同濟大學
TONGJI UNIVERSITY

类 Rust 编译器实现：
目标代码生成器设计与实现

课　程	编译原理课程设计
学　号	2252429
姓　名	蔡宇轩
学　院	计算机科学与技术学院
专　业	计算机科学与技术
教　师	卫志华、高珍

二〇二五 年　　八 月　　二十 日

目 录

1. 实验概述

1.1 目的与意义

Rust 语言是一门能够安全高效地编写系统级程序的语言，在内存安全方面具有显著优势。本次课程设计旨在通过深入研究、掌握并灵活运用编译原理课程的相关知识，全面完成一个类似于 Rust 语言的编译器的开发。

在大作业 1 和大作业 2 所完成的词法分析器、语法分析器和中间代码生成器的基础之上，本次课程设计需要继续完成编译器的目标代码生成部分。

实验目的如下：

（1）掌握使用高级程序语言实现一个一遍完成的、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 主要任务

本次课程设计要求使用高级语言实现一个类 Rust 语言编译器，可以提供词法分析、语法分析、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 Rust 语言的编译器。编码实现编译器的组成部分；

（2）要求的类 Rust 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）使用语法制导翻译，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 Rust 语言源程序，输出中间代码表示的程序；

（5）要求输入类 Rust 语言源程序，输出目标代码（可汇编执行）的程序。

- **拓展功能：**

除了要求实现的基本功能以外，本次课程设计还额外实现了包括复杂词法分

析、嵌套注释、拓展节点的语法分析和语义分析（包括函数调用、引用、数组、元组等）、基本块划分与优化等拓展功能；此外，可视化应用界面具备防抖效果，并能实时展示结果而无需用户等待。

2. 需求分析

2.1 程序输入

2.1.1 源程序输入

后端程序有两种处理源程序输入的方式：（1）从根目录下的 rust 文件夹中读取 test.rs 作为待处理的源程序（2）从可视化窗口左侧的编辑器中读取内容作为待处理的源程序。在运行后端代码前，可以根据需要选择命令行（对应第一种源程序输入方式）或可视化界面（对应第二种源程序输入方式）使用应用。

程序输入的反例如下：

```
    fn program_pre() {

    }

    fn program(mut a:i32, mut n:i32, mut x:i32,mut y:i32, mut aa:[i32;3],
mut aaa:(i32,i32)) -> i32 {
    a=32;
        let mut c:i32=1;
        let mut b=1;
        let d:i32;
        let e=2;

        a;
        (0);
        ((a));
        (((7)));

        1*2/3;
        4+5/6;
        7<8;
        1*2+3*4<4/2-3/1;

        program_pre();
```

```
if a>0 {
        b = a+1;
} else if a<0 {
        b = a-1;
} else {
        b = 0;
}

while n>0 {
        n=n-1;
    break;
}

let mut f:&mut i32=&mut c;
let g:i32=*f;
*f=2;

let mut z={
        let mut t=x*x+x;
        t=t+x*y;
        t
    };

let mut z=if a>0 {
        1
    } else {
        0
    };

let mut m=loop {
        break 2;
    };

let mut bb:i32=aa[0];
aa[0]=1;
let mut cc:[i32;3];
cc=[1,2,3];

let dd:(i32,i32,i32);
dd=(1,2,3);
let mut j:i32=dd.0;
dd.0=1;
```

```
        return 1;
    }
```

2.1.2 文法输入

将编译器可识别的文法保存在 grammar.txt 文件中，在应用加载时读取并进行文法规则的解析。在本次课程设计中，完成了对所有基本和拓展节点文法的词法分析和语法分析，并按照申优标准完成了大部分的拓展语义分析、中间代码生成和目标代码生成。后续若需要强化编译器功能，可以在文法文件中进一步添加产生式。

文法输入的范例如下：

```
    Program -> M DeclarationString
    M -> Epsilon
    0.1
    Identifier_inside -> mut ID
    0.2
    Type -> i32
    0.3
    Assignable_element -> ID
    1.1
    DeclarationString -> Epsilon | Declaration DeclarationString
    Declaration -> Function_Declaration
    Function_Declaration -> N Function_Head_Declaration Line_Block
    Function_Head_Declaration -> fn ID ( Formal_Parameter_List )
    Formal_Parameter_List -> Epsilon
    Line_Block -> { Line_String }
    Line_String -> Epsilon
    1.2
    Line_String -> Line Q Line_String
    Q -> Epsilon
    Line -> ;
    N -> Epsilon
    1.3
    Line -> Return_Line
    Return_Line -> return ;
    1.4
    Formal_Parameter_List -> Formal_Parameter | Formal_Parameter , Formal
_Parameter_List
```

```
    Formal_Parameter -> Identifier_inside : Type
    3.1
    Line -> Expression ;
    Expression -> Plus_Expression
    Plus_Expression -> Term
    Term -> Factor
    Factor -> Element
    Element -> NUM | Assignable_element | ( Expression )
    1.5
    Function_Head_Declaration -> fn ID ( Formal_Parameter_List ) -> Type
    Return_Line -> return Expression ;
    2.1
    Line -> Identifier_Line
    Identifier_Line -> let Identifier_inside : Type ;
    Identifier_Line -> let Identifier_inside ;
    2.2
    Line -> Assign_Line
    Assign_Line -> Assignable_element = Expression ;
    2.3
    Line -> Identifier_Assign_Line
    Identifier_Assign_Line -> let Identifier_inside : Type = Expression ;
    Identifier_Assign_Line -> let Identifier_inside = Expression ;
    3.2
    Expression -> Expression Comparison Plus_Expression
    Plus_Expression -> Plus_Expression AS_operator Term
    Term -> Term MD_operator Factor
    Comparison -> < | <= | > | >= | == | !=
    AS_operator -> + | -
    MD_operator -> * | /
    3.3
    Element -> ID ( Real_Parameter_List )
    Real_Parameter_List -> Epsilon | Expression | Expression , Real_Param
eter_List
    4.1
    Line -> IF_Line
    IF_Line -> if Expression B Q Line_Block ELSE_Part
    ELSE_Part -> Epsilon | P else Q Line_Block
    4.2
    ELSE_Part -> P else if Q Expression B Q Line_Block ELSE_Part
    B -> Epsilon
    P -> Epsilon
    5.1
    Line -> Cycle_Line
    Cycle_Line -> While_Line
```

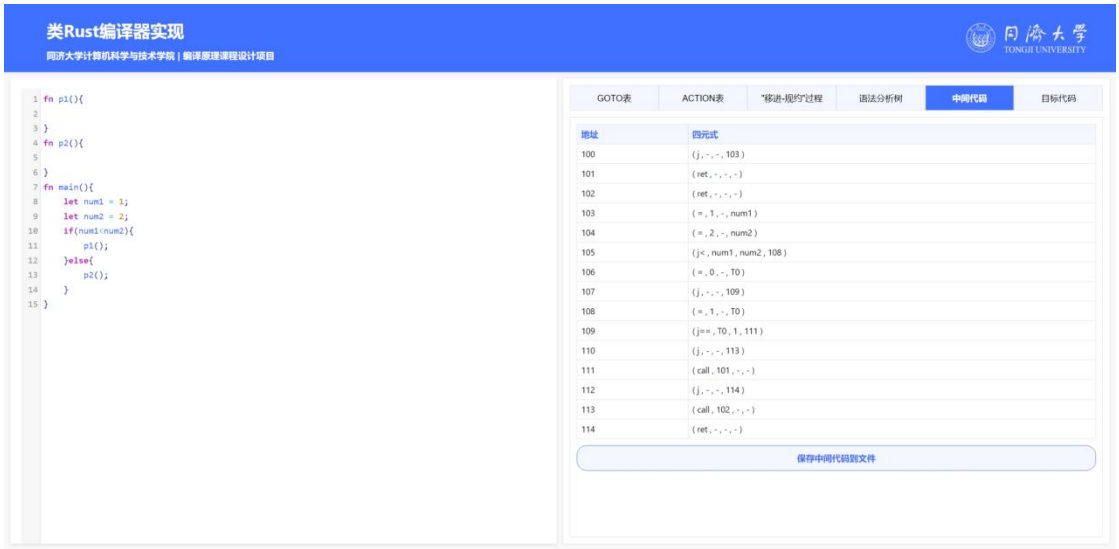
```
    While_Line -> while Q Expression B Q Line_Block
    5.2
    Cycle_Line -> For_Line
    Identifier_Iterative_Structure -> for Identifier_inside in Iterative_
Structure
    For_Line -> Identifier_Iterative_Structure Q Line_Block
    Iterative_Structure -> Expression .. Expression
    5.3
    Cycle_Line -> Loop_Line
    Loop_Line -> loop Q Line_Block
    5.4
    Line -> break ; | continue ;
    6.1
    Identifier_inside -> ID
    6.2
    Factor -> * Factor | & mut Factor | & Factor
    Type -> & mut Type | & Type
    7.1
    Expression -> Function_Expression_Block
    Function_Expression_Block -> { Function_Expression_String }
    Function_Expression_String -> Expression | Line Function_Expression_S
tring
    7.2
    Function_Declaration -> Function_Head_Declaration Function_Expression
_Block
    7.3
    Expression -> Choose_Expression
    Choose_Expression -> if Expression Function_Expression_Block else
    Function_Expression_Block
    7.4
    Expression -> Loop_Line
    Line -> break Expression ;
    8.1
    Type -> [ Type ; NUM ]
    Factor -> [ Array_List ] | Array_Element
    Array_List -> Epsilon | Expression | Expression , Array_List
    8.2
    Assignable_element -> Element [ Expression ]
    Iterative_Structure -> Element
    9.1
    Type -> ( Tuple_Inside )
    Tuple_Inside -> Epsilon | Type , Type_List
    Type_List -> Epsilon | Type | Type , Type_List
    Factor -> ( Tuple_Assign_Inside )
```

```
    Tuple_Assign_Inside -> Epsilon | Expression , Tuple_Element_List
    Tuple_Element_List -> Epsilon | Expression | Expression , Tuple_Eleme
nt_List
    9.2
    Assignable_element -> Factor . NUM
```

2.2 程序输出

2.2.1 中间过程输出

在可视化应用界面右侧，对编译的中间过程进行展示，包含 ACTION 表和 GOTO 表、语法分析树、“移进-规约”过程、中间代码以及目标代码的渲染。



2.2.2 目标代码输出

在目标代码界面的下方，可选择是否将当前目标代码保存到文件中。若选择，将在浏览器中自动下载“code.asm”并保存到本机的下载路径中。

目标代码采用 Mips 汇编格式，范例如下：

```
    .data

    .text
        lui $sp, 0x1004
        j main
    p1:
        sw $ra, 4($sp)
```

```
    lw $ra, 4($sp)
    jr $ra
p2:
    sw $ra, 4($sp)
    lw $ra, 4($sp)
    jr $ra
main:
    li $s0, 1
    li $s1, 2
    blt $s0, $s1, block2
block1:
    li $s0, 0
    sw $s0, 0($sp)
    j block3
block2:
    li $s0, 1
    sw $s0, 0($sp)
block3:
    lw $s0, 0($sp)
    li $s1, 1
    beq $s0, $s1, block5
block4:
    j block7
block5:
    sw $sp, 4($sp)
    addi $sp, $sp, 4
    jal p1
    lw $sp, 0($sp)
block6:
    j block8
block7:
    sw $sp, 4($sp)
    addi $sp, $sp, 4
    jal p2
    lw $sp, 0($sp)
block8:
    j end
end:
```

2.3 程序功能

2.3.1 基本功能

（1）输入一个类 Rust 的源程序；

（2）进行词法分析，以高亮形式展示词法分析的结果；

（3）从配置文件读入文法基本信息，即文法的产生式集合等（文法的起始符号默认为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、FIRST 集合、ACTION 表和 GOTO 表；

（4）在词法分析基础上进行语法分析，测试该源程序是否为符合既定文法的源程序。若符合则显示“移进-规约”分析过程和语法分析树，若不符合则给出错误提示并可视化；

（5）在“移进-规约”分析的过程中，针对当前执行的动作进行语义分析，并同步进行中间代码生成，若存在语义上的错误，则在进行可视化展示的同时终止中间代码的生成；

（6）针对中间代码（四元式）以函数为单位划分基本快，计算每个基本块的出口活跃变量集合、每条四元式的变量活跃与待用信息；

（7）根据活跃信息与待用信息分配寄存器，生成汇编格式的目标代码。

2.3.2 拓展功能

（1）复杂词法分析与高亮：

在大作业 1 中，在原有词法符号的基础上深入研究 Rust 语言规则，增加了字符、字符串和“@”等一系列特殊符号的词法分析。此外在词法分析上实现了嵌套注释，当块注释包含多层，输入缓冲区能够记录当前层数并识别每层注释的开始与结束，跟踪 DFA 状态转移，正确识别注释与程序代码。

在大作业 2 中，优化了代码的词法分析逻辑和编辑区的高亮函数，使得类 Rust 编译器可以识别更为复杂的词法情况，如跨行注释、行内注释、跨行字符串等。

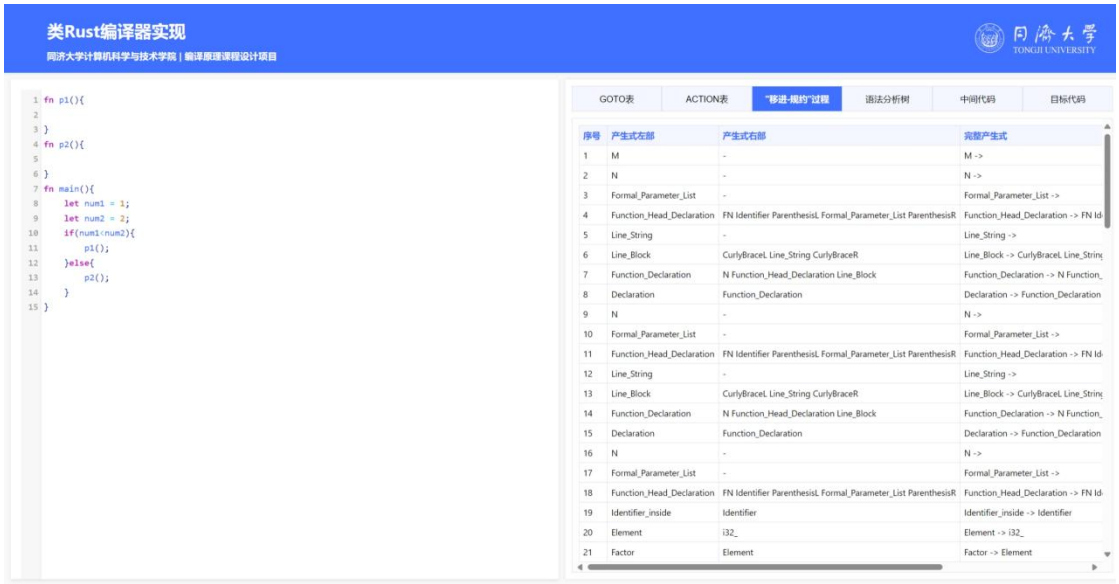
（2）拓展规则：

在词法和语法分析阶段，完成了题目要求的全部基础规则和拓展规则。

在语义分析与目标代码生成阶段，按照申优要求完成了包括 6.2 在内及之前的全部基础和拓展文法的语义分析与中间代码生成，并能对应生成出目标代码。

（3）实时前端界面：

基于类 Rust 编译工具，采用 Web 技术开发了支持高亮展示词法分析结果、语法错误定位标红、语法分析树渲染、中间代码和目标代码展示等功能的 Web 应用。除此之外，前端界面需通过多子进程调用技术实现防抖和请求加速，从而为用户带来更好的使用体验。



