课 程 设 计 报 告

设计题目：一个简单文法的编译器的设计与实现

班　　级：人工智能1902班

组长学号：20195063

组长姓名：黄元通

主　　笔：黄元通

设计时间：2021 年 7 月

设计分工

组长学号及姓名：20195063 黄元通

分工：

1. 文法：基本文法的设计，数组文法的设计与实现，复合语句的结构设计。
2. 词法分析：支持整数、小数、指数形式的常数处理机，未声明变量以及变量重复声明情况的筛查与报错，支持转义符处理和长度错误判断的字符型常量处理，词法分析器的封装与衔接。
3. 语法分析：变量声明及其标识符表填写模块，LL(1)实现的数学复合表达式模块，等式模块，判别式模块，支持多维任意长度的数组定义模块，变量访问，支持以左值或右值方式访问的数组元素访问模块。
4. 语义分析：符号表设计与实现，语法制导实现上述模块的语义分析、四元式生成以及符号表填写，语义栈逆波兰形式实现的表达式处理及数组元素虚拟寻址计算，活动记录。
5. 目标代码生成：目标代码指令集合定义、目标指令生成。

组员 1 学号及姓名：20195286 于之晟

分工：

1. 文法：for 语句模块的设计与实现。
2. 语法分析：for 语句模块的递归向下子程序法实现。
3. 语义分析：语法制导实现上述模块的语义分析、以及四元式生成。上述模块的翻译文法设计。
4. 优化：以中间代码四元式为对象进行的完备的基本块划分，常值表达式节省，公共子表达式节省，删除无用赋值，循环优化。

组员 2 学号及姓名：20195215 李欣宇

分工：

1. 文法：基本文法的实现、while 语句模块的设计与实现。
2. 语法分析：递归向下子程序法实现文法前半部分的语法分析，for 语句模块的递归向下子程序法实现。
3. 语义分析：语法制导实现前半部分文法以及 while 语句模块的语义分析、以及四元式生成，上述模块的翻译文法设计。
4. 目标代码生成：活跃信息的生成，活跃信息表的填写。

组员 3 学号及姓名：20195184 付大川

分工：

1. 文法： if 语句模块的设计与实现。
2. 词法分析：支持文件读取的词法分析器的整体设计与实现，关键字表、分隔符表、运算符表、类型表的初始化，实现整数常量、实数常量、字符型常量、字符串常量分别存储的常数表设计与实现，支持变量查找、添加的标识符表的初步填写，支持整数、小数、指数形式的常数处理机，Token 结构设计及 Token 的生成与存储机制。
3. 语法分析：if 语句模块的递归向下子程序法实现。
4. 语义分析：语法制导实现 if 语句模块的语义分析、以及四元式生成，上述模块的翻译文法设计。
5. 其他：数据输出及界面的可视化。

摘　要

本次课设的题目是通过编译器相关子系统的设计，掌握所学编译原理的基本理论和编译程序构造的基本方法和技巧，从而实现一个简单文法的编译器的设计与实现。

在这次课设中小组实现了一个简单文法的编译器的设计与实现，完成了基本的前端和后端功能设计，也附加了如识别多维数组等功能。

程序所有数据存储在自定义类 data 中，首先数据库的构造函数完成关键字 K 表、界符 P 表的构架及 Type 表的初始化。

编译器前端以语法分析进程为主线，综合使用递归向下子程序法和 LL(1)分析法结合来实现语法分析；辅以词法分析的 NEXTw()函数，并通过语法制导技术进行中间代码四元式的同步生成，在语法分析的同时通过插入语义动作，实现中间代码四元式序列的生成。

编译器后端采用 DAG 图的优化方式对初步生成的四元式进行优化；然后进行活跃信息填写，逆序遍历四元式，将得到的所有活跃信息表临时存储在 Active 中，以供目标指令生成时使用；最终依据四元式进行目标代码生成，根据四元式操作符的不同进行相应的目标指令生成，完成编译的实现。

关键词：编译原理，编译器前端，编译器后端，Token，递归向下，LL(1)分析法，算术表达式，四元式，多维数组，活跃信息，DAG，SYN 语法栈，SEM 语义栈

目　　录

1 概述

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生的创新能力及团队协作精神。

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的设计与实现。

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。

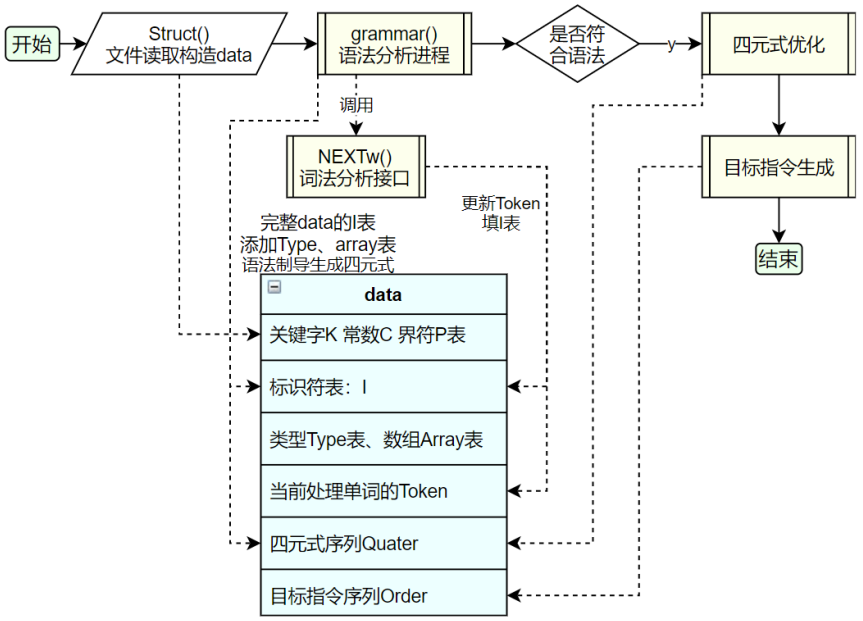
3 编译系统总体设计

3.1 编译器结构设计

3.1.1 程序进程示意图

编译器主体结构、及各进程间的关系，如下图 3.1-1 所示。

图 3.1-1 编译器主体结构及各进程示意图



3.1.2 主要模块

1. Data 类的对象 data，是所有数据的存储体。将其称为数据模块。
2. Morphology 类，是词法分析实现的函数及临时数据结合体，以“Data.txt” 为输入，分析出下一个完整的单词，并更新 data 中 Token 数据为该单词 Token。其功能封装为函数接口 NEXTw()。
3. Grammar 类的对象 grammar，是语法分析实现的函数及临时数据集合体，继承 Morphology 将 NEXTw()作为类成员，以实现词法、语法分析的同步进行；并使用语法制导技术同步生成四元式。将其称为前端功能模块。
4. Optimize 类，是实现四元式优化的函数及临时数据集合体，以 data 中的四元式序列为输入进行四元式优化，并将优化后四元式序列保存回 data 中。其功能封装为函数接口 optimize()。
5. Order 类的对象 order，是目标指令生成实现的函数及临时数据集合体，还包括中间步骤四元式的活跃信息表的构造与填写；继承 Optimize 将 optimize()作为类成员，从而一起组成编译器的后端功能模块。

3.1.3 主要过程

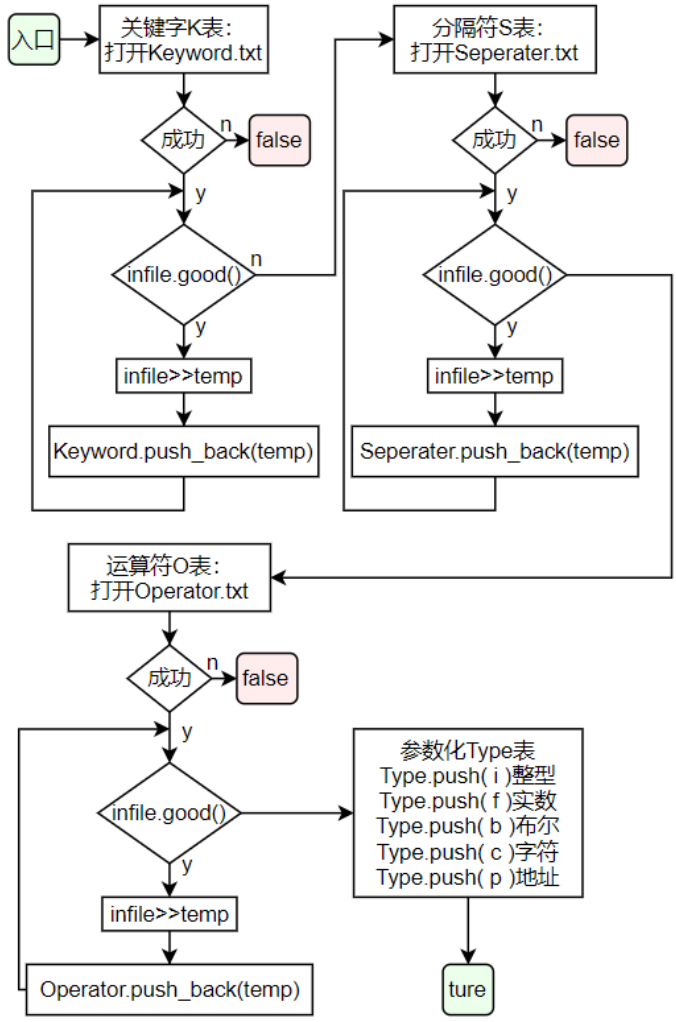
过程1　data 成员函数 Struct()构造完成关键字 K 表、界符 P 表（细分为操作符 O 表，分隔符 S 表），并初始化好 Type 类型表。

a) 类型表中前 5 个默认的数据类型分别为整型 int、浮点型（实数）float、布尔型 bool、字符型 char、地址型 p。

b) 该函数以 “Keyword.txt” 、“Operator.txt” 、“Separater.txt”三

个文件为输入，以文件读取形式构造 K 表、O 表、S 表，因此若有文件缺失，将在此步报错，并停止后续过程的启动。

Struct()函数流程图如下图所示：



过程2　编译器前端以语法分析进程为主线，辅以词法分析的NEXTw()函数，并通过语法制导技术进行中间代码四元式的同步生成。

a) NEXTw()主要通过更新 data 中的当前 Token，与语法分析进行衔接通信。

b) data 中的 I 表在 var{}中由词法分析添加，并填写 name 一栏，I 表的 type、cat、add 等其他几项由语法分析填写。

c) 语法分析结束后，四元式序列即全部生成，并保存在 data 的 Quarter 中。

过程3　若语法分析进程没有发现语法错误，则进行下一步的编译器后端工作。主要由四元式优化、活跃信息填写、目标指令生成三部分组成。

a) 四元式优化：以 data 中的四元式序列为输入，采用 DAG 图方案进行四元式优化，并将优化后的四元式序列保存回 data 中。

b) 活跃信息填写：逆序遍历四元式，将得到的所有活跃信息表临时存储在 Order:: stack <Active_Node> Active 中，以供目标指令生成时使用。

c) 目标指令生成：以 data 中的 Quarter 四元式序列、Order 中的 Active 活跃信息表临时数据为输入，根据四元式操作符的不同进行相应的目标指令生成，完成后 Active 栈将被清空以优化程序的空间利用，目标指令序列保存在 data 的 Order 中。

3.2 文法设计

文法主要部分使用递归向下子程序法，复合算术表达式部分文法主要使用 LL(1)分析法，辅以其他以条件判断等形式进行的文法处理方式。

3.2.1 文法及说明

1、递归向下子程序法部分的消去左递归后的文法如下：

```
grammar->"main"    "{" <Sub_Program> "}"
Sub_Program-> Var_Define Sentence_ Define
Var_Define ->"VAR {" Variable_Define "}"
Variable_Define-> { (Variable_Table | Array_Define) ";" }*
Variable_Table-> Type Identity "," Variable_Table | Identity
Type->"int" | "float" | "bool" | "char"
Array_Define-> Type "array" Identity "[" Math_Expression "]"
                    { "[" Math_Expression "]" }*
Sentence_Define -> { Equal_Sentence ";" | If | While | For| String}
Equal_Sentence-> Variable "=" Math_Expression
Compare_Define-> Compare_Sentence [ ω3 Compare_Sentence]
Compare_Sentence-> Math_Expression   ω4 Math_Expression
If->"if (" Compare") {" Sentence_ Define"}"
      [ "else {" Sentence_Define"}" ]
While->"while (" Compare")
            {" Sentence_ Define"}"
For->"for ("[Equal_Sentence]";" Compare ";" [Equal_Sentence]")
        {" Sentence_Define"}"
Variable-> cat=="v"&&( Type[type_p] =="i"||"f"||"b"||"c" )
```

其中:

ω_3 -> == | != |< | <= | > | >=

ω_4 -> && | ||

2、Math_Expression()即为使用 LL(1)分析法处理复合算术表达式的过程接口。算术表达式部分文法如下文本框 3.2-1 所示。转换后的 LL(1)文法、及各式 select 集如下文本框 3.2-1 所示。

文本框 3.2-1

```
E -> T { ω1T }
T -> F { ω2 F }
F -> i | ( E )
```

文本框 3.2-2

```
E -> TE′  ①  { i,( }
E′ -> ω1 T E′ ②  { ω1} | ε  ③  { ),# }
T -> FT′ ④  { i,( }
T′ -> ω2 F T′ ⑤  { ω2} | ε  ⑥  { ω1,),# }
F -> i  ⑦  { i } | (E)  ⑧  { ( }
```

其中： ω_1 -> + | - ； ω_2 -> * | /

3、关于文法的几点说明：

1. 只有在 VAR{}中才能进行变量的声明，有如下两种：
   a） 示例 1：float x,wd1234; 声明了 2 变量 x、wd1234，其类型都为 float 型。对于基本数据类型，可在一行声明一个或多个变量，用逗号 “,” 分隔，以分号 “;” 结束。
   b） 示例 2：bool array ar[10][2][5]; 声明了 1 个数组变量 ar，其元素的类型 bool 型，共有 3 维，各维的长度分别为：10、2、5。声明数组变量时，一行仅可声明一个变量，所声明的数组维度可为任意多维，各维长度需为大于 0 的正整数。
2. 若 VAR{}中出现重复定义的变量，由词法分析进程检测并报错。
3. 在后续代码中若出现未定声明的变量，由词法分析进程检测并报错。
4. 一个完整单词、一个完整数字、一个字符常量、一个字符串常量等基本字符级别的文法，由词法分析的扫描器一级统一处理；语法分析的基本单元为一个单词（Token）。

3.2.2 语句示例及说明

文法可实现的程序示例如下文本框 3.2-1 所示。该示例程序同样也作为后续测试及运行结果的 Data.txt 文件输入内容。

1. 字符型、布尔型允许作为左值与右值参与运算，字符型以字符的 ASCII 码参与运算，例如，int a; a =‘A’;则 a 的值为 65（字符‘A’的 ASCII 码为 65）。
2. 布尔型以 0(false)或 1(true) 参与运算。
3. 数组在 var{} 中声明，复合语句中数组元素可以左值或右值出现进行访问或修改。
4. 字符串可作为单独的一句在复合语句中出现，无法参与运算。

文本框 3.2-1 文法实现程序示例

```
main
{
   var
   {
      int i, x, y;
      char ch;
      bool bol;
      float array ar[10][5][2];
   }
   ch = 'A'; ch = 'A' + 2 ;
   bol = true;
   "This is a test";
   ar[0][0][0] = 1;
   for (i = 0; i <= 10; i = i + 2)
   {
      x = 10;
      ar[0][x - i][ i + ar[0][0][0] ] = 3.14e-1 * i;
      if (i >= 2 && i != 4)
      {
         while (ch <= 'Z' || bol == true)
         {
            ch = ch + bol;
            bol = false;
            }
            ch = 'A';
         }
         y = y + x + 4 + 5 - 0.1e+2 + 0.2e1;
      }
      x = 1 + 2;    y = 1 + 2;
}
```

3.3 符号表设计

3.3.1 概要

符号表在程序中的具体实现，即为 data 中的各个不同类型的 vector 动态数组，进行数据存储，如 3.1.1 程序进程示意图中的“编译器主体结构及各进程示意图 3.1-1”所示。

在启动编译器前端、编译器后端进程时，传入 Data &data 对数据存储集合引用，以实现在两个功能模块中各函数对数据库的修改。

符号表中共有以下 7 项：

1. vector <string> Keyword 关键字表
2. vector <string> Operator 运算符表、vector <char> Separater 分隔符表
3. vector <int> Constant_int 常量整数表、vector <float> Constant_float 常量浮点数表、vector <char> Constant_char 常量字符表、vector <string> Constant_string 常量字符串表
4. vector <Identity_Node> Identity 标识符表
5. vector <Type_Node> Type 类型表
6. vector <Array_Node> Array 数组信息表
7. 以及当前单词的一个 Token 信息，Token_Node Token

此外，还存储编译器后端所产生的 2 项数据：

1. vector <Quater_Node> Quarter 四元式序列
2. vector <Order_Node> Order 目标指令序列

3.3.2 主要符号表实现及说明

I 表 Identity

I 表 Identity 为符号表主表，存储各函数、自定义类型、变量的部分信息及其信息索引。其结构体、结构体数据命名及类型、各数据含义如下表 3.3-1 所示。

表 3.3-1 I 表结构体及信息说明

I 表结点 Identity_Node					
name	type_p	cat	add		active
			info	offset	
string 变量名	int 在类型表中的序号	string 种类	int 所属函数/表示 offset 存储长度	int 区距/长度	bool 变量当前活跃信息

对结构体 Identity_Node 做以下几点说明：

1、type_p 的实际实现问题：

其中 type_p 所实现功能为“指向 Type 表的指针”，但由于 vector 数组在自动转移所在内存空间时（vector 底层实现时，数组扩容时需要进行开辟新空间及原数据的复制），所有指向 vector 的指针、迭代器、引用都会失效，所以对于此采用 vector 存储的信息，只能以 int 型变量存储在 Type 表中序号（数组下标）的形式模拟实现指针的功能。

2、cat 可能值及含义：

f(函数)，c(常量)，t(类型)，d(域名)，v(变量)，vn(换名形参)，vf(赋值形参)。

3、active 的使用：

active 记录当前结点（变量）的活跃信息，在编译器后端功能模块中，填写活跃信息表时需要使用到。bool 类型的 active 为 true 时，表示该

变量活跃，为 false 时为不活跃。

4、add 的具体说明：

add 一项为一个 I_Address_Node 结构体，包含两项：int info、int offset。

a) 若 I 表该结点为函数名或变量，则 info = 所在函数的编号（编号为 0 至 n 的正整数），offset 存储该函数运行所需总空间（结点为函数名时）或该变量在活动记录中的 offset（结点为变量时）。

b) 若 I 表该结点为自定义类型的类型名时，则 info=-1，offset 存储该类型的长度。

从而实现利用同一个 add，实现以下多项功能：

a) 实现所有函数的临时变量都存储在该表中：若有不同函数中的局部变量名字相同，只需在查找时多加一项 add.info 的比较，若 add.info 为当前函数对应编号且 name 为查找值，才确定其 Token。从而解决了不同函数中的局部变量可使用相同 name 的冲突性。

b) 实现了存储数据为类型长度还是变量 offset 的区别：只需判断 add.info 是否为-1，是，则 offset 存储的为长度；否，则 offset 存储的是区距。

Type 类型表

Type 类型表存储各数据类型。对于当前文法，只加入了自定义数组类型功能，未加入自定义结构体类型的文法，因此 Type 表精简为 2 项。

第一项 string name 为代码，类型代码： i(整型 int)，f(实型 float)，b(布尔型 bool)，c(字符型 char)，a(数组型 array)，p(指针型/地址型)。int、float 占 4 字节，bool、char 占 1 字节，指针/地址占 2 字节。

第二项 int array_p 为指向 Array 数组表的虚拟指针。若该结点类型代码为 i、f、c、b、p，则该项为-1；若该结点类型代码为 a，则该项为其所指向的 Array 表元素的序号（下表）。

Type 表结构体、结构体数据命名及类型、各数据含义如下表 3.3-2 所示。

表 3.3-2 Type 表结构体

Type 表结点 Type_Node	
name	array_p
string 类型编码	int 在数组表中的序号

表 3.3-3 Array 表结构体

Array 表结点 Array_Node			
type_p	total_space	dimension	dimension_length
int 类型表中序号	int 该数组结构总空间大小	int 数组维度	vector <int> 各维度长度

Array 数组结构信息表

Array 表结构体、结构体数据命名及类型、各数据含义如上表 3.3-3 所示。

第一项 int type_p 同 I 表中的 type_p，为指向 Type 表的虚拟指针。

第二项 int total_space 为该数组所占总空间（单位为 Bit 字节），其计算公式为：type_p 指向类型所占大小*Di（ Di 为各维长度，$i \in (1, dimension)$ ）。

dimension 数组维度在 var{}声明语句中声明时确定，例如 int array ar[10][2][5][2]声明的数组维度为 4，其 dimension_length 数组长度为 4，所有元素分别为{ 10 , 2 , 5 , 2 }。

从而实现了一个 Array 结点存储一个数组结构所有信息，无需采用 Array 表中递归定义的数组信息存储方式。

3.4 数据结构设计

1、Data 类：除以上介绍过的数据，还有以下 4 个结构体设计：

Token 结构体

```
struct Token_Node
{
    string Type = "#";
    int Place = -1;
};
```

目标指令结构体

```
struct Order_Node
{
    string operation;
    string rgst;
    string address;
};
```

目标指令结构体

```
struct Order_Node
{
    string operation;
    string rgst;
    string address;
};
```

四元式结构体

```
struct Quater_Node
{
    string operation;
    string object1;
    string object2;
    string result; };
```

2、Order 类：还有以下 2 个结构体：

活跃信息表节点

```
struct Active_Node
{
    bool object1 = false;
    bool object2 = false;
    bool result = false;
};
```

寄存器描述表节点

```
struct RDL_Node
{
    string name = "";
    bool active = false;
};
```

