

**课 程 实 验 报 告**

**课程名称： 编 译 技 术 实 验**

**专业班级： 软件2203**

**学 号： U202217241**

**姓 名： 莫嘉豪**

**指导教师： 胡雯蔷**

**报告日期： 2024年 12月 14 日**

**软件学院**

目录

**一、概述……………………………………………………………………………1**

**二、mini C语言的定义及描述……………………………………………………1**

**三、词法分析………………………………………………………………………8**

**四、语法分析………………………………………………………………………12**

**五、语义分析………………………………………………………………………16**

**六、符号表…………………………………………………………………………18**

**七、中间代码生成…………………………………………………………………21**

**八、总结……………………………………………………………………………24**

编译技术实验及课程设计报告

**一、概述**

本实验报告详细阐述了“mini C语言”编译器的开发过程，这是一个基于编译原理课程的项目，旨在实现一个简化版的C语言编译器。该编译器涵盖了词法分析、语法分析、语义分析、中间代码生成等编译器核心环节，遵循编译原理中的基本概念和方法。报告解释了编译器各个模块的设计与实现，提供了对编译器工作机制的深刻理解，为编译原理的学习提供了实践经验。

**二、mini C语言的定义及描述**

**2.1 mini C语言的语法概述**

1. 程序结构（Program Structure）:

prog：构成了程序的基础框架，由一系列全局定义组成，这些定义可以是函数或者变量。

2. 全局定义（Global Definitions）:

GlobDefList：描述了全局定义的集合，可以是空的，或者由单个全局定义和后续的全局定义序列组成。

GlobDef：表示一个全局定义，可能是一个数据类型的声明，后跟变量声明列表和一个分号；或者是函数的声明，包括数据类型、名称、参数列表和一段复合语句；也可能是一个仅声明函数原型而不包含函数体的情况。

3. 类型和声明（Types and Declarations）:

typeSpec：指定了可用的数据类型，例如字符型、整型和浮点型。

varDec：描述了变量的声明方式，可以是单个变量名或者一个数组。

paramDec：定义了函数参数的数据类型和名称。

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

compStmt：构建了复合语句的结构，由一系列局部变量声明和语句序列组成。

stmt：涵盖了多种语句类型，包括表达式语句、复合语句、返回语句、条件判断语句、循环语句以及跳转语句等。

5. 表达式（Expressions）:

expr：构建了表达式的结构，包括赋值表达式、算术运算、比较运算等。

支持的运算包括基本的算术运算（如加法、减法、乘法、除法）和比较运算（如大于、等于）等。

表达式可以是一个变量名、一个整数、一个浮点数或者是一个括号内的更复杂的表达式。

**2.2 mini C语言的BNF描述**

prog: GlobDefList

GlobDefList:

| GlobDef GlobDefList

GlobDef: typeSpec varDecList SEMI

| typeSpec ID LP paramList RP compStmt

| typeSpec ID LP paramList RP SEMI

typeSpec: CHAR

| INT

| FLOAT

varDecList: varDec

| varDec COMMA varDecList

varDec: ID

| varDec LB INT RB

paramList:

| paramDec

| paramList COMMA paramDec

paramDec: typeSpec ID

compStmt: LC varDecList stmtList RC

stmtList:

| stmt stmtList

varDecList:

| varDec varDecList

varDec: ID

| varDec ASSIGN expr

stmt: expr SEMI

| compStmt

| RETURN expr SEMI

| RETURN SEMI

| IF LP expr RP stmt %prec LOWER\_THEN\_ELSE

| IF LP expr RP stmt ELSE stmt

| WHILE LP expr RP stmt

| FOR LP expr SEMI expr SEMI expr RP stmt

| BREAK SEMI

| CONTINUE SEMI

| error SEMI

expr: expr ASSIGN expr

| expr PLUS expr

| expr MINUS expr

| expr STAR expr

| expr DIV expr

| LP expr RP

| MINUS expr %prec UMINUS

| PLUS expr %prec UPLUS

| expr GT expr

| expr GE expr

| expr LT expr

| expr LE expr

| expr NE expr

| expr EQ expr

| ID LP args RP

| ID

| INT

| FLOAT

args: | expr

| args COMMA expr

**2.3 mini C语言的符号表结构定义**

1. 基本符号信息（BaseSymbol 类）:

SymbolName: 标识符的名称。

SymbolType: 标识符的类型，可以是char、int、float。

SymbolTypeCode: 标识符类型的编码，用于快速判断类型。

SymbolKind: 标识符的种类，例如变量（V）、函数（F）、参数（P）、数组（A）等。

typedef enum {

SYMBOL\_VARIABLE,

SYMBOL\_FUNCTION,

SYMBOL\_PARAMETER,

SYMBOL\_ARRAY

} SymbolKind;

typedef enum {

TYPE\_CHAR,

TYPE\_INT,

TYPE\_FLOAT

} SymbolType;

typedef struct BaseSymbol {

char\* SymbolName;

SymbolType SymbolType;

SymbolKind SymbolKind;

} BaseSymbol;

2. 变量符号信息（VariableSymbol 类）:

继承自 BaseSymbol 类，专门用于表示变量。

AliasName: 变量的别名，用于解决作用域嵌套时同名变量的问题。

Offset: 变量在活动记录中的偏移量。

IsGlobal: 表示变量是否为全局变量。

Dimensions: 数组的维度，用于表示数组类型的变量。

typedef struct VariableSymbol {

BaseSymbol base;

char\* AliasName;

int Offset;

int IsGlobal;

int Dimensions;

} VariableSymbol;

3.函数符号信息（FunctionSymbol 类）:

继承自 BaseSymbol 类，专门用于表示函数。

StackSize: 函数的活动记录大小，用于函数调用时分配内存单元。

ParamCount: 函数的形式参数个数。

ParamTable: 指向参数符号表的指针。

IsDefined: 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。

typedef struct FunctionSymbol {

BaseSymbol base;

int StackSize;

int ParamCount;

VariableSymbol\*\* ParamTable;

int IsDefined;

} FunctionSymbol;

4. 作用域内的符号表（ScopeSymbols 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列表。

typedef struct ScopeSymbols {

BaseSymbol\*\* Symbols;

int Size;

} ScopeSymbols;

5.符号表栈（SymbolStack 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（FindInCurrentScope）和全局作用域（FindGlobalScope）查找符号的功能。

typedef struct SymbolStack {

ScopeSymbols\* scopes;

int size;

int capacity;

} SymbolStack;

// 函数声明

BaseSymbol\* FindInCurrentScope(SymbolStack\* stack, const char\* name);

BaseSymbol\* FindGlobalScope(SymbolStack\* stack, const char\* name);

**2.4 mini C语言的中间代码结构定义**

1. 四元式结构

操作数（Operand 类）

操作数类（Operand）用于表示中间代码中的操作数，可以是变量、常量或者函数名。它包含以下属性：

varName：表示变量的别名或者函数名，如果为空，则表示该操作数是一个常量。

dataType：表示操作数的类型，可以是字符（CHAR）、整数（INTEGER）或浮点数（FLOAT）。

isGlobalVar：表示该操作数是否为全局变量。

一个联合体，用于存储变量在活动记录（AR）中的偏移量、符号表指针或常量值，具体使用哪个取决于操作数的类型。

中间代码（Quadruple 类）

中间代码（Quadruple）用于表示编译过程中的指令，它包含以下属性：

operation：表示四元式的操作类型，如加法、减法、乘法、除法等。

arg1 和 arg2：分别表示操作的第一和第二操作数。

output：表示操作的结果，也是一个操作数。

class Operand {

public:

string varName;

int dataType;

bool isGlobalVar;

union {

int offset;

void \*symbolPtr;

char constantChar;

int constantInt;

float constantFloat;

};

