CS323 Project3

Yifei Li 11811905

Zhuochen Xiong 11811806

Introduction

In this project, we aim to generate the particular **intermediate representation** (IR) for the source code that has been syntax- and semantic-checked. In particular, the middle representation of the code we are talking about here is the triple address code (TAC).

The requirements of the project clearly state that the test samples will not contain code with syntax errors or semantic errors. However, to maintain the integrity of the project, we still performed a semantic check using the semantic checker we implemented in the previous project before generating the intermediate representation of the code.

Our project can be tested in all test cases and pass. In other words, legal intermediate representation can be generated and run with **irsim** to get correct results. In addition to achieving the basic requirements, we have also made certain optimizations to the generated intermediate representation so that they can obtain the correct results with fewer instructions executed. In addition, we also implemented the bonus of the project: support for both structure and array data structures.

Design & Implement

IR Represent

We define the **Tac** class as the data structure of IR, in which we use the **TacType** enumeration type to represent the type of IR statement, opands and operands to represent some operators and necessary operands (not all parameters are needed for every type of IR statement).

```
class Tac {
public:
    string op;
    string operands[3]; // {ARG1,ARG2,RESULT}
    vector<int> suffix; // suffix mem
    vector<int> sizes;
    Type *type;
    enum TacType{
        LABEL, FUNC,
        ASSIGN,
        ARITH,
        ASSIGN_ADDRESS, ASSIGN_VALUE, COPY_VALUE, CALL,
        IF,
        DEC,
        GOTO, RETURN, PARAM, ARG, READ, WRITE,
        EXIT} tac_type;
    Tac(TacType tac_type);
    Tac(TacType tac_type, const string& op, const string& arg1);
    Tac(TacType tac_type, const string& op, const string& arg1, const string& res);
    Tac(TacType tac_type, const string& op, const string& arg1, const string& arg2,
const string& res);
    Tac(TacType tac_type, const string& op, const string& res, vector<int> arr, string
name);

    void to_string(); // generate IR stmt
    string append_self(); // return variable name
}
```

To store some additional information, we also define the following structures:

1. tac_vector is the set of the final ir we generate, which we manipulate during generation and optimization.
2. value_info is used to record the mapping between the variable names in the code and the variable names we assign to the variables in the ir, which makes it easier for us to call the variables repeatedly when they are
3. glo_symbolTable is the symbol table for this project, with some of the logic related to the definition domain removed compared to the symbol table in the previous project (the sample in this project does not involve the determination of the definition domain)

```
vector<Tac*> tac_vector; // ir list
unordered_map<string, string> value_info; // key: original var name, value: var name in
ir, i.e. t1
extern SymbolTable glo_symbolTable; // extension symbol table for this proj.
```

Translate scheme

In this project, we parse the syntax tree in a similar way as in the previous project. However, in order to decouple the process of code generation and semantic checking, we choose to re-run the recursion of the syntax numbers after we finish the semantic checking. The order of recursion is roughly the same, but the process does not require type checking, but only the generation of the corresponding intermediate representation according to the structure.

```
// function declared
void ir_starter(ast_node *root);
void ir_ext_def_list(ast_node *node);
void ir_ext_dec_list(ast_node *node, Type * type);
void ir_ext_def(ast_node *node);
Type *ir_specifier(ast_node *node);
void ir_func(ast_node *node);
void ir_comp_stmt(ast_node *node);
void ir_def_list(ast_node *node);
Type* ir_structSpecifier(ast_node * node);
void ir_def(ast_node *node);
void ir_dec_list(ast_node *node, Type *type);
void ir_stmt(ast_node *node);
void ir_stmt_list(ast_node *node);
void ir_dec(ast_node *node, Type *type);
Tac* ir_var_dec(ast_node *node, Type* type, bool isParam);
void ir_var_list(ast_node *node);
void ir_param_dec(ast_node *node);
void translate_cond_exp(ast_node *exp, string lb_t, string lb_f);
void translate_exp(ast_node *exp, string& place);
void translate_args(ast_node *exp, list<string>* arg_list);
```

Optimization

In order to make our code execution more efficient, we have optimized IR for the following three cases.

