同济大学
TONGJI UNIVERSITY

同济大学编译原理课程设计
类C编译器实验报告

姓　　名　郑博远

学　　号　**2154312**

学　　院　电子与信息工程学院

专　　业　计算机科学与技术

授课老师　丁志军

日　　期　**2024 年 5 月**

目录

1. 实验概述

1.1. 实验目的

1. 掌握使用高级程序语言实现一遍完成的、简单语言的编译器的方法。

2. 掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法。

3. 掌握将生成代码写入文件的技术。

1.2. 实验要求

使用高级程序语言实现一个类 C 语言编译器，可以提供词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能。具体要求如下：

基本功能：

（1） 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。

（2） 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。

（3） 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。

（4） 要求输入类 C 语言源程序，输出中间代码表示的程序;

（5） 要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。

扩展功能：

（6） 实现过程、函数调用的代码编译。

（7）~~实现包含数组的中间代码以及目标代码生成~~（暂未实现）。

其中（1）（2）（3）（4）（5）是必做内容，（6）（7）是选作内容。

PS：int 宽度默认 4 字节，数组按行存储。

2. 需求分析

2.1. 程序输入

2.1.1. 源程序输入

程序从与 main.py 同目录下的 test.c 文件中读取待处理的源程序；同时，也可以在可视化窗口左侧的编辑器中输入或修改。程序输入范例如下：

```
int a;
int b;
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
        j=i;
    }
    return i;
}

int demo(int a)
{
    a=a+2;
    return a*2;
}
```

```
void main(void)
{
   int a;
   int b;
   int c;
   a=3;
   b=4;
   c=2;
   a=program(a,b,demo(c));
   return;
}
```

2.1.2. 文法输入

为了实现部分功能，添加 epsilon 转移进行简化，修改后的文法输入如下：

```
Program -> S DeclarationString
DeclarationString -> Declaration Declarations | Declaration
Declarations -> Declaration Declarations | Declaration
Declaration -> int identifier DeclarationType | float identifier
DeclarationType | void identifier P FunctionDeclaration
DeclarationType -> VarDeclaration | P FunctionDeclaration
VarDeclaration -> ;
FunctionDeclaration -> ( HeadVarStatements ) Block | ( ) Block |
( void ) Block
HeadVarStatements -> HeadVarStatements , VarStatement |
VarStatement
VarStatement -> int identifier | float identifier
Block -> { InnerStatement StatementString } | { InnerStatement }
| { StatementString }
InnerStatement -> VarStatement ; InnerVarStatements |
VarStatement ; | epsilon
InnerVarStatements -> VarStatement ; InnerVarStatements |
VarStatement ;
StatementString -> StatementString M Statement | Statement
Statement -> IfStatement | WhileStatement | ReturnStatement |
AssignStatement | FunctionCallStatement
AssignStatement -> identifier = RelopExpression ;
ReturnStatement -> return RelopExpression ; | return ;
WhileStatement -> while M ( RelopExpression A ) M Block
IfStatement -> if ( RelopExpression A ) M Block | if
( RelopExpression A ) M Block N else M Block
FactorExpression -> identifier | integer_constant |
```

```
floating_point_constant | ( RelopExpression ) | FactorExpression / FactorExpression | FactorExpression * FactorExpression | FunctionCallExpression
AddExpression -> AddExpression + AddExpression | AddExpression - AddExpression | FactorExpression
RelopExpression -> RelopExpression < RelopExpression | RelopExpression > RelopExpression | RelopExpression == RelopExpression | RelopExpression <= RelopExpression | RelopExpression >= RelopExpression | RelopExpression != RelopExpression | AddExpression
FunctionCallStatement -> FunctionCallExpression ;
FunctionCallExpression -> identifier ( Arguments ) | identifier ( )
Arguments -> Arguments , RelopExpression | RelopExpression
M -> epsilon
N -> epsilon
A -> epsilon
S -> epsilon
P -> epsilon
```

2.2. 程序输出

2.2.1. 目标程序输出

目标程序被输出到“code.asm”文件中，对应输出如下：

```
.data
    a: .word 0
    b: .word 0

.text
    lui $sp, 0x1004
    j main
program:
    sw $ra, 4($sp)
    li $s0, 0
    lw $s1, 12($sp)
    lw $s2, 16($sp)
    add $s3, $s1, $s2
```

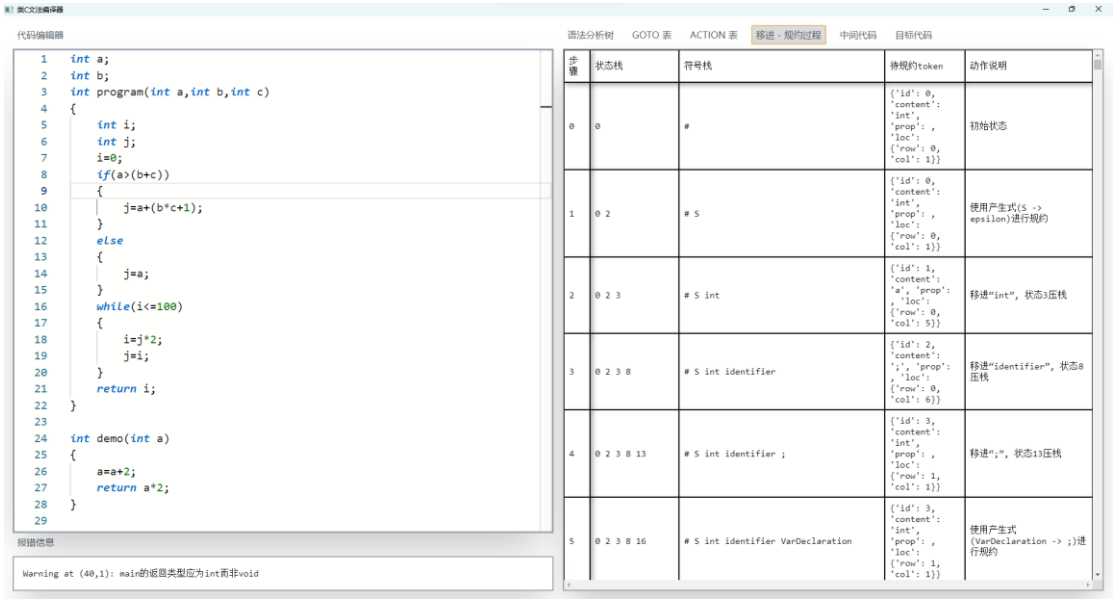
```
    lw $s4, 8($sp)
    sgt $s5, $s4, $s3
    sw $s0, 20($sp)
    bnez $s5, block2
block1:
    j block3
block2:
    lw $s0, 12($sp)
    lw $s1, 16($sp)
    mul $s0, $s0, $s1
    li $s1, 1
    add $s0, $s0, $s1
    lw $s1, 8($sp)
    add $s1, $s1, $s0
    sw $s1, 24($sp)
    j block4
block3:
    lw $s0, 8($sp)
    sw $s0, 24($sp)
block4:
    lw $s0, 20($sp)
    li $s1, 100
    sle $s2, $s0, $s1
    bnez $s2, block6
block5:
    j block7
block6:
    lw $s0, 24($sp)
    li $s1, 2
    mul $s0, $s0, $s1
    sw $s0, 20($sp)
    sw $s0, 24($sp)
    j block4
block7:
    lw $v0, 20($sp)
    lw $ra, 4($sp)
    jr $ra
```

```
demo:
    sw $ra, 4($sp)
    lw $s0, 8($sp)
    li $s1, 2
    add $s0, $s0, $s1
    li $s1, 2
    mul $s2, $s0, $s1
    add $v0, $s2, $zero
    lw $ra, 4($sp)
    jr $ra
main:
    li $s0, 3
    li $s1, 4
    li $s2, 2
    sw $s1, 4($sp)
    sw $s0, 0($sp)
    sw $s2, 20($sp)
    sw $sp, 12($sp)
    addi $sp, $sp, 12
    jal demo
    lw $sp, 0($sp)
    add $s2, $v0, $zero
    sw $s2, 12($sp)
block8:
    lw $s0, 0($sp)
    sw $s0, 24($sp)
    lw $s0, 4($sp)
    sw $s0, 28($sp)
    lw $s0, 12($sp)
    sw $s0, 32($sp)
    sw $sp, 16($sp)
    addi $sp, $sp, 16
    jal program
    lw $sp, 0($sp)
    add $s0, $v0, $zero
    sw $s0, 16($sp)
block9:
```

```
    lw $s0, 16($sp)
    j end
end:
```

2.2.2. 中间过程展示

包含语法分析树展示、action 表与 goto 表展示、移进规约过程展示以及中间代码展示，同时也会展示目标代码。



2.3. 程序功能

- 选择一个类 C 的源程序文件输入；
- 进行词法分析，输出词法分析的结果；
- 从文件读入文法基本信息，即文法的产生式集合等（文法的起始符号由产生式集合中第一个产生式的左边的文法符号），产生 LR（1）分析表 action 表和 goto 表；
- 根据之前词法分析的结果，测试该源程序文件是不是一个符合既定文法

的源程序。若符合既定文法则进行移进规约过程和显示最终的语法树，若不符合既定文法则给出错误提示；

· 在移进规约过程中，针对该程序做语义分析，并同步进行中间代码生成，若存在语义 warning 则会进行可视化展示，若存在语义 error，则会在进行可视化展示的同时终止中间代码的生成；

· 对四元式以函数为单位划分基本块，并计算每个基本块的出口活跃变量集合，并计算每条四元式的变量活跃信息与待用信息；

· 根据活跃信息与待用信息分配寄存器，生成目标代码。

2.4. 测试数据

测试数据涵盖正确、错误数据多种，详见后文测试部分。此处对错误测试数据作简单列举：

1. 词法错误：

```
int main(){
    编译原理;
}
```

2. 语法错误：

```
int main(){
    int a{

    }
}
```

3. 变量重定义：

```
int main(){
    int a;
```

```
    int a;
    a = 5;
    return 0;
}
```

4. 变量未定义：

```
int main(){
    a = 5;
    return 0;
}
```

5. 函数未定义：

```
int main(){
     a();
}
```

6. 缺少返回值：

```
int main(){
}
```

7. 函数调用与定义不匹配：

```
void a(){
}
int main(){
    a(5);
    return 0;
}
```

3. 概要设计

3.1. 任务分解

本次类 C 编译器的编译过程主要可以分解为以下 5 个步骤：**词法分析、语**

法分析、语义分析、中间代码生成以及**目标代码生成**。进一步，将目标代码生成器拆分出**基本块划分**这一步骤。此外根据任务的一遍扫描要求，词法分析作为子程序被语法分析调用，且采用语法制导的翻译技术进行语义分析。语法分析的过程中，语义分析、中间代码生成同步进行。

3.2. 数据类型定义

3.2.1. 词法分析器模块

DFA 状态：

重点函数/变量	说明
transfer	用于状态转移的字典
tokenType	状态所对应的 token 类型 (标识符，常量等)

DFA 自动机：

重点函数/变量	说明
start_state	DFA 的起始状态
cur	当前 DFA 所在的状态
len	当前输入长度
initAlpha()	为保留字和标识符添加对应的状态和转移
initDigit()	为整型、浮点型常量添加对应的状态与转移
initSymbol()	为所有的符号添加对应的状态和转移
initToken()	对 token 串添加对应的状态和转移，返回新增加状态集合
forward()	DFA 读入一个字符，转移至对应状态
reset()	重置 DFA 状态，从初始态重新开始

词法分析器：

重点函数/变量	说明
dfa	DFA 对象
lines	以行为字符串的代码
row	当前读到的行位置
col	当前读到的列位置
id	token 的序号
annotation	是否在读注释 /* */
token	上一次读到一半的 token
token_len	上一次读的 token 长度
getNextToken()	由 Parser 调用，每次得到一个 token

3.2.2. 语法分析器模块

产生式：

重点函数/变量	说明
id	在所有产生式中的编号
lhs	产生式左部
rhs	产生式右部 (list)
__lt__	比较函数，用于排序

拓广文法的 LR（1）项目：

重点函数/变量	说明
production_id	产生式编号
dot_pos	点的位置

forw	前瞻终结符
__lt__	比较函数，用于排序
__eq__	相等函数，用于比较
__hash__	哈希函数，用于比较

项目集的闭包：

重点函数/变量	说明
id	闭包编号
items	项目集
go	闭包的转移
__lt__	比较函数，用于排序
__eq__	相等函数，用于比较

语法分析器：

重点函数/变量	说明
productions	记录所有的产生式
terminals	终结符
none_terminals	非终结符，需从配置文件中读取
firsts	非终结符的 First 集
closures	项目集的闭包
goto_table	GOTO 表
action_table	ACTION 表
readProductions()	读取产生式配置文件
getFirsts()	构造每个非终结符的 First 集
getStrFirst()	辅助函数，计算串的 First 集合

getClosure()	给定项目集 I，计算闭包
getItemsets()	计算项目集族
getTable()	填充 GOTO 和 ACTION 表
getParse()	进行语法分析

3.2.3. 语义分析器模块

语法树节点的属性：

重点函数/变量	说明
type	值类型，如 int、float、word、tmp_word
place	如果是 word/tmp_word，则存放在此处
quad	下一条四元式位置
truelist	true 条件跳转目标
falselist	false 条件跳转目标
nextlist	顺序执行下一目标
queue	队列（用于函数参数）
has_return	是否有一个一定能执行到的 return （用于错误分析）

代码中声明的变量：

重点函数/变量	说明
id	变量的标识符
name	变量的名称
type	变量的类型
__eq__(other)	判断两个 Word 对象 或 Word 对象与字符串是否相等

四元式：

重点函数/变量	说明
op	操作符
src1	第一个操作数
src2	第二个操作数
tar	目标变量
info_src1	记录 src1 的活跃信息
info_src2	记录 src2 的活跃信息
info_tar	记录 tar 的活跃信息

代码中定义的函数：

重点函数/变量	说明
name	函数名
return_type	函数返回类型
actual_returns	函数实际返回值列表
start_address	函数起始地址
words_table	函数的变量表
param	函数参数列表

语义分析器：

重点函数/变量	说明
words_table	全局变量表
tmp_words_table	所有的临时变量表
process_table	函数的 Process 表

quaternion_table	四元式表
productions	从 Parser 获取到的所有产生式
start_address	四元式开始地址（默认为 100）
error_occur	是否出错
error_msg	错误信息列表
createProcess(start_address)	创建一个函数 Process 表项
createWord(word)	在当前进程符号表中创建 Word
getWord(word_name)	依次在当前进程符号表和全局变量符号表中查找 Word 并返回
getType(item)	根据 attribute 获取类型
getName(item)	获取名称（如果是常量，则是字面量；如果是 Word，则是对应的名称）
raiseError(type, loc, msg)	记录错误信息
analyse(production_id, loc, item, rhs_list)	进行一次语义分析

3.2.4. 基本块划分模块

基本块：

重点函数/变量	说明
name	基本块的名称
start_addr	基本块中第一个四元式的起始地址
codes	属于该基本块的四元式列表
next1	基本块的第一个后继基本块
next2	基本块的第二个后继基本块
use_set	在基本块中首次出现且是引用形式的变量集合
def_set	在基本块中首次出现且是定义形式的变量集合
in_set	基本块的入口活跃变量集合

out_set	基本块的出口活跃变量集合

四元式符号的活跃与待用信息：

重点函数/变量	说明
use	变量的使用位置
active	变量是否活跃

基本块划分器：

重点函数/变量	说明
quaternion_table	四元式表
start_address	四元式开始地址（默认为 100）
block_cnt	基本块计数器，用于分配唯一名称
func_blocks	函数名称到基本块列表的映射
getBlockName()	为新基本块生成唯一名称
divideBlocks(func_table)	根据函数表划分基本块
_is_var(name)	判断字符串是否为变量
computeBlocks(func_table)	计算基本块的活跃和待用信息

