華中科技大學

编译原理课程设计

C-minus编译器的设计与实现

院　　系　软件学院

专业班级

姓　　名

学　　号

指导教师　胡雯蔷

2023 年 12 月 19 日

目 录

编译原理课程实验报告

1 概述

本实验报告详细介绍了"C minus"编译器的设计与实现，它是一个简化的编译器，旨在处理"C minus"语言——一种简化版的C语言。整个编译器的设计遵循了编译原理的基本概念和方法，包括词法分析、语法分析、语义分析、中间代码生成、目标代码生成等关键步骤。本报告全面地覆盖了编译器各个模块的具体实现方法和过程，从而为理解编译器的内部工作机制提供了深入的视角。通过对编译器各个模块的深入分析，本报告不仅展示了编译器的内部工作机制，而且为编译原理的学习和理解提供了宝贵的实践经验。

2 系统描述

2.1 自定义语言概述

本次实验中设计的语言取名为"C minus"语言，下面给出其详细介绍。

2.1.1 设计目的和应用背景

"C minus" 语言被设计成一个简化的编程语言，其核心目的是理解编译原理的基本概念，包括语法分析、语义分析、中间代码生成、优化和目标代码生成等。该语言模拟了许多现代编程语言的特性，但以更简化的形式呈现，使得学习者可以更容易地把握编译器设计的核心原理。

2.1.2 语法和特性

"C minus" 语言支持基础的程序结构，包括变量声明、函数定义、基本数据类型（如整数、浮点数、字符）、以及控制流语句（如循环和条件判断）。语言设计充分考虑了编译器实现的需要，如明确的数据类型声明、函数声明和定义，以及表达式和控制流的标准化处理。

2.1.3 编译器设计的特点

"C minus" 语言的编译器实现包括多个阶段，每个阶段都展现了编译原理的关键环节。从词法分析器和语法分析器的构建，到符号表的管理，再到中间代码的生成和优化，最终到目标代码生成，每个环节都是编译原理课程中不可或缺的部分。

2.1.4 目标代码指令集 - MIPS架构

编译器的目标代码选择了 MIPS 指令集。MIPS 作为一种 RISC 架构，其指令集的简洁性和高效性使它成为教学和实践编译原理的理想选择。通过将 "C minus" 语言的中间代码转换为 MIPS 指令，编译器展示了如何将高级语言映射到特定体系结构的机器代码上，这是理解编译器后端设计的关键步骤。

2.1.5 自定义语言总结

综上所述，"C minus" 语言及其编译器的设计和实现，提供了一个理想的平台，用于教授和学习编译原理的关键概念。它通过简化的语言特性和对经典 MIPS 架构的应用，允许学习者在实践中深入理解从源代码到机器代码的整个转换过程。

2.2 单词文法与语言文法

"C minus" 语言的文法反映了其结构和语法规则。以下是一些关键的文法规则：

2.2.1 程序结构（Program Structure）：

- program: ExtDefList 定义了程序的最高级结构，即由一系列外部定义（ExtDefList）组成。

2.2.2 外部定义（External Definitions）：

- ExtDefList 可以为空，或者由一个外部定义和更多的外部定义列表组成。
- ExtDef 可以是类型声明后跟一个变量声明列表和分号，函数声明（包括类型、名称、参数列表和复合语句），或者不带函数体的函数声明。

2.2.3 类型和声明（Types and Declarations）：

- Specifier 定义了基本的数据类型（如 TYPE）。
- VarDec 定义了变量声明，可以是一个标识符或数组声明。
- ParamDec 定义了函数参数的类型和名称。

2.2.4 复合语句和控制流（Compound Statements and Control Flow）：

- CompSt 定义了复合语句，由局部变量定义列表和语句列表组成。
- Stmt 定义了不同类型的语句，包括表达式语句、复合语句、返回语句、条件语句、循环语句、跳转语句等。

2.2.5 表达式（Expressions）：

- Exp 定义了表达式的结构，可以是赋值表达式、算术运算、逻辑运算、比较运算等。
- 支持的操作包括基本的算术运算（如加、减、乘、除）、逻辑运算（如与、或、非）、关系运算（如大于、等于）等。
- 表达式可以是标识符、整数、浮点数或括号中的表达式。

2.2.6 BNF语法描述

```
program: ExtDefList

ExtDefList:
          | ExtDef ExtDefList

ExtDef:     Specifier ExtDecList SEMI
          | Specifier ID LP ParamList RP CompSt
          | Specifier ID LP ParamList RP SEMI

Specifier:  TYPE

ExtDecList: VarDec
          | VarDec COMMA ExtDecList

VarDec:     ID
          | VarDec LB INT RB

ParamVarDec:ID

ParamList:
          | ParamDec
          | ParamList COMMA ParamDec

```

```
ParamDec:   Specifier ParamVarDec

CompSt:     LC DefList StmList RC

StmList:
          | Stmt StmList

DefList:
          | Def DefList

Def:        Specifier DecList SEMI

DecList:    Dec
          | Dec COMMA DecList

Dec:        VarDec
          | VarDec ASSIGN Exp

Case:       CASE Exp COLON StmList

CaseList:   Case
          | Case CaseList

Stmt:       Exp SEMI
          | CompSt
          | RETURN Exp SEMI
          | RETURN SEMI
          | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
          | IF LP Exp RP Stmt ELSE Stmt
          | WHILE LP Exp RP Stmt
          | FOR LP Exp SEMI Exp SEMI Exp RP Stmt
          | SWITCH LP Exp RP LC CaseList RC
          | SWITCH LP Exp RP LC CaseList DEFAULT COLON StmList RC
          | BREAK SEMI
          | CONTINUE SEMI
          | error SEMI

Exp:        Exp ASSIGN Exp
          | Exp PLUS Exp
          | Exp MINUS Exp
          | Exp STAR Exp
          | Exp DIV Exp
          | Exp MOD Exp
          | LP Exp RP
          | MINUS Exp %prec UMINUS
          | PLUS Exp %prec UPLUS

          | Exp AND Exp
          | Exp OR Exp
          | NOT Exp

          | Exp GT Exp
          | Exp GE Exp
          | Exp LT Exp
          | Exp LE Exp
          | Exp NE Exp
```

```
          | Exp EQ Exp

          | DPLUS Exp
          | DMINUS Exp
          | PLUSD Exp
          | MINUSD Exp

          | ID LP Args RP
          | ID
          | ID SubList
          | INT
          | FLOAT

Args:
          |  Exp
          |  Args COMMA  Exp

Sub:         LB Exp RB

SubList:     Sub
          |  SubList Sub
```

2.3 符号表结构定义

在 "C minus" 语言的编译器设计中，符号表是一个关键的数据结构，用于存储关于源代码中标识符（如变量名、函数名）的信息。以下是符号表的详细结构和功能描述：

2.3.1 基本符号结构（Symbol 类）：

- Name: 标识符的名称。
- Type: 标识符的类型，可以是基本类型（字符、整型、浮点型、空类型）。
- Kind: 标识符的种类，例如变量（V）、函数（F）、参数（P）、数组（A）等。

```
/**************符号表定义**********************/
class Symbol {
public:
    string Name;
    BasicTypes Type;        //符号类型，目前仅基本类型T_CHAR,T_INT,T_FLOAT, T_VOID
    char Kind;       //符号种类：基本变量V，函数名F，参数P，数组A等
};
```

2.3.2 变量符号（VarSymbol 类）：

- 继承自 Symbol 类，专门用于表示变量。
- Alias: 变量的别名，解决中间代码中作用域嵌套变量同名的显示时的二义性问题。
- Offset: 变量在相应活动记录（AR）中的偏移量。
- isGlobal: 表示变量是否为全局变量。
- Dims: 数组的维度，用于表示数组类型的变量。

```
class VarSymbol : public Symbol {
public:
    string Alias;   //别名，为解决中间代码中，作用域嵌套变量同名的显示时的二义性问题
    int Offset;     //变量在对应AR中的偏移量
    int isGolbal = 0;
    vector<int> Dims;
};
```

2.3.3 函数符号（FuncSymbol 类）：

- 继承自 Symbol 类，专门用于表示函数。
- ARSize: 函数的活动记录（AR）的大小，用于函数调用时分配内存单元。
- ParamNum: 函数的形式参数个数。
- ParamPtr: 指向参数符号表的指针。
- Declaration: 表示函数是否已在符号表中定义，0 表示已定义。
- Params: 定义时指出的参数类型和数目列表。

```
class FuncSymbol : public Symbol {
public:
    int ARSize;       //函数AR的大小，作为调用时分配单元的依据
    int ParamNum;     //形式参数个数
    SymbolsInAScope *ParamPtr;  //指向参数的符号表
    int Declaration;   //目前在符号表中的是否已经定义，0表示已经定义
    //定义时指出参数类型和参数数目，声明时指出参数类型和参数数目，在定义时参考
    vector<ParamAST *> Params;

};
```

2.3.4 作用域内的符号表（SymbolsInAScope 类）：

- 表示单一作用域内的符号表，每个复合语句对应一个符号表。
- Symbols: 存储该作用域内所有符号的向量。

```
class SymbolsInAScope {    //单一作用域的符号名，每个复合语句对应一个符号表
public:
    vector<Symbol *> Symbols;
};
```

2.3.5 符号表栈（SymbolStackDef 类）：

- 用栈结构来管理不同作用域的符号表。
- 栈底通常存储全局变量和函数定义，每个复合语句对应一张局部符号表。
- 提供了在当前作用域（LocateNameCurrent）和全局作用域（LocateNameGlobal）查找符号的功能。

```
class SymbolStackDef {
    //符号表类定义,栈结构栈底为全局变量和函数定义，每个复合语句对应一张局部符号表
public:
    vector<SymbolsInAScope *> Symbols;

    Symbol *LocateNameCurrent(const string& Name);//在当前作用域中查找该符号是否有定义
    Symbol *LocateNameGlobal(string Name); //由内向外，在全部作用域中查找该符号是否有定义
};
```

2.3.6 符号表总结

这个符号表结构设计不仅支持基本的变量和函数的声明与定义，而且考虑了作用域管理、参数传递和内存布局等编译器设计的关键方面。它通过精心设计的数据结构和管理方法，有效地支持了编译过程中的名字解析、类型检查和内存分配等任务。这种符号表的设计是 "C minus" 编译器实现中的一个重要组成部分，有助于实现语言的静态语义和支持后续的代码生成。

2.4 错误类型码定义

在 "C minus" 语言的编译器设计中，错误处理是一个至关重要的环节。为此，编译器中定义了一个 Errors 类，用于记录和管理在语法分析和语义分析阶段发现的错误。以下是该类的详细描述及其在编译器中的作用：

2.4.1 错误存储（Errors 类）：

- Errs: 静态向量，用于存储错误信息。每个 Error 元素包含错误的位置（行和列）和错误消息。
- 这种存储机制允许编译器在整个编译过程中收集并保存发现的所有错误。

```
class Errors            //用来记录语法、语义错误
{
public:
    static vector <Error> Errs;

    static void ErrorAdd(int Line, int Column, string ErrMsg);

    static void ErrorsDisplay();

    static inline bool IsEmpty() { return Errs.empty(); }
};
```

2.4.2 错误添加（ErrorAdd 方法）：

- 静态函数，用于向 Errs 向量中添加新的错误。
- 接受行号、列号和错误消息作为参数，创建一个新的 Error 实例，并将其添加到 Errs 向量中。
- 这使得编译器能够在发现错误时立即记录相关信息。

```
void Errors::ErrorAdd(int Line, int Column, string ErrMsg) {
    Error e = {Line, Column, std::move(ErrMsg)};
    Errs.push_back(e);
}
```

2.4.3 错误展示（ErrorsDisplay 方法）：

- 静态函数，用于展示收集到的所有错误信息。
- 遍历 Errs 向量，并打印每个错误的详细信息（包括位置和消息）。
- 这对于用户理解和修正代码中的错误至关重要。

```
void Errors::ErrorsDisplay() {
    for (const auto& a: Errs)
        cout << "第" << a.Line << "行、第" << a.Column << "列处错误: " << a.ErrMsg << endl;
}
```

2.4.4 错误检查（IsEmpty 方法）：

- 提供快速检查 Errs 向量是否为空的功能，即编译过程中是否发现了错误。
- 这对于在编译器的不同阶段判断是否存在错误非常有用。

```
static inline bool IsEmpty() {
    return Errs.empty();
}
```

2.4.5 错误类型总结

通过 Errors 类的设计，"C minus" 编译器能够有效地处理和报告在代码编译过程中遇到的各种错误。这种错误处理机制不仅对编译器用户（程序员）友好，提供了清晰的错误信息和定位，也对编译器的开发和维护至关重要，因为它使得识别和修复编译器本身的缺陷变得更加容易。

2.5 中间代码结构定义

在 "C minus" 语言的编译器设计中，中间代码是源代码与目标代码之间的重要阶段。中间代码的设计旨在提供一个与特定机器无关的代码表示，使得编译器的后端可以更容易地将其转换为目标机器代码。在本项目中，中间代码采用了四元式的形式，以下是其详细结构和功能描述：

2.5.1 操作数（Opn 类）：

- Name: 表示变量的别名（对于变量而言）或函数名。如果为空，则表示该操作数是一个常量。
- Type: 表示操作数的类型（如整数、浮点数、字符等）。
- isGlobal: 表示该操作数是否为全局变量。
- Offset/SymPtr/constCHAR/constINT/constFLOAT: 根据操作数的不同类型，这个联合体可以存储变量在活动记录（AR）中的偏移量、符号表指针或常量值。

```
/**************中间代码（四元式）定义**********************/

class Opn {
public:
    string Name;        //变量别名（为空时表示常量）或函数名
    int Type{};
    int isGolbal = 0;
    union {
        int Offset{};      //AR中的偏移量
        void *SymPtr;     //符号表指针
```

```
        char constCHAR;
        int constINT;
        float constFLOAT;
    };

    Opn(string Name, int Type, int Offset, int isGolbal) : Name(std::move(Name)),
Type(Type), Offset(Offset), (isGolbal) {};

    Opn() {};
};
```

2.5.2 中间代码（四元式，IRCode 类）：

- Op: 表示四元式的操作类型，如加法、减法、乘法、除法等。
- Opn1 和 Opn2: 表示操作的第一和第二操作数。
- Result: 表示操作的结果。

```
class IRCode       //四元式结构
{
public:
    int Op;
    Opn Opn1;
    Opn Opn2;
    Opn Result;

    IRCode(int Op, Opn Opn1, Opn Opn2, Opn Result) : Op(Op), Opn1(std::move(Opn1)),
Opn2(std::move(Opn2)), Result(std::move(Result)) {}
};

```

2.5.3 四元式的表示和应用:

- 每个 IRCode 实例表示一个中间代码的操作，包括了进行什么样的操作、操作的对象以及操作的结果。
- 例如，一个四元式可以表示为 Opn1 + Opn2 -> Result，其中 Op 指定了 + 操作，Opn1 和 Opn2 作为输入，Result 作为输出。

```
default:     //处理关系运算符
    Opn Opn1 = LeftExp->GenIR(TempVarOffset);
    Opn Opn2 = RightExp->GenIR(TempVarOffset);
    it = IRCodes.end();
    IRCodes.splice(it, LeftExp->IRCodes);
    it = IRCodes.end();
    IRCodes.splice(it, RightExp->IRCodes);
    IRCode IR(JGT, Opn1, Opn2, Opn(LabelTrue, 0, 0, 0));
    if (Op == GE) IR.Op = JGE;
    else if (Op == LT) IR.Op = JLT;
    else if (Op == LE) IR.Op = JLE;
    else if (Op == EQ) IR.Op = JEQ;
    else if (Op == NE) IR.Op = JNE;
    IRCodes.emplace_back(IR);
    IRCodes.emplace_back(GOTO, Opn(), Opn(), Opn(LabelFalse, 0, 0, 0));
```

2.5.4 中间代码总结

这种中间代码的设计使得编译器能够以一种标准化和简化的形式表示各种复杂的操作，从而为代码优化和目标代码生成提供了便利。四元式作为一种通用的中间表示，可以容易地被转换成各种目标机器的指令集，提高了编译器的可移植性和灵活性。此外，四元式的清晰结构也便于进行各种中间代码优化技术的实现，如公共子表达式消除、死代码消除等，从而提升了编译后代码的效率和性能。

2.6 目标代码指令集选择

在 "C minus" 编译器项目中，目标代码指令集选择了 MIPS（Microprocessor without Interlocked Pipeline Stages）架构。MIPS 架构是一种经典的精简指令集计算（RISC）体系结构，广泛用于教育和嵌入式系统。以下是 MIPS 架构的详细介绍：

2.6.1 RISC 设计理念:

- MIPS 架构基于 RISC 设计理念，这意味着它使用了一组简单且常用的指令，与复杂指令集计算（CISC）体系结构如 x86 形成对比。
- RISC 架构的优势在于指令执行的高效率和更简单的硬件实现。

2.6.2 寄存器使用:

- MIPS 指令集包含了一组固定的 32 个通用寄存器，每个寄存器均为 32 位宽。
- 包括专用寄存器如程序计数器（PC）、栈指针（$sp）、全局指针（$gp）、返回地址（$ra）等，以及用于函数参数传递（$a0-$a3）和函数返回值（$v0, $v1）的寄存器。

2.6.3 指令格式:

- MIPS 指令集有三种基本格式：R型（寄存器指令）、I型（立即数指令）和 J型（跳转指令）。
- 这种统一的指令格式简化了指令的解码过程，使得硬件实现更为高效。

2.6.4 流水线执行:

- MIPS 支持流水线指令执行，这是实现高指令吞吐率的关键技术之一。
- 流水线执行允许在一个时钟周期内同时执行多个指令的不同阶段，从而提高执行效率。

2.6.5 系统调用和中断处理:

- MIPS 架构提供了系统调用指令（syscall）用于执行操作系统级别的功能，如输入输出操作。
- 它还支持中断和异常处理，这对于实现多任务操作系统和响应外部事件至关重要。

2.6.6 应用领域:

- 由于其简单和高效的特性，MIPS 架构被广泛应用于教育领域，用于教授计算机组成原理和编译原理。
- 同时，MIPS 架构也被广泛用于嵌入式系统和网络设备，如路由器和交换机。