2.4 测试数据

本次课程设计的测试样例涵盖多种正确样例、错误样例，详见后文运行结果部分。针对错误样例，主要包含：**词法错误**（出现不合法的非 Rust 语言符号或文字）、**语法错误**（不按照提供的文法规则书写源程序，如括号不匹配、关键字错误等）、**语义错误**（在变量定义、赋值、函数调用与返回等多个层面上出现的源程序错误）等。

3. 系统方案设计

3.1 模块划分

本次课程设计所需完成的类 Rust 编译器的编译过程可以划分为 5 个主要模块：

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**。为了方便最后一阶段的工作，将目标代码生成器中拆分出**基本块划分**这一预备步骤。

此外，根据一遍扫描要求，词法分析器将作为子程序被语法分析器调用，同时语法分析器将采用语法指导的翻译技术进行语义分析。在语法分析的过程中，语义分析和中间代码生成是同步进行的。

3.2 数据结构

3.2.1 词法分析器模块

（1）输入缓冲器 InputBuffer：

重点成员变量（函数）	说明
source	完整源代码
clean_code	删除注释后的代码
line_breaks	删除注释后的换行下标
index	扫描器 scanner 当前指针，指向下一个将读取的位置

（2）词法单元 Token：

重点成员变量（函数）	说明
type	词法单元种类，枚举类型
value	词法单元的值，仅在有取值含义时有效
line	所在行号
column	所在列号
length	词法单元的字符长度

（3）扫描器 Scanner：

重点成员变量（函数）	说明
ReservedWordsTable[]	回溯 TokenType 所用数组
input	输入缓冲区
ch	存放当前读入字符
strToken	存放单词的字符串
GetChar()	读取下一个字符的函数

GetBC()	滤除各种空字符（串）的函数
Retract()	搜索指针回退一个字节的函数
Concat()	把 ch 中的字符拼入 strToken 的函数
Clear()	清空 strToken，为下一轮识别做准备的函数

3.2.2 语法分析器模块

（1）产生式符号 Symbol：

重点成员变量（函数）	说明
isTerminal	布尔型变量，用于判断是否为终结符
name	符号名称

（2）产生式 Production：

重点成员变量（函数）	说明
left	产生式左侧符号
right	vector 类型，产生式右侧符号集合

（3）LR1 项目 LR1Item：

重点成员变量（函数）	说明
productionIndex	与本项目对应的产生式符号
dotPosition	当前扫描指针在产生式中的位置
lookahead	下一个即将入栈的符号类型，存储为 tokenType 类型

（4）LR1 分析表的表项 ActionTableEntry：

重点成员变量（函数）	说明
act	“移进-规约”过程中需要做出的动作
num	“移进-规约”过程中使用的产生式序号

（5）语法分析错误 ParseError：

重点成员变量（函数）	说明
line	语法错误所在行号
column	语法错误所在列号

length	语法错误所占的字符长度
message	语法错误信息

（6）语法分析器 Parser：

重点成员变量（函数）	说明
lexer	词法分析器，作为语法分析器的子程序
semanticer	语义分析器，作为语法分析器的子程序
productions	所有产生式
nonTerminals	产生式中出现的所有非终结符<名称，序号>
firsts	所有非中终结符的首部集合
Itemsets	所有 LR1 项目集
actionTable	横坐标 Itemsets 下标，纵坐标 TokenType 值
gotoTable	横坐标 Itemsets 下标，纵坐标 nonTerminals 下标
reduceProductionLists	规约过程中使用的产生式
parseErrors	语法错误

3.2.3 语义分析器模块

（1）符号（变量）表表项 SymbolTableEntry：

重点成员变量（函数）	说明
ID	符号（变量）名称
kind	符号（变量）为常量或变量
type	符号（变量）类型
isNormal	符号（变量）是否为形参
isAssigned	符号（变量）是否已赋值
addr	符号（变量）的存储地址

（2）符号（变量）表 SymbolTable：

重点成员变量（函数）	说明
table	本层符号表，存储所有符号列表
width	所有名字占用的总宽度
prev	上一层符号表
isExist	查找（仅本层）：给出名字确定它是否在表中

put	填入（仅本层）：填入新名字和相关信息
get	访问（所有层）：根据名字，访问符号表项
update	更新（所有层）：返回符号是否存在
erase	删除（仅本层）：删除符号表中后添加的符号

（3）函数表表项 ProcedureTableEntry：

重点成员变量（函数）	说明
ID	函数名称
returnType	返回类型
subProcedureTable	子函数函数表指针
paraType	形参类型
addr	函数起始四元式地址
symbolTable	当前函数对应的符号表指针

（4）四元式 Quadruple：

重点成员变量（函数）	说明
op	四元式运算符
arg1	四元式源操作数 1
arg2	四元式源操作数 2
result	四元式目的操作数

（5）节点基类 ASTNode：

重点成员变量（函数）	说明
line	符号起始字符的行号
column	符号起始字符的列号

以此节点基类衍生出表达式节点 ExprNode（同时用于表示产生式中的<表达式>、<加法表达式>、<项>、<因子>和<元素>）、标识符节点 Id、参数列表节点 ParaListNode、数值节点 I32Node、非终结符中间节点 BridgeNode（同时用于表示产生式中的<变量声明内部>、<类型>和<可赋值元素>）、语句节点 Stmt 同时用于表示产生式中的<语句>、<语句串>、<语句块>和<if 语句>）、运算符节点 Op、for 变量迭代结构节点 ForIri、Else 语句节点 ElseStmt。

此外 Q、B、P 三个衍生节点类分别用于表示下一条四元式地址、该四元式的真假链 truelist 和 falselist、nextlist。

（6）语义分析器 SemanticAnalyzer：

重点成员变量（函数）	说明
START_STMT_ADDR	四元式起始地址
qList	产生的所有四元式集合
tblptr	保存各外层过程的符号表指针
proptr	存放各嵌套过程的当前相对地址
procedureList	保存各外层过程的函数表指针
nodeStack	随语法分析过程产生的结点
semanticErrors	归约过程中遇到的语义错误
nextstat	输出序列中下一条四地址语句的地址索引
refCount	引用跟踪计数<不可变引用计数，可变引用计数>
makelist()	创建一个仅含 quad 的新链表的函数
merge()	将 list1 和 list2 两条链合并为一的函数
mkleaf()	建立新 ASTNode 节点的函数
backpatch()	回填函数

3.2.4 基本块划分器模块

（1）变量待用/活跃信息 SymbolInfo：

重点成员变量（函数）	说明
next_use	变量下一次使用的位置，-1 表示未使用
is_active	变量是否活跃

（2）四元式中的变量待用/活跃信息 QuadrupleInfo：

重点成员变量（函数）	说明
arg1_info	四元式中源操作数 arg1 的变量活跃信息
arg2_info	四元式中源操作数 arg2 的变量活跃信息
result_info	四元式中目的操作数 result 的变量活跃信息

（3）基本块 Block：

重点成员变量（函数）	说明
name	基本块名称
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集合
next1	后续基本块 1（分支结构）
next2	后续基本块 2（分支结构）
use_set	块内首次出现为引用变量的集合
def_set	块内首次出现为定值变量的集合
in_set	入口活跃变量集合
out_set	出口活跃变量集合
symbolInfoTable	基本块内变量的活跃/待用信息表
codesInfo	对应每个四元式的变量活跃/待用信息

（4）基本块划分器 BlockDivider：

重点成员变量（函数）	说明
qList	待划分的四元式集合
start_address	起始地址
block_cnt	已经划分生成的基本块计数
func_blocks	函数与基本块之间的映射关系
blocks	指向所有基本块指针
divideBlocks()	依据函数表和四元式划分基本块的函数
computeBlocks()	计算各基本块的变量活跃信息等的函数

3.2.5 目标代码生成器模块

（1）寄存器管理器 RegManager：

重点成员变量（函数）	说明
RValue	寄存器中存储变量的标记
AValue	变量所处寄存器的标记
regs_num	可用寄存器数量
frame_size	当前栈帧大小
free_regs	当前空闲寄存器列表
memory	变量内存地址映射

func_vars	局部变量集合
data_vars	全局变量集合（本次要求为空）

（2）参数活跃信息 ParamInfo：

重点成员变量（函数）	说明
name	参数名称
is_active	参数是否活跃

（3）目标代码生成器 CodeGenerator：

重点成员变量（函数）	说明
regMgr	寄存器管理器
codes	目标代码
func_blocks	函数与基本块之间的映射关系
func_vars	局部变量集合
data_vars	全局变量集合（本次要求为空）
param_list	函数调用参数列表

3.3 主程序流程

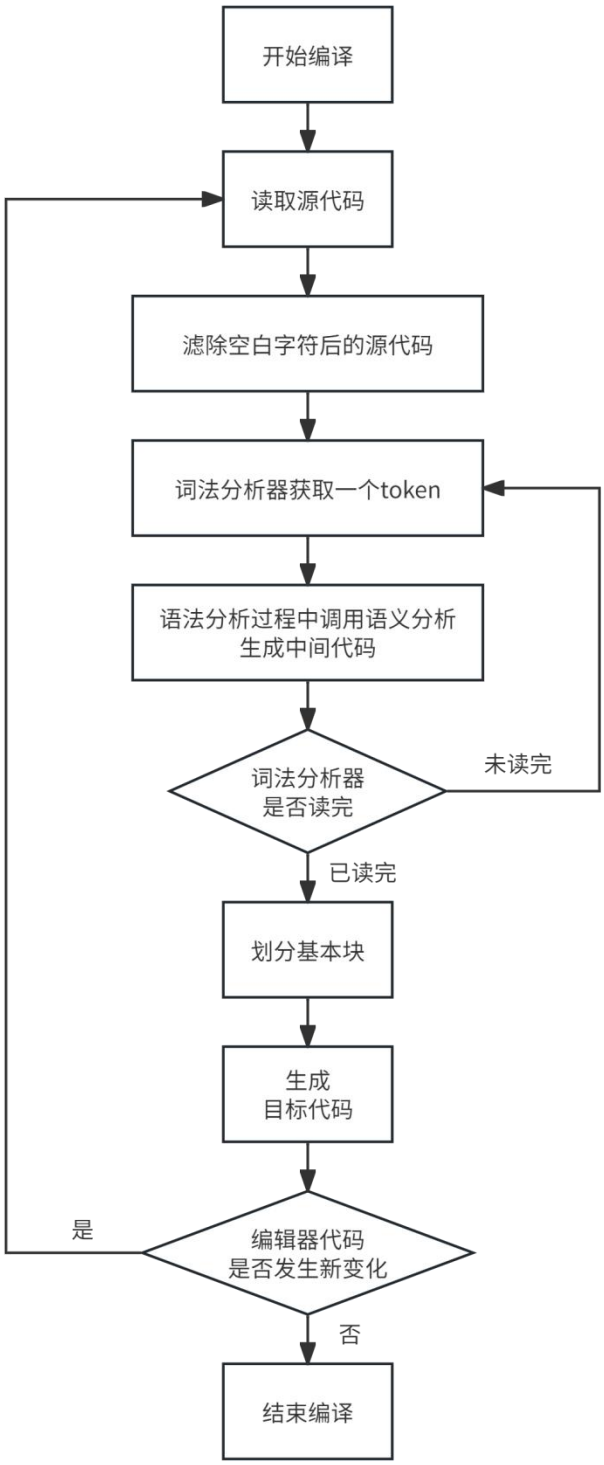
整个主程序的流程按照词法分析、语法分析、语义分析、目标代码生成的过程顺序进行。其中，词法分析和语义分析作为子程序被语法分析调用。

整个编译过程可以总结如下：每次将从去除了空白注释的源代码中，由词法分析器读取一个 token（词法单元）存入语法分析器中。然后由语法分析器对当前的 token 进行语法分析，同时调用语义分析并生成四元式（中间代码）。

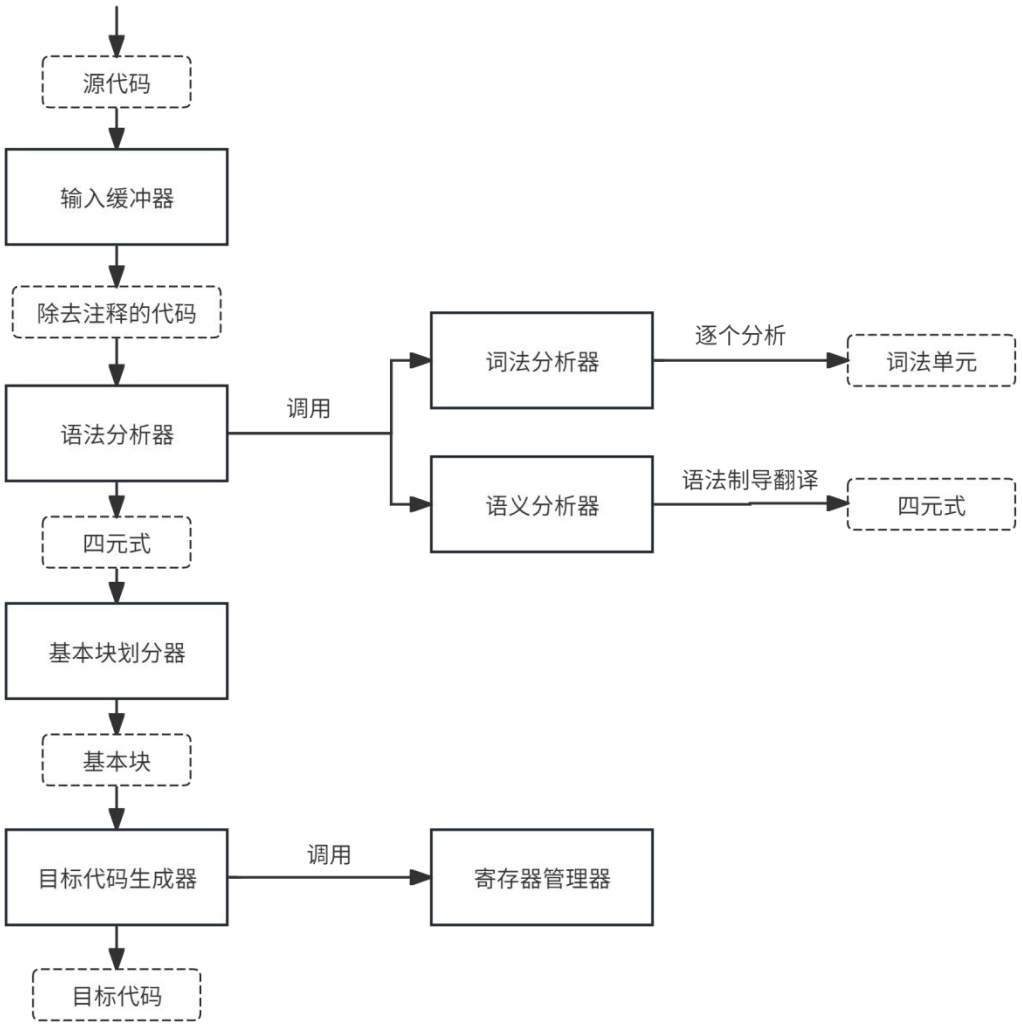
在所有词法单元读入完毕后，语法分析和语义分析也同步完成，所有四元式生成完毕。

基本块划分器接受四元式并划分基本块、明确各函数与基本块之间的映射关系 func_blocks。最后由目标代码生成器接受 func_blocks 和对应的四元式，生成针对 Mips 汇编语言的目标代码。

