

课 程 设 计 报 告

设计题目：编译器前端及中间代码优化

班 级：计算机科学与技术 1802 班

组长学号：20184552

组长姓名：邱晓鹏

主 笔：邱晓鹏

设计时间：2020 年 7 月

设计分工

组长学号及姓名：20184552 邱晓鹏

分工：

- 1、设计了符号表。
- 2、完成了变量初始化、分支语句、赋值语句、函数调用、算术表达式、复合语句等的四元式设计和代码实现。
- 3、写了一个小的辅助程序，可以从带有语义函数调用的文法自动生成对应的代码，大幅降低了修改语义后插入到语法分析所花费的时间

组员 1 学号及姓名：20184446 寇凯淇

分工：

- 1、完成了词法分析器的设计和实现。词法分析器使用自动机对关键字、标识符、字符串、界符等进行分析判断，并生成相应表进行记录，同时产生对应的 Token 序列。
- 2、语法分析器的设计和实现。语法分析器采用 LL(1) 文法进行分析，接收来自于词法分析器产生的 Token 序列，根据文法产生 first 集与 follow 集，在生成 select 集的同时，填写预测表。并根据文法的预测表对 Token 序列进行读取，判断代码的合法性。

- 3、整理文法内容，简化文法。

组员 2 学号及姓名：20184574 张琳

分工：

负责四元式的优化部分。完善了基本块的划分算法，增加了对函数基本块、复合语句基本块的划分方法，对基本块进行了划分。对已有的 DAG 图的构建算法和四元式重组算法进行了适当的修改，能够处理像数组元素引用四元式、函数调用和函数返回四元式等特殊形式的四元式。实现了常值表达式节省，公共子表达式节省，并删除无用赋值四元式，多次调用优化函数实现循环优化。

组员 3 学号及姓名：20184323 黄达

分工：

完成了 逻辑表达式，跳转语句， 循环语句的语义分析。完成了寄存器的数据结构定义，寄存器分配函数，能够将非待用寄存器的值清空。本应完成的目标代码生成部分调试组装出现 bug，完成失败。

摘 要

本设计意在设计一个简单的文法编译器前端及中间代码优化部分。分别设计了一个支持数组、分支、循环、跳转、函数、复合等结构的文法，完整的词法分析器，基于 LL(1) 方法的语法分析器，符号表，基于语法制导的翻译，基于 DAG 图的局部优化。

本文在编译原理的教学基础上，对生成语法分析、符号表、语义分析、DAG 图部分均有一定改进，使其效果更好，设计过程效率更高。例如：LL(1) 分析是根据文法使用自己设计的程序自动分析生成，中间代码优化时的基本块划分的优化等。

关键字：编译原理，前端，中间代码优化，DAG 图，符号表，自动生成 LL(1) 分析

目 录

1 概述	7
2 课程设计任务及要求	7
2.1 设计任务.....	7
2.2. 设计要求.....	7
3 编译系统总体设计	8
3.1 文法.....	8
3.1.1 文法实现功能	8
3.1.2 文法内容.....	8
3.2 词法分析.....	10
3.3 语法分析.....	10
3.4 符号表.....	10
3.5 语义分析.....	10
3.6 中间代码优化.....	11
4 编译器前端设计	11
4.1 词法分析.....	11
4.1.1 功能.....	11
4.1.2 处理过程.....	11
4.1.3 数据结构.....	12
4.1.4 流程图.....	13
4.1.5 部分展示截图.....	13
4.2 语法分析.....	13
4.2.1 功能.....	13
4.2.2 处理过程.....	14
4.2.3 数据结构.....	14
4.2.4 流程图.....	15
4.2.5 部分展示截图.....	16
4.3 翻译文法与语义分析.....	17
4.3.1 变量定义及其初始化语句.....	17
4.3.2 分支语句	18
4.3.3 循环语句.....	18

4.3.4 赋值语句.....	19
4.3.5 函数调用语句.....	19
4.3.6 跳转语句	19
4.3.7 复合语句	19
4.4 符号表.....	19
4.4.1 总表结构	19
4.4.2 类型表.....	20
4.4.3 数组表.....	20
5 编译器后端设计	20
5.1 中间代码优化.....	20
5.1.1 功能.....	20
5.1.2 数据结构.....	21
5.1.3 算法.....	22
5.1.4 流程图.....	25
5.1.5 优化截图:	31
5.2 目标代码生成.....	32
6 总结	33
7 参考文献	34
8 收获、体会和建议	34
8.1 邱晓鹏心得体会	34
8.2 寇凯淇心得体会	34
8.3 张琳心得体会	35
8.4 黄达心得体会	36

1 概述

本次课程设计实现了一个编译器的前端部分以及中间代码的优化，共包括词法分析、语法分析、语义分析和中间代码优化四个部分。预期目标是实现一个完整的编译器，在现有基础上还应该添加目标代码生成的功能模块，但由于时间分配和接口的问题，目标代码生成部分未能成功整合到一起。编译器前端定义了一个类 c 语言的文法，包括函数的声明与调用、变量声明、变量初始化以及变量的使用、常量的使用、数组声明与使用、结构体声明与使用、选择语句、循环语句、赋值语句、break 和 continue 跳转语句、复合语句、算术运算、逻辑运算。编译器后端包含了中间代码的优化及目标代码的生成。

程序编译过程中，输入的源程序首先经过词法分析器，将程序生成为 Token 序列；在词法分析后，进入语法分析器，检查输入的程序是否符合语法规则，若符合语法规则则编译通过，否则输出错误提示与错误原因；语法分析通过后，利用符号表完成对变量的定义、属性等检查，将程序翻译成四元式。在此过程中符号表等贯穿整个程序，辅助进行定义和重定义检查，类型匹配校验，数据的越界和溢出检查，值单元存储分配信息，函数的参数传递与校验；得到四元式序列后，采用基于 DAG 的局部优化算法对四元式序列进行优化，实现了常数表达式节省，公共子表达式节省，删除了无用赋值四元式并多次调用优化功能实现循环优化。最终，产出优化后的四元式序列。

2 课程设计任务及要求

2.1 设计任务

一个简单文法的编译器前端及中间代码优化的设计和实现

2.2.设计要求

1、在深入理解编译原理基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；

4、确定测试方案，选择测试用例，对系统进行测试；

5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；

6、提交课程设计报告。

3 编译系统总体设计

3.1 文法

3.1.1 文法实现功能

设计了变量定义初始化、分支语句、循环语句、赋值语句、复合语句、函数、数组、逻辑表达式和算术表达式。变量类型支持整形、浮点数、字符型等多种类型。

3.1.2 文法内容

- (1) 程序→外部声明 程序 sub
- (2) 程序 sub→程序 | NULL
- (3) 外部声明→FUNCTION 函数定义 | 声明语句
- (4) 函数定义→类型说明 函数声明 复合语句
- (5) 函数声明→IDENTIFIER (函数声明 sub)
- (6) 函数声明 sub→NULL | 参数列表
- (7) 参数列表→参数声明 参数列表 sub
- (8) 参数列表 sub→NULL | 参数列表
- (9) 参数声明→类型说明 变量名声明
- (10) 声明语句→头文件声明 | 变量声明 ; | 结构体声明 ;
- (11) 头文件声明→#INCLUDE < 文件名 >
- (12) 文件名→IDENTIFIER . IDENTIFIER | STR
- (13) 变量声明→类型说明 初始化声明列表
- (14) 初始化声明列表→初始化声明 初始化声明列表 sub