3.2.5. 目标代码生成模块

寄存器管理器：

重点函数/变量	说明
RValue	寄存器到变量的映射， 记录每个寄存器中存储的变量
AValue	变量到位置的映射， 记录每个变量存储的位置（寄存器或内存）

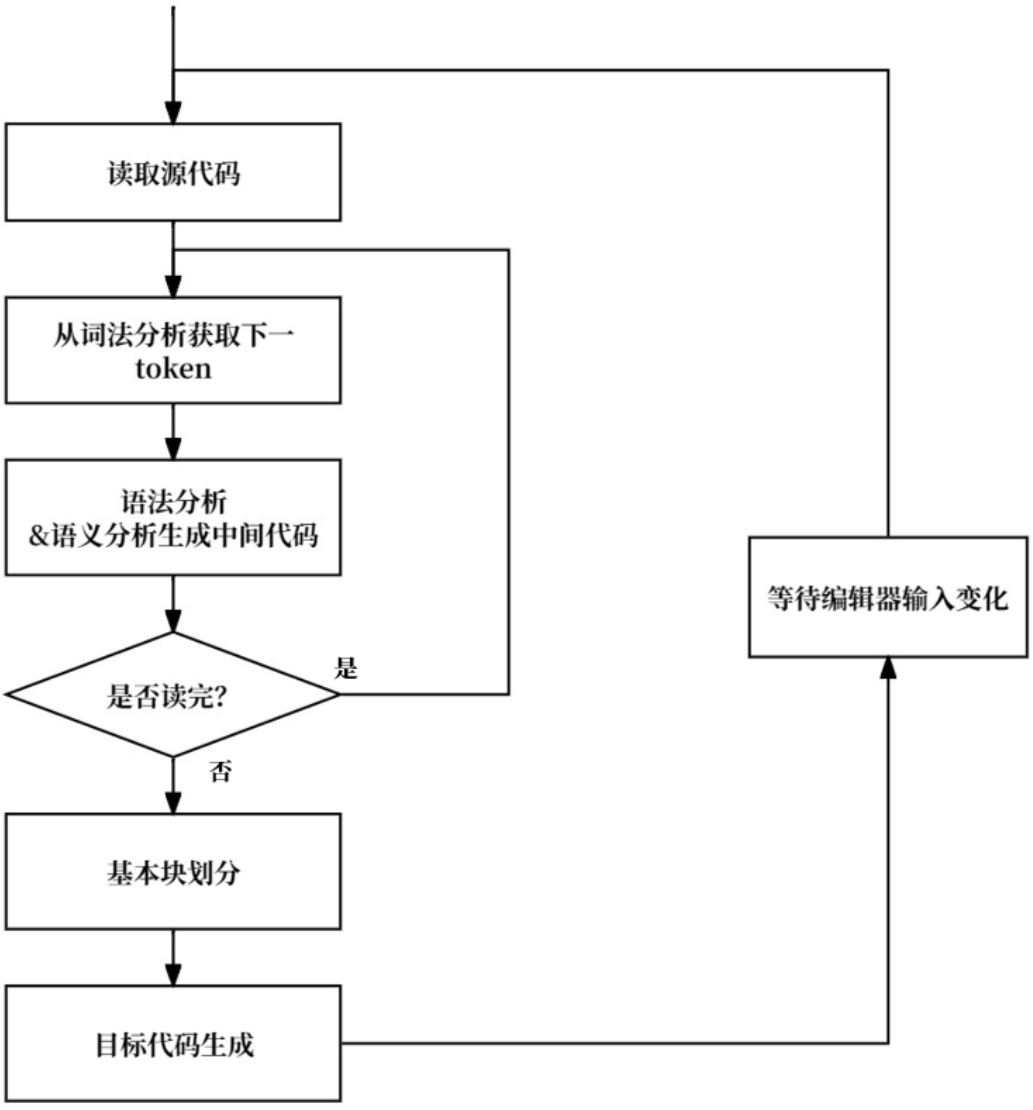
num_registers	寄存器的数量（默认为 8）
frame_size	当前栈帧的大小
free_registers	当前空闲的寄存器列表
memory	变量在内存中临时存储的地址
func_vars	当前函数的局部变量
data_vars	全局变量（数据段）
freeVarRegisters(var)	释放指定变量所占用的寄存器
freeAllRegisters(in_set)	清空所有寄存器 并将入口活跃变量标记为在内存中
storeVariable(var, reg, codes)	将指定变量存储到内存中
storeOutSet(out_set, codes)	将出口活跃变量存储到内存中
allocateFreeRegister(quars, cur_quar_index, out_set, codes)	分配一个新的寄存器
_is_variable(name)	判断字符串是否为变量
getSrcRegister(src, quars, cur_quar_index, out_set, codes)	为四元式的源操作数分配寄存器
getTarRegister(tar, quars, cur_quar_index, out_set, codes)	为四元式的目标操作数分配寄存器

目标代码生成器：

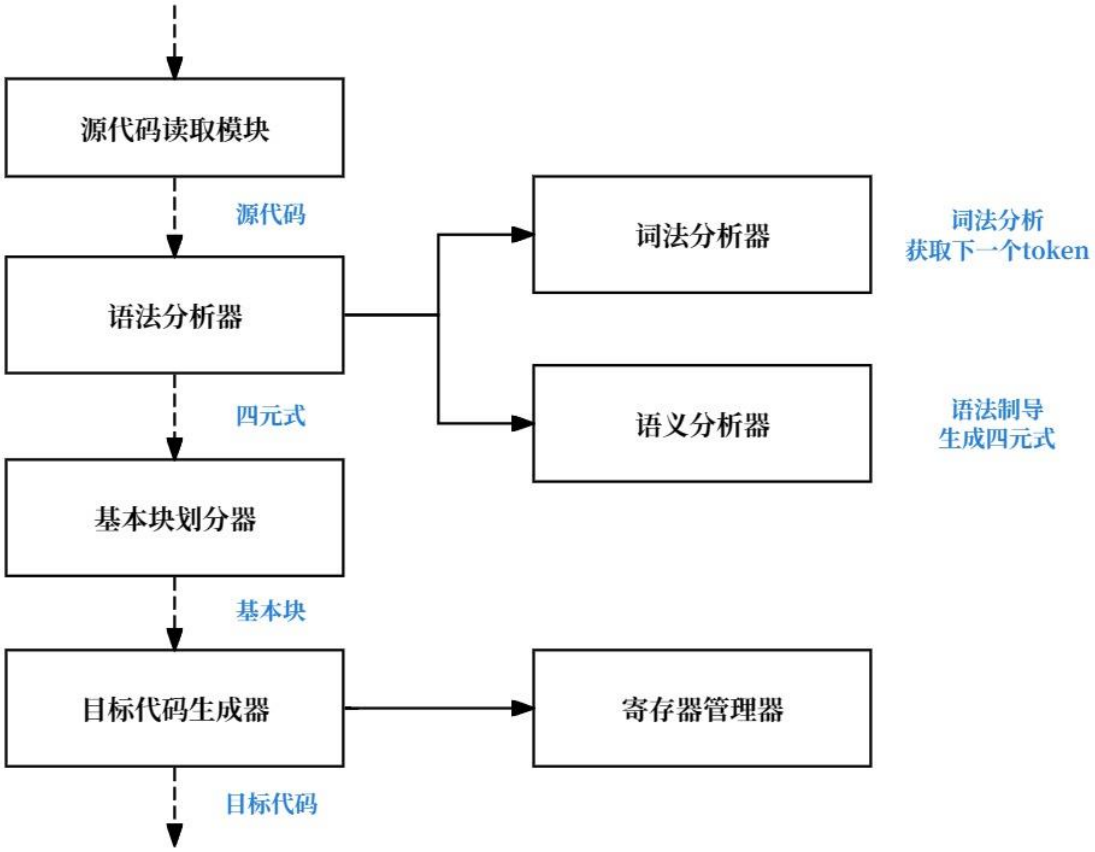
函数/变量	说明
func_blocks	函数名称到基本块列表的映射
func_words	各函数定义的局部变量
data_words	全局变量，存放在数据段
param_list	参数列表，在每个基本块开始时清空
regManager	寄存器管理器的实例
codes	最终生成的目标代码列表
error_occur	标记是否发生错误
error_msg	错误信息列表

getObjectCode()	获取整个程序的目标代码
getFuncObjectCode(func_name, blocks)	获取指定函数的目标代码
getBlockObjectCode(block, func_name)	获取指定基本块的目标代码
getQuarObjectCode(quad, qindex, block, func_name)	获取指定四元式的目标代码

3.3. 主程序流程图



3.4. 模块间调用关系

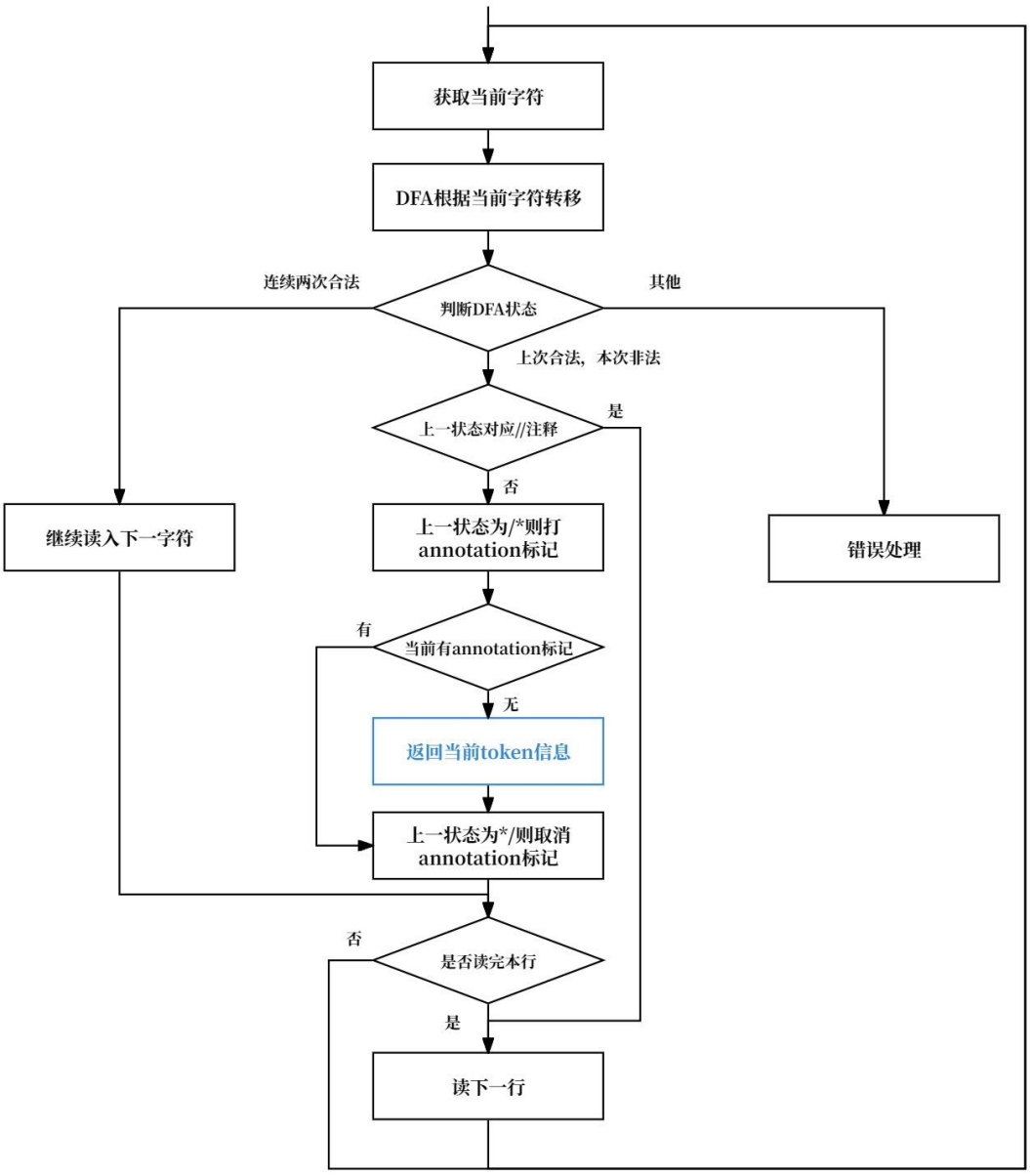

源代码读取模块
源代码
语法分析器
四元式
基本块划分器
基本块
目标代码生成器
目标代码
词法分析器
词法分析
获取下一个token
语义分析器
语法制导
生成四元式
寄存器管理器

4. 详细设计

4.1. 词法分析器

4.1.1. 词法分析流程图

本次词法分析程序在之前的基础上进行改进，不同点在于由于采用一边扫描，词法分析器作为语法分析器的子程序进行调用，因此一次只返回一个 token，而非一次性扫描生成 token 列表。具体流程如下：



4.1.2. 重点函数分析

1. 为给定单词添加状态与转移（initToken）：

该函数用于在 DFA 中为给定的单词（保留关键字，如 int、void 等）进行初始化。在函数内部，首先从 DFA 的根节点开始遍历输入单词的每个字符。对于每个字符，函数检查当前状态的转移集合中是否包含该字符，如果没有，则创建一个新的状态。在遍历过程中，将当前状态添加到一个集合中，以便最后返回。最终函数将输入单词的最后一个字符所在的状态标记为给定的标记类型。函数返回包含与输入单词中的字符对应的 DFA 状态的集合。

```
# 对 token 串添加对应的状态和转移，返回新增加的状态集合
def initToken(self, word: str, type):
    cur = self.start_state
    stateSet = set()
    for char in word:
        if char not in cur.transfer:
            cur.transfer[char] = DFA_state()
        cur = cur.transfer[char]
        stateSet.add(cur)
    cur.tokenType = type

    return stateSet
```

2. 初始化识别关键字和标识符（initAlpha）：

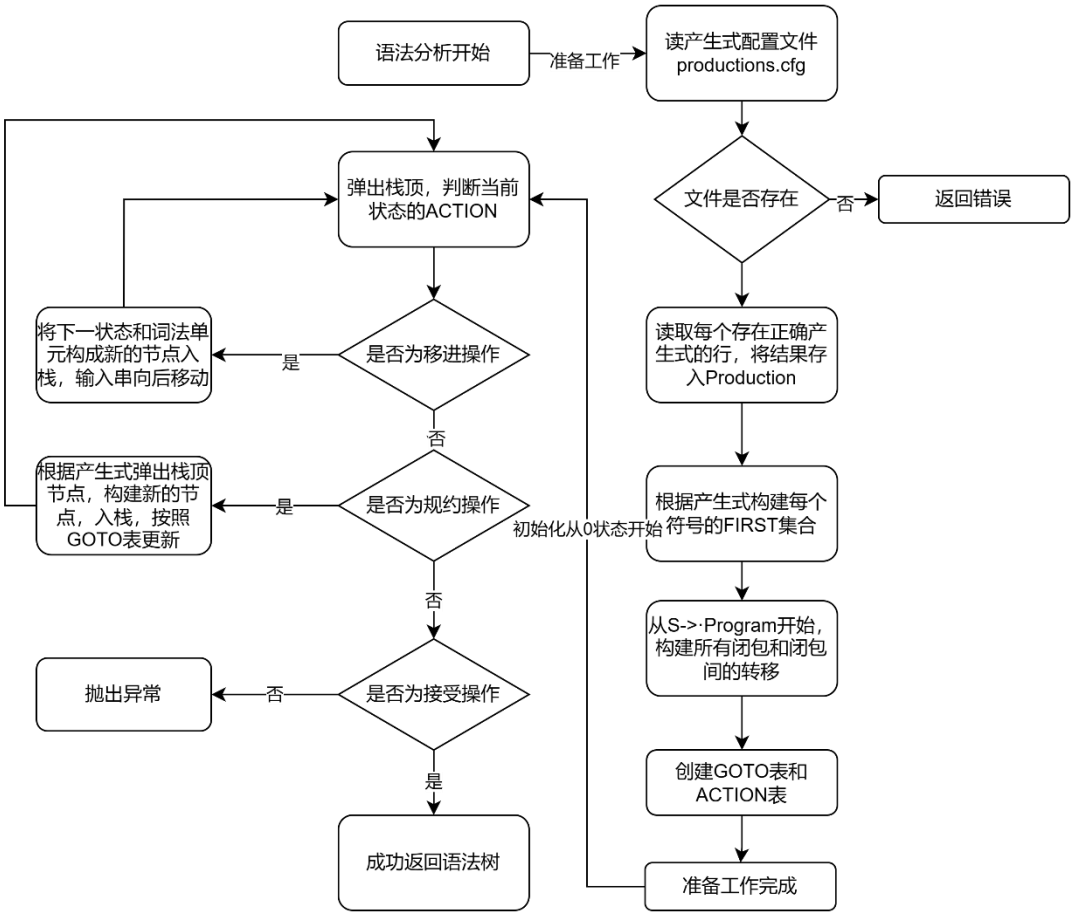
initAlpha 函数用于在 DFA 中初始化关键字和标识符。对于关键字，通过调用 initToken 函数，为每个关键字创建相应的 DFA 状态，并记录这些状态形成的集合。此外，还为标识符创建接受态 identifierState，并建立从初始态到 identifierState 的字母转移（标识符首字符必须是字母），并将所有该状态下的字母和数字的转移都指向本身。然后，遍历之前记录的关键字状态集合中的每个状态。若状态并非关键字接受态，则将其标记为标识符（例如对于关键字“else”，输入“els”应该识别为标识符）。此外，对于每个状态，检查其字母、数字转移集合。如果某个字母或数字在当前状态的转移集合中不存在，将其转移到 identifierState（例如对于关键字“else”，输入“els”后再输入“a”应当跳

