

同济大学计算机系

编译原理课程设计报告

类 C 编译器设计与实现



学 号 1951444

姓 名 林佳奕

专 业 计算机科学与技术

授课老师 高秀芬

目录

| | | |
|---------|---------------------|----|
| 一、 | 类C编译器系统设计..... | 1 |
| 1.1 | 概述..... | 1 |
| 1.2 | 词法分析..... | 1 |
| 1.2.1 | 输入输出描述..... | 1 |
| 1.2.2 | 方法描述..... | 1 |
| 1.2.3 | DFA 展示..... | 2 |
| 1.3 | LL1 语法分析..... | 3 |
| 1.3.1 | 输入输出描述..... | 3 |
| 1.3.2 | LL1 文法..... | 4 |
| 1.3.2.1 | 消除左递归..... | 4 |
| 1.3.2.2 | FIRST 集合求法..... | 4 |
| 1.3.2.3 | FOLLOW 集合求法..... | 5 |
| 1.3.3 | 预测分析法..... | 6 |
| 1.3.3.1 | 预测分析表构造..... | 6 |
| 1.3.3.2 | 预测分析方法..... | 6 |
| 1.4 | 语义分析..... | 7 |
| 1.4.1 | 输入输出说明..... | 7 |
| 1.4.2 | 中间代码格式说明..... | 7 |
| 1.4.3 | 变量声明处理..... | 8 |
| 1.4.4 | 算术表达式和布尔表达式的处理..... | 8 |
| 1.4.5 | 赋值语句处理..... | 10 |
| 1.4.6 | 循环语句处理..... | 10 |
| 1.4.7 | 分支语句处理..... | 11 |
| 1.4.8 | 函数调用处理..... | 12 |
| 1.4.9 | 数组相关处理..... | 12 |
| 1.4.9.1 | 数组声明..... | 13 |
| 1.4.9.2 | 数组使用..... | 13 |
| 1.4.10 | 其他处理..... | 13 |
| 1.4.11 | 语义分析错误处理..... | 14 |
| 1.5 | 目标代码生成..... | 14 |
| 1.5.1 | 输入输出描述..... | 14 |
| 1.5.2 | Mips32 指令集简述..... | 14 |
| 1.5.2.1 | 寄存器使用..... | 14 |
| 1.5.2.2 | 栈帧格式..... | 15 |
| 1.5.2.3 | 指令格式..... | 16 |
| 1.5.3 | 目标代码生成方法描述..... | 16 |
| 1.5.3.1 | 大体流程..... | 16 |
| 1.5.3.2 | 指令确认..... | 16 |
| 1.5.3.3 | 寄存器分配..... | 18 |
| 1.5.3.4 | 内存地址的获取..... | 18 |
| 1.5.3.5 | 栈帧处理..... | 18 |
| 1.6 | 总结..... | 19 |

| | | |
|-------|------------------------------|----|
| 二、 | 程序实现..... | 19 |
| 2.1 | 编译器代码实现..... | 19 |
| 2.1.1 | 词法分析类 LexicalAnalyzer | 20 |
| 2.1.2 | 语法类 LL1Grammar..... | 20 |
| 2.1.3 | 语法分析类 Parser..... | 21 |
| 2.1.4 | 语义分析类 SemanticAnalyzer | 21 |
| 2.1.5 | 目标代码生成器类 AsmGenerator | 21 |
| 2.2 | 界面代码实现..... | 22 |
| 2.2.1 | 功能描述..... | 22 |
| 2.2.2 | 代码描述..... | 22 |
| 三、 | 运行结果..... | 24 |
| 3.1 | 给定程序运行结果..... | 24 |
| 3.1.1 | 打开文件..... | 24 |
| 3.1.2 | 编译..... | 26 |
| 3.1.3 | 其他功能展示..... | 28 |
| 3.2 | 自定义程序运行结果..... | 31 |
| 3.3 | 报错信息展示..... | 34 |
| 3.3.1 | 词法分析报错..... | 34 |
| 3.3.2 | 语法分析报错..... | 35 |
| 3.3.3 | 语义分析报错..... | 36 |
| 四、 | 问题及解决方案..... | 37 |
| 五、 | 心得体会..... | 38 |
| 六、 | 附录..... | 39 |

一、 类 C 编译器系统设计

1.1 概述

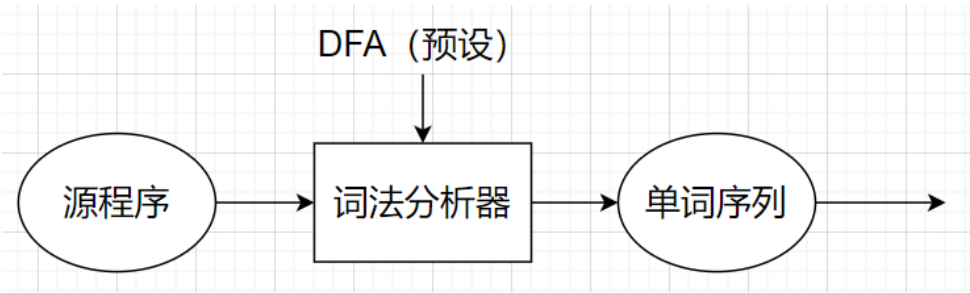
本次设计实现了一个简单 C 语言的编译器，可以生成 mips32 指令集汇编代码。具体的模块概述如下：

| | |
|--------|---|
| 词法分析 | 基于有限状态机（DFA）的词法分析器 |
| 语法分析 | 基于 LL1 的预测分析法的语法分析器 LL1 文法由外部输入得到。 |
| 语义分析 | 基于自下而上的语法制导方法的语义分析器 可生成由四元式构成的中间代码 |
| 目标代码生成 | 基于简单代码生成器的目标代码生成器 参考 Mips32 指令集架构，可以生成 Mips32 汇编代码 |

1.2 词法分析

1.2.1 输入输出描述

输入为源程序，输出为单词序列，用于语法分析。



1.2.2 方法描述

词法分析采用有限状态机（DFA）扫描实现。

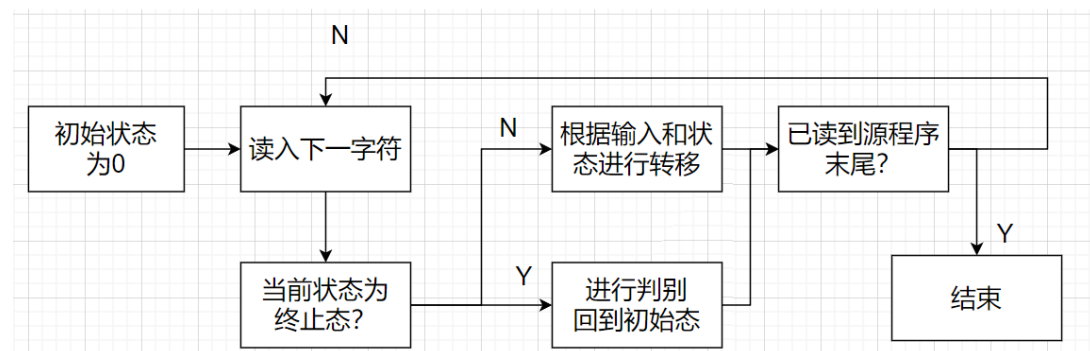
待读取完源程序后，从源程序取出一个字符，根据字符和状态来决定下一状

态。对某些状态，执行相应的动作，例如添加到单词序列、报错等。当读取到终止符‘#’（在读取源程序时，添加到源程序末尾），分析完毕。

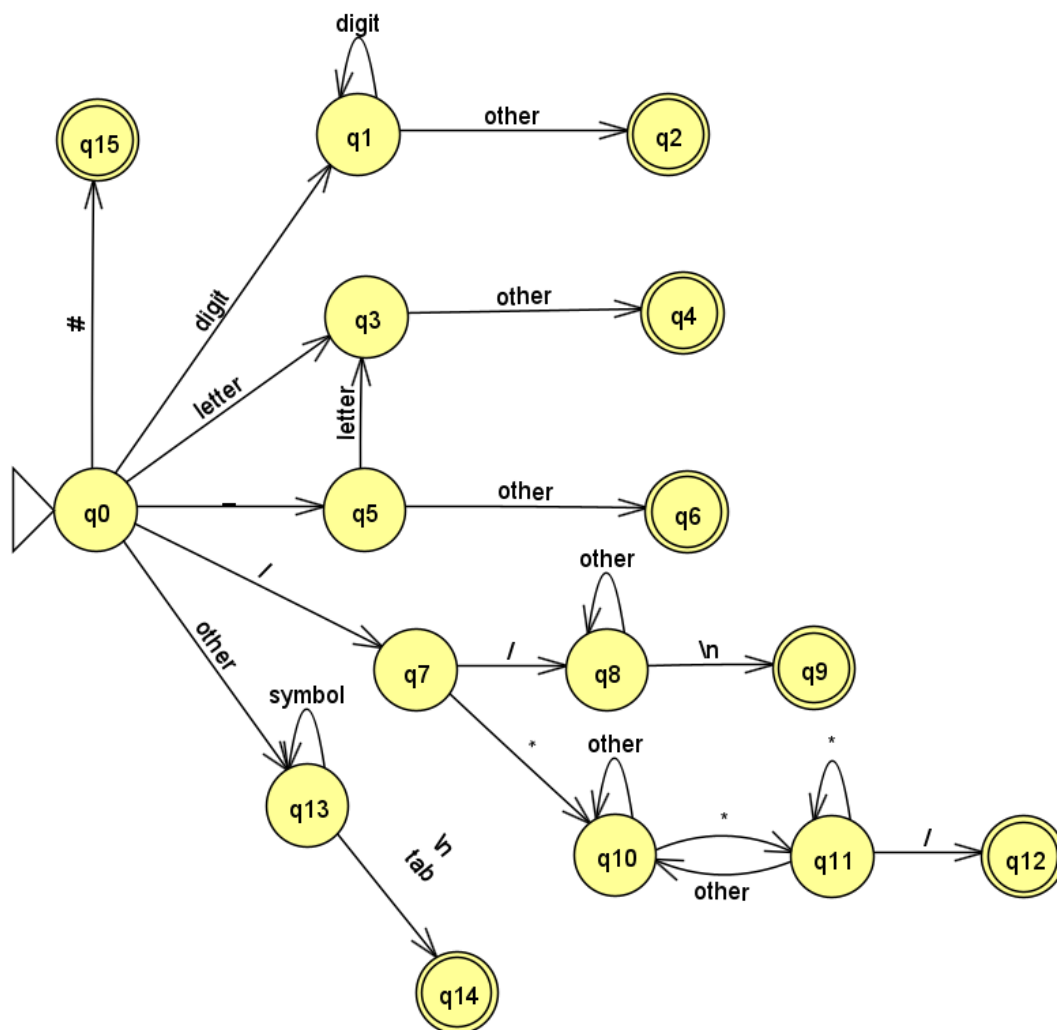
具体的程序实现为：用一个数字代表状态，0 代表初态。用 switch 语句对不同状态进行分支，内部再用 if else 语句判断不同输入情况。根据输入情况，会用一个字符串填入输入字符，直到终止态判别。

本实验中，可识别的单词符号包括：关键字（标准 C 语言的部分关键字，例如 int, void, if, else, while, return）、标识符（字母、数字和下划线组成）、常数（数字组成，即 int 类型整数）、运算符（四则运算符、比较符号和赋值符号等）、间隔符（分号、逗号、括号等）和注释符号（标准 C 语言中，用// 注释一行，用/* */注释星号内部内容）。

