要的地方。一开始我对符号表的理解也没有很深入，一开始设计的符号表也仅仅是记录了源代码中出现的函数和变量。直到在写代码生成的时候，出现的一些临时变量无法在内存中正确的找到位置，所以我才再一次修改了符号表。在符号的种类中加入了 kind=”temp”,表明是一个临时变量，在代码生成的过程中要同时将临时变量入栈。

# 7.中间代码生成

## 7.1 数据结构：

midcode

| 数据结构 | 说明 |
| --- | --- |
| string op | 操作码 |
| string arg1 | 操作数 1 |
| string arg2 | 操作数 2 |
| string aresult | 结果 |

<vector>midcode_vector

| 数据结构 | 说明 |
| --- | --- |
| <vector>midcode_vector | 递归下降过程得到的中间代码序列 |

## 7.2 函数：

| 函数 | 说明 |
| --- | --- |
| void gen(string op, string arg1, string arg2, string result); | 生成一条四元式 |
| std::string new_label(); | 生成一个新的 Label |
| std::string new_temp(); | 生成一个新的临时变量。 |

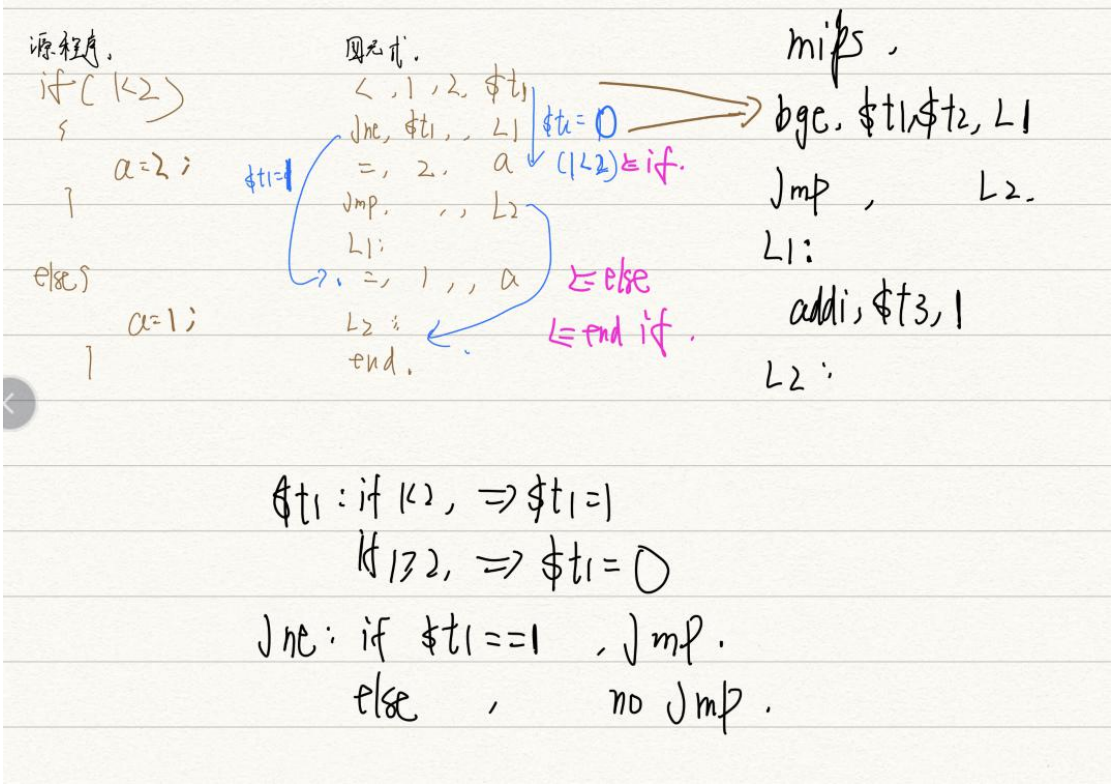
## 7.3 四元式翻译方案

| 代码类型 | op | arg1 | arg2 | result |
| --- | --- | --- | --- | --- |
| 变量定义 | var | int | | ID |
| 常量定义 | const | int | num | ID |
| 变量数组 | car | Int_array | size | ID |
| 常量数组定义 | const | Int_array | size | ID |
| 实参 | para | Int | | ID |
| Void 函数定义 | func | void | | ID |
| Int 函数定义 | func | int | | ID |

