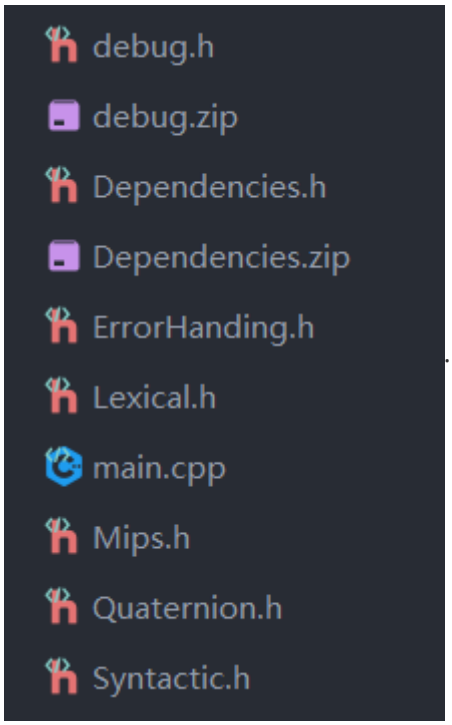
参考编译器介绍

主要参考了教科书给出的PL/0简单编译系统 与 Pascal-S编译系统

并参考了mips架构对编译器的寄存器分配等思路

编译器总体设计

本编译器目录结构如下



先介绍各文件作用

Dependencies.h：底层依赖文件，在其中定义了全部数据结构、常用的无关结构的打印函数、导入了文件依赖的c++库文件、定义了部分需要实现的函数、并用注释方式对其行为及逻辑进行了规定。

其定义的数据包括：词法分析表、词法分析结构表、语法分析结果表、符号表、运行栈、模拟寄存器数组、保留词表、各级别的输出开关

其提供的方法：仅包括针对各数据类型提供的读写方法

Debug.h：Debug文件，仅依赖于Dependencies.h文件，提供对各个关键列表的读取、检查操作，是用于debug的程序文件，舍去不影响程序正常运行，其提供的方法包括： 查看词法分析结果表、查看语法分析结果表、查看符号表、查看运行栈、查看寄存器分配情况、查看程序分析进度等

其可在任意文件中进行调用，以进行程序的调试。

ErrorHanding.h：仅依赖于Dependencies.h文件，提供了错误处理的输出部分及错误处理的判断逻辑函数，并提供了符号表的构建函数

Lexical.h： 词法分析程序，依赖于Dependencies.h，对外表现为仅进行词法分析

Syntactic.h: 语法分析程序，依赖于Lexical.h、Quaternion.h、Dependencies.h、ErrorHanding.h

调用Lexical.h提供的分析结果，并调用Quaternion.h中的函数完成四元式生成、调用ErrorHanding.h中的函数完成错误处理及符号表生成、调用Dependencies.h中的对应数据结构存储其输出