转到标识符的接受态 identifierState）。

```
# 为保留字和 identifier 添加对应的状态和转移
def initAlpha(self):
    # 初始化所有的 keyword 对应的状态，并记录状态集合
    stateSet = set()
    for keyword in tokenKeywords:
        stateSet |= self.initToken(keyword,
tokenKeywords[keyword])

    # 处理剩余的字母、数字转移，应该识别为 identifier
    identifierState = DFA_state()
    identifierState.tokenType = tokenType.IDENTIFIER

    # 进入 identifier state 之后，再读入数字/字母都还是 identifier
    for letter in ascii_letters + digits:
        identifierState.transfer[letter] = identifierState

    # 从初始状态到 identifier state 必须是读入字母
    for alpha in ascii_letters:
        if alpha not in self.start_state.transfer:
            self.start_state.transfer[alpha] = identifierState

    for state in stateSet:
        # 不是单词末尾，则类型为 identifier
        # e.g. 关键字 else，如果只读入前三个字母应该是 identifier
        state.tokenType = (
            state.tokenType
            if state.tokenType != tokenType.UNKNOWN
            else tokenType.IDENTIFIER
        )

        # 其他转移应该回归到 identifier state
        # e.g. else 在读到 els 后，读入一个 a，应该作为 elsa 被识别为
identifier
        for alpha in ascii_letters:
            if alpha not in state.transfer:
                state.transfer[alpha] = identifierState
        for d in digits:
            state.transfer[d] = identifierState
```

4.2. 语法分析器

4.2.1. 语法分析流程图



4.2.2. 重点函数分析

1. （非）终结符 First 集的构造：

在开始构建之前，首先进行初始化：终结符的 First 集就是其自身，非终结符 First 集设置为空集。接着通过不断迭代更新的方式，逐步完善每个非终结符的 First 集。对于每个产生式，检查右侧的符号串，根据每个符号的类型（终结符或非终结符）来更新左侧符号的 First 集。如果遇到终结符，直接将其加入左侧符号的 First 集中；如果遇到非终结符，则将其 First 集中除了 ε 之外的部分加

入左侧符号的 First 集，并继续检查下一个符号。如果某个符号串最终能推导出 ε，则将 ε 也加入左侧符号的 First 集中。这样不断迭代直到没有新的 First 集合变化，便能得到完整的 First 集合。

```
# 构造每个（非）终结符的 First 集
def getFirsts(self):
    # 对于终结符，First 集就是本身
    self.firsts = {term: {term} for term in self.terminals}
    # 对于非终结符，初始化空集合
    for non_term in self.none_terminals:
        self.firsts[non_term] = set()
    while True:
        changeFlag = False      # 记录本次循环是否有 First 变化
        for prod in self.productions:
            lhs = prod.lhs
            for item in prod.rhs:
                # 如果是终结符（包含 epsilon）
                if item in self.terminals:
                    # 还没有添加到左侧的 First 集
                    if item not in self.firsts[lhs]:
                        self.firsts[lhs].add(item)
                        changeFlag = True
                    break   # 遇到了终结符，不必再往后看了
                else:
                    # 当前非终结符的 First 没被左侧完全涵盖 还得继续循环
                    if not self.firsts[item] - {"epsilon"} <=
self.firsts[lhs]:
                        changeFlag = True
                        self.firsts[lhs].update(self.firsts[item]
- {"epsilon"})
                    # 没有 epsilon，不必再往后看了
                    if "epsilon" not in self.firsts[item]:
                        break
                    # 遍历到产生式最右侧了 该加入 epsilon
                    elif item == prod.rhs[-1] and "epsilon" not
in self.firsts[lhs]:
                        changeFlag = True
                        self.firsts[lhs].add("epsilon")

        # First 集固定 不再循环
        if not changeFlag:
            break
```

2. 句子 First 集的构造

在构造了终结符与非终结符的 First 集合的基础上，可以计算给定句子的 First 集合。对输入的符号串进行遍历，针对每个符号进行以下操作：首先，将当前符号的 First 集（除了 ε）加入到 first 集合中；然后检查当前符号的 First 集合是否包含 ε。若不包含 ε，则后续的符号不再对当前串的 First 集合产生影响，停止继续遍历。此外，若当前符号是符号串的最后一个符号且 First 集包含 ε，则将 ε 加入到当前串的 First 集合中。具体算法如下：

```
# 辅助函数，计算句子的 first 集合
def getStrFirst(self, str_list: List[str]) -> Set[str]:
    first = set()
    for item in str_list:
        # 加入当前的 First 集（除 epsilon）
        first.update(self.firsts[item] - {"epsilon"})
        # 如果不含 epsilon，则不必要继续了
        if "epsilon" not in self.firsts[item]:
            break
        # 句子遍历完毕，加上 epsilon
        elif item == str_list[-1]:
            first.add("epsilon")
    return first
```

3. 闭包的构造

闭包是指对于项目集合中的每个项目，如果该项目中“ • ”所在位置（以下简称项目的位置）的下一个符号是非终结符，就将该非终结符对应的所有可能推导出的项目都添加到项目集合中。构造闭包需要遍历项目集合对每个项目进行分析。首先，若项目的位置已经在产生式的最右侧，那么该项目不会引入新的闭包，直接跳过；若下一个符号是终结符（包括 ε）也不会引入新的闭包，因此跳过。若下一个符号是非终结符，函数会找到对应该非终结符的所有可能产生的项目，并将它们加入闭包中。在加入闭包时，函数会构建新的项目，并考虑项目的推导序列，最终将所有新项目加入到项目集合中。

```
# 给定项目集 I，计算闭包
def getClosure(self, closure: Closure):
    for item in closure.items:
        # · 已经在最右侧，不会带来闭包增加
        if item.dot_pos == len(self.productions[item.production
_id].rhs):
            continue
        # 若是终结符（含 epsilon），则不会带来闭包增加
        elif self.productions[item.production_id].rhs[item.dot_
pos] in self.terminals:
            continue
        # · 的下一个是非终结符，找对应的项目加入
        else:
            next = self.productions[item.production_id].rhs[ite
m.dot_pos]
            for prod in self.productions:
                if prod.lhs != next:
                    continue
                else:
                    # 建立新的项目，加入闭包
                    str_list = [item for item in \
                                self.productions[item.productio
n_id].rhs[item.dot_pos+1:] + [item.forw] \
                                if item != "epsilon"]
                    forws = self.getStrFirst(str_list)
                    for forw in forws:
                        new_item = Item(prod.id, 0, forw)
                        if new_item not in closure.items:
                            closure.items.append(new_item)
```

4. 项目集族的构造

构造项目集族首先需对文法进行拓广，即添加新起始产生式 S'->Program。将其闭包加入到项目集族后，遍历项目集族中的每个闭包，检查是否存在新的转移符号，若存在则构建新的闭包，并将其与当前闭包之间的转移关系加入到项目集族中。在构建新的闭包时，函数会遍历当前闭包中的每个项目，并根据下一个符号与转移符号的匹配关系，生成新的项目，并将其加入到新的闭包中。

```
# 计算项目集族
```

```
def getItemsets(self):
    # 拓广文法，加上 S'->Program
    prod = Production()
    prod.id = len(self.productions)
    prod.lhs = "S'"
    prod.rhs = [self.none_terminals[0]]      # 即'Program'
    self.none_terminals.append("S'")
    self.productions.append(prod)

    # 创建项目：S'->.Program, #
    new_item = Item(prod.id, 0, "#")

    # 创建初始闭包以及项目集族
    root_closure = Closure()
    root_closure.id = 0
    root_closure.items.append(new_item)
    self.getClosure(root_closure)                    # 计算闭包
    self.closures.append(root_closure)               # 加入项目集族

    id = 0
    # 遍历每个闭包去看有没有新的转移带来新闭包的产生
    while id < len(self.closures):
        closure = self.closures[id]

        # 对于每个可能转移的文法符号 X 构建一个新的 closure，然后勾连
GO(I,X)
        for symbol in self.terminals + self.none_terminals:
            # 排除 epsilon
            if symbol == "epsilon":
                continue

            tmp_closure = Closure()

            # 遍历每一个项目
            for item in closure.items:
                # · 已经在最右侧，不会带来闭包增加
                if item.dot_pos ==
len(self.productions[item.production_id].rhs):
                    continue
                # 下一个符号就是当前文法符号
                elif symbol ==
self.productions[item.production_id].rhs[item.dot_pos]:
                    new_item =
Item(self.productions[item.production_id].id, item.dot_pos + 1,
item.forw)
```

```
                    tmp_closure.items.append(new_item)

            # 项目集非空
            if tmp_closure.items:
                # 计算闭包
                self.getClosure(tmp_closure)

                # 如果找到相同的闭包，则更新映射
                if tmp_closure in self.closures:
                    closure.go[symbol] =
self.closures.index(tmp_closure)
                else:
                    tmp_closure.id = len(self.closures)
                    self.closures.append(tmp_closure)
                    closure.go[symbol] = tmp_closure.id

        id += 1
```

5. 语法分析过程

由于采用了一遍扫描和语法分析制导，词法分析器和语义分析器都在语法分析过程中作为子程序调用。语法分析首先需要初始化用于移进-规约分析过程的栈，并压入初始状态。此后函数进入循环，不断地进行移进和规约操作，直到达到语法分析的结束条件（接受状态）。

在每次循环中，语法分析器首先根据当前栈顶状态和当前 token 查询 action 表，得到应该进行的操作（移进、规约、接受）。根据操作的不同，函数执行相应的移进或规约操作，并更新栈的状态。若为移进，则还需调用词法分析器得到下一个待分析的 token；若为规约，则处理栈状态更新后还需查 goto 表进行状态跳转；若为接受，则规约成功，返回对应的语法分析树。同时，函数会记录下语法分析过程中的各个步骤，包括步骤数、状态栈、符号栈、待规约 token 以及动作说明等，这些信息会通过前端进行可视化展示。

如果在分析过程中发现了语法错误，函数会设置相应的错误标志，并返回错误信息。当分析完成后，函数会返回分析得到的语法树（或者报告语法错误），同时也会存储语义分析器生成的四元式中间代码。

```
def getParse(self, codes):
    # 重新构造 lexer 和 semantic（reset 或许更好）
    self.lexer = Lexer(codes)
    self.semantic = Semantic(self.productions)

    stack = []           # 用于移进规约分析的栈
    stack.append({"state": 0, "tree": {"token": "#"}})

    last_loc = { "row":0, "col": 0 }     # 规约出错时应该报上一次的
last_loc，故记录
    cur = self.lexer.getNextToken()      # 当前规约的 token
    cnt = 0                              # 为了打印规约步骤的标号

    self.parse_process_display = []
    self.parse_process_display.append(["步骤", "状态栈", "符号栈
", "待规约 token", "动作说明"])
    self.parse_process_display.append(["0", "0", "#", f"{cur}",
"初始状态"])

    while True:
        cnt += 1
        cur_token = tokenType_to_terminal(cur["prop"])  # 根据
token 类型取出对应字符串
        cur_loc = cur["loc"]    # 当前分析到代码中的位置（用于报错）
        cur_content = cur["content"] # 当前的内容（即代码中字面量）

        action = self.action_table[stack[-1]["state"]]

        # 用于展示
        new_display_item = [None] * 5
        new_display_item[0] = str(cnt)

        if cur_token not in action:
            self.semantic_quaternation = '代码中包含 Error，中间代
码暂不可用'
            self.semantic_error_occur = True
            self.semantic_error_message = [f"Error at
({cur_loc['row']},{cur_loc['col']}): 代码不符合语法规则"]
            return {"token": "语法错误/代码不完整，无法解析",
"err": "parser_error"}

        # 移进
        if action[cur_token][0] == "s":
            next_state_id = action[cur_token][1]
            stack.append({"state": next_state_id, "tree":
```

```
{"token": cur_token, "content": cur_content, "children": []}})
            cur = self.lexer.getNextToken()  # 移动到下一个 token
            new_display_item[4] = f"移进“{cur_token}”, 状态
{next_state_id}压栈"
        # 规约
        elif action[cur_token][0] == "r":
            prod_id = action[cur_token][1]
            prod = self.productions[prod_id]

            children = []
            content = ""        # 当前节点对应的代码字面量
            for item in prod.rhs:
                if item == "epsilon":
                    continue
                else:
                    child = stack.pop()["tree"]
                    content = f"{child['content']}{content}"
                    children.insert(0, child)

            next_state_id = self.goto_table[stack[-
1]["state"]][prod.lhs]
            item = {"state": next_state_id, "tree": {"token":
prod.lhs, "content": content, "children": children}}
            self.semantic.analyse(prod_id, last_loc, item,
children)
            stack.append(item)
            new_display_item[4] = f"使用产生式({prod.lhs} -> {'
'.join(prod.rhs)})进行规约"
        # acc
        elif action[cur_token][0] == "acc":
            self.semantic_quaternation =
self.semantic.getQuaternationTable()
            self.semantic_error_occur =
self.semantic.error_occur
            self.semantic_error_message =
self.semantic.error_msg

            # 返回时不带上 content 和 attribute 这两个中间信息
            def removeRedundancy(dictionary):
                if isinstance(dictionary, dict):
                    if 'content' in dictionary:
                        del dictionary['content']
                    if 'attribute' in dictionary:
                        del dictionary['attribute']
                    for key, value in dictionary.items():
```

```
                        dictionary[key] =
removeRedundancy(value)
                elif isinstance(dictionary, list):
                    for i in range(len(dictionary)):
                        dictionary[i] =
removeRedundancy(dictionary[i])
                return dictionary
            return removeRedundancy(stack[-1]["tree"])

        last_loc = cur_loc

        state_stack = " ".join([str(item["state"]) for item in
stack])
        symbol_stack = " ".join([str(item["tree"]["token"]) for
item in stack])
        pending_token = f"{cur}"
        new_display_item[1:4] = state_stack, symbol_stack,
pending_token
        self.parse_process_display.append(new_display_item)
```

4.3. 语义分析器

4.3.1. 全局变量与局部变量

参照课件中保留作用域信息的部分修改文法，添加 ε 产生式：

```
Program -> S DeclarationString
DeclarationString -> Declaration Declarations | Declaration
Declarations -> Declaration Declarations | Declaration
Declaration -> int identifier DeclarationType | float
identifier DeclarationType | void identifier P
FunctionDeclaration
DeclarationType -> VarDeclaration | P FunctionDeclaration
VarDeclaration -> ;
S -> epsilon
P -> epsilon
```

对于每个过程，语义分析器会维护一个 Process 类，用一张 words_table 记录其中的局部变量（文法不支持过程嵌套）；此外，还有一张全局 words_table

记录全局变量。每个 Process 类的创建在规约产生式“P -> epsilon”时完成。这样，在此后的函数体语句的规约时，Process 类已经形成，可以修改或访问其中的 words_table。