4 编译器前端设计

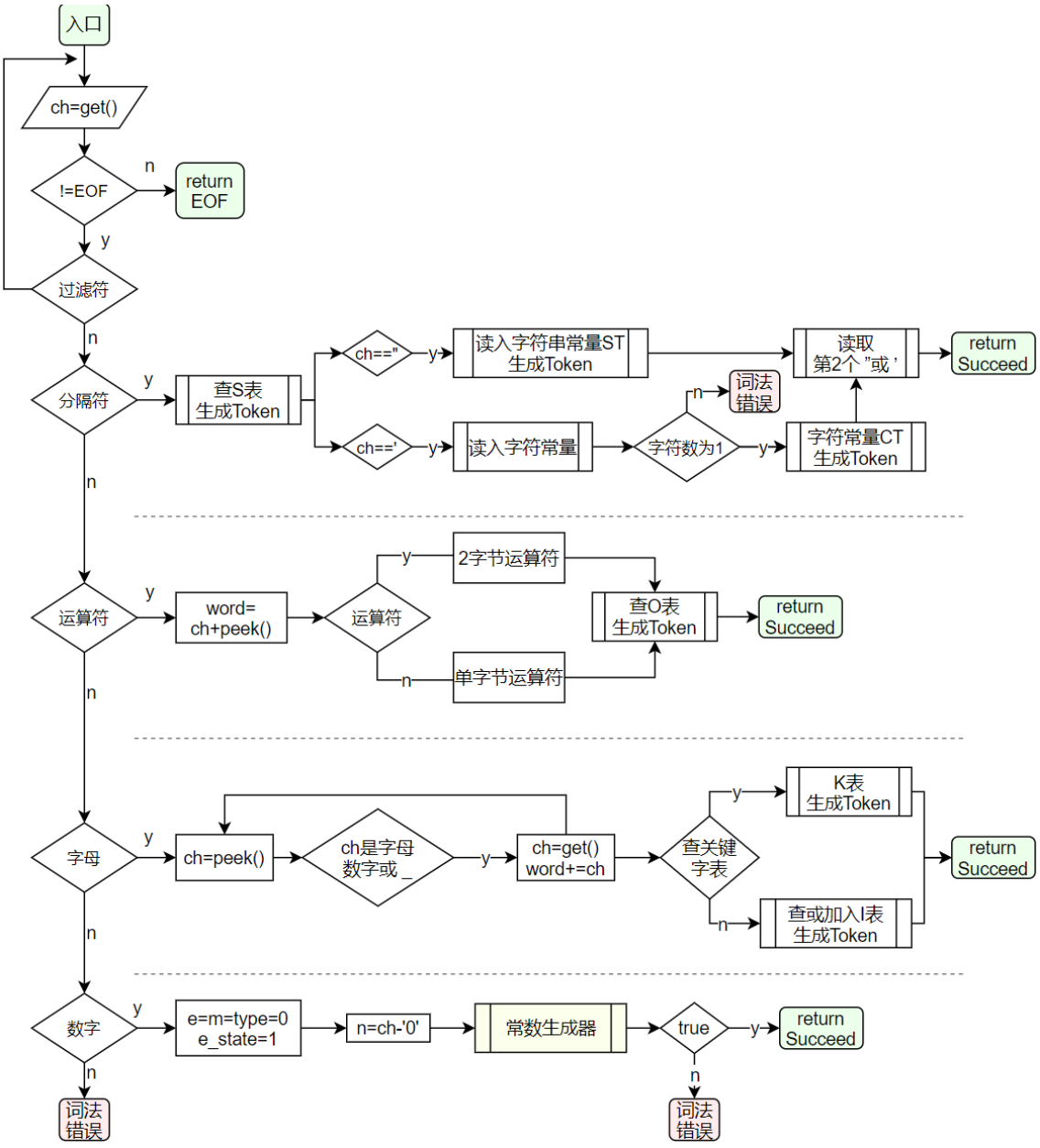
4.1 扫描器

4.1.1 功能

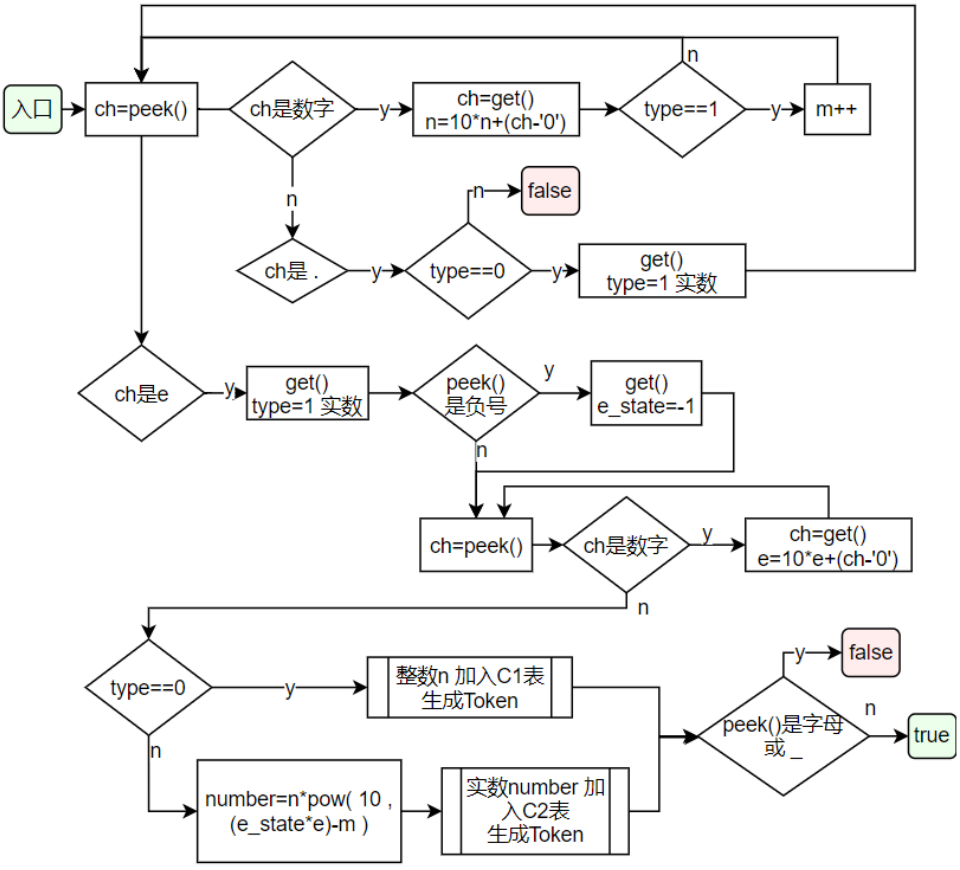
1. 识别关键字、运算符、分隔符、常量、标识符
2. 支持小数、科学计数法表示实数的常数处理器
3. 支持整数常量和实数常量分别单独保存的常量表 CI、CF
4. 支持字符常量和字符串常量分别单独保存的常量表 CT、ST
5. 标识符读取同时完成 I 表中 name 一项填写
6. 支持重复声明、未声明变量检测的标识符读取
7. 词法错误判断功能。如：
   a） 一对单引号 ’’ 间出现多个字符。如 ’ab’
   b） 实数以小数点开头。如 .1
   c） 同一个实数有多个小数点。如 1.22.2
   d） 一个实数整数或实数后紧接出现关键字或标识符。如 3.1e-2y
   e） 尚未定义的字符。如‘￥’
8. 支持单引号 ’’ 间\n、\r、\t、\\、\’ 等以表示转义符号的使用。如 ’\’’ 表示字符常量 ’
9. 类实现数据存储，可拓展性强

4.1.2 流程图

1、词法分析 NEXTw 函数流程图：



2、常数处理器流程图：



4.1.3 函数及具体实现

1. 词法分析整体封装成为语法分析的步骤中的函数 NEXTw()函数，语法分析作为词法分析的附属函数两过程同时进行。在语法分析中每调用一次 NEXTw()，data.Token 便被更新为下一个单词的 Token，从而实现两模块的衔接及数据传递。
2. 分隔符识别函数：IsSeparater(char ch)
   a) 分隔符以 char 型 vector 数组形式存储在 data.Separater 中，对于词法分析当前处理的字符 ch，函数进行 data.Separater[i]的遍历。

b） 若 data.Separater[i] == ch 则 ch 为分隔符，data.Token ={“S”, i}，函数返回 i（!= NotFind，NotFind 为宏定义的 int，值为-2）

c） 否则 ch 不是分隔符，返回 NotFind

3. 运算符识别函数：IsOperator (char ch)

a） 运算符以 string 型 vector 数组形式存储在 data.Operator 中，对于词法分析当前处理的字符 ch：

i. if ((ch == '<' || ch == '=' || ch == '>' || ch == '!') && infile.peek() == '=')，则下一个单词为 “<=、==、>=、!=” 之一

ii. if(ch == '+' && (infile.peek() == '+' || infile.peek() == '=')，则下一个单词为 “++、+=” 之一

iii. if (ch == '-' && (infile.peek() == '-' || infile.peek() == '=')，则下一个单词为 “--、-=” 之一

iv. if (ch == '&' && (infile.peek() == '&'))，则下一个单词为 “&&”

v. if (ch == '|' && (infile.peek() == '|')) ，则下一个单词为 “||”

b） 若符合上述 5 种情况之一，则 string word = ch + infile.get()；否则 word = ch。

c） 然后进行 data. Operator [i]的遍历。

d） 若 data. Operator [i] == word 则 word 为分隔符，data.Token ={“O”, i}，函数返回 i

e） 否则 word 不是分隔符，返回 NotFind

4. 常量识别函数：IsConstant_xxx( )

a） IsConstant_int(int number)、IsConstant_float(float number)、IsConstant_char(char ch)、IsConstant_string(string word)分别处

理整型 int、浮点型（实数）float、字符型 char、字符串型常量。

b） 对于传入函数的形参，各函数分别在 data 中其所处理数据类型所对应的：CI 表 vector <int> Constant_int、CF 表 vector <float> Constant_float、CC 表 vector <char> Constant_char、CS 表 vector <string> Constant_string 中遍历。

c） 若找到，则 data.Token ={“CI/ CF/ CC/ CS”, i}，函数返回 i

d） 否则未查找到该形参，则将该形参添加到 data 的对应 vector 中，data.Token ={“CI/ CF/ CC/ CS”, i}，i 为更新后该表最后一个元素序号，并返回 i

5. 字符常量、字符串常量处理：

a） 在词法分析的主流程函数 NEXTw 中继续判断，若 ch == '\''，则其后必为字符型常量，操作如下：

i. ch = infile.get()，如果 ch != '\''，则表示 2 个单引号间有内容，IsConstant_char(ch)，即完成了一个字符常量的识别及其 Token 生成

ii. 再 ch = infile.get()，若 ch != '\''，则表示 2 个单引号间有超过 2 个字符的内容，将此情况在词法分析阶段判断为错误，输出提示“字符表达式输出超过一个字符”，NEXTw 返回 Wrong（Wrong 为宏定义的 int，值为-1）

b） 在词法分析的主流程函数 NEXTw 中继续判断，若 ch == '\''，则其后必为字符型常量，操作如下：

i. while (infile.peek() != ' ” '），ch = infile.get()，word += ch

ii. IsConstant_string(word)，即完成了一个字符常量的识别及

其 Token 生成

iii. 最后 infile >> ch，读取掉第二个双引号"

6. ch 为数字时：

a) 下一个单词必定为数字常量，将 ch 传给常数处理器 Number_Generator(char ch)，由 Number_Generator()读取完整的数字，并根据其形式的不同，分别调用 IsConstant_int(int number)或 IsConstant_float(float number)函数。

b) 出现小数点、以科学计数法表示的默认取为 float 型，否则默认为 int 型。

7. ch 为字母时：

a) 下一个单词必定为关键字或标识符。

b) 读取完一个完整的单词 word 后（单词的首字母后，可为字母、数字、下划线的任意组合），先调用 IsKeyword(word)遍历 data 中的关键字表 vector <string> Keyword。

c) IsKeyword 若找到 data.Keyword[i] == word 则返回 i，data.Token ={“K”, i}，否则返回 NotFind。

d) 若 IsKeyword 返回的是 NotFind，则 word 必为标识符，调用 IsIdentity(string word)函数遍历 data 中的标识符表 vector <string> Identity。

e) IsIdentity 若找到 data.Identity[i].name == word，重复声明检测通过后（重复声明检测的实现在接下来的第 8 条中），data.Token ={“I”, i}，并返回 i

f) 若没有找到，则先进行未声明变量使用检测（未声明变量使用检测的实现在接下来的第 8 条中），通过后即可将该标识符添

加到 data.Identity 中，data.Token ={“I”, i}，i 为更新后该表最后一个元素序号，并返回 i

8. 重复声明、未声明变量检测的实现：

a） 在 IsIdentity()函数中，检测使用到：flag_var，为语法分析进程定义的标记变量，标记“当前是否进行变量声明”。true 表示当前语法分析进行到 var{}中，只能进行变量声明；false 表示当语法分析进行到复合语句中，只能进行变量使用。

b） 重复声明检测：

如找到 data.Identity[i].name == word，进行标记变量 flag_var 的判断，若为 true，表示正在 var{}中进行变量声明。则该变量以声明过，进行"变量重复声明"的显示，并返回 Wrong；否则正常进行后续程序。

c） 未定义变量使用检测：

如未找到 data.Identity[i].name == word，进行标记变量 flag_var 的判断，若为 false，表示在复合语句中进行变量的使用。则该变量未声明，进行"使用未声明的变量"的显示，并返回 Wrong；否则正常进行后续程序。

4.2 语法分析

1、采用方法

语法分析时主要按照递归向下子程序法运行过程分析，复合算术表达式部分文法主要使用 LL(1)分析法过程进行分析，辅以其他以条件判断等形式进行的语法分析。

2、语法分析主过程

读取字符串进行词法分析后，进行语法分析

识别语法起始部分“main”,判断语法是否有误，错误就立刻停止

判断下一个是否是“{”

进入主程序

判断下一句是否符合

<Sub_Program> -> <Type_Define><Var_Define><Sentence_ Define>

符合进入下一步

判断语句是否满足<Var_Define> -> "VAR {" <Variable_Define> "}"

语句满足进行<Var_Define> -> "VAR {" <Variable_Define> "}"分析，并进行<Variable_Define>部分的语法分析

判断语句是否满足

<Sentence_ Define>->{<Equal_Sentence>";" |<If>|<While>|<For>}

根据语句满足的类型分别进行<Equal_Sentence>，

If->”if (“ Compare_Define”) {“ Sentence_ Define”}”，

While->”while (“ Compare_Define”) {“ Sentence_ Define”}”

For->”for (“ Equal_Sentence Compare_Define Equal_Sentence”) {“ Sentence_ Define”}”分析

如果遇到不等式，需要与<Equal_Sentence> -> <Math_Expression> ω3 <Math_Expression>进行语法匹配分析，并进行<Compare_Define> -> <Compare_Sentence> {ω4 <Compare_Sentence>}和算术表达式分析

分析完毕遇到“}”时，语法分析完成。

注：在语法分析过程中如果 return false 即代表语法分析出错

伪代码实现示例：

NEXTw();//读取字符

```
//开始语法分析(递归向下子程序法)
//"main"
if (Token_Search_K_O_S() != "main") flag = false; if (flag == false) return flag;
Add_Quater("start", "main", "", "");
Identity_Node temp;
temp.name = "main"; temp.cat = "f";
temp.add.info = level; data.Identity.push_back(temp);
NEXTw();
//"{"
if (Token_Search_K_O_S() != "{") flag = false; if (flag == false) return flag;
//cout << "进入main函数  {\n";
NEXTw();
//Sub_Program()分程序
if (Sub_Program() == false) return flag;
//"}"
if (Token_Search_K_O_S() != "}") flag = false; if (flag == false) return flag;
//cout << "} main函数结束\n\n";
Add_Quater("end", "main", "", "");
      return flag;
```