Operand(string varName, int dataType, int offset, bool isGlobalVar) : varName(std::move(varName)), dataType(dataType), offset(offset), isGlobalVar(isGlobalVar) {};

Operand() : dataType(-1), isGlobalVar(false) {};

};

enum OperationCode {

ADD, SUB, MUL, DIV, MOD,

EQ, NE, LT, GT, LE, GE,

INC, DEC,

ASSIGN, ADD\_ASSIGN, SUB\_ASSIGN, MUL\_ASSIGN, DIV\_ASSIGN,

IF, WHILE, BREAK, CONTINUE, FOR

};

class Quadruple {

public:

int operation;

Operand arg1;

Operand arg2;

Operand output;

Quadruple(int operation, Operand arg1, Operand arg2, Operand output) : operation(operation), arg1(std::move(arg1)), arg2(std::move(arg2)), output(std::move(output)) {}

};

1. 使用四元式作为中间代码的好处

中间代码的规范化：四元式提供了一种统一的方式来表示各种操作，简化了编译器的设计和实现。

代码优化的便利性：四元式的形式便于进行代码优化，如死代码消除、循环优化等。

代码生成的灵活性：四元式与目标机器无关，使得代码生成阶段更加灵活，易于适配不同的目标平台。

错误检测和调试的便捷：四元式的形式有助于错误检测和程序调试，因为它提供了清晰的操作和数据流信息。

**2.5 mini C语言的错误提示**

1. 错误提示涉及的代码结构

错误存储（ErrorManager 类）:

errors: 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 ErrorManager

{

public:

static vector <Error> errors;

static void AddError(int line, int column, string message);

static void DisplayErrors();

static inline bool HasErrors() { return !errors.empty(); }

};

错误添加（AddError 方法）:

静态函数，用于向errors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errors向量中。

void ErrorManager::AddError(int line, int column, string message) {

Error error = {line, column, move(message)};

errors.push\_back(error);

}

错误展示（DisplayErrors 方法）:

静态函数，用于展示收集到的所有错误信息。

遍历errors向量，并打印每个错误的详细信息（包括位置和消息）。

void ErrorManager::DisplayErrors() {

for (const auto& error : errors)

cout << "Error at line " << error.line << ", column " << error.column << ": " << error.message << endl;

}

错误检查（HasErrors 方法）:

提供快速检查errors向量是否为空的功能，即编译过程中是否发现了错误。

static inline bool HasErrors() {

return !errors.empty();

}

**三、词法分析**

**3.1 词法分析方法**

词法分析器是编译器前端的关键组件，负责将源代码分解成一系列词法单元（tokens），这些词法单元是语法分析和后续编译步骤的基础。以下是对mini C语言词法分析器的主要特点和工作机制的简要说明：

1. 词法单元识别

词法分析器利用正则表达式定义了mini C语言的词法单元，涵盖了数据类型关键字（如char, int, float）、控制流语句（如if, while, for）、运算符、分隔符以及标识符和常量（整数和浮点数）。这些规则确保词法分析器能够精确地从源代码中识别出这些基本的编程元素。

2. 动作代码执行

每当词法分析器匹配到一个词法规则时，它会执行一个动作代码。这些动作通常涉及将匹配到的文本（例如，常量值或标识符名）存储到一个数据结构中，并返回一个与该词法单元对应的标记给语法分析器。例如，当识别到一个整数常量时，词法分析器会将其值存储在yylval.type\_int中，并返回INT\_CONST标记。

3. 错误检测与报告

词法分析器具备错误检测功能，能够识别非法字符或结构，例如未识别的符号或不完整的注释。对于每个非法符号，词法分析器会输出错误信息，并增加错误计数器ErrorCharNum，以便跟踪源代码中的错误。

4. 位置信息跟踪

词法分析器使用yylineno和yycolumn变量来跟踪词法单元的位置信息，在每个词法单元的动作代码中，这些位置信息会被更新，以确保它们反映了当前词法单元的准确位置。