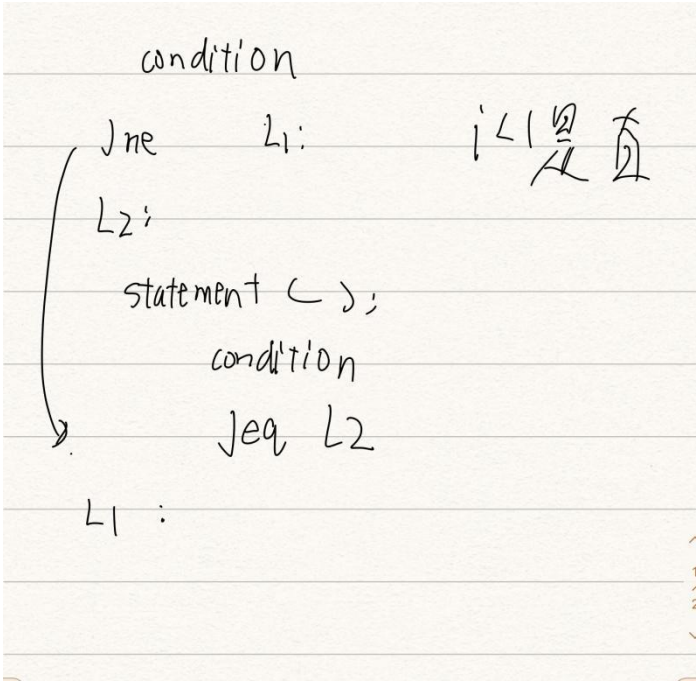
| | | | | |
|---|---|---|---|---|
| Main 函数定义 | func | void | | main |
| 函数调用 | void_call | ID | | |
| 返回值函数调用 | int_call | ID | | (Temp) |
| 函数定义结束 | end_func | | | |
| 函数返回 | ret | | | （Temp） |
| 算术运算 | op | val1 | val2 | result |
| 赋值 | = | val | | temp |
| 获取数组元素 | =[] | ID | index | temp |
| 数组元素复制 | []= | temp | index | ID |
| 逻辑运算 | op | val1 | val2 | temp |
| Label | L | | | label |
| 无条件跳转 | jmp | | | label |
| 条件跳转 | jne | temp | | label |
| 条件跳转 | jeq | temp | | label |

条件跳转：
Mips 对应的指令可以翻译，不用刻意记下 temp 的数值

If 语句的中间代码生成方案：




while 语句的中间代码生成方案：




## 7.4 问题和解决方法：

为了更好的生成汇编代码，四元式设计之后还进行过修改，导致整个编码工作量变大了

变量的全局信息和局部信息在 symbol table 中体现（通过函数作用域进行体现），四元式不表示变量的作用域。

对于中间变量的表示，采用符号”$”是因为如果采用其他字母进行开头的标号表示，可能会导致与定义的变量重名的情况发生。

关于嵌套的 while 循环 void condition(string )，因为 while 循环的实现需要两次条件判断，因此需要保存当前 midcode（条件判断的中间代码），因此通过一个栈进行存储当前的中间代码的条件判断。

```
stack<midcode> record;
```