2.6.7 目标代码指令集总结

在 "C minus" 编译器项目中，选择 MIPS 作为目标代码指令集是基于其教学价值以及简洁高效的指令集设计。通过将中间代码转换为 MIPS 指令，编译器不仅能有效地演示编译原理的关键概念，如指令选择、寄存器分配和函数调用约定，还能体现 RISC 设计理念的优势。

```
1 #include "def.h"
2 #include <fstream>
```

```

#define YYSTYPE int   //此行是为了包含parser.tab.hpp不引起错误而加,可以在后面使用相关常量

#include "parser.tab.hpp"

string LoadFromMem(const string& Reg1, const Opn& opn, string Reg2) {
    string load;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn.isGolbal == 0)))
        Reg2 = "$gp";
    load = "  lw " + Reg1 + ",  " + to_string(opn.Offset) + "(" + Reg2 + ")";
    return load;
}

string LoadFromMem(const string& Reg1, const Opn& opn1, const Opn& opn2, string Reg2)
{
    string load;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn1.isGolbal == 0)))
        Reg2 = "$gp";
    load = "  lw $t4,  " + to_string(opn2.Offset) + "(" + "$sp" + ")\n" +
           "  add " + Reg2 + ", " + Reg2 + ", " + "$t4\n" +
           "  lw " + Reg1 + ",  " + to_string(opn1.Offset) + "(" + Reg2 + ")\n" +
           "  sub " + Reg2 + ", " + Reg2 + ", $t4";
    return load;
}

string StoreToMem(const string& Reg1, const Opn& opn, string Reg2) {
    string store;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn.isGolbal == 0)))
        Reg2 = "$gp";
    store = "  sw " + Reg1 + ",  " + to_string(opn.Offset) + "(" + Reg2 + ")";
    return store;
}

string StoreToMem(const string& Reg1, const Opn& opn1, const Opn& opn2, string Reg2) {
    string store;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn1.isGolbal == 0)))
        Reg2 = "$gp";
    store = "  lw $t4,  " + to_string(opn2.Offset) + "(" + "$sp" + ")\n" +
            "  add " + Reg2 + ", " + Reg2 + ", " + "$t4\n" +
            "  sw " + Reg1 + ",  " + to_string(opn1.Offset) + "(" + Reg2 + ")\n" +
            "  sub " + Reg2 + ", " + Reg2 + ", $t4";
    return store;
}
```

3 系统设计与实现

3.1 词法分析器

词法分析器是编译过程中的首要阶段，它负责将源代码文本转换为一系列标记（tokens），为后续的语法分析阶段做准备。在 "C minus" 语言的编译器设计中，词法分析器的实现基于 Flex 工具。以下是对该词法分析器的详细描述：

3.1.1 词法规则定义:

- 词法分析器通过一系列正则表达式规则定义了 "C minus" 语言的词法单元，包括关键字（如 if, else, while）、标识符、常量（整型和浮点型）、运算符（如 +, -, *, /）、分隔符等。
- 这些规则确保了词法分析器可以准确地识别源代码中的各种基本元素。

3.1.2 动作执行:

- 对于每个匹配的规则，词法分析器执行相应的动作，这些动作通常包括返回一个特定的标记给语法分析器，并可能包括将识别的值（如常量的值、标识符的名称）保存到相应的数据结构中。
- 例如，对于匹配到的标识符，词法分析器会将其名称保存并返回 ID 标记。

3.1.3 错误处理:

- 词法分析器能够识别并报告不合法的字符或构造，如未知的符号或不完整的注释。
- 对于每个不可识别的符号，词法分析器打印错误信息并增加错误计数。

3.1.4 位置跟踪:

- 词法分析器通过内置的 yylineno 和自定义的 yycolumn 变量跟踪当前标记的行号和列号，这对于错误报告和调试非常重要。
- 在每个标记的动作代码中，更新这些变量以反映当前标记的位置。

3.1.5 注释处理:

词法分析器通过特定的规则识别并忽略源代码中的注释，支持单行（//）和多行（/* ... */）注释，如下列的正则表达式所示：

```
Oneline_comment     "//"[^\n]*
Multiline_comment   "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/"
```

3.1.6 lex.l 源代码

```
%option  yylineno
%{
#include "string.h"

int ErrorCharNum=0;
int yycolumn=1;

#define YY_USER_ACTION    \
    yylloc.first_line=yylloc.last_line=yylineno; \
    yylloc.first_column=yycolumn;\
    yylloc.last_column=yycolumn+yyleng-1;\
    yycolumn+=yyleng;

typedef struct {
    int type_int;
    int type_float;
    char type_id[32];
} YYLVAL;
#define YYSTYPE YYLVAL
#include "parser.tab.hpp"

```

```
%}

id   [A-Za-z][A-Za-z0-9]*
intconst    0|([1-9][0-9]*)
floatconst    [0-9]*\.?[0-9]?([eE][-+]?[0-9]+)?
Oneline_comment     "//"[^\n]*
Multiline_comment   "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/"

%%
"int"       {strcpy(yylval.type_id,  yytext);return TYPE;}
"float"     {strcpy(yylval.type_id,  yytext);return TYPE;}
"void"      {strcpy(yylval.type_id,  yytext);return TYPE;}

"return"    {return RETURN;}
"if"        {return IF;}
"else"      {return ELSE;}
"while"     {return WHILE;}
"for"       {return FOR;}

"break"     {return BREAK;}
"continue"  {return CONTINUE;}

"switch"    {return SWITCH;}
"case"      {return CASE;}
"default"   {return DEFAULT;}

{id}    {strcpy(yylval.type_id,yytext); return ID;}
":"     {return COLON;}
";"     {return SEMI;}
","     {return COMMA;}
">="    {strcpy(yylval.type_id, yytext);return GE;}
">"     {strcpy(yylval.type_id, yytext);return GT;}
"<="    {strcpy(yylval.type_id, yytext);return LE;}
"<"     {strcpy(yylval.type_id, yytext);return LT;}
"!="    {strcpy(yylval.type_id, yytext);return NE;}
"=="    {strcpy(yylval.type_id, yytext);return EQ;}
"="     {return ASSIGN;}
"++"    {return DPLUS;}
"--"    {return DMINUS;}
"+"     {return PLUS;}
"-"     {return MINUS;}
"*"     {return STAR;}
"/"     {return DIV;}
"%"     {return MOD;}
"&&"    {return AND;}
"||"    {return OR;}
"!"     {return NOT;}
"("     {return LP;}
")"     {return RP;}
"{"     {return LC;}
"}"     {return RC;}
"["     {return LB;}
"]"     {return RB;}
{intconst}      { yylval.type_int=atoi(yytext); return INT;}
{floatconst}    { yylval.type_float=atof(yytext);     return FLOAT;}
[ \r\t]         {}
```

```
{Oneline_comment}           {}      //单行注释
{Multiline_comment}         {}      //多行注释


[\n]            {yycolumn=1;}
.               { printf("在第 %d 行出现不可识别的符号 \'%s\' \n",yylineno,yytext);
ErrorCharNum++;}

%%

int yywrap()
{
return 1;
}
```

3.1.7 词法分析器总结

通过上述设计和实现，"C minus" 语言的词法分析器能够高效地将源代码文本转换为一系列标记，为后续的编译过程奠定了坚实的基础。它的实现展示了编译原理中词法分析阶段的关键技术和方法，如词法规则的定义、标记的生成、位置跟踪和错误处理，是编译器设计的基础和关键部分。

3.2 语法分析器

语法分析器在编译过程中扮演着至关重要的角色，负责根据词法分析器提供的标记序列，构建源程序的抽象语法树（AST）。在 "C minus" 编译器中，语法分析器的实现基于 Bison 工具。以下是对该语法分析器的全面描述：

3.2.1 语法规则定义:

语法分析器利用一系列产生式规则定义了 "C minus" 语言的语法结构，包括程序结构、函数定义、变量声明、表达式、控制流语句等。这些规则以递归下降的方式编写，确保了语言的语法结构能够被准确地识别和解析。

3.2.2 AST节点创建:

- 对于每个产生式规则，语法分析器在匹配时创建相应的AST节点，例如 ProgAST、FuncDefAST、VarDecAST 等。
- 通过这种方式，语法分析器将源代码转换为结构化的AST，为后续的语义分析和代码生成阶段奠定了基础。

```
#define SavePosition t->Line=yylloc.first_line;t->Column=yylloc.first_column
typedef struct YYLVAL {
    int                 type_int;
    float               type_float;
    char                type_id[32];

    ProgAST                     *program;
    vector <ExtDefAST *>        ExtDefList;         //外部定义（外部变量、函数）列表
    ExtDefAST                   *ExtDef;
    vector <VarDecAST*>         ExtDecList;        //外部、局部变量列表
    TypeAST                     *Specifier;
    VarDecAST                   *VarDec;
    CompStmAST                  *CompSt;
    vector <ParamAST *>         ParamList;         //形参列表
    ParamAST                    *ParamDec;

    vector <StmAST *>           StmList;
    StmAST                      *Stmt;
```

```
    vector <DefAST *>           DefList;
    DefAST                      *Def;
    vector <VarDecAST *>        DecList;
    VarDecAST                   *Dec;
    ExpAST                      *Exp;
    vector <ExpAST *>           Args;                   //实参列表
    CaseStmAST                  *Case;
    vector <CaseStmAST *>       CaseList;
}YYLVAL;
#define YYSTYPE YYLVAL
```

3.2.3 错误处理和恢复:

- 语法分析器能够识别语法错误，并通过 yyerror 函数输出错误信息。
- 错误处理机制包括记录错误位置和错误信息，这对于开发者和用户调试程序至关重要。

```
#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
    Errors::ErrorAdd(yylloc.first_line,yylloc.first_column,fmt);
}
```

3.2.4 语义值和位置跟踪:

- Bison 的 %locations 指令用于启用位置追踪，记录每个语法单元的行和列信息。
- %type 指令用于定义非终结符的语义值类型，如 program、ExtDefList 等，这些类型对应于不同的AST节点。

```
//  %type 定义非终结符的语义值类型
%type  <program>    program
%type  <ExtDefList> ExtDefList
%type  <ExtDef>     ExtDef
%type  <ExtDecList> ExtDecList
%type  <Specifier>  Specifier
%type  <VarDec>     VarDec
%type  <VarDec>     ParamVarDec
%type  <CompSt>     CompSt
%type  <ParamList>  ParamList
%type  <ParamDec>   ParamDec
%type  <DefList>    DefList
%type  <StmList>    StmList
%type  <Stmt>       Stmt
%type  <Def>        Def
%type  <DecList>    DecList
%type  <Dec>        Dec
%type  <Exp>        Exp
%type  <Exp>        Sub
%type  <Args>       SubList
%type  <Case>       Case;
%type  <CaseList>   CaseList


%type  <Args>  Args
```

3.2.5 操作符优先级和结合性:

- %left、%right 和 %nonassoc 指令定义了操作符的优先级和结合性，这对于解析具有潜在歧义的表达式（如算术表达式）非常重要。

```
%left COMMA
%left ASSIGN
%left OR
%left AND
%left LT LE GT GE
%left NE EQ
%left MOD
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT DPLUS DMINUS UPLUS
%left PLUSD MINUSD
%left ARRPRO

%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE
```

3.2.6 语法分析过程:

- yyparse 函数控制整个语法分析过程，从读取输入（源代码）开始，到最终的AST构建结束。
- 在解析过程中，语法分析器不断从词法分析器获取标记，并根据定义的产生式规则进行匹配和处理。

3.2.7 parser.ypp 源代码

```
%define parse.error verbose
%locations
%{
#include "def.h"
extern int ErrorCharNum;
extern int yylineno;
extern char *yytext;
extern FILE *yyin;
void yyerror(const char* fmt, ...);
extern "C" int yylex();
#define SavePosition t->Line=yylloc.first_line;t->Column=yylloc.first_column
typedef struct YYLVAL {
       int                  type_int;
    float                type_float;
    char                 type_id[32];

    ProgAST                     *program;
    vector <ExtDefAST *>        ExtDefList;         //外部定义（外部变量、函数）列表
       ExtDefAST                   *ExtDef;
       vector <VarDecAST*>         ExtDecList;        //外部、局部变量列表
       TypeAST                     *Specifier;
       VarDecAST                   *VarDec;
       CompStmAST                  *CompSt;
    vector <ParamAST *>         ParamList;         //形参列表
    ParamAST                    *ParamDec;

```

```
    vector <StmAST *>           StmList;
    StmAST                      *Stmt;
    vector <DefAST *>           DefList;
    DefAST                      *Def;
       vector <VarDecAST *>        DecList;
       VarDecAST                   *Dec;
       ExpAST                      *Exp;
    vector <ExpAST *>           Args;                   //实参列表
       CaseStmAST                  *Case;
       vector <CaseStmAST *>       CaseList;
}YYLVAL;
#define YYSTYPE YYLVAL

%}
//  %type 定义非终结符的语义值类型
%type  <program>    program
%type  <ExtDefList> ExtDefList
%type  <ExtDef>     ExtDef
%type  <ExtDecList> ExtDecList
%type  <Specifier>  Specifier
%type  <VarDec>     VarDec
%type  <VarDec>     ParamVarDec
%type  <CompSt>     CompSt
%type  <ParamList>  ParamList
%type  <ParamDec>   ParamDec
%type  <DefList>    DefList
%type  <StmList>    StmList
%type  <Stmt>       Stmt
%type  <Def>        Def
%type  <DecList>    DecList
%type  <Dec>        Dec
%type  <Exp>        Exp
%type  <Exp>        Sub
%type  <Args>       SubList
%type  <Case>       Case;
%type  <CaseList>   CaseList


%type  <Args>  Args


//% token 定义终结符的语义值类型
%token <type_int> INT                   /*指定INT常量的语义值是type_int，由词法分析得到的整数数值*/
%token <type_id>  ID TYPE               /*指定ID 的语义值是type_id，由词法分析得到的标识符字符串*/
%token <type_float> FLOAT               /*指定float常量的语义值是type_float*/

%token DPLUS DMINUS PLUSD MINUSD LP RP LB RB LC RC SEMI COMMA
/*用bison对该文件编译时，带参数-d，生成的exp.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码*/
%token PLUS MINUS STAR DIV MOD GE GT LE LT NE EQ ASSIGN AND OR NOT IF ELSE WHILE RETURN FOR
%token BREAK CONTINUE SWITCH CASE DEFAULT COLON

/*以下为接在上述token后依次编码的枚举常量，用于后续过程*/
```

```
%token ARRPRO EXT_DEF_LIST EXT_VAR_DEF FUNC_DEF FUNC_DEC EXT_DEC_LIST PARAM_LIST
PARAM_DEC VAR_DEF DEC_LIST DEF_LIST COMP_STM STM_LIST EXP_STMT IF_THEN IF_THEN_ELSE
%token FUNC_CALL ARGS FUNCTION PARAM ARG CALL CALL0 LABEL GOTO JLT JLE JGT JGE JEQ
JNE END ARRASSIGN ARRLOAD ARRDPLUS ARRDMINUS ARRPLUSD ARRMINUSD


%left COMMA
%left ASSIGN
%left OR
%left AND
%left LT LE GT GE
%left NE EQ
%left MOD
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT DPLUS DMINUS UPLUS
%left PLUSD MINUSD
%left ARRPRO

%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE


%%

program: ExtDefList  {$$=new ProgAST(); $$->ExtDefs=$1;
                      if (Errors::IsEmpty() && ErrorCharNum==0)
                           { $$->DisplayAST(0); }         //无词法、语法错误显示语法树
                      else {Errors::ErrorsDisplay();return 0;}
                      $$->Semantics0();                   //静态语义检查
                      if (Errors::IsEmpty())
                          $$->GenIR();                    //中间代码生成
                      exit(0);
                      }
         ;
ExtDefList: {$$=vector <ExtDefAST*>();}
          | ExtDef ExtDefList {$2.insert($2.begin(),$1);$$=$2;}
//将ExtDef所指外部定义对象增加到（程序对象的）ExtDefList中
          ;

ExtDef:   Specifier ExtDecList SEMI   { ExtVarDefAST *t=new ExtVarDefAST();
                                       //创建一个外部变量声明的对象
                                        t->Type=$1; t->ExtVars=$2; $$=t;
SavePosition;}
         | Specifier ID LP ParamList RP CompSt {FuncDefAST *t=new FuncDefAST();t-
>Type=$1;t->Name=$2;t->Params=$4; t->Body=$6;$$=t;SavePosition;}//对应一个函数定义对象
         | Specifier ID LP ParamList RP SEMI   {FuncDefAST *t=new FuncDefAST();t-
>Type=$1;t->Name=$2;t->Params=$4;$$=t;SavePosition;}//对应一个函数声明对象，Body为空
         ;
Specifier: TYPE  {  BasicTypeAST *t=new BasicTypeAST(); ;
                    if (string($1)==string("int"))    t->Type=T_INT;
                    if (string($1)==string("float"))  t->Type=T_FLOAT;
                    if (string($1)==string("void"))   t->Type=T_VOID;
 $$=t;SavePosition;}
             ;

