# Project Phase 3 Report

| SID | Name | Rate |
| --- | --- | --- |
| 11911202 | 袁恒宸 | 1/3 |
| 12011543 | 林洁芳 | 1/3 |
| 12011906 | 汤奕飞 | 1/3 |

## Basic

In the intermediate representation generation section, we use tac class to generate IR code, including v, t variable and label. Based on semantic analysis, we use tac to connect them in series.

### Test

We can pass all the tests.

## Optimization

In the process of implementation, the following aspects were optimized.

1. Numbers can be used directly as a variables.
2. Subtracting two identical numbers directly assigns the result to 0.
3. Any number plus or minus zero is itself.
4. 1 multiplied by any number and any number divided by 1 is itself.
5. Self-increment and self-decrement do not use additional temporary variables.
6. *The address of the first element of an array or the first variable of a structure is the starting address of the array or structure.

Several of the following attempted optimization strategies are incorrect.

1. A number multiplied by a number and divided by the same number does not necessarily equal itself.
2. Since it is not possible to ensure whether a variable has been modified again, the same operation on the same variable cannot use the previously computed result.

What is listed above is roughly the entirety of the optimization.

As for tac.h:

```
#ifndef _TAC
#define _TAC

typedef struct tac
{
```

```
    enum { FUNC = 0, READ, WRITE, LABEL, GOTO, IF, ASS, OPER, RETURN,
PARAM, ARG, DEC} title;
    char* target;
    char* op;
    char* arg1;
    char* arg2;
    struct tac* next;
} Tac;

Tac * newTac(char* target, char* op, char* arg1, char* arg2);

void printTacs(Tac* head);

char* generateV(int v);

char* generateT(int t);

char* generateLabel(int lbl);

#endif
```

# Bonus

We can pass all bonus part tests in GitHub. Since these tests are complicate enough and some of our group members are sick, we write relatively simple extra tests which can cover all bonus parts we writed.

## Multi-dimensional array

We can handle one-dimensional and even multi-dimensional nested arrays, including INT arrays as well as custom structure arrays.

In test:

```
struct Apple
{
    int a;
    int number[10][5][7][10];
};

int add(int a, int b){
    return a + b;
}

struct Apple test(int c, int d)
{
    struct Apple aa;
    aa.a = 15;
    aa.number[0][2][3][4] = add(c, d);
    return aa;
}
```

```
FUNC add :
PARAM v0
PARAM v1
t0 := v0 + v1
RETURN t0

FUNC test :
PARAM v2
PARAM v3
DEC v4 14004
t1 := &v4
*t1 := #15
t3 := &v4 + #4
t4 := #2 * #280
t5 := t3 + t4
t6 := #3 * #40
t7 := t5 + t6
t8 := #4 * #4
t9 := t7 + t8
ARG v3
ARG v2
*t9 := CALL add
RETURN &v4
```

## Structure

We can handle structure. Structure variables can appear in the program, and they can be declared as function parameters. The internal member variables of a structure can be either arrays or basic data types, in this case INT.

In test:

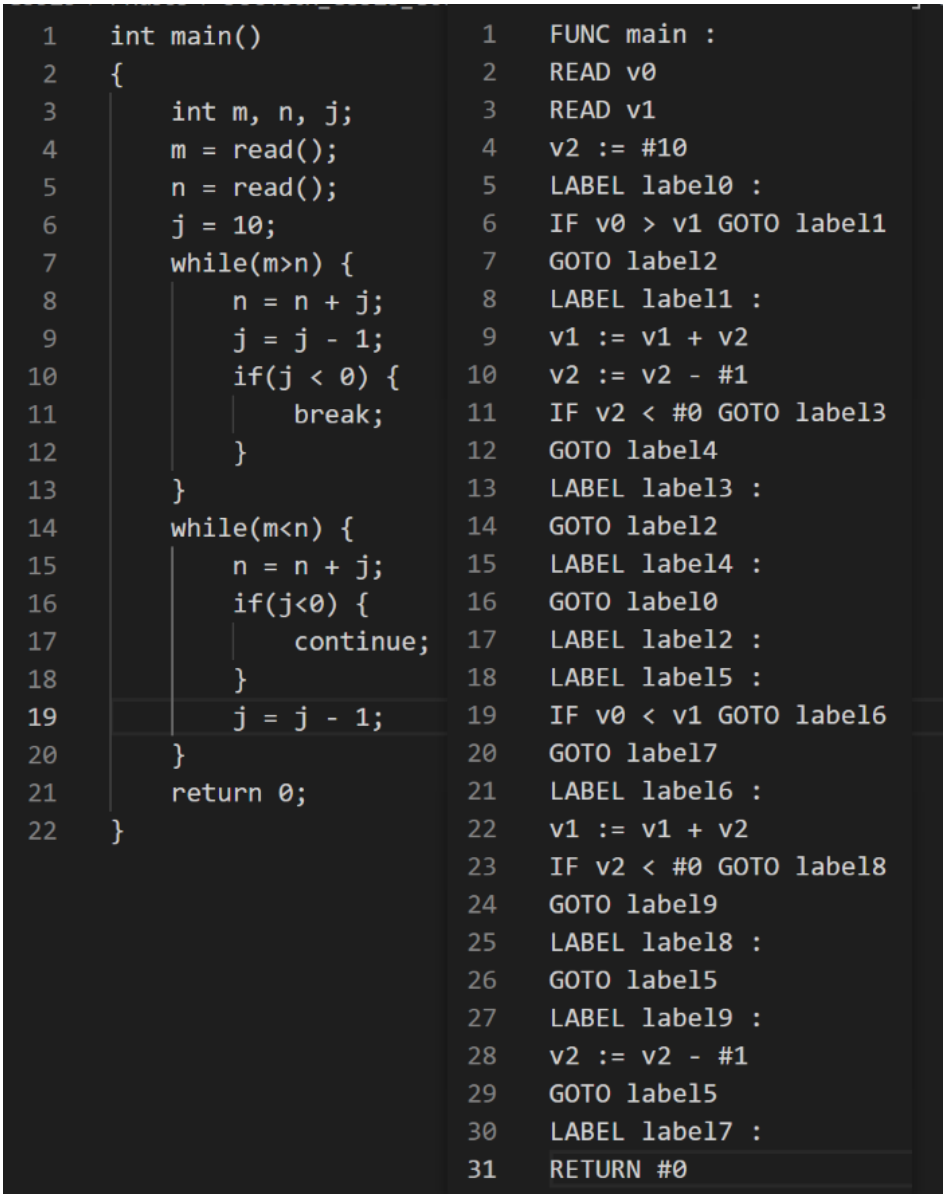
```
struct Student
{
    int ID;
    int score;
};

int main()
{
    struct Student s1, s2;
    s1.ID = 1;
    s1.score = 70;
    s2.ID = 2;
    s2.score = 90;
    write(s2.ID);
    write(s1.score);
    return 0;
}
```

```
FUNC main :
DEC v0 8
DEC v1 8
t0 := &v0
*t0 := #1
t2 := &v0 + #4
*t2 := #70
t4 := &v1
*t4 := #2
t6 := &v1 + #4
*t6 := #90
t8 := &v1
t9 := *t8
WRITE t9
t10 := &v0 + #4
t11 := *t10
WRITE t11
RETURN #0
```

## Continue & Break

We can handle continue and break instructions, which can jump correctly when encountered in the while loop.

In test:



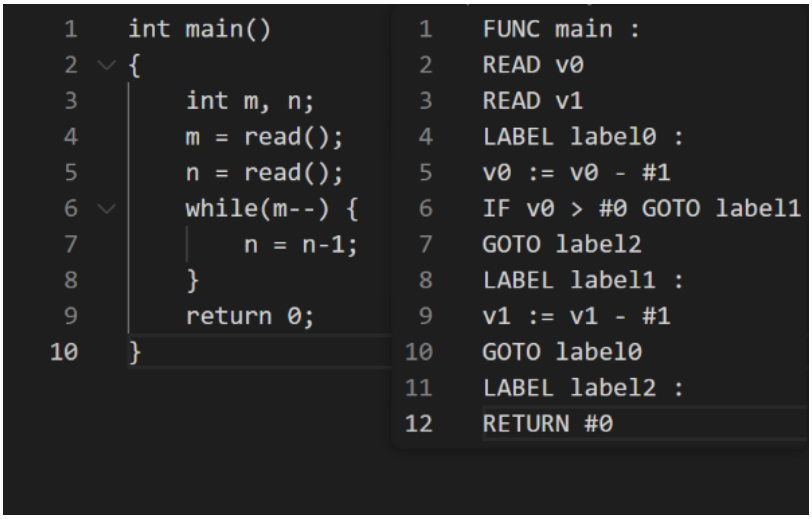
```
int main()
{
    int m, n, j;
    m = read();
    n = read();
    j = 10;
    while(m>n) {
        n = n + j;
        j = j - 1;
        if(j < 0) {
            break;
        }
    }
    while(m<n) {
        n = n + j;
        if(j<0) {
            continue;
        }
        j = j - 1;
    }
    return 0;
}
```

```
FUNC main :
READ v0
READ v1
v2 := #10
LABEL label0 :
IF v0 > v1 GOTO label1
GOTO label2
LABEL label1 :
v1 := v1 + v2
v2 := v2 - #1
IF v2 < #0 GOTO label3
GOTO label4
LABEL label3 :
GOTO label2
LABEL label4 :
GOTO label0
LABEL label2 :
LABEL label5 :
IF v0 < v1 GOTO label6
GOTO label7
LABEL label6 :
v1 := v1 + v2
IF v2 < #0 GOTO label8
GOTO label9
LABEL label8 :
GOTO label5
LABEL label9 :
v2 := v2 - #1
GOTO label5
LABEL label7 :
RETURN #0
```

## while(T--/--T)

We can handle while(T--){} block, where the loop body will execute T times for any non-negative integer T, and the loop terminates when T reaches zero.

In test:



```
int main()
{
    int m, n;
    m = read();
    n = read();
    while(m--) {
        n = n-1;
    }
    return 0;
}
```

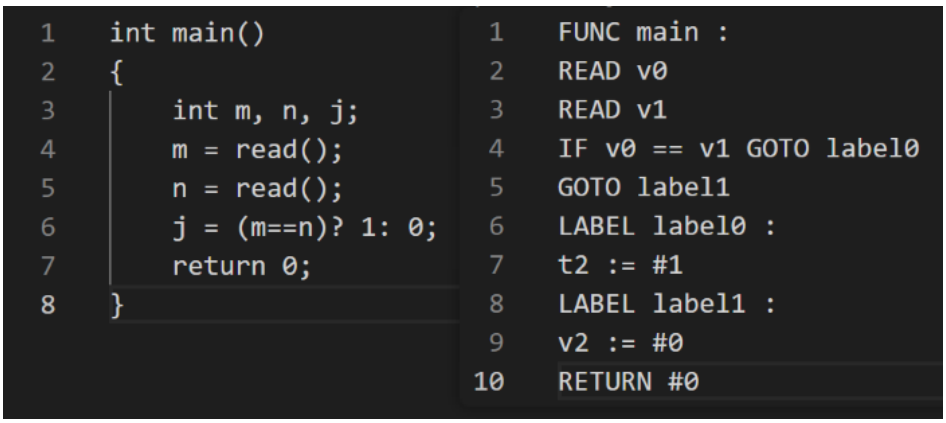
```
FUNC main :
READ v0
READ v1
LABEL label0 :
v0 := v0 - #1
IF v0 > #0 GOTO label1
GOTO label2
LABEL label1 :
v1 := v1 - #1
GOTO label0
LABEL label2 :
RETURN #0
```

## Ternary operator(... ? ... : ...)

In translation.c:

```
else if(!strcmp(NDtypes[operator->type],"QM")){
    //Exp QM Exp COLON Exp
    ...
}
```

In test:



```
int main()
{
    int m, n, j;
    m = read();
    n = read();
    j = (m==n)? 1: 0;
    return 0;
}
```

```
FUNC main :
READ v0
READ v1
IF v0 == v1 GOTO label0
GOTO label1
LABEL label0 :
t2 := #1
LABEL label1 :
v2 := #0
RETURN #0
```