Quaternion.h 生成四元式，依赖于Mips.h，在生成四元式的过程中调用Mips.h中的函数同步生成汇编代码

Mips.h 依赖于Dependencies.h 提供生成汇编代码的函数接口

本编译器实质上仅进行了一次扫描，因此未能做到窥孔优化、活跃度寄存器分配、寄存器共享等优化

但本编译器实现了大幅度的常量传播，以及对寄存器的回收机制，并对寄存器进行了一定程度的重新分配，因此总体性能不差

词法分析

编码前设计：

词法分析本质是一个自动机，但是需要额外记录行号，

编码时设计

依赖编码前设计，需要额外注意的细节在于对于'/r/n'的处理和对于不合法字符的判断

具体实现过程中词法分析的自动机实质上是多个自动机的组合，确定好状态就能较为顺利地完成

还需要考虑注释的问题，以及在注释中产生的换行

语法分析

编码前设计：

语法分析编码设计为使用递归下降分析，并将对应输出使用全局变量控制

需要注意的是原文法存在直接左递归的情况，需要进行改写

```
        | UnaryOp UnaryExp // 存在即可
单目运算符   UnaryOp → '+' | '-' | '!' 注：'!'仅出现在条件表达式中 // 三种均需覆盖
函数实参表   FuncRParams → Exp { ',' Exp } // 1.花括号内重复0次 2.花括号内重复多次 3.
Exp需要覆盖数组传参和部分数组传参
乘除模表达式  MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp // 1.UnaryExp
2.* 3./ 4.% 均需覆盖
加减表达式   AddExp → MulExp | AddExp ('+' | '-') MulExp // 1.MulExp 2.+ 需覆盖 3.-
需覆盖
关系表达式   RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp // 1.AddExp
2.< 3.> 4.<= 5.>= 均需覆盖
相等性表达式  EqExp → RelExp | EqExp ('==' | '!=') RelExp // 1.RelExp 2.== 3.!= 均
需覆盖
逻辑与表达式  LAndExp → EqExp | LAndExp '&&' EqExp // 1.EqExp 2.&& 均需覆盖
逻辑或表达式  LOrExp → LAndExp | LOrExp '||' LAndExp // 1.LAndExp 2.|| 均需覆盖
```

对于该部分及类似部分的改写思路是，进行带入展开

例如 RelExp->AddExp|RelExp (< | > | <= | >=) AddExp

实际上是RelExp->AddExp{<|>|<=|>= AddExp} 但是需要注意的是，进行此种展开之后，为与原语义保持一致，我们实际上是左优先的，这一点在输出和后续的进一步分析中都十分重要，也就是说，我们每遇到一个AddExp，在其结束时实际上就要立即将其规约为RelExp，并做对应输出。

编码时设计：

编码时遇到的一大问题就是条件判断，因为没有具体看后续错误处理的要求，导致在编写的时候多处存在不必要的if，这使得后续在语法分析基础上做的进一步分析显得十分凌乱。

编码时增加了调试函数用于确认当前程序运行进度，总体而言语法分析的设计在使用递归下降法之后较为简单，出现的问题多在后续对其进行扩充修改时。

错误处理

编码前设计：

对于括号缺失等错误，需要考虑括号的Follow集进行处理，而针对循环break、continue，则需要额外维护一个标志括号层级的变量

对于符号表，设计为边分析边填，具体数据结构如下

```
struct identTable {
    string name = "error!";
    int ident_kind = 0; // 0 is a / 1 is a[] / 2 is a[][] / -1 is bool
    int dimension_one = 0;
    int dimension_two = 0;
    int value = 0;
    bool init = false;
    bool changeable = true;
    bool isGlobal = false;
    vector<int> arrayValue;
    string reg = "NULL!";
    int fg_offset = -1;
} ErrorIdent;

struct funcTable {
    string name = "error!";
    int return_kind = 0;    // 0 is int / 1 is void
    int params_num = 0;
    vector<identTable> params_list; // in fact, wo only need keep the ident_kind is true
    bool effective = true;
} ErrorFunc;

vector<funcTable> FuncTable;    // the vector of func
vector<identTable> ConstIdentTable; // the vector of global var
vector<identTable> UsualIdentTable; // the vector of global const
vector<vector<identTable>> allSymbolTable;  //the table of func
```

采用函数与变量分开存储的设计方法，对于变量，将普通变量与一维二维统一存储，并使用reg标志为其分配的寄存器，或使用fg_offset标志为其分配的栈空间距离fp的偏移，从而实现本地变量列表到mips体系结构的映射

并设计了插入函数，在该过程中进行符号表的检查与去重

编码时修改：

符号表实现的较为顺利，遇到的主要问题在于对参数类型的检查。

起初尝试通过不修改语法分析文件，仅传参的方式完成匹配检查，但在尝试中发现这样做不仅费时费力，而且不利于后续功能的集成，因此修改了语法分析程序，主要修改了其返回值，并新增了全局变量用于处理实参列表，这一点在设计的时候没有花太多时间考虑，导致最终花费了大量时间去进行修改与验证。