3、四个附属函数的说明

string Token_Search_K_O_S()：根据当前 Token（Token.Type = K、O、S 表的情况下），返回该 Token 所对应的 name。

void Make_T_Variable(string &name,char type)：生成一个临时变量并添加到 Identity_Temp 临时 I 表中，其类型指向 type，并将传入的 name 赋值为生成的该变量的 name。

bool Find_Identity(int place, string& location)：根据传入的 place 找到其在 I 表所对应的 Identity 结点，并构造在在四元式中出现的 name（或结点为地址型变量时构造虚拟地址）。结点为普通变量时 location = name，结点为数组变量名时，四元式构造方式参见 4.3.5 数组元素访问详细说明 的详细说明。

int Add_Quater(string operation, string object1, string object2, string result) 将新四元式压入 data 四元式序列，并返回该四元式对应序号（下表）。

4.3 翻译文法及语义分析

程序在语法分析进程中，中间文法、语义分析同步进行，结束后即完成所有四元式生成，因此，按照程序实现时的逻辑，将此两部分合并在一起介绍。

4.3.1 中间文法形式

1、 中间文法采用四元式，基本形式为(ω O_1 O_2 t)，各项分别为：算符，对象 1，对象 2，结果。

2、 四元式翻译法则如下：

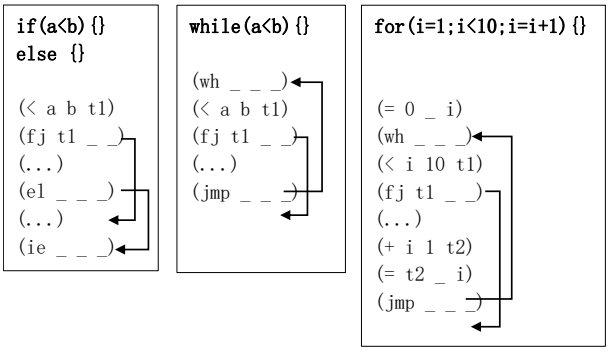
① quat(E_1 ω E_2)= quat(E_1)
quat(E_2)
q:(ω res(E_1) res(E_2) ti)

② quat((E))= quat(E)

③ quat(i)= 空 , res(i)= i

4.3.2 if while for

1、if、while、for 语句四元式结构设计如下图所示：



几点说明：

a) if 和 else 语句中 else{}为可选项

b) for()中第一项(等式)及第三项(等式)为可选项

c) 在代码实现上，其中 for 的小括号()中第三项通过 queue <Quater_Node> temp_quarter 将生成的四元式进行临时存储。直到读取到 for 的右大括号 } 时将 temp_quarter 再全部 push 到 data 的 Quarter 中。

2、for()内第三项四元式提至 for 四元式结构末端的解决方案

引入标记变量 flag_temp_store，为 false 表示生成的四元式可直接存入 data 中，为 true 表示生成的四元式需要临时存储在 queue <Quater_Node> temp_quarter 中。

现单独考虑第三项存在的情况。

1) 在 for 的语法分析中，读完第二个分号 ';' 后，将 flag_temp_store 标记为 true，因此进入到 Equal_Sentence()以及 Math_Expression()中时，生成的四元式会存储在临时数据集 temp_quarter 中。

2) 在回到 for 的语法分析后，将 flag_temp_store 标记回 false。这样第三项的四元式就被单独存储在了 temp_quarter 中。

3) for 语法分析语句的最后一个 '}' 读完后、在填写 Add_Quater("jmp", "", "", "")四元式前，再将 temp_quarter 的结点全部添加到 data 的 Quarter 中。

从而实现了第三项的四元式提至 for 四元式结构末端。

4.3.3 总体简单介绍

1、语法制导技术生成四元式

插入语义动作的文法如下表：

```
grammar->"main" push "{"<Sub_Program>"}" push
Sub_Program->Type_Define
             Var_Define
             Sentence_ Define
Type_Define ->"TYPE" "{" "}"
Var_Define ->"VAR {" Variable_Define "}"
Variable_Define-> {Type    Variable_Table ";"}
Type->"int" | "float"
Variable_Table-> Variable "," Variable_Table | Variable
Sentence_Define -> { Equal_Sentence ";" | If | While | For }
Equal_Sentence-> Variable "=" Math_Expression cclt
Compare_Sentence-> Math_Expression ω3 Math_Expression cclt
If->"if push (" Compare_Sentence") {" Sentence_ Define"}"
    [ "else {" Sentence_Define"}" ]
While->"while push (" Compare_Sentence")
        {" Sentence_ Define"}"
For->"for push ("[Equal_Sentence]";" Compare_Sentence";" [Equal_Sentence]")
    {" Sentence_Define"}"
```

```
Math_Expression:
E -> TE′ ① { i,( }
E′-> ω1 T push1 E′② { ω1} | ε ③ { ),# }
T -> FT′④ { i,( }
T′-> ω2 F push1 T′⑤ { ω2} | ε ⑥ { ω1,),# }
F -> i push2 ⑦ { i } | (E) ⑧ { ( }
```

递归向下生成的文法中：

"str" push：指生成四元式(str,　,　,　)

" cclt" push：指过程

object1 = SEM.top();　Math_Expression();　object2 = SEM.top();

Make_T_Variable(result, 'b');　生成四元式(ch, object1, object2, result)

LL(1)进行的 Math_Expression 中：

i push2：向 SEM 语义栈中压入 i（常量或变量）

ω push1：生成四元式(ω, SEM[top-1], SEM[top], temp)

并将 temp 压回 SEM

当产生式逆序压栈时，动作符号同样逆序压栈

当动作符号位于栈顶时，从 SEM、SEM_Operate 栈取元素执行相应动作

在程序中，’@’代表 ”push1”， ’$’代表 ”push2”

2、全部符号表的填写

1. 在词法分析 NEXTw()中：

   会将声明的变量 name 填入 I 表，并通过全局变量 flag_var 进行区分是否在 VAR{}中，从而判断是否为变量重复定义或变量未声明。

2. 在语法分析的 Variable_Define()中：

   通过变量声明前的关键字填写下一个分号前所有变量的

   type_p；其 cat 都是” v ”；

   add.info[地址]：若 info=-1，表示自定义类型，add.offset 储存长度信息，否则储存函数层数，main 为 0

   add.offset[地址]：若 info!=-1，表示为变量，add.offset 储存该变量区距。

   各类型默认长度：int:4, float:4, bool:1, char:1

3. 在生成四元式时：

   1. Compare_Sentence()中会生成 bool 型临时变量，

      Math_Expression()中会生成 float 型临时变量。

   2. 临时变量的 I 表现暂存在 queue <Identity_Node>

      Identity_Temp 中，完成四元式优化后，再将需要的临时变量压入 I 表中。

3. 临时变量的命名规则为：Temp_Variable + to_string (number_of_t ++)，其中 Temp_Variable 为宏定义的字符串"$temp"， number_of_t 为 int 型全局变量进行记录。

4.3.4 算术表达式详细说明

1、文法及动作

插入动作后的文法：

```
G``(E)：E -> T | E+T{GEQ(+)} | E-T{GEQ(-)}
        T -> F | T*F{GEQ(*)} | T/F{GEQ(/)}
        F -> i{PUSH(i)} | ( E )
```

其中：

PUSH(i)：压栈函数(把当前 i 压入语义栈)，在程序中用向 SYN 压入$表示压入此动作。

GEQ(ω) – 表达式四元式生成，在程序中用向 SYN 压入@表示压入此动作。包括以下三个步骤：

```
① t := NEWT; { 申请临时变量函数；}
② SEND(ω SEM[m-1],SEM[m],t)
③ POP;POP;PUSH(t)
```

2、实际数据结构

1. 所用语法栈为 stack <char> SYN，考虑 LL1(1)文法具体实现时程序的简洁性，加之其 char 数据类型的限制，SYN 栈中压入数据时，Math_Expression_ch()函数实现将 Token（单词）转化为以下九个字符：C、I、+、-、*、/、#、@、$之一。仅使以上九种字符进入 LL(1)分析法。

2. 语义栈为 stack <string> SEM，但 SYN 中动作$所对应原始单词常量或变量名被丢失，因此需要添加变量 string SEM_Object，记录语义栈要压入的终结符的实际 name（或代表数字、地址的 string）。每比对到一个 I 表、C 表中的终结符 temp，就需要操作 SEM_Object = temp 将 temp 保存，以便在 SYN 栈顶为动作时提供运算对象。
3. 同样，SYN 中压入动作$时，所对应的原始操作符被丢失，因此需要添加 stack <string> SEM_Operate，每当向 SYN 中压入@ SYN.push('@')时，便进行 string temp= ch; SEM_Operate.push(temp)操作，将该@所对应的运算符保存在 SEM_ Operate，以便在 SYN 栈顶为动作时提供运算符对象。其长度为 SYN 栈中现有的@个数。

3、函数及具体实现

1. 在程序的具体实现中，算数表达式主控函数为 Grammar 中的 Math_Expression()。top_ch 为 SYN 栈顶元素，ch 为语法分析当前处理单词（经 Math_Expression_ch()转化后），对以下共 5 种不同情况的判别，采取相应的过程：
   i. top_ch == ch == #：该部分 LL(1)文法结束且正确。
   ii. top_ch == ch != #：栈顶和当前匹配，NEXTw()。
   iii. top_ch != ch：Math_Expression_Search(top_ch, ch)进行 LL(1)分析表查询。
   iv. top_ch == '$'：SEM.push(SEM_Object)。
   top_ch == '@'：运算符为 SEM_Operate 栈顶元素，对象 1 为次栈顶元素，对象 2 为栈顶元素，并生成一个 float 型

的临时变量存储本个四元式的 result，Make_T_Variable 功能为：生成最新的一个临时变量并将其压入 Identity_Temp 临时 I 表中，其类型设置为传入的第二个参数，并将传入的 name 赋为该临时变量的 name。

operate= SEM_Operate.top()

object2 = SEM.top(); SEM.pop();

object1 = SEM.top(); SEM.pop();

result; Make_T_Variable(result,'f')；

Add_Quater(operate, object1, object2, result)。

2. 依据 SYN 栈顶单词、当前单词对比，查询到使用语句，并将该语句进行逆序压入 SYN 栈的工作由 Math_Expression_Search(char top_ch, char ch)进行完成。需要注意的是，在逆序压栈的过程中，动作符号也要按照在文法中的位置逆序压栈。
3. Math_Expression_ch()函数：
   a) Token.Type == "CI"、"CF"时，ch = 'C'，SEM_Object = temp。
   b) Token.Type == "CC"时，在此步骤将字符转化为其 ASCII 码数值，从而实现字符型数据参与运算，ch = 'C'，SEM_Object = temp。
   c) Token.Type == "K"且当前单词为"true"或"false"时，在此步骤将 bool 型转化为 0 或 1 值，从而实现布尔型数据参与运算。ch = 'C'，SEM_Object = temp。
   d) Token.Type == "I"时，ch = 'I'，SEM_Object = temp。
   e) 遇到运算符(+ - * /)时 ch 就为相应运算符（char 型）。
   f) 在文法中，算数表达式的 follow 集包括“；、] 、)} 、> 、

< 、! 、& 、|”等 8 个符号，因此判断若为这些字符则将 ch 的值设置为 '#'。

4.3.5 数组元素访问详细说明

1) 文法定义如下：

   Array_Define-> Type “array” Variable

   “[“ Math_Expression ”]”{ “[“ Math_Expression ”]” }*

   单句中，可声明一个多维数组。

2) 数组结构信息在 data 中的 Array 数组表中单独存储，单个结点中有数据：int type_p 类型指针、int total_space 占空间总长度、int dimension 维度、vector <int> dimension_length 各维长度，以实现一个结点存储一个任意维度数组类型信息。

3) 检测到最后一个”] ”后表示定义完成，将变量 I 表中的指针指向新建的 Type 表{ “a” , array_p }。

以上文法的语法分析、语义分析、符号表修改集成在 Variable_Define() 函数中。

1) 为实现数组元素可使用变量、表达式访问（例如 arr[x][y][z]），需在运行时刻计算实际值。复合语句中的数组元素访问及存储时，通过计算，利用语义栈 SEM_array、语法栈 SYN_array，将元素 ar[x][y][z]转化为该元素在活动记录中的虚拟地址。

2) 例：访问数组 int array arr[10][5][2]的元素 ar[x][y][z]，计算法如下：

   for (i = 0, length = arr. total_space; i < arr.dimension; ++i)

   a) 遇到 [：

      length /= arr.dimension_length[i];

SYN_array. push(“+“)，SYN_array. push(length);

b) 遇到 x ：

SEM_array.push(x);

c) 遇到] ：

计算：$t = SEM_array.pop() * SEM_array.pop();

SEM_array.push(t);

d) for 循环结束后，将 SEM_array 栈弹空并累加得$t4，再加上 arr 在 I 表的 offset，即可得到元素 ar[x][y][z]在活动记录的 offset，设为”->t5”以区分其与其他临时变量，并将其类型设置为 p，所占长度为 2 字节。