流程图见下



1.2.3 DFA 展示



如上图，其中 q0 为初态，双线圆为终止态（在程序中，除了 q15 状态以外，遇到其他终止态会再回到初始态）。

上图中，“digit”表示数字 0-9，letter 为字母，other 为其他字符

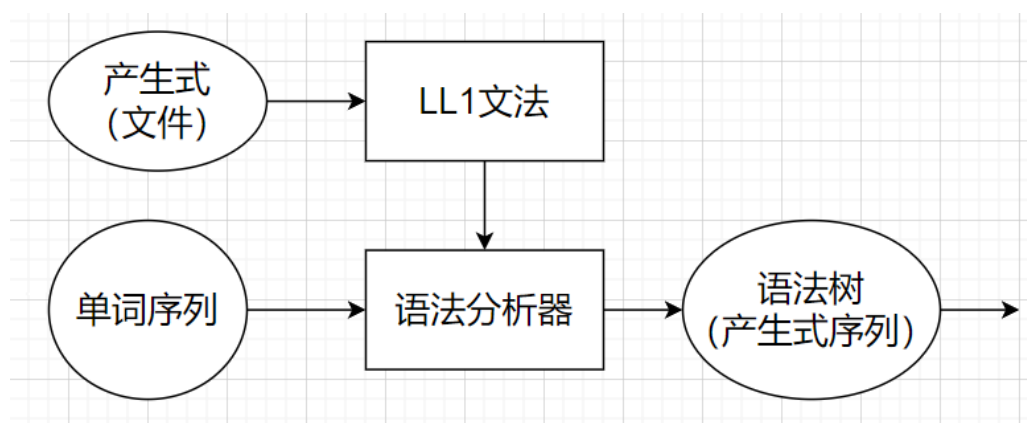
在终止态时，会对当前遇到的字符串进行判断。其中，q2 用于判别为常数，q4 用于判别为标识符或关键字（优先判断是否为关键字），q6 用于判别为出错，q9 和 q12 用于判别为注释内容，q14 判别为运算符或间隔符号，q15 判别为分析完毕。

1.3 LL1 语法分析

1.3.1 输入输出描述

语法分析输入为单词序列和 LL1 文法，输出为语法树（内含产生式序列）。

其中，LL1 文法来自于预先写好的文件，文件将设计说明书提供的文法加以修改，以便程序使用。单词序列来自于词法分析结果



1.3.2 LL1 文法

LL1 文法是自上而下分析所使用的一种文法，其构造有着一定的要求：

- (1) 产生式无直接或间接左递归
- (2) 对于一个非终符的所有产生式，其所有产生式的右部的 FIRST 无重复元素
- (3) 对于一个非终结符，如果存在着形如 $A \rightarrow \epsilon$ 的产生式，则要求非终符 A 的 FIRST 集合和 FOLLOW 集合交集为空

1.3.2.1 消除左递归

这一点在修改 LL1 文法的时候已经保证，具体的验证可以使用拓扑排序，这里不再作为重点叙述。

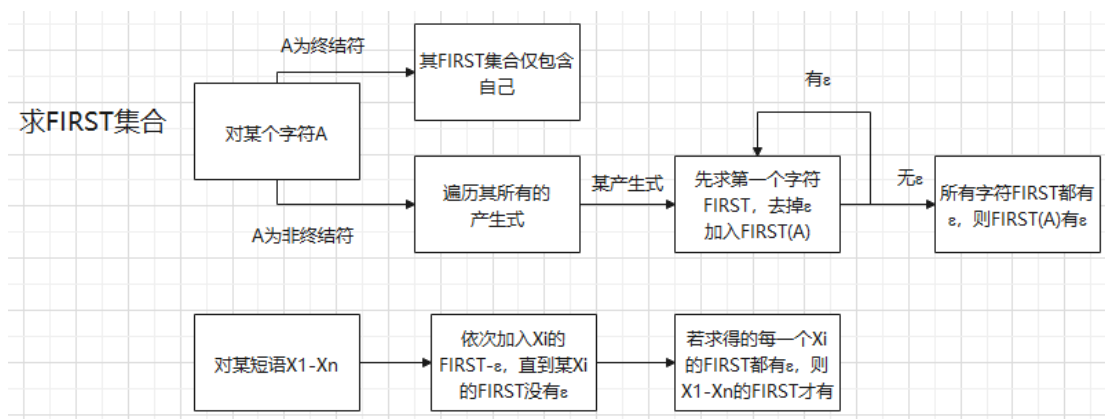
1.3.2.2 FIRST 集合求法

这里不再从理论上描述 FIRST 集合的求法，而专注于程序实现。

我们知道，对于 FIRST 集合，若想求某个产生式右部的 FIRST，需要遍历产生式右部的每一个字符。首先求第一个字符的 FIRST，如果他的 FIRST 有空字符，我们才去看第二个字符，否则不去看。并且当且仅当每个字符的 FIRST 都有空字符的时候，我们才认为产生式右部的 FIRST 才有空字符。

而对于某个字符的 FIRST 集合，若是终结符，则是其自身。否则，要看其每个产生式的右部，使用上面的方法去求。因此有可能要求非终结符 A 的 FIRST，需要先求 B 的 FIRST，而很可能要求 B 的 FIRST，又先要求 C 的 FIRST。

这实际上是一种依赖关系，不过幸运的是，可以证明 LL(1) 文法是不会产生环状的依赖关系，即 A 和 B 互相依赖。因此用深度优先搜索加上记录就可以求出每个非终结符的 FIRST。

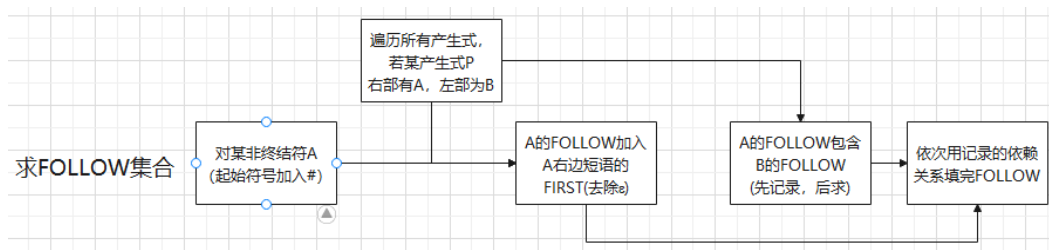


1.3.2.3 FOLLOW 集合求法

若是 FOLLOW，则更为复杂。因为想求某个非终结符的 FOLLOW 集合，需要遍历所有的产生式，看其右部有无该终结符，若有，则该终结符的 FOLLOW 包含该终结符的右边部分的 FIRST 集合。若该终结符位于某产生式末尾，或是其右部的所有字符都可以推出空字符（即某种意义上也是最后一个字符），则还需要知道该产生式的左部的 FOLLOW。因此可以看到，相较于 FIRST，FOLLOW 集合更难求。

所以我的办法是，先遍历所有的产生式，用非依赖关系先扩充部分非终结符的 FOLLOW 集合，再记录非终结符之间关于 FOLLOW 的依赖关系。接下来，用依赖关系，递归地补全所有非终结符的 FOLLOW 集合。

幸运的是，LL1 文法同样不会出现环状的依赖关系，因此可以递归的求出。



1.3.3 预测分析法

1.3.3.1 预测分析表构造

课本和课件上已经给出了预测分析表的定义，这里主要是描述具体的实现。

对于每一个非终结符 A ，输入终结符 x ，我们可以得到最多一个产生式，记为 $P(A, x)$ 。那么，对于所有以 A 为左部的产生式 $A \rightarrow \alpha$ ，我们进行如下处理：