- (15) 初始化声明列表 sub→NULL| , 初始化声明列表
- (16) 初始化声明→变量名声明 初始化声明 sub
- (17) 初始化声明 sub→NULL| = 初始化初始化 ;
- (18) 变量名声明→IDENTIFIER 向量
- (19) 向量→NULL | 向量纬度
- (20) 向量纬度→[CONSTANT] 向量纬度 sub
- (21) 向量纬度 sub→NULL| 向量纬度
- (22) 初始化→项| {初始化列表}
- (23) 初始化列表→项 初始化列表 sub
- (24) 初始化列表 sub→NULL| , 初始化列表
- (25) 结构体声明→struct IDENTIFIER { 结构体成员列表} 结构体声明
sub
- (26) 结构体声明 sub→NULL| IDENTIFIER
- (27) 结构体成员列表→变量声明 ; 结构体成员列表 sub
- (28) 结构体成员列表 sub→NULL| 结构体成员列表
- (29) 复合语句→{ 语句列表 }
- (30) 语句列表→声明语句 语句列表 sub| 语句 语句列表 sub
- (31) 语句列表 sub→NULL| 语句列表
- (32) 语句→复合语句| 选择语句|循环语句| 跳跃语句 ; | 赋值语句 ;
- (33) 选择语句→IF (逻辑表达式) 复合语句 选择语句 sub
- (34) 选择语句 sub→NULL| ELSE 复合语句
- (35) 循环语句→while (逻辑表达式) 复合语句|do 复合语句 while(逻辑
表达式)| for (定义 ; 逻辑表达式 ; 赋值) 复合语句
- (36) 定义→| 赋值语句| NULL
- (37) 逻辑表达式→逻辑与表达式 逻辑表达式 sub
- (38) 逻辑表达式 sub→OR_OP 逻辑表达式| NULL
- (39) 逻辑与表达式→逻辑非表达式 逻辑与表达式 sub
- (40) 逻辑与表达式 sub→AND_OP 逻辑与表达式| NULL
- (41) 逻辑非表达式→NOT_OP 逻辑非表达式|LOGIC (逻辑表达式) |
表达式语句 逻辑非表达式 sub
- (42) 逻辑非表达式 sub→RE_OP 表达式语句| < 表达式语句|>表达式语
句 |NULL
- (43) 赋值→赋值语句| NULL
- (44) 跳跃语句→CONTIUNE| BREAK| RETURN 表达式语句
- (45) 赋值语句→变量名声明 = 表达式语句
- (46) 表达式语句→一级算数 加减算数
- (47) 加减算数→+一级算数 加减算数| -一级算数 加减算数| NULL
- (48) 一级算数→二级算数 乘除算数
- (49) 乘除算数→*二级算数 乘除算数| /二级算数 乘除算数| NULL
- (50) 二级算数→项| (表达式语句)
- (51) 项→IDENTIFIER 项 sub| CONSTANT| -CONSTANT| CHAR|
STRING

- (52) 项 $\text{sub} \rightarrow (\text{参数引用列表}) | [\text{CONSTANT}] \text{数组或结构体} | . \text{IDENTIFIER} \text{数组或结构体} | \text{NULL}$
- (53) 参数引用列表 \rightarrow 项 参数引用列表 sub
- (54) 参数引用列表 $\text{sub} \rightarrow \text{NULL} | , \text{参数引用列表}$
- (55) 数组或结构体 $\rightarrow [\text{CONSTANT}] \text{数组或结构体} | . \text{IDENTIFIER} \text{数组或结构体} | \text{NULL}$
- (56) 类型说明 $\rightarrow \text{VOID} | \text{INT} | \text{FLOAT} | \text{CHAR} | \text{DOUBLE}$

3.2 词法分析

通过自动机对源程序内容进行词法分析，把单词从源文件中分离开来。区分保留字、标识符、界符、常数、字符与字符串，并将其分辨保存于对应的表格之中。同时将单词转化为机内表示，即 Token 序列进行保存以用于后续处理。

3.3 语法分析

采用 LL(1) 文法分析方法。通过读取 C 语言文法，生成文法对应的 First 集与 Follow 集，在生成 Select 集的同时填写预测分析表。对词法分析产生的 Token 序列进行语法分析，分析其语法的正确性，如有错误则给出简单的错误提示。在分析的同时执行必要的语义动作函数。

3.4 符号表

符号表中的每一项都包含名字、类型、类别等。每种类型都有对应的类型表，还有针对数组而设立的数组表，可以提供数组的相关信息。并且数组类型定义（实际是同定义数组变量一同进行的）、常数、换名变量、临时变量在符号表中都拥有相对的命名格式区分。实现了对变量相关信息的插入、查询功能

3.5 语义分析

通过一种语句一种结构的方式来表示，使得语义栈中的值尽可能的少穿插到语法分析中时，采取特定开头的元素来代表语义动作，并且通过一个函数进行分析执行对应的语义函数。制作了一个辅助工具可以将带有语义的文法自动生成出解析代表语义动作元素的函数。实现了生成四元式和更新符号表的功能

3.6 中间代码优化

为了使程序具有更高的时间和空间效率,对中间代码进行了等价变换的优化操作。本次课程设计完成的是在中间代码级上的局部优化,采用的方法为基于 DAG 的局部优化算法。通过构造基本块内优化的 DAG,重组四元式来对四元式进行优化。通过优化过程,实现了常值表达式节省,公共子表达式节省,并删除无用赋值四元式和实现了循环优化。

4 编译器前端设计

4.1 词法分析

4.1.1 功能

输入源程序文件名,输出对应 Token 序列,如图:

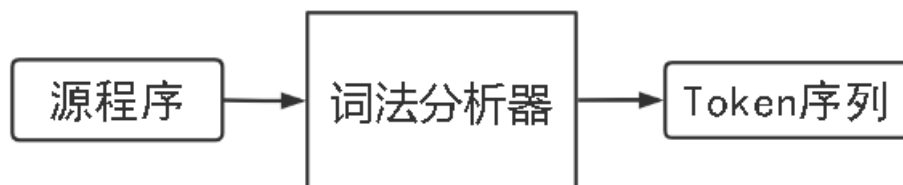


图 4.1.1-1

4.1.2 处理过程

首先从源程序读取内容,生成包含全部信息的字符串。对字符串进行预处理,去除注释语句。将字符串输入词法分析器的自动机中,通过状态转换判断词法,区分关键字、标识符、常数、界符、字符与字符串。记录并保存信息于相应的表内,并同时生成对应的 Token 序列。

(1) 关键字 : “include”,“logic”,“auto”,“break”,“function”,“case”等 37 个关键字,保存于 KT 表。

(2) 标识符 : 变量名,函数名等,保存于 iT 表。

(3) 常数 : 自然数,保存于 CT 表。

- (4) 界符：如“=”, “>”, “<”, “&”, “|”等 48 个界符，保存于 PT 表。
- (5) 字符：保存于 cT 表。
- (6) 字符串：保存于 sT 表。

4.1.3 数据结构

- (1) Token 结构

```
struct Token {    //Token 结构
    string Word;  //表名
    int Key;
};
```

- (2) 各类表结构

```
vector<string> iT;    //标识符表
vector<string> cT;    //字符表
vector<string> sT;    //字符串表
vector<string> CT;    //常数表
vector<string> KT;    //关键字表
vector<string> PT;    //界符表
```