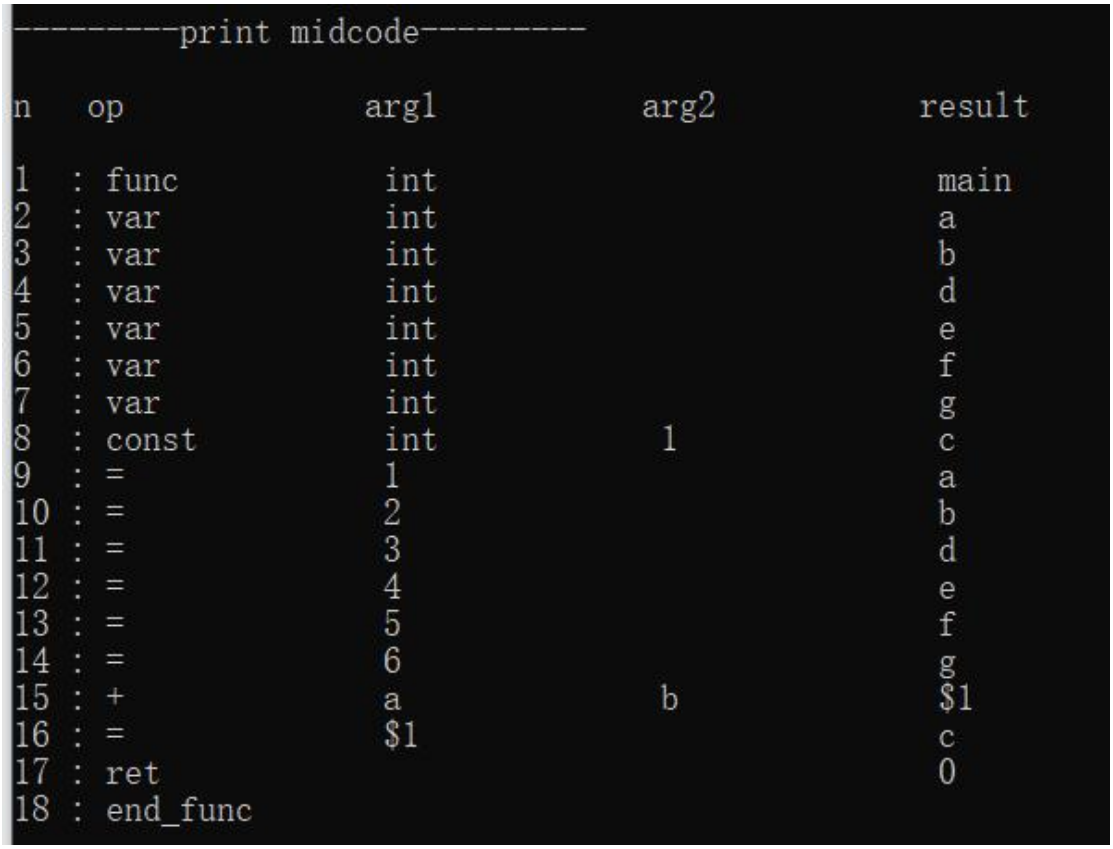
通过栈的 push，pop,top 操作进行存储。

函数参数：

利用符号表进行语义检查是否符合个数

函数声明的时候，形参不再单独生成四元式，而是通过向符号表的函数表插入函数的形参个数，向变量表插入函数形参，symbol.kind 设置成”para”

## 7.5 使用方法



```
---------print midcode---------

n   op              arg1            arg2            result

1  : func           int                              main
2  : var            int                              a
3  : var            int                              b
4  : var            int                              d
5  : var            int                              e
6  : var            int                              f
7  : var            int                              g
8  : const          int             1                c
9  : =              1                                a
10 : =              2                                b
11 : =              3                                d
12 : =              4                                e
13 : =              5                                f
14 : =              6                                g
15 : +              a               b                $1
16 : =              $1                               c
17 : ret                                             0
18 : end_func
```

控制台打印中间代码

# 8. MIPS 代码生成方案：

## 8.1 函数

```
void mips_to_text();
void midcode_to_mips();
void lw_mips(string register, string arg1);
void sw_mips(string register, string arg1);
int var_address(string name);
void ret_mips();
void int_call_mips();
void void_call_mips();
void para_mips();
void main_function_mips();
void algbra_mips();
void logic_mips();
void assign_mips();
void assign_array();
void equal_array();
void push_to_stack();
void jump_mips(string s);
void genmips(string s);
void function_mips();
void main_mips();
```

## 8.2 运行时栈和相关的语句翻译方案：

| |
|---|
| 参数 |
| 返回地址 |
| 保存的 fp 旧值 |
| 被调用函数的运行时栈<br>临时变量和数据 |
| .text |

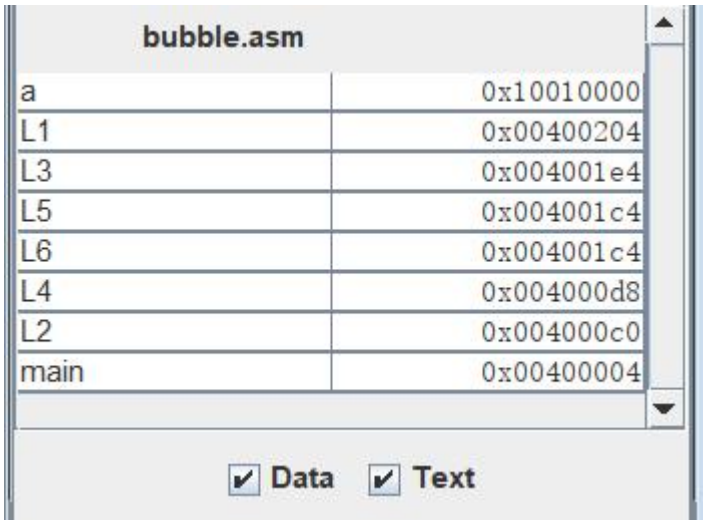
.data

.data:
存放输出字符串语句，全局变量和全局常量
.text:
存放代码段
通过 j main 指令跳转到 main 函数的代码段，开始执行 mips 汇编代码，然后开始在运行时栈分配存储空间。