(1) 先求出 $FIRST(\alpha)$ ，记为 F ，然后对任意 a 属于 F (a 不是空字符)，令 P

$$(A, a) = A \rightarrow \alpha$$

(2) 如果空字符在 F 中，则需要求出 $FOLLOW(A)$ ，记为 F' 。然后对任意 b 属

于 F' ，令 $P(A, b) = A \rightarrow \alpha$ 。

依次处理所有的产生式。若对某个 A, x ，没有求出对应的产生式，说明正确的归约不应该出现这种情况，若遇到了只能说是归约出错。

1.3.3.2 预测分析方法

在构造完预测分析表后，便可以进行预测分析。

在预测分析方法中，我们使用一个分析栈存放终符或非终符。初始依次放入终止符 $\#$ 和起始符号。

在分析中，对以下五种情况进行分支处理

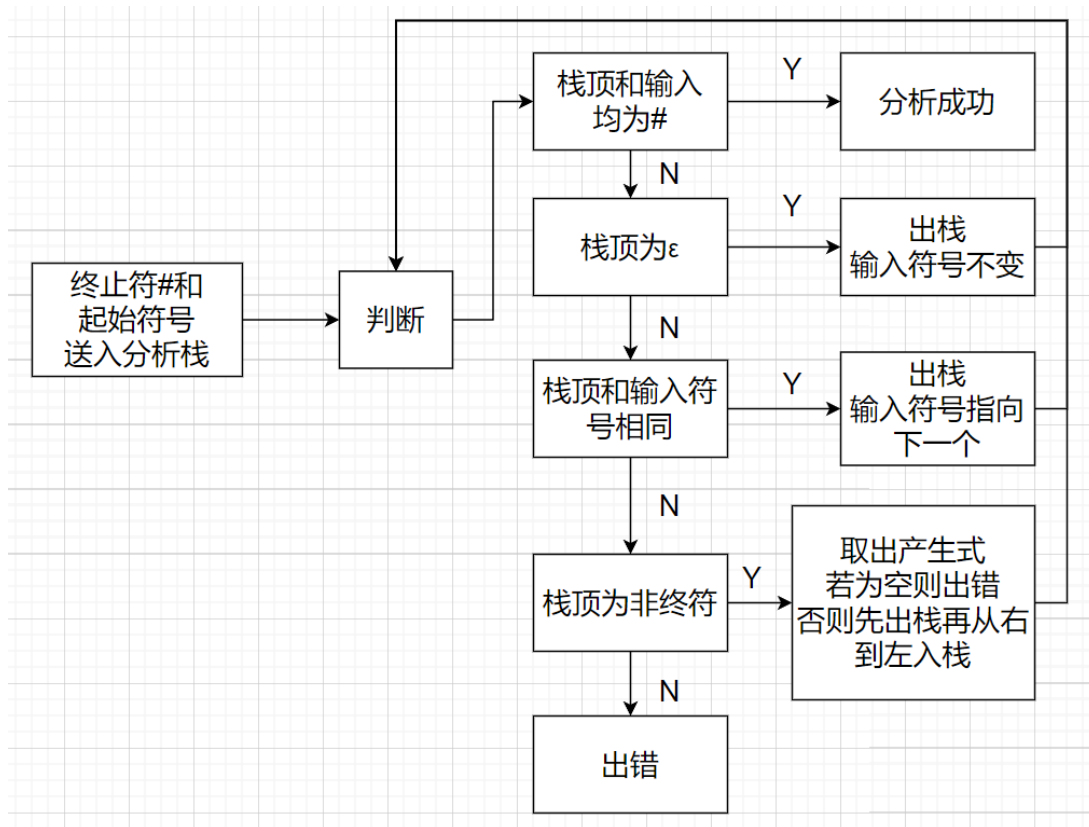
(1) 栈顶是终止符 $\#$ ，输入符号是 $\#$ ：分析成功

(2) 栈顶是 ϵ ，输入符号是任意：出栈，输入符号不变，即跳过

(3) 栈顶和输入符号相同，但不是终止符：出栈，输入下一符号。

(4) 栈顶是非终符，输入任意：使用预测分析表取出将要使用的产生式。若产生式为空，意味着语法分析失败。否则，先出栈，再将产生式右部从右到左依次入栈。

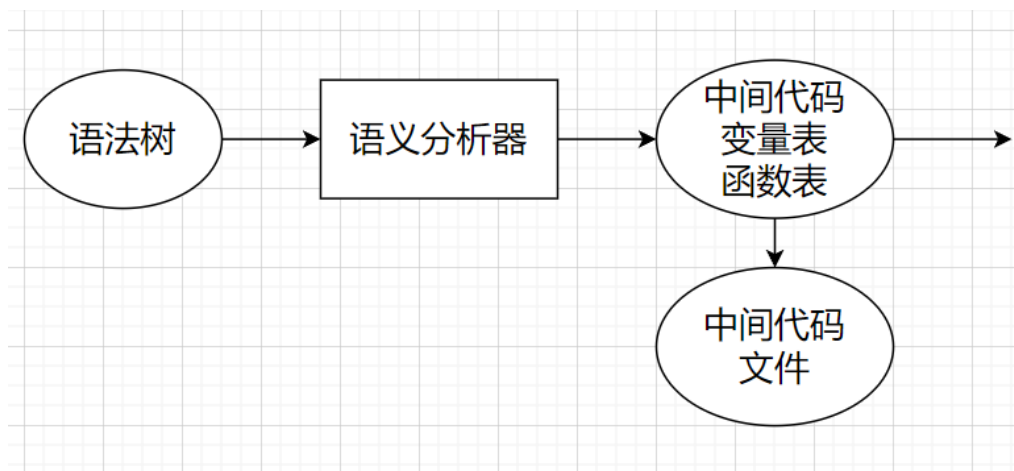
(5) 其他情况视为出错。



1.4 语义分析

1.4.1 输入输出说明

语义分析输入为语法分析中生成的语法树，输出为四元式序列和变量表、函数表等内容，用于目标代码生成。四元式序列可以写入到文件中。



1.4.2 中间代码格式说明

中间代码为四元式，格式为：< op , arg1 , arg2 , result >

由于四元式在某种程度上也可以理解为汇编代码，以下介绍中也会将四元式称为“指令”。

其中 op 为操作类型，一般是符号或者是对应的 mips 汇编指令名，例如“beq”表示相等跳转。

arg1 和 arg2 为操作数，如果是一操作数指令则不用 arg2。

result 为目的数，存放运算结果。跳转类指令用于指明跳转地址相对位移。

1.4.3 变量声明处理

本实验中，变量的声明包括整数的声明和数组的声明，以及全局变量声明和局部变量的声明。

涉及到的主要产生式包括：

```
InVarDecl -> $int $id ArrayDecl
ArrayDecl -> $[ $const $] ArrayDecl
ArrayDecl -> $eps
```

在翻译以上产生式的时候，需要判断 ArrayDecl 是否推出了数组部分，若是，则声明为数组，否则为变量。

变量声明不涉及到中间代码生成，只需要在变量表中记录即可。（将未赋值的变量初始化为 0）。若是数组，则需要记录数组各维的宽度（各维下限为 0）。

由于翻译模式为自下而上翻译，因此局部变量的翻译要提前于函数声明的翻译，因此翻译时，将变量表中的作用域记为“current”，待函数翻译完后再将 current 改为函数名称。若为全局变量翻译，则作用域记为“public”。

1.4.4 算术表达式和布尔表达式的处理

本实验中，算术表达式和布尔表达式视为同样的处理，即布尔表达式的返回值为 1 或 0。涉及到的布尔表达式仅包括比较运算。

涉及到的主要产生式包括：

```
Expr -> AddExpr RelopExpr
```

```

RelopExpr -> Relop AddExpr
RelopExpr -> $eps
Relop -> $<
Relop -> $<=
Relop -> $>
Relop -> $>=
Relop -> $==
Relop -> $!=

AddExpr -> MulExpr AddExpr1
AddExpr1 -> $+ MulExpr AddExpr1
AddExpr1 -> $- MulExpr AddExpr1
AddExpr1 -> $eps

MulExpr -> Factor MulExpr1
MulExpr1 -> $* Factor MulExpr1
MulExpr1 -> $/ Factor MulExpr1
MulExpr1 -> $eps

Factor -> $const
Factor -> $( Expr $)
Factor -> $id Factor1

Factor1 -> CallList
Factor1 -> ArrayPart

ArrayPart -> $eps
ArrayPart -> $[ Expr $] ArrayPart

```

例如算术表达式 $1+2*3$ ，会先翻译出因子 (Factor) 1、2 和 3，再翻译 $2*3$ ，最后是 $1+2*3$ 。因此在翻译时，上述涉及到的 Factor 和 Expr 相关的非终结符都要记录所使用的值，或是常数，或是临时变量，或是局部变量。

此外还要记录运算类型，例如翻译 $\text{MulExpr1} \rightarrow \$* \text{Factor MulExpr1}$ 时，若右部的 MulExpr1 非空，则 Factor 和 MulExpr1 实际上蕴含了一个算式，需要将其翻译出来。

1.4.5 赋值语句处理

本实验中，赋值运算的处理涉及到的主要产生式包括：

| |
|---|
| $\text{AssignCode} \rightarrow \$id \text{ ArrayPart } \$= \text{Expr}$ |
|---|

处理是较为简单的，只需要取出右部 Expr 的值，赋值到左部 id 即可（id 是局部或全局变量）。如果是数组部分，需要存入数组某一格中，此部分留到数组处理讲解。

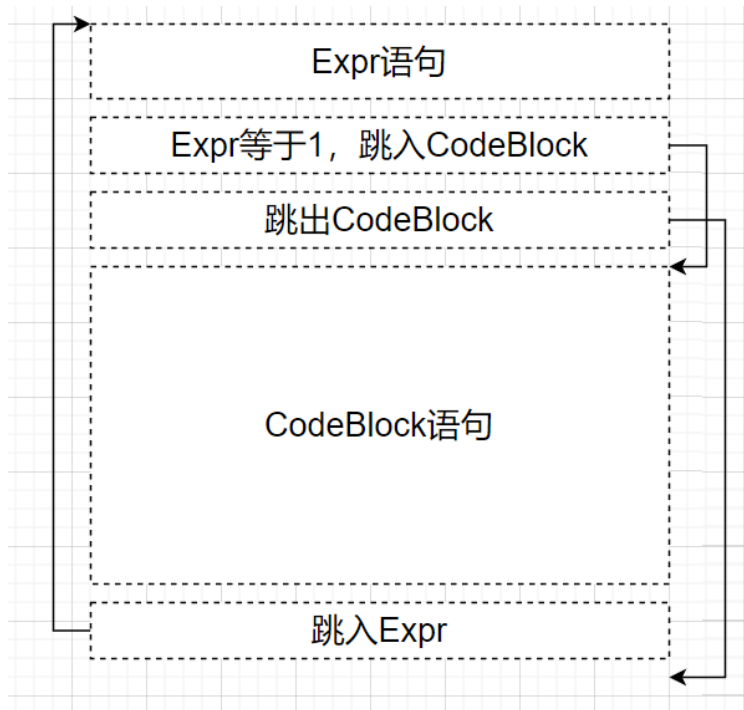
1.4.6 循环语句处理

本实验中支持使用 while 循环，涉及到的产生式包括：

| |
|--|
| $\text{WhileCode} \rightarrow \$while \ $(\text{Expr } \$) \text{ CodeBlock}$ |
|--|

根据自下而上语法分析，Expr 部分和 CodeBlock 部分先被翻译。由于本实验中采用数值的方式处理布尔表达式，因此循环的进入条件是 Expr 对应的值为 1。

因此在 Expr 的产生式后面加入两条四元式：等于 1 跳入 CodeBlock 首地址、跳入 CodeBlock 最后一句的后面（注意：是 CodeBlock 后面加入的跳转语句后面），这是实现了循环的跳入和跳出。此外，还需要加入一条四元式：跳入 Expr 首地址，这是用来实现循环。



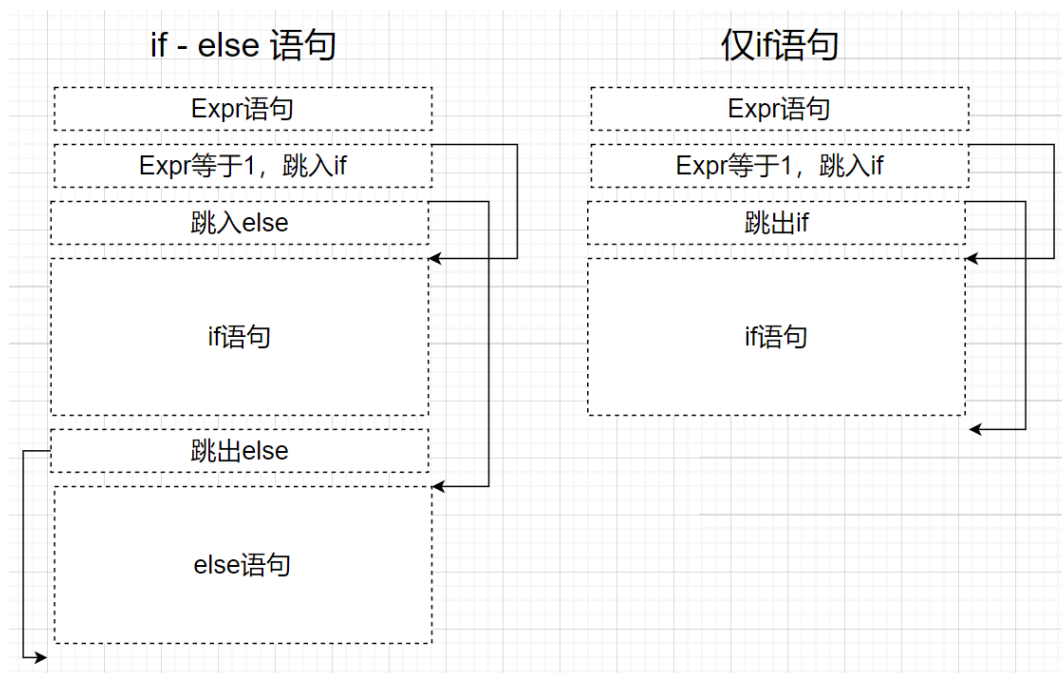
1.4.7 分支语句处理

本实验支持分支语句 if else，涉及的产生式包括：

```

IfCode -> $if $( Expr $) CodeBlock ElseCode
ElseCode -> $else CodeBlock
ElseCode -> $eps
  
```

类似 while 语句，Expr，CodeBlock 语句先被翻译，因此同样要先在 Expr 语句后面加入两条产生式，分别是跳入 if 为真的 CodeBlock 和 if 为假的 CodeBlock（若没有 if，则跳出 if 为真的 CodeBlock）。此外，如果有 else 部分，则需要在 if 为真的后面加入一条跳出 else 语句，以避免执行了 else 部分。



1.4.8 函数调用处理

本实验支持函数调用，涉及到的主要产生式包括：

```

Factor -> $id Factor1
Factor1 -> CallList
Factor1 -> ArrayPart
CallList -> $( RPara $)
RPara -> RParaList
RPara -> $eps
RParaList -> Expr RParaList1
RParaList1 -> $, Expr RParaList1
  
```

在翻译 RParaList 的时候，遇到 Expr 则产生四元式：将 Expr 代表的值送入堆栈，这是参考了栈帧的处理方式。翻译完 RPara 后，在 Factor 遇到标识符 id 时，则生成一条四元式：调用 id 对应的函数。

1.4.9 数组相关处理

本实验支持数组声明和调用。

1.4.9.1 数组声明

数组声明涉及到的主要产生式包括：

```
InVarDecl -> $int $id ArrayDecl
ArrayDecl -> $[ $const $] ArrayDecl
ArrayDecl -> $eps
```

如前面所说，数组声明不需要生成中间代码，只需要记录各维宽度即可。

1.4.9.2 数组使用

数组使用涉及到的主要产生式包括：

```
Factor -> $id Factor1
Factor1 -> ArrayPart
ArrayPart -> $eps
ArrayPart -> $[ Expr $] ArrayPart
```