```

```
ExtDecList:  VarDec  {$$=vector < VarDecAST*>();$$.push_back($1);}
/*ExtDecList对应一个外部变量VarDec的序列,目前后续只考虑是标识符，可扩展为数组*/
           | VarDec COMMA ExtDecList {$3.insert($3.begin(),$1);$$=$3;}
           ;
VarDec:  ID   {VarDecAST *t=new VarDecAST(); t->Name=string($1); $$=t; SavePosition;}
            //变量对象，dims.size()为0表示简单变量，大于0表示数组
          | VarDec LB INT RB   {$1->Dims.push_back($3);$$=$1;}
//将数组的每维大小添加到属性Dims中
          ;
ParamVarDec:ID   {VarDecAST *t=new VarDecAST(); t->Name=string($1); $$=t;
SavePosition;}             //变量对象，dims.size()为0表示简单变量，大于0表示数组
          ;

ParamList: {$$=vector < ParamAST *>();}
        | ParamDec  {$$=vector < ParamAST *>(); $$.push_back($1); }
    //初始化形式参数序列
        | ParamList COMMA  ParamDec  {$1.push_back($3); $$=$1;}
    //添加一个形式参数
        ;
ParamDec: Specifier ParamVarDec   {ParamAST* t=new ParamAST();t->Type=$1;t-
>ParamName=$2; $$=t; SavePosition;}
         ;

CompSt: LC DefList StmList RC    {CompStmAST *t=new CompStmAST();t->Decls=$2;t-
>Stms=$3;$$=t;SavePosition;}
       ;
StmList: {$$=vector <StmAST *>(); }
        | Stmt StmList  {$$=$2;$$.insert($$.begin(),$1);}
        ;
DefList: {$$=vector <DefAST *>();  }
        | Def DefList {$$=$2;$$.insert($$.begin(),$1);}
        ;
Def:   Specifier DecList SEMI {DefAST *t=new DefAST();t->Type=$1;t-
>LocVars=$2;$$=t;SavePosition;}
              ;
DecList: Dec  {$$=vector <VarDecAST *>(); $$.push_back($1);}
       | Dec COMMA DecList  {$$=$3;$$.insert($$.begin(),$1);}
       ;
Dec:   VarDec  {$$=$1;}
    //如何将多种形式的局部变量加上一个父类，简单，数组，初始化
       | VarDec ASSIGN Exp  {$$=$1;$$->Exp=$3; }
    //带初始化的变量定义
      ;

Case:   CASE Exp COLON StmList  {CaseStmAST *t=new CaseStmAST(); t->Cond=$2; t-
>Body=$4; $$=t; SavePosition;}

CaseList:Case               {$$=vector <CaseStmAST *>(); $$.push_back($1); }
       | Case CaseList      {$$=$2; $$.insert($$.begin(),$1); }

Stmt:   Exp SEMI            {ExprStmAST *t=new ExprStmAST();t-
>Exp=$1;$$=t;SavePosition;}
      | CompSt              {$$=$1;}      //复合语句不再生成新的结点
      | RETURN Exp SEMI     {ReturnStmAST *t=new ReturnStmAST();t-
>Exp=$2;$$=t;SavePosition;}
```

```
176       | RETURN SEMI     {ReturnStmAST *t=new ReturnStmAST();t->Exp=NULL;$$=t;SavePosition;}
177       | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE {IfStmAST *t=new IfStmAST();t->Cond=$3;t->ThenStm=$5;$$=t; SavePosition;}
178       | IF LP Exp RP Stmt ELSE Stmt             {IfElseStmAST *t=new IfElseStmAST();t->Cond=$3;t->ThenStm=$5;t->ElseStm=$7;$$=t;SavePosition;}
179       | WHILE LP Exp RP Stmt    {WhileStmAST *t=new WhileStmAST();t->Cond=$3;t->Body=$5; $$=t; SavePosition; }
180       | FOR LP Exp SEMI Exp SEMI Exp RP Stmt
181               {ForStmAST *t=new ForStmAST(); t->SinExp=$3; t->Cond=$5; t->EndExp=$7; t->Body=$9; $$=t; SavePosition;}
182       | SWITCH LP Exp RP LC CaseList RC   {SwitchStmAST *t=new SwitchStmAST(); t->Exp=$3; t->Cases=$6; t->containDefault=0; $$=t; SavePosition;}
183       | SWITCH LP Exp RP LC CaseList DEFAULT COLON StmList RC
184               {SwitchStmAST *t=new SwitchStmAST(); t->Exp=$3; t->Cases=$6; t->containDefault=1; t->Default=$9; $$=t; SavePosition;}
185       | BREAK SEMI          {BreakStmAST *t=new BreakStmAST(); $$=t; SavePosition; }
186       | CONTINUE SEMI       {ContinueStmAST *t=new ContinueStmAST(); $$=t; SavePosition; }
187       | error SEMI   {$$=NULL;}
188       ;
189
190 Exp:    Exp ASSIGN Exp {AssignAST *t=new AssignAST();t->Op=ASSIGN;
191                 t->LeftValExp=$1;t->RightValExp=$3;$$=t;SavePosition;}
192
193       | Exp PLUS Exp {BinaryExprAST *t=new BinaryExprAST();t->Op=PLUS;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}    //算术运算符
194       | Exp MINUS Exp{BinaryExprAST *t=new BinaryExprAST();t->Op=MINUS;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
195       | Exp STAR Exp {BinaryExprAST *t=new BinaryExprAST();t->Op=STAR;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
196       | Exp DIV Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=DIV;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
197       | Exp MOD Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=MOD;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
198       | LP Exp RP    {$$=$2;}
199       | MINUS Exp %prec UMINUS  {UnaryExprAST *t=new UnaryExprAST();t->Op=UMINUS;t->Exp=$2;$$=t;SavePosition;}  //单目减
200       | PLUS Exp %prec UPLUS    {UnaryExprAST *t=new UnaryExprAST();t->Op=UPLUS;t->Exp=$2;$$=t;SavePosition;}   //单目加
201
202       | Exp AND Exp   {BinaryExprAST *t=new BinaryExprAST();t->Op=AND;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;} //逻辑运算符
203       | Exp OR Exp    {BinaryExprAST *t=new BinaryExprAST();t->Op=OR;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
204       | NOT Exp       {UnaryExprAST *t=new UnaryExprAST();t->Op=NOT;t->Exp=$2;$$=t;SavePosition;}
205
206       | Exp GT Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=GT;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}  //关系运算符
207       | Exp GE Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=GE;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
208       | Exp LT Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=LT;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
209       | Exp LE Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=LE;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
```

```
      | Exp NE Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=NE;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}
      | Exp EQ Exp  {BinaryExprAST *t=new BinaryExprAST();t->Op=EQ;t->LeftExp=$1;t->RightExp=$3;$$=t;SavePosition;}


      | DPLUS  Exp      {UnaryExprAST *t=new UnaryExprAST();t->Op=DPLUS;t->Exp=$2;$$=t;SavePosition;}    //自增、自减运算符，可区分前后缀形式
      | DMINUS  Exp     {UnaryExprAST *t=new UnaryExprAST();t->Op=DMINUS;t->Exp=$2;$$=t;SavePosition;}
      | Exp DPLUS       {UnaryExprAST *t=new UnaryExprAST();t->Op=PLUSD;t->Exp=$1;$$=t;SavePosition;}
      | Exp DMINUS      {UnaryExprAST *t=new UnaryExprAST();t->Op=MINUSD;t->Exp=$1;$$=t;SavePosition;}


      | ID LP Args RP %prec ARRPRO  {FuncCallAST *t=new FuncCallAST();t->Name=$1;t->Params=$3;$$=t;SavePosition;}
      | ID              {VarAST *t=new VarAST();t->Name=$1;$$=t;SavePosition;}
      | ID SubList          {VarAST *t=new VarAST();t->Name=$1;t->index=$2;$$=t;SavePosition;}
      | INT             {ConstAST *t=new ConstAST();t->Type=T_INT;t->ConstVal.constINT=$1;$$=t;SavePosition;}
      | FLOAT           {ConstAST *t=new ConstAST();t->Type=T_FLOAT;t->ConstVal.constFLOAT=$1;$$=t;SavePosition;}
       ;
Args:   {}
       |  Exp {$$=vector <ExpAST *>(); $$.push_back($1); }
       |  Args COMMA  Exp   {$$=$1;$$.push_back($3);}
       ;

Sub:     LB Exp RB           {$$=$2; }

SubList: Sub                 {$$=vector <ExpAST *>(); $$.push_back($1); }
       | SubList Sub         {$$=$1; $$.push_back($2);}

%%

int main(int argc, char *argv[]){
    yyin=fopen(argv[1],"r");
    if (!yyin) return 0;
    yylineno=1;
    yyparse();
    return 0;
    }

#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
    Errors::ErrorAdd(yylloc.first_line,yylloc.first_column,fmt);
}
```

3.2.8 语法分析总结

通过这种设计，"C minus" 语法分析器能够准确地解析源代码，构建出反映程序结构的AST。这不仅展现了编译原理中语法分析的关键技术，也反映了构建高效且准确语法分析器的实际挑战。该语法分析器的实现是理解编译器核心功能的重要部分，特别是在处理复杂的语法结构和确保编译器的健壮性方面。

3.3 符号表管理

在编译过程中，符号表管理是一个关键环节，它负责存储和管理源程序中使用的标识符（如变量名、函数名）及其属性（如类型、作用域）。在 "C minus" 语言编译器中，符号表管理的实现具有以下特点：

3.3.1 符号表结构:

- 符号表使用堆栈结构（SymbolStackDef 类）来管理不同作用域的符号表（SymbolsInAScope 类）。
- 符号表项（Symbol 类）包含基本属性，如标识符名称、类型、种类（变量、函数、形参、数组等）。
- 特定于变量的符号表项（VarSymbol 类）还包括别名、偏移量、是否全局以及数组维度信息。
- 函数符号表项（FuncSymbol 类）包含函数的活动记录大小、参数个数、参数列表指针等。

```
class SymbolStackDef {
    //符号表类定义,栈结构栈底为全局变量和函数定义，每个复合语句对应一张局部符号表
public:
    vector<SymbolsInAScope *> Symbols;
    //在当前作用域中查找该符号是否有定义
    Symbol *LocateNameCurrent(const string& Name);
    //由内向外，在全部作用域中查找该符号是否有定义
    Symbol *LocateNameGlobal(const string& Name);

};
```

3.3.2 符号表操作:

- 符号表支持在当前作用域和全局作用域中查找标识符，以及将新的标识符加入符号表。
- 通过 LocateNameCurrent 和 LocateNameGlobal 函数实现标识符的查找。
- 新的符号表项在语义分析过程中创建并添加到符号表中。

```
Symbol *SymbolStackDef::LocateNameCurrent(const string& Name)  //在当前(最内层)作用域中查找该符号是否有定义
{
    SymbolsInAScope *curScope = Symbols.back();
    for (auto & Symbol : curScope->Symbols)
        if (Symbol->Name == Name)
            return Symbol;
    return nullptr;
}

Symbol *SymbolStackDef::LocateNameGlobal(const string& Name)//由内向外，整个符号表中查找该符号是否有定义
{
    for (int i = Symbols.size() - 1; i >= 0; i--) {
        for (auto & Symbol : Symbols.at(i)->Symbols)
            if (Symbol->Name == Name)
```

```
                return Symbol;

    }
    return nullptr;
}
```

3.3.3 符号表的展示:

- 提供了 DisplaySymbolTable 函数来打印符号表的当前状态，这对于编译过程的调试非常有用。
- 该函数显示每个作用域的符号，包括名称、别名、类型、类别和其他相关信息。

```

void DisplaySymbolTable(SymbolStackDef *SYM) {
    for (auto i = 0; i < SYM->Symbols.size(); i++) {
        cout << "------------------------------------------------------------------------
-" << endl;
        cout << " 层号: " << i << endl;
        cout << " 符 号 名           别名     类型     种 类    其它信息" << endl;
        cout << "------------------------------------------------------------------------
-" << endl;
        if (SYM->Symbols.at(i)->Symbols.empty())
            cout << "  空 表" << endl;
        else
            for (auto SymPtr : SYM->Symbols.at(i)->Symbols) {
                //取第i层第j个符号对象的指针
                cout.width(20);
                cout << SymPtr->Name;
                cout.width(8);
                if (SymPtr->Kind == 'V' || SymPtr->Kind == 'A' || SymPtr->Kind == 'P')
  //符号是变量,形参,显示别名
                    cout << ((VarSymbol *) SymPtr)->Alias;
                else cout << " ";
                cout.width(8);
                cout << SymbolMap[SymPtr->Type];
                cout.width(8);
                cout << KindName[SymPtr->Kind];
                if (SymPtr->Kind == 'V' || SymPtr->Kind == 'P')   //符号是变量,形参
                    cout << "偏移量: " << ((VarSymbol *) SymPtr)->Offset << "  全局: "
                         << ((VarSymbol *) SymPtr)->isGolbal;
                else if (SymPtr->Kind == 'F')  //符号是函数
                {
                    cout << "形参数: " << ((FuncSymbol *) SymPtr)->ParamNum;
                    cout << "  变量空间: " << ((FuncSymbol *) SymPtr)->ARSize;
                } else if (SymPtr->Kind == 'A') //符号是数组，需要显示各维大小
                {
                    cout << "偏移量: " << ((VarSymbol *) SymPtr)->Offset << "  全局: "
                         << ((VarSymbol *) SymPtr)->isGolbal << "  ";
                    cout << ((VarSymbol *) SymPtr)->Dims.size() << "维: ";
                    for (int Dim : ((VarSymbol *) SymPtr)->Dims)
                        cout << Dim << " ";
                }
                cout << endl;
            }
        cout << "------------------------------------------------------------------------
-" << endl;
```

```
    }
}
```

3.3.4 函数调用表:

- 编译器还维护了一个函数调用表（FunctionCallTable 类），用于记录函数调用的信息，包括行号、列号和函数名。这对于检查未定义函数的调用和处理函数调用的语义分析非常重要。

```
class FunctionCallTable {
public:
    vector <FunctionCall> FuncCalls;

    void addFuncCalls(int Line, int Column, string Name);

    void deleteFuncCalls(string Name);
};
```

3.3.5 错误处理:

编译器使用符号表进行静态语义检查，例如检查变量和函数的重复声明、类型匹配等。通过符号表，编译器可以判断标识符是否已在当前作用域中定义，从而捕获重复定义等错误。通过 Errors 类记录和显示编译过程中发现的错误，例如变量重复定义、类型不匹配等。错误信息包括错误的位置（行和列）和描述性消息。详情请见3.5节。

3.3.6 符号表总结

在 "C minus" 编译器中，符号表的管理是实现编译器功能的关键部分。它不仅支持编译器的静态语义分析，还为代码生成阶段提供必要的信息。符号表的设计和实现体现了编译器设计中对于数据结构和算法的精细考虑，确保了编译器的准确性和高效性。

3.4 语义检查

在 "C minus" 编译器中，语义检查是编译过程的一个重要环节，它负责确保源代码中的各种构造在语义上是合法和一致的。以下是对该编译器中语义检查实现的详细分析：

3.4.1 基本原理和目的:

- 语义检查的主要目的是验证源程序的语义正确性，包括标识符的声明与使用、类型的一致性、表达式的有效性等。
- 通过在解析源代码的同时进行语义分析，编译器能够及时发现并报告语义错误。

3.4.2 错误检测与报告:

- 通过 Errors::ErrorAdd 方法，编译器在检测到语义错误时记录错误信息，包括错误的位置（行号和列号）和描述。
- Errors::ErrorsDisplay 方法用于输出所有收集的错误信息，为用户提供问题的具体位置和原因。

3.4.3 符号表的应用:

- 在语义分析过程中，编译器依赖于符号表来验证标识符的声明和作用域。
- 例如，通过符号表，编译器检查变量是否在使用前已经声明，函数调用是否符合函数定义等。

```
int Offset = 12;                //局部变量偏移量初始化,预留12个字节存放返回地址等信息，可根据实际情况修改
```

```
MaxVarSize = 12;            //计算函数变量需要的最大容量
FuncDefPtr = ((FuncSymbol *) SymbolStack.LocateNameCurrent(Name));
if (((BasicTypeAST *) Type)->Type != FuncDefPtr->Type)
    Errors::ErrorAdd(Line, Column, "函数声明和定义的返回类型不同");
if (FuncDefPtr->ParamNum != Params.size())
    Errors::ErrorAdd(Line, Column, "函数声明和定义的参数数目不同");

auto *Local = new SymbolsInAScope();  //生成函数体作用域变量表
FuncDefPtr->ParamPtr = Local;                    //函数符号表项，指向形参
SymbolStack.Symbols.push_back(Local);          //函数体符号表（含形参）进栈
int i = 0;
for (auto a: Params) {
    a->Semantics(Offset);               //未考虑参数用寄存器，只是简单在AR中分配单元
    auto *param = (ParamAST *) ((FuncDefPtr->Params).at(i++));
    if (((BasicTypeAST *) (param->Type))->Type != ((BasicTypeAST *) (a->Type))->Type)
{
        Errors::ErrorAdd(Line, Column, "函数声明和定义的形参类型不一致 ");
        break;
    }
}
Body->LocalSymbolTable = Local;
```

3.4.4 类型检查:

- 编译器执行类型检查以确保变量赋值、表达式计算和函数参数传递的类型一致性。
- 例如，对于二元运算符，编译器检查两个操作数的类型，并确定结果的类型。

```
void AssignAST::Semantics(int &Offset) {
    LeftValExp->Semantics(Offset);
    if (!IsLeftValue(LeftValExp))
        Errors::ErrorAdd(Line, Column, "非左值表达式赋值");
    RightValExp->Semantics(Offset);
    if (LeftValExp->Type == T_VOID || RightValExp->Type == T_VOID)
        Errors::ErrorAdd(Line, Column, "弱类型语言里void类型也不允许赋值");
    Type = LeftValExp->Type;
}
```

3.4.5 函数调用处理:

- 函数调用时，编译器验证函数是否已声明或定义，并检查实参和形参的数量及类型是否匹配。
- 未定义函数的调用会被记录并报告。

```
void FuncCallAST::Semantics(int &Offset) {
    if (FuncRef = (FuncSymbol *) SymbolStack.LocateNameGlobal(Name)) {
        if (FuncRef->Kind != 'F')
            Errors::ErrorAdd(Line, Column, "对非函数名采用函数调用形式 ");
        else if (FuncRef->ParamNum != Params.size())
            Errors::ErrorAdd(Line, Column, "实参表达式个数和形参不一致 ");
        else {
            int i = 0;
            Type = FuncRef->Type;

            for (auto a: Params) {
```

```
                //未考虑参数用寄存器，只是简单在AR中分配单元
                a->Semantics(Offset);
                if (Name != "write") {
                    auto *param = (ParamAST *) ((FuncRef->Params).at(i++));
                    if (((BasicTypeAST *) (param->Type))->Type != a->Type) {
                        Errors::ErrorAdd(Line, Column, "实参表达式类型和形参不一致 ");
                        break;
                    }
                }
            }
            if (FuncRef->Declaration == 1)
                functionCallTable.addFuncCalls(Line, Column, Name);
        }
    } else Errors::ErrorAdd(Line, Column, "引用未定义的函数 " + Name);
}

```

3.4.6 左值和常量检查:

- 对于赋值表达式和自增自减运算，编译器检查左侧表达式是否为有效的左值。
- 对于 case 语句中的条件，编译器检查是否为常量。

```
void UnaryExprAST::Semantics(int &Offset) {
    Exp->Semantics(Offset);
    if (!IsLeftValue(Exp) && (Op != UPLUS && Op != UMINUS && Op != NOT))
        Errors::ErrorAdd(Line, Column, "非左值表达式自增、自减");
    Type = Exp->Type;
}
```

3.4.7 循环和分支语句验证:

- 对于循环（如 while、for）和条件分支（如 if）语句，编译器检查条件表达式的有效性。
- 对于 break 和 continue 语句，编译器验证它们是否在循环或 switch 语句的合法范围内使用。

```

void BreakStmAST::Semantics(int &Offset, int canBreak, int canContinue, int &isReturn,
BasicTypes returnType) {
    if (canBreak == 0)
        Errors::ErrorAdd(Line, Column, "break语句不在循环语句或switch语句中");
}

void ContinueStmAST::Semantics(int &Offset, int canBreak, int canContinue, int
&isReturn, BasicTypes returnType) {
    if (canContinue == 0)
        Errors::ErrorAdd(Line, Column, "continue语句不在循环语句中");
}

```

3.4.8 返回语句检查:

- 对于函数定义，编译器检查是否有符合函数返回类型的 return 语句。
- 对于无返回值（void 类型）的函数，编译器验证是否避免了返回值。

```
void ReturnStmAST::Semantics(int &Offset, int canBreak, int canContinue, int &isReturn,
BasicTypes returnType) {
    if (Exp) Exp->Semantics(Offset);
    if ((returnType == T_VOID && Exp) || (returnType != T_VOID && !Exp) ||
        (returnType != T_VOID && Exp && returnType != Exp->Type))
        Errors::ErrorAdd(Line, Column, "函数返回值类型与函数定义的返回值类型不匹配");
    isReturn = 1;
}
```

3.4.9 语义检查总结

通过上述语义检查机制，"C minus" 编译器能够确保源代码在语义上的正确性和一致性。这些检查不仅提高了编译器的可靠性，还有助于程序员及早发现和修正代码中的错误。语义检查的实现展示了编译器如何将语法分析与深入的语义验证结合起来，体现了编译器设计中对程序语言理论和实现技术的综合运用。

3.5 报错功能

在 "C minus" 编译器中，报错功能是确保编译过程可靠性和用户友好性的关键组成部分。这一功能涉及检测、记录和报告编译过程中遇到的各类错误。以下是对该编译器中报错功能的详细分析：

3.5.1 错误检测机制:

- 编译器在各个阶段（词法分析、语法分析、语义分析等）实施错误检测。
- 错误类型包括语法错误、语义错误（如类型不匹配、未声明的标识符使用）、运行时错误等。

3.5.2 错误信息记录:

- 错误信息通过 Errors 类的静态成员 Errs 进行集中管理。
- ErrorAdd 函数用于在检测到错误时向 Errs 中添加错误记录，包括错误发生的行号、列号和错误消息。

3.5.3 错误报告:

- ErrorsDisplay 函数负责遍历 Errs 并打印所有收集的错误信息。
- 错误报告提供了详细的位置信息和描述性消息，帮助用户定位和理解错误的原因。

3.5.4 错误上下文和定位:

- 编译器利用 Bison 的位置跟踪功能和 Flex 的行号跟踪功能来精确地定位错误发生的位置。
- yyerror 函数被设计用于输出语法错误信息，它使用 Bison 生成的位置信息来指示错误位置。

```
#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
    Errors::ErrorAdd(yylloc.first_line,yylloc.first_column,fmt);
}

```

3.5.5 用户友好的错误信息:

- 错误消息被设计为用户友好和描述性强，使非专业用户也能理解错误的本质。
- 例如，“未声明的标识符”、“类型不匹配”等错误信息直接指向问题的实质，便于用户快速定位问题。

```
第5行、第9列处错误：变量 a 重复定义
第7行、第7列处错误：引用未定义的符号 c
第8行、第8列处错误：对非函数名采用函数调用形式
第9行、第9列处错误：对函数名采用非函数调用形式访问
第10行、第11列处错误：引用未定义的函数 test
第11行、第11列处错误：实参表达式个数和形参不一致
第12行、第10列处错误：break语句不在循环语句或switch语句中
第13行、第13列处错误：continue语句不在循环语句中
第19行、第5列处错误：case中不是常量
第23行、第5列处错误：switch语句的key值相等
第23行、第5列处错误：switch语句的key值相等
第31行、第1列处错误：函数声明和定义的参数数目不同
第30行、第22列处错误：实参表达式个数和形参不一致
第30行、第34列处错误：实参表达式个数和形参不一致
第30行、第35列处错误：函数返回值类型与函数定义的返回值类型不匹配
第24行、第11列处错误：调用的函数未定义
```

3.5.6 报错功能总结

通过以上的报错功能实现，"C minus" 编译器在有效地检测和报告错误的同时，也提供了对用户友好的错误信息。这种设计不仅提高了编译器的鲁棒性，也提升了用户的编码体验，使得调试过程更加高效和直观。报错功能的实现展示了编译器设计中对于用户交互和体验的关注，是编译器可用性的重要体现。

3.6 中间代码生成

"C minus" 编译器中的中间代码生成阶段是编译过程的关键部分，它将源程序的抽象语法树（AST）转换为中间表示形式。这一阶段的目标是生成与平台无关的中间代码，为后续的目标代码生成做准备。以下是对该编译器中中间代码生成功能的详细分析：

3.6.1 基本原理:

中间代码生成阶段遍历抽象语法树，为每个节点生成相应的中间代码。使用四元式（IRCode 类）作为中间表示，每个四元式包含操作符、操作数和结果。中间语言代码的定义如表3-1所示。

表 3-1 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号x	LABEL			x
FUNCTION f:	定义函数f	FUNCTION			f
x := y	赋值操作	ASSIGN	y		x
x := y + z	加法操作	PLUS	y	z	x
x := y - z	减法操作	MINUS	y	z	x
x := y * z	乘法操作	STAR	y	z	x
x := y / z	除法操作	DIV	y	z	x
GOTO x	无条件转移	GOTO			x
IF x [relop] y GOTO z	条件转移	[relop]	x	y	z
RETURN x	返回语句	RETURN			x
ARG x	传实参x	ARG			x
x:=CALL f	调用函数(有返回值)	CALL	f		x
CALL f	调用函数(无返回值)	CALL	f		
PARAM x	函数形参	PARAM			x

3.6.2 临时变量和标签生成:

- 通过 NewTemp 和 NewLabel 函数动态生成临时变量和标签名称，以支持复杂表达式和控制结构的处理。

```
string NewTemp() {
    static int num = 0;
    return "Temp_" + to_string(++num);
}

string NewLabel() {
    static int num = 0;
    return "Label_" + to_string(++num);
}
```

3.6.3 中间代码的类型:

- 中间代码包括各种操作符，如赋值（ASSIGN）、算术运算（PLUS、MINUS、STAR、DIV等）、跳转（GOTO）、条件分支（JLE、JLT、JGE等）。
- 特殊的中间代码表示函数调用（CALL）、参数传递（PARAM、ARG）、返回指令（RETURN）。

```
map<int, string> Instruction = {{LABEL,     "LABEL "},
                                {FUNCTION,  "FUNCTION  "},
                                {ASSIGN,    ":="},
                                {PLUS,      "+"},
                                {UPLUS,     "+"},
                                {MINUS,     "-"},
                                {UMINUS,    "-"},
                                {NOT,       "!"},
                                {DPLUS,     "++"},
                                {DMINUS,    "--"},
                                {PLUSD,     "++"},
                                {MINUSD,    "--"},
                                {STAR,      "*"},
                                {DIV,       "/"},
                                {GOTO,      "  GOTO  "},
```

```
                                {ARRDPLUS,  "++"},
                                {ARRDMINUS, "--"},
                                {ARRPLUSD,  "++"},
                                {ARRMINUSD, "--"},
                                {GT,        ">"},
                                {GE,        ">="},
                                {LT,        "<"},
                                {LE,        "<="},
                                {EQ,        "=="},
                                {NE,        "!="},
                                {JGT,       ">"},
                                {JGE,       ">="},
                                {JLT,       "<"},
                                {JLE,       "<="},
                                {JEQ,       "=="},
                                {JNE,       "!="},
                                {RETURN,    "  RETURN  "},
                                {ARG,       "  ARG  "},
                                {PARAM,     "  PARAM  "}};
```

3.6.4 表达式的处理:

- 对于每个表达式节点，生成相应的中间代码序列。例如，对于二元运算符，生成对应的算术运算四元式。
- 复杂表达式可能需要多个中间代码，并可能涉及临时变量的使用。

```
Opn VarAST::GenIR(int &TempVarOffset) {
    //通过语义检查后，VarRef指向对应表项，否则为空，程序会崩溃
    if (VarRef->Kind == 'V' || VarRef->Kind == 'P') {
        Opn VarOpn(VarRef->Alias, VarRef->Type, VarRef->Offset, VarRef->isGolbal);
        return VarOpn;
    } else if (VarRef->Kind == 'A') {
        auto it = IRCodes.end();
        Opn Result = index[0]->GenIR(TempVarOffset);
        it = IRCodes.end();
        IRCodes.splice(it, index[0]->IRCodes);
        for (int i = 1; i < index.size(); i++) {
            //生成临时变量保存常量值
            Opn DimValue(NewTemp(), T_INT, TempVarOffset + MaxVarSize, 0);
            TempVarOffset += TypeWidth[T_INT];           //修改临时变量的偏移量
            if (TempVarOffset > MaxTempVarOffset)
                MaxTempVarOffset = TempVarOffset;
            Opn Opn1("_CONST", T_INT, 0, 0);             //别名或临时变量名为_CONST时，表示常量
            Opn1.constINT = VarRef->Dims[i];
            IRCodes.emplace_back(ASSIGN, Opn1, Opn(), DimValue);

            Opn MultiResult(NewTemp(), T_INT, TempVarOffset + MaxVarSize, 0); //生成临时变量保存运算结果，结果类型应该根据运算结果来定
            TempVarOffset += TypeWidth[T_INT];           //这里只是简单处理成和左右操作数类型相同，修改临时变量的偏移量
            if (TempVarOffset > MaxTempVarOffset)
                MaxTempVarOffset = TempVarOffset;
            IRCodes.emplace_back(STAR, Result, DimValue, MultiResult);

```

```
            Opn IndexValue = index[i]->GenIR(TempVarOffset);
            it = IRCodes.end();
            IRCodes.splice(it, index[i]->IRCodes);

            Opn AddResult(NewTemp(), T_INT, TempVarOffset + MaxVarSize, 0); //生成临时变量保存运算结果，结果类型应该根据运算结果来定
            TempVarOffset += TypeWidth[T_INT];           //这里只是简单处理成和左右操作数类型相同，修改临时变量的偏移量
            if (TempVarOffset > MaxTempVarOffset)
                MaxTempVarOffset = TempVarOffset;
            IRCodes.emplace_back(PLUS, MultiResult, IndexValue, AddResult);

            Result = AddResult;
        }

        Opn TypeValue(NewTemp(), T_INT, TempVarOffset + MaxVarSize, 0); //生成临时变量保存常量值
        TempVarOffset += TypeWidth[T_INT];           //修改临时变量的偏移量
        if (TempVarOffset > MaxTempVarOffset)
            MaxTempVarOffset = TempVarOffset;
        Opn Opn2("_CONST", T_INT, 0, 0);             //别名或临时变量名为_CONST时，表示常量
        Opn2.constINT = TypeWidth[VarRef->Type];
        IRCodes.emplace_back(ASSIGN, Opn2, Opn(), TypeValue);

        Opn OffsetResult(NewTemp(), T_INT, TempVarOffset + MaxVarSize, 0); //生成临时变量保存运算结果，结果类型应该根据运算结果来定
        TempVarOffset += TypeWidth[T_INT];           //这里只是简单处理成和左右操作数类型相同，修改临时变量的偏移量
        if (TempVarOffset > MaxTempVarOffset)
            MaxTempVarOffset = TempVarOffset;
        IRCodes.emplace_back(STAR, Result, TypeValue, OffsetResult);

        Opn LoadEnd(NewTemp(), VarRef->Type, TempVarOffset + MaxVarSize, 0); //生成临时变量保存常量值
        TempVarOffset += TypeWidth[VarRef->Type];           //修改临时变量的偏移量
        if (TempVarOffset > MaxTempVarOffset)
            MaxTempVarOffset = TempVarOffset;

        Opn Arr(VarRef->Alias, VarRef->Type, VarRef->Offset, VarRef->isGolbal);

        IRCodes.emplace_back(ARRLOAD, Arr, OffsetResult, LoadEnd);

        return LoadEnd;
    }
}
```

3.6.5 函数调用和参数传递:

- 函数调用会生成 CALL 类型的中间代码，其中包括函数名称和返回值的处理。
- 参数传递通过 ARG 类型的中间代码实现，与函数调用紧密关联。

```

Opn FuncCallAST::GenIR(int &TempVarOffset) {
```

```
    list<IRCode> ARGS;
    list<IRCode>::iterator it;
    Opn Opn1, Result;
    SymbolsInAScope *ParamPtr = FuncRef->ParamPtr;
    int i = 0;
    for (auto a: Params) {
        if (Name != string("write"))  //write函数特殊处理，参数传递用的寄存器
        {   //用Opn1的Offset保存形参的偏移量,方便目标代码参数传递,将实参值保存在AR中
            auto *Sym = (VarSymbol *) ((ParamPtr->Symbols).at(i++));
            Opn1.Offset = Sym->Offset;
        }
        Result = a->GenIR(TempVarOffset);         //计算实参表达式的值
        it = IRCodes.end();
        IRCodes.splice(it, a->IRCodes);
        ARGS.emplace_back(ARG, Opn1, Opn(), Result);
    }
    it = IRCodes.end();
    IRCodes.splice(it, ARGS);
    Opn1.Name = Name;
    Opn1.Type = FuncRef->Type;
    Opn1.SymPtr = FuncRef;
    if (FuncRef->Type != T_VOID) {
        //临时变量保存返回结果
        Result = Opn(NewTemp(), FuncRef->Type, TempVarOffset + MaxVarSize, 0);
        TempVarOffset += TypeWidth[FuncRef->Type];
        if (TempVarOffset > MaxTempVarOffset)
            MaxTempVarOffset = TempVarOffset;
        IRCodes.emplace_back(CALL, Opn1, Opn(), Result);
    } else IRCodes.emplace_back(CALL0, Opn1, Opn(), Opn()); //返回值为void
    return Result;
}


void FuncCallAST::GenIR(int &TempVarOffset, string LabelTrue, string LabelFalse) {
    Opn Result = GenIR(TempVarOffset);
    Opn Zero("_CONST", T_INT, 0, 0);
    Zero.constINT = 0;
    IRCodes.emplace_back(JNE, Result, Zero, Opn(LabelTrue, 0, 0, 0));
    IRCodes.emplace_back(GOTO, Opn(), Opn(), Opn(LabelFalse, 0, 0, 0));
}

```

3.6.6 中间代码的显示和输出:

- 通过 DisplayIR 函数输出生成的中间代码，方便调试和验证。
- 中间代码以人类可读的格式显示，展示了每条指令的操作符、操作数和结果。

```
void DisplayIR(const list<IRCode>& IRCodes) {
    for (const auto& a: IRCodes) {
        string OpnStr1, OpnStr2 = a.Opn2.Name, ResultStr = a.Result.Name;
        if (a.Opn1.Name == string("_CONST"))
            switch (a.Opn1.Type) {
                case T_CHAR:
                    OpnStr1 = string("#") + to_string(a.Opn1.constCHAR);
```

```
                    break;
                case T_INT:
                    OpnStr1 = string("#") + to_string(a.Opn1.constINT);
                    break;
                case T_FLOAT:
                    OpnStr1 = string("#") + to_string(a.Opn1.constFLOAT);
                    break;
            }
        else OpnStr1 = a.Opn1.Name;

        switch (a.Op) {
            case ASSIGN:
                cout << "  " << ResultStr << " := " << OpnStr1 << endl;
                break;
            case UPLUS:
            case UMINUS:
            case NOT:
                cout << "  " << ResultStr << " := " << Instruction[a.Op] << " " <<
OpnStr1 << endl;
                break;
            case PLUS:
            case MINUS:
            case STAR:
            case DIV:
            case LE:
            case LT:
            case GE:
            case GT:
            case EQ:
            case NE:
                cout << "  " << ResultStr << ":= " << OpnStr1 << Instruction[a.Op] <<
OpnStr2 << endl;
                break;
            case JLE:
            case JLT:
            case JGE:
            case JGT:
            case JEQ:
            case JNE:
                cout << "  " << "IF " << OpnStr1 << Instruction[a.Op] << OpnStr2 << "
GOTO " << ResultStr << endl;
                break;
            case DPLUS:
            case DMINUS:
                cout << "  " << ResultStr << ":= " << Instruction[a.Op] << OpnStr1 <<
endl;
                break;
            case ARRDPLUS:
            case ARRDMINUS:
                cout << "  " << ResultStr << ":= " << Instruction[a.Op] << OpnStr1 <<
"[" << OpnStr2 << "]" << endl;
                break;
            case PLUSD:
            case MINUSD:
                cout << "  " << ResultStr << ":= " << OpnStr1 << Instruction[a.Op] <<
endl;
```

```
                break;
            case ARRPLUSD:
            case ARRMINUSD:
                cout << "  " << ResultStr << ":= " << OpnStr1 << "[" << OpnStr2 << "]"
<< Instruction[a.Op] << endl;
                break;
            case GOTO:
            case PARAM:
            case ARG:
            case RETURN:
                cout << Instruction[a.Op] << ResultStr << endl;
                break;
            case FUNCTION:
            case LABEL:
                cout << Instruction[a.Op] << ResultStr << ":" << endl;
                break;
            case CALL:
                cout << "  " << ResultStr << " := " << "CALL " << OpnStr1 << endl;
                break;
            case CALL0:
                cout << "  CALL " << OpnStr1 << endl;
                break;
            case ARRLOAD:
                cout << "  " << ResultStr << ":=" << OpnStr1 << "[" << OpnStr2 << "]"
<< endl;
                break;
            case ARRASSIGN:
                cout << "  " << ResultStr << "[" << OpnStr1 << "]" << ":=" << OpnStr2
<< endl;
                break;
            case END:
                cout << "  End Of Program" << endl;
                break;
        }
    }
}