4.3.2. 过程调用与函数返回

参照课件中对于过程调用部分的四元式生成，设计如下文法：

```
FunctionCallExpression -> identifier ( Arguments ) | identifier ( )
Arguments -> Arguments , RelopExpression | RelopExpression
```

在 Arguments 规约过程中，会记录一个 queue 属性，从而记录下函数调用部分传入的所有表达式（及其类型）。在规约成 FunctionCallExpression 时，会逐一生成“param”四元式。与课件直接“call 过程名”不同的是，我们修改了对函数调用“call”四元式的定义，改为了“(call, 被调用过程入口地址, _, 返回值存储的临时变量（void 则为空）)”这一格式。对于函数返回，由于没有找到课件中的相关样例，我们自行定义了“(ret, 返回值（void 则为空）, _, _)”的四元式格式。具体样例如下：

```
int sum(int a, int b){
  return a + b;
}
int main(){
  int a;
  a = sum(5, 7);
  return 0;
}
```

地址	四元式
100	(int+, a, b, T0)
101	(ret, T0, _, _)
102	(param, 5, _, _)
103	(param, 7, _, _)
104	(call, sum, _, T1)
105	(=, T1, _, a)
106	(ret, 0, _, _)

此外，在过程调用的部分还会检查调用与声明的变量类型、个数是否匹配，具体的错误分析将在后文中详细介绍。

对于 main 函数的入口跳转，在上学期的大作业中的处理方式是在四元式开

头插入一句无条件跳转指令。具体来说是进行了如下处理：在 Program 程序初始的“S -> epsilon”规约时，在空的四元式表中插入一句“(j, _, _, _)”无条件跳转语句。当规约到 main 函数的函数体时，再将 main 的起始地址回填到四元式表首的最后一个单元中，从而实现对 main 函数的跳转。**而本次实现则直接在目标代码生成时完成，因此取消了这一设计。**

4.3.3. 关系表达式

由于文法中 RelopExpression 可以通过“FactorExpression -> (RelopExpression)”被规约为 FactorExpression，从而继续参与表达式计算；因此对于关系表达式，不能采用记下真假出口的 truelist、falselist 再回填的方式（这样就没有临时变量存放表达式的值了）。因此，我们借鉴了课件中的“数值表示法”，即布尔表达式也像普通运算一样计算真假值。此时，在作为 if、while 语句的分支或循环条件时，则需要额外的两条跳转指令来完成跳转。但当归约到 IfStatement 或 WhileStatement 时，内部的 block 已经产生了很多新的四元式，这两条跳转指令在此时插入则会位置错误。因此，我们修改了文法如下：

```
WhileStatement -> while M ( RelopExpression A ) M Block
IfStatement -> if ( RelopExpression A ) M Block | if
( RelopExpression A ) M Block N else M Block
M -> epsilon
N -> epsilon
A -> epsilon
```

产生式“A -> epsilon”规约时，会插入 jnz、j 两条跳转四元式，判断条件与转移地址都待回填，并将这两条四元式的地址记录在 truelist、falselist 属性中。这样，当归约到 IfStatement 或 WhileStatement 时，就可以根据该地址进行转移条件与转移地址的回填。

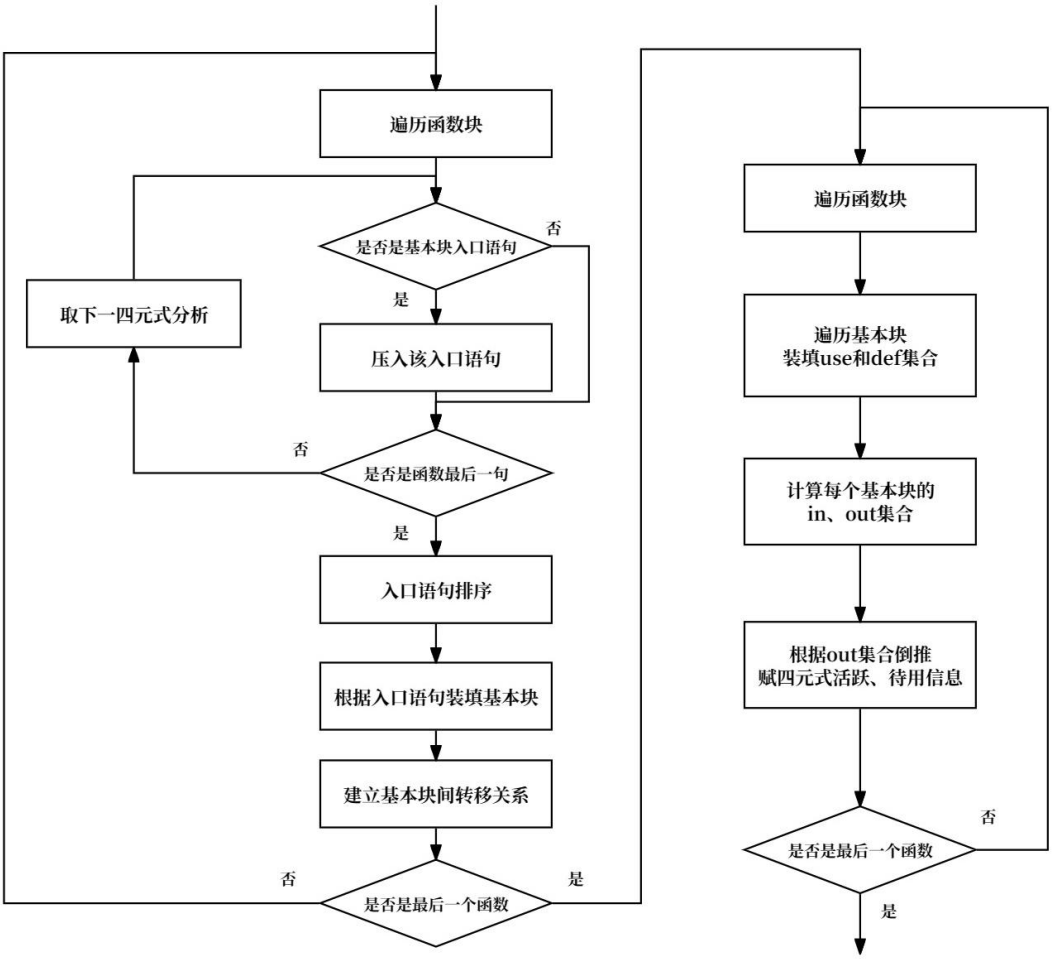
此外，由于目标代码可以直接使用 slt、sgt、seq 等指令来进行判断，四元式生成时无需再额外用“j<”、“j>”、“j==”等逻辑跳转赋值，而是改为用“<”、“>”、“==” 直接表示运算。程序示例如下：

```
int main(){
  int a;

  a = 10;

  while(a > 0){
    a = a - 1;
  }

  return 0;
}
```

地址	四元式
100	(=, 10, _, a)
101	(>, a, 0, T0)
102	(jnz, T0, _, 104)
103	(j, _, _, 107)
104	(int-, a, 1, T1)
105	(=, T1, _, a)
106	(j, _, _, 101)
107	(ret, 0, _, _)

4.4. 基本块划分器

4.4.1. 基本块划分流程图



4.4.2. 重点函数分析

1. 基本块划分

基本块划分主要分为三个步骤：1. 找到所有入口语句的位置并记录；2. 压入每个基本块的代码；3. 建立基本块之间的转移关系。divideBlocks 函数即负责将程序中的代码划分为基本块，并建立这些基本块之间的转移关系。

首先通过迭代函数表 func_table 来处理每个函数。在处理每个函数时，它首先找到所有入口语句的位置，并记录下来。入口语句通常是函数块的第一个语句、转移语句的目标语句以及有条件转移的下一条语句。然后，压入每个基本块的代码（介于两个入口语句之间，或遇转移语句时停止）。最后建立基本块之间的转移关系，即确定每个基本块的后继基本块。

```
def divideBlocks(self, func_table: List):
    # 按照函数块划分基本块
    func_index = 0
    while func_index < len(func_table):
        func = func_table[func_index]
        block_enter = []    # 当前函数块的所有入口语句

        """
        第一步：找到所有入口语句的位置并记录
        """
        # 函数块的第一个语句是入口语句
        block_enter.append(func["enter"] - self.start_address)

        func_start = func["enter"] - self.start_address
        func_end = len(self.quaternion_table) if func ==
func_table[-1] \
                   else func_table[func_index + 1]["enter"] -
self.start_address

        # 遍历当前函数块的四元式寻找入口语句
        for quar_index in range(func_start, func_end):
            quar = self.quaternion_table[quar_index]
            if quar.op[0] == "j":
                # 无条件转移（转移到的语句是入口语句）
                if quar.op == "j":
```

```
                    # return 在最后可能没有回填，这样的 j 语句作废
                    if int(quar.tar) != 0:
                        block_enter.append(int(quar.tar) -
self.start_address)
                # 有条件转移（下一条/转移到的语句是入口语句）
                else:
                    if quar_index < func_end - 1:
                        block_enter.append(quar_index + 1)
                    block_enter.append(int(quar.tar) -
self.start_address)
            elif quar.op == "call":
                if quar_index < func_end - 1:
                    block_enter.append(quar_index + 1)

        func_index += 1
        block_enter = sorted(list(set(block_enter)))  # 对入口语
句去重并排序

        """
        第二步：压入每个基本块的代码（介于两个基本块起始地址之间 / 到一个
转移语句停止）
        """
        self.func_blocks[func["name"]] = []
        for enter_index in range(len(block_enter)):
            block_start = block_enter[enter_index]
            block_end = block_enter[enter_index + 1] if
enter_index < len(block_enter) - 1 else func_end
            block = Block()
            block.name = func["name"] if enter_index == 0 else
self.getBlockName()
            block.start_addr = block_start
            # 压入每一个四元式
            for quar_index in range(block_start, block_end):
                cur_quar = self.quaternion_table[quar_index]
                block.codes.append(cur_quar)
                # 包含到一个转移语句（可能是无条件转移） 该基本块停止
                if cur_quar.op[0] == "j" or cur_quar.op in
["call", "ret"]:
                    break

            self.func_blocks[func["name"]].append(block)

        """
        第三步：建立基本块之间的转移关系
        """
```

```
        for block_index in
range(len(self.func_blocks[func["name"]])):
            block = self.func_blocks[func["name"]][block_index]
            next_block =
self.func_blocks[func["name"]][block_index + 1] if block_index
< len(self.func_blocks[func["name"]]) - 1 else None
            last_quar = block.codes[-1]
            if last_quar.op[0] == "j":
                des_addr = int(last_quar.tar) -
self.start_address
                des_block = next((block for block in
self.func_blocks[func["name"]]
                                  if block.start_addr ==
des_addr), None)
                # 无条件转移
                if last_quar.op == "j":
                    block.next1 = des_block
                    block.next2 = None
                # 有条件转移
                else:
                    block.next1 = next_block
                    block.next2 = des_block
                # 修改四元式跳转目标为标签
                last_quar.tar = des_block.name
            elif last_quar.op == "ret":
                block.next1 = None
                block.next2 = None
            else:
                block.next1 = next_block
                block.next2 = None
```

2. 活跃、待用信息计算

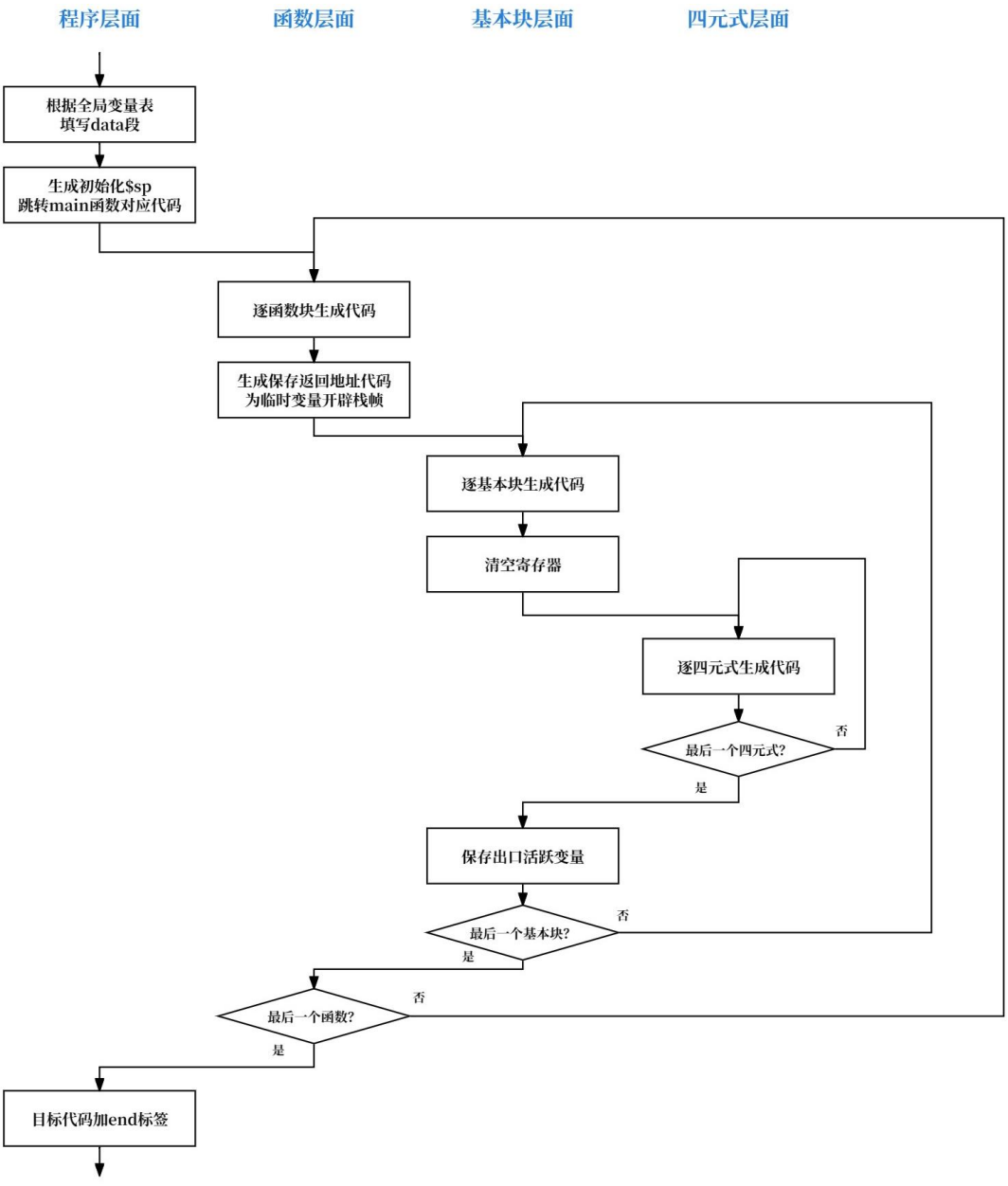
在划分基本块之后，对于每个函数逐个分析每个基本块，计算其的 Use 和 Def 集合（Use 集合包含在基本块中第一次被使用而非被赋值的变量，Def 集合包含了在基本块中第一次被赋值而非被使用的变量）。然后根据 Use 和 Def 集合，迭代更新每个基本块的 In 和 Out 活跃集合。根据每个基本块的活跃信息，倒推分析四元式，并为每个四元式中的变量赋予相应的活跃和待用信息。

```
# 计算变量的活跃和待用信息
def computeBlocks(self, func_table: List):
    self.divideBlocks(func_table)

    # 以函数为单位进行分析
    for func, blocks in self.func_blocks.items():
        # 下面对每个基本块进行分析
        # 计算每个基本块的 Use 和 Def 集合
        for block in blocks:
            # 逐个四元式分析
            for quar in block.codes:
                if quar.op == "j":
                    continue
                if quar.op != "call":
                    if self._is_var(quar.src1) and quar.src1
not in block.def_set:
                        block.use_set.add(quar.src1)
                    if self._is_var(quar.src2) and quar.src2
not in block.def_set:
                        block.use_set.add(quar.src2)
                if quar.op[0] != "j":    # 不是条件跳转
                    if self._is_var(quar.tar) and quar.tar not
in block.use_set:
                        block.def_set.add(quar.tar)

            block.in_set  = set(block.use_set)
            block.out_set = set()

        # 更新每个基本块的 In 和 Out 活跃集
        changeFlag = True
        while changeFlag:
            changeFlag = False
            for block in blocks:
                next1 = block.next1
                next2 = block.next2
                if next1:
                    # 将后继 block 入口活跃集元素插入当前出口活跃集中
                    # 若在 def 集中不存在，还需加到自己的入口活跃集中
                    # OUT = ∪IN(后继)   IN = USE ∪ (OUT - DEF)
                    for var in next1.in_set:
                        if var not in block.out_set:
                            block.out_set.add(var)
                            changeFlag = True
                            if var not in block.def_set:
                                block.in_set.add(var)
```

```
                if next2:
                    for var in next2.in_set:
                        if var not in block.out_set:
                            block.out_set.add(var)
                            changeFlag = True
                            if var not in block.def_set:
                                block.in_set.add(var)

        # 记录当前每个基本块中变量的活跃、待用信息
        blockSymbolInfoTable = { }
        # 记录每个基本块的出口活跃变量的信息
        for block in blocks:
            symbolInfoTable = {}
            for var in block.out_set:
                symbolInfoTable[var] = SymbolInfo(None, True)
            blockSymbolInfoTable[block] = symbolInfoTable


        # 倒推给每个四元式赋上活跃与待用信息
        for block in blocks:
            index = len(block.codes)
            for quar in reversed(block.codes):
                index -= 1
                # 不存在变量信息更新
                if quar.op in ["j", "call"]:
                    continue
                if self._is_var(quar.src1):
                    quar.info_src1 =
blockSymbolInfoTable[block].get(quar.src1, SymbolInfo())
                    blockSymbolInfoTable[block][quar.src1] =
SymbolInfo(index, True)
                if self._is_var(quar.src2):
                    quar.info_src2 =
blockSymbolInfoTable[block].get(quar.src2, SymbolInfo())
                    blockSymbolInfoTable[block][quar.src2] =
SymbolInfo(index, True)
                if quar.op[0] != "j":  # 不是条件跳转（tar 是地址）
                    if self._is_var(quar.tar):
                        quar.info_tar =
blockSymbolInfoTable[block].get(quar.tar, SymbolInfo())
                        blockSymbolInfoTable[block][quar.tar] =
SymbolInfo(None, False)
```

