Report for CS323-Project3

Jin Zhaoxuan
11911413

Xu Tianqi
11912921

Zhao Yunlong
11911309

I. Overview

Our program had used *L-attributed syntax-directed definition* (SDD) to implement a semantic analysis which has mentioned in our report 2. Two models, *Type* and *Symbol table*, are used in this report as well. We will introduce new data structures to do intermediate-code generation.

The content of our algorithm can be divided into two parts. The first one is the implementation of three-address code(TAC) generation. The second is parsing syntax tree and use the TAC we written to generate final code.

- Multiple scope design. Variables in different scopes can share the same identifier, and variables defined in the outer scope will be shadowed by the inner variables with the same identifier.
- Non-conditional expressions in control flow statements. We can use an integer to express the conditional expression like C.

In this report, we will start from introducing some basic data structures we used in this project to support the analysis. Then we will talk about the process of the intermediate code generation and introduce the implementation of the features above in detail. After that, we will talk about the optimization we introduced to shrink the size of final code size and count of variables. Finally, bonus features are introduced to show the extra work we did to make our SPL more powerful.

II. Data Structure

We used Three-Address Code (TAC) as our Intermediate Representation (IR), and we designed a series of structures to store the content of TAC instead of directly storing the TAC in the form of string. This is mainly out of the consideration that we want to reserve the information of TAC as much as we can so that we can optimize the codes easier and better.

We stored the TAC nodes in a doubly-linked list so we can do insertion, deletion and element replacement easily.

```
struct TACNode
{
    TAC *tac;
    TACNode *pre;
    TACNode *next;
};
```

The structure `TAC` represents an TAC instruction.

```
struct TAC
{
    TACType type;
    Operator operator1;
    const char *result;
    const char *operand1;
    Operator operator2;
    const char *operand2;
};
```

To print the TAC more easily and reserve more information, we divided all the TAC instructions into 14 types according to the structure of the instructions.

```
enum TACType
{
    LBL = 0,        // LABEL x :
    FUNC = 1,       // FUNCTION x :
    ASSIGN = -1,    // z := x [op] y
    COPY = -2,      // [op] z := [op] x
    CONB = -4,      // IF x [relop] y GOTO z
    CALL = -5,      // z := CALL x
    DEC = -6,       // DEC x [y]
    RET = 2,        // RETURN x
    PARAM = 3,      // PARAM x
    ARG = 4,        // ARG x
    READ = 5,       // READ x
    WRITE = 6,      // WRITE x
    GOTO = 7,       // GOTO x
    EMPTY = -7
};
```

In the struct `TAC`, `x` will be stored in `operand1`, `y` will be stored in `operand2` and `z` will be stored in `result`. The operator on the left side will be stored in `operator1` and the operator at the right side will be stored in `operator2`.

We offered some functions to let the main analyzer generate and manipulate the TAC codes. Take `gen_single()` as an example.

```
TACNode *gen_single(TACType type, char *operand)
{
    TAC *tac = malloc(sizeof(TAC));
    TACNode *node = malloc(sizeof(TACNode));

    tac->type = type;
    tac->operator1 = tac->operator2 = NONE;
    tac->operand1 = operand;
    tac->result = tac->operand2 = NULL;

    node->tac = tac;
    node->next = node->pre = node;
    return node;
}
```

Function `gen_single()` is used to generate TAC instructions which has only one operand in it, such as `LABEL x`. In generate functions we will generate a `TACNode` object according to the information provided by the main analyzer and then return the pointer.

```
TACNode *combine(int num, ...)
{
    TACNode *head = NULL, *cur, *temp;
    va_list valist;
```

```
    va_start(valist, num);

    for (int i = 0; i < num; i++)
    {
        cur = va_arg(valist, TACNode *);
        if (head == NULL)
            head = cur;
        else
        {
            head->pre->next = cur;
            cur->pre->next = head;
            temp = head->pre;
            head->pre = cur->pre;
            cur->pre = temp;
        }
    }
    return head;
}
```

The function `combine()` is used to combine the linked list of `TACNode`, which represents a block of TAC codes, and then return a combined block of codes. The main analyzer can use this to join the TAC codes together.

At last we provided a function `TAC_code_gen()` to allow the main analyzer to output the linked list of `TACNode`, so that when this function is called, it will print the TAC codes stored in linked list into a specified file.

III. Optimization

A. Methodology

By following the expression translation schemes provided in project document, we can generate TAC correctly but ineffectively. We propose a method to reduce redundant code in our generation.

In the provided basic expression translation schemes, a line of spl code `a = 1` will be translated as:

```
t1 := #1
a := t1
```

The temporary variable `t1` is unnecessary for this translation, which can be simplified to one line:

```
a := #1
```

This problem is occurred in translating two types of production, $Exp \rightarrow INT$ and $Exp \rightarrow ID$.

To fix the problem, during translating these two types of production, we will directly change the variable name stored in `place` and do not return any TAC code rather than assign the value to a temporary variable.

```
/* Exp -> INT */

value = to_int(INT)
place = '#' + value
```

```
/* Exp -> ID */

variable = symtab_lookup(ID)
place = variable
```

B. Experiment

We tested the improvement of our optimization by comparing the instruction number executed between optimization and unoptimization. The results embody effectiveness of our optimization.

Testcase	Unoptimization	Optimization	Improvement
test_a(12,18)	92	63	31.52%
test_a(128,72)	116	79	31.90%
test_b(10,50)	141	79	43.97%
test_c(10)	165	91	44.85%

IV. Bonus Features

A. Scope

In the translated TAC, there is no concept of scope and all the variables are global. As a result, if the source code defined variables with same names in different scopes and we directly translate them into TAC, the same names will cause unexpected results.

We solved this problem by renaming the variables with a unique alias. There is a global counter in our symbol table to generate a increasing variable id, and when a new variable is inserted into the symbol table, an alias will be created as `var_[id]` and give to the inserted variable, then stored in the symbol table.

We offered a function `find_alias()` to the main analyzer to let it query the alias of one variable in the current scope. Then when generating TAC, the main analyzer will replace a variable's original name with its unique alias, so that there will be no name conflict in the TAC codes generated.

B. Non-conditional expressions in control flow statements

Since C is weakly typed, non-conditional expressions can also be used in control flow statements. For example, the loop body of `while(T--){}` will execute T times for any non-negative integer T, and the loop terminates when T reaches zero. We borrow this feature in C: integer 0 represents `False` while all other non-zero values, including positive and negative integer, represent `True`.

To implement this feature, we have modified the translation schemes of conditional expressions. If function `translate_cond_Exp` receives a non-conditional expression, it will take the following actions.

```
t1 = new_place()
code1 = translate_Exp(node, t1)
code2 = [IF t1 != #0 GOTO lb_t] + [GOTO lb_f]
return code1 + code2
```

We construct 4 testcases to verify the validation of these bonus features.