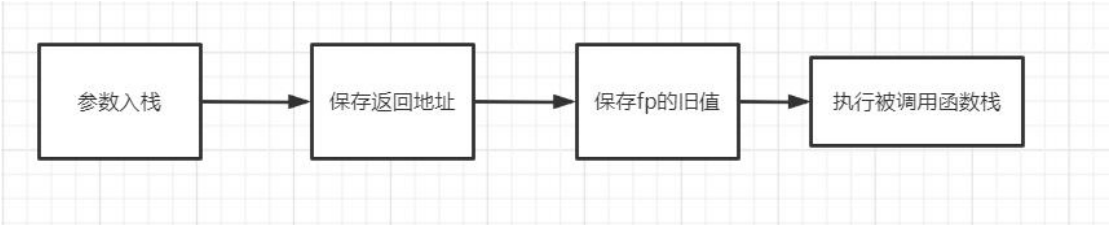
**编译阶段的符号：**
在递归下降过程中得到的符号表，在生成汇编代码时依次入栈

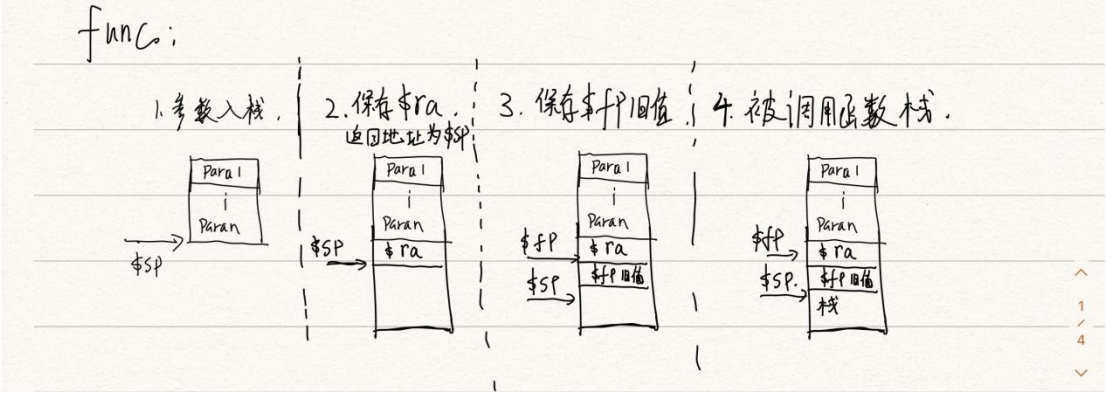
varname: .data word value
arrayname: .data word space
函数名和 label