4.5. 目标代码生成器

4.5.1. 目标代码生成流程图



4.5.2. 重点函数分析

1. 寄存器分配

分配寄存器时，首先优先检查是否存在空闲的寄存器。若存在空闲的寄存器，则可以直接分配给对应变量。若没有空闲寄存器，则需要选择一个最远引用的变量来让渡寄存器，具体来说：遍历当前分配的所有寄存器，查找每个寄存器中下次引用最远的变量，然后选择引用最远的变量对应的寄存器，从而最大程度地减少寄存器的使用冲突。找到合适的寄存器后，则释放该寄存器并保存其中的数据，将腾空的寄存器分配给需要的变量。

```
# 分配一个新的寄存器
def allocateFreeRegister(self, quars, cur_quar_index, out_set,
codes):
    free_reg = ""
    # 有寄存器空闲，则直接分配
    if len(self.free_registers):
        free_reg = self.free_registers.pop()
        return free_reg

    # 若无，则需要寻找最远引用的变量让渡寄存器
    farest_usepos = float('-inf')
    for reg, vars in self.RValue.items():
        cur_usepos = float('inf')
        # 看看存在这个 reg 中引用最近的那个 var
        for var in vars:
            # 如果存在别的地方，则很好
            if len(self.AValue[var]) > 0:
                continue

            for idx in range(cur_quar_index, len(quars)):
                quar = quars[idx]
                if var in [quar.src1, quar.src2]:
                    cur_usepos = min(cur_usepos, idx -
cur_quar_index)
                    break
                if var == quar.tar:
                    break

        if cur_usepos == float('inf'):
            free_reg = reg
            break
        elif cur_usepos > farest_usepos:
            farest_usepos = cur_usepos
            free_reg = reg
```

```
    # 释放寄存器，保存数据
    for var in self.RValue[free_reg]:
        self.AValue[var].remove(free_reg)
        # 若无其他地方存法数据，才需要 sw
        if len(self.AValue[var]) == 0:
            need_store = None
            for idx in range(cur_quar_index, len(quars)):
                quar = quars[idx]
                if var in [quar.src1, quar.src2]:
                    need_store = True
                    break
                if var == quar.tar:
                    break
            # 没有被引用、定值，检查是不是出口活跃
            if need_store == None:
                need_store = True if var in out_set else False
            if need_store:
                self.storeVariable(var, free_reg, codes)

    self.RValue[free_reg].clear()

    return free_reg
```

2. 为源操作数分配寄存器

为源操作数分配寄存器首先需要检查 AValue 中是否已经有了对应源操作数的现成寄存器。如果有，则可以直接返回该寄存器；如果没有，则调用寄存器分配函数分配新的寄存器。此外，若源操作数是变量，则通过“lw”从内存加载；否则，则通过“li”装载立即数。

```
# 为四元式的 src 获取寄存器
def getSrcRegister(self, src, quars, cur_quar_index, out_set,
codes):
    reg = ""
    # 先查 AValue 有无现成
    for pos in self.AValue[src]:
        if pos != "Memory":
            return pos
```

```
    # 没有则分配一个
    reg = self.allocateFreeRegister(quars, cur_quar_index,
out_set, codes)

    if self._is_variable(src):
        # 局部变量或中间变量
        if src in self.func_vars or src[0] == 'T':
            codes.append(f"lw {reg}, {self.memory[src]}($sp)")
        elif src in self.data_vars:
            codes.append(f"lw {reg}, {src}")
        # 更新 AValue RValue
        self.AValue[src].add(reg)
        self.RValue[reg].add(src)
    else:   # 立即数
        codes.append(f"li {reg}, {src}")

    return reg
```

3. 为目的操作数分配寄存器

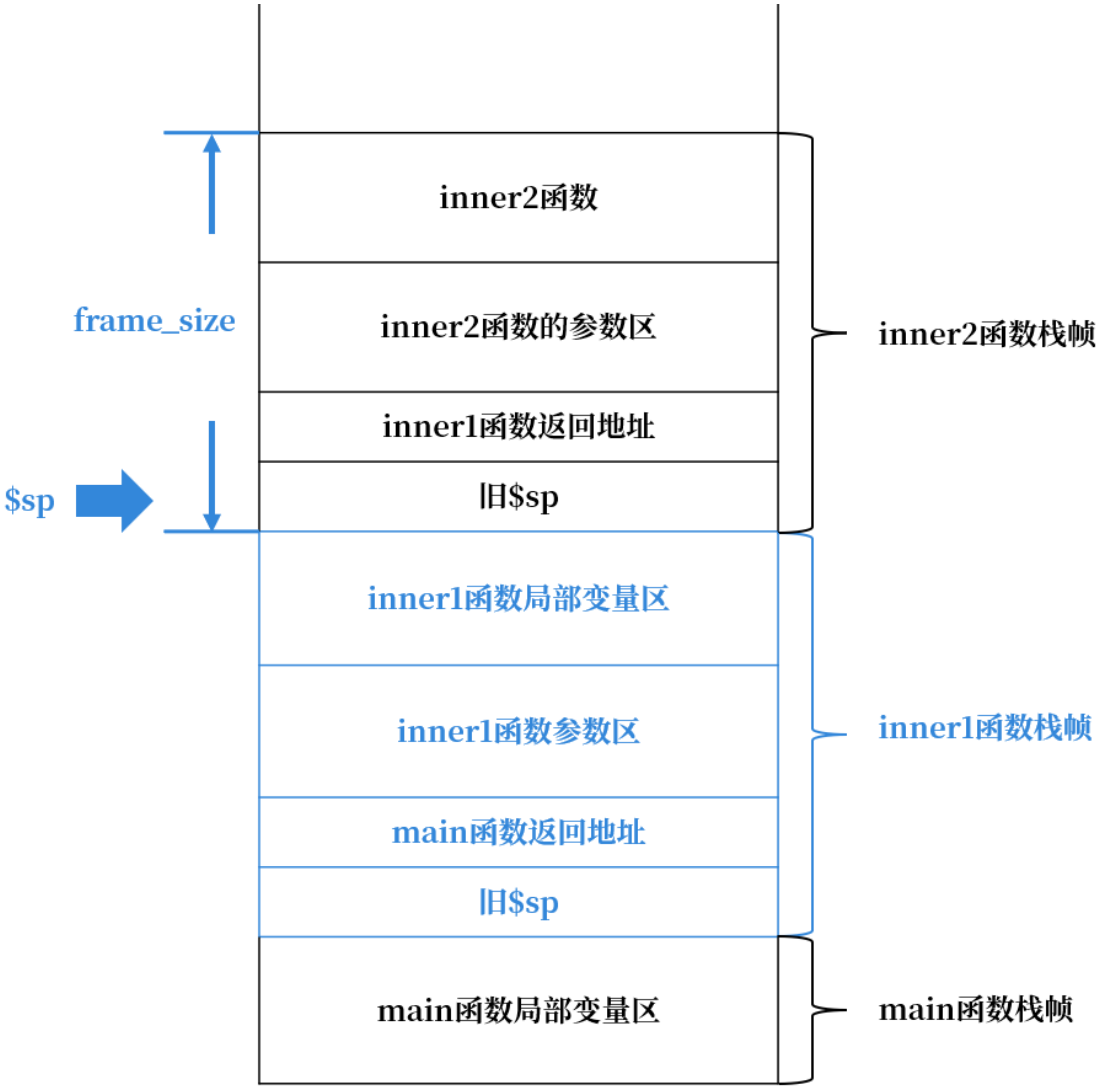
为目的操作数分配寄存器时，优先考虑是否有可复用的源操作数的寄存器。即，若源操作数在此四元式后不再活跃，则直接将此分配器转而分配给目的操作数对应变量，并进行相应的更新。若不可复用源操作数的寄存器，则调用寄存器分配函数分配新的寄存器。

```
# 为四元式的 tar 获取寄存器
def getTarRegister(self, tar, quars, cur_quar_index, out_set,
codes):
    quar = quars[cur_quar_index]
    src1 = quar.src1
    # 看能否复用操作数的寄存器
    # 首先保证 src1 不是数字，其次不抢占全局变量的寄存器
    if self._is_variable(src1) and src1 not in (self.data_vars
- self.func_vars):
        for pos in self.AValue[src1]:
            if pos != "Memory" and len(self.RValue[pos]) == 1:
                if not quar.info_src1.active:
                    self.RValue[pos].remove(src1)
                    self.RValue[pos].add(tar)
                    self.AValue[src1].remove(pos)
```

```
                    self.AValue[tar].add(pos)
                    return pos

    # 重新分配
    reg = self.allocateFreeRegister(quars, cur_quar_index,
out_set, codes)
    self.RValue[reg].add(tar)
    self.AValue[tar].add(reg)

    return reg
```

4.5.3. 重点设计分析

1. 函数栈管理



函数栈的管理与设计如上图所示。函数栈被设计在堆栈段上（首地址0x10040000）。$sp 寄存器中存放的是当前函数栈的底部地址，regManager 中的 frame_size 变量用于存储当前栈的大小。当函数调用时，“param”四元式会将变量压入一个列表中，在“call”时逐一将变量填入要压入函数的参数区，地址为当前$sp+frame_size+2×4（返回地址$ra 与旧$sp）+变量顺序×4。此后，调用者函数将$sp 压栈。进入被调用函数后，首先将$ra 函数中的返回地址（jal 填写）转存到对应位置（防止嵌套调用$ra 丢失），之后便可以通过$sp+相对地址来访问参数和局部变量。

函数返回时，按照 MIPS 规范将返回值放入$v0 寄存器中，然后从对应地址取出返回地址装填$ra，并通过 jr 指令返回。此后，$sp 加载旧值，就成功恢复了调用者函数的栈帧状态。返回值从$v0 寄存器中取得。

2. 全局变量与局部变量

不同于局部变量，全局变量存储在 data 段（0x10010000 起）。存储在 data 段的变量可以直接通过变量名在目标代码中访问对应地址，而无需额外通过字典记录相较$sp 的偏移量。根据低层屏蔽高层的原理，在局部变量表中寻找不到的变量才会被视作全局变量。此外，赋值过的全局变量将在每个基本块后被保存到内存中（而不根据后续活跃信息），从而便于通过 data 段观察。

5. 调试分析

5.1. 正确用例

5.1.1. 题目用例

```
int a;
```

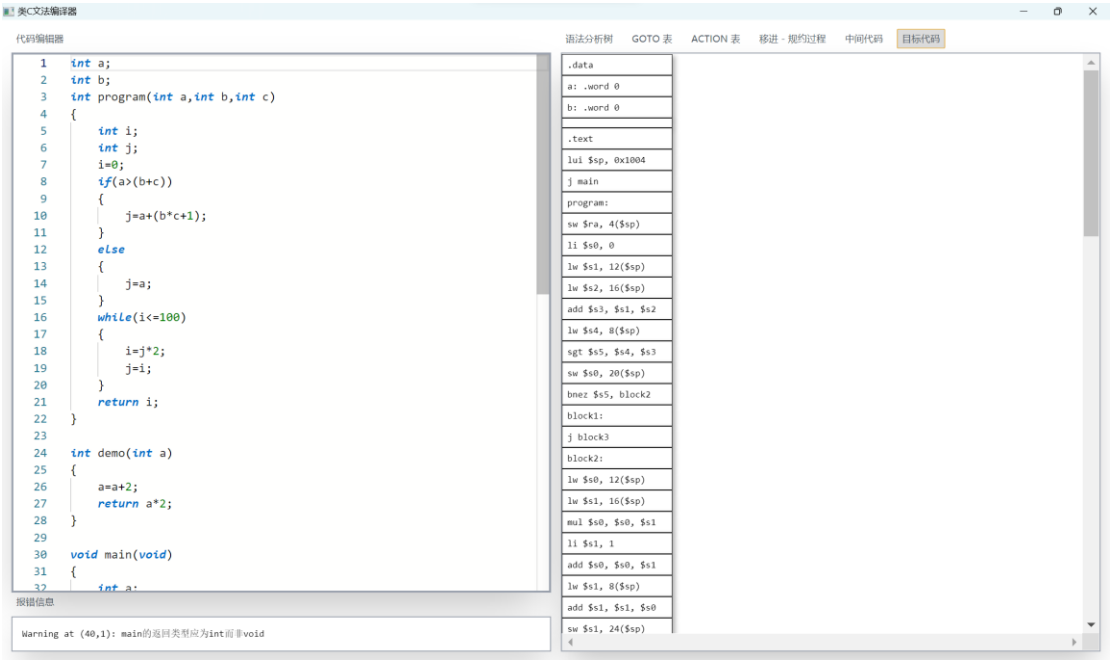
```
int b;
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
        j=i;
    }
    return i;
}

int demo(int a)
{
    a=a+2;
    return a*2;
}

void main(void)
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a,b,demo(c));
    return;
}
```