3) 在目标指令中，以形式 [f0:->t5] 以表示->t5 需要以地址的含义进行访问。

4) 数组的括号[ ]间可以是数字、变量、数学表达式，甚至是含有数组元素的数学表达式。例如：ar[0][x - i][i + ar[0][0][0]]

```
13 ( =   ,1       ,         ,[f0:->t8])
14 ( =   ,0       ,         ,i        )
15 ( wh  ,        ,         ,         )
16 ( <=  ,i       ,10       ,$t9      )
17 ( fj  ,$t9     ,         ,         )
18 ( =   ,10      ,         ,x        )
19 ( *   ,80      ,0        ,$t11     )
20 ( -   ,x       ,i        ,$t12     )
21 ( *   ,8       ,$t12     ,$t13     )
22 ( *   ,80      ,0        ,$t14     )
23 ( *   ,8       ,0        ,$t15     )
24 ( *   ,4       ,0        ,$t16     )
25 ( +   ,$t15    ,$t16     ,$t17     )
26 ( +   ,$t14    ,$t17     ,$t18     )
27 ( +   ,18      ,$t18     ,->t19    )
28 ( +   ,i       ,[f0:->t19],$t20    )
29 ( *   ,4       ,$t20     ,$t21     )
30 ( +   ,$t13    ,$t21     ,$t22     )
31 ( +   ,$t11    ,$t22     ,$t23     )
32 ( +   ,18      ,$t23     ,->t24    )
33 ( *   ,0.314000,i        ,$t25     )
34 ( =   ,$t25    ,         ,[f0:->t24])
```

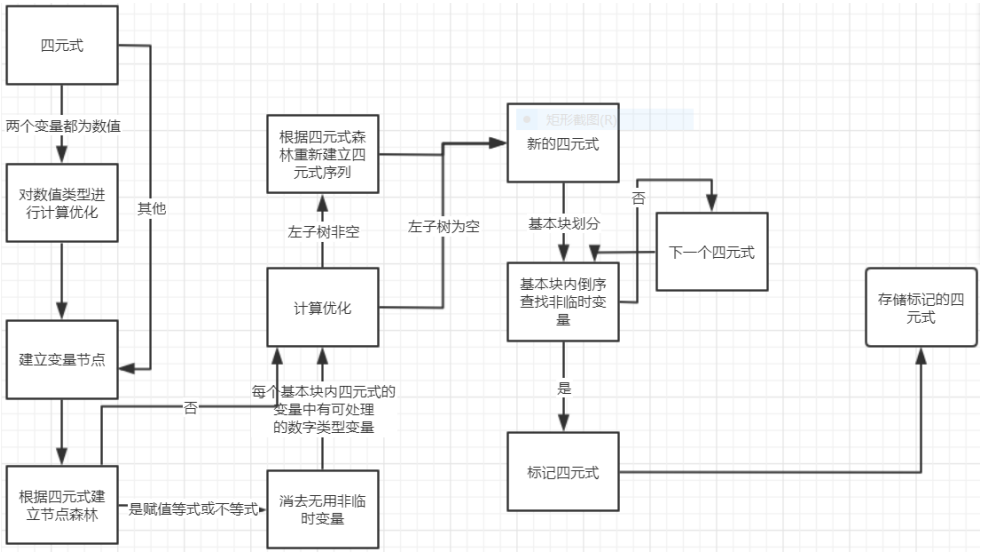
```
ar[0][0][0] = 1;
for (i = 0; i <= 10; i = i + 2)
{
    x = 10;
    ar[0][x - i][ i + ar[0][0][0] ] = 3.14e-1 * i;
    if (i >= 2 && i != 4)
    {
        while (ch <= 'Z' || bol == true)
        {
            ch = ch + bol;
            bol = false;
        }
        ch = 'A';
    }
    y = 1 + 2 + 3 + 4 + 5 - 0.1e+2 + 0.2e1 * (ar[
}
}
```

5 编译器后端设计

5.1 优化

5.1.1 功能及流程图

仿照所学 DAG 优化形式，可以实现基本块划分，常值表达式节省，公共子表达式节省，删除无用赋值，循环优化。流程图如下图所示。



5.1.2 数据结构设计

1、DAG 图所用结构体如下：

```
typedef struct DAG_Node
{
string id;                    //结点的值(一般是基础变量）或者是运算符
int left = -1;                //左子树位置
int right = -1;               //右子树位置
vector<string>var;            //存放的值
}DAG_Node;
```

2、类Optimize成员如下：

```
public:
    Optimize(Data& data);                          //构造函数
    void optimize();                               //四元式优化进程接口,DAG优化四元式
    int offset;                                    //填符号表，当前可用区距
private:
    int num = 0;
    vector <Quater_Node> new_Quater;               //存储优化后的四元式
    Quater_Node new_Q;                             //临时存储四元式
    int cnt = 0;                                   //cnt  为当前结点的个数
    vector <Quater_Node> ans;                      //存储最后输出的优化后的四元式
    Quater_Node temporary_ans;                     //临时存储优化四元式
    bool* flas = NULL;                             //标记可以输出的四元式
    vector<DAG_Node> DAG;                          //节点
    DAG_Node dag;                                  //临时节点
    vector <Identity_Node> new_Identity;           //临时使用的非临时标识符表：I
    Identity_Node linshi;                          //临时非临时标识符点
    Identity_Node temporary_Identity_Temp;         //临时使用的临时标识符点
    Data& data;                                    //引用操作data
    bool find_Identity(string c);
    bool find_Identity_1(string C);                //查找标识符表
    int find_Related_word(string c);               //查找关联词
    void find_Identity_Temp(string c);             //查找非临时变量
    bool find_var(int i, string c);                //遍历查找变量
```

```
int add_node(string c);                          //  建立结点
void add_operator(string c, string op, int l, int r);//添加表达式
void dfs(int i);                                 //标记表达式结点
void replace_DAG(int cnt,int temp1,int i);       //替换多余变量
void cout_DAG(int cnt);                          //计算再优化数字型四元式
string ok(int i);                //查询结点的值，合并的部分的变量的节点可以删去
bool number(string a);                           //判断值是否是数字;
string operatornum(string a, double b, double c);
inline double str_change_num(string a);          //string型转换成double型
```

5.1.3 函数及具体实现

（1）消去一些多余的临时变量等

```
void Optimize::replace_DAG(int cnt, int temp1, int i)
{
if (DAG[cnt - 1].id == "=" && data.Quater[i].object2 == "")//删掉赋值等式里不
需要的变量链接，如一些非临时变量
{
if (DAG[temp1].id == "=")
        {
            DAG[cnt - 1].left = DAG[temp1].left;
            DAG[cnt - 1].right = DAG[temp1].right;
        }
else if (DAG[temp1].id == "+" || DAG[temp1].id == "-" || DAG[temp1].id == "*" ||
DAG[i].id == "/")
        {
            DAG[cnt - 1].id = DAG[temp1].id;
            DAG[cnt - 1].left = DAG[temp1].left;
            DAG[cnt - 1].right = DAG[temp1].right;
        }
    }
}
```

(2)按照基本块对建立好的四元式森林里的 int 类型的变量进行计算优化处理

```
void Optimize::cout_DAG(int cnt)
{
    int nn = 0;
    for (int i = 0; i < cnt - 1; i++)
    {
        nn = i;
```

```
        int l = -2;
        int r = -2;
        if (DAG[nn].id == "=" && DAG[DAG[nn].left].id == "=")
        {
            DAG[nn].left = DAG[DAG[nn].left].left;
        }
        if (DAG[i].id == "+" || DAG[i].id == "-" || DAG[i].id == "*" || DAG[i].id
== "/")//判断是不是运算的四元式
        {
            l = DAG[i].left;
            r = DAG[i].right;
        }
        else continue;
        if (number(DAG[l].id) && number(DAG[r].id))//两个 object 是数
        {
            double a1 = str_change_num(DAG[l].id);
            double a2 = str_change_num(DAG[r].id);
            string object1 = operatornum(DAG[i].id, a1, a2);
            string object2 = "";
            string operation = "=";
            int ll = add_node(object1);//左子树
            int rr = add_node(object2);//右子树
            DAG[i].id = operation;
            DAG[i].left = ll;
            DAG[i].right = rr;

        }
        else if (number(DAG[l].id) && !number(DAG[r].id))//object1 是数
        {
            if (DAG[r].id == "=")
            {
                r = DAG[r].left;
            }
            else continue;
            if (number(DAG[r].id))
            {
                double a1 = str_change_num(DAG[l].id);
                double a2 = str_change_num(DAG[r].id);
                string object1 = operatornum(DAG[i].id, a1, a2);
                string object2 = "";
                string operation = "=";
                int ll = add_node(object1);//左子树
                int rr = add_node(object2);//右子树
                DAG[i].id = operation;
                DAG[i].left = ll;
                DAG[i].right = rr;
            }
```

```
        else continue;
    }
    else if (!number(DAG[l].id) && number(DAG[r].id))//object2 是数
    {
        if (DAG[l].id == "=")
        {
            l = DAG[l].left;
        }
        else continue;
        if (number(DAG[l].id))
        {
            double a1 = str_change_num(DAG[l].id);
            double a2 = str_change_num(DAG[r].id);
            string object1 = operatornum(DAG[i].id, a1, a2);
            string object2 = "";
            string operation = "=";
            int ll = add_node(object1);//左子树
            int rr = add_node(object2);//右子树
            DAG[i].id = operation;
            DAG[i].left = ll;
            DAG[i].right = rr;
        }
        else continue;
    }
    else//都不是数
    {
        if (DAG[l].id == "=" && DAG[r].id == "=")
        {
            l = DAG[l].left;
            r = DAG[r].left;
        }
        else continue;
        if (number(DAG[l].id) && number(DAG[r].id))
        {
            double a1 = str_change_num(DAG[l].id);
            double a2 = str_change_num(DAG[r].id);
            string object1 = operatornum(DAG[i].id, a1, a2);
            string object2 = "";
            string operation = "=";
            int ll = add_node(object1);//左子树
            int rr = add_node(object2);//右子树
            DAG[i].id = operation;
            DAG[i].left = ll;
            DAG[i].right = rr;
        }
        else continue;
    }
```

```
        }
    }
```

(3)判断字符串是否是临时变量

```
void Optimize::find_Identity_Temp(string C)
{
    int t = 0;
    int length = data.Identity_Temp.size();
    for (int j = 0; j < length; j++)
    {
        if (C == data.Identity_Temp[j].name)
        {
            temporary_Identity_Temp.active = data.Identity_Temp[j].active;
            temporary_Identity_Temp.add = data.Identity_Temp[j].add;
            temporary_Identity_Temp.cat = data.Identity_Temp[j].cat;
            temporary_Identity_Temp.name = data.Identity_Temp[j].name;
            temporary_Identity_Temp.type_p = data.Identity_Temp[j].type_p;
            temporary_Identity_Temp.add.info = 0;
temporary_Identity_Temp.add.offset = offset;
            //修改当前 offset 记录变量
            offset += data.Find_Size(temporary_Identity_Temp.type_p);
            data.Identity.push_back(temporary_Identity_Temp);
            swap(data.Identity_Temp[j], data.Identity_Temp[length - 1]);//删除
已经用过的标识符
            data.Identity_Temp.pop_back();
            break;
        }
        else t = t + 1;
    }
}
```

(4)判断字符串是否是非临时变量，是的话删除已将找到的非临时变量

```
bool Optimize::find_Identity(string C)
{
    if (C[0] == '['|| C[0] == '-')return true;
    int t = 0;
    int length = new_Identity.size();
    for (int j = 0; j < length; j++)
    {
        if (C == new_Identity[j].name)
        {
            swap(new_Identity[j], new_Identity[length - 1]);//删除已经用过的
标识符
            new_Identity.pop_back();
            return true;
        }
        else t = t + 1;
```

```
        }
        if (t == length)return false;
        return false;
}
int Optimize::find_Related_word(string c)
{
        for (int j = 0; j < 8; j++)
                if (c == Related_word[j])return 1;
        return 0;
}
bool Optimize::find_var(int i, string c)
{
        // 遍历一个结点的所有值
        int len = DAG[i].var.size();
        for (int k = 0; k < len; k++)
                if (DAG[i].var[k] == c)
                        return true;
        return false;
}
```

(5)根据变量建立叶子节点

```
int Optimize::add_node(string c)
{
        // 遍历当前已经建立的结点
        for (int j = cnt - 1; j >= 0; j--)
                // 如果该结点建立过，或者和其他结点运算内容一样。合并这些变
量
                if (DAG[j].id == c || find_var(j, c)) return j;
        // 需要新建立结点
        dag.id = c;
        DAG.push_back(dag);
        dag.id = "";
        dag.left = -1;
        dag.right = -1;
        dag.var.clear();
        //if(c=="")return -1;
        return cnt++;
}
```