5. 注释忽略

词法分析器通过定义的正则表达式规则来识别并忽略单行（//）和多行（/\* ... \*/）注释。确保注释在词法分析阶段就被忽略，不会影响后续的语法分析。

6. 空白字符处理：

词法分析器会忽略所有空白字符，包括空格、制表符和换行符。

7. 词法单元的返回值：

对于每个词法单元，词法分析器会返回一个预定义的标记值，这些标记值在语法分析器中用于构建解析树。例如，关键字if会返回一个特定的标记值，以便语法分析器能够识别它并采取相应的解析动作。

**3.2 词法分析代码**

%option yylineno

%{

#include "string.h"

#include "parser.tab.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;

float type\_float;

char type\_id[32];

} YYLVAL;

#define YYSTYPE YYLVAL

%}

/\* 词法单元定义 \*/

id [A-Za-z\_][A-Za-z0-9\_]\*

intconst [1-9][0-9]\*|0

floatconst [0-9]+\.[0-9]+([eE][-+]?[0-9]+)?

Oneline\_comment "//"[^\n]\*

Multiline\_comment "/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/"

%%

"char" { strcpy(yylval.type\_id, yytext); return CHAR; }

"int" { strcpy(yylval.type\_id, yytext); return INT; }

"float" { yylval.type\_float = atof(yytext); return FLOAT; }

"if" { return IF; }

"else" { return ELSE; }

"while" { return WHILE; }

"break" { return BREAK; }

"continue" { return CONTINUE; }

"for" { return FOR; }

{id} { strcpy(yylval.type\_id, yytext); return ID; }

[0-9]+ { yylval.type\_int = atoi(yytext); return INT\_CONST; }

{floatconst} { yylval.type\_float = atof(yytext); return FLOAT\_CONST; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return STAR; }

"/" { return DIV; }

"%" { return MOD; }

"++" { return INC; }

"--" { return DEC; }

"==" { return EQ; }

"!=" { return NEQ; }

">" { return GT; }

"<" { return LT; }

">=" { return GE; }

"<=" { return LE; }

"=" { return ASSIGN; }

"&&" { return AND; }

"||" { return OR; }

"!" { return NOT; }

";" { return SEMICOLON; }

":" { return COLON; }

"," { return COMMA; }

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

"[" { return LBRACKET; }

"]" { return RBRACKET; }

{Oneline\_comment} {}

{Multiline\_comment} {}

[ \t\r]+ { /\* ignore whitespace \*/ }

\n { yycolumn = 1; }

. { printf("在第 %d 行出现错误的符号 '%s' \n", yylineno, yytext); ErrorCharNum++; }

%%

int yywrap()

{

return 1;

}

**四、语法分析**

**4.1 语法分析方法**

语法分析器是编译器前端的关键组件，位于词法分析器之后，负责将词法单元序列（tokens）组织成符合特定编程语言语法规则的结构。以下是对mini C语言语法分析器的主要特点和工作机制的简要说明：

1.预处理指令和宏定义

语法分析文件开始部分包含了一系列的预处理指令和宏定义，这些指令对后续的解析器行为进行了配置。例如，%define parse.error verbose指令定义了错误处理的行为，使得解析器在遇到错误时提供详细的错误信息。%locations指令启用了位置跟踪，这对于记录错误位置和构建AST时保存位置信息至关重要。YYLVAL结构体定义了各种类型的语义值，这些值将在解析过程中被填充并用于构建抽象语法树（AST）。

%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;

ExtDefAST \*ExtDef;

vector <VarDecAST\*> ExtDecList;

TypeAST \*Specifier;

VarDecAST \*VarDec;

CompStmAST \*CompSt;

vector <ParamAST \*> ParamList;

ParamAST \*ParamDec;

vector <StmAST \*> StmList;

StmAST \*Stmt;

vector <DefAST \*> DefList;

DefAST \*Def;

vector <VarDecAST \*> DecList;