生成的目标代码：

```
.data
    a: .word 0
    b: .word 0

.text
    lui $sp, 0x1004
    j main
```

```
program:
    sw $ra, 4($sp)
    li $s0, 0
    lw $s1, 12($sp)
    lw $s2, 16($sp)
    add $s3, $s1, $s2
    lw $s4, 8($sp)
    sgt $s5, $s4, $s3
    sw $s0, 20($sp)
    bnez $s5, block2
block1:
    j block3
block2:
    lw $s0, 12($sp)
    lw $s1, 16($sp)
    mul $s0, $s0, $s1
    li $s1, 1
    add $s0, $s0, $s1
    lw $s1, 8($sp)
    add $s1, $s1, $s0
    sw $s1, 24($sp)
    j block4
block3:
    lw $s0, 8($sp)
    sw $s0, 24($sp)
block4:
    lw $s0, 20($sp)
    li $s1, 100
    sle $s2, $s0, $s1
    bnez $s2, block6
block5:
    j block7
block6:
    lw $s0, 24($sp)
    li $s1, 2
    mul $s0, $s0, $s1
    sw $s0, 20($sp)
    sw $s0, 24($sp)
    j block4
block7:
    lw $v0, 20($sp)
    lw $ra, 4($sp)
    jr $ra
demo:
    sw $ra, 4($sp)
```

```
    lw $s0, 8($sp)
    li $s1, 2
    add $s0, $s0, $s1
    li $s1, 2
    mul $s2, $s0, $s1
    add $v0, $s2, $zero
    lw $ra, 4($sp)
    jr $ra
main:
    li $s0, 3
    li $s1, 4
    li $s2, 2
    sw $s1, 4($sp)
    sw $s0, 0($sp)
    sw $s2, 20($sp)
    sw $sp, 12($sp)
    addi $sp, $sp, 12
    jal demo
    lw $sp, 0($sp)
    add $s2, $v0, $zero
    sw $s2, 12($sp)
block8:
    lw $s0, 0($sp)
    sw $s0, 24($sp)
    lw $s0, 4($sp)
    sw $s0, 28($sp)
    lw $s0, 12($sp)
    sw $s0, 32($sp)
    sw $sp, 16($sp)
    addi $sp, $sp, 16
    jal program
    lw $sp, 0($sp)
    add $s0, $v0, $zero
    sw $s0, 16($sp)
block9:
    lw $s0, 16($sp)
    j end
end:
```



5.1.2. 排序用例

```
int sorted1;
int sorted2;
int sorted3;
int sorted4;
int sorted5;

// 插入排序算法
void insertionSort(int unsorted1, int unsorted2, int unsorted3,
int unsorted4, int unsorted5) {
    sorted1 = unsorted1; // 将第一个元素放入已排序序列中

    // 将第二个元素插入已排序序列中
    if (unsorted2 < sorted1) {
        sorted2 = sorted1;
        sorted1 = unsorted2;
    } else {
        sorted2 = unsorted2;
    }

    // 将第三个元素插入已排序序列中
    if (unsorted3 < sorted1) {
        sorted3 = sorted2;
```

```
        sorted2 = sorted1;
        sorted1 = unsorted3;
    } else {
        if (unsorted3 < sorted2) {
            sorted3 = sorted2;
            sorted2 = unsorted3;
        } else {
            sorted3 = unsorted3;
        }
    }

    // 将第四个元素插入已排序序列中
    if (unsorted4 < sorted1) {
        sorted4 = sorted3;
        sorted3 = sorted2;
        sorted2 = sorted1;
        sorted1 = unsorted4;
    } else {
        if (unsorted4 < sorted2) {
            sorted4 = sorted3;
            sorted3 = sorted2;
            sorted2 = unsorted4;
        } else {
            if (unsorted4 < sorted3) {
                sorted4 = sorted3;
                sorted3 = unsorted4;
            } else {
                sorted4 = unsorted4;
            }
        }
    }

    // 将第五个元素插入已排序序列中
    if (unsorted5 < sorted1) {
        sorted5 = sorted4;
        sorted4 = sorted3;
        sorted3 = sorted2;
        sorted2 = sorted1;
        sorted1 = unsorted5;
    } else {
        if (unsorted5 < sorted2) {
            sorted5 = sorted4;
            sorted4 = sorted3;
            sorted3 = sorted2;
            sorted2 = unsorted5;
```

```
        } else {
            if (unsorted5 < sorted3) {
                sorted5 = sorted4;
                sorted4 = sorted3;
                sorted3 = unsorted5;
            } else {
                if (unsorted5 < sorted4) {
                    sorted5 = sorted4;
                    sorted4 = unsorted5;
                } else {
                    sorted5 = unsorted5;
                }
            }
        }
    }
}

int main() {
    insertionSort(9, 0, 1, 6, 3);
    return 0;
}
```

生成的目标代码：

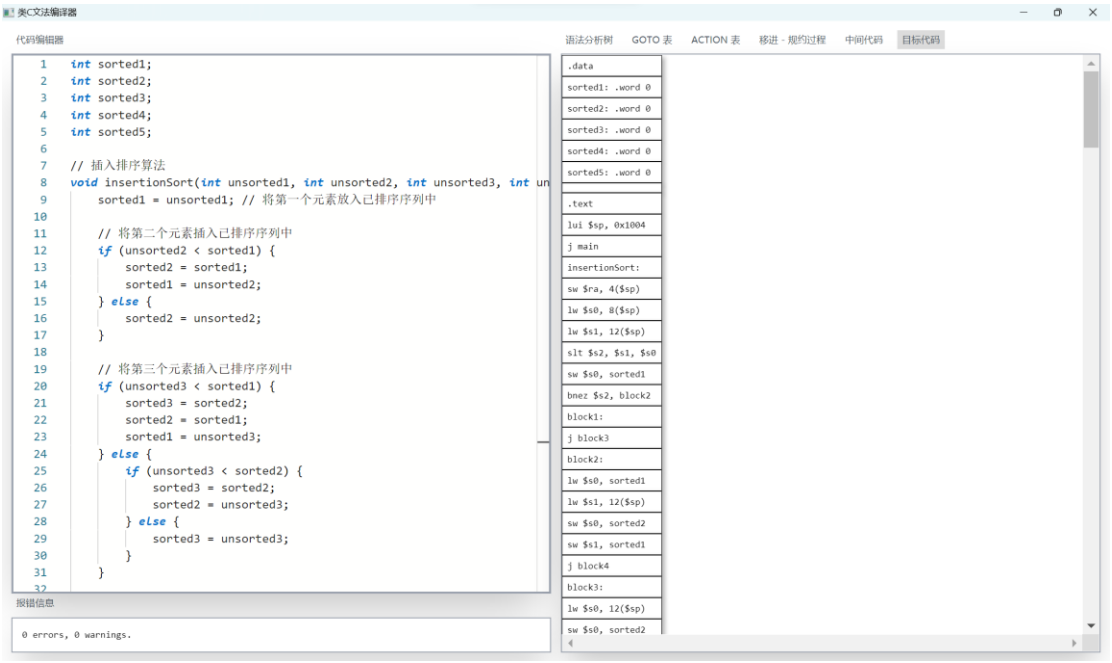
```
.data
    sorted1: .word 0
    sorted2: .word 0
    sorted3: .word 0
    sorted4: .word 0
    sorted5: .word 0

.text
    lui $sp, 0x1004
    j main
insertionSort:
    sw $ra, 4($sp)
    lw $s0, 8($sp)
    lw $s1, 12($sp)
    slt $s2, $s1, $s0
    sw $s0, sorted1
    bnez $s2, block2
block1:
```

```
    j block3
block2:
    lw $s0, sorted1
    lw $s1, 12($sp)
    sw $s0, sorted2
    sw $s1, sorted1
    j block4
block3:
    lw $s0, 12($sp)
    sw $s0, sorted2
block4:
    lw $s0, 16($sp)
    lw $s1, sorted1
    slt $s2, $s0, $s1
    bnez $s2, block6
block5:
    j block7
block6:
    lw $s0, sorted2
    lw $s1, sorted1
    lw $s2, 16($sp)
    sw $s0, sorted3
    sw $s1, sorted2
    sw $s2, sorted1
    j block11
block7:
    lw $s0, 16($sp)
    lw $s1, sorted2
    slt $s2, $s0, $s1
    bnez $s2, block9
block8:
    j block10
block9:
    lw $s0, sorted2
    lw $s1, 16($sp)
    sw $s0, sorted3
    sw $s1, sorted2
    j block11
block10:
    lw $s0, 16($sp)
    sw $s0, sorted3
block11:
    lw $s0, 20($sp)
    lw $s1, sorted1
    slt $s2, $s0, $s1
```

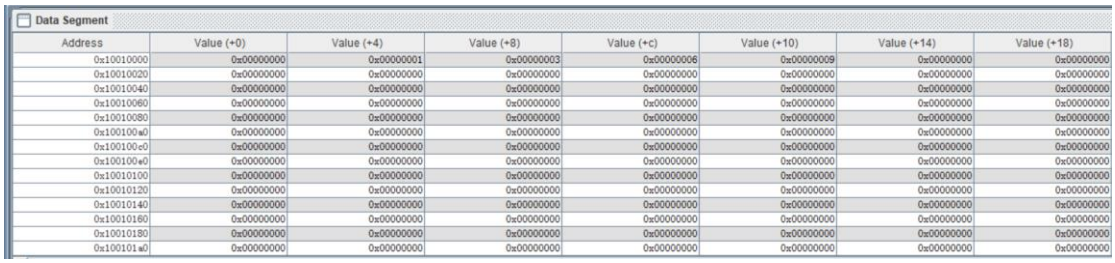
```
    bnez $s2, block13
block12:
    j block14
block13:
    lw $s0, sorted3
    lw $s1, sorted2
    lw $s2, sorted1
    lw $s3, 20($sp)
    sw $s0, sorted4
    sw $s1, sorted3
    sw $s2, sorted2
    sw $s3, sorted1
    j block21
block14:
    lw $s0, 20($sp)
    lw $s1, sorted2
    slt $s2, $s0, $s1
    bnez $s2, block16
block15:
    j block17
block16:
    lw $s0, sorted3
    lw $s1, sorted2
    lw $s2, 20($sp)
    sw $s0, sorted4
    sw $s1, sorted3
    sw $s2, sorted2
    j block21
block17:
    lw $s0, 20($sp)
    lw $s1, sorted3
    slt $s2, $s0, $s1
    bnez $s2, block19
block18:
    j block20
block19:
    lw $s0, sorted3
    lw $s1, 20($sp)
    sw $s0, sorted4
    sw $s1, sorted3
    j block21
block20:
    lw $s0, 20($sp)
    sw $s0, sorted4
block21:
```

```
    lw $s0, 24($sp)
    lw $s1, sorted1
    slt $s2, $s0, $s1
    bnez $s2, block23
block22:
    j block24
block23:
    lw $s0, sorted4
    lw $s1, sorted3
    lw $s2, sorted2
    lw $s3, sorted1
    lw $s4, 24($sp)
    sw $s0, sorted5
    sw $s1, sorted4
    sw $s2, sorted3
    sw $s4, sorted1
    sw $s3, sorted2
    j block34
block24:
    lw $s0, 24($sp)
    lw $s1, sorted2
    slt $s2, $s0, $s1
    bnez $s2, block26
block25:
    j block27
block26:
    lw $s0, sorted4
    lw $s1, sorted3
    lw $s2, sorted2
    lw $s3, 24($sp)
    sw $s0, sorted5
    sw $s1, sorted4
    sw $s2, sorted3
    sw $s3, sorted2
    j block34
block27:
    lw $s0, 24($sp)
    lw $s1, sorted3
    slt $s2, $s0, $s1
    bnez $s2, block29
block28:
    j block30
block29:
    lw $s0, sorted4
    lw $s1, sorted3
```

```
    lw $s2, 24($sp)
    sw $s0, sorted5
    sw $s1, sorted4
    sw $s2, sorted3
    j block34
block30:
    lw $s0, 24($sp)
    lw $s1, sorted4
    slt $s2, $s0, $s1
    bnez $s2, block32
block31:
    j block33
block32:
    lw $s0, sorted4
    lw $s1, 24($sp)
    sw $s0, sorted5
    sw $s1, sorted4
    j block34
block33:
    lw $s0, 24($sp)
    sw $s0, sorted5
block34:
    lw $ra, 4($sp)
    jr $ra
main:
    li $s0, 9
    sw $s0, 8($sp)
    li $s1, 0
    sw $s1, 12($sp)
    li $s2, 1
    sw $s2, 16($sp)
    li $s3, 6
    sw $s3, 20($sp)
    li $s4, 3
    sw $s4, 24($sp)
    sw $sp, 0($sp)
    addi $sp, $sp, 0
    jal insertionSort
    lw $sp, 0($sp)
block35:
    li $v0, 0
    j end
end:
```



对应代码在 MARS 中能够正确运行，观察数据段排序正确。



5.2. 错误用例

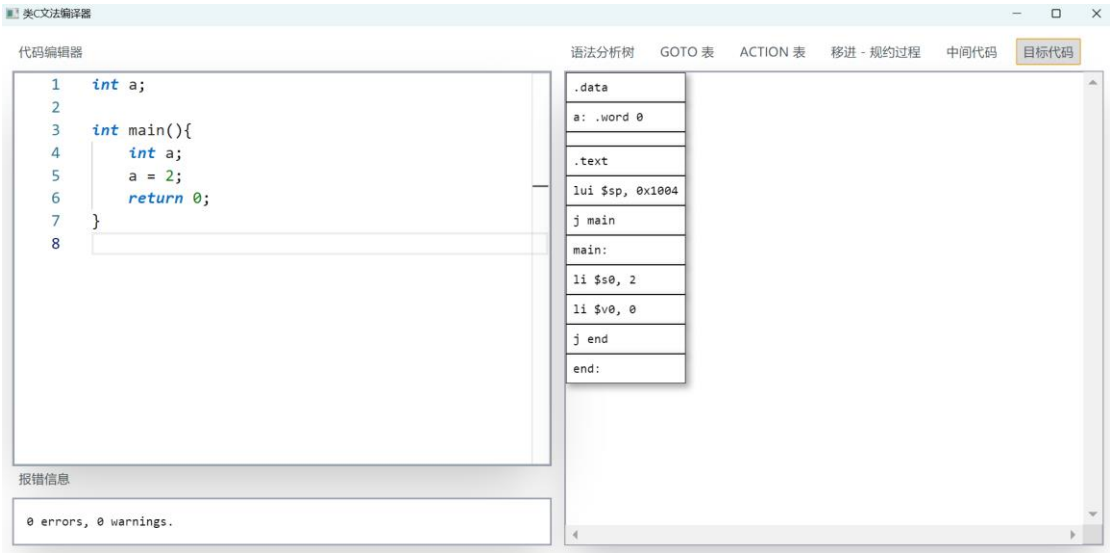
5.2.1. 变量定义问题

```
int main(){
    int a;
    int a;
    return 0;
}
```