数组使用与声明不同的地方是，允许各维不是常量，并且实际上只包含了一个值，因此需要先把位移算出来。具体的算法如下：

对于数组 $a[x_1][x_2]\cdots[x_k]$ ，计算位移如下：（设各维上限为 n_i ）

$$offset = (x_k + n_k \times (x_{k-1} + n_{k-1} \times (x_{k-2} + \cdots)))$$

显然，这是一个含有多个括号的式子，但是处理起来较为方便，我们可以记录下各维的位移量，然后从第一维开始，先加上位移量，再乘上第二维的宽度，重复直到处理完所有维度。

之后，若是数组取数，则产生一条四元式：数组取数，并记下数组名，位移和结果存放的变量（中间变量）。若是数组存数，则产生一条四元式：数组存数，记下数组名，位移和存数的来源。

1.4.10 其他处理

翻译完后，判断有无 **main** 函数，有则调用之（同函数调用），没有则直接结束程序。

1.4.11 语义分析错误处理

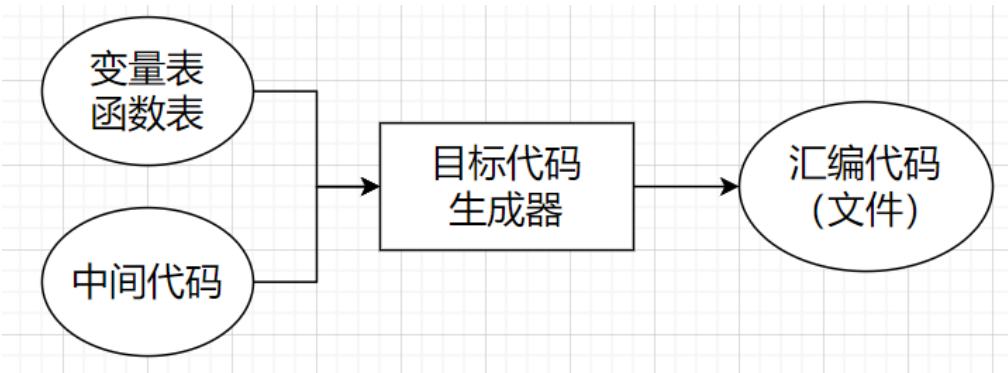
本程序可以指出以下错误：

- (1) 变量、数组未命名
- (2) 变量、数组重命名
- (3) 函数未命名
- (4) 函数重命名
- (5) 函数参数个数不匹配
- (6) 数组维数错误
- (7) 函数返回值错误
- (8) 引用返回类型为 void 的函数

1.5 目标代码生成

1.5.1 输入输出描述

输入为语义分析生成的函数表、变量表和中间代码，输出为 mips32 汇编代码（后缀为 asm，可在汇编模拟器程序 Mars 上汇编执行）



1.5.2 Mips32 指令集简述

1.5.2.1 寄存器使用

Mips32 架构中，有 32 个通用寄存器，其具体的功能如下：

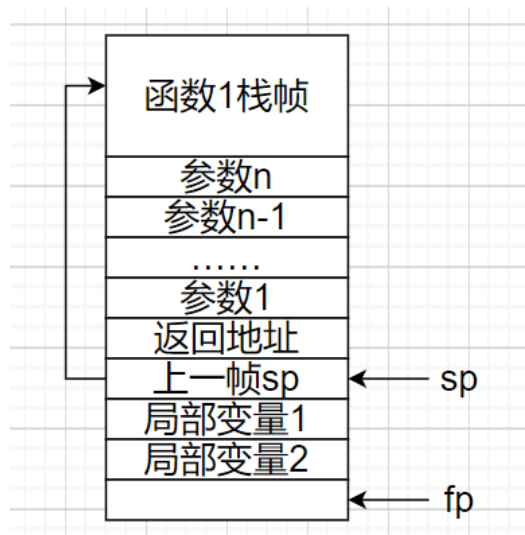
| 编号 | 助记符 | 用法 |
|----|-----|----|
|----|-----|----|

| | | |
|-------|-------|---------------------|
| 0 | zero | 恒为 0 |
| 1 | at | 汇编器的暂时变量你 |
| 2-3 | v0-v1 | 子函数返回值 |
| 4-7 | a0-a3 | 子函数参数 |
| 8-15 | t0-t7 | 暂时变量，不需要保存 |
| 24-25 | t8-t9 | 暂时变量，不需要保存 |
| 16-23 | s0-s7 | 子函数寄存器变量，需要保存和恢复 |
| 26-27 | k0-k1 | 中断、异常用 |
| 28 | gp | 全局指针（本实验中，指向首个全局变量） |
| 29 | sp | 堆栈指针（栈帧基址） |
| 30 | fp | 框架指针（栈顶） |
| 31 | ra | 返回地址 |

其中，本实验并没有用到所有寄存器，只用到了 0、1、2-3、8-15、24-25、28-31 寄存器，后文会介绍具体用法，会略有区别。

1.5.2.2 栈帧格式

栈帧格式参考了 Unix 操作系统栈帧格式以及 Mips 指令集架构的栈帧格式。



上方为高地址，下方为低地址，fp、sp 为寄存器，指向内存地址。

在本实验中，约定初始 sp=0x1001fffc, fp=0x1001fffc, gp=0x10010000。这会在调用 main 函数之前设置好。

1.5.2.3 指令格式

Mips 汇编指令格式较为简单，只有三种格式指令：R 类型，I 类型和 J 类型。

对于汇编代码而言，不需要了解三种类型的具体格式，只需要了解指令操作类型和操作数个数即可。R 指令一般为 1-3 个操作数，I 指令为 3 个操作数，包括一个常数两个寄存器。J 指令包括一个操作数，为跳转地址。

由于 Mips 汇编属于 RISC 指令集，只有 load 或 store 指令可以访问存储器，因此操作数或为寄存器或为常数，也就要求合理分配寄存器。

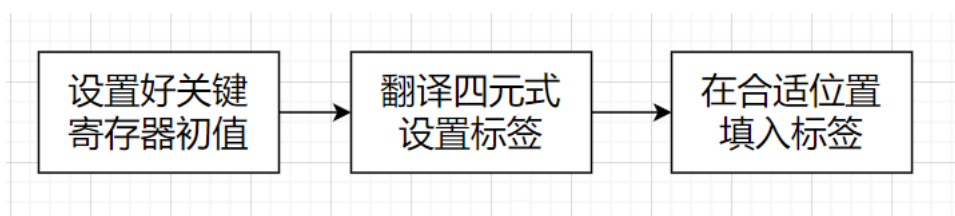
1.5.3 目标代码生成方法描述

1.5.3.1 大体流程

在翻译中间代码之前，先设置好寄存器的值。设置值如上。

之后，对于每一条四元式，根据四元式操作类型选取合适的汇编代码，并填充操作数，若是跳转类型，则设置标签，并记录标签位置。

最后，填入跳转标签和函数名到正确的位置。



1.5.3.2 指令确认

本实验中，涉及到的四元式类型包括：四则运算，比较运算，赋值运算，有条件跳转，无条件跳转，数组存取数，函数调用，函数返回，程序结束。

四则运算中，加减运算可以用 I 类型指令处理，从而少分配一个寄存器，而乘法除法较为复杂，在 Mips 架构中，乘法指令 mult 会将结果的高 32 位存入 hi 寄存器，低 32 位存入 lo 寄存器；除法指令 div 会将余数存入 hi 寄存器，商存入 lo 寄存器。乘法和除法都用 lo 寄存器，因此还需要用 mflo 指令取出 lo 寄存器的值存入通用寄存器再做处理。

比较运算使用 set 系列指令（例如 slt 指令），此条指令会比较两个寄存器的值，若比较条件为真则赋值为 1，否则为 0（例如 slt \$t3,\$t2,\$t1 若 t2 比 t1 小，则 t3=1，否则 t3=0）。部分 set 指令可以用 I 类型指令优化（例如 slti），从而少用一个寄存器。

赋值运算主要是存入变量中，因此往往伴随着变量写入内存。首先还是要为变量分配寄存器，再用 move 指令或 addi 指令存入变量对应的寄存器，最后视情况在合适的时候写入内存。

跳转指令包括有条件跳转和无条件跳转。有条件跳转中，本实验仅使用 beq 指令和 bne 指令，两种指令为 I 类型指令，需要申请寄存器，并设置标签。而无条件跳转指令则为 j 指令，只需要获取标签即可。

数组存取数和变量存取数不同的地方在于，数组存取数多了个数组内位移，而数组内位移不容易用数字表示，而存取数指令又是 I 类型，需要一个常数。因此把变量在栈帧内的位移当做常数（这个通过访问变量表很容易得到，是静态的），再把栈帧基址或全局变量基址加上数组位移作为新基址，用于 load 指令和 store 指令。

函数调用中，首先要将变量入栈，这个通过 store 指令和改变栈顶来实现。调用则用 jal 指令实现。在 jal 指令之后要修改栈顶，以消除参数区域，恢复到调用之前的状态。

调用函数的时候，首先要按照栈帧的格式构造栈帧，这个主要是通过 store 指令和修改栈顶和栈帧基址实现的。

函数返回时，首先要取出上一栈帧的基址和返回地址，然后用 jr 指令返回。（返回地址存入 ra 寄存器中）。

程序结束借用系统调用实现，设置的参数为 10。

总结用到的主要指令如下：

| | |
|-------|---------------------------------------|
| 四则运算 | add, addi, sub, subi, mult, div, mflo |
| 比较运算 | slt, slti, sgt, seq, sne, sge, sle |
| 赋值运算 | move, li, sw |
| 有条件跳转 | beq, bne |
| 无条件跳转 | j |

| | |
|-------|---------|
| 数组存取数 | lw, sw |
| 函数调用 | jal |
| 函数返回 | jr |
| 程序结束 | syscall |

1.5.3.3 寄存器分配

对于操作数类型，进行判断：

- (1) 常数类型：如果常数是 0，则分配 0 号寄存器。否则如果运算类型不能用 I 类型优化，则为其分配 \$t9 寄存器，否则暂不分配，在指令中处理。
- (2) 中间变量：不需要写入内存中，因此在有新中间变量出现时便会失效，因此简单分配 \$t0-\$t8 寄存器中的一个（以 FIFO 的方式分配）。
- (3) 局部变量/全局变量：若已存于某个寄存器中，则直接返回。否则分配。如果占用的寄存器存放的是局部或全局变量，则将其写回内存再分给其他操作数。
- (4) 函数返回值：分配 v0 寄存器。

因此主要的分配值 t0-t8 寄存器，这些寄存器的分配使用 FIFO 的方式，好处是简单，且考虑到临时变量较多，这样的处理方式比 LRU 的处理方式更快一些。

1.5.3.4 内存地址的获取

内存地址的获取分为三种情况：

- (1) 局部变量（包括参数）：基址为 sp 寄存器，若是局部变量，则位移为 $-4*n$ （n 为第 n 个声明局部变量），若是参数，则位移为 $4*(n+1)$ （n 为第 n 个参数）。
- (2) 全局变量：基址为 gp 寄存器，位移为 $4*n$ （n 为第 n 个全局变量）。
- (3) 数组：计算位移时还要算上数组内位移。