main.cpp 是对整个编译过程的主程序实现，其流程图如下所示：



整个编译过程中各模块之间的调用关系，如下图所示：



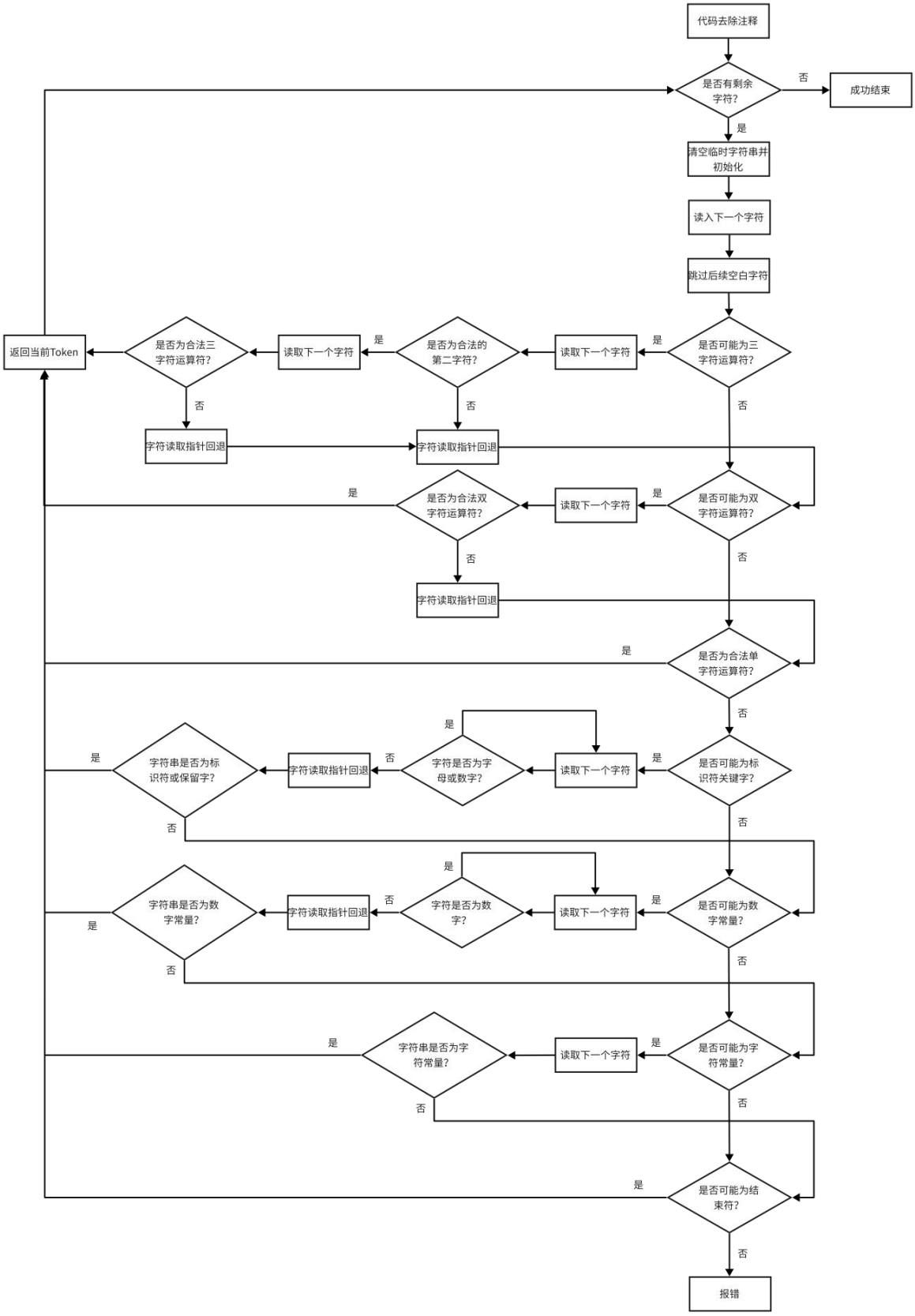
4. 详细设计

4.1 词法分析器

4.1.1 词法分析流程基本框图

按照课程设计要求，词法分析程序采用一遍扫描方法，作为语法分析器的成员在语法分析过程中进行调用。词法分析器在代码实现上本质是一个 DFA，每次只返回一个 token（词法单元），而不是像两遍扫描的方法一次性生成 token 列表。

词法分析的具体流程如下图所示：


代码去除注释
是否有剩余字符?
否
成功结束
是
清空临时字符串并初始化
读入下一个字符
跳过后续空白字符
是否可能为三字符运算符?
是
读取下一个字符
是否为合法的第二字符?
是
读取下一个字符
是否为合法三字符运算符?
是
返回当前Token
否
字符读取指针回退
否
字符读取指针回退
否
是否可能为双字符运算符?
是
读取下一个字符
是否为合法双字符运算符?
是
否
字符读取指针回退
否
是否为合法单字符运算符?
是
否
是否可能为标识符关键字?
是
读取下一个字符
字符是否为字母或数字?
是
否
字符读取指针回退
字符串是否为标识符或保留字?
是
否
是否可能为数字常量?
是
读取下一个字符
字符是否为数字?
是
否
字符读取指针回退
字符串是否为数字常量?
是
否
是否可能为字符常量?
是
读取下一个字符
字符串是否为字符常量?
是
否
是否可能为结束符?
是
否
报错

4.1.2 重点函数分析

- **空白字符过滤函数（Filter_comments）：**

本函数实现源代码中的注释部分的去除，过滤掉所有单行注释和块注释，最终得到去掉注释的“干净代码”。函数实现过程采用了自动机的状态转换的思路，首先定义出一个枚举列举出一系列在读取代码过程中的可能状态：普通状态、遇到单个‘/’字符的状态、单行注释状态、块注释状态等等。之后从头开始逐个字符地遍历源代码（source），每次读取到对应的字符时，进行状态的转换，在每个对应的状态下执行对应的功能，跳过注释的标志字符，忽略在注释中的字符，将实际代码输出（clean_code）。

Filter_comments 函数的算法总结如下：

```
// 初始化:
    lastch = '\0'  // 上一个字符
    currch = '\0'  // 当前字符
    sourceSize = length of source  // 源代码长度
    curr_state = normal  // 初始状态为正常
    in_block_nested = 0  // 块注释嵌套计数
// 遍历源代码每个字符
for i from 0 to sourceSize - 1:
    currch = source[i]  // 获取当前字符
        // 根据不同状态处理字符
    if curr_state == single_slash:
        if currch == '/':
            curr_state = in_line_comment
        else if currch == '*':
            curr_state = in_block_comment
        else:
            append '/' to clean_code
            curr_state = normal
    else if curr_state == in_line_comment:
        if currch in ['\r', '\n']:
            curr_state = normal
    else if curr_state == in_block_comment:
        if currch == '*':
            curr_state = in_block_comment_halfend
        else if currch == '/':
            curr_state = in_block_comment_slash
    else if curr_state == in_block_comment_halfend:
```

```
            if currch == '/':
                if in_block_nested > 0:
                    in_block_nested -= 1
                else:
                    curr_state = normal
            else if currch != '*':
                curr_state = in_block_comment
        else if curr_state == in_block_comment_slash:
            if currch == '*':
                curr_state = in_block_comment
                in_block_nested += 1
            else if currch != '/':
                curr_state = in_block_comment
            // 更新状态和处理字符
        if currch == '/':
            curr_state = single_slash
        else:
            if clean_code is not empty and (currch is redundant whitespace):
                continue
            append currch to clean_code
                    // 处理换行符
            if currch == '\r' and next character is '\n':
                record line break and skip next character
            else if currch in ['\r', '\n']:
                record line break
        lastch = currch  // 更新上一个字符
    // 处理未结束的注释
    if curr_state in [in_block_comment, in_block_comment_halfend,
in_block_comment_slash]:
        output unterminated block comment error
    else if curr_state == single_slash:
        append '/' to clean_code
```

- **词法单元扫描函数（Scan）：**

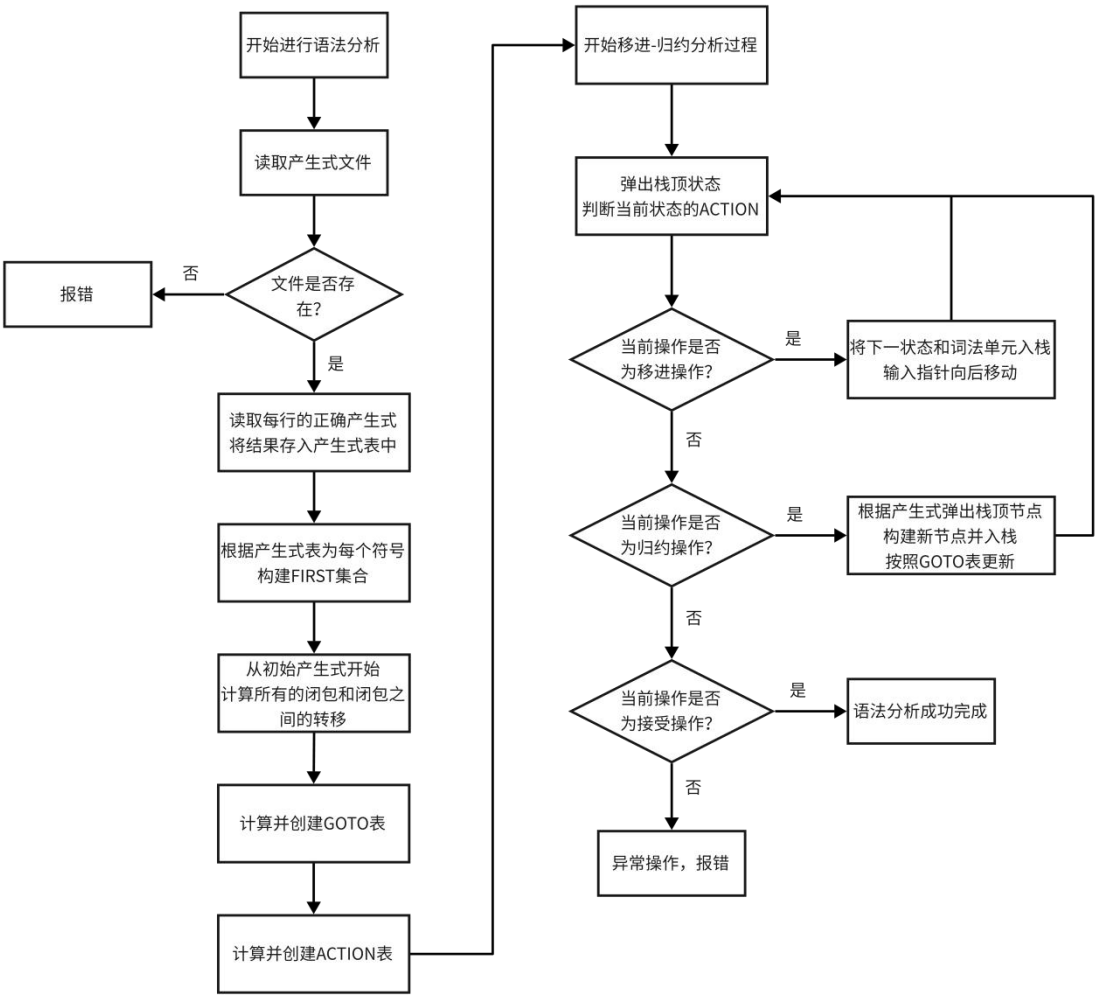
本函数扫描并识别下一个词法单元（token），通过逐个字符地分析输入流，并根据不同的字符组合来判断对应的语言成分，之后返回响应的 token 对象。在扫描过程中，依次调用 GetChar()和 GetBC()获取下一个字符并跳过后面的所有空白字符；然后会根据不同的关键字或操作符等进行逐字符的判断或回退。

Scan 函数的算法总结如下：

```
Get first char  // 获取首个字符
// 处理多字符运算符
if ch is special:
    check patterns  // 检查模式
    return 对应 Token
// 处理标识符和关键字
if ch is letter:
    read letters/digits  // 读取字母/数字
    check keyword  // 检查是否为关键字
    return Identifier or keyword Token
// 处理整数
if ch is digit:
    read digits  // 读取数字
    return i32_ Token
// 处理单字符运算符和分隔符
if ch is operator/delimiter:
    return 对应 Token
// 处理字符串
if ch is '"':
    read till closing '"'  // 读取直到闭合的引号
    handle escapes  // 处理转义字符
    return string_ Token
// 处理字符常量
if ch is '\'':
    read till closing '\''
    return char_ Token
// 处理文件结束
if EOF:
    return End Token
// 处理错误
else:
    error 处理
```

4.2 语法分析器

4.2.1 语法分析流程基本框图



4.2.2 重点函数分析

- **首部集 FIRST 计算函数（ComputeFirsts）：**

法分析时对符号串的判断。整个函数的核心思路就是用循环的方式，每次循环遍历所有的产生式，对 FIRST 集合进行更新，使用 changed 标志位控制迭代，直到所有 FIRST 集合不再变化为止。

ComputeFirsts 函数的算法总结如下：

```
bool changed = true; // 标记本轮循环是否有 FIRST 集合改变
```

```
    firsts.resize(nonTerminals.size()); // 初始化 FIRST 集合数组

    while (changed) { // 循环直到没有变化
        changed = false;
        for (Production prod : productions) { // 遍历所有产生式
            Symbol left = prod.left;
            if (left.isTerminal) continue; // 跳过终结符
             // 当前非终结符的 FIRST 集合
            set<TokenType>& leftFirst = firsts[left.nonterminalId];
            if (prod.right.empty()) { // 产生式右侧为空
                if (leftFirst.insert(EPSILON).second) // 添加ε
                    changed = true;
            }
            else { // 产生式右侧不为空
                bool allepsilon = true;
                for (Symbol sym : prod.right) { // 遍历右侧符号
                    if (sym.isTerminal) { // 遇到终结符
                        if (leftFirst.insert(sym.terminalId).second) // 添
加终结符
                            changed = true;
                        allepsilon = false;
                        break;
                    }
                    else { // 遇到非终结符
                        bool noepsilon = true;
                        for (TokenType first : firsts[sym.nonterminalId]) {
                         // 遍历非终结符的 FIRST 集合
                            if (first != EPSILON) {
                                if (leftFirst.insert(first).second)
                                  // 添加非终结符的 FIRST 元素（排除ε）
                                    changed = true;
                            }
                            else
                                noepsilon = false;
                        }
                        if (noepsilon) { // 如果非终结符的 FIRST 集合不包含ε
                            allepsilon = false;
                            break;
                        }
                    }
                }
                if (allepsilon && leftFirst.insert(EPSILON).second)
                 // 如果所有符号的 FIRST 集合都包含ε，则添加ε
```

```
                changed = true;
            }
        }
    }
```

●LR1 分析表构建函数（Items）：

本函数是 LR (1) 语法分析表构建的核心函数，负责生成所有 LR (1) 项集并构建 ACTION 和 GOTO 表。在函数执行过程中会调用 GOTO 函数用于计算 GOTO 表，而 ACTION 表则直接由本函数构造完成。