```

3.6.7 中间代码生产总结

通过上述方法，"C minus" 编译器能够有效地将源程序的语法结构转换为中间代码，为后续的优化和目标代码生成打下基础。中间代码生成阶段体现了编译器将抽象语法结构转换为具体指令序列的能力，是编译器设计中的核心部分。

3.7 汇编代码生成

在"C minus"编译器中，汇编代码生成是编译流程的最后阶段，它将前一阶段生成的中间代码转换为目标机器可以执行的汇编代码。以下是针对汇编代码生成阶段的详细分析：

3.7.1 基本过程

汇编代码生成过程涉及将中间代码（IRCode）转换为特定于目标机器的汇编语言指令。在这个示例中，目标机器是基于MIPS架构的，因此生成的汇编代码遵循MIPS指令集。中间代码与MIPS32指令的对应关系如表3-2所示：

表 3-2 中间代码与MIPS32指令对应关系

中间代码	MIPS32指令
x :=#k	li $t1,k sw $t1, x的偏移量($sp)
x := y	lw $t1, y的偏移量($sp) sw $t1, x的偏移量($sp)

x := y + z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	add $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y - z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	sub $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y * z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y / z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul $t3,$t1,$t2
	div $t1,$t2
	mflo $t3
	sw $t3, x的偏移量($sp)

RETURN x	move $v0, x的偏移量($sp) jr $ra

IF x==y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	beq $t1,$t2,z

IF x!=y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	bne $t1,$t2,z

IF x>y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	bgt $t1,$t2,z

IF x>=y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	bge $t1,$t2,z

IF x<y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	blt $t1,$t2,z

IF x<=y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	blt $t1,$t2,z

3.7.2 加载和存储操作

- LoadFromMem 和 StoreToMem 函数负责从内存中加载值到寄存器和从寄存器存储值到内存。
- 这些函数处理全局变量和栈变量的加载和存储操作，使用 *sp*（栈指针）和gp（全局指针）寄存器。

```

string LoadFromMem(const string& Reg1, const Opn& opn, string Reg2) {
    string load;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn.isGolbal == 0)))
        Reg2 = "$gp";
    load = "  lw " + Reg1 + ",  " + to_string(opn.Offset) + "(" + Reg2 + ")";
    return load;
}

string LoadFromMem(const string& Reg1, const Opn& opn1, const Opn& opn2, string Reg2)
{
    string load;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn1.isGolbal == 0)))
        Reg2 = "$gp";
    load = "  lw $t4,  " + to_string(opn2.Offset) + "(" + "$sp" + ")\n" +
           "  add " + Reg2 + ", " + Reg2 + ", " + "$t4\n" +
           "  lw " + Reg1 + ",  " + to_string(opn1.Offset) + "(" + Reg2 + ")\n" +
           "  sub " + Reg2 + ", " + Reg2 + ", $t4";
    return load;
}

string StoreToMem(const string& Reg1, const Opn& opn, string Reg2) {
    string store;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn.isGolbal == 0)))
        Reg2 = "$gp";
    store = "  sw " + Reg1 + ",  " + to_string(opn.Offset) + "(" + Reg2 + ")";
    return store;
}

string StoreToMem(const string& Reg1, const Opn& opn1, const Opn& opn2, string Reg2) {
    string store;
    if (!(Reg2 != "$sp" || (Reg2 == "$sp" && opn1.isGolbal == 0)))
        Reg2 = "$gp";
    store = "  lw $t4,  " + to_string(opn2.Offset) + "(" + "$sp" + ")\n" +
            "  add " + Reg2 + ", " + Reg2 + ", " + "$t4\n" +
            "  sw " + Reg1 + ",  " + to_string(opn1.Offset) + "(" + Reg2 + ")\n" +
            "  sub " + Reg2 + ", " + Reg2 + ", $t4";
    return store;
}
```

3.7.3 寄存器使用

- MIPS架构具有一组固定的寄存器，例如$t0-$t9用于临时存储，$v0用于存放函数返回值，$a0-$a3用于参数传递。
- 编译器在生成汇编代码时，需合理分配这些寄存器。

3.7.4 函数调用和返回

- 对于函数调用，生成jal（Jump And Link）指令，以及相关的参数准备和返回值处理代码。
- 特殊处理像read和write这样的内置函数。

```
/*预先给出read和write的目标代码*/
ObjectFile << "read:\n";
ObjectFile << "  li $v0,4\n";
ObjectFile << "  la $a0,_Prompt\n";
ObjectFile << "  syscall\n";
ObjectFile << "  li $v0,5\n";
ObjectFile << "  syscall\n";
ObjectFile << "  jr $ra\n\n";
ObjectFile << "write:\n";
ObjectFile << "  li $v0,1\n";
ObjectFile << "  syscall\n";
ObjectFile << "  li $v0,4\n";
ObjectFile << "  la $a0,_ret\n";
ObjectFile << "  syscall\n";
ObjectFile << "  move $v0,$0\n";
ObjectFile << "  jr $ra\n";
```

3.7.5 控制流程

- 生成条件跳转指令，如beq、bne等，来实现if-else和循环等控制结构。
- 使用标签（如Label_x）来标识跳转的目标位置。
- ```
  case JLE:
  case JLT:
  case JGE:
  case JGT:
  case JEQ:
  case JNE:
      ObjectFile << LoadFromMem("$t1", it->Opn1, "$sp") << endl;
      ObjectFile << LoadFromMem("$t2", it->Opn2, "$sp") << endl;
      if (it->Op == JLE) ObjectFile << "  ble $t1,$t2," << it->Result.Name << endl;
      else if (it->Op == JLT) ObjectFile << "  blt $t1,$t2," << it->Result.Name <<
  endl;
      else if (it->Op == JGE) ObjectFile << "  bge $t1,$t2," << it->Result.Name <<
  endl;
      else if (it->Op == JGT) ObjectFile << "  bgt $t1,$t2," << it->Result.Name <<
  endl;
      else if (it->Op == JEQ) ObjectFile << "  beq $t1,$t2," << it->Result.Name <<
  endl;
      else ObjectFile << "  bne $t1,$t2," << it->Result.Name << endl;
      break;
  ```

3.7.6 算术和逻辑操作

- 根据中间代码的操作符生成相应的MIPS指令，如add、sub、mul、div等。

```
case PLUS:
case MINUS:
case STAR:
```
```

```
case DIV:
case MOD:
ObjectFile << LoadFromMem("$t1", it->Opn1, "$sp") << endl;
ObjectFile << LoadFromMem("$t2", it->Opn2, "$sp") << endl;
if (it->Op == PLUS) ObjectFile << "  add $t3,$t1,$t2" << endl;
else if (it->Op == MINUS) ObjectFile << "  sub $t3,$t1,$t2" << endl;
else if (it->Op == STAR) ObjectFile << "  mul $t3,$t1,$t2" << endl;
else if (it->Op == DIV) {
    ObjectFile << "  div $t1, $t2" << endl;
    ObjectFile << "  mflo $t3" << endl;
} else {
    ObjectFile << "  div $t1, $t2" << endl;
    ObjectFile << "  mfhi $t3" << endl;
}
ObjectFile << StoreToMem("$t3", it->Result, "$sp") << endl;
break;
```

3.7.7 汇编代码的组织和输出：

- 使用fstream库将生成的汇编代码写入文件，通常是.s后缀的文件。
- 汇编代码以.data部分开始，定义了全局数据，接着是.text部分，包含了所有的指令。

```
ObjectFile << ".data\n";
ObjectFile << "_Prompt: .asciiz \"Enter an integer:  \"\n";
ObjectFile << "_ret: .asciiz \"\\n\"\n";
ObjectFile << ".globl main\n";
ObjectFile << ".text\n\n";
```

3.7.8 汇编代码生产总结

通过以上步骤，"C minus"编译器能够将中间代码有效地转换为可在MIPS架构上运行的汇编代码。这一阶段是编译过程的重要组成部分，它直接影响到生成程序的性能和效率。

4 系统测试与评价

在开发编译器的过程中，系统测试是一个关键环节，它确保了编译器的可靠性和稳定性。以下是对"C minus"编译器进行系统测试和评价的详细分析：

4.1 测试用例

4.1.1 基本测试：

包括变量声明、基本运算、控制结构（如if-else, for, while循环）等。本测试将采用fibo.c文件进行测试，代码如下：

```
int a, b, c;
float m ,n;
int fibo(int a){
    if (a == 1 || a == 2) return 1;
    return fibo(a - 1) + fibo(a - 2);
}

int main(){
    int m, n, i;
    m = read();
    i = 1;
    while (i <= m)
```

```
    {
        n = fibo(i);
        write(n);
        i = i + 1;
    }
    return 1;
}

```

4.1.2 复杂功能测试：

包括多维数组的处理、函数调用（包括递归调用）以及参数传递等。本测试将采用array.c文件进行测试，代码如下：

```
int fibo (int a) { // compute the Fibonacci sequence recursively
    if (a == 1 || a == 2)
        return 1;
    return fibo(a - 1)+fibo(a - 2);
}
int main () {
    int b[5][5];
    int i, j;

    for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
            b[i][j] = i+j;

    for(i = 0; i < 5; i++)
        write(b[i][i]);

    write(fibo(b[1][3]));
    return 0;
}
```

4.1.3 集成测试：

将基本测试和复杂功能测试集成在一起，测试编译器对复杂程序的处理能力。将采用quicksort.c文件进行测试，代码如下：

```
int arr[20];
int QuickSort(int begin, int end) {
    int tmp, i, j, t;
    if(begin > end)
        return 0;
    tmp = arr[begin];
    i = begin;
    j = end;
    while(i != j){
        while(arr[j] >= tmp && j > i)
            j--;
        while(arr[i] <= tmp && j > i)
            i++;
        if(j > i){
            t = arr[i];
            arr[i] = arr[j];
            arr[j] = t;
```

```
        }
    }
    arr[begin] = arr[i];
    arr[i] = tmp;
    QuickSort(begin, i-1);
    QuickSort(i+1, end);
    return 0;
}
int main () {
    int i, n;

    n=read();
    for(i=1;i<=n;i++)
        arr[i]=read();
    QuickSort(1, n);
    for(i=1;i<=n;i++)
        write(arr[i]);
    return 0;
}
```

4.2 正确性测试

4.2.1 构建方法

使用下面的命令进行编译并连接:

```
flex lex.l
bison -d  parser.ypp
gcc -c lex.yy.c
g++ -c ast.cpp semantics.cpp GenIR.cpp GenObject.cpp parser.tab.cpp -w
g++ -o parser ast.o semantics.o GenIR.o GenObject.o parser.tab.o lex.yy.o

```

或者直接使用编写好的脚本:

```
sudo chmod +x ./run.sh
./run.sh
```

运行parser,即可得到结果:

```
 ./parser test.c
```

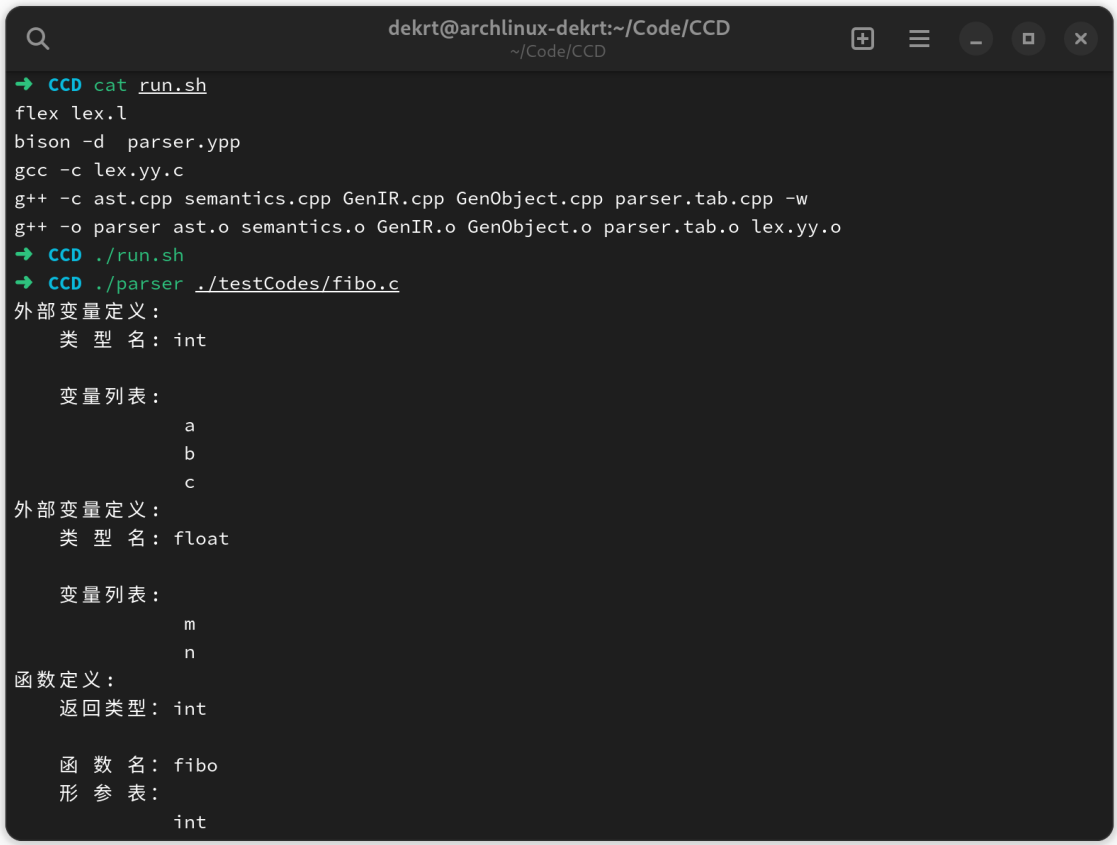


图 4-1 构建项目示意图

4.2.2 符号表与语法树：

4.2.2.1 fibo.c

```
外部变量定义：
    类 型 名：int

    变量列表：
                a
                b
                c
外部变量定义：
    类 型 名：float

    变量列表：
                m
                n
函数定义：
    返回类型：int

    函 数 名：fibo
    形 参 表：
                int
                a
    函 数 体：
            语句部分：
```

```
            if语句:
              条件:
                    ||
                        ==
                            a

                            1


                        ==
                            a

                            2


              if子句:
                    返回表达式:
                        1

            返回表达式:
                +
                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      1


                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      2




函数定义:
    返回类型: int

    函 数 名: main
    形 参 表: 无
    函 数 体:
          说明部分:
            类型: int

            变量列表:
                      m
                      n
                      i

          语句部分:
```

```
            表达式语句:
                赋值运算符: =
                  左值表达式:
                                m
                  右值表达式:
                                函数调用: 函数名: read <无实参表达式>

            表达式语句:
                赋值运算符: =
                  左值表达式:
                                i
                  右值表达式:
                                1

            while语句:
              循环条件:
                    <=
                        i

                        m

              循环体:
                    复合语句:
                      语句部分:
                        表达式语句:
                            赋值运算符: =
                              左值表达式:
                                                n
                              右值表达式:
                                                函数调用: 函数名: fibo
                                                          1个实参表达式:

                                                              i

                        函数调用: 函数名: write
                                  1个实参表达式:

                                        n
                        表达式语句:
                            赋值运算符: =
                              左值表达式:
                                                i
                              右值表达式:
                                                +
                                                    i

                                                    1


            返回表达式:
                1



**********当前复合语句符号表状态**********
```

```
--------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
a                 V_1     int     变量  偏移量: 0  全局: 1
b                 V_2     int     变量  偏移量: 4  全局: 1
c                 V_3     int     变量  偏移量: 8  全局: 1
m                 V_4     float   变量  偏移量: 0  全局: 1
n                 V_5     float   变量  偏移量: 8  全局: 1
fibo                      int     函数  形参数: 1  变量空间: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
a                 V_6     int     形参  偏移量: 12  全局: 0
--------------------------------------------------------------------------



**********当前复合语句符号表状态**********
--------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
a                 V_1     int     变量  偏移量: 0  全局: 1
b                 V_2     int     变量  偏移量: 4  全局: 1
c                 V_3     int     变量  偏移量: 8  全局: 1
m                 V_4     float   变量  偏移量: 0  全局: 1
n                 V_5     float   变量  偏移量: 8  全局: 1
fibo                      int     函数  形参数: 1  变量空间: 16
main                      int     函数  形参数: 0  变量空间: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
m                 V_7     int     变量  偏移量: 12  全局: 0
n                 V_8     int     变量  偏移量: 16  全局: 0
i                 V_9     int     变量  偏移量: 20  全局: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 2
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
  空 表
--------------------------------------------------------------------------


```

```
191
192
193 ********************当前复合语句符号表状态************************
194 ----------------------------------------------------------------------
195  层号: 0
196  符 号 名            别名     类型     种 类    其它信息
197 ----------------------------------------------------------------------
198 read                          int      函数   形参数: 0   变量空间: 12
199 write                         void     函数   形参数: 1   变量空间: 4
200 x                    x        int      形参   偏移量: 4   全局: 0
201 a                    V_1      int      变量   偏移量: 0   全局: 1
202 b                    V_2      int      变量   偏移量: 4   全局: 1
203 c                    V_3      int      变量   偏移量: 8   全局: 1
204 m                    V_4      float    变量   偏移量: 0   全局: 1
205 n                    V_5      float    变量   偏移量: 8   全局: 1
206 fibo                        int      函数   形参数: 1   变量空间: 16
207 main                        int      函数   形参数: 0   变量空间: 0
208 ----------------------------------------------------------------------
209 ----------------------------------------------------------------------
210  层号: 1
211  符 号 名            别名     类型     种 类    其它信息
212 ----------------------------------------------------------------------
213 m                    V_7      int      变量   偏移量: 12  全局: 0
214 n                    V_8      int      变量   偏移量: 16  全局: 0
215 i                    V_9      int      变量   偏移量: 20  全局: 0
216 ----------------------------------------------------------------------
217
218
219 ----------------------------------------------------------------------
220  层号: 0
221  符 号 名            别名     类型     种 类    其它信息
222 ----------------------------------------------------------------------
223 read                          int      函数   形参数: 0   变量空间: 12
224 write                         void     函数   形参数: 1   变量空间: 4
225 x                    x        int      形参   偏移量: 4   全局: 0
226 a                    V_1      int      变量   偏移量: 0   全局: 1
227 b                    V_2      int      变量   偏移量: 4   全局: 1
228 c                    V_3      int      变量   偏移量: 8   全局: 1
229 m                    V_4      float    变量   偏移量: 0   全局: 1
230 n                    V_5      float    变量   偏移量: 8   全局: 1
231 fibo                        int      函数   形参数: 1   变量空间: 16
232 main                        int      函数   形参数: 0   变量空间: 24
233 ----------------------------------------------------------------------
234
```

4.2.2.2 array.c