4.1.4 流程图

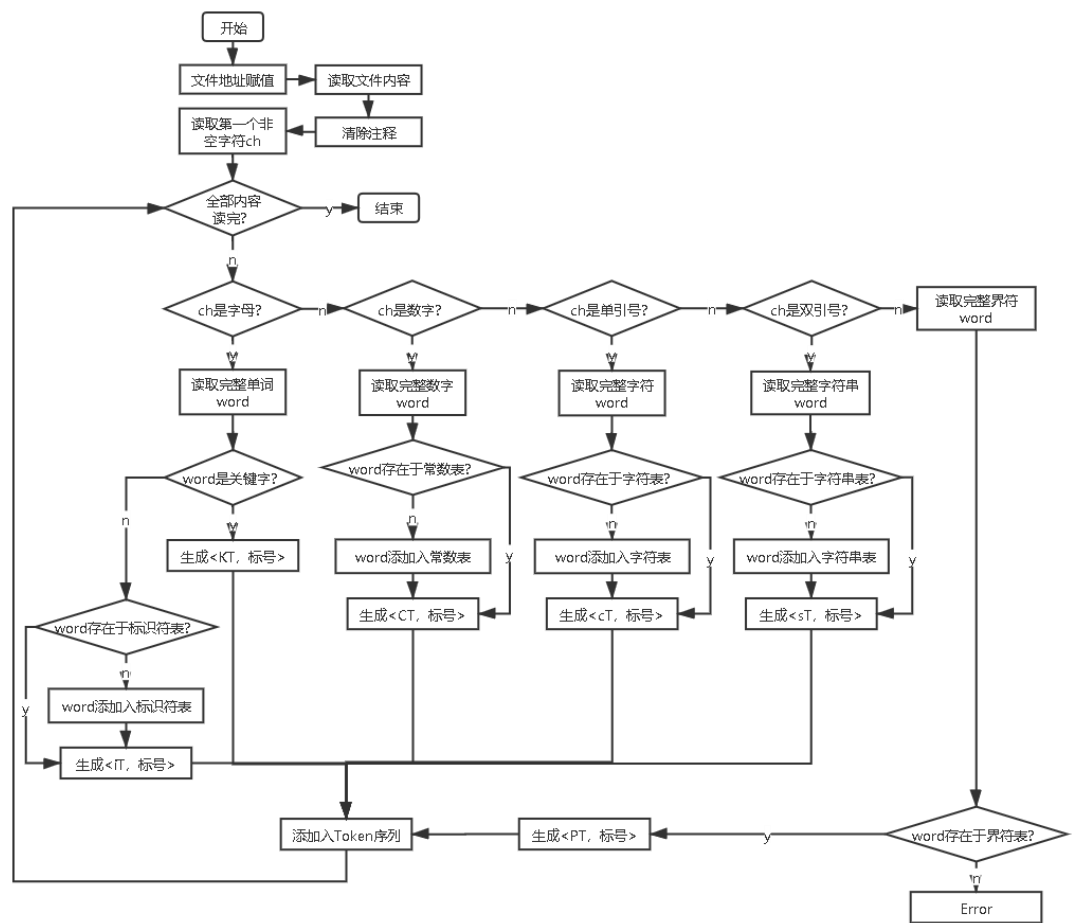


图 4.1.4-1

4.1.5 部分展示截图

```
Token:
Num: 39
<KT, 6> <IT, 1> <PT, 1> <CT, 1> <PT, 44> <KT, 37> <KT, 20> <IT, 2> <PT, 38> <PT, 39> <PT, 40> <KT, 20> <IT, 3> <PT, 1> <CT, 1> <PT, 44> <IT, 3> <PT, 1> <CT, 2> <PT, 44> <KT, 18> <PT, 38> <PT, 25> <KT, 2> <PT, 38> <IT, 3> <PT, 30> <IT, 1> <PT, 39> <PT, 39> <PT, 40> <KT, 24> <CT, 3> <PT, 44> <PT, 41> <KT, 24> <CT, 4> <PT, 44> <PT, 41>
```

图 4.1.5-1 Token 序列

4.2 语法分析

4.2.1 功能

生成文法所对应的 select 集和预测分析表，分析来自词法分析器的 Token

序列，判断语法的正确性。

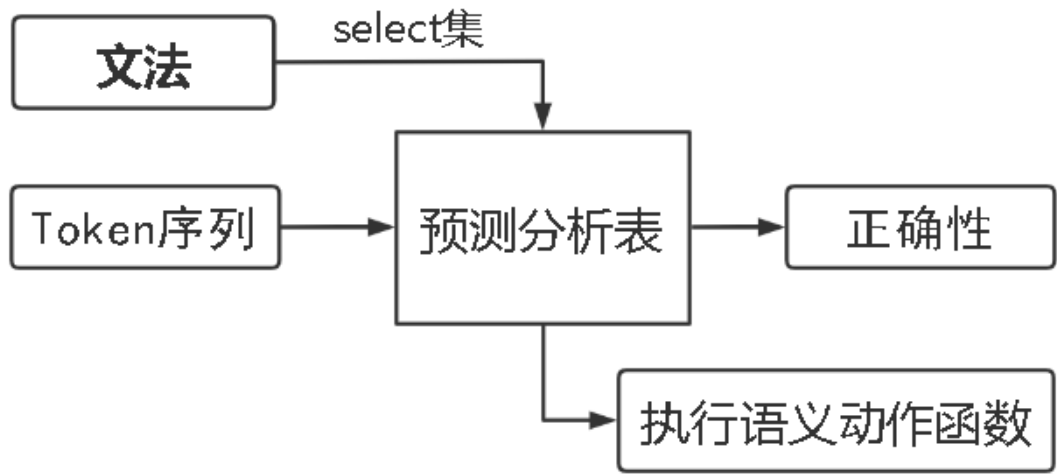


图 4. 2. 1-1

4.2.2 处理过程

(1) 预测分析表的生成

采用 LL(1) 文法分析方法。首先初始化文法，分析文法内容，遍历查找生成对应的 First 集与 Follow 集，并在生成 Select 集的同时填写预测分析表。