Items 函数的算法总结如下：

```
    // 清空项目集
    Itemsets.clear();
    // 初始化第一个项目集为闭包{S'→·S,$}
    Itemsets.push_back(Closure(LR1ItemSet{ set<LR1Item>{LR1Item(0, 0,
TokenType::End)} });
    // 获取非终结符数量
    int nonTerminalsNum = nonTerminals.size();
    // 初始化 action 表
    actionTable.emplace_back(vector<ActionTableEntry>(TokenType::End +
1));
    // 初始化 goto 表
    gotoTable.emplace_back(vector<int>(nonTerminalsNum, -1));
    bool added = true;  // 标记是否有新项目集加入
    while (added) {  // 循环直到无新项目集加入
        added = false;
        for (int index = 0; index < Itemsets.size(); ++index) {  // 遍历每
个项目集
            LR1ItemSet I = Itemsets[index];
            // 处理终结符
            for (int i = TokenType::None + 1; i <= TokenType::End; ++i) {
                Symbol X(TokenType(i));
                LR1ItemSet gotoset = Goto(I, X);
                if (!gotoset.items.empty() && find(Itemsets.begin(),
Itemsets.end(), gotoset) == Itemsets.end()) {
                    Itemsets.push_back(gotoset);  // 添加新项目集
                    actionTable.emplace_back(vector<ActionTableEntry>(Tok
enType::End + 1));
```

```
                    gotoTable.emplace_back(vector<int>(nonTerminalsNum,
-1));
                    added = true;
                }
                if (!gotoset.items.empty()) {
                    writeActionTable(index, i,
ActionTableEntry{Action::shift, GetItemSetIndex(gotoset)});
                }
            }
            // 处理非终结符
            for (auto& sym : nonTerminals) {
                Symbol X = GetNonTerminal(sym.first);
                LR1ItemSet gotoset = Goto(I, X);
                if (!gotoset.items.empty() && find(Itemsets.begin(),
Itemsets.end(), gotoset) == Itemsets.end()) {
                    Itemsets.push_back(gotoset);
                    actionTable.emplace_back(vector<ActionTableEntry>(Tok
enType::End + 1));
                    gotoTable.emplace_back(vector<int>(nonTerminalsNum,
-1));
                    added = true;
                }
                if (!gotoset.items.empty()) {
                    gotoTable[index][sym.second] =
GetItemSetIndex(gotoset);
                }
            }
            addReduceEntry(index);  // 添加规约项
        }
    }
```

- **语法分析过程核心函数（SyntaxAnalysis）：**

本函数是 LR (1) 语法分析的核心实现，完整串联起整个语法分析流程。

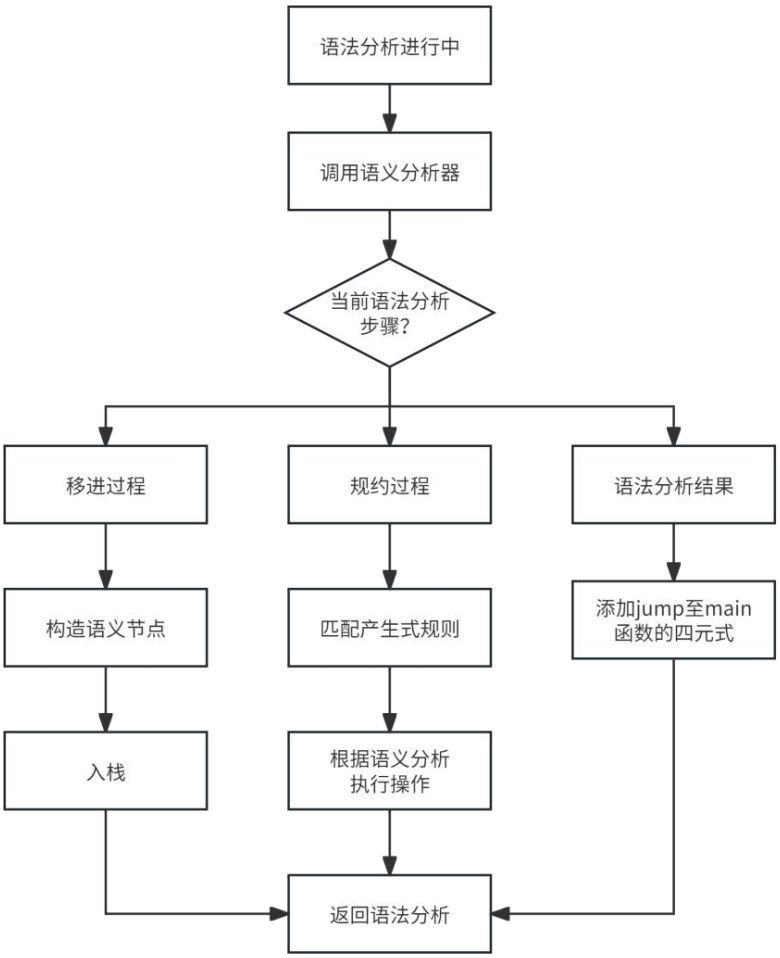
```
void Parser::SyntaxAnalysis()
{
    stack<int> stateStack;//状态栈
    stack<Symbol> tokenStack;//符号栈
    stateStack.push(0);//拓广文法的起始变元对应 LR1 项目集开始
```

```
        tokenStack.push(TokenType::End);

        GetToken();//令 look 为w#的第一个符号
        while (true) {
            int s = stateStack.top();//令 s 是栈顶的状态
            ActionTableEntry action = actionTable[s][look.type];//s 和 look
对应行动
            if (action.act == Action::shift) {//ACTION[s,a]=移入新状态（sx）
                stateStack.push(action.num);//将新状态压入栈中
                tokenStack.push(look.type);
                GetToken();
            }
            else if (action.act == Action::reduce) {//ACTION[s,a]=归约 A→β
                int betalength = productions[action.num].right.size();//|β|
                for (int i = 0; i < betalength; ++i) {//从栈中弹出|β|个符号;
                    stateStack.pop();
                    tokenStack.pop();
                }
                int t = stateStack.top();//令 t 为当前的栈顶状态;
                unsigned int Aid =
productions[action.num].left.nonterminalId;//A
                stateStack.push(gotoTable[t][Aid]);//将 GOTO[t,A]压入栈中
                tokenStack.push(Symbol(Aid,
productions[action.num].left.name));

    reduceProductionLists.push_back(productions[action.num]);//输出产生式
A→β
            }
            else if (action.act == Action::accept)
                break;//语法分析完成
            else if (action.act == Action::error) {
                //错误展示
                lexer.ProcError(string(TokenTypeToString(look.type)));
                //记录错误
                AddParseError(look.line, look.column, look.length,
```

```
string(TokenTypeToString(look.type)));
                //展示此状态可接受的词法单元类型
                DEBUG_CERR << "预期的词法单元: ";
                bool hasExpected = false;
                for (int i = 1; i <= int(TokenType::End); i++)
                    if (actionTable[s][i].act != Action::error) {
                        if (hasExpected)
                            DEBUG_CERR << ", ";
                        DEBUG_CERR << TokenTypeToString(TokenType(i));
                        hasExpected = true;
                    }
                if (!hasExpected)
                    DEBUG_CERR << "无法确定";
                DEBUG_CERR << endl;
                DEBUG_CERR << "跳过当前词法单元" << endl << endl;
                if (look.type == TokenType::End)
                    break;
                GetToken();
            }
        }
    }
```

4.3 语义分析器

4.3.1 语义分析流程基本框图



4.3.2 重点函数分析

- **抽象语法树链合并函数（merge）：**

merge 函数用于合并两个索引列表，通过预分配内存来避免多次扩容，高效地将两个列表的元素依次添加到结果中，merge 函数在处理需要合并的跳转列表时起到关键作用。

本函数的实现如下：

```
   std::vector<size_t> SemanticAnalyzer::merge(std::vector<size_t> list
1, std::vector<size_t> list2)
   {
```

```
        std::vector<size_t> result;
        result.reserve(list1.size() + list2.size()); // 预分配内存，提高性能
        result.insert(result.end(), list1.begin(), list1.end());
        result.insert(result.end(), list2.begin(), list2.end());
        return result;
    }
```

- **抽象语法树回填函数（backpatch）：**

backpatch 函数则实现了回填机制，它遍历给定的列表，将每个索引对应的四元式结果字段更新为指定的目标地址，这是实现条件跳转和循环结构的核心操作。

本函数的实现如下：

```
    void SemanticAnalyzer::backpatch(std::vector<size_t> list, size_t quad)
    {
        for (const size_t& i : list)
            qList[i - START_STMT_ADDR - 1].result = to_string(quad);
    }
```

- **抽象语法树节点生成函数（mkleaf）：**

mkleaf 函数组采用重载设计，根据不同的参数类型创建相应的语法树节点：处理标识符时创建 Id 节点，处理整数常量时创建 I32num 节点，处理操作符时创建 Op 节点，以及创建通用的 ASTNode 节点。所有节点均使用 std::unique_ptr 管理生命周期，通过 std::move 转移所有权到 nodeStack 中，确保内存安全并避免不必要的拷贝。这些函数共同支持了自底向上的语法树构建过程，为后续的代码生成和语义检查奠定了基础。

各个重载函数的实现如下：

```
    void SemanticAnalyzer::mkleaf(size_t line, size_t column, std::string id)
    {
```

```
        unique_ptr<Id> idnode = make_unique<Id>(line, column, id);
        nodeStack.push(move(idnode));
    }
    void SemanticAnalyzer::mkleaf(size_t line, size_t column, int val)
    {
        unique_ptr<I32num> i32node = make_unique<I32num>(line, column, va
l);
        nodeStack.push(move(i32node));
    }
    void SemanticAnalyzer::mkleaf(size_t line, size_t column, TokenType o
p)
    {
        unique_ptr<Op> opnode = make_unique<Op>(line, column, op);
        nodeStack.push(move(opnode));
    }
    void SemanticAnalyzer::mkleaf(size_t line, size_t column)
    {
        unique_ptr<ASTNode> astnode = make_unique<ASTNode>(line, column);
        nodeStack.push(move(astnode));
    }
```

- **语义分析过程核心函数（analyze）：**

本函数接收一个 Production 类型的常量引用 prod 作为参数，表示当前的产生式。函数内部通过一系列的 if-else 语句对不同的产生式进行匹配和处理。对于每个匹配的产生式，会执行相应的语义动作，包括节点的创建、符号表的操作、语义错误的检查和四元式的生成等。

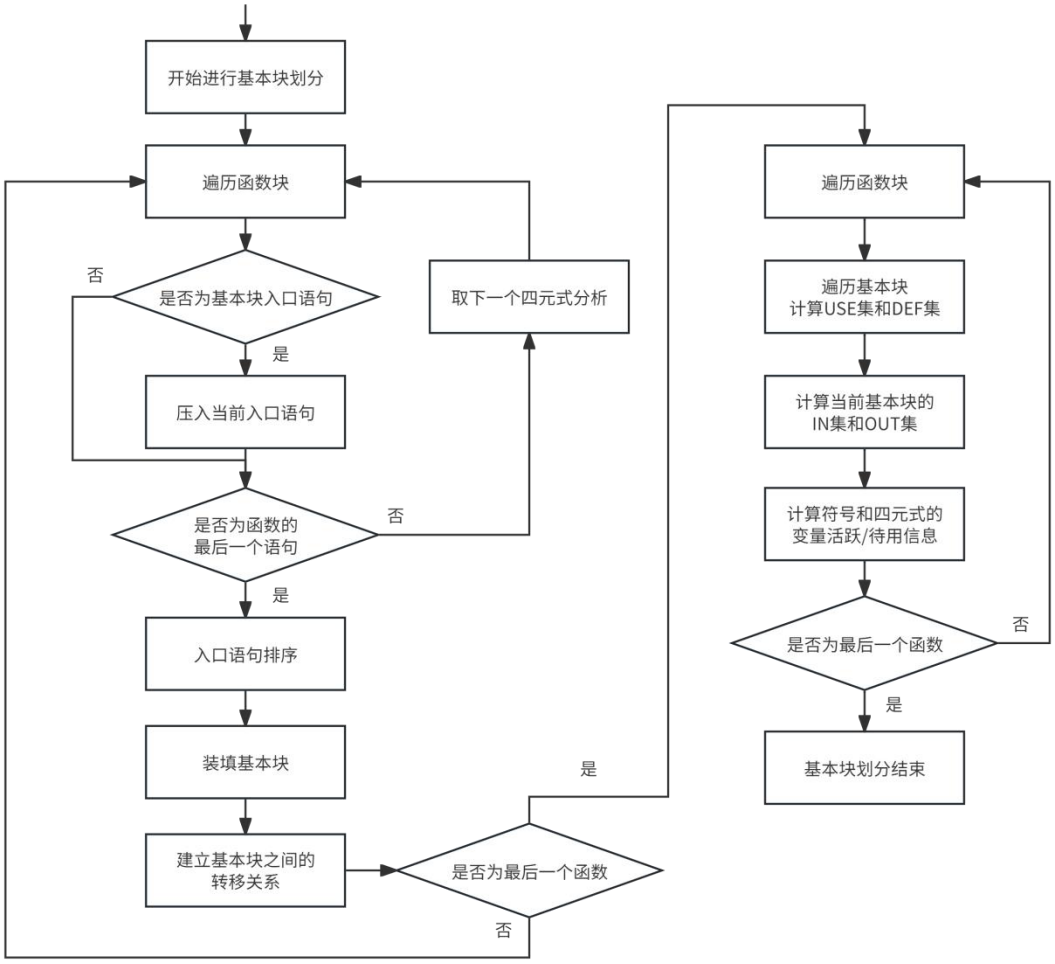
例如，在处理变量声明产生式时，会创建相应的节点并将其压入节点栈中；在处理函数声明产生式时，会检查 break 和 continue 语句的使用是否合法，以及变量类型是否能够推导出来，并进行相应的错误处理和四元式生成。这个函数是语义分析模块的核心，它确保了编译器能够对输入的语法结构进行准确的语义分析，生成正确的中间代码。

```
    void SemanticAnalyzer::analyze(const Production& prod)
```

```
    {
        //0.1 变量声明内部----------<变量声明内部>->mut <ID>
        if (prod.left.name == "Identifier_inside" && prod.right.size() ==
 2 && prod.right[0].name == TokenTypeToString(TokenType::MUT) && prod.rig
ht[1].name == TokenTypeToString(TokenType::Identifier)) {
            unique_ptr<Id> idnode = unique_ptr<Id>(static_cast<Id*>(nodeS
tack.top().release()));
            nodeStack.pop();
            unique_ptr<ASTNode> mutnode = move(nodeStack.top());
            nodeStack.pop();
            unique_ptr<BridgeNode> nonterminal = make_unique<BridgeNode>
(mutnode->line, mutnode->column, SymbolKind::Variable, idnode->name);
            nodeStack.push(move(nonterminal));
        // 下面逐个分支进行文法的解析，此处省略
        // ......
    }
```

4.4 基本块划分器

4.4.1 基本块划分流程基本框图



4.4.2 重点函数分析

- **基本块划分函数（divideBlocks）：**

本函数将完整的四元式集合划分为基本块需要经过三大步骤：（1）确定入口语句；（2）压入对应四元式以创建基本块；（3）建立基本块之间的转移关系。在函数实现上，始终通过迭代函数表（procedureList）的方式逐个函数进行处理，因为不同函数之间的语句不会划分在同一个基本块内。

本函数的算法实现如下：

（1）确定入口语句：

```
    /* 第一步：确定入口语句位置并记录 */
    func_start = proc.addr - this->start_address
    if i == procTable.size() - 1:
        // 若为最后一个函数，结束位置为四元式列表末尾
        func_end = this->qList.size()
    else:
        // 否则结束位置为下一个函数起始位置
        func_end = procTable[i + 1].addr - this->start_address
    // 将函数起始语句作为入口语句
    block_enter.push_back(func_start)

    for j from func_start to func_end - 1:
        q = this->qList[j]
        if q.op starts with 'j':  // 处理跳转语句
            if q.op == "j":  // 无条件跳转
                block_enter.push_back(std::stoi(q.result) -
this->start_address)
            else:  // 条件跳转
                if j < func_end - 1:
                    // 下一条语句作为入口语句
                    block_enter.push_back(j + 1)
                    // 跳转目标语句作为入口语句
                    block_enter.push_back(std::stoi(q.result) -
this->start_address)
        else if q.op == "call" and j < func_end - 1:
            // 函数调用后的下一条语句作为入口语句
            block_enter.push_back(j + 1)

    // 对入口语句位置进行排序并去重
    block_enter.sort()
    block_enter.erase_duplicates()
```