将全局变量存入.data 段，（通过 la 指令对全局变量进行存取）
符号还包含函数名和 label 信息。



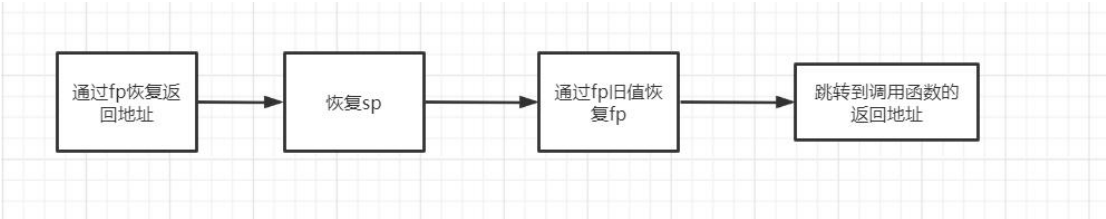

**func 的翻译流程：**

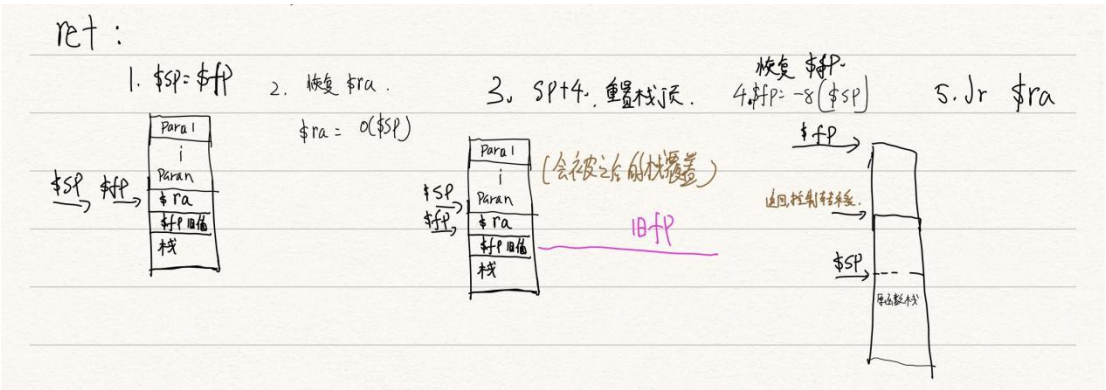





**ret 的翻译：**

**$sp=$fp，返回地址存入$ra**




## 8.3 指令选择：

| 指令 | 示例 | 注释 |
| --- | --- | --- |
| add | add $s1,$s2,$s3 | 加法,结果存放在$s1 |
| sub | sub $s1,$s2,$s3 | 减法，结果存放在$s1 |
| addi | addi $s1,$s1,1 | 立即数加 |
| subi | subi $s1,$s1,1 | 立即数减 |
| mul | mul $s1,$s2,$s3 | 乘法,结果存放在$s1 |

| | | |
|---|---|---|
| div | div $s1,$s2,$s3 | 除法,结果存放在$s1 |
| | | |
| beq | beq $s1,Ss2,L | 相等跳转 |
| bne | bne $s1,$s2,L | 不相等跳转 |
| j | J L | |
| jr | jr $ra | 跳转到返回地址 |

| 伪指令 | la | 注释 |
|---|---|---|
| la | la $t1,x | 将.data 段的变量 x 的地址存入$t1 |

全局变量的查找.
第一种实现方案是使用一个$gp 寄存器，将所有的全局变量存入运行栈，通过全局变量相对于 gp 的偏移找到地址，然后通过 lw 和 sw 完成内存和寄存器的交换。但这种方法使用的时候需要更加小心，有可能会覆盖到
伪指令 la 指令会在解释执行的之后翻译成一条 lui 指令和一条 ori 指令

```
lui $1, 0x00001001        11: la $t3, x
ori $11, $1, 0x00000000
```

**逻辑运算指令的翻译。**
以四元式(>,a,b,$t1)为例
需要对下一条四元式做查看。
如果下一条四元式是条件跳转指令 jeq(相等时跳转，即条件判断为真)，则说明通过逻辑运算之后进行了跳转。因此生成一条 bgt 指令，即当前四元式通过逻辑运算产生跳转的条件
如果下一条指令不是跳转指令，则逻辑运算后没有发生跳转，因此生成一条 ble 指令。

# 9.局部优化：

## 9.1 基本块划分：

采用对四元式遍历的方法得到基本块的划分。
局部数据流分析
因为基本块的性质有：
1 控制流只能从基本块中的第一条指令进入，因此，没有跳转到基本块中间的转移指令
2 除了基本块的最后一个指令，控制流在离开之前不会停机或者跳转
因此，采用遍历四元式的方法，找到一个函数的开始语句作为首指令，持续遍历中间代码，直到遇到跳转指令，标签指令，函数调用语句，函数返回语句作为基本块的结束。由此可以划分出不同的基本块。
在基本块内做优化。
采用的优化有：
代数恒等式优化，删除公共子表达式，常量传播，删除冗余临时变量，

## 9.2 代数恒等式的优化

将较为复杂的指令替换为简单的指令，
比如将加减乘除替换为赋值操作$t=a
当一个乘法含有操作数 0 的时候直接替换为$t=0 的赋值操作
对于四则运算的结果，如果出现的操作数都是常量，则对其进行合并。
优化结果如下：

| 操作 | 优化前 | 优化后 |
|---|---|---|
| 加法 | (+,a,0,$t1) | (=,a,,$t1) |
| | (+,0,a,$t1) | (=,a,,$t1) |
| 减法 | (-,a,0,$t1) | (=,a,0,$t1) |
| 乘法 | (*,2,a,$t1 | (+,a,a,$t1) |
| | (*,a,2,$t1) | (+,a,a,$t1) |
| | (*,1,a,$t1) | (=,a,,$t1) |
| | (*,a,1,$t1) | (=,a,,$t1) |
| | (*,0,a,$t1) | (=,0,,$t1) |

| | | |
|---|---|---|
| | (*,a,0,$t1) | (=,a,,$t1) |
| 除法 | (/,0,a,$t1) | (=,0,,$t1) |
| | (/,a,1,$t1) | (=,a,,$t1) |
| | (%,a,0,$t1) | (=,0,,$t1) |
| 常量合并 | (op,1,2,$t1) | (=,1+2,,$t1) |