VarDecAST \*Dec;

ExpAST \*Exp;

vector <ExpAST \*> Args;

CaseStmAST \*Case;

vector <CaseStmAST \*> CaseList;

}YYLVAL;

#define YYSTYPE YYLVAL

2.类型定义和语义值分配：

通过%type指令定义了非终结符的语义值类型。例如，%type <program> program指定了program非终结符的语义值类型为program，这是一个指向ProgAST结构的指针。这样的定义允许在解析过程中为每个非终结符分配一个具体的语义值，这些值将在构建AST时使用。

%}

%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.终结符和语义值类型：

%token指令定义了终结符及其对应的语义值类型。例如，%token <type\_int> INT指定了INT终结符的语义值类型为type\_int，这是一个整数。这样的定义允许词法分析器在识别到INT类型的词法单元时，将其值传递给语法分析器。

%token <type\_int> INT

%token <type\_id> ID TYPE

%token <type\_float> FLOAT

%token CHAR INT FLOAT IF ELSE WHILE BREAK CONTINUE FOR

%token PLUS MINUS STAR DIV MOD EQ NEQ GT LT GE LE

%token AND OR NOT SEMICOLON COLON COMMA

%token LP RP LB RB

%token INC DEC ASSIGN

%token LBRACE RBRACE LBRACKET RBRACKET

4.运算符优先级和结合性

通过一系列的指令定义了运算符的优先级和结合性。例如，%left PLUS MINUS指令定义了PLUS和MINUS运算符为左结合，而%right NOT指令则定义了NOT运算符为右结合。这些规则确保了解析器能够正确地解析包含这些运算符的表达式。

%left COMMA

%left ASSIGN

%left OR

%left AND

%left LT LE GT GE

%left NE EQ

%left PLUS MINUS

%left STAR DIV

%right NOT

%left INC DEC

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

5.语法规则和AST节点创建：

parser.y文件的核心部分是一系列的产生式规则，这些规则定义了语言的语法结构。每个产生式规则不仅定义了语法结构，而且在规则的右侧包含了创建相应AST节点的代码。例如，在program规则中，当解析器匹配到ExtDefList时，会创建一个ProgAST节点，并将ExtDefList作为子节点。这样的结构化表示为后续的语义分析和代码生成提供了基础。

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);

}

;

6.错误处理和恢复：

yyerror函数是错误处理的核心，它接收一个格式化字符串和可变参数列表，然后使用va\_list和va\_start等宏来处理这些参数。该函数将错误信息和位置信息添加到错误处理系统中,通过记录错误的位置（如行号和列号），可以更容易地定位和修复代码中的问题。

#include<stdarg.h>

void yyerror(const char\* fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

Errors::ErrorAdd(yylloc.first\_line, yylloc.first\_column, fmt, ap);

va\_end(ap);

}

7.主函数和解析器驱动：

main函数是整个解析器的入口点。它打开指定的源代码文件，初始化行号，然后调用yyparse函数开始解析过程。yyparse函数是Bison生成的解析器的入口点，它将根据定义的语法规则和词法分析器提供的词法单元来构建AST。

int main(int argc, char \*argv[]){

yyin = fopen(argv[1], "r");

if (!yyin) return 0;

yylineno = 1;

yyparse();

return 0;

}

**4.2 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。

**五、语义分析**

**5.1 语义分析方法**

在mini C语言编译器的实现中，语义分析部分主要负责检查程序的语义正确性，并生成中间代码，这部分与语法分析部分共同进行。以下是语义分析部分的关键方法和机制的主要说明：

1.类型检查

在Specifier规则中，根据类型关键字（如int、float）设置类型信息。这一步确保了变量声明、函数定义等的类型正确性。例如，当解析器遇到类型声明时，它会根据关键字创建相应的TypeAST节点，并设置其类型。

Specifier:

TYPE

{

BasicTypeAST \*t = new BasicTypeAST();

if (strcmp($1, "int") == 0) t->Type = T\_INT;

if (strcmp($1, "float") == 0) t->Type = T\_FLOAT;

if (strcmp($1, "void") == 0) t->Type = T\_VOID;

$$ = t;

SavePosition;

}

2.函数定义和声明检查

在ExtDef规则中，处理函数和变量的定义。这包括检查函数的返回类型、参数列表和函数体。编译器会创建函数定义的AST节点，包括函数的返回类型、名称、参数列表和函数体。这确保了函数定义的语法和语义正确性。

ExtDef:

Specifier ExtDecList SEMICOLON

{

ExtVarDefAST \*t = new ExtVarDefAST();

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;

}

3.函数调用检查：

在FunctionCallTable类中，管理函数调用信息，包括添加和删除函数调用。这确保了函数调用的合法性，例如检查函数是否存在以及参数数量和类型是否匹配。

class FunctionCallTable {

public:

std::vector<std::tuple<int, int, std::string>> FuncCalls;

void addFuncCall(int Line, int Column, const std::string& Name);

void deleteFuncCall(const std::string& Name);

};

4.控制流检查

在Stmt规则中处理控制流语句，如if、while、for和return。这包括检查条件表达式的类型和返回语句的上下文。确保了控制流语句的语法和语义正确性，例如检查if语句的条件是否为布尔表达式，return语句是否在函数内部等。

Stmt:

IfStmAST \*t = new IfStmAST();

t->Cond = $3;

t->ThenStm = $5;

$$ = t;

SavePosition;

| WhileStmAST \*t = new WhileStmAST();

t->Cond = $3;

t->Body = $5;

$$ = t;

SavePosition;

| ReturnStmAST \*t = new ReturnStmAST();

t->Exp = $2;

$$ = t;

SavePosition;

**5.2 错误处理**

在yyerror函数中，处理分析过程中的错误，并记录错误信息。使用可变参数列表来处理不同类型的错误信息，并将错误的位置信息（行号和列号）一同记录下来。这个函数是编译器错误处理机制的一部分，它将错误信息格式化并添加到错误列表中。

void yyerror(const char\* fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

errors.ErrorAdd(yylloc.first\_line, yylloc.first\_column, fmt, ap);

va\_end(ap);

}

**六、符号表**

**6.1 符号表管理**

符号表是编译器前端的重要组成部分，负责存储和管理程序中出现的标识符（如变量名、函数名等）的相关信息。它确保了程序中标识符的一致性和正确性，并为语义分析、代码生成等后续阶段提供了必要的信息。以下是对mini C语言编译器中符号表的主要特点和工作机制的简要说明：

1.符号表结构设计

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）。允许处理不同层级的作用域，例如全局作用域和局部作用域（如函数内部）

class SymbolStackDef {

public:

std::vector<SymbolsInAScope\*> scopes;

Symbol\* LocateNameCurrent(const std::string& Name);

Symbol\* LocateNameGlobal(const std::string& Name);

void DisplaySymbolTable();

};

class SymbolsInAScope {

public:

std::vector<Symbol\*> Symbols;

};

2.符号表项定义

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）。这是符号表中每个条目的基本结构。

class Symbol {

public:

std::string Name;

SymbolKind Kind;

DataType Type;

Symbol(std::string name, SymbolKind kind, DataType type) : Name(name), Kind(kind), Type(type) {}

virtual ~Symbol() {}

};

3.变量和函数符号表项扩展

实现点：特定于变量的符号表项（VarSymbol 类）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。

class VarSymbol : public Symbol {

public:

std::string Alias;

int Offset;

bool isGlobal;

std::vector<int> Dims;

varSymbol(std::string name, DataType type, bool isGlobal = false) : Symbol(name, VAR, type), isGlobal(isGlobal) {}

};

class FuncSymbol : public Symbol {

public:

int ParamNum;

int ARSize;

FuncSymbol(std::string name, DataType type) : Symbol(name, FUNC, type) {}

};