注：数组视为一次性声明多个变量，这样便于处理局部变量的位移。

1.5.3.5 栈帧处理

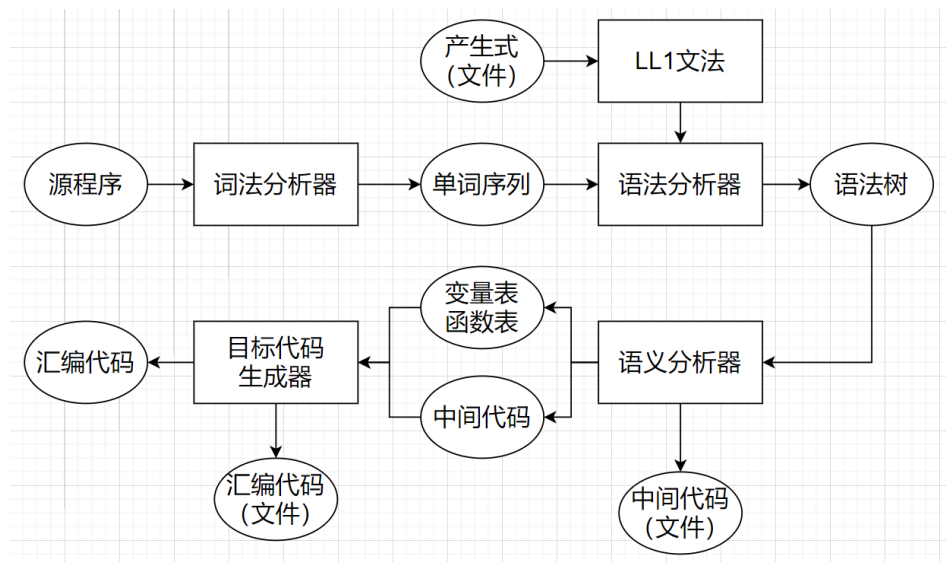
栈帧的建立分为栈帧的建立和栈帧的撤销。

栈帧建立：在调用函数前，逆序送入参数，调整栈顶；在函数开头，存入返回地址（存入 **ra** 寄存器中），存入上一帧的 **sp**（即将 **sp** 入栈），修改 **sp**，使其指向内存中上一帧的 **sp**。修改 **fp**，为局部变量预留地方。

栈帧撤销：函数返回前，先取出返回地址，再令 $fp=sp$ ，取出上一帧的 sp 值，送入 sp 寄存器，执行 jr 指令，再调整 fp 寄存器消除参数部分。

1.6 总结

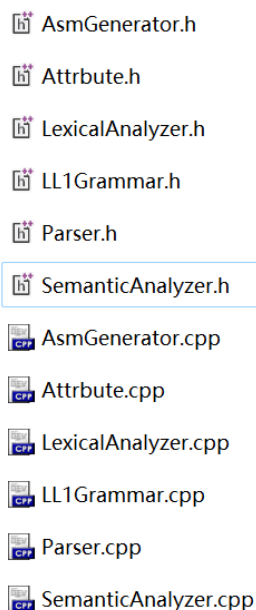
以下是本次实验中编译器部分各模块的运行关系。



二、程序实现

2.1 编译器代码实现

编译器代码涉及到的头文件如下:



AsmGenerator.h
Attribute.h
LexicalAnalyzer.h
LL1Grammar.h
Parser.h
SemanticAnalyzer.h
AsmGenerator.cpp
Attribute.cpp
LexicalAnalyzer.cpp
LL1Grammar.cpp
Parser.cpp
SemanticAnalyzer.cpp

2.1.1 词法分析类 LexicalAnalyzer

此类的功能是：实现词法分析，并为语法分析提供单词序列。

对应文件：LexicalAnalyzer.h, LexicalAnalyzer.cpp

在这个文件中除了此类的定义，还包含了单词结构体 **Word** 的定义，包含标识符和内容两部分，单词序列用一个 **vector** 容器盛放。

2.1.2 语法类 LL1Grammar

此类的功能是：从外部文件输入产生式，生成 **FIRST** 集合、**FOLLOW** 集合和预测分析表。为语法分析类 **Parser** 提供接口。

对应文件：LL1Grammar.h, LL1Grammar.cpp。

在这个文件中，还包含了对 **FIRST** 集合、**FOLLOW** 集合的定义。对于这两个集合，用一个结构体定义，内含一个 **set** 集合，以及相关的成员函数。在 **LL1Grammar** 类中，所有终结符的 **FIRST** 集合，**FOLLOW** 集合是一个终结符类型 (**string**) 到 **FIRST** 结构体或 **FOLLOW** 结构体的映射 (**map**)

产生式 (**Production**) 也做出了定义，也是一个结构体，内含左部 (**string**) 和右部 (**vector<string>**) 以及相关成员函数。

预测分析表是一个从终结符、非终结符对 (**pair**) 到产生式 (**Production**) 的

映射。

2.1.3 语法分析类 Parser

此类的功能是：进行语法分析，生成语法树。

对应的文件：Parser.h, Parser.cpp。

在这个文件中，还定义了语法树 ParserTree 类。此类主要是存放语法树的内容，包括节点和边。节点以语法分析中出现的顺序编号，用 vector 容器盛放；边则用 map 表示，键为父节点编号，值为所有子节点。此外提供相应的成员函数，可以将语法树按照 dfs, bfs 的遍历方式输出。

在 Parser 类中，包含 LL1Grammar 类和 ParserTree 类的指针。

2.1.4 语义分析类 SemanticAnalyzer

此类的功能是：进行语义分析，生成变量表，函数表和中间代码。

对应的文件：SemanticAnalyzer.h, SemanticAnalyzer.cpp。

此类需要引入头文件 Attribute.h，此头文件包含了所有非终结符的属性结构体定义（有些未用到）。在 SemanticAnalyzer 类中，每个非终结符有一个处理子函数，用于处理以其为左部的产生式。

变量表项和函数表项的结构体也定义在 Attribute.h 中。变量表项包含的内容有：变量名称、类型、作用域、第几个局部或全局变量，各维上限（仅用于数组）；函数表项包括的内容有：函数名称、返回类型、参数数量、四元式起始位置和包含的四元式个数。

2.1.5 目标代码生成器类 AsmGenerator

此类的功能是：进行目标代码生成，生成 mips 汇编代码。

对应的文件：AsmGenerator.h, AsmGenerator.cpp。

这个类实现了寄存器分配，栈帧的构造，目标代码生成等一系列功能。Mips 汇编代码用一个结构体定义，包含指令类型和操作数，并提供相应成员函数以便输出。

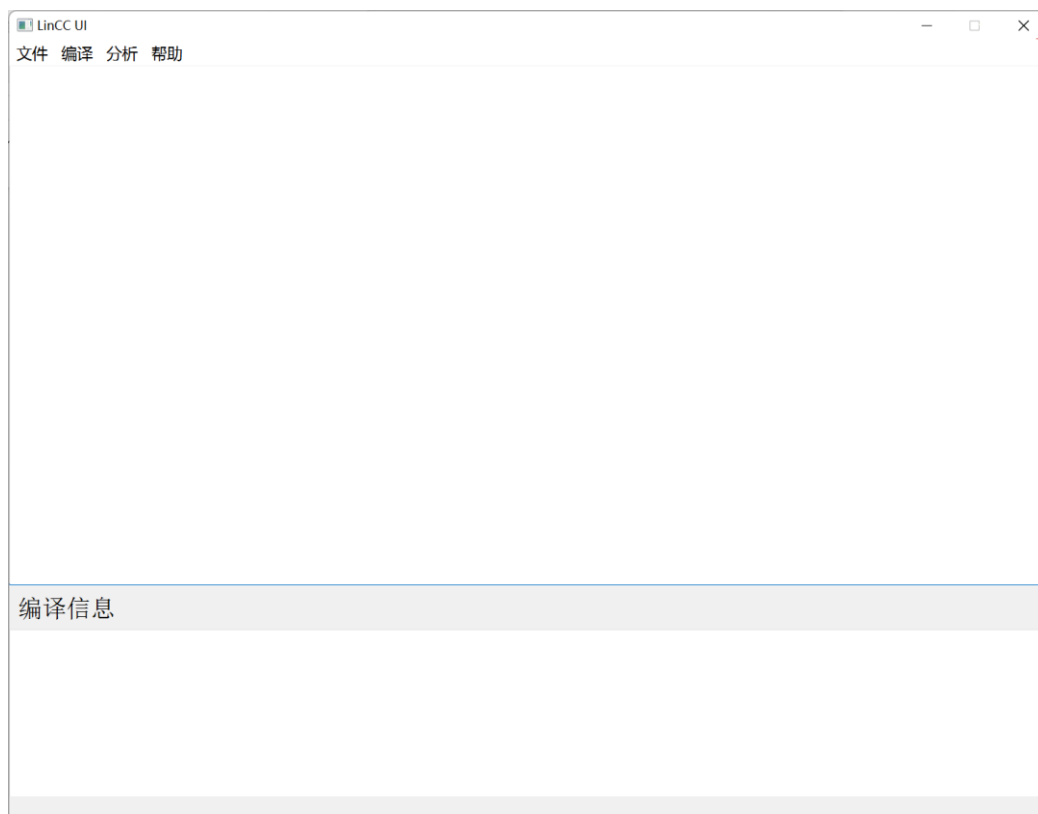
2.2 界面代码实现

2.2.1 功能描述

本实验界面实现的功能包括：

- (1) 打开关闭文件
- (2) 编译，生成中间代码和目标代码
- (3) 分析，展示词法分析、语法分析、语义分析结果以及展示 LL1 文法、预测分析表和语法树。
- (4) 帮助文档。

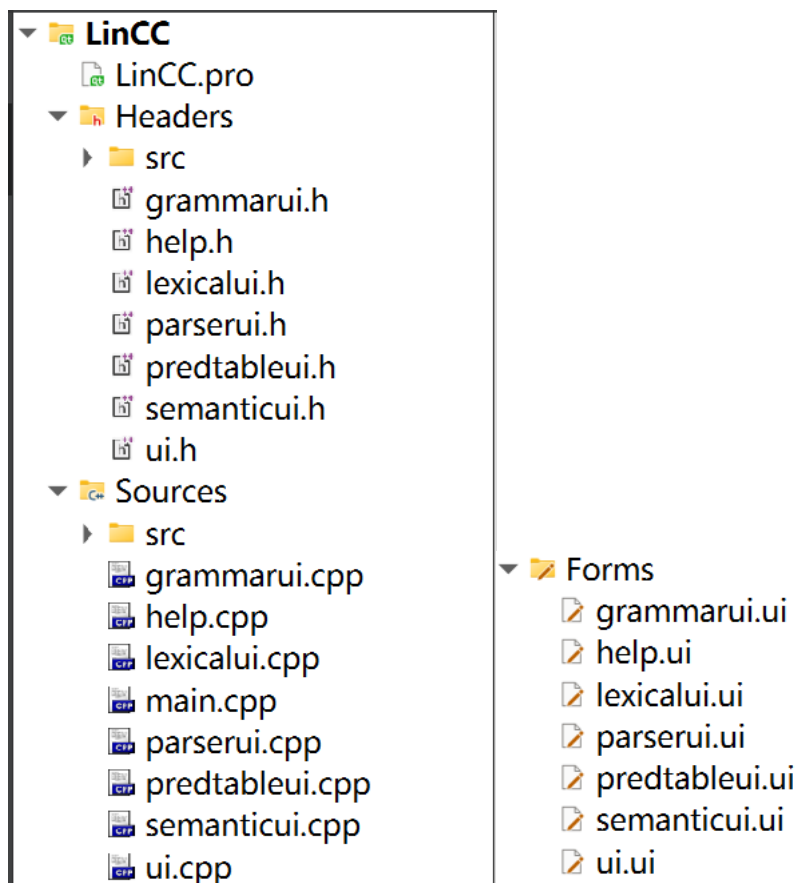
其中，(3) (4) 功能会打开新的界面，其余不会。以下是界面的样式



上方是源代码，下方是编译信息。功能框在上方。

2.2.2 代码描述

下图是界面涉及到的文件（src 文件夹是编译器代码）



可以看到，每个界面都配有一个头文件、源文件和 ui 文件，头文件和源文件包含功能的实现，ui 文件包含界面的布局。

本程序功能和对应的文件如下：

| 功能 | 文件 |
|-----------|--|
| 展示词法分析 | lexicalui.h lexicalui.cpp lexicalui.ui |
| 展示 LL1 文法 | grammarui.h grammarui.cpp grammarui.ui |
| 展示预测分析表 | predtableui.h predtableui.cpp predtableui.ui |
| 展示语法分析 | parserui.h parserui.cpp |

| | |
|---------------------|---|
| | parserui.ui |
| 展示语义分析 | semanticui.h semanticui.cpp semanticui.ui |
| 帮助文档 | help.h help.cpp help.ui |
| 打开关闭文件、编译、 展示语法树 | ui.h ui.cpp ui.h |