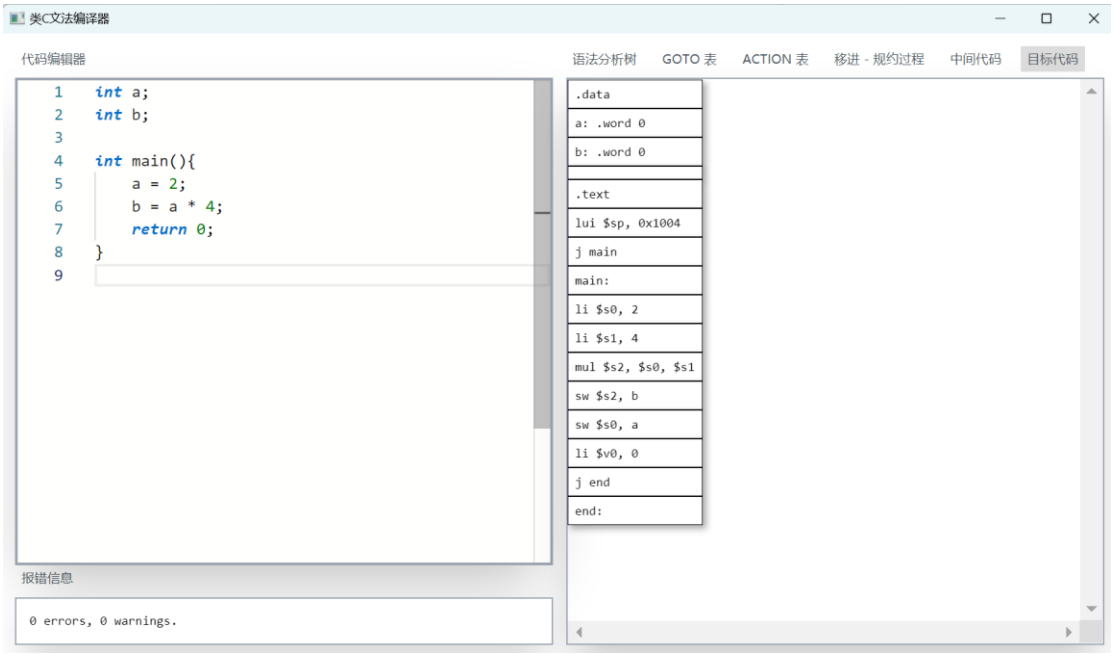
报错信息

```
Error at (2,9): 变量a重定义
```

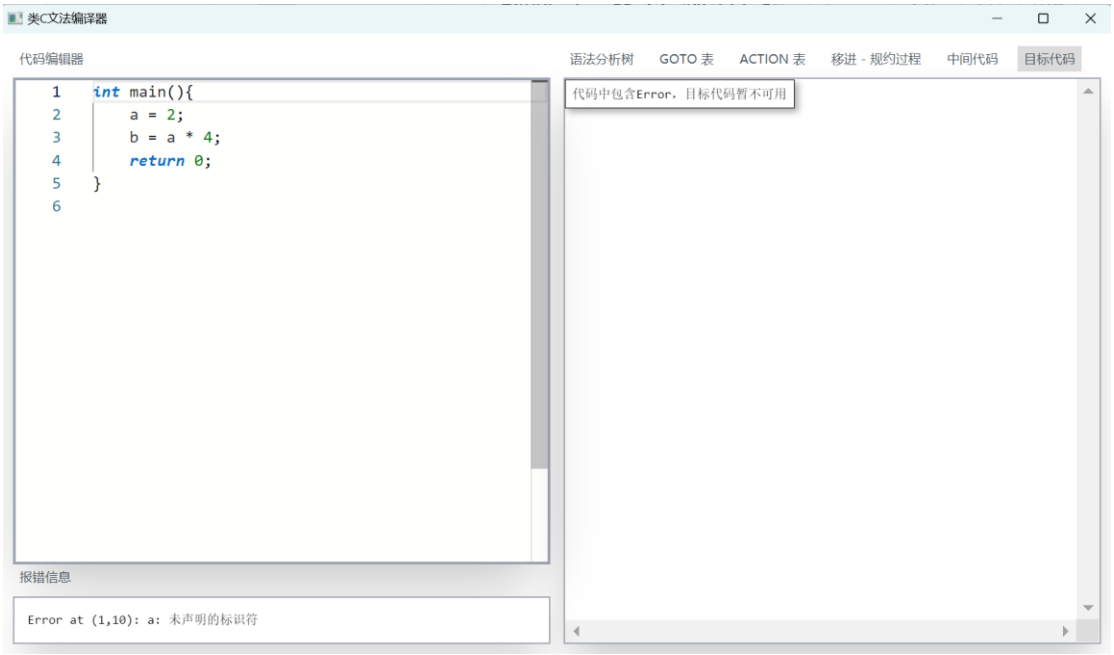
如果同名变量中一个为全局变量，按低层屏蔽高层的原则处理，不报错。



全局变量可以在任意一个函数体内使用，不报错。

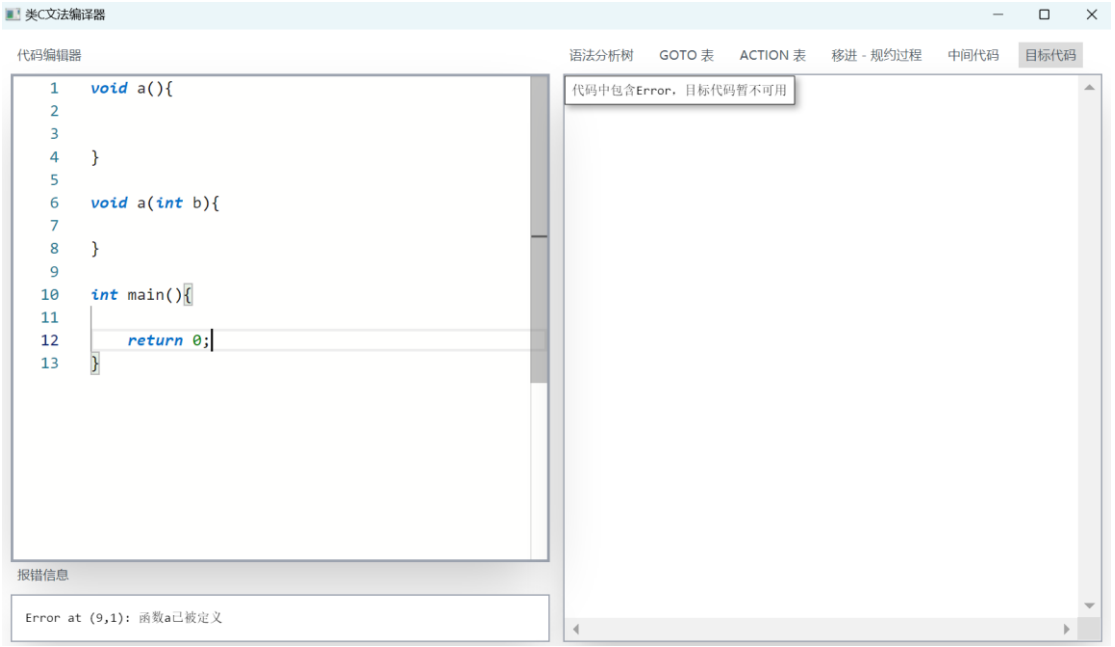


未声明的标识符，报错为 error，不生成中间代码与目标代码。

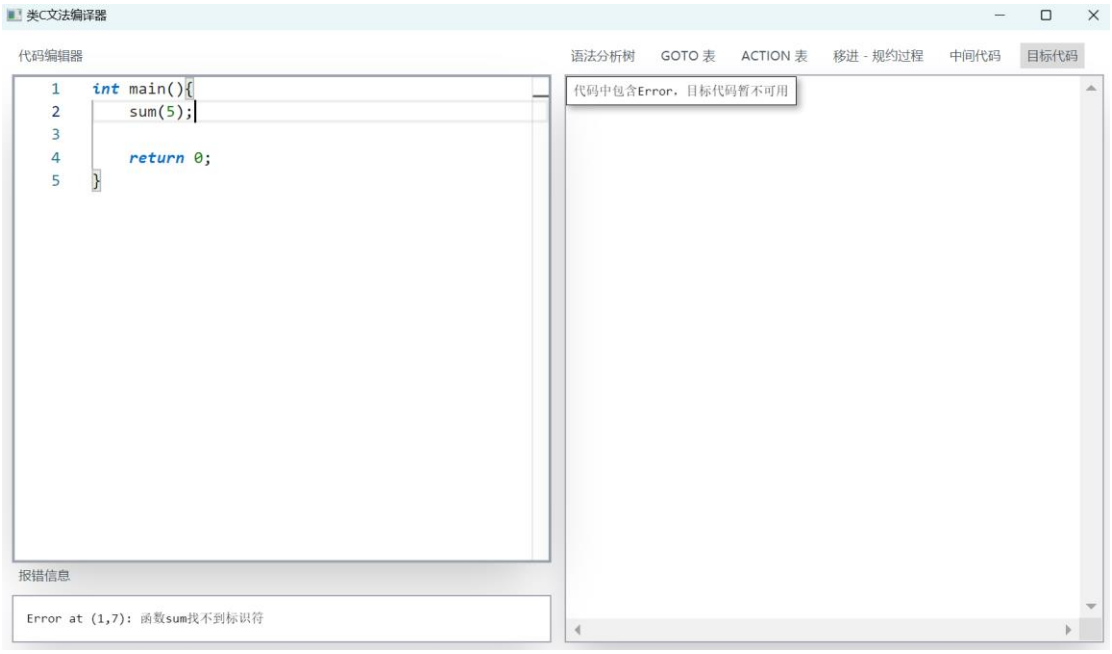


5.2.2. 函数定义问题

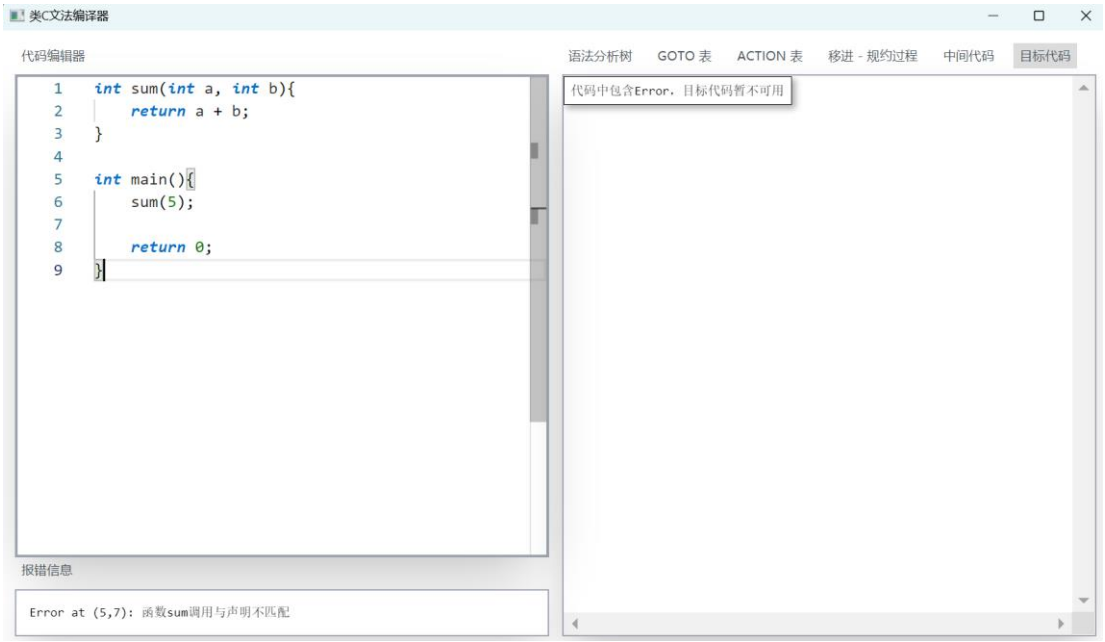
函数重定义问题，报错为 error，不生成中间代码与目标代码。



函数未定义问题，报错为 error，不生成中间代码与目标代码。

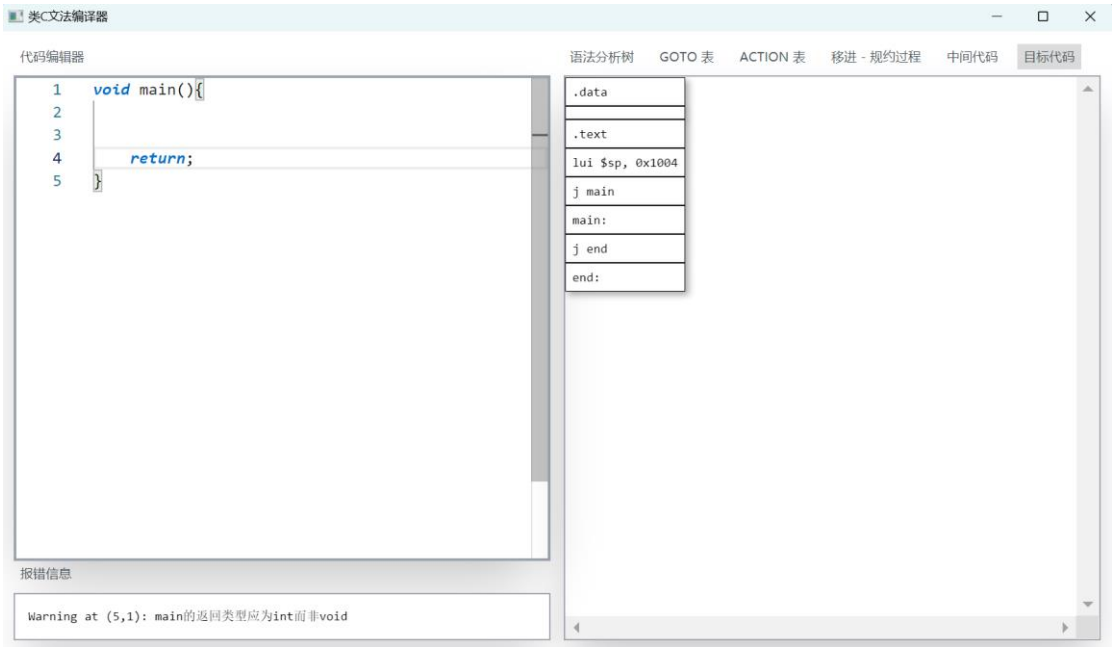


函数声明与调用不匹配问题，报错为 error，不生成中间代码与目标代码。

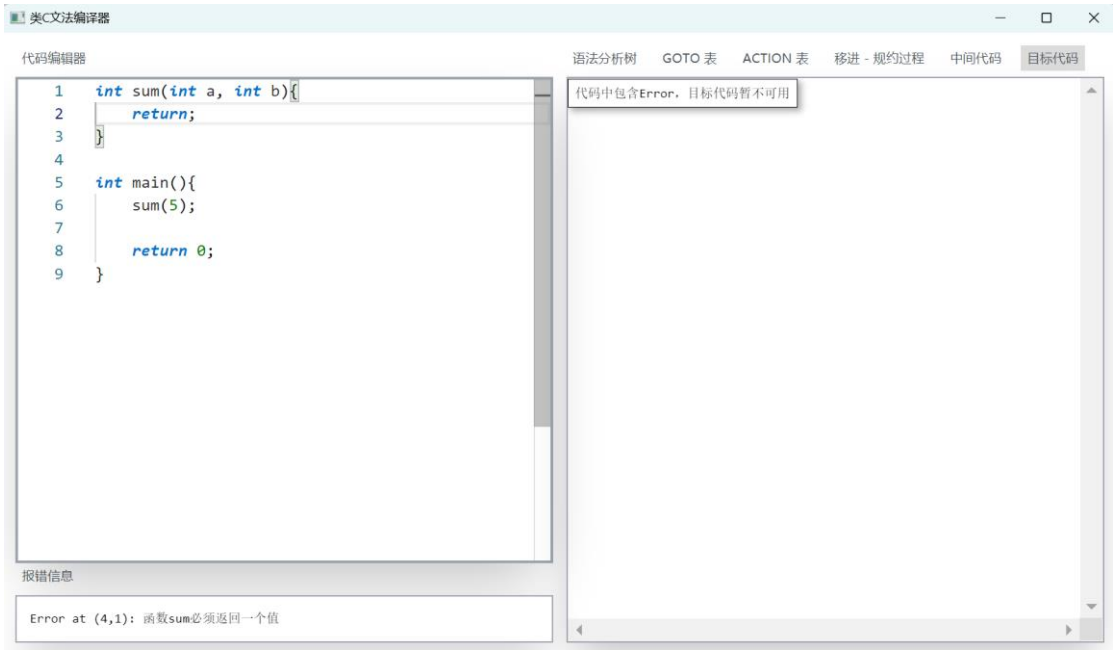


5.2.3. 函数返回值问题

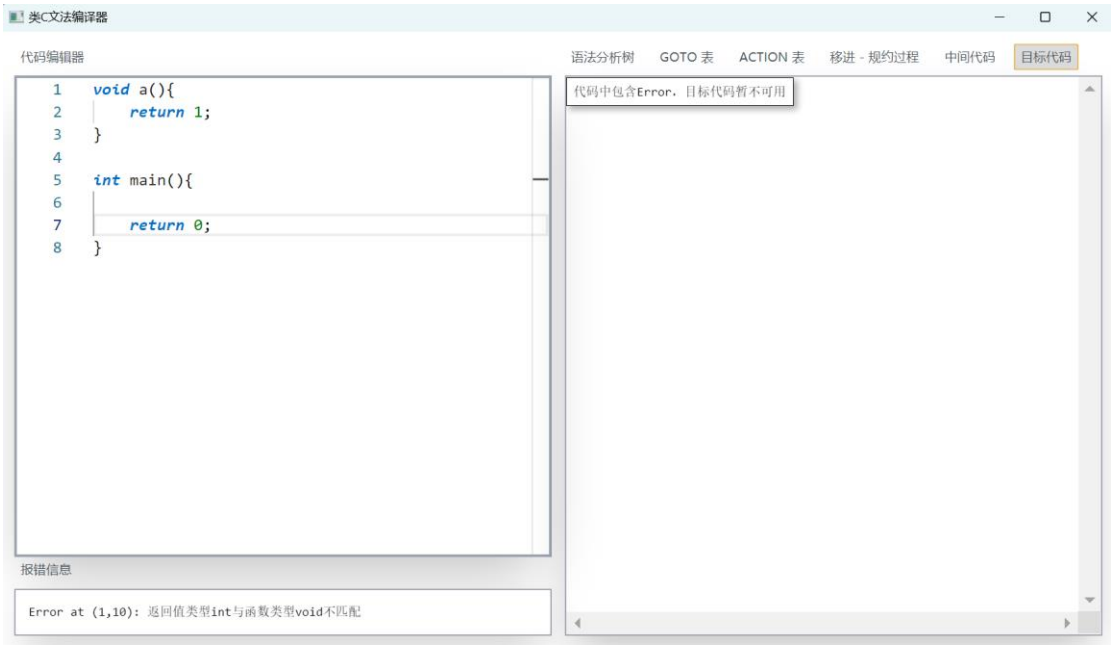
main 函数返回类型为 void，报错为 warning，不影响后续代码生成。