（2）创建基本块：

```
        /*  第二步：创建基本块 */
        this->func_blocks[proc.ID] = std::vector<Block*>();

        for (size_t j = 0; j < block_enter.size(); ++j) {
            Block* blk = new Block();

            int blk_start = block_enter[j];
```

```
            int blk_end = (j < block_enter.size() - 1) ? block_enter[j + 1] :
func_end;

            // 设置块名和起始地址
            blk->setName(j == 0 ? proc.ID : this->getBlockName());
            blk->setStartAddr(blk_start);

            // 压入每个基本块的四元式（介于两个基本块起始地址之间 / 到一个转移
语句停止)
            for (int k = blk_start; k < blk_end; ++k) {
                blk->addCode(this->qList[k]);
                const std::string op = this->qList[k].op;
                if (op[0] == 'j' || op == "call" || op == "ret") {
                    break;
                }
            }

            func_blocks[proc.ID].push_back(blk);
            blocks.push_back(blk);
        }
```

（3）建立基本块之间的转移关系：

```
    /* 第三步：建立基本块之间的转移关系*/
    auto& blks = this->func_blocks[proc.ID];
    for (size_t j = 0; j < blks.size(); ++j) {
        Block* blk = blks[j];
        Block* next_blk = (j < blks.size() - 1) ? blks[j + 1] : nullptr;
        if (blk->getCodes().empty()) continue;  // 空块跳过
        const auto& lastq = blk->getLastCode();
        if (lastq.op[0] == 'j') {  // 跳转指令
            Block* dest_blk = FindBlock(std::stoi(lastq.result) -
this->start_address);
            if (lastq.op == "j") {
                blk->setNext(dest_blk, nullptr);  // 无条件跳转
            } else {
                blk->setNext(next_blk, dest_blk);  // 条件跳转
            }
            lastq.result = dest_blk->getName();
        } else if (lastq.op == "call") {
```

```
            Block* dest_blk = FindBlock(std::stoi(lastq.arg1) -
this->start_address);
            blk->setNext(dest_blk, nullptr);  // 调用
            lastq.arg1 = dest_blk->getName();
        } else if (lastq.op == "ret") {
            blk->setNext(nullptr, nullptr);  // 返回
        } else {
            blk->setNext(next_blk, nullptr);  // 默认到下一个块
        }
    }
```

- **变量活跃/待用信息计算函数（computeBlocks）：**

在完成基本块的划分后，调用本函数对每个基本块内的符号（变量）计算其活跃/待用信息。该过程的具体实现依赖于 Block 类的 computeUseDefSet 和 computeInOutSet 两个成员函数。在这些集合计算完毕后，才可以进而计算变量的活跃/待用信息（SymbolInfo）。

（1）计算 USE 和 DEF 集合：

```
    void Block::computeUseDefSets()
    {
        this->use_set.clear();
        this->def_set.clear();
        for (const auto& q : this->getCodes()) {
            if (q.op == "j") {
                continue;   // 无条件跳转语句不参与分析
            }
            // 分析剩余四元式
            if (q.op != "call") {
                // 源操作数 arg1、arg2
                if (this->is_variable(q.arg1) && this->def_set.find(q.arg1)
== this->def_set.end()) {
                    this->use_set.insert(q.arg1);
                }
                if (this->is_variable(q.arg2) && this->def_set.find(q.arg2)
== this->def_set.end()) {
                    this->use_set.insert(q.arg2);
                }
            }
```

```
            if (q.op[0] != 'j') {
                // 目标操作数
                if (this->is_variable(q.result) &&
this->use_set.find(q.result) == this->use_set.end()) {
                    this->def_set.insert(q.result);
                }
            }
        }
        // 计算完毕，使用 USE 集初始化 IN 集
        this->in_set = this->use_set;
    }
```

（2）计算 IN 和 OUT 集合：

```
    bool Block::computeInOutSet()
    {
        // 保存 IN、OUT 集原大小以检测变化
        size_t prev_in_size = this->in_set.size();
        size_t prev_out_size = this->out_set.size();

        Block* next1 = this->next1;
        Block* next2 = this->next2;
        if (next1) {
            for (const auto& var : next1->in_set) {
                this->out_set.insert(var);
                if (this->def_set.find(var) == this->def_set.end()) {
                    this->in_set.insert(var);
                }
            }
        }
        if (next2) {
            for (const auto& var : next2->in_set) {
                this->out_set.insert(var);
                if (this->def_set.find(var) == this->def_set.end()) {
                    this->in_set.insert(var);
                }
            }
        }
        // 检查更新
        if (this->in_set.size() != prev_in_size || this->out_set.size() !=
prev_out_size)
```

```
            return true;
        else
            return false;
    }
```

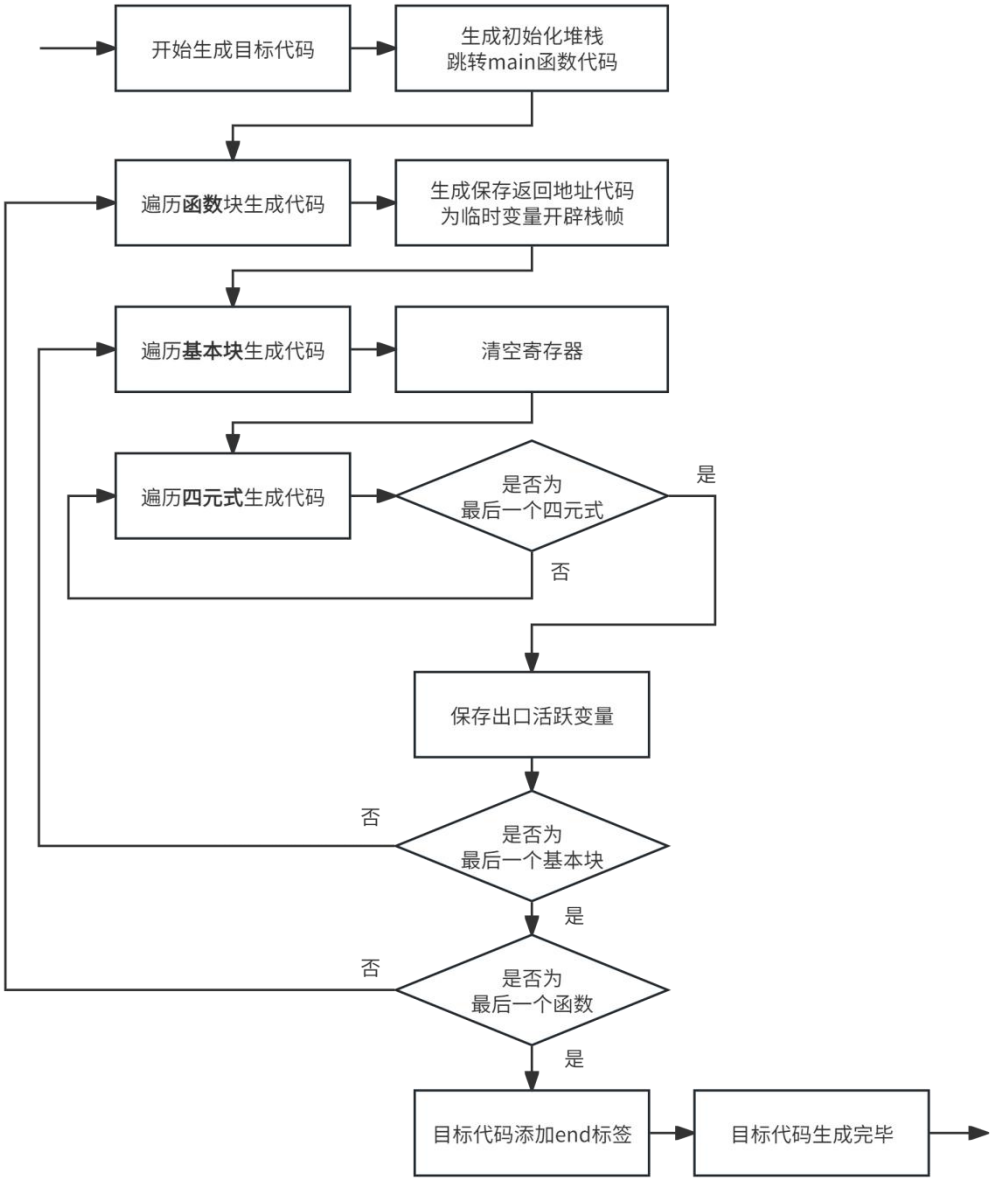
（3）计算变量活跃/待用信息：

```
    void Block::computeSymbolInfo(std::unordered_map<std::string,
SymbolInfo>& initSymbolInfoTable)
    {
        this->symbolInfoTable = initSymbolInfoTable;
        // 从后向前扫描四元式
        int idx = this->codes.size();
        for (auto it = this->codes.rbegin(); it != this->codes.rend(); ++it)
{
            --idx;
            const auto& q = *it;
            QuadrupleInfo qInfo;// 当前四元式的变量活跃信息
            if (q.op == "j" || q.op == "call") {
                this->codesInfo[idx] = qInfo;// 直接保存空活跃信息，无额外操
作
                continue;
            }
            // 处理源操作数
            if (this->is_variable(q.arg1)) {
                qInfo.arg1_info = this->symbolInfoTable[q.arg1];
                this->symbolInfoTable[q.arg1] = SymbolInfo(idx, true);
            }
            if (this->is_variable(q.arg2)) {
                qInfo.arg2_info = this->symbolInfoTable[q.arg2];
                this->symbolInfoTable[q.arg2] = SymbolInfo(idx, true);
            }
            // 处理目的操作数
            if (q.op[0] != 'j') {// 非跳转类指令
                if (this->is_variable(q.result)) {
                    qInfo.result_info = this->symbolInfoTable[q.result];
                    this->symbolInfoTable[q.result] = SymbolInfo(-1,
false);
                }
            }
            // 更新四元式对应的变量活跃信息
```

```
            this->codesInfo[idx] = qInfo;
        }
    }
```

4.5 目标代码生成器

4.5.1 目标代码生成流程基本框图



4.5.2 重点函数分析

- **寄存器分配函数（allocFreeReg）：**

编译过程中，寄存器分配原则如下：（1）优先检查是否存在空闲的寄存器，存在则可以直接分配给对应变量；（2）若没有空闲寄存器，则选择一个最远引用变量所在的寄存器让渡。具体实现上，需要遍历已经分配的所有寄存器，查找每个寄存器中距离下次使用位置最远的变量，然后将该变量占用的寄存器腾空给当前更为紧急的变量使用。

本函数的算法总结如下：

```
    if (!free_regs.empty()) {  // 若有空闲寄存器，直接分配
            return free_regs.back();
        }
        // 无空闲寄存器，选择最远使用的寄存器
        double farest_usepos = -1;
        std::string selected_reg;
        for (const auto& [reg, vars] : RValue) {
            double cur_usepos = std::numeric_limits<double>::infinity();
            for (const auto& var : vars) {
                if (AValue[var].size() > 1) continue;  // 多副本变量跳过
                cur_usepos = findNearestUse(qList, var, cur_q_idx);  // 查找最近位置
                if (cur_usepos < std::numeric_limits<double>::infinity()) break;
            }
            if (cur_usepos > farest_usepos) {
                farest_usepos = cur_usepos;
                selected_reg = reg;
            }
        }
        // 处理选中寄存器变量，需保存时存入内存
        for (const auto& var : RValue[selected_reg]) {
            AValue[var].erase(selected_reg);
            if (AValue[var].empty() && needStoreVar(var, qList, cur_q_idx, out_set)) {
                storeVarToMem(var, selected_reg, codes);  // 保存变量
            }
        }
        RValue[selected_reg].clear();  // 释放寄存器
    return selected_reg;
```

- **为源操作数分配寄存器函数（getArgReg）：**

为源操作数分配寄存器的原则如下：（1）优先检查 AValue 中是否已经存储了当前操作数的寄存器，若有则直接返回该寄存器；（2）若没有，则调用寄存器分配函数为当前操作数申领新的寄存器。若操作数为立即数，直接通过“li”命令装载；若操作数为变量，使用“lw”命令从存储器（memory）加载。

本函数的代码实现如下：

```
    std::string RegManager::getArgReg(const std::string& arg, const
std::vector<Quadruple>& qList, int cur_q_idx, const std::set<std::string>&
out_set, std::vector<std::string>& codes)
    {
        std::string reg;
        // 先检查 AValue 中是否已经有副本
        for (const auto& pos : this->AValue[arg]) {
            if (pos != "Mem") {
                return pos;
            }
        }
        // 没有其他副本可以直接使用，则需要请求分配一个寄存器
        reg = this->allocFreeReg(qList, cur_q_idx, out_set, codes);
        // 生成存取命令
        if (this->is_variable(arg)) { // 局部变量或中间变量
            if (this->func_vars.count(arg) || arg[0] == 'T') {
                std::string code = "lw " + reg + ", " +
std::to_string(this->memory[arg]) + "($sp)";
                codes.push_back(code);
            }
            else if (this->data_vars.count(arg)) {
                std::string code = "lw " + reg + ", " + arg;
                codes.push_back(code);
            }
        }
        else { // 立即数
            std::string code = "li " + reg + ", " + arg;
            codes.push_back(code);
        }
        return reg;
    }
```

- **为目的操作数分配寄存器函数（getResultReg）：**

为目的操作数分配寄存器的原则如下：（1）优先检查是否有可复用的源操作数寄存器，若当前的源操作数在本条四元式之后不再活跃，则可以直接让渡目前占用的寄存器给目的操作数；（2）若无可复用的寄存器，则调用寄存器分配函数为目的操作数申领新的寄存器。

本函数的代码实现如下：

```
    std::string RegManager::getResultReg(const std::string& result, const
std::vector<Quadruple>& qList, const std::vector<QuadrupleInfo>& qInfoList,
int cur_q_idx, const std::set<std::string>& out_set,
std::vector<std::string>& codes)
    {
        const auto& q = qList[cur_q_idx];
        const auto& qInfo = qInfoList[cur_q_idx];
        std::string arg1 = q.arg1;
        // 判断能否复用 arg1 的寄存器（确保 arg1 是变量，且不抢占全局变量的寄存器）
        if (this->is_variable(arg1)) {
            if (this->data_vars.find(arg1) != this->data_vars.end() &&
                this->func_vars.find(arg1) == this->func_vars.end()) {
                for (const auto& pos : this->AValue[arg1]) {
                    if (pos != "Mem" && this->RValue[pos].size() == 1) {
                        if (qInfo.arg1_info.getIsActive() == false) {
                            // 更新 RValue 和 AValue 关系
                            this->RValue[pos].erase(arg1);
                            this->RValue[pos].insert(result);

                            this->AValue[arg1].erase(pos);
                            this->AValue[result].insert(pos);
                            return pos;
                        }
                    }
                }
            }
        }
        // 重新分配寄存器
        std::string reg = this->allocFreeReg(qList, cur_q_idx, out_set,
codes);
        this->RValue[reg].insert(result);
        this->AValue[result].insert(reg);
        return reg;
```