由于代码较长，且并非本实验的重点，因此不在进行过多介绍。

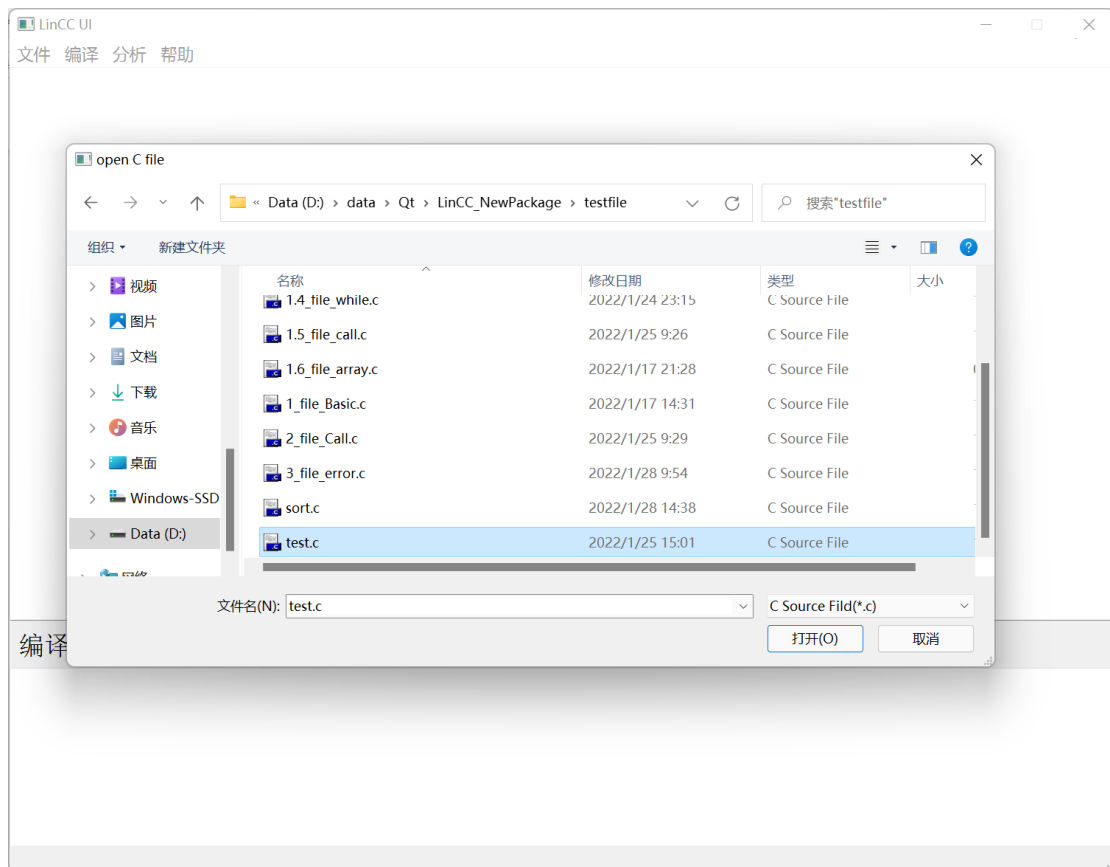
三、 运行结果

3.1 给定程序运行结果

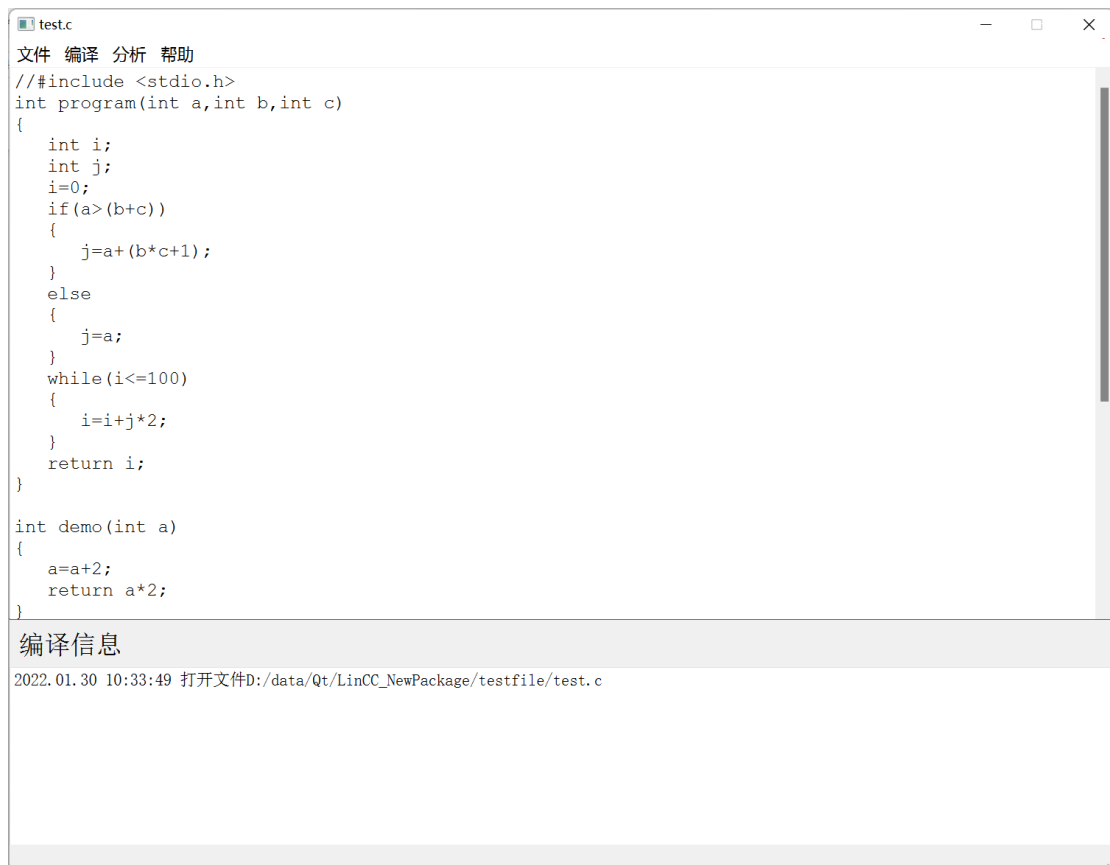
3.1.1 打开文件

将课设任务书中给出的数组翻译的程序写入到 C 源文件中，并进行修改使得能正确运行，命名为 test.c。

打开本程序，打开此文件。



如下图，程序已打开，并有相应信息显示。



3.1.2 编译

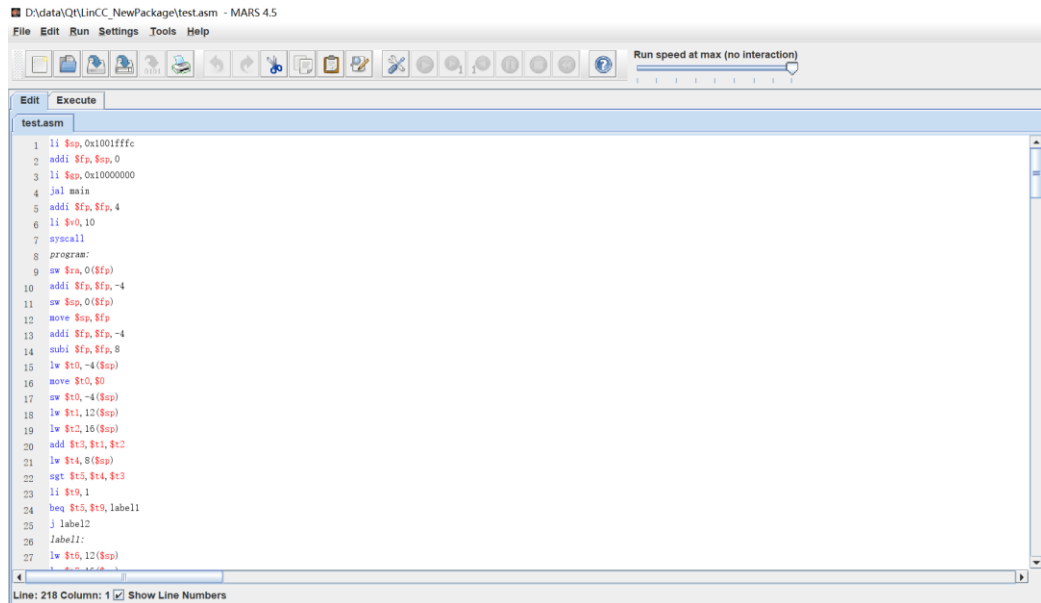
点击编译-生成目标代码，显示编译成功



文件如预期一样生成。

- test.asm
- test_Quat.txt
- Grammar.txt
- LinCC.exe
- readme.txt
- testfile
- Graphviz

这里检查生成的汇编代码。使用 Mars 程序打开之并汇编执行。



生成的汇编代码较长，这里不再展示。

汇编运行，显示运行成功。

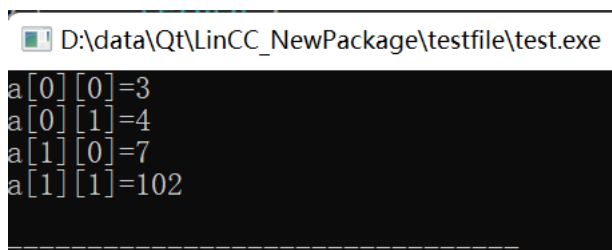
以下是部分内存单元结果

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) |
|------------|------------|------------|------------|------------|-------------|-------------|
| 0x1001ffe0 | 18 | 0 | 102 | 7 | 4 | 3 |

考虑到本编译器规定栈帧基址为 0x1000fff0, $0x1001ffe0+14=0x1001fff4$, 则上图中右部四个值恰好对应源代码中 main 函数定义的数组。(0x1001fff8 存放上一帧 sp, 0x1001fff0 存放返回地址, main 函数无参数)。

```
void main(void)
{
    int a[2][2];
    int b;
    a[0][0]=3;
    a[0][1]=a[0][0]+1;
    a[1][0]=a[0][0]+a[0][1];
    a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
}
```