## 9.3 删除公共子表达式

对于一个基本块中的局部公共子表达式(+,a,b,$t1)，从当前下标开始开始遍历四元式，在基本块内部循环遍历，找出具有相同操作符（+,-,*,/,=），相同左操作数，相同右操作数的操作符为+,-,*,/,=的四元式，如果找到一个四元式满足上述条件，且这两个四元式之间他们中间的任何变量都没有被修改，则可以确定这两个四元式为公共子表达式。
通过三个循环变量进行遍历

```
int i;//第一个四元式
int j;//第二个四元式
int k;//检查中间有没有被修改的四元式
```

优化结果如下：

| 优化前 | 优化后 |
|---|---|
| （+,a,b,$t1） | (+,a,b,$t1) |
| 没有对 a,b,t1 的修改 | 没有对 a,b,t1 的修改 |
| (+,a,b,$t3) | (=,$t1,,$t3) |

## 9.4 删除冗余临时变量

对于生成的冗余临时变量进行删除
优化结果如下：

| 操作 | 优化前 | 优化后 |
|---|---|---|
| 赋值 | (=,5,,$t1) | (=,5,,b) |
| | (=,$t1,,b) | |
| 算术运算 | (+,a,b,$t1) | (+,a,b,c) |
| | (=,$t1,,c) | |

## 9.5 局部常量传播

对于是常量的操作数进行优化，从而减少访存的时间，
修改的是第二个表达式的操作数，不对结果进行修改。
优化的结果如下

| 操作 | 优化前 | 优化后 |
|---|---|---|
| 基本类型 | (=,1,,a) | (=,1,,a) |
| | (=,1,,b) | (=,1,,b) |
| | (+,a,b,c) | (+,1,1,c) |
| 数组 | (=[],5,1,a) | (=[],5,1,a) |
| | (=,a,1,$t) | (=,a,1,$t1) |
| | (=,$t1,,b) | (=,$5,,) |

## 9.6 函数

```
void algebraic_identity(); //代数
void del_temp();
void del_common_expression();
void const_propogation();
void del_jump();
void gen_test_mide_code();
bool isin_basic_block(midcode m);
void local_opt();
void opt_test();
```

# 11. 寄存器分配策略：

没有采用特殊的寄存器分配策略
代码生成用到的 MIPS 寄存器组：

| 寄存器 | 说明 |
| --- | --- |
| $t0-$t7 | 变量寄存器 |
| $s0-$s7 | |
| $sp | 栈指针寄存器 |
| $ra | 返回地址寄存器 |
| $fp | 帧指针寄存器 |

使用空余的寄存器，在函数开始时初始化列表寄存器描述符。

依次从$t0-$t7,$s0-&s7 使用空余的寄存器，如果寄存器满，则不再进行寄存器分配，采用栈式存储。

**相关函数**

| 函数 |
| --- |
| string allocate_register(string temp); |
| void init_register(); |
| string find_in_register(string temp); |

# 12.错误处理

error.cpp 模块对错误进行处理
error(string s);
用于对编译过程中每一种错误类型定位，通过 token 数据结构的 row 变量输出对应的代码行数。

错误检查点：

| 错误检查点 |
| --- |
| 非法字符 |
| ；缺失 |
| 常量必须初始化 |
| 初始化的右值一定是一个常量 |
| 函数、变量错误 |
| 变量初始化错误 |
| 变量重命名 |
| ( )括号缺失 |
| 函数返回值错误 |
| 赋值语句错误 |
| 数组下标错误 |
| []括号缺失 |
| 函数调用错误 |

# 13.覆盖与测试和结果：

对于每个模块，都设计了多组测试用例，设计测试用例的目的是为了找出程序运行的 bug 以修改源代码。

针对可能出现的编程过程中的错误，对可能出现的程序错误设计广泛的覆盖

在本报告中，因为测试结果太过冗长仅给出覆盖情况，**具体的测试用例设计，测试结果，遗留的问题可以参见单独的测试报告（再提交的'\测试报告'目录中）。**

## 13.1 词法覆盖情况：

在词法分析单元中开始进行错误处理，针对非法格式的数字和非法格式的字符设计测试用例，最后测试一组可以通过词法分析器的测试用例，并加入行数用于错误处理。

仅当词法分析模块不产生错误的时候，可以进行接下来的语法分析。

| 序号 | 覆盖情况 | 测试结果 | 是否通过 |
|---|---|---|---|
| 1 | 形如 14a 的非法数字 | 报错 ✗ | 通过 |
| 2 | 形如【】；等非法字符 | 报错 ✗ | 通过 |
| 3 | 编写的可以通过的测试用例 | | 通过 |