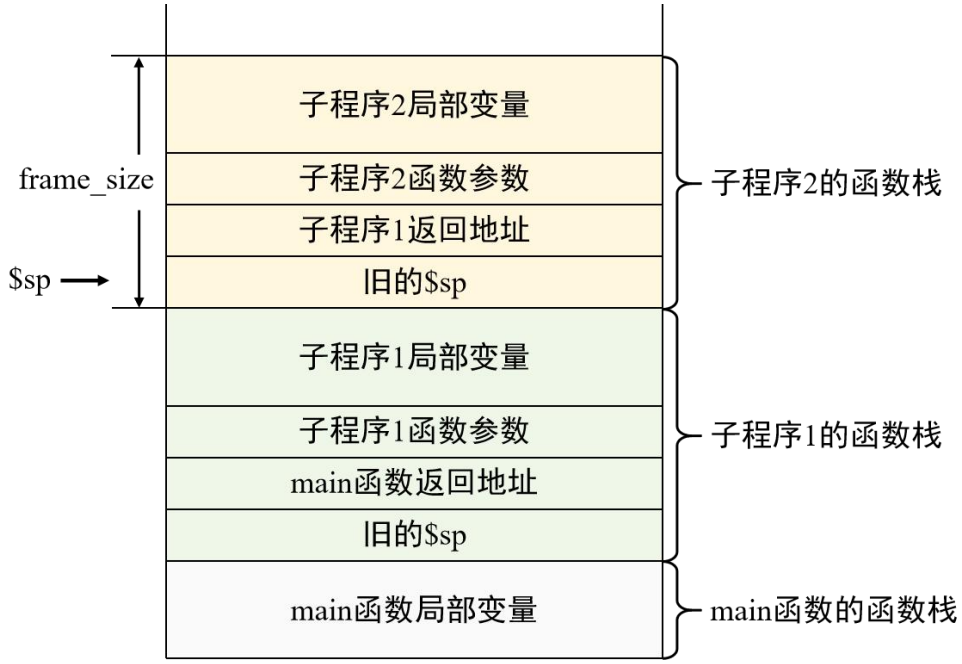
```
}
```

4.5.3 针对目标语言的设计

- **函数栈帧设计：**

本次课程设计的堆栈设计如下：堆栈段首地址为 0x10040000，函数栈按照调用顺序在堆栈段之上。$sp 寄存器中存放当前函数栈的底部地址，目标代码生成器中的寄存器管理器的成员变量 frame_size 存放当前函数栈的栈帧大小。

假设在 main 函数中调用了子程序 1，又在子程序 1 中嵌套调用了子程序 2。则依照本次的堆栈设计规则，编译器的函数栈帧应如下图所示：



发生函数调用时，当扫描到操作符为“param”的四元式，将变量压入变量列表 param_list。直到扫描到操作符为“call”的四元式时，就将前述变量压入当前函数栈的参数区。之后，直接将$sp 压栈即可完成函数调用前的准备工作。

进入被调用函数后，首先将$ra 寄存器中保存的函数返回地址转存到对应位置，从而防止在函数嵌套调用时返回地址丢失。然后，直接通过“$sp+相对地址”的方式来访问当前函数的参数和局部变量。

函数返回时，按照 Mips32 的规则将返回值放入$v0 寄存器中，再从对应地址

取出返回地址装填$ra 寄存器，并通过产生目标代码“jr”指令返回。然后，访问$sp 寄存器加载旧的 sp 值，即可恢复到函数调用前的栈帧状态。

- **数据管理：**

依照本次课程设计要求，必须含有 main 函数且不允许存在全局变量定义（属于语法错误），因此全局变量在编译过程中始终为空。但是为保证数据结构和目标代码结构的完整性，仍然将 data 段保留并保存在 0x10010000 处。

5. 执行界面与运行结果

5.1 执行方法

本项目脚本需要在代码文件夹目录中执行。

用户需要确保本地包含 Node.js 运行环境及 npm（或 pnpm、yarn）工具。

推荐使用 Visual Studio 或 VS Code 打开项目，或直接使用命令行操作。

5.1.1 安装依赖

进入项目根目录下的 code 文件夹，使用“npm install”命令安装相关依赖。

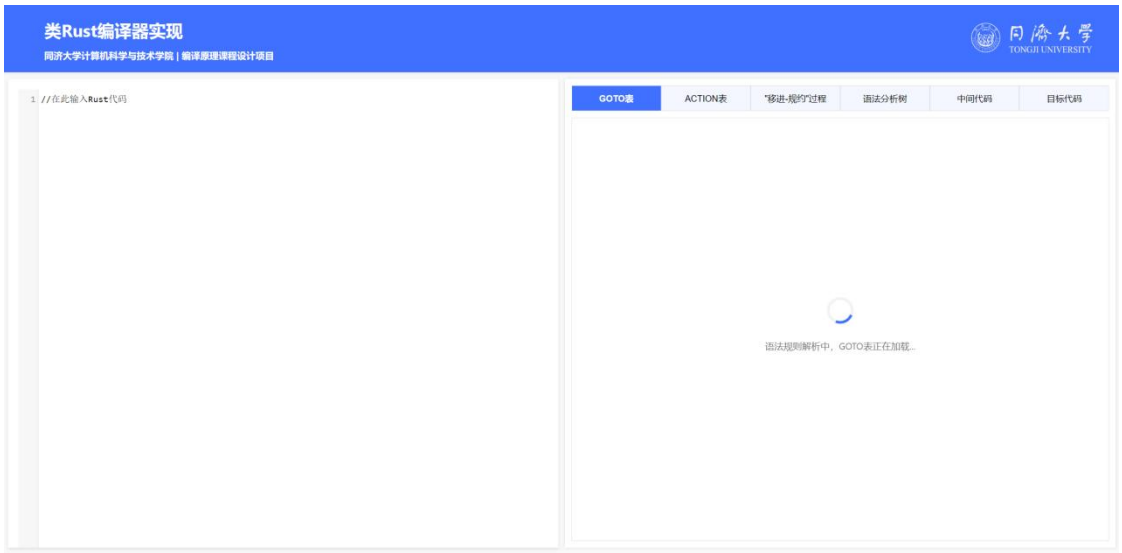
```
D:\Desktop\课设_目标代码生成>cd code

D:\Desktop\课设_目标代码生成\code>npm install

up to date in 4s

14 packages are looking for funding
  run `npm fund` for details
```

5.1.2 运行界面

在 code 文件夹下启动服务器，使用“node server.js”命令。若看到终端显示“Server running on port 3000”，则表示服务器启动成功。

```
D:\Desktop\课设_目标代码生成\code>node server.js
Server running on port 3000
```

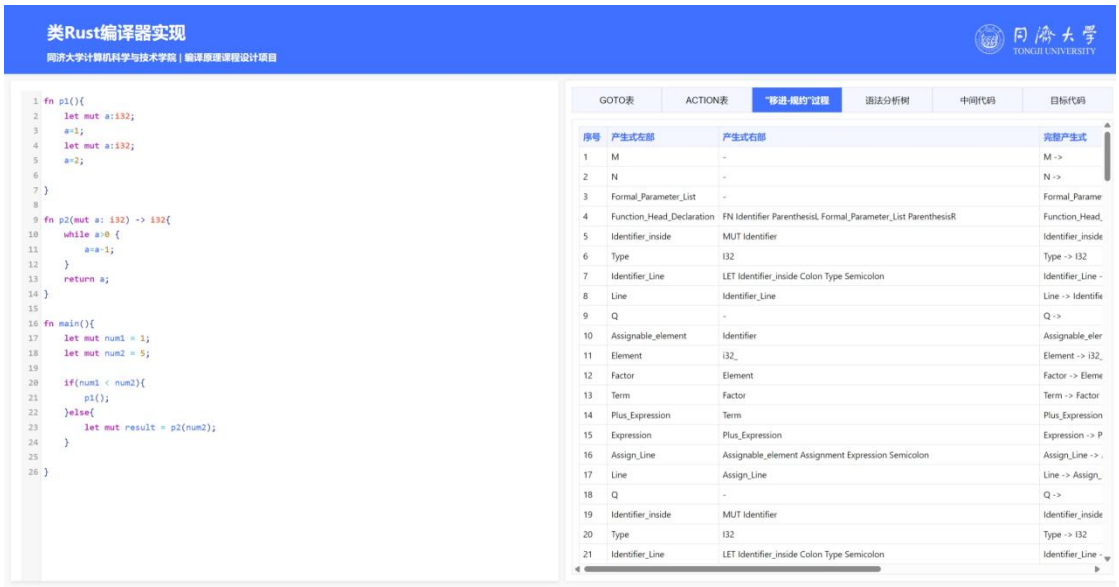
在 code 文件夹下双击 index.html，即可启动 web 应用进入初始化界面。



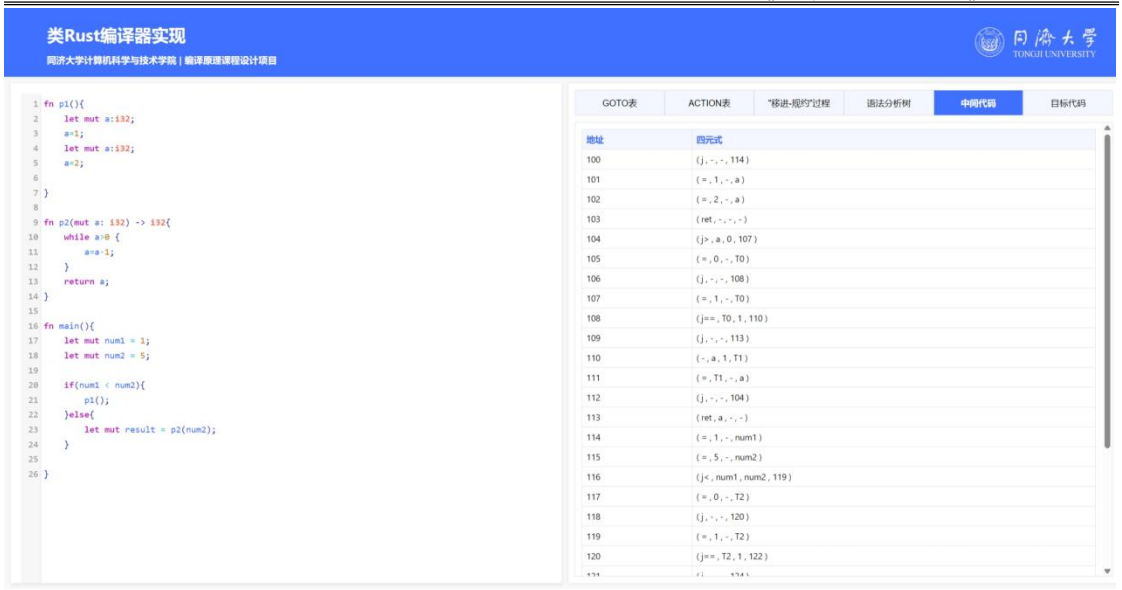
5.2 正确用例

在 web 应用左侧的编辑器中输入如下 rust 源代码，选择“目标代码”即可进行完整的编译过程。选择其他选项可以看到编译过程中其他数据的可视化展示。

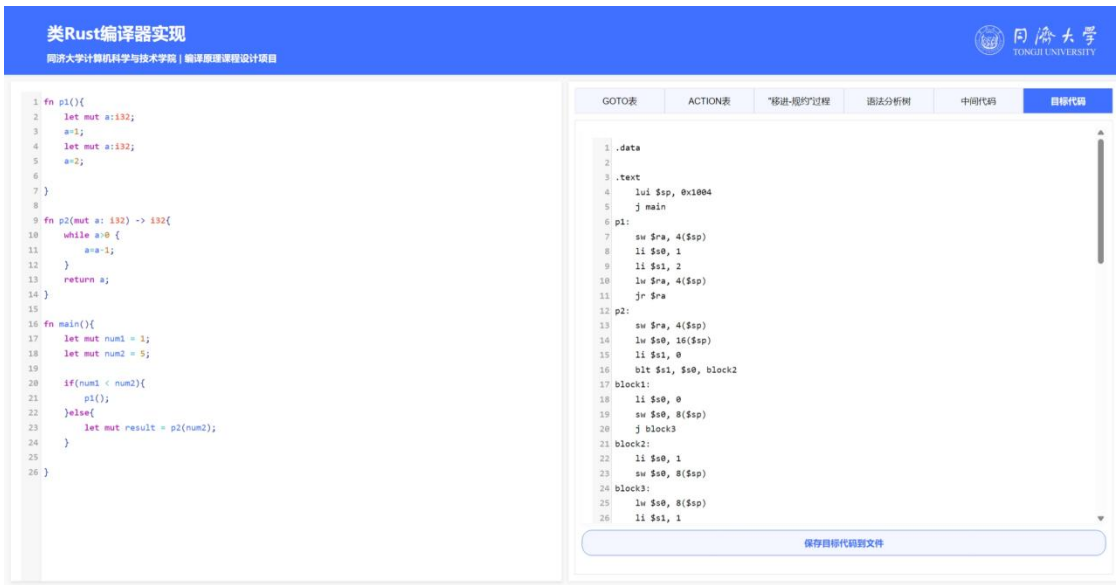
“移进-规约”过程的可视化结果如下：



中间代码的可视化结果如下：



目标代码的可视化结果如下：



输入的源程序：

```
fn p1(){
    let mut a:i32;
    a=1;
    let mut a:i32;
    a=2;
}
fn p2(mut a: i32) -> i32{
    while a>0 {
        a=a-1;
    }
    return a;
```

```
}
fn main(){
    let mut num1 = 1;
    let mut num2 = 5;

    if(num1 < num2){
        p1();
    }else{
        let mut result = p2(num2);
    }
}
```

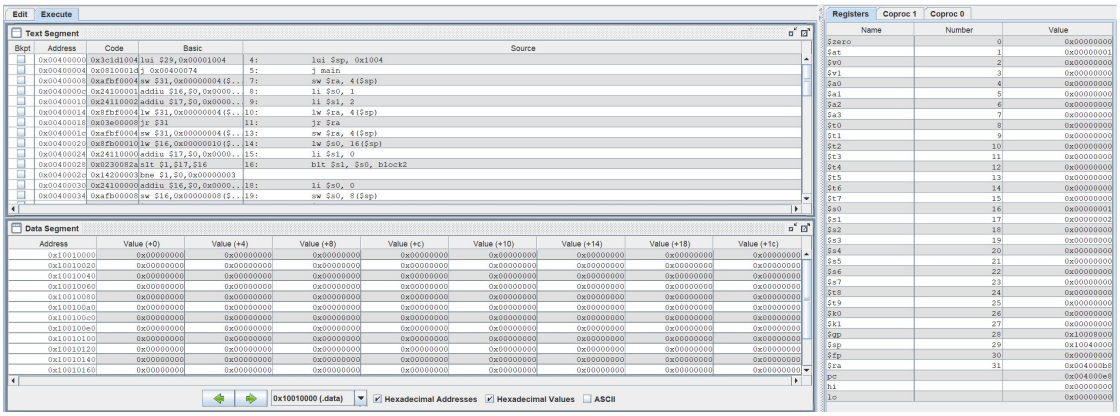
对应生成的目标代码：

```
.data
.text
    lui $sp, 0x1004
    j main
p1:
    sw $ra, 4($sp)
    li $s0, 1
    li $s1, 2
    lw $ra, 4($sp)
    jr $ra
p2:
    sw $ra, 4($sp)
    lw $s0, 16($sp)
    li $s1, 0
    blt $s1, $s0, block2
block1:
    li $s0, 0
    sw $s0, 8($sp)
    j block3
block2:
    li $s0, 1
    sw $s0, 8($sp)
block3:
    lw $s0, 8($sp)
    li $s1, 1
    beq $s0, $s1, block5
block4:
    j block6
```

```
block5:
    lw $s0, 16($sp)
    li $s1, 1
    sub $s2, $s0, $s1
    sw $s2, 16($sp)
    j p2
block6:
    lw $v0, 16($sp)
    lw $ra, 4($sp)
    jr $ra
main:
    li $s0, 1
    li $s1, 5
    blt $s0, $s1, block8
    sw $s1, 12($sp)
block7:
    li $s0, 0
    sw $s0, 0($sp)
    j block9
block8:
    li $s0, 1
    sw $s0, 0($sp)
block9:
    lw $s0, 0($sp)
    li $s1, 1
    beq $s0, $s1, block11
block10:
    j block13
block11:
    sw $sp, 20($sp)
    addi $sp, $sp, 20
    jal p1
    lw $sp, 0($sp)
block12:
    j block15
block13:
    lw $s0, 12($sp)
    sw $s0, 28($sp)
    sw $sp, 20($sp)
    addi $sp, $sp, 20
    jal p2
    lw $sp, 0($sp)
    add $s1, $v0, $zero
```

```
    sw $s1, 4($sp)
block14:
    lw $s0, 4($sp)
block15:
    j end
end:
```