(2) 语义分析

对词法分析产生的 Token 序列进行语法分析，分析其语法的正确性，如有错误则给出简单的错误提示。在分析的同时执行必要的语义动作函数。

4.2.3 数据结构

(1) 预测分析表

```
int table[200][200]; //预测分析表
```

(2) first/follow 集

```
vector<int> first[200];  
vector<int> follow[200];
```

(3) 语义栈

```
stack<string> ST;
```

4.2.4 流程图

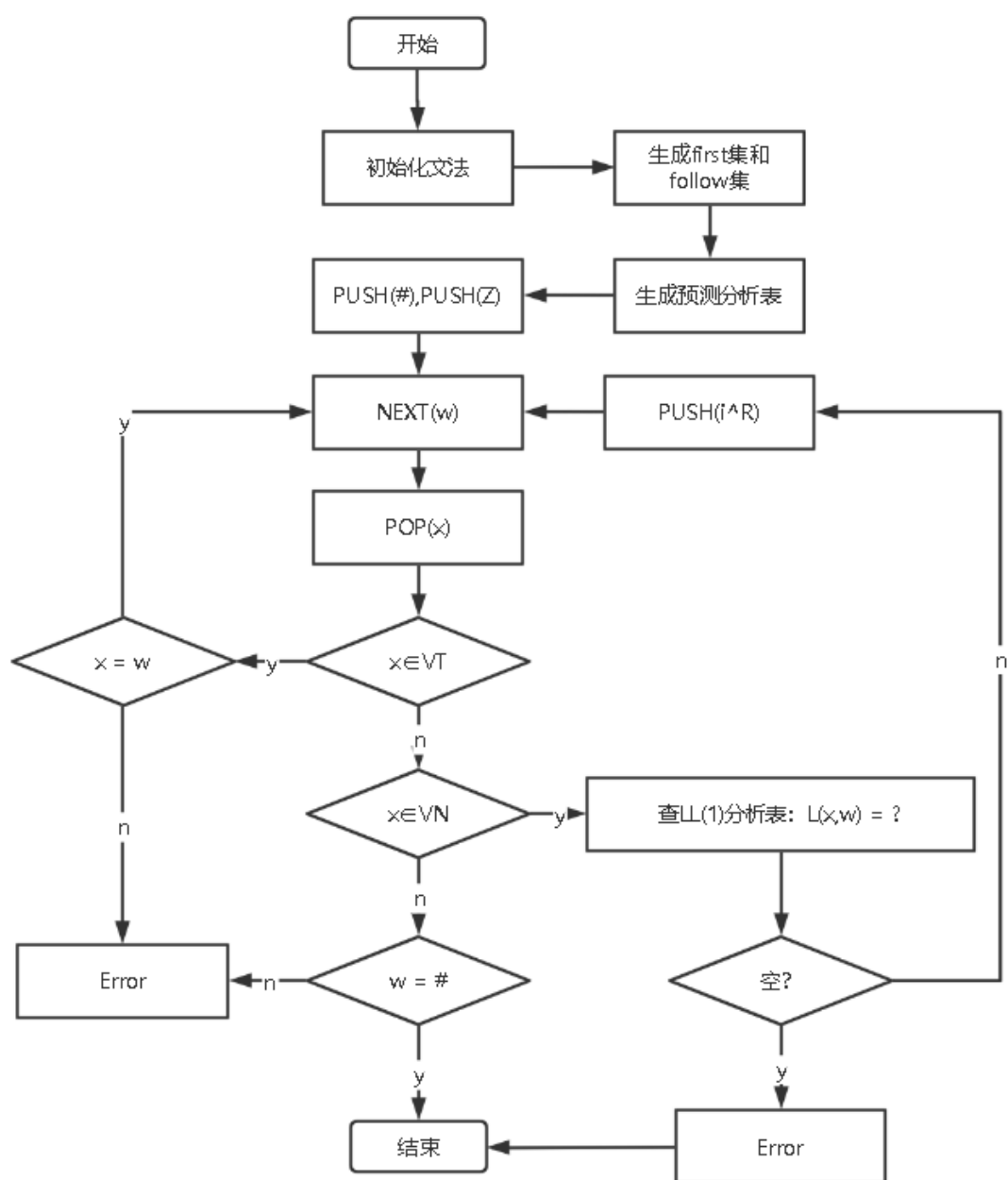


图 4.2.4-1

4.2.5 部分展示截图

```
Table:
No. 0:P--- FUNCTION # STRUCT VOID INT FLOAT CHAR DOUBLE
No. 1:Psub--- FUNCTION # STRUCT VOID INT FLOAT CHAR DOUBLE
No. 2:Psub--- END#
No. 3:ODC--- FUNCTION
No. 4:ODC--- # STRUCT VOID INT FLOAT CHAR DOUBLE
No. 5:FDF--- VOID INT FLOAT CHAR DOUBLE
No. 6:FDC--- IDENTIFIER
No. 7:FDCsub--- )
No. 8:FDCsub--- VOID INT FLOAT CHAR DOUBLE
No. 9:AL--- VOID INT FLOAT CHAR DOUBLE
No. 10:ALsub--- )
No. 11:ALsub--- ,
No. 12:ADC--- VOID INT FLOAT CHAR DOUBLE
No. 13:DCS--- #
No. 14:DCS--- VOID INT FLOAT CHAR DOUBLE
No. 15:DCS--- STRUCT
No. 16:HDC--- #
No. 17:FN--- IDENTIFIER
No. 18:FN--- STR
No. 19:VDC--- VOID INT FLOAT CHAR DOUBLE
No. 20:IDL--- IDENTIFIER
No. 21:IDLsub--- ;
No. 22:IDLsub--- ,
No. 23:IDC--- IDENTIFIER
No. 24:IDCsub--- , ;
No. 25:IDCsub--- =
No. 26:VNDC--- IDENTIFIER
No. 27:VEC--- , ; ) =
No. 28:VEC--- [
No. 29:V--- [
No. 30:Vsub--- , ; ) =
No. 31:Vsub--- [
No. 32:INIT--- IDENTIFIER CONSTANT CH STR -
No. 33:INIT--- {
No. 34:IL--- IDENTIFIER CONSTANT CH STR -
No. 35:ILsub--- }
No. 36:ILsub--- ,
No. 37:SDC--- STRUCT
No. 38:SDCsub--- ;
No. 39:SDCsub--- IDENTIFIER
No. 40:SML--- VOID INT FLOAT CHAR DOUBLE
No. 41:SMLsub--- }
No. 42:SMLsub--- VOID INT FLOAT CHAR DOUBLE
No. 43:CS--- {
No. 44:SL--- # STRUCT VOID INT FLOAT CHAR DOUBLE
No. 45:SL--- IDENTIFIER { IF WHILE DO FOR CONTINUE BREAK RETURN
No. 46:SLsub--- }
No. 47:SLsub--- IDENTIFIER # { STRUCT IF WHILE DO FOR CONTINUE BREAK RETURN VOID INT FLOAT CHAR DOUBLE
No. 48:S--- {
No. 49:S--- IF
No. 50:S--- WHILE DO FOR
No. 51:S--- CONTINUE BREAK RETURN
No. 52:S--- IDENTIFIER
```

图 4.2.5-1 select 集-1


```

No. 52:S--- IDENTIFIER
No. 53:CSS--- IF
No. 54:CSSsub--- IDENTIFIER # { } STRUCT IF WHILE DO FOR CONTINUE BREAK RETURN VOID INT FLOAT CHAR DOUBLE
No. 55:CSSsub--- ELSE
No. 56:TS--- WHILE
No. 57:TS--- DO
No. 58:TS--- FOR
No. 59:DF--- IDENTIFIER
No. 60:DF--- ;
No. 61:LE--- IDENTIFIER ( CONSTANT CH STR - LOGIC NOT_OP
No. 62:LEsub--- OR_OP
No. 63:LEsub--- ; )
No. 64:LA--- IDENTIFIER ( CONSTANT CH STR - LOGIC NOT_OP
No. 65:LASub--- AND_OP
No. 66:LASub--- ; ) OR_OP
No. 67:LN--- NOT_OP
No. 68:LN--- LOGIC
No. 69:LN--- IDENTIFIER ( CONSTANT CH STR -
No. 70:LNsub--- RE_OP
No. 71:LNsub--- <
No. 72:LNsub--- >
No. 73:LNsub--- ; ) OR_OP AND_OP
No. 74:AE--- IDENTIFIER
No. 75:AE--- )
No. 76:JS--- CONTINUE
No. 77:JS--- BREAK
No. 78:JS--- RETURN
No. 79:AS--- IDENTIFIER
No. 80:ES--- IDENTIFIER ( CONSTANT CH STR -
No. 81:A--- +
No. 82:A--- -
No. 83:A--- ; < > ) OR_OP AND_OP RE_OP
No. 84:T--- IDENTIFIER ( CONSTANT CH STR -
No. 85:B--- *
No. 86:B--- /
No. 87:B--- ; < > ) + - OR_OP AND_OP RE_OP
No. 88:F--- IDENTIFIER CONSTANT CH STR -
No. 89:F--- (
No. 90:I--- IDENTIFIER
No. 91:I--- CONSTANT
No. 92:I--- -
No. 93:I--- CH
No. 94:I--- STR
No. 95:Isub--- (
No. 96:Isub--- [
No. 97:Isub--- .
No. 98:Isub--- , ; < > } ) + - * / OR_OP AND_OP RE_OP
No. 99:PRL--- IDENTIFIER CONSTANT CH STR -
No. 100:PRLsub--- )
No. 101:PRLsub--- ,
No. 102:VOS--- [
No. 103:VOS--- .
No. 104:VOS--- , ; < > } ) + - * / OR_OP AND_OP RE_OP
No. 105:TYPE--- VOID
No. 106:TYPE--- INT

```

图 4.2.5-2 select 集-2

```

No. 107:TYPE--- FLOAT
No. 108:TYPE--- CHAR
No. 109:TYPE--- DOUBLE

```

图 4.2.5-3 select 集-3

```

Error! Expected: ")" or "*" or "+" or "-" or "/" or ";" or "<" or ">" or "AND_OP" or "OR_OP" or "RE_OP" behind "0"

```

图 4.2.5-4 语法分析错误提示

4.3 翻译文法与语义分析

4.3.1 变量定义及其初始化语句

变量的类型固定在变量名的前面，所以执行到变量类型时便可确定基本元素的类型，执行到变量名时无法确定其是否是数组还是单独的变量，所以只能先暂存。在之后若无数组定义标志[]则为单独变量，可以立即生成插入到符号表中，否则则是数组，需要暂存数值直到最后结束时才能确定数组的维数，则从最小维

度依次查询该数组类型，有则利用无则生成，最后才能确定数组变量的类型。至此变量定义完毕。之后若有初始化则按照赋值语句处理即可。

4.3.2 分支语句

分支语句 if else，分别记录下 if 后逻辑表达式语句、if 后的复合语句、else 后的复合语句，在结束时调整顺序，产生 if 对应的四元式即可。其中 else 即使不存在也不会影响最后处理，无需特判。

四元式格式：

逻辑表达式

(if, 逻辑表达式结果, _, 偏移量)

If 后的复合语句

[(e1, _, _, 偏移量)

else 后的复合语句

]

(ie, _, _, _)

4.3.3 循环语句

循环语句一共有三种 while() {} do{}while() for(;;) {}。选择将三种循环语句转化为一种相同的语句。把 do A while(B) 变成 A while(B) A。把 for(A;B;C) D 变成 A while(B) DC，所以可以产生同一个形式的四元式。

四元式格式：

(wh, _, _, _)

逻辑表达式

(do, 逻辑表达式结果, _, 偏移量)

复合语句

(we, _, _, 偏移量)

4.3.4 赋值语句

$A = B$ 。B 有多种类型，需要将 B 转化为同一种结构，之后生成四元式
(=, B, _, A)

4.3.5 函数调用语句

四元式格式

(param, entry(t1), _, _)

...

(call, entry(f), 参数格式, 返回值)

4.3.6 跳转语句

对于 continue 和 break 来说，第一次生成时只生成为

(jmpc, _, _, 1)

(jmpb, _, _, 1)

之后在所有四元式生成结束之后，才会将其偏移量进行修正。

return 则会生成

(ret, _, _, 返回值)

4.3.7 复合语句

将产生的语句都放入复合语句给后续处理。

4.4 符号表

4.4.1 总表结构

总表的每一项共有四个部分，分别是

name 名称

typ 类型 指向类型表

cat 种类 种类编码: f 函数, c 常量, t 类型, d 域名 (结构体定义),
v 变量 vf 赋值形参 vn 换名变量

addr 地址, 函数指向函数表, 类型、域名指向(int)长度, 常量直接指向(类型同 typ)值。变量、赋值变量指向相对位移量

4.4.2 类型表

类型表总共分为两部分 tav1 和 p 指针

tval i 整形 r 浮点型 c 字符型 a 数组型

p 指针, 基本类型(i r c)时为 NULL, 数组指向数组表

4.4.3 数组表

数组表有三部分

up 数组上界

ctp 成分类型指针

crlen 成分类型的长度

5 编译器后端设计

5.1 中间代码优化

5.1.1 功能

由程序直接转化得到的四元式不一定是最优的,为了使程序具有更高的时间和空间效率,对中间代码进行了等价变换的优化操作。本次课程设计完成的是在中间代码级上的局部优化,采用的方法为基于 DAG 的局部优化算法。通过构造基本块内优化的 DAG,重组四元式来对四元式进行优化。通过优化过程,实现了常值表达式节省,公共子表达式节省,并删除无用赋值四元式和循环优化。

5.1.2 数据结构

以结构体的形式存储四元式和 DAG 节点。四元式结构体的内容为操作符、操作数 1、操作数 2 和结果。DAG 节点结构体包含节点的编号、左、右孩子、操作符和标记，其中，为了可以存储多个标记，将存储标记的 mark 设计为 vector 类型，主标记放在开始位置以同副标记做区别。