## 13.2 语法、语义、中间代码覆盖情况：

在递归下降过程中进行代码生成，因此，只有当错误的数量为 0 的时候，生成的中间代码才会是正确的。

在调试过程中利用中间代码检查逻辑是否正确。

对于递归下降的过程，通过分析语法分析的错误提示信息（调用 error.cpp 模块），打印出的符号表以及中间代码进行测试。

| 功能 | 检查点 | 是否通过 |
|---|---|---|
| 函数定义 | 返回值检查 | 通过 |
| | 函数重命名 | 通过 |
| | 返回值 | 通过 |
| | main 函数声明和 return | 通过 |
| | 参数与局部变量命名 | 通过 |
| 变量和常量定义 | 变量重命名 | 通过 |
| | 常量未初始化 | 通过 |
| | 初始化错误 | 通过 |

| | | |
|---|---|---|
| 表达式 | 表达式优先级 | 通过 |
| 函数调用 | 参数传递，型参与实参匹配 | 通过 |
| | 返回值 | 通过 |
| 条件判断及循环 | if 语句 | 通过 |
| | while 语句 | 通过 |
| | break 语句 | 未通过 |
| 数组 | 数组下标检查 | 通过 |
| 多维数组 | 多维数组实现 | 未通过 |

## 13.3 优化覆盖情况：

在进行测试用例设计的时候，既要设计可以优化的用例，确保用例被优化。又要设计不能被优化的用例。

还有对于公共子表达式和常量传播的测试的时候，要生成中间的跳转指令，函数结束指令等语句对基本块的划分进行测试，以及修改公共子表达式的值，来检测判定是否正确。

| 测试项目 | 是否通过 |
|---|---|
| 代数恒等式优化 | 通过 |
| 删除冗余临时变量 | 通过 |
| 删除公共子表达式 | 通过 |
| 常量传播 | 通过 |
| 集成测试 | 通过 |

## 13.4 代码生成覆盖情况：

代码生成阶段，对中间代码进行翻译，生成 MIPS 汇编。

| 测试用例 | 输出结果 | 覆盖情况 | 是否通过 |
|---|---|---|---|
| 0_condition.cpp | 0_condition.asm | 条件表达式 | 通过 |
| 0_exp.cpp | 0_exp.asm | 表达式优先级 | 通过 |
| 0_function_return.cpp | 0_function_return.asm | 函数返回语句 | 通过 |
| 0_global.cpp | 0_global.asm | 全局变量常量 | 通过 |

| | | | |
|---|---|---|---|
| 0_variable.cpp | 0_variable.asm | 变量 | 通过 |
| 1_array.cpp | 1_array.asm | 数组 | 通过 |
| 1_array_function.cpp | 1_array_function.asm | 数组与函数 | 通过 |
| 1_while.cpp | 1_while.asm | while 语句 | 通过 |
| 1_nest_if.cpp | 1_nest_if.asm | 嵌套 if 语句 | 通过 |
| 1_nestwhile.cpp | 1_nestwhile.asm | 嵌套 while 语句 | 通过 |
| 1_integration.cpp | 1_integration.asm | | 通过 |

## 13.5 综合测试覆盖情况：

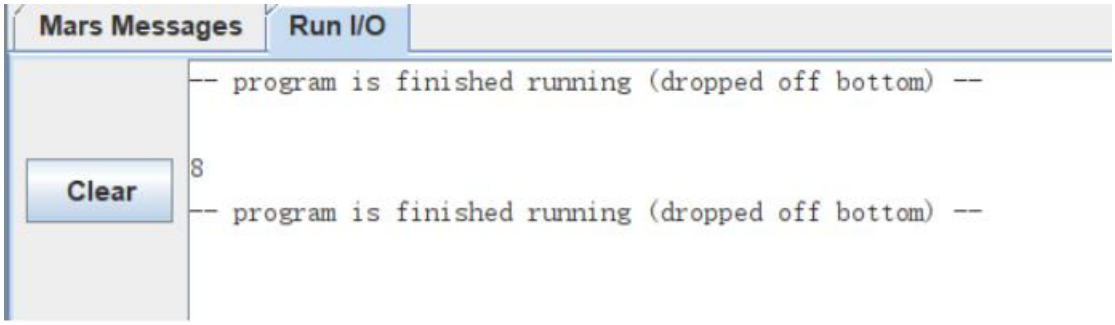
选用了自己设计的两个综合测试程序，递归斐波那契数列和冒泡排序，以及 sysyruntime 中的测试用例进行综合测试。