(6)根据四元式建立变量之间的相互关系，构成森林

```
void Optimize::add_operator(string c, string op, int l, int r)
{
        string temp;
        for (int j = cnt - 1; j >= 0; j--)
        {
                //如果该表达式已经存在，将该表达式左边的值放到结点的 var 中即
可。
```

```
            if (op == DAG[j].id && DAG[j].left == l && DAG[j].right == r)
            {
                num = j;
                DAG[j].var.push_back(c);
                if (find_Identity(c) && !find_Identity(DAG[j].var[0]))//c 是非临时
变量，把它放在 var[0]处
                {
                    int length = DAG[j].var.size();
                    temp = DAG[j].var[0];
                    DAG[j].var[0] = c;
                    DAG[j].var[length - 1] = temp;
                }
                return;
            }
        }
        // 如果没有，则需要把整个表达式都加进去
        // id 是运算符符，左右结点为变量，结果存到父节点的 var 中
        dag.id = op;
        dag.left = l;
        dag.right = r;
        dag.var.push_back(c);
        DAG.push_back(dag);
        dag.id = "";
        dag.left = -1;
        dag.right = -1;
        dag.var.clear();
        cnt++;
        num = cnt - 1;
    }
```

(7)判断四元式是否需要标记保存

```
    void Optimize::dfs(int i)
    {
        if (DAG[i].left != -1)
        {
            flas[i] = 1;
            dfs(DAG[i].left);//左右子树的值的表达式需不需要保留
            dfs(DAG[i].right);
        }
        else flas[i] = 0;
    }
```

(8)返回可替换的变量

```
    string Optimize::ok(int i)
    {
        int len = DAG[i].var.size();
        return DAG[i].var[0];
```

```
}
```

(9)进行 int 型变量的运算

```
string Optimize::operatornum(string a, double b, double c)
{
    double x = 0.0;
    stringstream ss;          //stringstream 需要 sstream 头文件
    string str;
    if (a == "/" && b != 0.0)x = b / c;
    else if (a == "/" && b == 0.0)x = 0.0;
    else if (a == "*")x = b * c;
    else if (a == "-")x = b - c;
    else if (a == "+")x = b + c;
    ss << x;
    ss >> str;
    return str;
}
```

(10)将字符串转换为数值

```
inline double Optimize::str_change_num(string a)
{
    double b = atof(a.c_str());
    return b;
}
```

(11)优化主要流程：

```
void Optimize::optimize()
{
    cnt = 0;
//输入四元式数量以及四元式内容
    int size = data.Quater.size();
    for (int i = 0; i < size; i++)
    {
        //判断是否两个都是数字，是的话需要计算存储
        if (number(data.Quater[i].object1) && number(data.Quater[i].object2))
        {
            double a1 = str_change_num(data.Quater[i].object1);
            double a2 = str_change_num(data.Quater[i].object2);
            data.Quater[i].object1 = operatornum(data.Quater[i].operation, a1, a2);
            data.Quater[i].object2 = "";
            data.Quater[i].operation = "=";
        }
        //  使用 add_node 建立结点
        int l = add_node(data.Quater[i].object1);//左子树
        int r = add_node(data.Quater[i].object2);//右子树
        string result = data.Quater[i].result;
        //  将表达式构成树
```

```
            add_operator(data.Quater[i].result, data.Quater[i].operation, l, r);
            int temp1;
            temp1 = DAG[cnt - 1].left;
            replace_DAG(cnt, temp1, i);//替换多余变量，大多是临时变量
        }
        cout_DAG(cnt);//计算再优化
        flas = new bool[cnt];
        //优化四元式并存储
        for (int i = 0; i < cnt; i++)
        {
            // 有左子树的节点就是一个表达式
            if (DAG[i].left != -1) {
                //得到存储的替代值
                temporary_ans.result = ok(i);
                // 左右子树节点
                DAG_Node ll = DAG[DAG[i].left];
                DAG_Node rr = DAG[DAG[i].right];
                // 如果这个结点的左子树不是空的，值就是他的 var
                // 如果是空的，那就是左子树本身的值
                temporary_ans.object1 = ll.left != -1 ? ok(DAG[i].left) : ll.id;
                temporary_ans.operation = DAG[i].id;
                temporary_ans.object2 = rr.left != -1 ? ok(DAG[i].right) : rr.id;
                ans.push_back(temporary_ans);
                //ans[i][4] = "\0";
            }
            else //没有左子树为空
            {
                temporary_ans.result = "";
                temporary_ans.object1 = "";
                temporary_ans.operation = "";
                temporary_ans.object2 = "";
                ans.push_back(temporary_ans);
            }
        }
        for (int i = 0; i < cnt; i++)//每个基本块内标记应该保留的四元式
        {
            int first = i;
            for (int h = first + 1; h < cnt; h++)
            {
            if (find_Related_word(ans[h].operation))//遇到基本块的结束点
            {
                int last = h; new_Identity = data.Identity;
                for (int m = last; m >= first; m--)//基本块内逆序查找，防止多次定
义

if(find_Identity(ans[m].result)||find_Related_word(ans[m].operation))
                        dfs(m);
```

```
                        break;
                    }
                }
            }
            for (int i = 0; i < cnt; i++)
            {
                if (flas[i] == 1)
                {
                    new_Q.operation = ans[i].operation;
                    new_Q.object1 = ans[i].object1;
                    new_Q.object2 = ans[i].object2;
                    new_Q.result = ans[i].result;
                    find_Identity_Temp(new_Q.result);
                    new_Quater.push_back(new_Q);
                }
            }
            data.Quater.swap(new_Quater);
}
```

5.2 活跃信息填写

5.2.1 输入、输出及数据结构基础

1. 在 I 表增加一项：bool active，为 true 表示活跃，false 表不活跃。
2. 四元式的活跃信息表单独使用 stack <Active_Node> Active 储存，一个 Active_Node 结点储存一条四元式后三项（object1，object2，result）的活跃信息。
3. 逆序读取四元式表，依次生成各条四元式的活跃信息压入 Active 栈中，完成后（一次全部读取）栈长度与四元式个数相同。

5.2.2 功能函数

1. void Make_Active()：活跃信息表填写过程主控函数。通过对其他 2 个函数的调用实现判断、活跃信息查询与修改、活跃信息表生成功能。
2. void Reset_Identity_Active()：初始化 I 表中各变量的活跃信息。非临时变量置为 true，临时变量置为 false（临时变量以$开头,非临时变量以字母开头,地址临时变量的形式为 “->tn”，同样作为非临时变量处理）。
3. bool Search_Change(string name, bool to)：在 I 表查询到 name 所对应的变量，返回其当前活跃信息，并将其活跃信息更改为传入的 to。

5.2.3 实现步骤

1. Make_Active()每遇到一条跳转四元式（第四项为空），表示一个基本块的开始（或结束），就调用 Reset_Identity_Active()将 I 表中的 active 记录值初始化。
2. Make_Active()每遇到一条操作四元式，其活跃信息表结点 temp 生成方式如下：
   a) 判断其 object 和 result 的三项是否为变量
   b) 若变量为 object 位置，则先将 I 表中它当前活跃状态填入 temp，再将 I 表它的活跃状态置为 true；
   c) 若变量为 result 位置，则先将 I 表中它当前活跃状态填入 temp，再将 I 表它的活跃状态置为 false。

下左图四元式对应的活跃信息表填写截图如下中图。右侧为测试代码。

```
==============四元式==============
( mian,       ,       ,       )
( wh  ,       ,       ,       )
( <   ,i      ,10     ,$t1    )
( fj  ,$t1    ,       ,       )
( +   ,1      ,1      ,$t2    )
( =   ,$t2    ,       ,i      )
( +   ,i      ,i      ,$t3    )
( =   ,$t3    ,       ,x      )
( =   ,5      ,       ,x      )
( +   ,i      ,x      ,$t4    )
( *   ,i      ,$t4    ,$t5    )
( +   ,1      ,$t5    ,$t6    )
( =   ,$t6    ,       ,y      )
( jmp ,       ,       ,       )
( end ,       ,       ,       )
```

```
( 0 , 0 , 0)
( 0 , 0 , 0)
( 0 , 0 , 1)
( 0 , 0 , 1)
( 1 , 0 , 1)
( 1 , 1 , 1)
( 0 , 0 , 1)
( 0 , 0 , 0)
( 1 , 1 , 1)
( 0 , 0 , 1)
( 0 , 0 , 1)
( 0 , 0 , 0)
( 1 , 0 , 1)
( 0 , 0 , 0)
( 0 , 0 , 0)
```

```
main
{
  TYPE{}
  VAR{
  int i, x, y,a;
  }
  while(i<10){
 i=1+1;
x=i+i;
x=5;
y=1+i*(i+x);}
}
```

5.3 目标代码生成

5.3.1 输入、输出及数据结构基础

1) 程序中使用虚拟指令，模拟单寄存器下的目标指令。
2) 使用 RDL（寄存器描述表，结构体为{变量 name，变量 active}，可见 3.4 数据结构设计下的寄存器描述表节点结构图）记录当前寄存器 R 中存储的信息所代表的变量。
3) 使用 stack <int> SEM 语义栈，临时记录待返填的目标地址。
4) 生成后的所有目标指令存储在 data 的 Order 中。

5.3.2 功能函数

1. void order()：目标代码生成主控函数。函数先调用 Make_Active() 生成所有四元式的活跃信息表；再从头到尾遍历四元式，依据四元式的不同（共可分为 8 类），进行相应操作，进行目标指令生成。
2. int Add_Order(string operation, string rgst, string object)：按照传入参数向 data 的 Order 添加新的命令这一步骤的封装后的附属函数。
   a) 根据传入的 operation 在目标指令集合 map <string, string> order_set 中查找到相应的目标指令操作即为目标指令中的 temp_order.operation。
   b) 目标指令的寄存器一项 temp_order.rgst 即等于传入的 rgst。
   c) 目标指令中的操作内存（虚拟）地址一项 temp_order.address：
      i. 若传入的 object 为常数、指针、地址、空等四种情况时，则 temp_order.address=object。
      ii. 否则 object 为变量名，在 I 表中查找到该变量并依据活动记录及其 offset 构造（虚拟）地址一项。

5.3.3 实现方法

四元式可分为以下几类来处理：

1. (fj B _ _)：
   a) 若 RDL 为空，则 Add_Order(LD R,B ; FJ R,_); PUSH(指令位置)；
   b) 若 RDL==B，若 B(y)则 Add_Order(ST R,B ; FJ R,_); 否则 Add_Order(FJ R,_);再 PUSH(指令位置)；RDL 置为空。
   c) 若 RDL==D（D!=B）若 D(y) 则 Add_Order(ST R,D ; LD R,B ; FJ R,_); 否则 Add_Order(LD R,B ; FJ R,_); 再 PUSH(指令位置)；RDL 置为空。

1、 (el _ _ _)：向后跳转，其下一句为上一个跳转的返填位置

```
if (RDL.name != "" && RDL.active == true)
    Add_Order("ST", "R", RDL.name);
//先返填上一条的转向地址
int temp = SEM.top(); SEM.pop();
data.Order[temp].address = "[ord " +
to_string(data.Order.size() + 1) + "]";
//再生成本句的转向指令
SEM.push(Add_Order("JMP", "", ""));
```

2、 (ie _ _ _)：跳转到，是上一个跳转的返填位置

```
if (RDL.name != "" && RDL.active == true)
    Add_Order("ST", "R", RDL.name);
int temp = SEM.top(); SEM.pop();
data.Order[temp].address = "[ord " +
to_string(data.Order.size()) + "]";
```

3. (wh _ _ _)：跳转到，是下一个跳转的返填位置。可视为下一个基

本块的开始。

```
if (RDL.name != "" && RDL.active == true)
    Add_Order("ST", "R", RDL.name);
RDL.name = "";
SEM.push(data.Order.size());
```

4. (jmp _ _ _)：向前跳转，其下一句为上一个跳转的返填位置

```
if (RDL.name != "" && RDL.active == true)
      Add_Order("ST", "R", RDL.name);
RDL.name = "";
int temp = SEM.top(); SEM.pop();
data.Order[temp].address = "[ord " +
to_string(data.Order.size() + 1) + "]";
temp = SEM.top(); SEM.pop();
Add_Order("JMP", "","[ord " + to_string(temp) + "]");
```

5. (start f_name _ _)：一个函数开头语句，需要定义开辟 f_name 函数所需空间的虚拟指令，该函数所需空间存储在 I 表中其结点的 add.offset 中。

   Add_Order(i->operation, i->object1, "[New "+to_string(data.Identity[0].add.offset) + "]");

6. (end f_name _ _)：一个函数结尾语句。需定义释放 f_name 函数空间的虚拟指令。

   Add_Order(i->operation, i->object1, "[Release]");