1. reduce the use of labels in jump statements
2. reduce the number of assignments in the read function
3. precompute the expressions with constants in ir and merge multiple expressions

GOTO & IF-GOTO & Label Stmt

We reduce the number of labels and goto's by advancing the post-label through the conditional judgment in the reverse conditional statement, and then entering another label if the condition is not met.

```
// before optimization
if a > b goto label1
goto label2
label label1
...
label label2
...

// after optimization
if a <= b goto label2
...
label label2
...
```

READ

In our IR, `read()` is executed as a built-in function. So the original IR would use a temporary variable to get the result of `read()` and assign it to a variable. We simplify the statement by reading the result of the `read()` function directly in a variable.

```
// before optimization
read t1
v1 := t1

// after optimization
read v1
```

Constant

For some special temporary variables precomputed. Complex expression computations in some cases of source code use some intermediate variables to store the intermediate results of the computation. In the optimization phase, we iterate through and find these intermediate variables, precompute the final result and assign it directly, which will reduce the intermediate computation process when it finally runs.

```
// before optimization
t1 := #10 + #5
t2 := #10 + t1
v1 := t1 + t2

// after optimization
v1 := #40
```

Bonus

This project implements support for structures and arrays and their hybrid structures. In the addressing of array structures, we can support immediate addressing and do not support the addressing of nested functions or expressions.

```
unordered_map<string,int> ref_map;
```

To implement these two data structures, we create a **ref_map** to record their reference relationships, where key is the variable name and value has a value of 0 or 1, indicating whether it is a reference type.

Structure

In a struct structure, we need to calculate the size of the space it needs to request, i.e., iterate over the type of each field in the struct and calculate its corresponding size.

```
int cal_struct_size(Type *type){
    if(dynamic_cast<StructureType*>(type) == NULL) return 0;
    int res = 0;
    StructureType* tmp = dynamic_cast<StructureType*>(type);
    for(int i = 0; i < tmp->field_size ;i++){
        Type* child = tmp->fields[i];
        string str_Array = "Array";
        string str_structure = "Structure";
        if (child->name == "Primitive_int"){
            res += 4;
        }else if(child->name.compare(0,str_structure.size(),str_structure) == 0){
            res += cal_struct_size(child);
        }else if (child->name.compare(0,str_Array.size(),str_Array) == 0){
            res += cal_array_size(child);
        }
    }
    return res;
}
```

Array

Unlike structure, array is calculated as the product of the space occupied by its own data structure and the length of space it opens up. However only `int` can be used for addressing, which means `a[1][1]` is available but `a[i][j]` is not.

```
Type* get_array_dim(Type* type, vector<int> * vec){
    if( dynamic_cast<ArrayType*>(type) == NULL){
        return type;
    }else{
        vec->push_back(dynamic_cast<ArrayType*>(type)->size);
        return get_array_dim(dynamic_cast<ArrayType*>(type)->base,vec);
    }
}

Type* get_array_type(Type* type){
    if( dynamic_cast<ArrayType*>(type) == NULL){
        return type;
    }else{
        return get_array_type(dynamic_cast<ArrayType*>(type)->base);
    }
}

int cal_array_size(Type *type){
    if(dynamic_cast<ArrayType*>(type) == NULL) return 0;
    vector<int> * dim = new vector<int>;
    Type* root_type = get_array_dim(type, dim);
    int cnt = 1;
    for(int i = 0; i < dim->size() ;i++){
        cnt *= (*dim)[i];
    }
    if(dynamic_cast<StructureType*>(root_type) != NULL){
        return cnt* cal_struct_size(root_type);
    }else{
        return cnt*4;
    }

}
```