四元式和 DAG 节点采用 vector 进行存储。为了能在 DAG 图中快速找到含相应标记的节点，以 map 类型对标记和其所在的节点编号进行存储，DAG 中的标记作为 key，其对应的节点编号作为 value。map 内部的实现自建一颗红黑树，这颗树具有对数据自动排序的功能，能够快速查找到映射结果。

```
struct quat_item{    //四元式
    string w;    //操作符
    string op1; //操作数 1
    string op2; //操作数 2
    string t;    //结果
};

struct dagNode {    //DAG 节点
    int num;        //编号
    int leftChild=-1; //左孩子
    int rightChild=-1; //右孩子
    string opt="";    //操作符
    vector<string> mark;//标记 主标记放在第一个
};

vector <quat_item> QUATS;    //四元式
vector <dagNode> DAG;    //dag 图
map <string,int> mapDag; //用于查找标记
```

5.1.3 算法

(1) 基本块划分算法

由于局部优化算法是以基本块为单位进行的,基本块也是目标代码生成的基本单位,

对基本块划分是必不可少的一部分。基本块是程序中一段顺序执行的语句序列,其中只有一个入口和一个出口。

具体算法如下:

根据基本块划分算法:

1. 找出基本块的入口语句,它们是:

- (1) 程序的第一个语句或 转向语句转移到的语句;
- (2) 紧跟在转向语句后面的语句。

2. 对每一入口语句,构造其所属的基本块:

- (1) 从该入口语句到另一入口语句之间的语句序列;
- (2) 从该入口语句到另一转移语句(或停止语句)之间的语句序列。


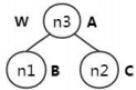

结合上述基本块的划分算法和设计的具体四元式,得到基本块的入口语句为:四元式①-④的下一条语句,四元式⑤-⑩的下一条语句及其能跳转到的语句。基本块即为位于入口语句到另一入口语句或者转移语句(或停止语句)之间的语句序列。

四元式:

- ①(call, entry(f), 参数格式, 返回值)
- ②(ret, _, _, 返回值)
- ③(pvall, _, _, _)
- ④(pvret, _, _, _)
- ⑤(if, 逻辑表达式结果, _, 偏移量)
- ⑥(e1, _, _, 偏移量)
- ⑦(do, 逻辑表达式结果, _, 偏移量)
- ⑧(we, _, _, 偏移量)
- ⑨(jmpc, _, _, 偏移量)
- ⑩(jmpb, _, _, 偏移量)

(2) DAG 构建算法

四元式在 DAG 中的节点表示：

语句	对应的四元式	DAG 表示
赋值	$(= B _ A)$ 或 $A = B$	
双目运算	$(\omega B C A)$ 或 $A = B \omega C$	
单目运算	$(\omega B _ A)$ 或 $A = \omega B$	

对于一些形式非常规的四元式，处理方式如下：

将四元式(if, 逻辑表达式结果, _, 偏移量)和四元式(do, 逻辑表达式结果, _, 偏移量)中的“if_偏移量”和“do_偏移量”近似处理为节点的运算符 ω ，生成的节点没有标记。

将数组下标“[]”作为运算符处理。

将四元式(ret, _, _, 返回值)中的“ret”做运算符处理，返回值做第一个操作数。

记录四元式(call, entry(f), 参数格式, 返回值)中的返回值，因为这里返回值以临时变量形式存在，有关四元式在四元式重组过程中可能被删除。

对于其他非常规的四元式，以叶子节点的形式直接插入到 DAG 图中，插入的作用是保持该四元式在四元式序列中原来的顺序，不具有优化作用。

DAG 具体构建算法：

【假设】(1) n_i 为已知结点号； n 为新结点号；

(2) 访问各结点信息时，按结点号逆序进行；

开始：DAG 置空，依次读取四元式。

若读取到的四元式为赋值四元式： $A=B$

① 在已构建的 DAG 中找 B 标记，若未找到，则创建一个主标记为 B 的节点；

② A 附加 B 上；

③ A 在其他节点已经定义过，删去这个节点当中的 A（主标记免删）。

若读取到的四元式为常值表达式： $A=C1 \omega C2$ 或 $A=\omega C1$

①计算常值 $C=C1 \omega C2$ （或 $C=\omega C1$ ）；

②修改四元式为赋值四元式 $A=C$ ，以添加赋值四元式的方式向 DAG 中添加节点。

若读取到的四元式为其他表达式 $A=B \omega C$ 或 $A=\omega B$ ；

① 在某节点存在公共表达式 $B \omega C$ 或 ωB ，则把 A 附加于此节点；

② 不存在公共表达式，申请新节点；

③若在某节点已经定义过 A，则删除此节点上的 A，主标记免删。

其中，对数组四元式进行特殊处理。由于需要先定义一个变量([, 数组名, 偏移量, #p_编号)，这个定义的四元式不能在优化的过程中被删除，那么当插入一个以 #p_编号做结果的四元式时，如果在构建好的 DAG 图中发现了变量 #p_编号，还需要判断是不是定义的节点，是的话免删。

注意：在优秀小组成果汇报时，发现该算法存在的一个问题。由于主标记免删，导致同一个标记在 DAG 图中可出现多次，后出现的标记相对于其前出现的同一个标记可能被附加在编号更小的节点上。但是四元式重组的过程中，按照编号由小到大进行新的四元式的生成，因而可能导致重组生成的四元式的顺序得到破坏，导致程序的错误。因而，在将标记 m 附到节点 n_i 的时候需要判断该节点后是否含有主标记为 m 的节点，如果有的话，则不能直接附到节点 n_i 上去，而应该重新申请节点。

（3）基于 DAG 重组四元式算法

【假设】(1) 临时变量的作用域是基本块内；

(2) 非临时变量的作用域也可以是基本块外。

开始：按结点编码顺序，依次读取每一结点 n_i 信息：

(1) 若 n_i 为带有附加标记的叶结点：

若 A_i 为副标记且为非临时变量，B 为主标记，则生成： $(=, B, _, A_i)$ ；

若 A_i 为副标记且为记录的函数返回值中的临时变量 #temp_i，则生成 $(=, \#temp_i, _, B)$ 。

(2) 若 ni 为带有附加标记的非叶节点:

设 A 为主标记, A_i 为副标记, B 和 C 或者 B 为其孩子节点, 则生成 (ω, B, c, A) 或者 $(\omega, B, _, A)$;

若 A_i 为非临时变量, 则生成 $(=, A, _, A_i)$;

若 A_i 为记录的函数返回值中临时变量 $\#temp_i$, 则生成 $(=, \#temp_i, _, A)$

另外处理:

(1) 将 ret 四元式的操作数 (返回值) 调整到最后。

(2) 将直接插入成为叶子节点的四元式恢复为回原来的四元式。

(3) 由于跳转语句的四元式的最后一项为偏移量, 在优化后偏移量可能会改变。调整方法为, 在优化过程中, 记录跳转语句跳转到的语句的新的位置, 优化完成后根据新位置更新偏移量。

