本项目的所实现的语言是静态类型的面向过程语言。

- 支持三种数据类型：int、int*(intpointer)、void
- 支持数组声明（分配一块数组内存和一个指向该数组的指针）和下标访问，能使用字符串给数组赋值。
- 支持函数声明，函数传参，函数调用
- 支持以下类型的运算（优先级从低到高）：

```
int,int*(int pointer),void
exp,exp
int=int,(*int)=(*int),(*int)=(STRINGCONSTANT),int=(INTCONSTANT)
int?int:int,int?(int*):(int*),(int?void:void)
int || int
int && int
int | int
int ^ int
int & int
int == int,int != int
int < int,int <= int,int > int,int >= int
int << int,int >> int
int + int,int - int,(*int)+int,int+(int*)
int * int,int / int,int % int
!int
-int
*(int*)
function,constant,varible
```

- 支持 continue、break、return、while、if、else、putchar、read、output语句，具体语法与c语言相似
- 实现了包括类型检查在内的绝大部分可能出现的报错
- 链接器的实现尚未完成，但是只需要对"FakeLinker.cpp"稍作修改即可完成

内存分配：

- 0-199为寄存器和临时变量：

  RIP 0

  RBQ 1

  RSP 2 //RSP points to the last data in the stack

RAX 3

T0 4

T1 5

T2 6

CON0 101 // Constant 0

CON1 102 // Constant 1

CON 100 // Temp Constant

- 200开始，全局变量向上开始分配内存
- 50000开始，局部变量向下开始分配内存
- 当前函数局部变量和临时变量的存放为为$RBQ~$RSP。其中$RBQ内存放着当前函数return的位置，$RBQ+1的内存存放着上一个stack block对应的$RBQ的值

以下是部分代码实例：

```
int gcd(int a,int b){
    if(b == 0) return a;
    else return gcd(b,a%b);
}

int main(){
    int a;
    int b;
    read &a;
    read &b;
    print gcd(a,b);
    return 0;
}
```

```
void printf(int *a){
    while(*a){
        if(*a == 111)
            break;
        putchar *a;
        a = a+1;
    }
}

int a[20];
int main(){
    a = "Hello World\n";
    printf(a);
    return 0;
}
//其中*a == 111是用来表示 *a == 'o'
```

关于虚拟机：

- 由于部分操作（putchar、位运算）不便在原虚拟机上执行，所以添加了部分指令
- 具体指令如下：

OP	x	y	z	注释
0	i		addr	常量存储，语义：MEM[addr] = i
1	x	B	idx	数组存储运算符，语义：MEM[B+MEM[idx]] = MEM[x]
2	B	idx	x	数组元素读取，语义：MEM[x] = MEM[B+MEM[idx]]
3	x		y	拷贝指令，语义：MEM[y] = MEM[x]
4,5,⋯,13 15,,,19	x	y	z	二目运算符指令，语义：MEM[z] = MEM[x] op MEM[y] 运算和 op 值的对应关系：4 +，5 -， 6 *，7 /，8 %， 9 ==，10 >，11 <，12 &&，13 ||。15 &,16 |,17 ^,18<<,19>> 其中，等于、大于、小于判断时，如果关系成立，结果为 1；否则为 0； 与、或、非运算的结果也是 0（FALSE）或者 1（TRUE）
14	x		z	MEM[z] = !MEM[x]
20	x		B	如果 MEM[x]的值非零，跳转到第 B 条指令继续执行。
30			B	无条件直接跳转到第 B 条指令继续执行。
40			x	无条件跳转到第 MEM[x]条指令继续执行。
50	x			输出 MEM[x]
60	x			读取一个 int 存放到 MEM[x]
70	x			输出一个字符，其 ascii 码为 MEM[x]
100				退出