```
1 函数定义:
2     返回类型: int
3
4     函 数 名: fibo
5     形 参 表:
6               int
7               a
8     函 数 体:
9           语句部分:
```

```
            if语句:
              条件:
                    ||
                        ==
                            a

                            1


                        ==
                            a

                            2


              if子句:
                    返回表达式:
                        1

            返回表达式:
                +
                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      1


                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      2




函数定义:
    返回类型: int

    函 数 名: main
    形 参 表: 无
    函 数 体:
          说明部分:
            类型: int

            变量列表:
                      b[5][5]

            类型: int

            变量列表:
```

```
                    i
                    j

        语句部分：
          for语句：
            单次表达式：
                  赋值运算符：=
                    左值表达式：
                                i
                    右值表达式：
                                0
            循环条件：
                  <
                      i

                      5

            末尾循环体：
                  单目：N++
                        i
            循环体：
                  for语句：
                    单次表达式：
                          赋值运算符：=
                            左值表达式：
                                        j
                            右值表达式：
                                        0
                    循环条件：
                          <
                              j

                              5

                    末尾循环体：
                          单目：N++
                                j
                    循环体：
                          表达式语句：
                              赋值运算符：=
                                左值表达式：
                                            b
                                                    2个下标：

                                                        i
                                                        j
                                右值表达式：
                                            +
                                                i

                                                j


          for语句：
            单次表达式：
                  赋值运算符：=
```

```
                    左值表达式:
                                i
                    右值表达式:
                                0
            循环条件:
                  <
                      i

                      5

            末尾循环体:
                  单目: N++
                          i
            循环体:
                  函数调用: 函数名: write
                            1个实参表达式:

                                b
                                        2个下标:

                                                i
                                                i
          函数调用: 函数名: write
                    1个实参表达式:

                        函数调用: 函数名: fibo
                                  1个实参表达式:

                                      b
                                                2个下标:

                                                      1
                                                      3
          返回表达式:
              0



**********当前复合语句符号表状态**********
--------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------
read                      int     函数   形参数: 0  变量空间: 12
write                     void    函数   形参数: 1  变量空间: 4
x                 x       int     形参   偏移量: 4  全局: 0
fibo                      int     函数   形参数: 1  变量空间: 0
--------------------------------------------------------------------
--------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------
a                 V_1     int     形参   偏移量: 12  全局: 0
--------------------------------------------------------------------

```

```
**********当前复合语句符号表状态**********
--------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
fibo                      int     函数  形参数: 1  变量空间: 16
main                      int     函数  形参数: 0  变量空间: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
b                 V_2     int     数组  偏移量: 12  全局: 0  2维: 5 5
i                 V_3     int     变量  偏移量: 112  全局: 0
j                 V_4     int     变量  偏移量: 116  全局: 0
--------------------------------------------------------------------------


--------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
fibo                      int     函数  形参数: 1  变量空间: 16
main                      int     函数  形参数: 0  变量空间: 120
--------------------------------------------------------------------------
```

4.2.2.3 quicksort.c

```
外部变量定义:
    类 型 名: int

    变量列表:
                arr[20]
函数定义:
    返回类型: int

    函 数 名: QuickSort
    形 参 表:
              int
              begin
              int
              end
    函 数 体:
          说明部分:
            类型: int

```

```
        变量列表:
                tmp
                i
                j
                t

    语句部分:
        if语句:
          条件:
                >
                    begin

                    end

          if子句:
                返回表达式:
                    0

        表达式语句:
            赋值运算符: =
              左值表达式:
                            tmp
              右值表达式:
                            arr
                                      1个下标:

                                            begin

        表达式语句:
            赋值运算符: =
              左值表达式:
                            i
              右值表达式:
                            begin

        表达式语句:
            赋值运算符: =
              左值表达式:
                            j
              右值表达式:
                            end

        while语句:
          循环条件:
                !=
                    i

                    j

          循环体:
                复合语句:
                  语句部分:
                    while语句:
                      循环条件:
                            &&
                                >=
```

```
                        arr
                                1个下标:

                                    j

                        tmp


                    >
                        j

                        i


    循环体:
        表达式语句:
            单目: N--
                j

while语句:
    循环条件:
        &&
            <=
                arr
                                1个下标:

                                    i

                        tmp


                    >
                        j

                        i


    循环体:
        表达式语句:
            单目: N++
                i

if语句:
    条件:
        >
            j

            i

    if子句:
        复合语句:
          语句部分:
            表达式语句:
                赋值运算符: =
                  左值表达式:
                                    t
```

```
                                右值表达式:
                                        arr
                                            1个下标:

                                                i

                            表达式语句:
                                赋值运算符: =
                                  左值表达式:
                                        arr
                                            1个下标:

                                                i
                                  右值表达式:
                                        arr
                                            1个下标:

                                                j

                            表达式语句:
                                赋值运算符: =
                                  左值表达式:
                                        arr
                                            1个下标:

                                                j
                                  右值表达式:
                                        t

        表达式语句:
            赋值运算符: =
              左值表达式:
                        arr
                            1个下标:

                                    begin
              右值表达式:
                        arr
                            1个下标:

                                    i

        表达式语句:
            赋值运算符: =
              左值表达式:
                        arr
                            1个下标:

                                    i
              右值表达式:
                        tmp

        函数调用: 函数名: QuickSort
                  2个实参表达式:

                      begin
```

```
                        -
                            i

                            1

            函数调用：函数名：QuickSort
                      2个实参表达式：

                        +
                            i

                            1

                        end
            返回表达式：
                0


函数定义：
    返回类型：int

    函 数 名：main
    形 参 表：无
    函 数 体：
          说明部分：
            类型：int

            变量列表：
                      i
                      n

          语句部分：
            表达式语句：
                赋值运算符：=
                  左值表达式：
                                n
                  右值表达式：
                                函数调用：函数名：read <无实参表达式>

            for语句：
              单次表达式：
                    赋值运算符：=
                      左值表达式：
                                    i
                      右值表达式：
                                    1
              循环条件：
                    <=
                        i

                        n

              末尾循环体：
                    单目：N++
                            i
              循环体：
```

```
                    表达式语句:
                        赋值运算符: =
                          左值表达式:
                                              arr
                                                        1个下标:

                                                              i
                          右值表达式:
                                              函数调用: 函数名: read <无实参表达式>

            函数调用: 函数名: QuickSort
                       2个实参表达式:

                          1
                          n
            for语句:
              单次表达式:
                       赋值运算符: =
                         左值表达式:
                                          i
                         右值表达式:
                                          1
              循环条件:
                       <=
                           i

                           n

              末尾循环体:
                       单目: N++
                              i
              循环体:
                       函数调用: 函数名: write
                                 1个实参表达式:

                                     arr
                                               1个下标:

                                                        i
            返回表达式:
                0



**********当前复合语句符号表状态**********
--------------------------------------------------------------------
 层号: 0
 符 号 名          别名     类型     种 类    其它信息
--------------------------------------------------------------------
read                       int     函数   形参数: 0  变量空间: 12
write                      void    函数   形参数: 1  变量空间: 4
x                  x       int     形参   偏移量: 4  全局: 0
arr                V_1     int     数组   偏移量: 0  全局: 1  1维: 20
QuickSort                  int     函数   形参数: 2  变量空间: 0
--------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
begin              V_2     int     形参  偏移量: 12  全局: 0
end                V_3     int     形参  偏移量: 16  全局: 0
tmp                V_4     int     变量  偏移量: 20  全局: 0
i                  V_5     int     变量  偏移量: 24  全局: 0
j                  V_6     int     变量  偏移量: 28  全局: 0
t                  V_7     int     变量  偏移量: 32  全局: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 2
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
  空 表
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 3
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
  空 表
--------------------------------------------------------------------------



**********当前复合语句符号表状态**********
--------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
read                       int     函数  形参数: 0  变量空间: 12
write                      void    函数  形参数: 1  变量空间: 4
x                  x       int     形参  偏移量: 4  全局: 0
arr                V_1     int     数组  偏移量: 0  全局: 1  1维: 20
QuickSort                  int     函数  形参数: 2  变量空间: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
begin              V_2     int     形参  偏移量: 12  全局: 0
end                V_3     int     形参  偏移量: 16  全局: 0
tmp                V_4     int     变量  偏移量: 20  全局: 0
i                  V_5     int     变量  偏移量: 24  全局: 0
j                  V_6     int     变量  偏移量: 28  全局: 0
t                  V_7     int     变量  偏移量: 32  全局: 0
--------------------------------------------------------------------------
--------------------------------------------------------------------------
 层号: 2
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------
  空 表
--------------------------------------------------------------------------

```

```
**********当前复合语句符号表状态**********
--------------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
arr               V_1     int     数组  偏移量: 0  全局: 1  1维: 20
QuickSort                 int     函数  形参数: 2  变量空间: 0
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------------
begin             V_2     int     形参  偏移量: 12  全局: 0
end               V_3     int     形参  偏移量: 16  全局: 0
tmp               V_4     int     变量  偏移量: 20  全局: 0
i                 V_5     int     变量  偏移量: 24  全局: 0
j                 V_6     int     变量  偏移量: 28  全局: 0
t                 V_7     int     变量  偏移量: 32  全局: 0
--------------------------------------------------------------------------------



**********当前复合语句符号表状态**********
--------------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
x                 x       int     形参  偏移量: 4  全局: 0
arr               V_1     int     数组  偏移量: 0  全局: 1  1维: 20
QuickSort                 int     函数  形参数: 2  变量空间: 36
main                      int     函数  形参数: 0  变量空间: 0
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
 层号: 1
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------------
i                 V_8     int     变量  偏移量: 12  全局: 0
n                 V_9     int     变量  偏移量: 16  全局: 0
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
 层号: 0
 符 号 名          别名    类型    种 类   其它信息
--------------------------------------------------------------------------------
read                      int     函数  形参数: 0  变量空间: 12
write                     void    函数  形参数: 1  变量空间: 4
```

```
x                   x       int     形参   偏移量: 4   全局: 0
arr                 V_1     int     数组   偏移量: 0   全局: 1   1维: 20
QuickSort                   int     函数   形参数: 2   变量空间: 36
main                        int     函数   形参数: 0   变量空间: 20
-------------------------------------------------------------------
```

4.2.3 中间代码生成:

检查中间代码是否正确表示了源代码的逻辑。

4.2.3.1 fibo.ir

```
FUNCTION  fibo:
  PARAM  a
  Temp_1 := #1
  IF V_6==Temp_1 GOTO Label_1
  GOTO  Label_3
LABEL Label_3:
  Temp_2 := #2
  IF V_6==Temp_2 GOTO Label_1
  GOTO  Label_2
LABEL Label_1:
  Temp_3 := #1
  RETURN  Temp_3
LABEL Label_2:
  Temp_4 := #1
  Temp_5:= V_6-Temp_4
  ARG  Temp_5
  Temp_6 := CALL fibo
  Temp_7 := #2
  Temp_8:= V_6-Temp_7
  ARG  Temp_8
  Temp_9 := CALL fibo
  Temp_10:= Temp_6+Temp_9
  RETURN  Temp_10
FUNCTION  main:
  Temp_11 := CALL read
  V_7 := Temp_11
  Temp_12 := #1
  V_9 := Temp_12
LABEL Label_4:
  IF V_9<=V_7 GOTO Label_5
  GOTO  Label_6
LABEL Label_5:
  ARG  V_9
  Temp_13 := CALL fibo
  V_8 := Temp_13
  ARG  V_8
  CALL write
  Temp_14 := #1
  Temp_15:= V_9+Temp_14
  V_9 := Temp_15
  GOTO  Label_4
LABEL Label_6:
  Temp_16 := #1
  RETURN  Temp_16
```

```
  End Of Program
```

4.2.3.2 array.ir

```
FUNCTION  fibo:
  PARAM  a
  Temp_1 := #1
  IF V_1==Temp_1 GOTO Label_1
  GOTO  Label_3
LABEL Label_3:
  Temp_2 := #2
  IF V_1==Temp_2 GOTO Label_1
  GOTO  Label_2
LABEL Label_1:
  Temp_3 := #1
  RETURN  Temp_3
LABEL Label_2:
  Temp_4 := #1
  Temp_5:= V_1-Temp_4
  ARG  Temp_5
  Temp_6 := CALL fibo
  Temp_7 := #2
  Temp_8:= V_1-Temp_7
  ARG  Temp_8
  Temp_9 := CALL fibo
  Temp_10:= Temp_6+Temp_9
  RETURN  Temp_10
FUNCTION  main:
  Temp_11 := #0
  V_3 := Temp_11
LABEL Label_4:
  Temp_12 := #5
  IF V_3<Temp_12 GOTO Label_5
  GOTO  Label_6
LABEL Label_5:
  Temp_14 := #0
  V_4 := Temp_14
LABEL Label_8:
  Temp_15 := #5
  IF V_4<Temp_15 GOTO Label_9
  GOTO  Label_10
LABEL Label_9:
  Temp_17 := #5
  Temp_18:= V_3*Temp_17
  Temp_19:= Temp_18+V_4
  Temp_20 := #4
  Temp_21:= Temp_19*Temp_20
  Temp_22:= V_3+V_4
  V_2[Temp_21]:=Temp_22
LABEL Label_11:
  Temp_16:= V_4++
  GOTO  Label_8
LABEL Label_10:
LABEL Label_7:
```

```
  Temp_13:= V_3++
  GOTO  Label_4
LABEL Label_6:
  Temp_23 := #0
  V_3 := Temp_23
LABEL Label_12:
  Temp_24 := #5
  IF V_3<Temp_24 GOTO Label_13
  GOTO  Label_14
LABEL Label_13:
  Temp_26 := #5
  Temp_27:= V_3*Temp_26
  Temp_28:= Temp_27+V_3
  Temp_29 := #4
  Temp_30:= Temp_28*Temp_29
  Temp_31:=V_2[Temp_30]
  ARG  Temp_31
  CALL write
LABEL Label_15:
  Temp_25:= V_3++
  GOTO  Label_12
LABEL Label_14:
  Temp_32 := #1
  Temp_33 := #5
  Temp_34:= Temp_32*Temp_33
  Temp_35 := #3
  Temp_36:= Temp_34+Temp_35
  Temp_37 := #4
  Temp_38:= Temp_36*Temp_37
  Temp_39:=V_2[Temp_38]
  ARG  Temp_39
  Temp_40 := CALL fibo
  ARG  Temp_40
  CALL write
  Temp_41 := #0
  RETURN  Temp_41
  End Of Program
```

4.2.3.3 quicksort.ir

```
FUNCTION  QuickSort:
  PARAM  begin
  PARAM  end
  IF V_2>V_3 GOTO Label_1
  GOTO  Label_2
LABEL Label_1:
  Temp_1 := #0
  RETURN  Temp_1
LABEL Label_2:
  Temp_2 := #4
  Temp_3:= V_2*Temp_2
  Temp_4:=V_1[Temp_3]
  V_4 := Temp_4
  V_5 := V_2
  V_6 := V_3
LABEL Label_3:
```