编码完成后修改：

在代码生成二中注意到实际上对于符号表的检查有误，导致出现多处错误引用的情况，进行了修复

中间代码生成

中间代码生成采用了自行设计的四元式，

```
int my
void decl()
const int a
const int a1[2]
int x
x = 2
int x1[2]
int repite()
para a1
print str_1
ret a1
int repite2()
para a1
para a2
print str_2
$t0 = a1 * a2
```

主要针对编译过程中mips代码的生成进行拆解

在这一步完成了常量传播的处理

处理方法如下：

设计了程序本地运行栈，用于对实际运行情况进行模拟

```
struct st{
    int kind = 0;   // 0 is value 1 is string
    int value = 0;  // 常量值
    string name = "error!";
//    bool key = true;
    int offset = -1;    //数组变量下标与起点的差值
    string reg = "NULL!";   //变量寄存器
    int fg_offset = -1; //变量与fg的差值
    bool melt = false;  //变量是否在使用一次后即可释放其对应寄存器
    bool isAddress = false; // 变量内容是否为地址，是针对于数组元素特化的
} ErrorSt;
```

采用栈对程序运行进行模拟，一旦遇到认为是常量的变量或常量，就对其进行值传播，即不进行寄存器分配、复制等操作，直接将其认为是常数带入运算，对于如下示例程序

```
const int a = 5;
int main() {
    int c = 5 * a + 1;
    return 0;
}
```

其效果为

```
const int a = 5
int main()
int c
c = 26
ret 0
```

省去了大量的运算时间

而对于寄存器的分配，采用了如下的分配方式

```
reg retReg[2]; // v0 v1
reg paramReg[4]; // a0 a1 a2 a3

reg varReg[10]; // t6 - t7 s0 - s7 是可分配给变量的临时寄存器
// t0 到 t5用作运算临时寄存器
// a0 到 a3实际上未使用，是很大的浪费，主要原因在于实际实现过程与最初设想的实现方式不同
//v0与v1用做返回值的临时寄存器
```

目标代码生成

选用的目标代码为mips

需要解决的问题有：对于寄存器的分配、对于变量的存储、对于数组对象的处理、参数传递、cond的处理

对于寄存器的分配策略具体为：

对于每块独立代码（以{}为分界线，一旦退出}，则释放其中分配的全部寄存器，循环除外）

若存在可分配的临时寄存器，则在定义时直接分配

若不存在，则在栈上为其开辟对应的空间

数组全部存放在栈中

对于全部的运算结果，都暂存于运算寄存器，由于运算的结构为递归下降分析，实际上同时最多只用到四个寄存器，保留一个寄存器用于防止出现未考虑到的意外情况

对于全部需要暂存的变量，如printf的%d，需要全部计算完成之后再进行输出，采用与变量相同的寄存器分配政策

对于参数，计算出其值之后暂存，全部计算完毕后直接按顺序压入栈中，并释放其占据的全部寄存器

对于数组对象：

在Lval这一级，仅计算其地址，存入寄存器中，在后续根据其需求，选择取该地址的值还是将值上传至该地址

而在传参过程中，计算维数与实际维数不匹配即可确定为参数，仅需将计算出的地址作为参数传入即可

而计算出的地址对于使用方与首地址的效果相同

对于Cond的处理：

采用了上下合并的方式

在||与&&中采用了短路求值的方式

实现思路为将 if_ok if_no作为参数传入，即条件为真应当跳转至何处，条件为假应当跳转至何处

而对于 == != 及其下次运算，使用snq系列mips指令，将结果存入临时运算寄存器中

从而获得了较为高效的跳转逻辑

代码优化

代码优化方面仅做了寄存器分配的处理和常量传播，在上文已经介绍完毕