- 上述两段代码对应的指令码如下：

```
111
0 50000 2
0 50000 1
0 0 101
0 1 102
0 3 100
4 0 100 4
1 4 0 2
30 63
100
5 2 102 2
2 -2 1 4
1 4 0 2
5 2 102 2
0 0 100
1 100 0 2
2 0 2 5
4 2 102 2
2 0 2 4
9 4 5 4
1 4 0 2
2 0 2 4
4 2 102 2
20 4 54
5 2 102 2
1 1 0 2
5 2 102 2
5 2 102 2
2 -2 1 4
1 4 0 2
5 2 102 2
2 -1 1 4
1 4 0 2
5 2 102 2
2 -2 1 4
1 4 0 2
2 0 2 5
4 2 102 2
2 0 2 4
8 4 5 4
1 4 0 2
0 2 100
4 100 2 1
3 0 5
0 5 100
4 100 5 5
1 5 0 1
30 9
4 1 102 2
2 0 2 1
```

```
1 3 0 2
2 0 2 3
2 0 1 4
40 4
30 60
5 2 102 2
2 -1 1 4
1 4 0 2
2 0 2 3
2 0 1 4
40 4
0 0 100
2 0 1 4
40 4
5 2 102 2
5 2 102 2
5 2 102 2
0 1 100
5 1 100 4
1 4 0 2
60 4
2 0 2 5
1 4 0 5
4 2 102 2
5 2 102 2
0 2 100
5 1 100 4
1 4 0 2
60 4
2 0 2 5
1 4 0 5
4 2 102 2
5 2 102 2
1 1 0 2
5 2 102 2
5 2 102 2
2 -1 1 4
1 4 0 2
5 2 102 2
2 -2 1 4
1 4 0 2
0 2 100
4 100 2 1
3 0 5
0 5 100
4 100 5 5
1 5 0 1
30 9
4 1 102 2
2 0 2 1
1 3 0 2
```

```
2 0 2 4
50 4
4 2 102 2
5 2 102 2
0 0 100
1 100 0 2
2 0 2 3
2 0 1 4
40 4
2 0 1 4
40 4
```

```
153
0 200 100
0 220 4
1 100 0 4
0 50000 2
0 50000 1
0 0 101
0 1 102
0 3 100
4 0 100 4
1 4 0 2
30 75
100
30 63
5 2 102 2
2 -1 1 4
1 4 0 2
2 0 2 4
2 0 4 4
1 4 0 2
5 2 102 2
0 111 100
1 100 0 2
2 0 2 5
4 2 102 2
2 0 2 4
9 4 5 4
1 4 0 2
2 0 2 4
4 2 102 2
20 4 31
30 32
30 72
0 0 100
5 2 102 2
2 -1 1 4
1 4 0 2
2 0 2 4
2 0 4 4
1 4 0 2
2 0 2 4
70 4
4 2 102 2
5 2 102 2
2 -1 1 4
1 4 0 2
5 2 102 2
0 1 100
1 100 0 2
2 0 2 5
```

```
4 2 102 2
2 0 2 4
4 4 5 4
1 4 0 2
5 2 102 2
0 1 100
5 1 100 4
1 4 0 2
2 0 2 4
4 2 102 2
2 0 2 5
1 5 0 4
1 5 0 2
4 2 102 2
5 2 102 2
2 -1 1 4
1 4 0 2
2 0 2 4
2 0 4 4
1 4 0 2
2 0 2 4
4 2 102 2
20 4 13
0 0 100
2 0 1 4
40 4
0 13 100
5 2 100 2
3 2 4
0 72 100
1 100 0 4
4 4 102 4
0 101 100
1 100 0 4
4 4 102 4
0 108 100
1 100 0 4
4 4 102 4
0 108 100
1 100 0 4
4 4 102 4
0 111 100
1 100 0 4
4 4 102 4
0 32 100
1 100 0 4
4 4 102 4
0 87 100
1 100 0 4
4 4 102 4
0 111 100
```

```
1 100 0 4
4 4 102 4
0 114 100
1 100 0 4
4 4 102 4
0 108 100
1 100 0 4
4 4 102 4
0 100 100
1 100 0 4
4 4 102 4
0 10 100
1 100 0 4
4 4 102 4
1 101 0 4
5 2 102 2
0 220 100
1 100 0 2
2 0 2 4
4 2 102 2
2 0 4 4
2 0 2 5
30 127
1 5 0 4
4 2 102 2
4 4 102 4
2 0 2 5
20 5 123
4 2 102 2
5 2 102 2
1 1 0 2
5 2 102 2
5 2 102 2
1 220 0 2
0 1 100
4 100 2 1
3 0 5
0 5 100
4 100 5 5
1 5 0 1
30 12
4 1 102 2
2 0 2 1
1 3 0 2
4 2 102 2
5 2 102 2
0 0 100
1 100 0 2
2 0 2 3
2 0 1 4
40 4
```

```
2 0 1 4
40 4
```

实现思路：

- 总体上，遵循Lexer->Parser->Semantic analyzer->Code generator的实现思路，其中Semantic analyzer被嵌入在Code generator的代码中
- 各个文件的实现内容如下：

  Shared：公共头文件

  Lexer：完成句法单位(lexeme)的提取

  Parser：完成语法树结构的建立，采用作业文档内描述的算法

  ASTNode：语法树节点的父类

  Instruction：低级代码指令类，注意本项目中指令是以链表的结构储存的

  Statement：语句的所有类型声明及定义

  Expression：表达式的所有类型声明及定义

  CodeGenerator：完成低级代码的生成及句法分析，采用stack machine的执行模式

  FakeLinker：完成了类链接工作