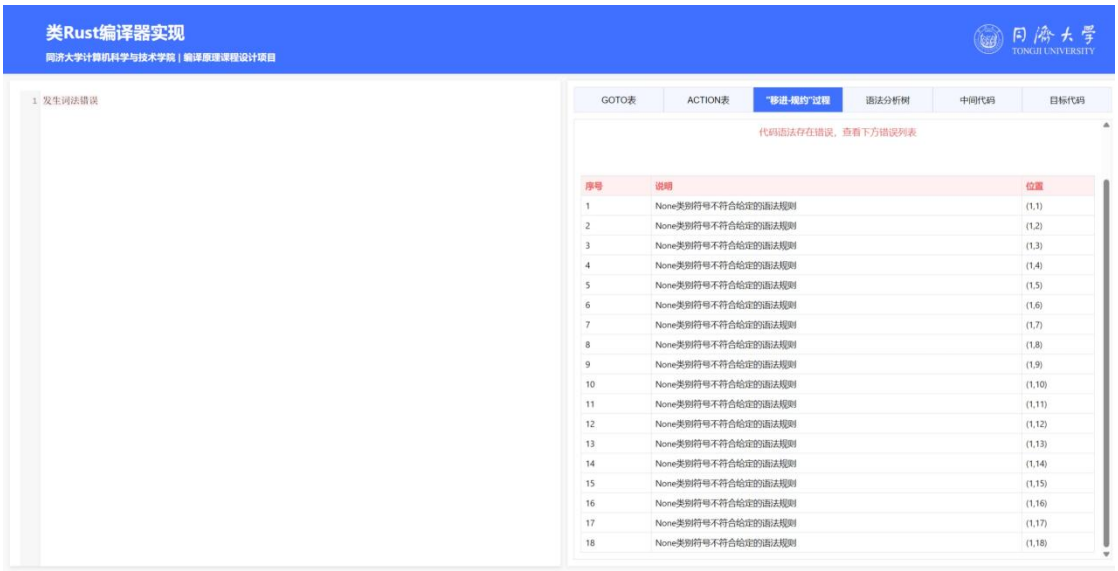
在 mars 中执行目标代码文件 code.asm，可以看到执行结果正确：



5.3 错误用例

5.3.1 词法错误

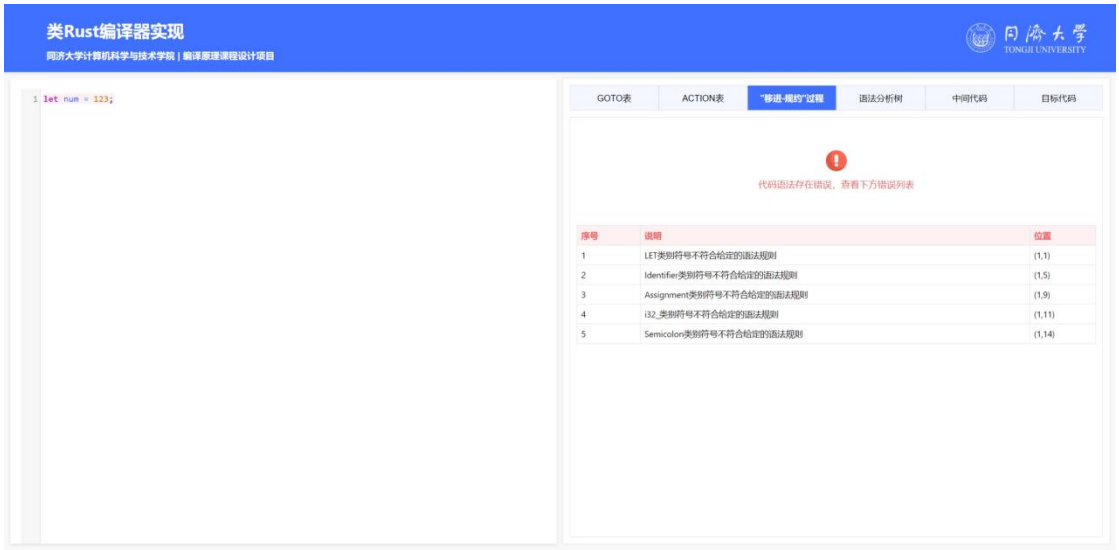
当词法分析错误时，后端将不解析该 token 的类型、前端不会高亮。



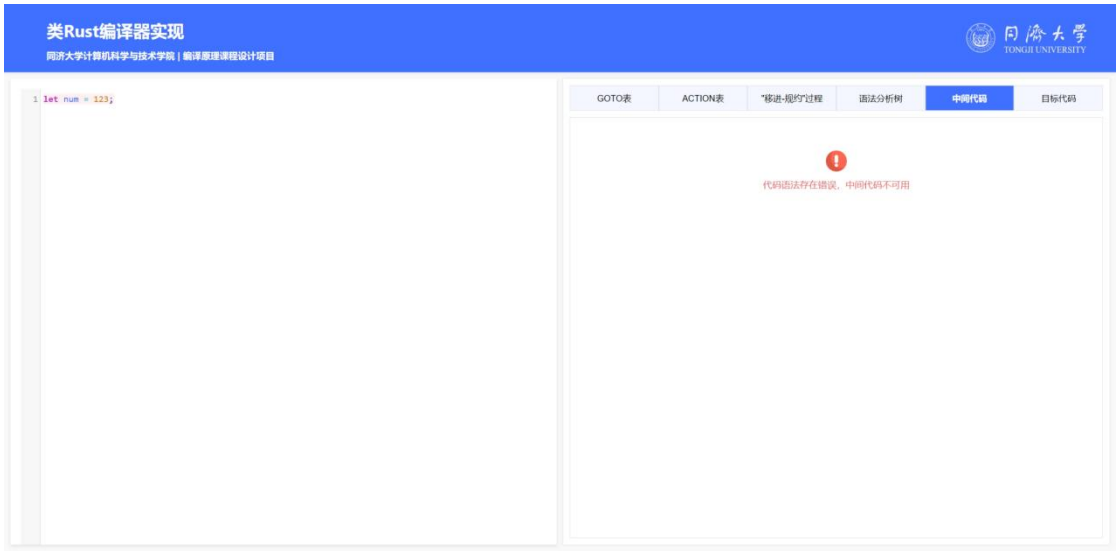
5.3.2 语法错误

当语法分析错误时，在“移进-规约”过程子界面上将显示出语法错误列表和

对应的出错位置，代码编辑区将红色标亮出错的代码。



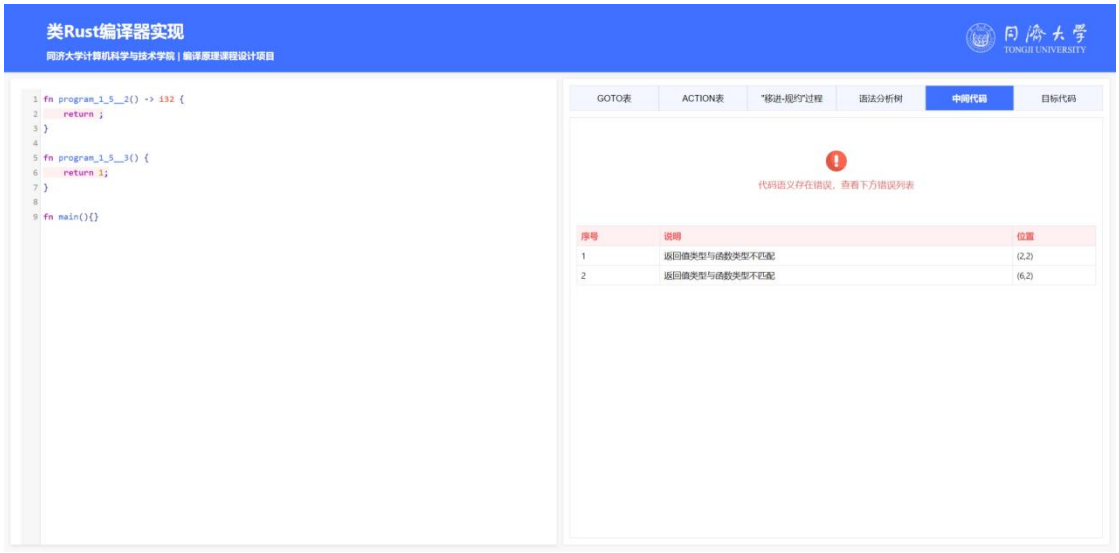
同时，当语法分析法发生错误时语义分析将停止，对应的“中间代码”子界面上将显示出“语法分析错误”的提示信息。



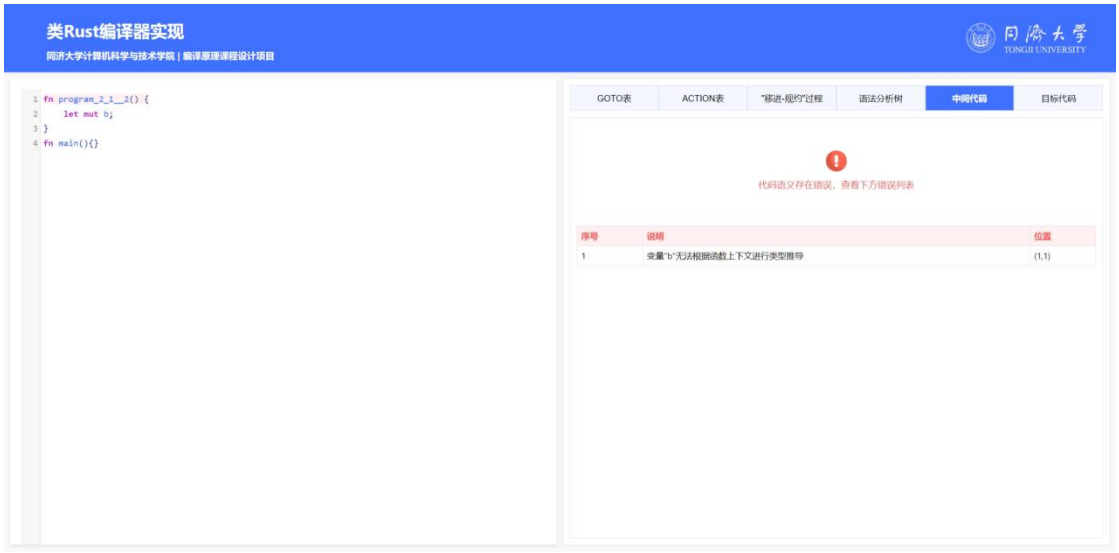
5.3.3 语义错误

依据题目要求的文法规则，对各种可能出现的语义错误逐一进行测试，查看中间代码生成器的语义分析功能是可以否正确运行。

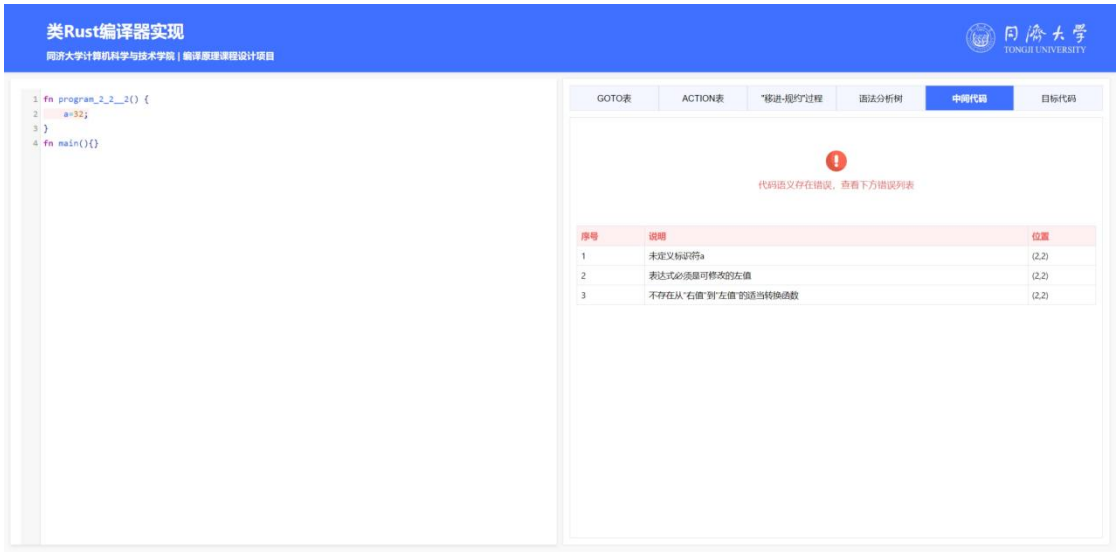
（1）**返回语句的类型与函数声明的返回类型不一致**。例如函数声明返回为空但函数体内返回了值，或函数声明了返回值类型但函数体内返回为空，语义分析器都将报错“返回类型与函数类型不匹配”并定位代码位置：



（2）**变量声明语句中无法判断变量类型**。例如在语句中声明变量为 mut 但未给出右值，语义分析将报错显示该变量“无法根据上下文进行类型推导”：

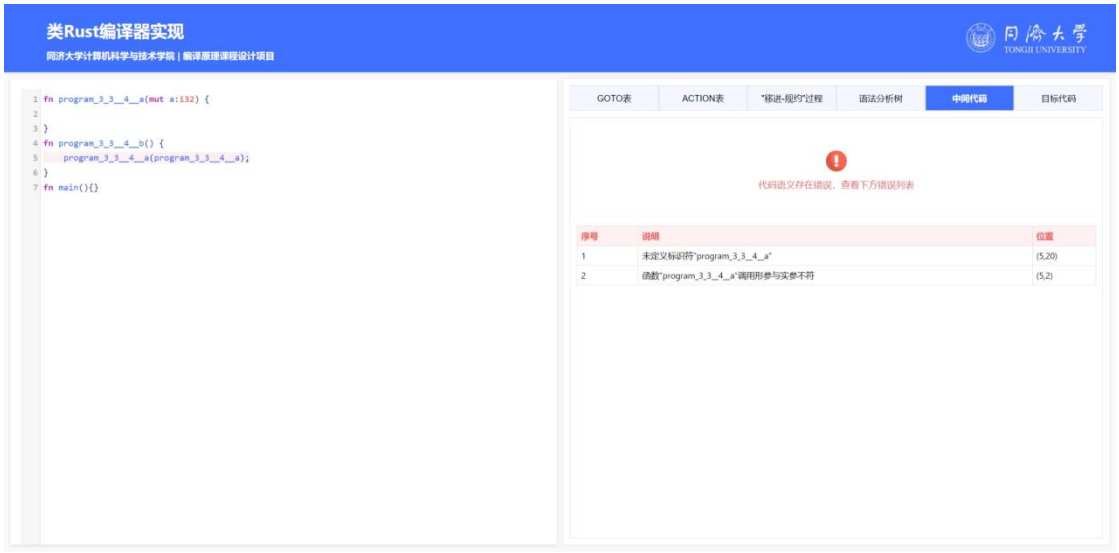


（3）**变量声明语句缺失，赋值语句中有未声明的变量**。例如，直接对一个未声明过的变量进行赋值，将显示该标识符未定义；由于无法确定该标识符的类型，将报错“表达式必须是可修改的左值”以及“不存在从右值到左值的适当转换函数”的信息：