对 test.c 进行少量修改，以输出数组全部值，并用 GCC 编译运行后得到如下结果

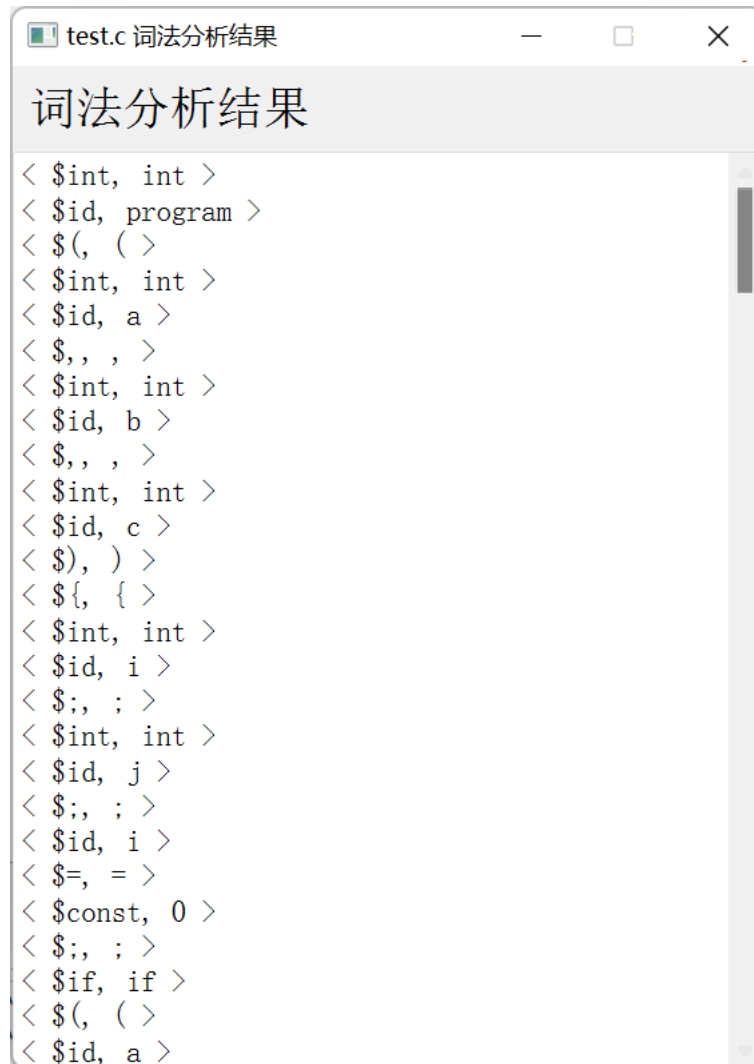


与上图中运行结果一致，这说明了编译器编译正确。

3.1.3 其他功能展示

编译成功后，依次展示其他功能。

词法分析结果：



```
< $int, int >
< $id, program >
< $[, ( >
< $int, int >
< $id, a >
< $[, , >
< $int, int >
< $id, b >
< $[, , >
< $int, int >
< $id, c >
< $), ) >
< ${, { >
< $int, int >
< $id, i >
< $[, ; >
< $int, int >
< $id, j >
< $[, ; >
< $id, i >
< $[, = >
< $const, 0 >
< $[, ; >
< $if, if >
< $[, ( >
< $id, a >
```

展示 LL1 文法：

| FIRST集合和FOLLOW集合 | | | 产生式 |
|------------------|----------------------------|---|---|
| | FIRST | FOLLOW | |
| AddExpr | \$(\$const \$id | \$!= \$) \$, \$; \$< \$<= \$== \$> \$>= \$} | AssignCode -> \$id ArrayPart \$= Expr ReturnCode -> \$return ReturnCode1 ReturnCode1 -> Expr ReturnCode1 -> \$seps WhileCode -> \$while \$(Expr \$) CodeBlock IfCode -> \$if \$(Expr \$) CodeBlock ElseCode -> \$else CodeBlock ElseCode -> \$seps Expr -> AddExpr RelopExpr RelopExpr -> Relop AddExpr RelopExpr -> \$seps Relop -> \$< Relop -> \$<= Relop -> \$> Relop -> \$>= Relop -> \$== Relop -> \$!= AddExpr -> MulExpr AddExpr1 AddExpr1 -> \$+ MulExpr AddExpr1 AddExpr1 -> \$- MulExpr AddExpr1 AddExpr1 -> \$seps MulExpr -> Factor MulExpr1 MulExpr1 -> \$* Factor MulExpr1 MulExpr1 -> \$/ Factor MulExpr1 MulExpr1 -> \$seps Factor -> \$const Factor -> \$(Expr \$) Factor -> \$id Factor1 Factor1 -> CallList Factor1 -> ArrayPart CallList -> \$(RPara \$) ArrayPart -> \$seps ArrayPart -> \$[Expr \$] ArrayPart RPara -> RParaList RPara -> \$seps RParaList -> Expr RParaList1 RParaList1 -> \$, Expr RParaList1 RParaList1 -> \$seps |
| AddExpr1 | \$+ \$- \$seps | \$!= \$) \$, \$; \$< \$<= \$== \$> \$>= \$} | |
| ArgsList | \$int | \$) | |
| ArgsList1 | \$, \$seps | \$) | |
| Argument | \$int | \$) \$, | |
| ArrayDecl | \$[\$seps | \$; | |
| ArrayPart | \$[\$seps | \$!= \$) \$* \$+ \$, \$- \$/ \$; \$< \$<= \$= \$== \$> \$>= \$} | |
| AssignCode | \$id | \$; | |
| CallList | \$(| \$!= \$) \$* \$+ \$, \$- \$/ \$; \$< \$<= \$== \$> \$>= \$} | |
| Code | \$id \$if \$return \$while | \$id \$if \$return \$while \$} | |
| CodeBlock | \$(| \$# \$else \$id \$if \$int \$return \$void \$while \$} | |

展示预测分析表

| LL1预测分析表 | | | | | |
|------------|-----------------------------|---------------------------|----------------------|----------------------|----------------------------------|
| | \$!= | \$(| \$) | \$* | \$+ |
| AddExpr | AddExpr -> MulExpr AddExpr1 | | | | |
| AddExpr1 | AddExpr1 -> \$seps | | AddExpr1 -> \$seps | | AddExpr1 -> \$+ MulExpr AddExpr1 |
| ArgsList | | | | | |
| ArgsList1 | | | ArgsList1 -> \$seps | | |
| Argument | | | | | |
| ArrayDecl | | | | | |
| ArrayPart | ArrayPart -> \$seps | | ArrayPart -> \$seps | ArrayPart -> \$seps | ArrayPart -> \$seps |
| AssignCode | | | | | |
| CallList | | CallList -> \$(RPara \$) | | | |
| Code | | | | | |
| CodeBlock | | | | | |
| CodeStr | | | | | |
| Decl | | Decl -> FuncDecl | | | |
| ElseCode | | | | | |
| Expr | | Expr -> AddExpr RelopExpr | | | |
| RPara | | | RPara -> \$seps | | |
| Factor | | Factor -> \$(Expr \$) | | | |
| Factor1 | Factor1 -> ArrayPart | Factor1 -> CallList | Factor1 -> ArrayPart | Factor1 -> ArrayPart | Factor1 -> ArrayPart |

展示语法分析结果

test.c 语法分析结果

LL1分析过程

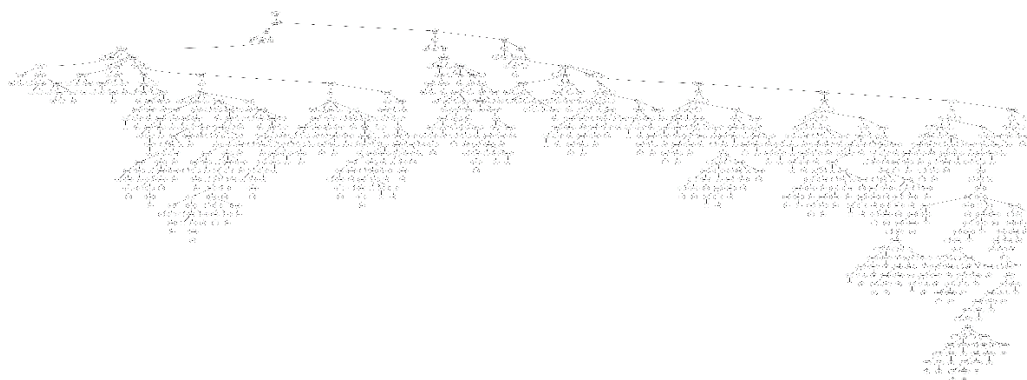
第870步, 栈顶为: \$;, 输入为: ;, 出栈
第871步, 栈顶为: CodeStr, 输入为:
return, 使用产生式: CodeStr -> Code
CodeStr
第872步, 栈顶为: Code, 输入为:
return, 使用产生式: Code -> ReturnCode
\$;
第873步, 栈顶为: ReturnCode, 输入为:
return, 使用产生式: ReturnCode ->
\$return ReturnCode1
第874步, 栈顶为: \$return, 输入为:
return, 出栈
第875步, 栈顶为: ReturnCode1, 输入
为: ;, 使用产生式: ReturnCode1 -> \$seps
第876步, 栈顶为: \$seps, 输入为: ;, 跳过
第877步, 栈顶为: \$;, 输入为: ;, 出栈
第878步, 栈顶为: CodeStr, 输入为: },
使用产生式: CodeStr -> \$seps
第879步, 栈顶为: \$seps, 输入为: }, 跳过
第880步, 栈顶为: \$}, 输入为: }, 出栈
第881步, 栈顶为: StateStr, 输入为: #,
使用产生式: StateStr -> \$seps
第882步, 栈顶为: \$seps, 输入为: #, 跳过
第883步, 栈顶为: #, 输入为: #, 分析成
功

使用的产生式

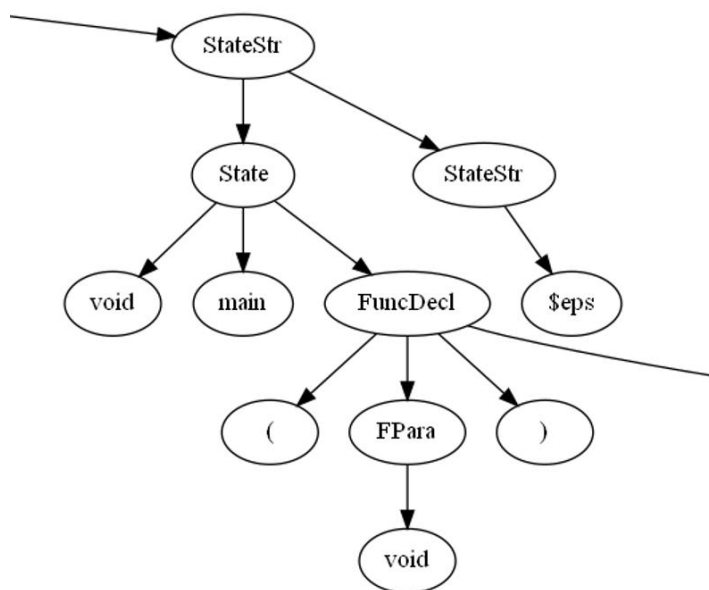
MulExpr1 -> \$seps
MulExpr -> Factor MulExpr1
AddExpr1 -> \$seps
AddExpr -> MulExpr AddExpr1
RelopExpr -> \$seps
Expr -> AddExpr RelopExpr
AssignCode -> \$id ArrayPart \$= Expr
Code -> AssignCode \$;
ReturnCode1 -> \$seps
ReturnCode -> \$return ReturnCode1
Code -> ReturnCode \$;
CodeStr -> \$seps
CodeStr -> Code CodeStr
CodeStr -> Code CodeStr
CodeStr -> Code CodeStr
CodeStr -> Code CodeStr
CodeStr -> Code CodeStr
CodeBlock -> \${ InState CodeStr \$}
FuncDecl -> \$(FPara \$) CodeBlock
State -> \$void \$id FuncDecl
StateStr -> \$seps
StateStr -> State StateStr
StateStr -> State StateStr
StateStr -> State StateStr
Program -> StateStr

展示语法树

由于语法树文件过大（2M 大小），这里只能缩小大小，具体可以自行生成并查看。



展示其一棵子树



展示语义分析结果

test.c 语法分析结果

生成的四元式

61 < + , %T22 , 0 , %T22 >
62 < * , %T22 , 4 , %T22 >
63 < load , a , %T22 , %T23 >
64 < mov , 0 , _ , %T24 >
65 < * , %T24 , 2 , %T24 >
66 < + , %T24 , 1 , %T24 >
67 < * , %T24 , 4 , %T24 >
68 < load , a , %T24 , %T25 >
69 < mov , 1 , _ , %T26 >
70 < * , %T26 , 2 , %T26 >
71 < + , %T26 , 0 , %T26 >
72 < * , %T26 , 4 , %T26 >
73 < load , a , %T26 , %T27 >
74 < push , %T27 , _ , _ >
75 < jal , _ , _ , demo >
76 < push , EAX , _ , _ >
77 < push , %T25 , _ , _ >
78 < push , %T23 , _ , _ >
79 < jal , _ , _ , program >
80 < mov , 1 , _ , %T28 >
81 < * , %T28 , 2 , %T28 >
82 < + , %T28 , 1 , %T28 >
83 < * , %T28 , 4 , %T28 >
84 < store , a , %T28 , EAX >
85 < jret , _ , _ , _ >

