11712639-Project3-Report

Basic Requirement

1. Basic Implements

- Use quadruples to store TAC, and doubly-linked list to represent IR data structure. As following:

```
typedef struct IRInstruction
{
    int index;
    char* target;
    enum {_LABEL, _FUNCTION, _ASSIGN, _PLUS, _MINUS, _MUL, _DIV, _ADDR, _VAL, _GOTO,
_LT, _LE, _GT, _GE, _EQ, _NE, _RETURN, _DEC, _PARAM, _ARG, _CALL, _READ, _WRITE,
_EMPTY} op;
    char* arg1;
    char* arg2;
    struct IRInstruction *next;
    struct IRInstruction *previous;
} IRInstruction;
```

- Implement translate_X functions according to the translation schemes in project3 file and traverse the syntax tree built by previous project to build the instruction list.

2. Optimizations

> local optimization

First, divide instructions into basic blocks according to the lecture knowledge:

Partitioning Three-Address Instructions into Basic Blocks

- **Input:** A sequence of three-address instructions
- **Output:** A list of basic blocks (each inst. is assigned to one block)
- **Method:**
  - Rules for finding *leader instructions* (首指令, the 1st instruction of a basic block)
    1. The first instruction in the entire intermediate code is a leader
    2. Any instruction that is the target of a conditional/unconditional jump is a leader
    3. Any instruction that immediately follows a conditional/unconditional jump is a leader
  - Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program

Second, optimizations in block

- constant calculation
    - calculate the result directly and assign it to target if both arg1 and arg2 are immediate numbers. For example, change "v := #2 + #3" to "v := #5".
    - simplify the instruction when there are some senseless calculations. For example, simplify "v := t * #1" to "v := t".

```
//解决常数计算
if ((ptr->op == _PLUS || ptr->op == _MINUS || ptr->op == _MUL || ptr->op == _DIV)
    && ptr->arg1[0] == '#' && ptr->arg2[0] == '#')
{
    caculate(ptr);
    ptr->op = _ASSIGN;
    ptr->arg2 = NULL;
}else if (ptr->op == _MINUS && strcmp(ptr->arg1, ptr->arg2) == 0)
{
    ptr->op = _ASSIGN;
    ptr->arg1 = "#0";
    ptr->arg2 = NULL;
}else if (ptr->op == _MUL && (strcmp(ptr->arg1, "#1") == 0 || strcmp(ptr->arg2, "#1") == 0))
{
    ptr->op = _ASSIGN;
    ptr->arg1 = strcmp(ptr->arg1, "#1") == 0 ? ptr->arg2 : ptr->arg1;
    ptr->arg2 = NULL;
}else if (ptr->op == _DIV && strcmp(ptr->arg2, "#1") == 0)
{
    ptr->op = _ASSIGN;
    ptr->arg2 = NULL;
}
```

- copy propagation

  If the value of some variables has not been changed and are assigned to others, the original value can be referred to directly.

  I use traverse to refer some arg1 or arg2 to original value.

```
//解决复写传播
if (ptr->op == _ASSIGN && (ptr->target[0] == 't'
                           || (ptr->target[0] == 'v' && ptr->arg1[0] != 't')))//如果是temp = value的情况不做替换
{
    char *early_def = ptr->arg1;
    char *curr_var = ptr->target;
    IRInstruction *ptr1 = ptr->next;
    while (1)
    {
        //复写传播的范围(从赋值开始到target被赋值或arg1被赋值或block结束)
        if((ptr1->target != NULL && (strcmp(ptr1->target, curr_var) == 0 || strcmp(ptr1->target, early_def) == 0))
           || ptr1->index >= end->index){
            break;
        }else if (ptr1->arg1 != NULL && strcmp(ptr1->arg1, curr_var) == 0)
        {
            ptr1->arg1 = early_def;
```

```
        }else if (ptr1->arg2 != NULL && strcmp(ptr1->arg2, curr_var) == 0)
        {
            ptr1->arg2 = early_def;
        }
        ptr1 = ptr1->next;
    }
}
```

> global optimization

First, traverse the instruction list from `top to bottom` to do optimization.

Traverse to see whether each variable is used after definition or not. If not, delete the definition.

Traverse again and again until there are no deletions.

(Since variables are used after definition, so it is up to down traversal.)

```
//从上往下遍历instructions，删除无用赋值
int changed = 0;
do{
    IRInstruction *uptodown_ptr = Start->next;
    changed = 0;
    while (uptodown_ptr != NULL)
    {
        //有 := 的代码才有可能是无用赋值
        if (uptodown_ptr->op == _ASSIGN || uptodown_ptr->op == _PLUS || uptodown_ptr-
>op == _MINUS
            || uptodown_ptr->op == _MUL || uptodown_ptr->op == _DIV)
        {
            int used = var_is_use(uptodown_ptr);
            if (!used)
            {
                delete_instruction(uptodown_ptr);
                changed = 1;
            }
        }
        uptodown_ptr = uptodown_ptr->next;
    }
} while (changed);
```

Second, traverse the instruction list from `bottom to top` to do optimization.

Traverse from bottom to see if label is useful or not. Since labels are used before definition, thus it is bottom to top traversal.

Besides, in this type of traversal, can also solve a simplification case that simplify "t := v + #1, v := t" to "v := v + #1". Variable "v" can be seen as used before definition in this case.

```
//从下往上遍历instruction做优化
IRInstruction *bottomup_ptr = End;
while (bottomup_ptr->index > Start->index)
{
    //删除无用label
    if (bottomup_ptr->op == _LABEL)
    {
        bottomup_ptr = delete_useless_label(bottomup_ptr);
    }
```

```
    //简化i=i+1这种赋值的代码
    else if (bottomup_ptr->op == _ASSIGN && bottomup_ptr->arg1[0] == 't')
    {
        bottomup_ptr = simplify_ii1(bottomup_ptr);
    }
    else{
        bottomup_ptr = bottomup_ptr->previous;
    }
}
```

No Bonus Implemented