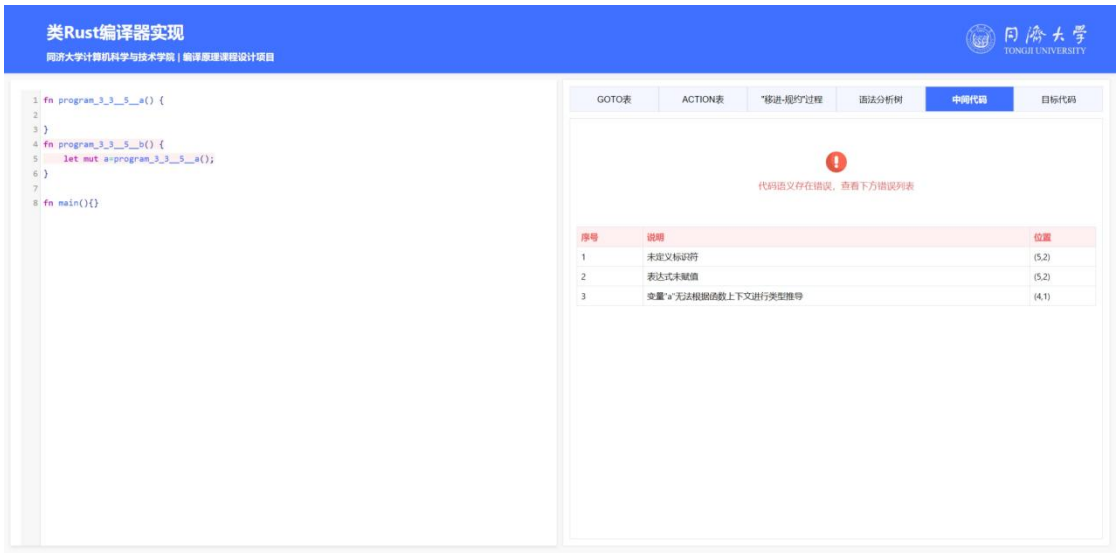


（4）**变量声明赋值语句中右值未声明或未赋值**。例如，在下面的两个函数体中，分别使用了未声明和未赋值的变量 a 对左值进行赋值，可以看到语义分析器均提示了对应的报错信息，即“未定义的标识符”对应右值未声明的情况，“表达式未赋值”对应右值未赋值的情况：

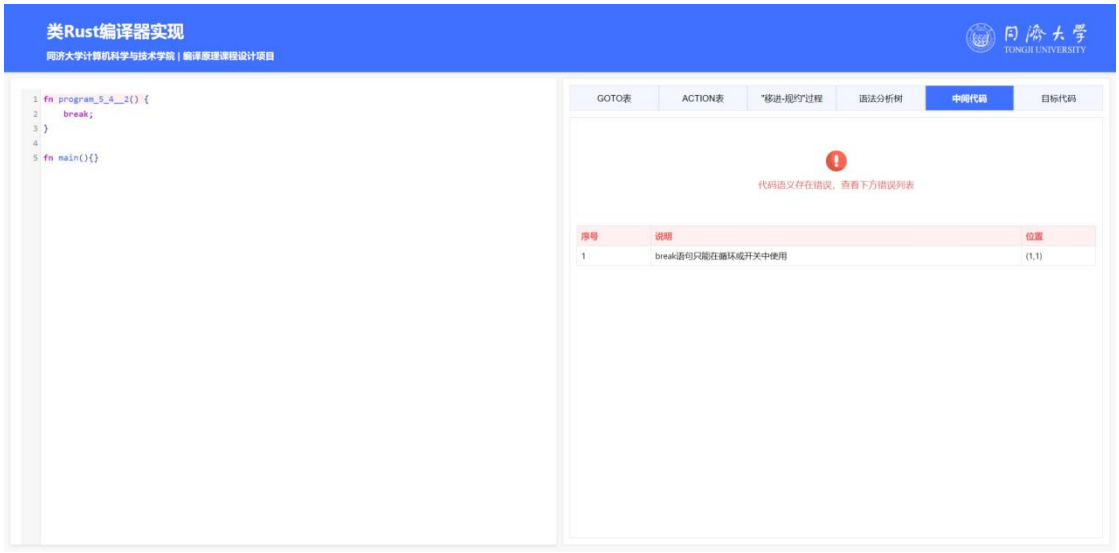
（5）**函数调用的实参与形参不一致**。例如，当函数的实参与形参的数量不一致或类型不一致时，语义分析器将报错提示“某某函数的实参与形参不一致”并给出代码错误位置定位：



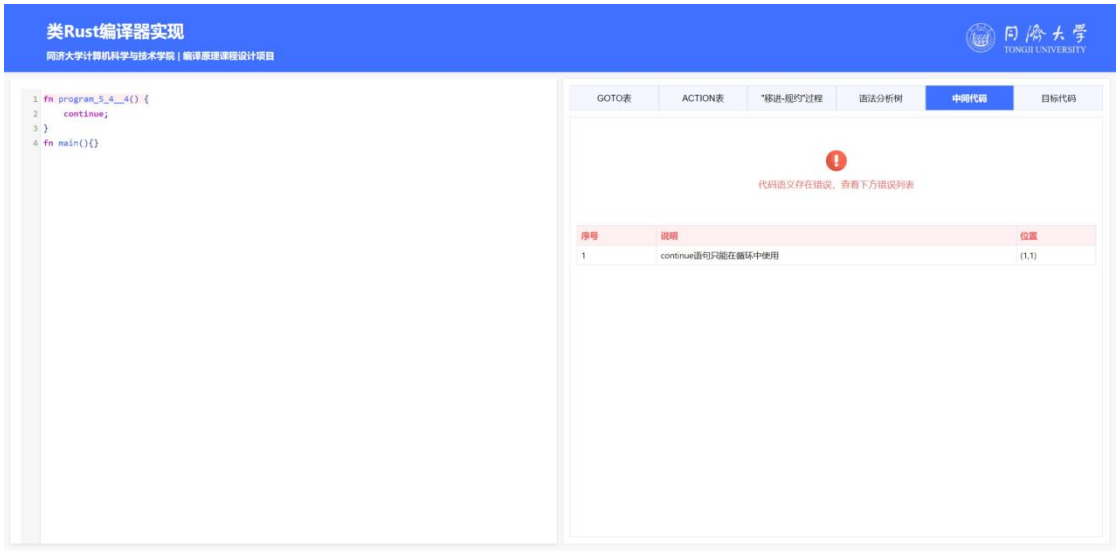
（6）**无返回值的函数不能作为右值**。例如，在变量赋值声明语句中，使用无返回值的函数作为右值，将报错提示“表达式未赋值”，且变量声明语句中的变量由于语句右值缺失，将提示“未定义标识符”和“该变量的值无法根据函数上下文进行类型推导”：



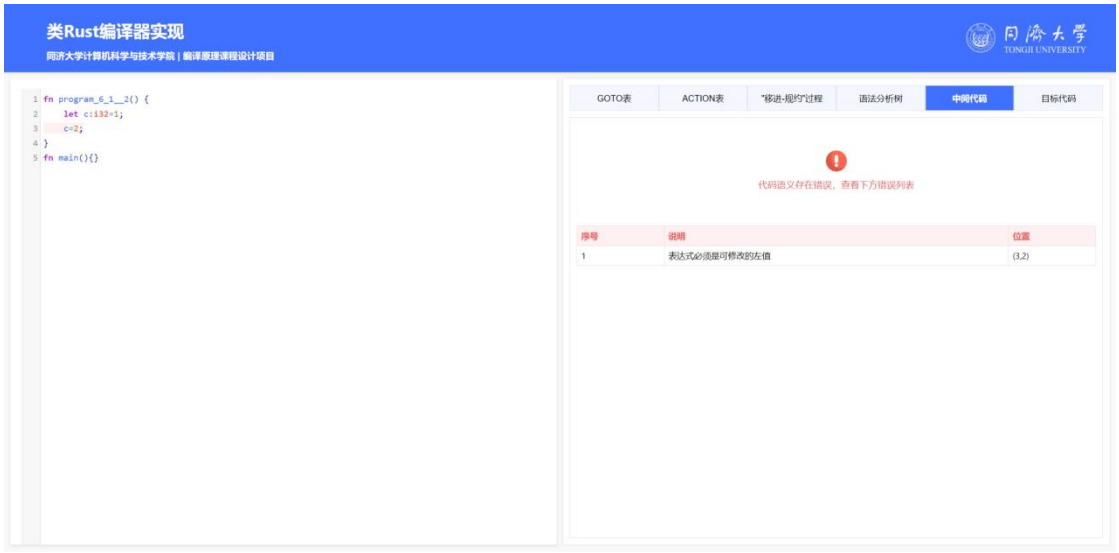
（7）**break 结构必须出现在循环体内**。在拓展文法的 loop 和 while 循环结构中可以使用 break 语句用于跳出循环，若 break 语句出现在循环体以外的地方，将报错提示“break 语句只能在循环或开关中使用”：



（8）**continue 语句只能在循环体内使用**。在拓展文法的 loop 和 while 循环结构中可以使用 continue 语句用于继续循环，若 continue 语句出现在循环体以外的地方，将报错提示“continue 语句只能在循环内使用”：



（9）**不可变变量不可二次赋值**。例如，在声明不可变变量后，若对其进行二次赋值，将报错提示“表达式必须是可修改的左值”：



（10）**不允许对非引用类型进行解引用**。例如，在对未事先声明为引用类型的变量进行解引用时，将报错提示“操作数不是引用类型，无法进行解引用操作”：

（11）**可变引用不能和其他类型的引用共存**。例如，当在已经定义可变或非可变引用的前提下创建同一变量的新的可变引用、或在已有可变引用的前提下创建同一变量的非可变引用时，将报错提示“可变引用不能和（对应变量名）的其他引用共存”：

（12）**仅支持从可变变量创建可变引用**。例如，对不可变变量创建可变引用时，将报错提示“不允许从不可变变量（对应变量名）创建可变引用”：

6. 调试分析

本次课程设计中，词法分析、语法分析和语义分析主要沿用了上学期的大作业 1 和大作业 2 中的内容，只进行了少许修改。因此本部分调试相关的内容，将围绕基本块划分和目标代码生成进行。

6.1 函数参数列表

在上学期完成语法分析和语义分析时，我们的函数表表项只按照声明顺序保存了函数参数的类型，但并没有保存参数对应的名称；而符号表又依赖于扫描过程层层嵌套，如果想在四元式的基础上生成目标代码，就必须再经过二次扫描获取当前对应的函数和与之对应的符号表。这显然与题目要求的一遍扫描不符。

因此，我修改了函数表表项创建过程的相关函数 enterproc，由于在创建函数表时仍然属于一遍扫描的范畴，且当时的符号表类 tblptr 栈的栈顶就是当前层次的符号表指针。因此，只需要取其栈顶 tblptr.top()，并在新的函数表表项中关联这个函数表，既可以找到函数参数的名称。

```
enterproc(
    proptr.top(),
    { idnode->name,
      nonterminal6->type,
      paranode2->paratype,
      nextstat,
      tblptr.top()
    }
);
// 创建新的函数表表项同时，添加当前的作用域（函数表）
```

6.2 引用重复计数

在上学期完成语义分析时，由于没有完成引用相关的拓展语义分析，因此对语义分析器 SemanticAnalyzer 的数据结构定义尚不完善。在本次课程设计中，为了能让编译器识别出重复引用相关的语义错误，向 SemanticAnalyzer 类添加一个新的私有成员 refCount，用于对已有变量的不可变和可变引用计数。

```
std::unordered_map<std::string, std::pair<int, int>> refCount;
//引用跟踪计数，判断多重引用是否合法，pair<不可变引用计数，可变引用计数>
```

6.3 设计体会

在设计与调试分析过程中，如果每次都直接使用 web 可视化界面调试，工作量很大且不易进行查错。因此，在后端的 C++代码文件中，我们使用了两种方式提供程序运行，通过宏定义的方式区分。当不需要向前端界面传送数据时，可以直接在 Visual Studio 中以命令行的形式运行程序，并直接在终端查看运行结果。如遇到代码错误，也可以直接利用 VS 中的调试工具或打印必要的调式语句来直接进行调试查错，如此能够有效地提升调试的效率，更快找出代码潜在的问题。

7. 心得体会

通过本次编译原理课程设计“类 Rust 编译器”的实现，我有效地将上学期编译原理课堂上学习到的有关词法分析、语法分析、语义分析和目标代码生成等编译全过程的知识运用到实际应用中。对比原本只在书本知识上学习到的理论知识，两次大作业和课程设计让我更好地加深了对这些知识的理解，并通过动手实操将这些知识转换为一个完整的链路，实现了一个自己的编译器。

对于整个项目而言，前期的词法分析和语法分析少不了我与组员的通力协作。由于我们在上学期的项目中已经提前完成了一遍扫描的代码实现，我独自完成课程设计时变得方便了许多。在一个学期多的实践中，我遇到了各种各样的问题，且由于我们在一开始就选择了使用 C++技术实现后端编译技术，整个调试过程的难度系数更是增加。通过多次的尝试与调整，我不但加深了对编译原理知识的理解，更提升了自己的 C++代码能力。

除了题目要求外，本次课程设计项目还拓展实现了复杂符号的词法分析、所有拓展文法节点的语法分析和大部分拓展节点的语义分析；中间代码和目标代码进行了可视化展示，同时支持文件保存的功能。然而，由于自己的暑期安排较多，我没有充分的时间和精力完成数组和元组部分的语义分析，实属遗憾。

我十分肯定地说，《编译原理》和《编译原理课程设计》是大学阶段我十分喜欢的专业课，因为通过“类 Rust 编译器”这一实践项目，让我对曾经望而却步的编译知识有了深入浅出之感。我很难相信，曾经觉得难度系数最大的这门专业课，可以让我学习得如此有干劲、有成就感。整个计算机的世界是神奇的，我何

其有幸能在好的课堂上，跟随好的老师去探究编译的底层逻辑。作为大学三年的最后一门必修专业课，我想这个句号已经算得上圆满。专业课的学习接近尾声，我的大学阶段也要渐渐走向落幕，这份报告应该是我的最后一份专业课实验报告了。回想过去三年在这台电脑上敲打出的无数代码与文字，不禁感慨来时路。未来，我能接触编译技术的机会或许少之又少，但如果可能，我仍旧希望自己能够在实践中继续加深对编译知识的理解。

继续向前吧，继续去探索这个广大而神奇的计算机宇宙。

8. 参考资料

[1] Alfred V A, Monica S L, Jeffrey D U. Compilers principles, techniques & tools[M]. pearson Education, 2007.

[2] 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.

[3] 张素琴,吕映芝. 编译原理[M]., 清华大学出版社.

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