7. (= B _ A)：

   a) 若 RDL 为空，则 CODE(LD R,B)

   b) 若 RDL==B，若 B(y) 则 Add_Order(ST R,B);

   c) 若 RDL==D（D!=B）若 D(y) 则 Add_Order(ST R,D ; LD R,B); 否则 Add_Order(LD R,B);

d) 再将 RDL 改为（A，A 的活跃信息）

8. 其他操作符代码实现如下：

```
//若 RDL 为空
if (RDL.name == "")
{
    Add_Order("LD", "R", i->object1);
    Add_Order(i->operation, "R", i->object2);
}
//若 RDL==object1
else if (RDL.name == i->object1)
{
    if(temp_active.object1 == true)
        Add_Order("ST", "R", i->object1);
    Add_Order(i->operation, "R", i->object2);
}
//若 RDL==object2
else if (RDL.name == i->object2)
{
//有可交换性的运算
if (i->operation == "==" || i->operation == "!=" || i->operation == "+" ||
i->operation == "-" || i->operation == "*"
|| i->operation == "&&" || i->operation == "||")
{
        if (temp_active.object2 == true)
Add_Order("ST", "R", i->object2);
        Add_Order(i->operation, "R", i->object1);
    }
}
//其他情况
else
{
    if (RDL.name != "" && RDL.active == true)
        Add_Order("ST", "R", RDL.name);
    Add_Order("LD", "R", i->object1);
    Add_Order(i->operation, "R", i->object2);
}
```

5.3.4 运行测试

下左图四元式对应的目标指令生成截图如下中图。右侧为测试代码。

```
==============四元式==============
( mian,       ,       ,       )
( wh  ,       ,       ,       )
( <   ,i      ,10     ,$t1    )
( fj  ,$t1    ,       ,       )
( >   ,i      ,5      ,$t2    )
( fj  ,$t2    ,       ,       )
( el  ,       ,       ,       )
( *   ,i      ,i      ,$t3    )
( =   ,$t3    ,       ,x      )
( =   ,0      ,       ,i      )
( wh  ,       ,       ,       )
( <   ,i      ,x      ,$t4    )
( fj  ,$t4    ,       ,       )
( +   ,i      ,x      ,$t6    )
( *   ,i      ,$t6    ,$t7    )
( +   ,1      ,$t7    ,$t8    )
( =   ,$t8    ,       ,y      )
( +   ,i      ,1      ,$t5    )
( =   ,$t5    ,       ,i      )
( jmp ,       ,       ,       )
( ie  ,       ,       ,       )
( jmp ,       ,       ,       )
( end ,       ,       ,       )
```

```
=========目标指令=====
0   (mian,    ,    )
1   (LD  , R  ,i   )
2   (<   , R  ,10  )
3   (FJ  , R  ,26  )
4   (LD  , R  ,i   )
5   (>   , R  ,5   )
6   (FJ  , R  ,8   )
7   (JMP ,    ,25  )
8   (LD  , R  ,i   )
9   (*   , R  ,i   )
10  (ST  , R  ,x   )
11  (LD  , R  ,0   )
12  (ST  , R  ,i   )
13  (LD  , R  ,i   )
14  (<   , R  ,x   )
15  (FJ  , R  ,25  )
16  (LD  , R  ,i   )
17  (+   , R  ,x   )
18  (*   , R  ,i   )
19  (+   , R  ,1   )
20  (ST  , R  ,y   )
21  (LD  , R  ,i   )
22  (+   , R  ,1   )
23  (ST  , R  ,i   )
24  (JMP ,    ,13  )
25  (JMP ,    ,1   )
26  (end ,    ,    )
```

```
main
{
  TYPE{}
  VAR{
   int i, x, y,a;
  }
  while(i<10){
  if(i>5){}
  else {
  x=i*i;
  for(i=0;i<x;i=i+1)
  { y=1+i*(i+x);}
  }}
}
```

6 实验结果

6.1 菜单

1、主菜单如下图 6.1-1 所示：

1. 选择 1，则展示未进行四元式优化的数据；
2. 选择 2，则展示进行了四元式优化后的数据；
3. 选择 3，则重新从 Data.txt 文件读取测试代码，重新进行整个编译过程；
4. 选择 0，则退出程序。

使用以下处理支持错误输入纠正：

1. 用 char 类型存储选择
2. 进行是否在选择区间内的判断，不在则引导用户重新输入选择
3. 每次读取后 cin.ignore(1024, '\n')清空缓存区，保障下次读取的为用户输入内容

图 6.1-1 主菜单

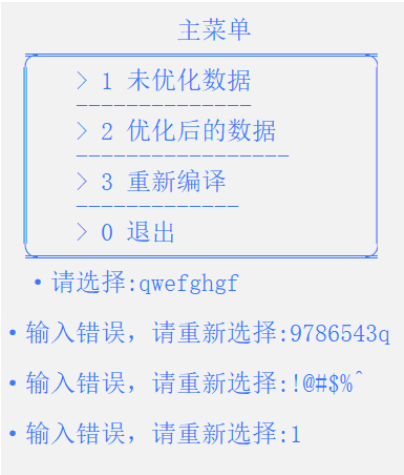
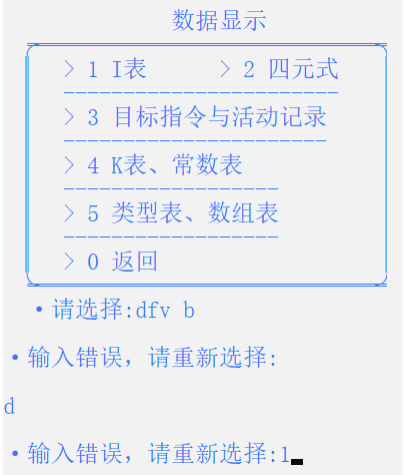


图 6.1-2 数据显示菜单



2、在主菜单选择 1、2 后，程序已不同数据集进入数据显示菜单，如上图 6.1-2 所示。不同输入分别显示 I 表、四元式序列、目标指令与活动记录、K 表与常数表、类型表与数组表等信息。

6.2 编译器前端及符号表数据

1、语法分析及词法分析。若语法正确，则显示图 6.2-1 所示界面后进入主菜单。

图 6.2-1 语法分析正确显示界面

```
            ·编译阶段·
main
{
    var
    {
        int i, x, y;
        char ch;
        bool bol;
        float array ar[10(整数)][5(整数)][2(整数)];
    }
    ch= 'A(字符); ch= 'A(字符) + 2(整数) ;
    bol= true;
    "This is a test(字符串);
    ar[0(整数)][0(整数)][0(整数)]= 1(整数);
    for (i= 0(整数); i <= 10(整数); i= i + 2(整数)){
        x= 10(整数);
        ar[0(整数)][x - i][ i + ar[0(整数)][0(整数)][0(整数)] ]= 0.314(浮点数) * i;
        if (i >= 2(整数) && i != 4(整数)){
            while (ch <= 'Z(字符) || bol == true){
                ch= ch + bol;
                bol= false;
            }
            ch(while结束)= 'A(字符);
        }
         y(if结束)= y + x + 4(整数) + 5(整数) - 10(浮点数) + 2(浮点数);
    }
    x(for结束)= 1(整数) + 2(整数);  y= 1(整数) + 2(整数);
}
            > 语法分析正确 <
```

2、语法分析进程出现错误时，进行图 6.2-2 界面显示，并会在错误处进行“未声明变量使用”、“变量重复声明”、“表达式返回值被忽略”等错误类型显示。

该情况下每键入任意键又重新从 Data.txt 文件读取输入进行新的编译，若新编译时 Data.txt 文件输入代码语法正确，则正常进入主界面。

3、如所需的四个文件出现任意缺失，则显示图 6.2-3 所示错误提示界面。

图 6.2-2 语法分析错误显示界面

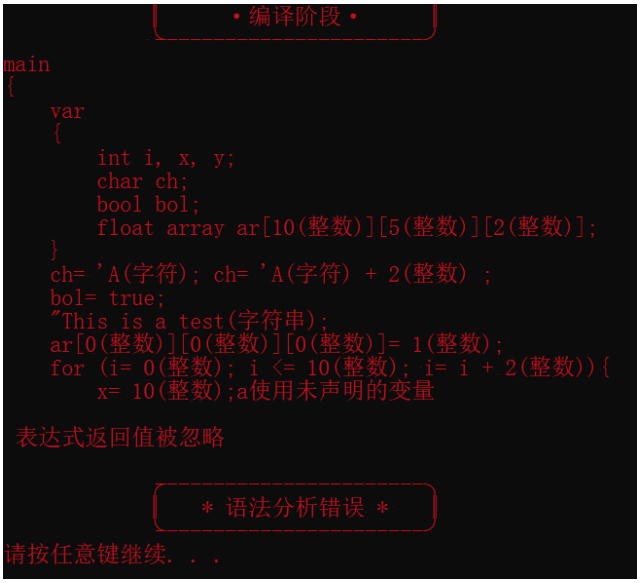
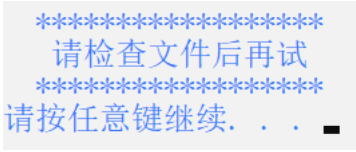


图 6.2-3 文件缺失显示界面



4、关键字 K 表、界符 O、S 表、常量 C 表输出界面如下图 6.2-4 所示。类型 Type 表、数组结构信息 Array 表输出界面如下图 6.2-5 所示。

图 6.2-4 关键字、界符、常量表显示

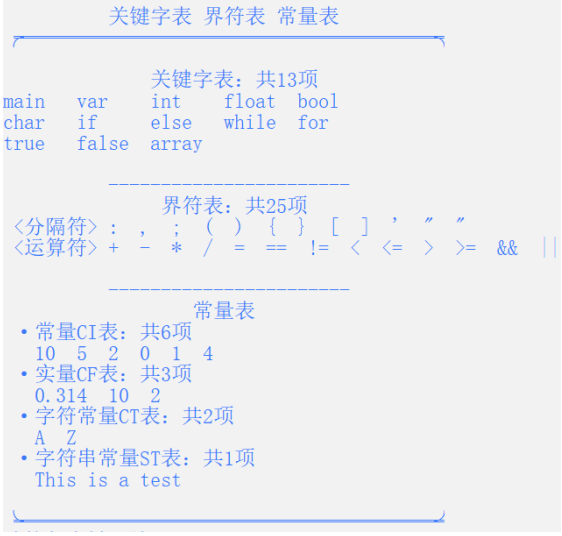
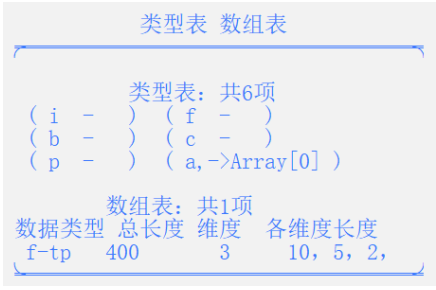


图 6.2-5 类型、数组表输出



5、未优化四元式情况下的 I 表展示如下图 6.2-6 所示，未优化四元式情况下的四元式展示如下图 6.2-7 所示（未优化时该测试数据共 61 条）。

图 6.2-6 未优化四元式时的 I 表

I表

name	type_p	cat	add
main		f	0，543
i	i-tp	v	0，4
x	i-tp	v	0，8
y	i-tp	v	0，12
ch	c-tp	v	0，16
bol	b-tp	v	0，17
ar	a-tp	v	0，18
$t1	f-tp	v	0，418
$t2	f-tp	v	0，422
$t3	f-tp	v	0，426
$t4	f-tp	v	0，430
$t5	f-tp	v	0，434
$t6	f-tp	v	0，438
->t7	p-tp	p	0，442
$t8	b-tp	v	0，444
$t9	f-tp	v	0，445
$t10	f-tp	v	0，449
$t11	f-tp	v	0，453
$t12	f-tp	v	0，457
$t13	f-tp	v	0，461
$t14	f-tp	v	0，465
$t15	f-tp	v	0，469
$t16	f-tp	v	0，473
$t17	f-tp	v	0，477
->t18	p-tp	p	0，481
$t19	f-tp	v	0，483
$t20	f-tp	v	0，487
$t21	f-tp	v	0，491
$t22	f-tp	v	0，495
->t23	p-tp	p	0，499
$t24	f-tp	v	0，501
$t25	b-tp	v	0，505
$t26	b-tp	v	0，506
$t27	b-tp	v	0，507
$t28	b-tp	v	0，508
$t29	b-tp	v	0，509
$t30	b-tp	v	0，510
$t31	f-tp	v	0，511
$t32	f-tp	v	0，515
$t33	f-tp	v	0，519
$t34	f-tp	v	0，523
$t35	f-tp	v	0，527
$t36	f-tp	v	0，531
$t37	f-tp	v	0，535
$t38	f-tp	v	0，539