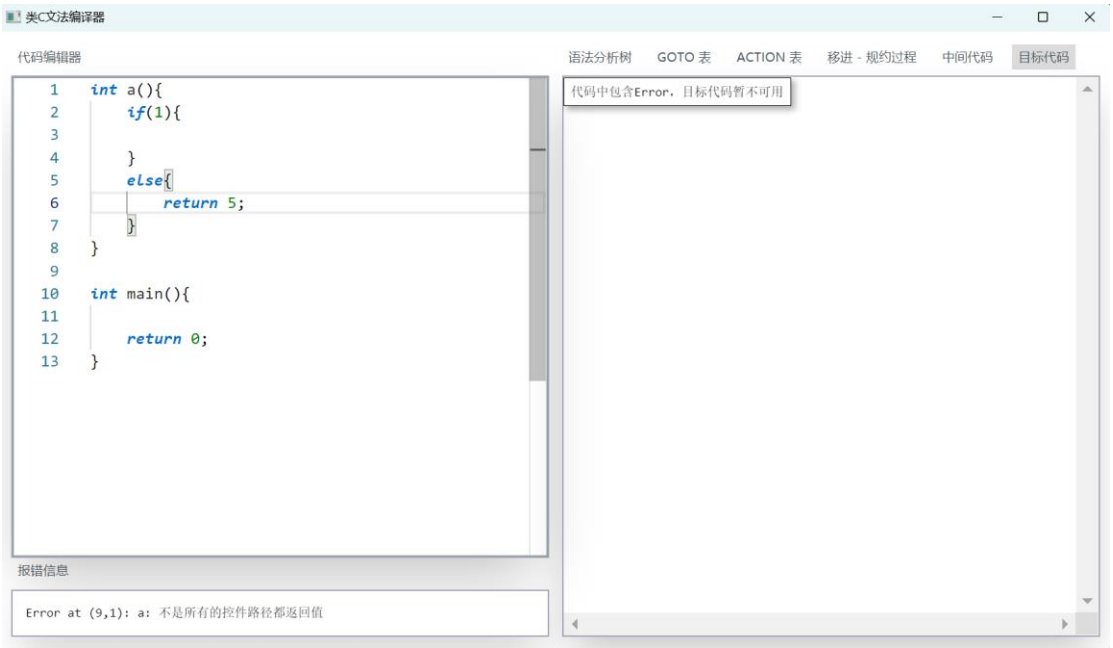
返回类型为 int 的函数没有返回值的情况，报错为 error，不生成后续代码。



返回类型为 void 的函数有返回值的情况，报错为 error，不生成后续代码。



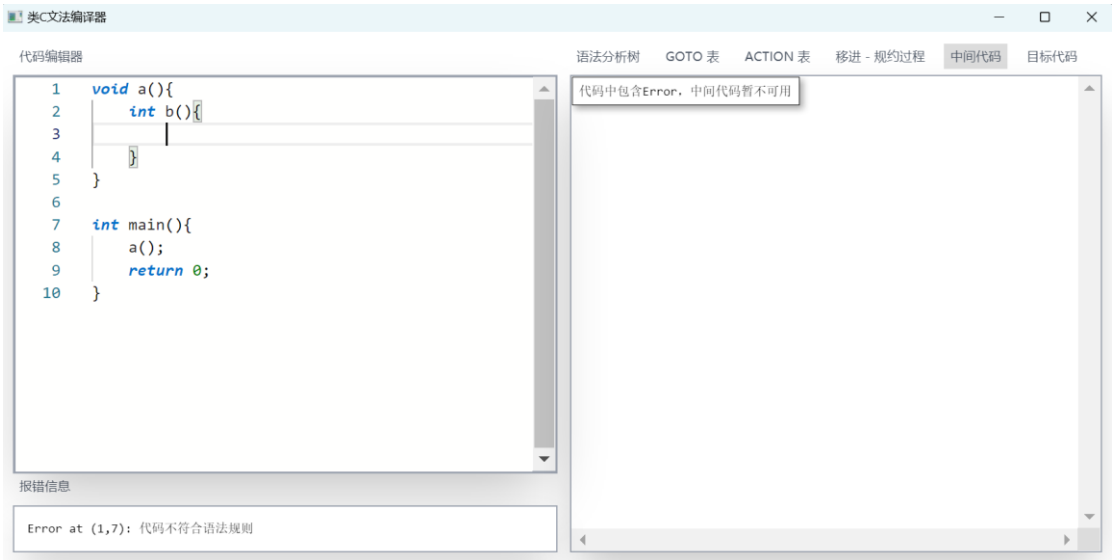
函数返回类型为 int，但存在 if 的某些分支没有返回值的情况，不是所有控件路径都返回值，报错为 error，不生成中间代码与目标代码。



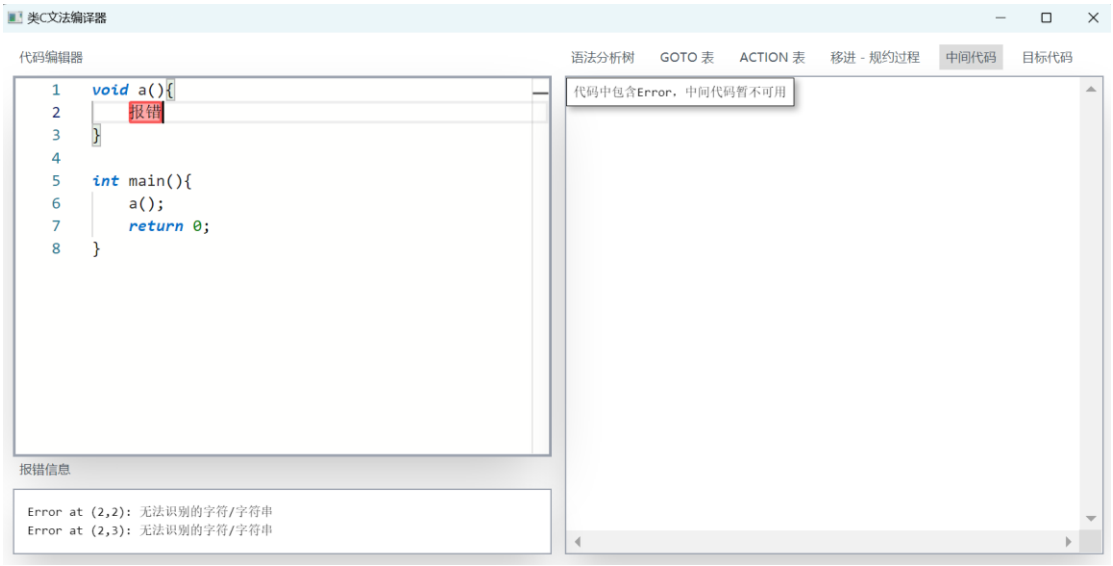
函数返回类型为 void 但没有 return。处理方法是在中间代码生成时，在函数体最后 emit 一个 ret 语句（对应本例的 105 地址的语句），规约时则会自动将所有 nextlist 都跳转到这里，防止 nextlist 未回填。

5.2.4. 语法分析报错

代码不符合规定的文法规则，同样也会报错为 error，不生成中间代码。



5.2.5. 词法分析报错



5.3. 调试问题思考

词法分析、语法分析、语义分析主要在上学期的大作业基础上进行修改，因此本部分主要围绕基本块划分和中间代码生成来进行。

5.3.1. 常数不释放寄存器

关于常数处理，未考虑 addi 等立即数指令，而是通过先 li 再运算的方式实现。在进行算术逻辑运算之后，会判断 src1 和 src2 是否之后活跃。若不活跃，则会释放对应的寄存器。但这样的判断是基于该操作数是变量的情况，而未考虑到常数的情况，会导致常数始终占用寄存器。因此，补上了对常数寄存器默认释放的判断与处理。

5.3.2. 全局变量处理

全局变量存在 data 段中，无论是否活跃都应该将最终的修改结果保存，否则会导致全局变量在内存中存储的值错误。然而，出口活跃变量存储的逻辑却不包含这一点。因此，在基本块保存变量的部分，添加了全局变量的保存处理。

5.3.3. 空函数体出错

在基本块划分时，会根据最后一句四元式（下标-1）来建立基本块之间的勾连关系。然而，空函数体内没有四元式，会导致取不到最后一句四元式，下标超出数组范围而报错。因此需要加上对应判断。

5.3.4. 数据段变量顺序错乱

由于使用集合存储全局变量，因此在输出 data 段时变量顺序会错乱，且每次顺序不同。因此最后通过列表 sorted 保证每次输出时的顺序相同。

5.3.5. 问题思考

编译器的设计过程中需要进行的错误处理纷繁错杂，十分容易疏漏。我认为比较好的方式是以测试用例来进行驱动，通过各式各样的测试用例来找出程序中可能潜在的问题。我自己也在调试实践过程中通过不一样的测试用例找出了不少问题（上述问题皆是）。在语义分析的过程中，我统一用 raiseError 函数进行错误信息管理，能够很好地防止某一步由于源程序语义错误导致的后续语义分析过程中程序崩溃的问题。

6. 用户使用说明

6.1. 环境配置说明

以下脚本需要在代码目录的命令行中执行。

需要先行安装 pnpm, Python 及 pip。建议使用 virtual env。

6.1.1. 安装依赖

```
pnpm install
pip install -r requirements.txt
```

6.1.2. 运行

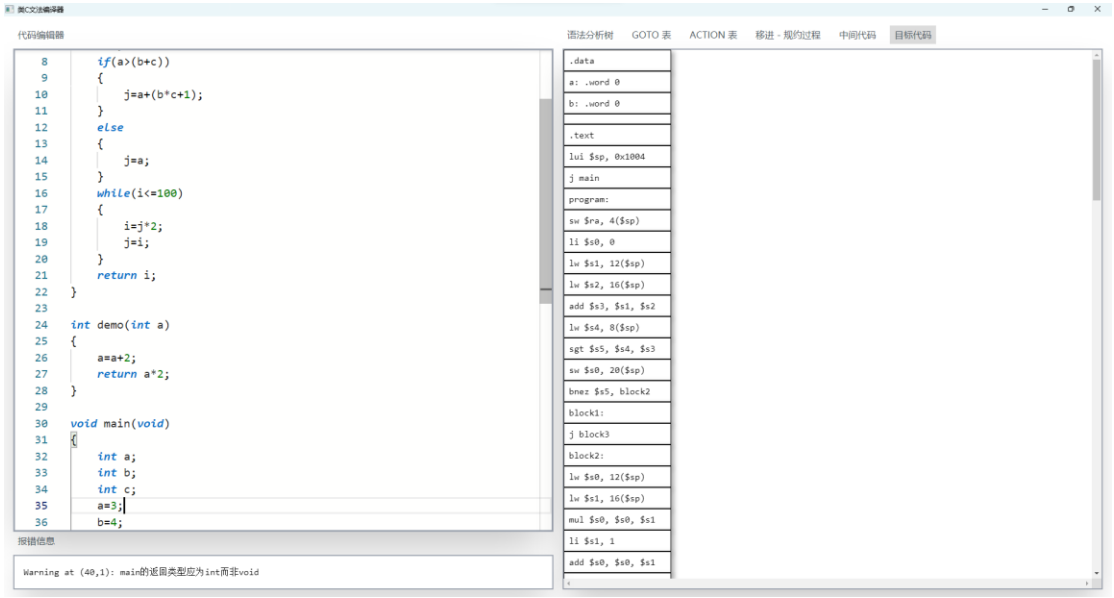
```
pnpm dev
```

6.1.3. 打包

```
pnpm build
```

6.2. 可执行文件使用说明

打开提供的“可执行文件”文件夹，双击运行 main.exe 即可。文件夹下的 productions.cfg 是供读取的文法规则，test.c 是默认打开的待编译源程序。程序运行后，会生成基本块划分后的信息 blockinfo.txt 与目标代码 code.asm，并可以通过图形界面可视化交互：



7. 课程设计总结

通过本次编译原理课程设计“类 C 编译器”的实现，我将编译原理课堂上学习到的词法分析、语法分析、语义分析等理论知识投入到实际运用中，在不断调试和实践的过程中给更好地理解了原本有些抽象的课程知识，并在将词法分析、语法分析、语义分析、目标代码生成几个部分连接到一起的过程中将原本模块化的离散知识串联成了有机的知识链条。

本次编译原理课程设计是基于上学期的词法分析、语法分析以及语义分析大作业而完成的。在上学期过程中，我就独立完成了词法分析器的设计与实现，并且参与到部分语法分析和语义分析的实现过程中。在本次课程设计的过程中，为求对整个流程的深入理解，我重构了自己未参与部分的语法分析和语义分析的全部代码，并继续实现了目标代码生成的部分。在整个项目的实现过程中，我遇到了各种各样的问题，并在调试的过程中对编译原理的知识有了更为深入的了解和认识。通过对本次类 C 语言编译器的设计和完成，我对所写的程序以及程序语言都有了更为本质的了解与认识。此外，我完成了选做项过程调用，了解和设计了 MIPS 中函数栈帧压栈的过程。很遗憾的是，由于时间受限，没能完成数组部分的实现和设计。此外，原本词法、语法、语义的部分都支持浮点数，也是受限于时间没能在目标代码生成的部分继续完善这一功能。在完成本项目的过程中，我感受到了对编译原理课程深深的热爱，但只能在有限的时间里将该项目尽自己所能完成至此。希望以后有机会还能进一步完善、改进这个项目，在实践中理解体悟编译原理知识，不断进步！

8. 参考文献

[1] Generation I C. Intermediate Code Generation[J]. 2006.

[2] Wilhelm R, Seidl H, Hack S. Compiler design: syntactic and semantic analysis[M]. Springer Science & Business Media, 2013.

[3] Hoe A V, Sethi R, Ullman J D. Compilers—principles, techniques, and tools[J]. 1986.

[4] 张素琴，吕映芝. 编译原理[M]., 清华大学出版社

[5] 蒋立源、康慕宁等，编译原理（第 2 版）[M]，西安：西北工业大学出版社

[6] 陈火旺,刘春林,谭庆平,等.程序设计语言编译原理:第 3 版 [M].国防工业出版社, 2008

[7] 孙冀侠, 迟呈英, 李迎春. LR (1) 语法分析的自动构造[J]. 鞍山科技大学学报, 2003, 26(2): 90-92.

[8] Microsoft (2021). Monaco Editor. Microsoft. https://microsoft.github.io/monaco-editor/

[9] Muhammad Yahya. (2021). Dynamically Building Nested List from JSON Data and Tree View with CSS3. Medium. https://medium.com/oli-systems/dynamically-building-nested-list-from-json-data-and-tree-view-with-css3-2ee75b471744

[10] Mozilla Developer Network. (2018). QWebEngineView.html. Mozilla Developer Network. https://developer.mozilla.org/en-US/search?q=QWebEngineView.html

[11] 戴桂兰, 张素琴, 田金兰, 等. 编译系统中间代码的一种抽象表示[J]. 电子学报, 2002, 30(S1): 2134.

[12] 李磊. C 编译器中间代码生成及其后端的设计与实现[D]. 电子科技大学, 2016.