5.1.4 流程图

(1) 主函数流程图

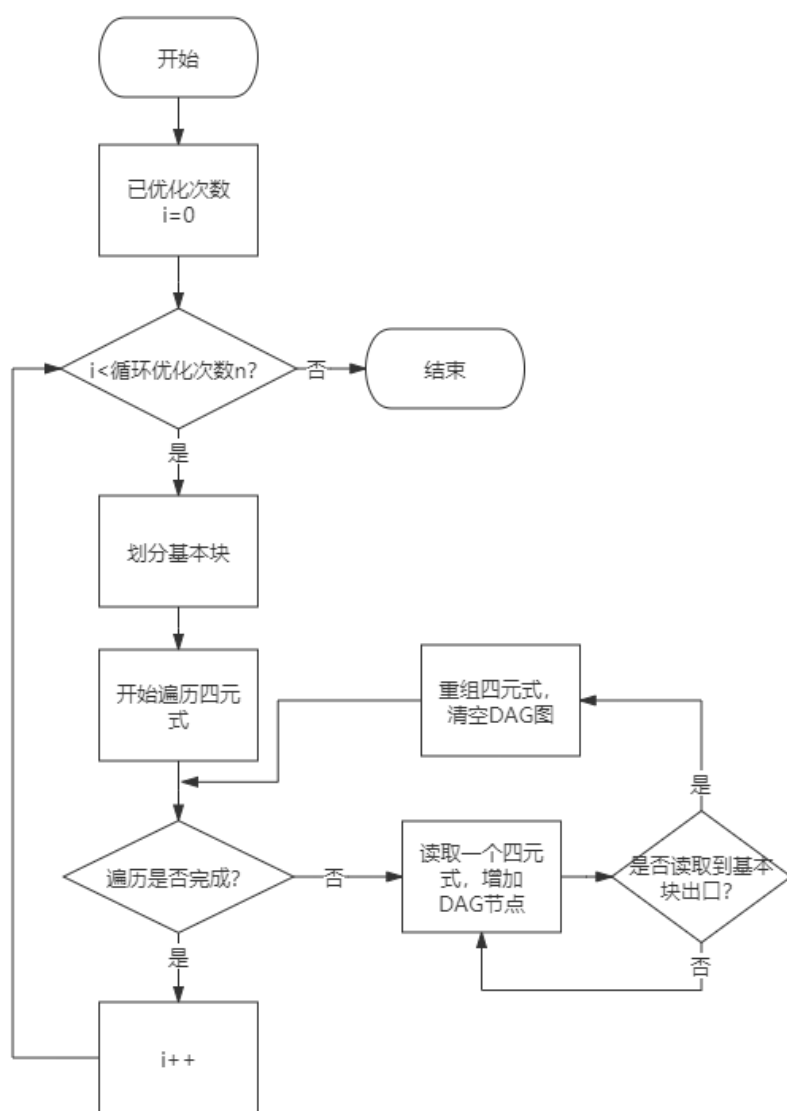


图 5.1.4.1 主函数流程图

(2) 构建赋值四元式的 DAG 节点流程图

四元式(=, B, _, A):

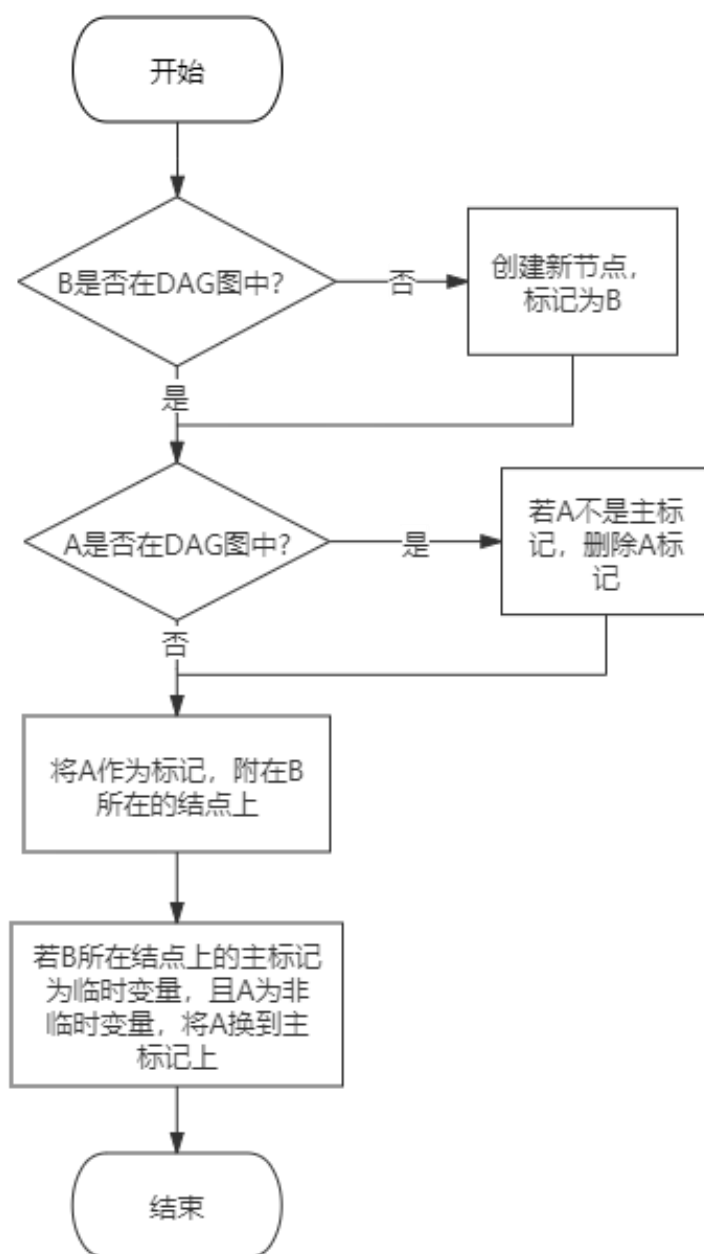


图 5.1.4.2 赋值四元式的 DAG 节点流程图

(3) 构建双目运算四元式的 DAG 节点流程图

四元式 (ω, a, b, A)