| 测试代码 | 是否通过 | 错误原因分析 |
|---|---|---|
| bubble.cpp | 通过 | |
| fib.cpp | 通过 | |
| conv0.sy | 未通过 | 数组作为函数参数没有实现<br>条件表达式短路没有实现 |
| fft0.sy | 未通过 | 数组作为函数参数没有实现<br>条件表达式短路没有实现<br>取模运算 |
| median0.sy | 未通过 | 数组作为函数参数没有实现<br>条件表达式短路没有实现 |
| shuffle0.sy | 未通过 | 数组维度可以是常量表达式 |
| transpose0.sy | 未通过 | 数组作为函数参数没有实现<br>条件表达式短路没有实现<br>初始化除法运算错误 |

## 13.6 测试结果：

具体的源代码、汇编代码生成太长，在测试报告中有进行详细分析。

斐波那契数列测试结果（n=5）：




冒泡排序测试结果：


# 14.产生的问题和解决方法：

**词法分析：**

在识别 token 类别的时候，当语法分析器已经分析到 token 序列末尾的时候，会返回一个空的 token，因此,TokenType 枚举类型还要增加 NONE 的类型来适应这种情况。

**语法分析、语义分析、中间代码生成：**

这一部分可以说是我整个项目中瓶颈期最长的时候。因为计划采用的是在递归下降的过程中进行语义分析和生成中间代码，因此这个模块的复杂度远远大于其他模块。

一开始参考 tiny 编译器，对递归下降的语法分析有了一个直接的认识，但是那本书里对错误处理的方法只是简单的概括，具体的实现却没有详细展开，龙书中的具体实现也没有很好的参考价值。

全局变量和局部变量的存取到运行时阶段来决定，符号表来存放变量的作用域，便于在代码生成阶段存放到相应的地址空间。递归下降过程中生成的中间代码不需要考虑变量的作用域信息（但一定要在符号表中进行保存）。一开始没有明白这一点，导致符号表设计和中间代码生成走了很多弯路。

还有就是函数参数的问题，函数声明的形参不生成中间代码，而是在符号表中记录函数的参数数量，通过符号表在语义分析阶段进行类型检查。函数实参需要通过表达式 expression 计算实参的值，并生成相应的中间代码 para,,$t1, 传递参数。

为了更好的生成汇编代码，四元式设计之后还进行过修改，导致整个编码工作量变大了

变量的全局信息和局部信息在 symbol table 中体现（通过函数作用域进行体现），四元式不表示变量的作用域。

对于中间变量的表示，采用符号”$”是因为如果采用其他字母进行开头的标号

表示，可能会导致与定义的变量重名的情况发生。

关于嵌套的 while 循环 void condition(string )，因为 while 循环的实现需要两次条件判断，因此需要保存当前 midcode（条件判断的中间代码），因此通过一个栈进行存储当前的中间代码的条件判断。

**代码生成方案：**

主要是运行时栈的概念，在阅读了多本参考资料《计算机组成与设计》、《深入理解计算机系统》、《编译原理》等书之后，才对程序的栈式存储有了深入的理解，因为计组实验做的是 MIPS 汇编，比较熟悉，因此这里采用 MIPS 汇编。

函数调用相关的语句翻译都在代码生成模块中进行了详细说明

# 15. 自我评价

这次的编译器课设是我目前写过代码量最大的项目了，写完之后感觉收获颇丰，一方面是第一次写大型的 C/C++项目，之前用 C/C++都是写一些算法和数据结构的题目，通过这样较大项目的开发，对 C/C++的一些关键字如 extern,static 有了更好的理解，也更清楚了代码在计算机的运行方式。另一方面，debug 能力有了很大的提升，在写语法分析模块的时候，代码量将近两千行，光是 debug 就花费了半个星期的时间，但完成之后收获很大。此外，也让我对计算机底层、栈室存储分配等知识有了更深的理解，虽然之前计算机组成原理和计算机系统基础都有接触底层相关的知识，但是还是要自己写过之后才有更深的理解。

对于整个编译器项目，也对设计的问题做一些总结：

1. 模块之间的耦合性不好，尤其是递归下降的实现，有很多重复性代码。
2. C++使用经验太少，导致 整个项目还是采用面向过程的思想进行设计的，封装性一般。
3. 因为一开始对编译器没有一个很好的理解，因此在初期没有一个很好的需求分析和总体设计，导致之后的代码在编写的过程中会经常修改之前的代码，大大加大了工作量。这个学期的《软件工程》课程也在强调这一点。我也会通过这次的 SysY 编译器项目吸取经验。