图 6.2-7 未优化四元式时的四元式

四元式

0	(start	,main	,	,)
1	(=	,65	,	,ch)
2	(+	,65	,2	,$t1)
3	(=	,$t1	,	,ch)
4	(=	,1	,	,bol)
5	(*	,40	,0	,$t2)
6	(*	,8	,0	,$t3)
7	(*	,4	,0	,$t4)
8	(+	,$t3	,$t4	,$t5)
9	(+	,$t2	,$t5	,$t6)
10	(+	,18	,$t6	,->t7)
11	(=	,1	,	,[f0:->t7])
12	(=	,0	,	,i)
13	(wh	,	,	,)
14	(<=	,i	,10	,$t8)
15	(fj	,$t8	,	,)
16	(=	,10	,	,x)
17	(*	,40	,0	,$t10)
18	(-	,x	,i	,$t11)
19	(*	,8	,$t11	,$t12)
20	(*	,40	,0	,$t13)
21	(*	,8	,0	,$t14)
22	(*	,4	,0	,$t15)
23	(+	,$t14	,$t15	,$t16)
24	(+	,$t13	,$t16	,$t17)
25	(+	,18	,$t17	,->t18)
26	(+	,i	,[f0:->t18]	,$t19)
27	(*	,4	,$t19	,$t20)
28	(+	,$t12	,$t20	,$t21)
29	(+	,$t10	,$t21	,$t22)
30	(+	,18	,$t22	,->t23)
31	(*	,0.314000	,i	,$t24)
32	(=	,$t24	,	,[f0:->t23])
33	(>=	,i	,2	,$t25)
34	(!=	,i	,4	,$t26)
35	(&&	,$t25	,$t26	,$t27)
36	(fj	,$t27	,	,)
37	(wh	,	,	,)
38	(<=	,ch	,90	,$t28)
39	(==	,bol	,1	,$t29)
40	(||	,$t28	,$t29	,$t30)
41	(fj	,$t30	,	,)
42	(+	,ch	,bol	,$t31)
43	(=	,$t31	,	,ch)
44	(=	,0	,	,bol)
45	(jmp	,	,	,)
46	(=	,65	,	,ch)
47	(ie	,	,	,)
48	(+	,y	,x	,$t32)
49	(+	,$t32	,4	,$t33)
50	(+	,$t33	,5	,$t34)
51	(-	,$t34	,10.000000	,$t35)
52	(+	,$t35	,2.000000	,$t36)
53	(=	,$t36	,	,y)
54	(+	,i	,2	,$t9)
55	(=	,$t9	,	,i)
56	(jmp	,	,	,)
57	(+	,1	,2	,$t37)
58	(=	,$t37	,	,x)
59	(+	,1	,2	,$t38)
60	(=	,$t38	,	,y)
61	(end	,main	,	,)

6.3 编译器后端

1、优化后的 I 表展示如下图 6.3-1 所示，优化后的四元式展示如下图 6.3-2 所示。

图 6.3-1 I 表-优化后

I表

name	type_p	cat	add
main		f	0 ,475
i	i-tp	v	0 ,4
x	i-tp	v	0 ,8
y	i-tp	v	0 ,12
ch	c-tp	v	0 ,16
bol	b-tp	v	0 ,17
ar	a-tp	v	0 ,18
$t2	f-tp	v	0 ,418
->t7	p-tp	p	0 ,422
$t8	b-tp	v	0 ,424
$t10	f-tp	v	0 ,425
$t12	f-tp	v	0 ,429
->t18	p-tp	p	0 ,433
$t19	f-tp	v	0 ,435
$t20	f-tp	v	0 ,439
$t21	f-tp	v	0 ,443
$t22	f-tp	v	0 ,447
->t23	p-tp	p	0 ,451
$t25	b-tp	v	0 ,453
$t26	b-tp	v	0 ,454
$t27	b-tp	v	0 ,455
$t28	b-tp	v	0 ,456
$t29	b-tp	v	0 ,457
$t30	b-tp	v	0 ,458
$t32	f-tp	v	0 ,459
$t33	f-tp	v	0 ,463
$t34	f-tp	v	0 ,467
$t35	f-tp	v	0 ,471

图 6.3-2 四元式序列-优化后

四元式

```
0 ( start,main   ,        ,        )
1 ( =    ,67     ,        ,ch      )
2 ( =    ,1      ,        ,[f0:->t7])
3 ( =    ,0      ,        ,$t2     )
4 ( =    ,18     ,        ,->t7    )
5 ( wh   ,       ,        ,        )
6 ( <=   ,$t2    ,10      ,$t8     )
7 ( fj   ,$t8    ,        ,        )
8 ( =    ,10     ,        ,x       )
9 ( =    ,0      ,        ,$t10    )
10 ( =    ,80     ,        ,$t12    )
11 ( =    ,18     ,        ,->t18   )
12 ( +    ,$t2    ,[f0:->t18],$t19    )
13 ( *    ,4      ,$t19    ,$t20    )
14 ( +    ,$t12   ,$t20    ,$t21    )
15 ( +    ,$t10   ,$t21    ,$t22    )
16 ( +    ,18     ,$t22    ,->t23   )
17 ( =    ,0      ,        ,[f0:->t23])
18 ( >=   ,$t2    ,2       ,$t25    )
19 ( !=   ,$t2    ,4       ,$t26    )
20 ( &&   ,$t25   ,$t26    ,$t27    )
21 ( fj   ,$t27   ,        ,        )
22 ( wh   ,       ,        ,        )
23 ( <=   ,ch     ,90      ,$t28    )
24 ( ==   ,[f0:->t7],1       ,$t29    )
25 ( ||   ,$t28   ,$t29    ,$t30    )
26 ( fj   ,$t30   ,        ,        )
27 ( =    ,68     ,        ,ch      )
28 ( =    ,0      ,        ,bol     )
29 ( jmp  ,       ,        ,        )
30 ( =    ,65     ,        ,ch      )
31 ( ie   ,       ,        ,        )
32 ( +    ,y      ,x       ,$t32    )
33 ( +    ,$t32   ,4       ,$t33    )
34 ( +    ,$t33   ,5       ,$t34    )
35 ( -    ,$t34   ,10.000000,$t35    )
36 ( +    ,$t35   ,2.000000,y       )
37 ( =    ,2      ,        ,i       )
38 ( jmp  ,       ,        ,        )
39 ( =    ,3      ,        ,x       )
40 ( =    ,3      ,        ,y       )
41 ( end  ,main   ,        ,        )
```

2、目标指令、活动记录展示如下图 6.3-3 所示。未进行四元式优化时目标指令共 96 条（过长不在此进行图片展示），优化后的目标指令共 63 条。

图 6.3-3 优化后的目标指令、活动记录展示

目标指令

```
 0 (start, main,[New 475] )
 1 (LD  , R ,67          )
 2 (ST  , R ,[f0:16]     )
 3 (LD  , R ,1           )
 4 (ST  , R ,[f0:->t7]   )
 5 (LD  , R ,0           )
 6 (LD  , R ,18          )
 7 (ST  , R ,->t7        )
 8 (LD  , R ,[f0:418]    )
 9 (LE  , R ,10          )
10 (FJ  , R ,[ord 60]    )
11 (LD  , R ,10          )
12 (ST  , R ,[f0:8]      )
13 (LD  , R ,0           )
14 (ST  , R ,[f0:425]    )
15 (LD  , R ,80          )
16 (ST  , R ,[f0:429]    )
17 (LD  , R ,18          )
18 (ST  , R ,->t18       )
19 (LD  , R ,[f0:418]    )
20 (ADD , R ,[f0:->t18])
21 (MUL , R ,4           )
22 (ADD , R ,[f0:429]    )
23 (ADD , R ,[f0:425]    )
24 (ADD , R ,18          )
25 (ST  , R ,->t23       )
26 (LD  , R ,0           )
27 (ST  , R ,[f0:->t23])
28 (LD  , R ,[f0:418]    )
29 (GE  , R ,2           )
30 (ST  , R ,[f0:453]    )
31 (LD  , R ,[f0:418]    )
32 (NE  , R ,4           )
33 (AND , R ,[f0:453]    )
34 (FJ  , R ,[ord 49]    )
35 (LD  , R ,[f0:16]     )
36 (LE  , R ,90          )
37 (ST  , R ,[f0:456]    )
38 (LD  , R ,[f0:->t7]   )
39 (EQ  , R ,1           )
40 (OR  , R ,[f0:456]    )
41 (FJ  , R ,[ord 47]    )
42 (LD  , R ,68          )
43 (ST  , R ,[f0:16]     )
44 (LD  , R ,0           )
45 (ST  , R ,[f0:17]     )
46 (JMP ,   ,[ord 35]    )
47 (LD  , R ,65          )
48 (ST  , R ,[f0:16]     )
49 (ST  , R ,[f0:16]     )
50 (LD  , R ,[f0:12]     )
51 (ADD , R ,[f0:8]      )
52 (ADD , R ,4           )
53 (ADD , R ,5           )
54 (SUB , R ,10.000000 )
55 (ADD , R ,2.000000  )
56 (ST  , R ,[f0:12]     )
57 (LD  , R ,2           )
58 (ST  , R ,[f0:4]      )
59 (JMP ,   ,[ord 8]     )
60 (LD  , R ,3           )
61 (ST  , R ,[f0:8]      )
62 (LD  , R ,3           )
63 (end , main,[Release] )
```

活动记录

```
Old SP = 0
返回地址
参数个数 = 0
i       ( 0,4)
x       ( 0,8)
y       ( 0,12)
ch      ( 0,16)
bol     ( 0,17)
ar      ( 0,18)
$t2     ( 0,418)
->t7    ( 0,422)
$t8     ( 0,424)
$t10    ( 0,425)
$t12    ( 0,429)
->t18   ( 0,433)
$t19    ( 0,435)
$t20    ( 0,439)
$t21    ( 0,443)
$t22    ( 0,447)
->t23   ( 0,451)
$t25    ( 0,453)
$t26    ( 0,454)
$t27    ( 0,455)
$t28    ( 0,456)
$t29    ( 0,457)
$t30    ( 0,458)
$t32    ( 0,459)
$t33    ( 0,463)
$t34    ( 0,467)
$t35    ( 0,471)
```

7 结论

在大家的努力下，我们完成了一个简单文法的编译器的设计与实现，实现了如 if、while、for 及复合语句等多种功能，且实现了复杂的多维数组设计，成功完成了前端和后端的基本设计与实现，取得了成功。

刚开始这次编译原理课程设计的课程拿到这次的课题时，就发觉这个课题并不简单，并且课题需要实现的内容完成起来工作量巨大。等到真正开始动手操作时，才发现课题比想象中的要困难多了，不仅需要实现的功能多，将这些功能一一用代码实现更是十分困难。只是编译一个简单语言的实现就如此困难，那完整地编译一门像 C/C++、Pascal 的语言难度就更难以想象了。

团队作业，更注重团队合作。然而编写代码时，每个人的代码命名等各种编程习惯都有不同，因此互相地兼容则显得十分重要了。所以编程过程中，各类变量、函数命名统一、以及注释的重要性就凸显出来，编程时一定要养成注释的好习惯；小组成员之间也得及时保持沟通，完成进度时互相保持进度统一。

8 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.

2、美 Alfred V.Aho Ravi Sethi Jeffrey D. Ullman 著.李建中，姜守旭译.《编译原理》.北京：机械工业出版社.2003.

3、美 Kenneth C.Louden 著.冯博琴等译.《编译原理及实践》.北京：机械工业出版社.2002.

4、金成植著.《编译程序构造原理和实现技术》. 北京：高等教育出版社. 2002.

9 收获、体会和建议

成员 1（匿名）：

就我个人而言，我在这次编译原理课程设计的课程中应该算是较有收获的。因为我本身编译原理学得不是特别好，基础知识不是特别扎实。在完成课程设计内容的时候，也在不断地巩固复习之前的知识，在网上查找获取资料，获取以前未曾了解的知识。总的来说就是旧知识有所巩固，新知识有所获取，算是很大的收获了。

我更体会到团队协作的重要性：要完成这么大内容如此繁多的一个课程设计，是绝对离不开团队的作用的，在团队中，成员互相帮助、鼓励，各司其职，才能最终完成一个不错的作品。

成员 2（匿名）：

在这次课设中，通过小组内合作，我学到了很多。自己平常写代码同时一个文件搞定一切，这次在小组合作中，为了使代码容易合并，自己尝试并完成了用类完成代码编写，自己的不良习惯有所改善。同时通过小组内的合作，提高了合作能力和同学间的感情，加强了互相之间的交流。

同时我也希望在之后的课设中老师能够多多指导一下同学们，让同学们可以从以前的案例中获取经验，少走弯路。同时也应该想办法让每位同学都能得到锻炼机会，提高自己，逐步成长；能够使得合作的部分多一些，提高大家的团结协作能力。

成员 3（匿名）：

在大家的努力下，我们完成了一个简单文法的编译器的设计与实现，

实现了如 if、while、for 及复合语句等多种功能，且实现了复杂的多维数组设计，成功完成了前端和后端的基本设计与实现，取得了成功。

在设计时课本知识提供了很大帮助，通过课本 DAG 优化形式的例子，加以改造，实现了优化部分的设计，使得它能够实现基本块划分，常值表达式节省，公共子表达式节省，删除无用赋值，循环优化。就目前而言个人还是比较满意的，课设完成后也有一定的成就感。