```
  IF V_5!=V_6 GOTO Label_4
  GOTO  Label_5
LABEL Label_4:
LABEL Label_6:
  Temp_5 := #4
  Temp_6:= V_6*Temp_5
  Temp_7:=V_1[Temp_6]
  IF Temp_7>=V_4 GOTO Label_9
  GOTO  Label_8
LABEL Label_9:
  IF V_6>V_5 GOTO Label_7
  GOTO  Label_8
LABEL Label_7:
  Temp_8:= V_6--
  GOTO  Label_6
LABEL Label_8:
LABEL Label_10:
  Temp_9 := #4
  Temp_10:= V_5*Temp_9
  Temp_11:=V_1[Temp_10]
  IF Temp_11<=V_4 GOTO Label_13
  GOTO  Label_12
LABEL Label_13:
  IF V_6>V_5 GOTO Label_11
  GOTO  Label_12
LABEL Label_11:
  Temp_12:= V_5++
  GOTO  Label_10
LABEL Label_12:
  IF V_6>V_5 GOTO Label_14
  GOTO  Label_15
LABEL Label_14:
  Temp_13 := #4
  Temp_14:= V_5*Temp_13
  Temp_15:=V_1[Temp_14]
  V_7 := Temp_15
  Temp_16 := #4
  Temp_17:= V_5*Temp_16
  Temp_18 := #4
  Temp_19:= V_6*Temp_18
  Temp_20:=V_1[Temp_19]
  V_1[Temp_17]:=Temp_20
  Temp_21 := #4
  Temp_22:= V_6*Temp_21
  V_1[Temp_22]:=V_7
LABEL Label_15:
  GOTO  Label_3
LABEL Label_5:
  Temp_23 := #4
  Temp_24:= V_2*Temp_23
  Temp_25 := #4
  Temp_26:= V_5*Temp_25
  Temp_27:=V_1[Temp_26]
  V_1[Temp_24]:=Temp_27
  Temp_28 := #4
  Temp_29:= V_5*Temp_28
```

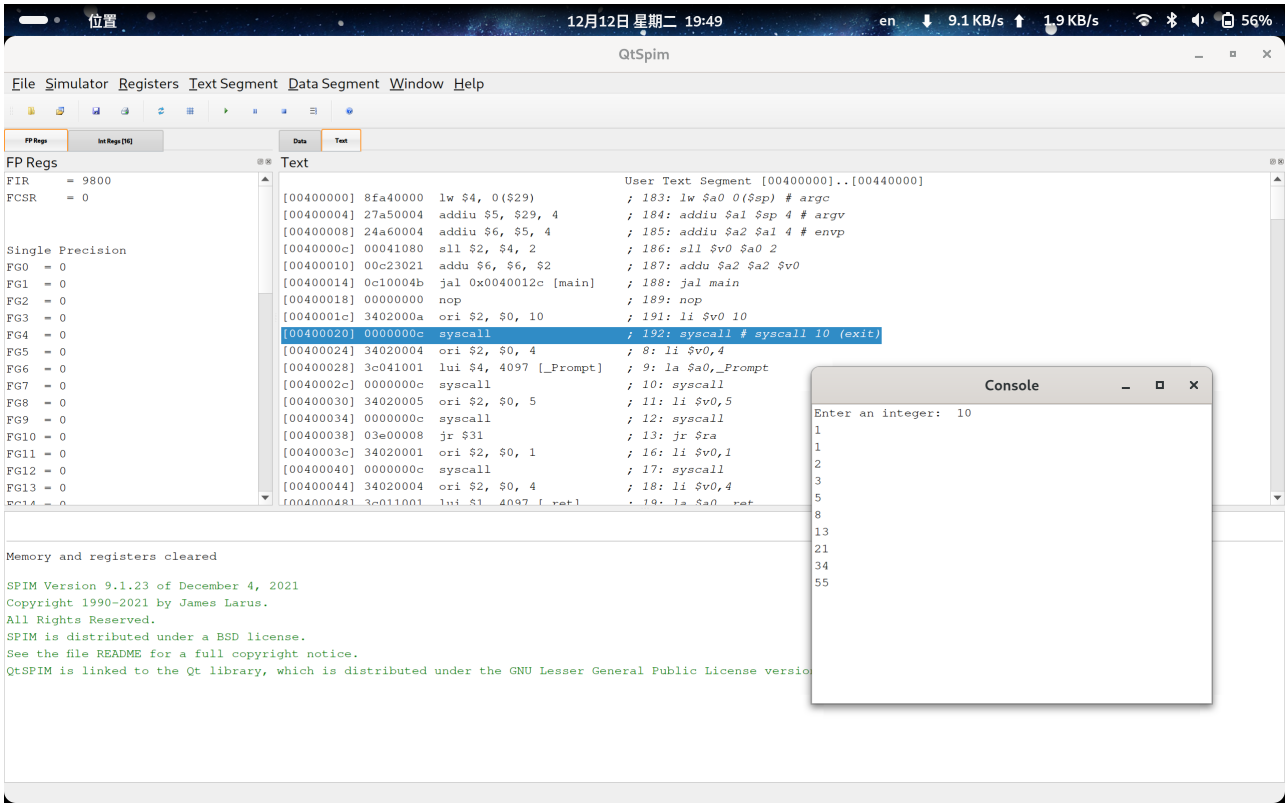
```
  V_1[Temp_29]:=V_4
  Temp_30 := #1
  Temp_31:= V_5-Temp_30
  ARG  V_2
  ARG  Temp_31
  Temp_32 := CALL QuickSort
  Temp_33 := #1
  Temp_34:= V_5+Temp_33
  ARG  Temp_34
  ARG  V_3
  Temp_35 := CALL QuickSort
  Temp_36 := #0
  RETURN  Temp_36
FUNCTION  main:
  Temp_37 := CALL read
  V_9 := Temp_37
  Temp_38 := #1
  V_8 := Temp_38
LABEL Label_16:
  IF V_8<=V_9 GOTO Label_17
  GOTO  Label_18
LABEL Label_17:
  Temp_40 := #4
  Temp_41:= V_8*Temp_40
  Temp_42 := CALL read
  V_1[Temp_41]:=Temp_42
LABEL Label_19:
  Temp_39:= V_8++
  GOTO  Label_16
LABEL Label_18:
  Temp_43 := #1
  ARG  Temp_43
  ARG  V_9
  Temp_44 := CALL QuickSort
  Temp_45 := #1
  V_8 := Temp_45
LABEL Label_20:
  IF V_8<=V_9 GOTO Label_21
  GOTO  Label_22
LABEL Label_21:
  Temp_47 := #4
  Temp_48:= V_8*Temp_47
  Temp_49:=V_1[Temp_48]
  ARG  Temp_49
  CALL write
LABEL Label_23:
  Temp_46:= V_8++
  GOTO  Label_20
LABEL Label_22:
  Temp_50 := #0
  RETURN  Temp_50
  End Of Program

```

4.2.4 目标代码的执行：

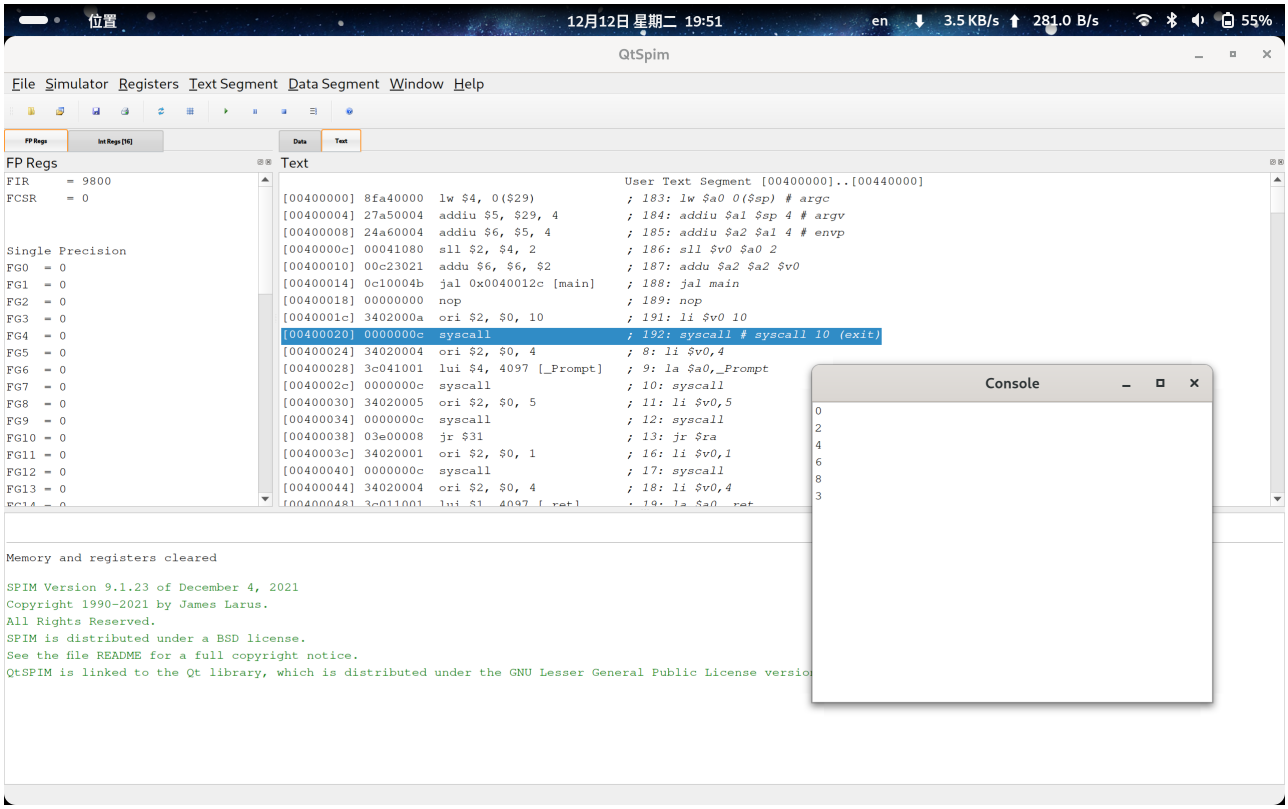
运行目标代码，验证程序输出是否符合预期。本部分将使用qtspim运行生成的MIPS代码：

4.2.4.1 fibo.c



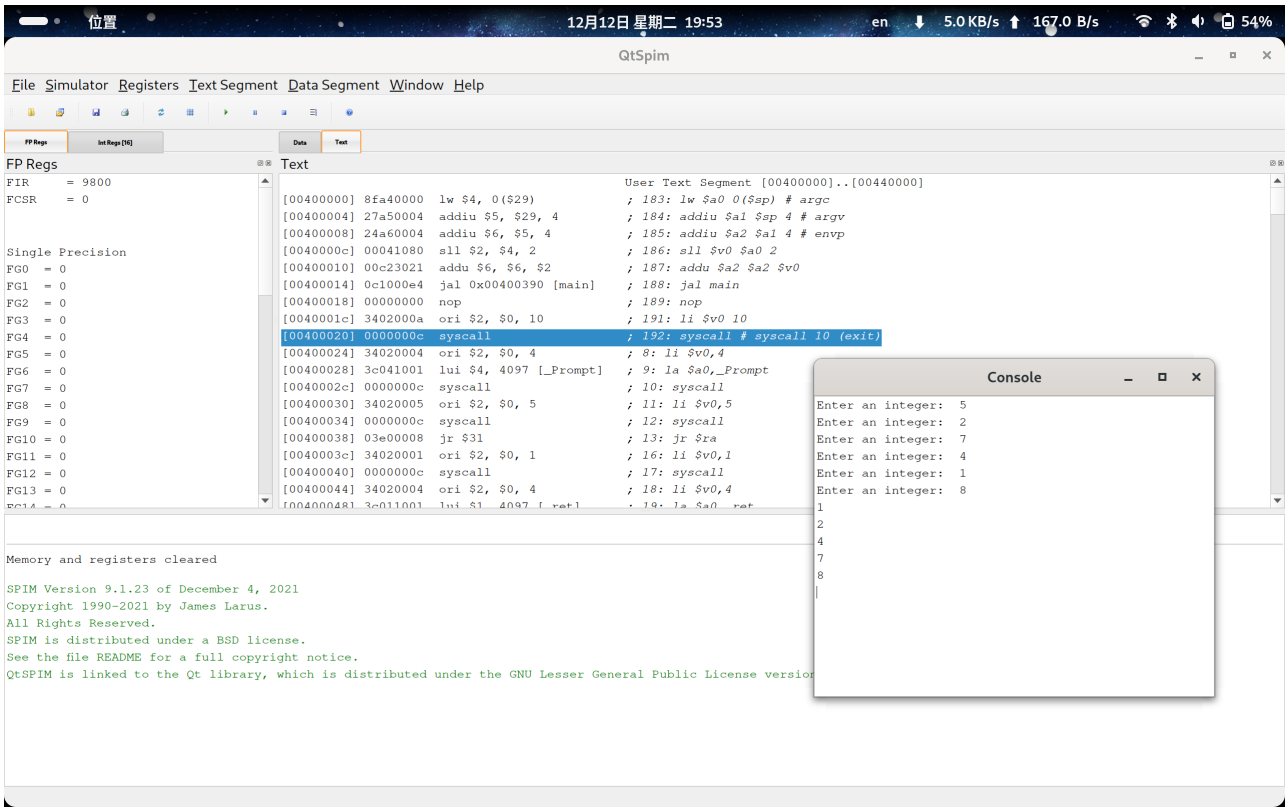
可以看到，程序正确打印除了斐波那契数组的前十项，通过了测试。

4.2.4.2 array.c



可以看到，程序正确正确打印了多维数组b中对角线的元素，成功验证了多维数组的正确性。

4.2.4.3 quicksort.c



可以看到，当输入了五个元素的一位数组之后，程序可以正确调用快速排序，使得原数组有序递增，完成了集成测试。

4.3 报错功能测试

4.3.1 概述

1. **语法错误检测**：输入具有语法错误的代码，验证编译器是否能够准确地识别并报告错误。

2. **语义错误检测**：输入违反语义规则的代码（如类型不匹配，未声明变量等），检查编译器的错误报告功能。

4.3.2 测试代码 error.c

```c
int fibo(int a, int b);
int jump(int a);
int main () {
    int a;
    int a;
    int b;
    c = 0;
    a(4);
    fibo=4;
    test(4);
    fibo(3);
    break;
    continue;
    switch (a)
    {
    case b:
        /* code */
```

```
        break;
    case 3:
    case 3:
    default:
        break;
    }
    jump(4);
    return 0;
}
int fibo (int a) { // compute the Fibonacci sequence recursively
    if (a == 1 || a == 2)
        return 1;
    return fibo(a - 1)+fibo(a - 2);
}
```

4.3.3 运行结果

```
函数定义:
    返回类型: int

    函 数 名: fibo
    形 参 表:
              int
              a
              int
              b

函数定义:
    返回类型: int

    函 数 名: jump
    形 参 表:
              int
              a

函数定义:
    返回类型: int

    函 数 名: main
    形 参 表: 无
    函 数 体:
          说明部分:
            类型: int

            变量列表:
                      a

            类型: int

            变量列表:
                      a

            类型: int

            变量列表:
                      b
```

```
40
41          语句部分:
42            表达式语句:
43                赋值运算符: =
44                  左值表达式:
45                                  c
46                  右值表达式:
47                                  0
48
49            函数调用: 函数名: a
50                      1个实参表达式:
51
52                          4
53            表达式语句:
54                赋值运算符: =
55                  左值表达式:
56                                  fibo
57                  右值表达式:
58                                  4
59
60            函数调用: 函数名: test
61                      1个实参表达式:
62
63                          4
64            函数调用: 函数名: fibo
65                      1个实参表达式:
66
67                          3
68            break语句
69            continue语句
70            表达式:
71                a
72            Case部分:
73                常量Key:
74                    b
75                语句部分:
76                    break语句
77                常量Key:
78                    3
79                常量Key:
80                    3
81            Default部分:
82                break语句
83            函数调用: 函数名: jump
84                      1个实参表达式:
85
86                          4
87            返回表达式:
88                0
89
90
91  函数定义:
92      返回类型: int
93
94      函 数 名: fibo
95      形 参 表:
```

```
            int
            a
    函 数 体:
          语句部分:
            if语句:
              条件:
                    ||
                        ==
                            a

                            1


                        ==
                            a

                            2


              if子句:
                    返回表达式:
                        1

            返回表达式:
                +
                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      1


                    函数调用: 函数名: fibo
                              1个实参表达式:

                                  -
                                      a

                                      2
```

```
********************当前复合语句符号表状态*************************
--------------------------------------------------------------------
 层号: 0
 符 号 名          别名     类型     种 类    其它信息
--------------------------------------------------------------------
read                       int      函数   形参数: 0  变量空间: 12
write                      void     函数   形参数: 1  变量空间: 4
x                  x       int      形参   偏移量: 4  全局: 0
fibo                       int      函数   形参数: 2  变量空间: 12
```

```
jump                        int     函数   形参数: 1   变量空间: 12
main                        int     函数   形参数: 0   变量空间: 0
------------------------------------------------------------------------
------------------------------------------------------------------------
 层号: 1
 符 号 名           别名     类型     种 类    其它信息
------------------------------------------------------------------------
a                   V_1     int     变量   偏移量: 12   全局: 0
b                   V_2     int     变量   偏移量: 16   全局: 0
------------------------------------------------------------------------



**********当前复合语句符号表状态**********
------------------------------------------------------------------------
 层号: 0
 符 号 名           别名     类型     种 类    其它信息
------------------------------------------------------------------------
read                        int     函数   形参数: 0   变量空间: 12
write                       void    函数   形参数: 1   变量空间: 4
x                   x       int     形参   偏移量: 4   全局: 0
fibo                        int     函数   形参数: 2   变量空间: 12
jump                        int     函数   形参数: 1   变量空间: 12
main                        int     函数   形参数: 0   变量空间: 20
------------------------------------------------------------------------
------------------------------------------------------------------------
 层号: 1
 符 号 名           别名     类型     种 类    其它信息
------------------------------------------------------------------------
a                   V_3     int     形参   偏移量: 12   全局: 0
------------------------------------------------------------------------



------------------------------------------------------------------------
 层号: 0
 符 号 名           别名     类型     种 类    其它信息
------------------------------------------------------------------------
read                        int     函数   形参数: 0   变量空间: 12
write                       void    函数   形参数: 1   变量空间: 4
x                   x       int     形参   偏移量: 4   全局: 0
fibo                        int     函数   形参数: 2   变量空间: 16
jump                        int     函数   形参数: 1   变量空间: 12
main                        int     函数   形参数: 0   变量空间: 20
------------------------------------------------------------------------
第5行、第9列处错误: 变量 a 重复定义
第7行、第7列处错误: 引用未定义的符号 c
第8行、第8列处错误: 对非函数名采用函数调用形式
第9行、第9列处错误: 对函数名采用非函数调用形式访问
第10行、第11列处错误: 引用未定义的函数 test
第11行、第11列处错误: 实参表达式个数和形参不一致
第12行、第10列处错误: break语句不在循环语句或switch语句中
第13行、第13列处错误: continue语句不在循环语句中
第19行、第5列处错误: case中不是常量
第23行、第5列处错误: switch语句的key值相等
第23行、第5列处错误: switch语句的key值相等
```

```
第31行、第1列处错误：函数声明和定义的参数数目不同
第30行、第22列处错误：实参表达式个数和形参不一致
第30行、第34列处错误：实参表达式个数和形参不一致
第30行、第35列处错误：函数返回值类型与函数定义的返回值类型不匹配
第24行、第11列处错误：调用的函数未定义

```

4.4 系统的优点

1. **强大的错误检测机制**：编译器具有有效的错误检测和报告机制，能够及时发现语法和语义错误。
2. **高效的中间代码生成**：中间代码生成步骤简洁高效，有助于优化最终的目标代码。
3. **目标代码优化**：生成的MIPS汇编代码紧凑高效，适用于资源受限的系统。
4. **模块化设计**：编译器采用模块化设计，便于维护和扩展。

4.5 系统的缺点

1. **性能优化有限**：在某些情况下，编译器生成的代码可能不是最优的，尤其是在复杂表达式的处理上。
2. **资源消耗**：对于大型程序，编译器可能会消耗较多的内存和CPU资源。
3. **用户界面**：缺乏一个友好的用户界面，对于非专业用户来说可能不太友好。
4. **错误提示不够详细**：在某些情况下，编译器的错误提示可能不够详细，使得用户难以快速定位问题所在。

5 实验中遇到的问题及解决

5.1 词法分析器

大体上参照实验指导书后面的部分，增添的部分如下：

5.1.1 注释的识别

```
Oneline_comment     "//"[^\n]*
Multiline_comment   "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/"

...