变量表

变量作用域:main
第几个变量:1
维数:2

变量名称:b
变量类型:int
变量作用域:main
第几个变量:5
维数:0

函数表

函数参数个数:1
函数长度:5
函数起始位置:22

函数名称:main
函数返回类型:\$void
函数参数个数:0
函数长度:59
函数起始位置:27

3.2 自定义程序运行结果

自行编写一份冒泡排序代码，并保存为 sort.c。

```
//#include<stdio.h>
```

```
//冒泡排序
int a[5];
int main() {
    int i;
    int j;
    int temp;

    a[0]=1;
    a[1]=2;
    a[2]=4;
    a[3]=5;
    a[4]=2;
    while(i<5) {
        j=i+1;
        while(j<5) {
            if(a[i]>a[j]) {
                temp=a[i];a[i]=a[j];a[j]=temp;
            }
            j=j+1;
        }

        i=i+1;
    }
    //printf("%d\n",a[4]);
    return 0;
}
```

打开之。



```
sort.c
文件 编译 分析 帮助
int a[5];
int main() {
    int i;
    int j;
    int temp;

    a[0]=1;
    a[1]=2;
    a[2]=4;
    a[3]=5;
    a[4]=2;
    while(i<5){
        j=i+1;
        while(j<5){
            if(a[i]>a[j]){
                temp=a[i];a[i]=a[j];a[j]=temp;
            }
            j=j+1;
        }

        i=i+1;
    }
    //printf("%d\n",a[4]);
    return 0;
}
```

编译信息

2022.01.30 10:58:56 打开文件D:/data/Qt/LinCC_NewPackage/testfile/sort.c

编译，得到目标代码 sort.asm。

编译信息

2022.01.30 10:58:56 打开文件D:/data/Qt/LinCC_NewPackage/testfile/sort.c
2022.01.30 10:59:22 编译成功，中间代码写入到同目录下sort_Quat.txt中
2022.01.30 10:59:22 编译成功，目标代码写入到同目录下sort.asm中

在 Mars 打开并汇编运行之。

EditExecute

sort.asm

```

1  li $sp, 0x1001ffff
2  addi $fp, $sp, 0
3  li $gp, 0x10000000
4  jal main
5  addi $fp, $fp, 4
6  li $v0, 10
7  syscall
8  main:
9  sw $ra, 0($fp)
10 addi $fp, $fp, -4
11 sw $sp, 0($fp)
12 move $sp, $fp
13 addi $fp, $fp, -4
14 subi $fp, $fp, 12
15 move $t0, $0
16 li $t9, 4
17 mult $t0, $t9
18 mflo $t0
19 li $t9, 1
20 add $v1, $gp, $t0
21 sw $t9, 0($v1)
22 li $t9, 1
23 move $t1, $t9
24 li $t9, 4
25 mult $t1, $t9
26 mflo $t1
27 li $t9, 2

```

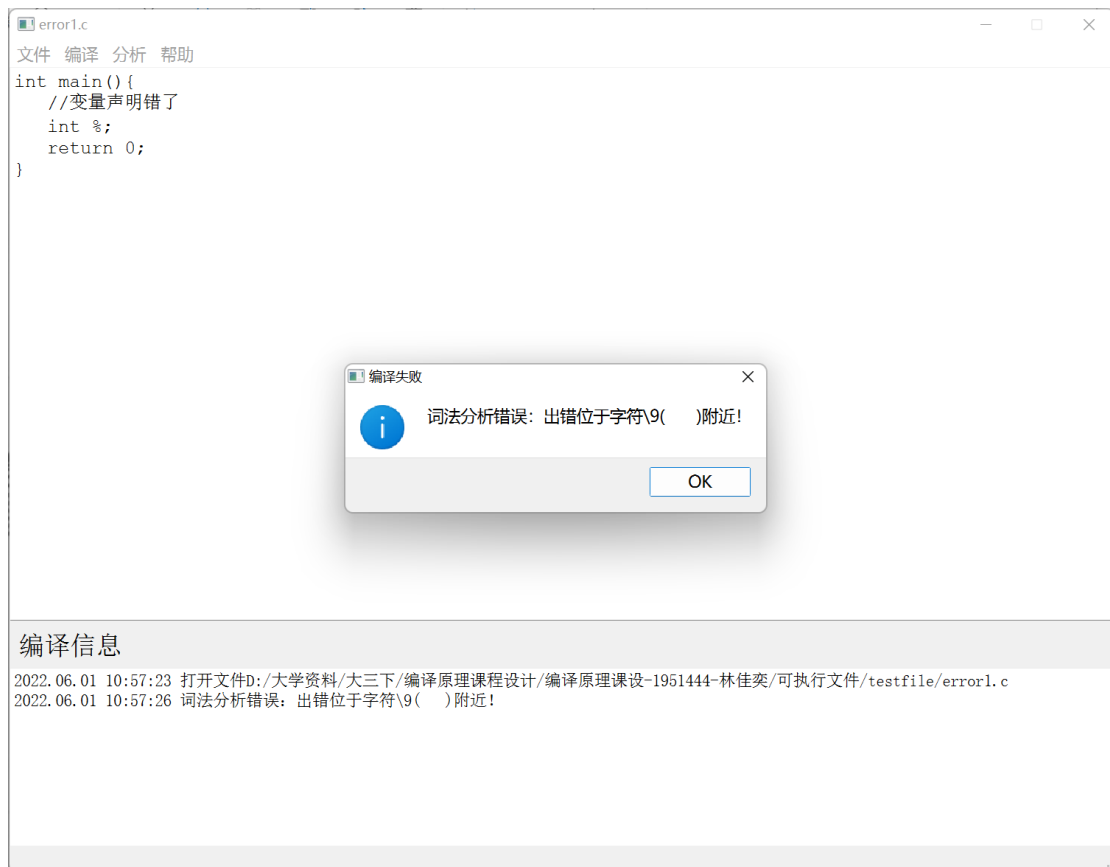
检查全局变量区（0x10000000）的内存单元

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) |
|------------|------------|------------|------------|------------|-------------|
| 0x10000000 | 1 | 2 | 2 | 4 | 5 |

发现已经按照升序排序，符合预期，这也说明编译成功且正确。
篇幅原因，不再展示其他功能。

3.3 报错信息展示

3.3.1 词法分析报错



如上图，变量声明出错，于是报错，并给出错误大致位置。

3.3.2 语法分析报错



如上图，在 b=1 后面少写一个分号，这是语法错误。

3.3.3 语义分析报错



如上图，函数 decl 未命名便调用，于是会报错。

四、 问题及解决方案

(1) 词法分析中，如何处理回车空格和制表符？

回车空格和制表符不参与单词的构成，因此需要跳过。但不能简单的指向下一个字符。如果当前状态为正在生成单词，需要进入终止态进行判别；如果处于初态，则继续维持在初态。

(2) 如何确保输入文法是 LL1 文法？

LL1 文法需要判别三个条件，分别是无左递归，产生式右部 FIRST 无交集，FIRST 与 FOLLOW 无交集（产生式右部为空时）。

第一个条件通过手动修改 LL1 文法即可保证（实际上，若存在左递归，在产生 FIRST 和 FOLLOW 的时候会陷入死循环），也可以用拓扑排序的方式保证。

后两个条件在求完 FIRST 和 FOLLOW 后程序验证。若不满足条件则修改产生式再次验证。

(3) 如何从语法树中获取产生式序列？

由于语法分析采用自上而下的方法，而语义分析要求自下而上，因此

要遍历语法树获得自下而上的顺序。这一点通过深度优先搜索（DFS）即可实现。

（4）如何处理分支、跳转语句的翻译和跳转地址的确定？

分支、跳转语句需要加入的四元式是确定的，但跳转地址不确定。因此需要记录翻译产生式时生成的首个四元式地址和四元式个数。由于非终结符有很多，但出现的时间不一样（在语法树的位置不一样），因此需要单独为其编号，并记录四元式首地址和个数。这样才可以确定跳转地址。

跳转地址采用相对地址的形式，原因在于若在此四元式前面再插入四元式，地址会发生变化，若采用绝对地址则会失效，而采用相对地址则不会失效。

在目标代码生成的时候，由于一个四元式可能对应多条汇编代码，因此需要标记每一个四元式产生的首个汇编代码行数，这样方便生成标签以实现跳转。

（5）如何区分局部变量和全局变量？

虽然本实验不允许局部变量和全局变量重名，但仍需要加以区分。在变量表中的“作用域”一行加以区分。全局变量的作用域为\$public，其余标记为函数名。

（6）如何将语法树以图形的方式输出？

Qt 自带的框架不利于展示语法树，因此考虑图片输出。采用了 dot 文件编译的方式输出，并打开。以上都是采用程序内调用命令行的方式实现。

（7）Qt 中文信息乱码如何处理

对于字符串常量，使用 Qt 宏 QStringLiteral 进行转换。若是 std::string 变量，则先定义编码转换类 QTextCodec，设置其编码为 GB2312，在调用其函数 toUnicode 进行转换。

五、 心得体会

通过一个学期的学习，并实现一个简单的 C 语言编译器，这其实不是一个简单的工作，涉及到很多复杂的算法，并且要时刻留心可能的陷阱。

因此在完成编译原理课程设计的时候，我按照进度一点点完成。由于大三上学期对于编译原理的理解不够深入，完成的作业代码利用率较低，加上课程设计有所提升，我选择在寒假重写词法分析，语法分析和语义分析代码，并生成 Mips 汇编代码。所幸经过两周的努力我完成了全部工作。

一直以来我以为编译原理就是处理字符串，其实不然。语义分析，目标代码生成部分，需要很多的计算机系统知识，例如函数调用的处理，寻址的处理等。此外还需要对指令集架构有所了解，要明确寄存器是用来干什么的，不能单纯的随便使用寄存器，否则很有可能生成了错误的汇编代码。

Mips 汇编是 RISC 指令集，在计算机组成原理、计算机系统结构等课程中我们实现了基于 Verilog 的 MipsCPU 设计，对指令有一定的了解，并且有相应的汇

编模拟器 Mars，可以验证汇编代码的正确性。因此我没有采用 x86 汇编而是采用了 Mips 汇编，因此在目标代码生成器部分我参考但并没有照搬书上的方法。

尽管如此，在实现函数调用这块我还是遇到了很大的难题，代码一直生成有误，原因是没有明确栈帧的格式。于是我参考了操作系统课件，了解了栈帧的格式，并根据 Mips 指令集架构设计了相应的栈帧格式，成功实现代码。

鉴于时间有限，我无法完全复现标准 C 语言的编译器，也没有生成可执行文件，这些内容或许是我在以后的学习生活中可以进一步研究的内容。但我按照要求完成了相应内容，我认为本次课设收获丰富，也让我进一步巩固了编译原理的相应知识。

六、 附录

- [1]. En Takahashi. MIPS 指令集及汇编完全解析[EB/OL].
https://blog.csdn.net/qq_41191281/article/details/85933985
- [2]. 陈火旺、钱家骅、孙永强.程序设计语言编译原理（第三版）[M].北京：国防工业出版社，2004-10
- [3]. 张冬冬、王力生、郭玉臣.数字逻辑与组成原理实践教材[M].北京：清华大学出版社，2018-8