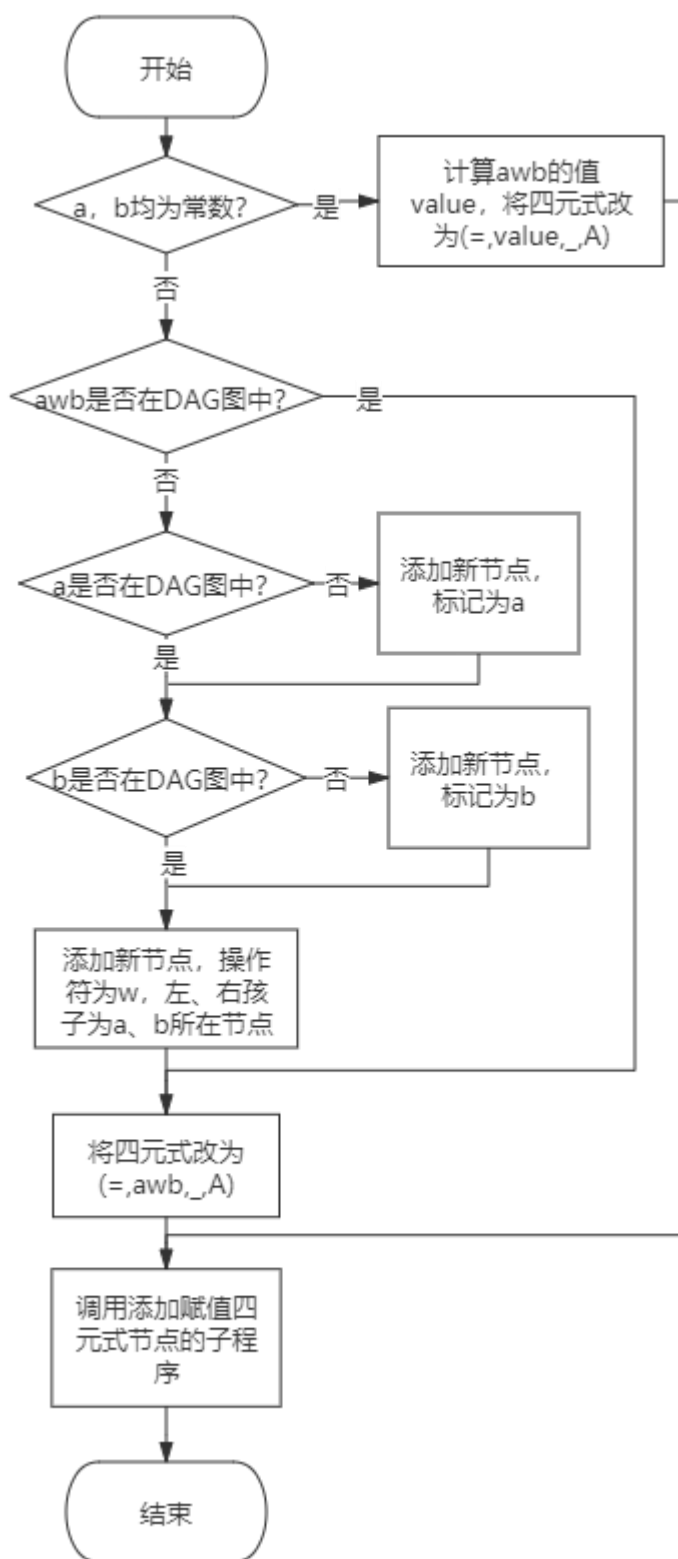


图 5.1.4.3 双目运算四元式的 DAG 节点流程图

(4) 构建单目运算四元式的 DAG 节点流程图

四元式 $(w, a, _, A)$

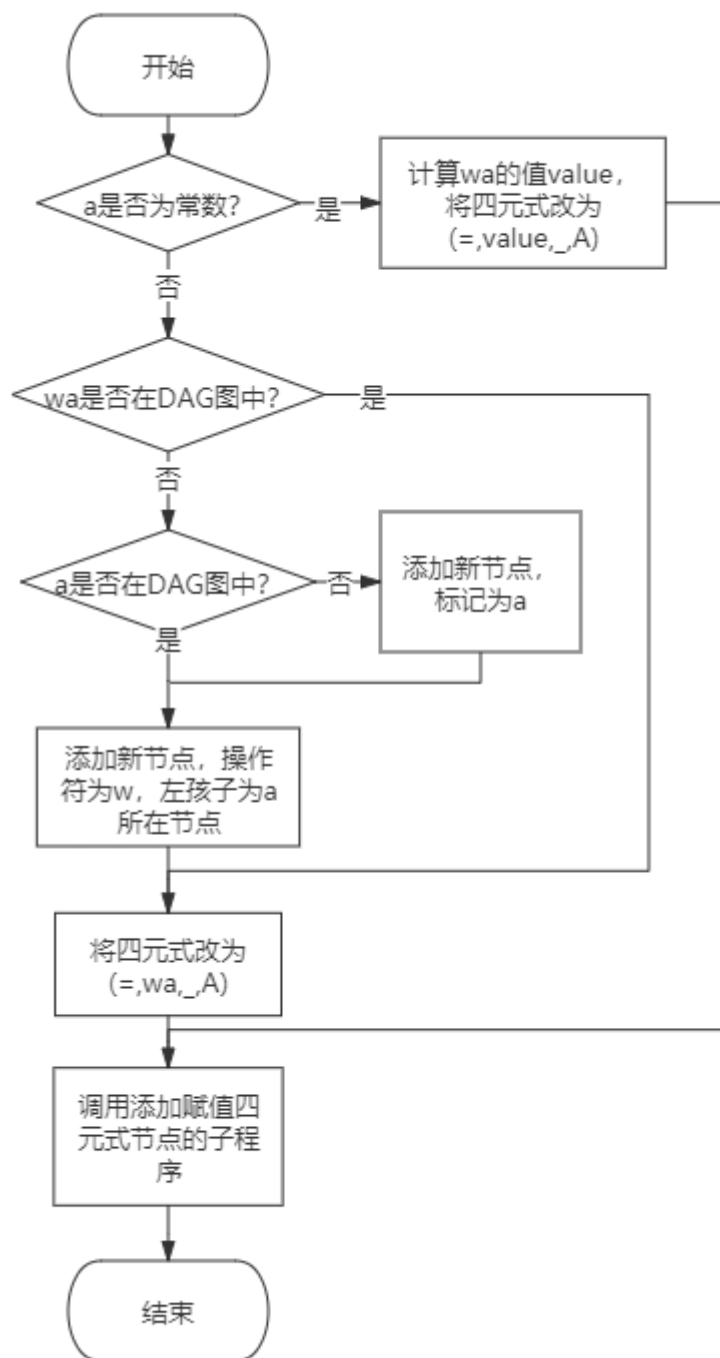


图 5.1.4.4 单目运算四元式 DAG 节点流程图

(5) 重组四元式流程图

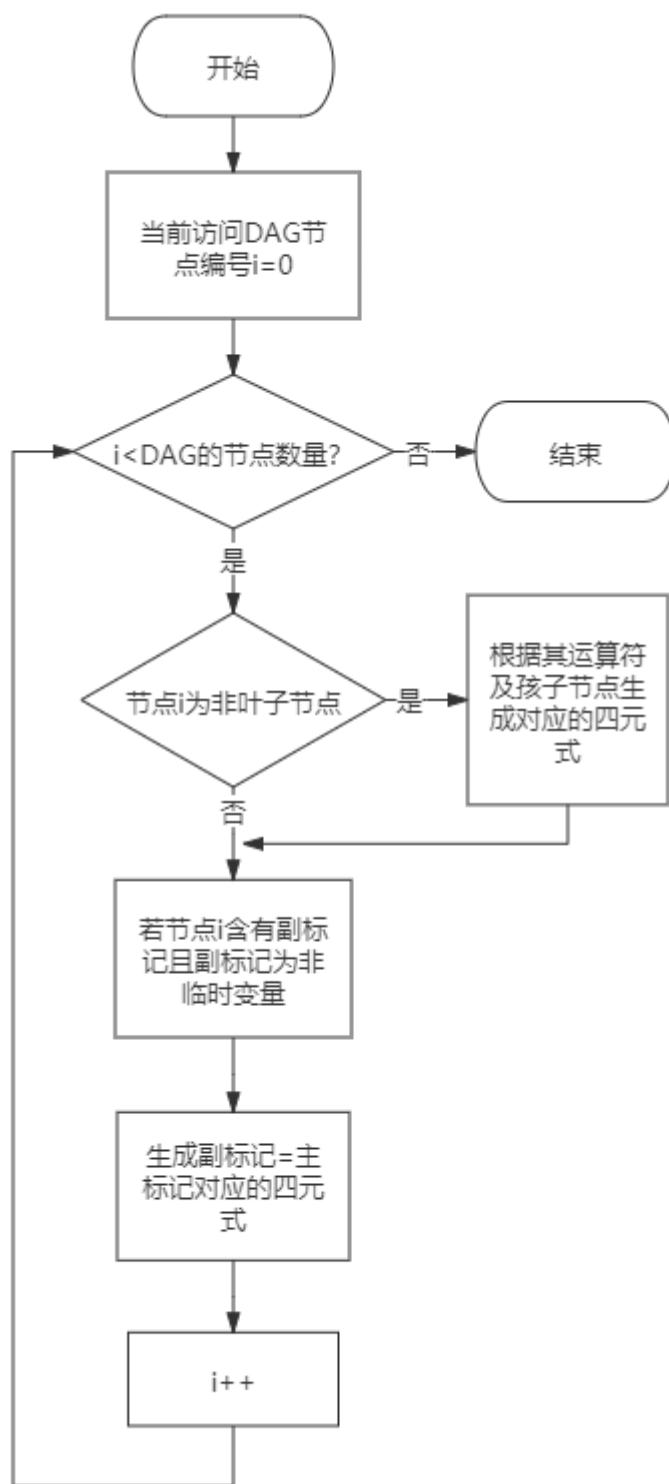


图 5.1.4.4 重组四元式流程图

5.1.5 优化截图:

优化前四元式序列:

```
(f_begin, _, _, main)
(pvall, _, _, _)
(=, #const_i_0, _, a)
(=, #const_i_5, _, b)
(=, #const_i_10, _, i)
(pvall, _, _, _)
(*, #const_i_10, #const_i_5, #temp_1)
(+, a, #temp_1, #temp_2)
(-, #temp_2, b, #temp_3)
(=, #temp_3, _, a)
(+, a, #const_i_50, #temp_4)
(-, #temp_4, #const_i_3, #temp_5)
(=, #temp_5, _, b)
(-, i, #const_i_1, #temp_6)
(=, #temp_6, _, i)
(pvret, _, _, _)
(wh, _, _, _)
(>, i, #const_i_0, #temp_7)
(do, #temp_7, _, 13)
(pvall, _, _, _)
(*, #const_i_10, #const_i_5, #temp_1)
(+, a, #temp_1, #temp_2)
(-, #temp_2, b, #temp_3)
(=, #temp_3, _, a)
(+, a, #const_i_50, #temp_4)
(-, #temp_4, #const_i_3, #temp_5)
(=, #temp_5, _, b)
(-, i, #const_i_1, #temp_6)
(=, #temp_6, _, i)
(pvret, _, _, _)
(we, _, _, -13)
(pvret, _, _, _)
(ret, _, _, #null_i_0)
(f_end, _, _, main)
```

优化后四元式序列:

```

-----
(f_begin, _, _, main)
(pvall, _, _, _)
(=, #const_i_0, _, a)
(=, #const_i_5, _, b)
(=, #const_i_10, _, i)
(pvall, _, _, _)
(+, a, #const_i_50, #temp_2)
(-, #temp_2, b, a)
(-, #temp_2, #const_i_3, b)
(-, i, #const_i_1, i)
(pvret, _, _, _)
(wh, _, _, _)
(>, i, #const_i_0, #temp_7)
(do, #temp_7, _, 8)
(pvall, _, _, _)
(+, a, #const_i_50, #temp_2)
(-, #temp_2, b, a)
(-, #temp_2, #const_i_3, b)
(-, i, #const_i_1, i)
(pvret, _, _, _)
(we, _, _, -8)
(pvret, _, _, _)
(ret, _, _, #null_i_0)
(f_end, _, _, main)