{Oneline_comment}           {}      //单行注释
{Multiline_comment}         {}      //多行注释
```

5.1.2 关键字的增添

```
"break"     {return BREAK;}
"continue"  {return CONTINUE;}

"switch"    {return SWITCH;}
"case"      {return CASE;}
"default"   {return DEFAULT;}
```

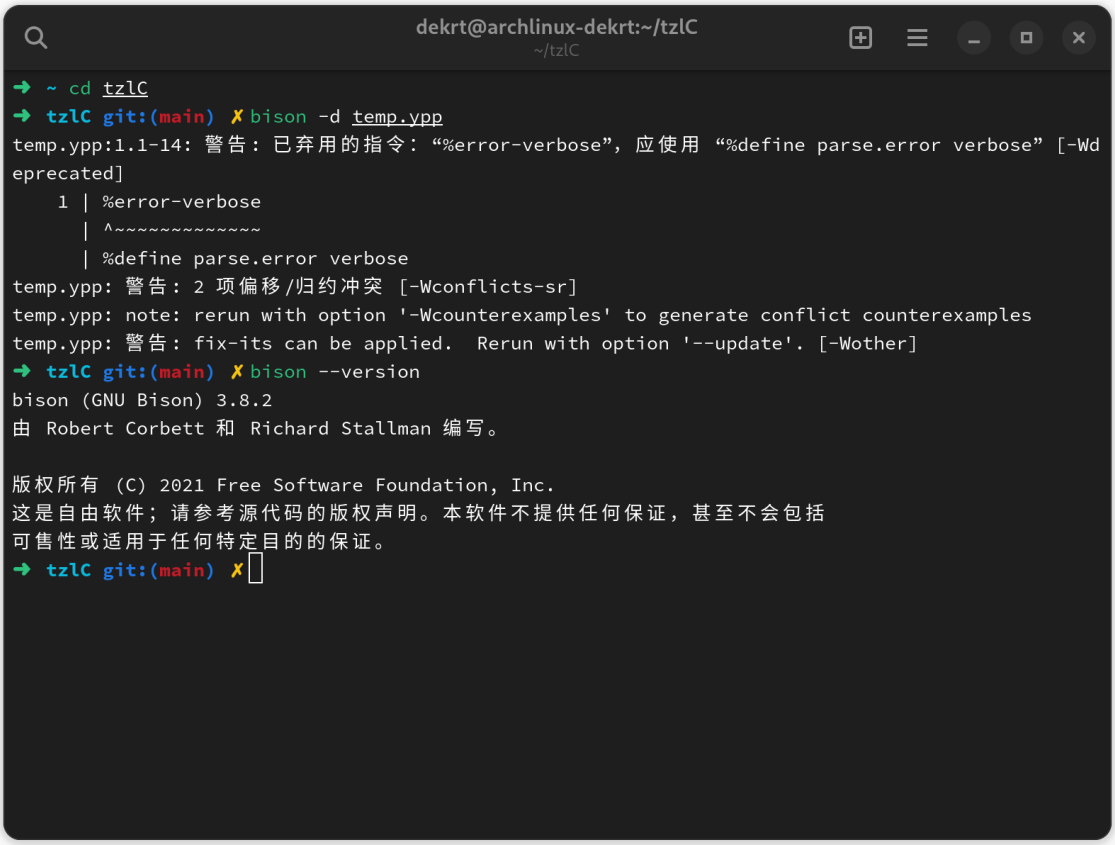
5.1.3 符号的识别

```
":"     {return COLON;}
"--"    {return DMINUS;}
"%"     {return MOD;}
"["     {return LB;}
"]"     {return RB;}
```

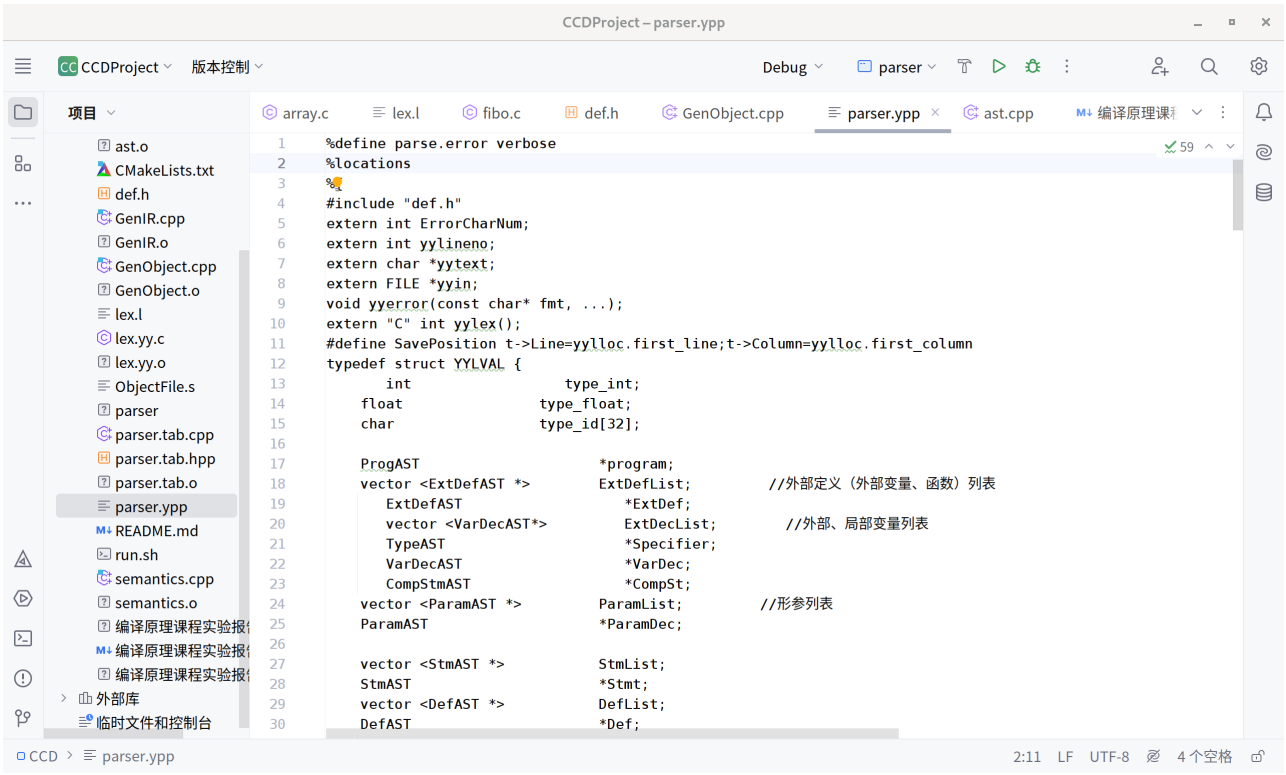
5.2 语法分析器

5.2.1 versose指令

在使用高版本bison时，会提示指令已弃用，按照要求修改即可：



将第一行的"%error-verbose"修改为"%define parse.error verbose"即可。



增加相关的stmt语法，并将$$替换为对应的指针：

```
| FOR LP Exp SEMI Exp SEMI Exp RP Stmt
{ForStmAST *t=new ForStmAST(); t->SinExp=$3; t->Cond=$5; t->EndExp=$7; t->Body=$9; $$=t; SavePosition;}
| SWITCH LP Exp RP LC CaseList RC   {SwitchStmAST *t=new SwitchStmAST(); t->Exp=$3; t->Cases=$6; t->containDefault=0; $$=t; SavePosition;}
| SWITCH LP Exp RP LC CaseList DEFAULT COLON StmList RC
{SwitchStmAST *t=new SwitchStmAST(); t->Exp=$3; t->Cases=$6; t->containDefault=1; t->Default=$9; $$=t; SavePosition;}
| BREAK SEMI           {BreakStmAST *t=new BreakStmAST(); $$=t; SavePosition; }
| CONTINUE SEMI        {ContinueStmAST *t=new ContinueStmAST(); $$=t; SavePosition; }
```

5.3 def.h头文件

添加函数调用表定义

```
class VarSymbol:public Symbol{
    public:
        string Alias;    //别名，为解决中间代码中，作用域嵌套变量同名的显示时的二义性问题
        int Offset;      //变量在对应AR中的偏移量
        int isGolbal=0;
        vector <int> Dims;
};
```

同时修改Opn的定义，增添isGlobal变量标志其是否为全局变量：

```
class Opn
{
    public:
        string  Name;       //变量别名（为空时表示常量）或函数名
        int     Type;
        int     isGolbal=0;
        union
        {
```

```
            int     Offset;     //AR中的偏移量
            void    *SymPtr;    //符号表指针
            char  constCHAR;
            int   constINT;
            float constFLOAT;
        };

        Opn(string Name,int Type,int Offset,int isGolbal):Name(Name),Type(Type),Offset(Offset),isGolbal(isGolbal){};
        Opn(){};
};
```

增加for、case、break、continue等语句的定义：

```

class ForStmAST : public StmAST {     //for语句
    public:
        ExpAST *SinExp, *Cond, *EndExp;
        StmAST *Body;

        void DisplayAST(int l) override;
        void Semantics(int &Offset, int canBreak, int canContinue, int &isReturn, BasicTypes returnType) override;
        void GenIR(string lableCase, string lableBreak, string lableContinue) override;
};

class CaseStmAST : public StmAST {     //for语句
    public:
        ExpAST *Cond;
        vector <StmAST *> Body;

        void DisplayAST(int l) override;
        void Semantics(int &Offset, int canBreak, int canContinue, int &isReturn, BasicTypes returnType) override;
        void GenIR(string lableCase, string lableBreak, string lableContinue) override;
};

class SwitchStmAST : public StmAST {     //for语句
    public:
        ExpAST *Exp;
        vector <CaseStmAST *> Cases;
        int containDefault;
        vector <StmAST *> Default;

        void DisplayAST(int l) override;
        void Semantics(int &Offset, int canBreak, int canContinue, int &isReturn, BasicTypes returnType) override;
        void GenIR(string lableCase, string lableBreak, string lableContinue) override;
};

```

5.4 ast.cpp

增加相应语句：

```
1
2  void ForStmAST::DisplayAST(int indent)
3  {   //显示for循环语句
4      space(indent);
5      cout<<"for语句：<<endl;
6      space(indent+2);
7      cout<<"单次表达式：<<endl;
8      SinExp->DisplayAST(indent+8);
9      space(indent+2);
10     cout<<"循环条件：<<endl;
11     Cond->DisplayAST(indent+8);
12     space(indent+2);
13     cout<<"末尾循环体：<<endl;
14     EndExp->DisplayAST(indent+8);
15     space(indent+2);
16     cout<<"循环体：<<endl;
17     Body->DisplayAST(indent+8);
18 }
19
20 void CaseStmAST::DisplayAST(int indent)
21 {   //显示case语句
22     space(indent);
23     cout<<"常量Key：<<endl;
24     Cond->DisplayAST(indent+4);
25     if (Body.size())
26     {
27         space(indent);
28         cout<<"语句部分:"<<endl;
29         for(auto a:Body)
30             a->DisplayAST(indent+4);
31     }
32 }
33
34 void SwitchStmAST::DisplayAST(int indent)
35 {   //显示case语句
36     space(indent);
37     cout<<"表达式：<<endl;
38     Exp->DisplayAST(indent+4);
39     if (Cases.size())
40     {
41         space(indent);
42         cout<<"Case部分:"<<endl;
43         for(auto a:Cases)
44             a->DisplayAST(indent+4);
45     }
46     if (containDefault && Default.size())
47     {
48         space(indent);
49         cout<<"Default部分:"<<endl;
50         for(auto a:Default)
51             a->DisplayAST(indent+4);
52     }
```

```
}

void BreakStmAST::DisplayAST(int indent)
{
    space(indent);
    cout<<"break语句"<<endl;
}

void ContinueStmAST::DisplayAST(int indent)
{
    space(indent);
    cout<<"continue语句"<<endl;
}
```

5.5 semantics.cpp

这个部分空缺较多，主要将有注释没有代码的部分进行补全：

```
else if (SymPtr->Kind=='A') //符号是数组，需要显示各维大小
{
    cout<<"偏移量: "<<((VarSymbol*)SymPtr)->Offset  << "  全局: " <<
((VarSymbol*)SymPtr)->isGolbal << "  ";
    cout<< ((VarSymbol*)SymPtr)->Dims.size() <<"维: ";
    for(int i=0;i<((VarSymbol*)SymPtr)->Dims.size();i++)
        cout<< ((VarSymbol*)SymPtr)->Dims[i] << " ";
 }
```

同时增添前文所述的语句，增加更多的错误检查，增加多维数组的部分：

```
void VarDecAST::Semantics(int &Offset, TypeAST *Type)
{
    if (!SymbolStack.LocateNameCurrent(Name))  //当前作用域未定义，将变量加入符号表
    {
        VarDefPtr=new VarSymbol();
        VarDefPtr->Name=Name;
        VarDefPtr->Dims=Dims;
        VarDefPtr->isGolbal=0;
        VarDefPtr->Alias=NewAlias();
        if (typeid(*Type)==typeid(BasicTypeAST))
            VarDefPtr->Type=((BasicTypeAST*)Type)->Type;
        VarDefPtr->Offset=Offset;

        if (!Dims.size())
        {
            VarDefPtr->Kind='V';
            Offset+=TypeWidth[VarDefPtr->Type];
        }
        else
        {
            VarDefPtr->Kind='A';
            int ans = 1;
            for(int i = 0; i < Dims.size(); i++)
                ans *= Dims[i];
            Offset+=ans*TypeWidth[VarDefPtr->Type];
        }
```

```
27
28          if (Exp)                        //有初值表达式时的处理
29          {
30              Exp->Semantics(Offset);
31              if(Dims.size())
32                  Errors::ErrorAdd(Line,Column,"数组用表达式初始化") ;
33          }
34
35          SymbolStack.Symbols.back()->Symbols.push_back(VarDefPtr);
36      }
37      else Errors::ErrorAdd(Line,Column,"变量 "+Name+" 重复定义") ;
38  }
39
40  void VarDecAST::Semantics(int &Offset, int &GolbalOffset, TypeAST *Type)
41  {
42      if (!SymbolStack.LocateNameCurrent(Name))  //当前作用域未定义，将变量加入符号表
43      {
44          VarDefPtr=new VarSymbol();
45          VarDefPtr->Name=Name;
46          VarDefPtr->Dims=Dims;
47          VarDefPtr->isGolbal=1;
48          VarDefPtr->Alias=NewAlias();
49          if (typeid(*Type)==typeid(BasicTypeAST))
50              VarDefPtr->Type=((BasicTypeAST*)Type)->Type;
51
52          VarDefPtr->Offset=GolbalOffset;
53
54          if (!Dims.size())
55          {
56              VarDefPtr->Kind='V';
57              GolbalOffset+=TypeWidth[VarDefPtr->Type];
58          }
59          else
60          {
61              VarDefPtr->Kind='A';
62              int ans = 1;
63              for(int i = 0; i < Dims.size(); i++)
64                  ans *= Dims[i];
65              GolbalOffset+=ans*TypeWidth[VarDefPtr->Type];
66          }
67
68          SymbolStack.Symbols.back()->Symbols.push_back(VarDefPtr);
69      }
70      else Errors::ErrorAdd(Line,Column,"变量 "+Name+" 重复定义") ;
71  }
```

5.6 GenIR.cpp

补全代码中空缺的部分，并增加了C++新特性。参考资料：https://silverbullettt.bitbucket.io/courses/compiler-2022/projects/Project_3.pdf

5.7 GenObject.cpp

参考网上资料编写，有点难。主要参考南京大学的指导书：https://silverbullettt.bitbucket.io/courses/compiler-2022/projects/Project_4.pdf

6 实验小结与体会

通过本次"C minus"编译器的设计与实现实验，我深刻体会到了编译原理的复杂性与魅力。在这个过程中，我不仅加深了对编译器各个模块功能的理解，还学习到了如何将理论知识应用于实践中，解决实际问题。

1. 对编译器各个阶段的理解加深

通过亲自设计并实现编译器的各个阶段，我对词法分析、语法分析、语义分析、中间代码生成、目标代码生成等环节有了更为深入的认识。特别是在处理复杂的语法结构和语义检查时，我意识到了精确定义语言规范的重要性。此外，中间代码的生成过程使我认识到了编译器如何将高级语言的抽象结构转换为接近机器语言的代码。

2. 编码实践与问题解决能力的提升

在实验过程中，我遇到了诸如符号表管理不当、语法规则定义错误、中间代码生成逻辑不清晰等一系列问题。通过查阅资料、反复调试和优化代码，我不仅解决了这些问题，还锻炼了我的编码实践能力和问题解决能力。这些经验对我未来在软件开发领域的学习和工作将大有裨益。

3. 系统测试的重要性

系统测试阶段使我认识到了测试在软件开发过程中的重要性。通过对编译器进行一系列的功能性和健壮性测试，我能够确保编译器的可靠性和稳定性。这一阶段的经历教会了我如何设计有效的测试用例，以及如何分析和利用测试结果来改进软件。

4. 对编译原理整体认识的提升

在本实验中，我不仅学到了关于编译器具体实现的技术细节，更重要的是，我对编译原理作为一个整体的认识有了明显提升。我了解到编译器不仅仅是一个将高级语言代码转换为机器代码的工具，它还涵盖了代码优化、资源管理等多个方面，是计算机科学中极其重要的一环。

5. 深化理论知识，激发进一步学习的兴趣

通过这次实验，我对编译原理课程中学到的理论知识有了更深的理解和体会。实践过程中遇到的挑战和问题激发了我进一步深入学习编译原理以及相关计算机科学领域知识的兴趣。我相信这次实验经历将对我未来的学术和职业生涯产生长远的影响。

综上所述，本次"C minus"编译器的设计与实现实验是一次极其宝贵的学习经历。它不仅使我掌握了编译器的设计与实现技术，更重要的是，它提高了我的编程能力、问题解决能力，并且加深了我对编译原理这一学科的认识和兴趣。

7 参考文献

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008