4.符号表操作实现

实现点：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。

Symbol\* SymbolStackDef::LocateNameCurrent(const std::string& Name) {

SymbolsInAScope\* curScope = scopes.back();

for (auto& Symbol : curScope->Symbols) {

if (Symbol->Name == Name) {

return Symbol;

}

}

return nullptr;

}

Symbol\* SymbolStackDef::LocateNameGlobal(const std::string& Name) {

for (int i = scopes.size() - 1; i >= 0; i--) {

for (auto& Symbol : scopes[i]->Symbols) {

if (Symbol->Name == Name) {

return Symbol;

}

}

}

return nullptr;

}

5.符号表的展示

提供了 DisplaySymbolTable 函数来打印符号表的当前状态，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 SymbolStackDef::DisplaySymbolTable() {

for (size\_t i = 0; i < scopes.size(); i++) {

std::cout << "Scope " << i << ":\n";

for (auto sym : scopes[i]->Symbols) {

std::cout << "Name: " << sym->Name << ", Type: " << sym->Type << ", Kind: " << sym->Kind << "\n";

}

}

}

**6.2 符号表应用**

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。

**七、中间代码生成**

**7.1 中间代码生成原理**

中间代码生成是编译器前端的关键步骤之一，它将抽象语法树（AST）转换为中间表示形式，通常是一种四元式结构代码。以下是对mini C语言编译器中间代码生成部分的主要方法和机制的简要说明：

1.临时变量和标签生成

在中间代码生成过程中，经常需要引入临时变量和标签以支持表达式的计算和控制流的转移。这些临时变量和标签用于存储中间结果和标记代码中的特定位置。

std::map<std::string, int> temp\_var\_count;

std::map<std::string, int> label\_count;

std::string NewTemp() {

std::string temp\_name = "T";

temp\_name += std::to\_string(++temp\_var\_count["T"]);

return temp\_name;

}

std::string NewLabel() {

std::string label\_name = "L";

label\_name += std::to\_string(++label\_count["L"]);

return label\_name;

}

2.中间代码类

中间代码通常以四元式的形式表示，每个四元式包含一个操作码、两个操作数和一个结果。操作码表示要执行的操作，操作数是操作的输入，结果则是操作的输出。

class IRCode {

public:

OpCode op;

Operand left, right, result;

IRCode(OpCode o, Operand l = Operand(), Operand r = Operand(), Operand res = Operand())

: op(o), left(l), right(r), result(res) {}

};

7.2 中间代码生成函数

1.部分基础的中间代码生成函数

这些函数将遍历AST，并为每个节点生成相应的中间代码。以下是一些关键节点的中间代码生成函数的详细说明：

ProgAST::GenIR：这个函数负责生成程序的中间代码。它遍历所有的外部定义（如函数定义和全局变量声明），为每个外部定义生成代码。

ExtDefAST::GenIR：这个函数处理外部定义，包括变量和函数的定义。如果是变量定义，它会生成变量声明的四元式；如果是函数定义，它会生成函数的符号表条目，并递归地生成函数体的代码。

VarDecAST::GenIR：这个函数生成变量声明的中间代码。它会为新变量分配存储空间，并生成一个四元式来表示这个变量的声明。

ExpAST::GenIR：这是一个虚函数，为所有的表达式节点提供默认的生成四元式的接口。具体的表达式节点（如二元表达式、一元表达式、赋值表达式等）将覆盖这个函数来生成具体的四元式。

AssignAST::GenIR：这个函数处理赋值表达式的中间代码生成。它会生成代码来计算右侧表达式的值，然后将值存储到左侧变量的存储位置。

这些函数通常需要维护一个临时变量偏移量（temp\_var\_offset），用于为中间代码中的临时变量分配唯一的名称。此外，这些函数还需要与符号表交互，以获取变量的存储位置和其他属性。

void GenIR(ASTNode\* node, std::list<IRCode>& ir\_codes) {

if (node == nullptr) return;

node->GenIR(ir\_codes, temp\_var\_offset);