```

5.2 目标代码生成

采用了较为简单的目标代码生成方法。

- 1、 依次将每条中间代码变换成等价的若干条目标代码；
- 2、 基本块内考虑充分利用寄存器的问題。

即： 基本块内计算出的变量值尽量保留在寄存器中，直至寄存器另有用途或已到基本块的出口；引用变量时尽量用其在寄存器中的值。

为合理分配寄存器，需掌握寄存器的使用情况，

为此建立寄存器描述结构体 registers。

registers. Rvalue 表示寄存器内的值，在结构体数组 reg 下实现寻找值以及判断某值是否存在的函数。最终整合成功能函数 GETREG。

GETREG 过程：

- 1、 若在寄存器中搜寻到已经存储的值并且附加信息一致，或者为同标识符，则返回该寄存器
- 2、 若不存在满足上述条件的寄存器， 则选择某未分配空白的寄存器返回
- 3、 若不存在满足上述条件的寄存器， 则选择一个已分配的寄存器返回，选择条件为：

- (1) 寄存器中变量的值已在内存。
- (2) 寄存器中变量的值在块内的下一个引用点最远。

目标代码生成流程：

对基本块中每条代码（ $i : A := B \text{ OP } C$ ）执行：

- 1、调用 GETREG，此过程返回一个寄存器 R，用于存放 A 的值。
- 2、查 reg 数组。确定 B，C 的现行存放位置 B'，C'。
- 3、若 $B' \neq R$ 则生成 $LD \ R \ B'$
 $P \ R' \text{ OP } R \ C$
 否则生成 $OP \ R \ C'$
- 若 $B' = R$ 或 $C' = R$ 则从 reg 数组中删除 R。
- 4、令其在寄存器中更新相关的值。
- 5、若 B,C 在块内不再被引用，在块后不再活跃，且其现行值已经被保存中，则调用函数 clearNotUse 从寄存器中删除相关的值。

最终整合程序时因为内存溢出未添加至上交版本。

6 总结

本次课程设计我们完成了简单类 C 语言文法的设计，并实现了类 C 语言编译器的前端工作及优化工作。该类 C 语言文法支持数组、分支、循环、跳转、函数、复合等结构。经过反复的调整与测试，词法分析器可以正确的对源程序文件进行词法分析，生成正确的 Token 序列。C 语言编译系统采用 LL(1) 分析文法进行语法分析。通过分析，编译器可以分析 Token 序列语法的正确性，在出现错误时可以给予简单的错误提示。在进行语法分析的同时，插入语义动作函数，生成正确的四元式序列。优化工作也可正确的构造 DAG 并获得正确的优化结果。

本次课程设计中，发现有很多不足之处。首先，因为初期编译器的设计与工作安排，以及个人能力有限，编译器进完成了前端及优化部分，未能实现编译器后端工作中目标代码生成的工作。其次，因为复杂度原因，也未能实现更复杂的

文法的编译器的实现。总结分析后，发现了很多工作上的问题，将会在之后的学习工作中改正与提升。

7 参考文献

- 1、陈火旺.《程序设计语言编译原理》(第3版). 北京:国防工业出版社. 2000.

8 收获、体会和建议

8.1 邱晓鹏心得体会

这次课设中完成了符号表和部分的语义分析的设计。其中，部分的语义分析大体完成了其应有的功能，如函数、算术表达式、分支、数组等。在最开始的时候，语义分析的设计经历过一次巨大的思路改变。开始是将每一个元素都作为一个值放入语义栈，最后处理出四元式。经过和黄达同学的讨论，认为这种方法会导致语义栈中的元素过多，不利于分析，易出现错误且不易检查。后来我们决定采用了每一个语句一个结构来表示，将元素都放入同一个结构，减少语义栈内元素的数量。这个方法取得了较好的效果，大幅减少了错误分析过程中的难度。符号表的设计是比较遗憾的，有重大的失误。过多嵌套增大了后端的难度，直接导致了进度问题。经过课设之后，对编译的知识有了更深一步的理解，再次十分感谢对我帮助巨大的老师和同学们。

8.2 寇凯淇心得体会

在本次编译原理课程设计中，我负责的工作是词法分析器与语法分析器的设计与实现，并负责了一部分文法的设计与整理。在工作的过程之中，我面临的第一大难题就是文法的整理与正确性的分析。文法的设计可以说对整个C语言编译器的各个部分都有着或多或少的的影响，十分重要。所以在工作初期，我花费了大

量时间在对文法的正确性进行分析与测试的工作上。即便如此，文法中很多的错误仍然很难在最初的设计时发现与指出。很多的错误是在语法分析的过程中才发现的。这使得我再工作的过程中必须格外的仔细、耐心，才能在语法分析算法和文法的错误中准确地判断错误发生的地点与原因。这些工作也让我对编译原理的词法分析部分与语法分析部分有了更深的认识与理解。其次是 LL(1) 文法分析算法的设计也给我带来的很大的困难。正如之前所言，语法分析部分的 bug 中夹杂着部分文法的错误，导致进度的缓慢。通过这次的编译原理课设，加深了我对课堂知识的理解与掌握，也认识到团队合作在解决问题方面的重要性与必要性，团队的合作能够发挥一个团队最大的能力与功效。通过此次课设，我的耐心与毅力方面也有提高。在之后的学习生活中我也将继续牢记此次编译原理课设所带给我的启发，努力做得更出色。

8.3 张琳心得体会

这次编译原理课设确实是我目前做过的课设中最难的课设，由于平时没有很好地把课件上的内容理解透彻和串连起来，在完成课设的过程中也遇到了很多问题，但幸运的是最后都把问题解决了。对这次课设印象比较深的点就是过程式考核，虽然频繁的进度检查对于我们每个人都有很大的压力，但是也正是因为这种压力推动着我去有规划地完成自己负责的部分，学长们监督我们也很不容易。在这次课程设计中，我负责的是中间代码的优化。其实，一开始对优化过程不太熟悉，也并没有很明显的思路。课设进行的前两天，我主要看一些优化相关的资料，尽力去理解优化的实现过程。对优化过程有了比较清晰的认识之后，就开始设计优化过程中所需要的数据结构。在实现的过程中，确实也遇到了一些问题，比如数组四元式如何处理、如何区别临时变量和非临时变量。当我想不明白这些问题的时候，就向组内的同学寻求帮助，很快这些问题都得到了解决，在这里，感谢组内的同学！对于优化的实现过程，我采用的是由易到难的实现过程。先把常规的四元式的 DAG 的框架搭建出来，进一步在此基础上进行扩充和完善。最终，优化的功能模块得以成功实现。

另一个感触比较深的点就是在编译原理课设过程中，需要做好接口工作，如果存在接口问题功能模块的连接会出现问题，大家都很辛苦写自己负责的部分，

但可能会因为连接工作没做好导致部分功能没用上。

课设设计中的优秀报告会也很丰富，看到了三个优秀小组实现出来的成果，确实是通过付出了很多努力才完成的，功能也很完善。因为我对优化部分比较了解，印象最深的就是有个小组同学找出了课件上算法的漏洞。其实，我实现时也遇到了类型问题，但是并没有找到本质原因。所以，要多思考，才能有更进一步的发现。

最后真诚感谢为课设付出的老师们、学长学姐们还有组内的同学，我们一起努力完成了课设！

8.4 黄达心得体会

课程设计是培养学生综合运用所学知识，发现，提出，分析和解决实际问题，锻炼实践能力的重要环节，是对学生实际工作能力的具体训练和考察过程，随着科学技术发展的日新月异，编译原理已经在计算机应用中占据不少的部分，在各种编程语言的学习种可谓是无处不在。因此作为二十一世纪的大学生来说对编译原理有一定的了解是十分重要的。

历经了两周的课程设计，现在回想起这样繁忙的一段时间，我仍然感慨颇多。在课设之前的编译原理的学习中，我对相关知识的掌握并不透彻，在课设设计中对编译原理的知识掌握程度有极高的要求，所以在开始代码编写之前，先对当初的知识重新进行了学习。在一边学习一边构思代码实现的过程中，比起当初听课的浅尝即止，对相关知识有了完全不一样的理解。语义分析是编译过程的一个逻辑阶段，语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查，在整个程序的编译过程中占据了一个验证是否正确的阀门作用。在和同学一起对语义分析的实现当中我明白了编译程序是最实质性的工作，它第一次对源程序的语义作出解释，引起了源程序质的变化。而目标代码生成是编译程序最后一个阶段。在我们的编译器实现中它在优化的中间代码上进行，并将中间代码转化为等价的目标代码。在编写相关代码的时候让我对本来只浮现于 PPT 上的目标代码生成过程有了更深刻的理解。但是因为语言应用能力不足，编写的程序存在运行的 BUG，添加上级接口之后无法正常运行导致程序崩溃，最终未能很好地完成分配的任务。我会在未来的时间里常常回想这段日子，提醒自己在相

关工作或学习上更加注意。

感谢同组同学在课设设计过程中的合作与教导以及得知我负责部分无法正常上线之后并未生气反而一起陪我检查程序的接口，感谢班级指导学长每日对我们工作的认真检查，感谢编译教师组对编译课设题目的精心安排和课题指导。