}

void ProgAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

for (auto& ext\_def : ExtDefs) {

ext\_def->GenIR(ir\_codes, temp\_var\_offset);

}

}

void ExtDefAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

if (Type != nullptr) {

Type->GenIR(ir\_codes, temp\_var\_offset);

}

for (auto& var\_dec : ExtVars) {

var\_dec->GenIR(ir\_codes, temp\_var\_offset);

}

}

void VarDecAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

Operand result(Name, Type, temp\_var\_offset);

ir\_codes.emplace\_back(ASSIGN, Operand(), Operand(), result);

temp\_var\_offset += sizeof(DataType);

}

void ExpAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

// 默认实现，具体表达式类型的IR生成将由子类覆盖

}

void AssignAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

LeftValExp->GenIR(ir\_codes, temp\_var\_offset);

RightValExp->GenIR(ir\_codes, temp\_var\_offset);

Operand left\_result(NewTemp(), LeftValExp->Type, temp\_var\_offset);

Operand right\_result(NewTemp(), RightValExp->Type, temp\_var\_offset + sizeof(DataType));

ir\_codes.emplace\_back(ASSIGN, right\_result, Operand(), left\_result);

}

2.控制流和跳转语句的处理

对于if语句、while语句、break和continue语句，需要生成相应的跳转代码来控制程序的执行流程。

IfStmAST::GenIR：对于if语句需要生成条件表达式的代码，然后根据条件的真假生成跳转代码。如果条件为真，则执行if块中的代码；如果条件为假，则跳过if块。

WhileStmAST::GenIR：对于while循环，需要生成循环条件的代码，并根据条件的真假生成跳转代码。如果条件为真，则执行循环体中的代码，然后跳回到循环条件的检查；如果条件为假，则跳出循环。这涉及到生成两个标签：一个用于循环的开始，另一个用于循环的结束。

void IfStmAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

Cond->GenIR(ir\_codes, temp\_var\_offset);

Operand cond\_result(NewTemp(), T\_INT, temp\_var\_offset);

ir\_codes.emplace\_back(ASSIGN, cond\_result, Operand(), Cond);

std::string label\_true = NewLabel();

std::string label\_false = NewLabel();

ir\_codes.emplace\_back(JEQ, cond\_result, Operand("\_CONST", T\_INT, 0), Operand(label\_false, 0, 0));

ThenStm->GenIR(ir\_codes, temp\_var\_offset);

ir\_codes.emplace\_back(GOTO, Operand(), Operand(), Operand(label\_true, 0, 0));

if (ElseStm != nullptr) {

ElseStm->GenIR(ir\_codes, temp\_var\_offset);

}

ir\_codes.emplace\_back(LABEL, Operand(label\_true, 0, 0), Operand(), Operand());

}

void WhileStmAST::GenIR(std::list<IRCode>& ir\_codes, int& temp\_var\_offset) {

std::string label\_start = NewLabel();

std::string label\_end = NewLabel();

ir\_codes.emplace\_back(LABEL, Operand(label\_start, 0, 0), Operand(), Operand());

Cond->GenIR(ir\_codes, temp\_var\_offset);

Operand cond\_result(NewTemp(), T\_INT, temp\_var\_offset);

ir\_codes.emplace\_back(ASSIGN, cond\_result, Operand(), Cond);

ir\_codes.emplace\_back(JEQ, cond\_result, Operand("\_CONST", T\_INT, 0), Operand(label\_end, 0, 0));

Body->GenIR(ir\_codes, temp\_var\_offset);

ir\_codes.emplace\_back(GOTO, Operand(), Operand(), Operand(label\_start, 0, 0));

ir\_codes.emplace\_back(LABEL, Operand(label\_end, 0, 0), Operand(), Operand());

}

**八、总结**

通过本次“mini C语言”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。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。

综上所述，本次“mini C语言”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的认识。