CS323 Project3

Intermediate-Code Generation

Team Members: Li Yuanzhao(11812420), Xu Xinyu(11811536), Jiang Yuchen(11812419)

I. Overview

In this project, we are required to implement the generation of intermediate code of given programs with SPL language. We suppose that there is no lexical or semantic error. **Three Address Codes(TAC)** are generated according to given rules. Our files can be run successfully with GCC version 7.4.0, GNU Flex version 2.6.4 and GNU Bison version 3.0.4 .

II. Design and Implementation

A. InterCode

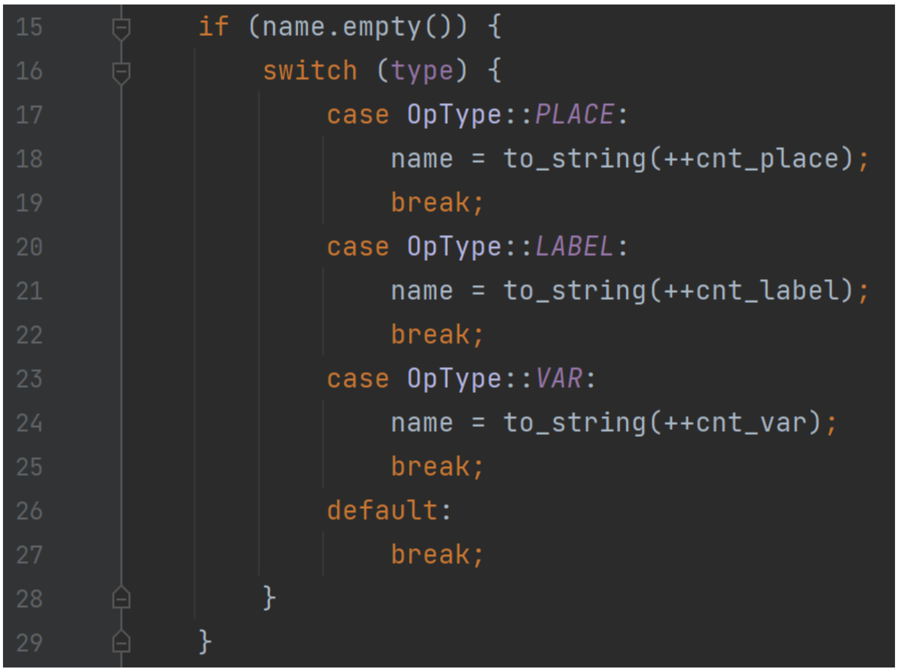
We construct the class named `InterCode` to record 24 kinds of TAC information according to appendix. An InterCode object can have at most three Operand objects for storing information. The `NONE` type is defined for unnecessary instructions which can be reduced for less instructions.

```
#include <string>
using namespace std;
enum class OpType { PLACE,LABEL,VAR,IMMEDIATE,NAME,NONE };
enum class AddrType { VALUE, ADDR };
class Operand {
public:
    OpType type;
    string name;
    AddrType addr_type = AddrType::VALUE;
    Operand(OpType type);
    Operand(OpType type, string name);
    Operand(OpType type, AddrType addrType);
    string get_name();
};
enum class InterCodeType {
    NONE = 0,LABEL,FUNCTION,ASSIGN,ADD, SUB, MUL,
DIV,ADDR,PTR,COPY,GOTO,IF_S,IF_SE,IF_B,IF_BE,IF_NE,IF_EQ,
    RETURN,DEC,PARAM,ARG,CALL,READ,WRITE
};
class InterCode {
public:
    InterCodeType interCodeType;
    Operand *x{};
    Operand *y{};
    Operand *z{};
    InterCode();
    InterCode(int type, Operand *x = nullptr, Operand *y = nullptr, Operand *z =
nullptr);
    void print();
};
```

B. Translate

After that, we continue to build up translate functions for expressions, statements, arguments and so on. According to the given schemes tables, we implement the translate action for each rule. The results are stored in `vector<InterCode>` since the result may contains more than one TAC instruction. We use merge method to construct TAC result for `ExtDef` , `CompSt`, `DefList` and `StmtList` since we can translate them using their children components.

When translating, we may need to new place, label, variable or immediate number. We set `new_place()` , `new_label()` , `get_varOp()` and `new_immediate()` functions to generate them. Specially, we will not give exact id for new place, new label and new variable, which will be given when constructing result(Designed in `InterCode.cpp`).

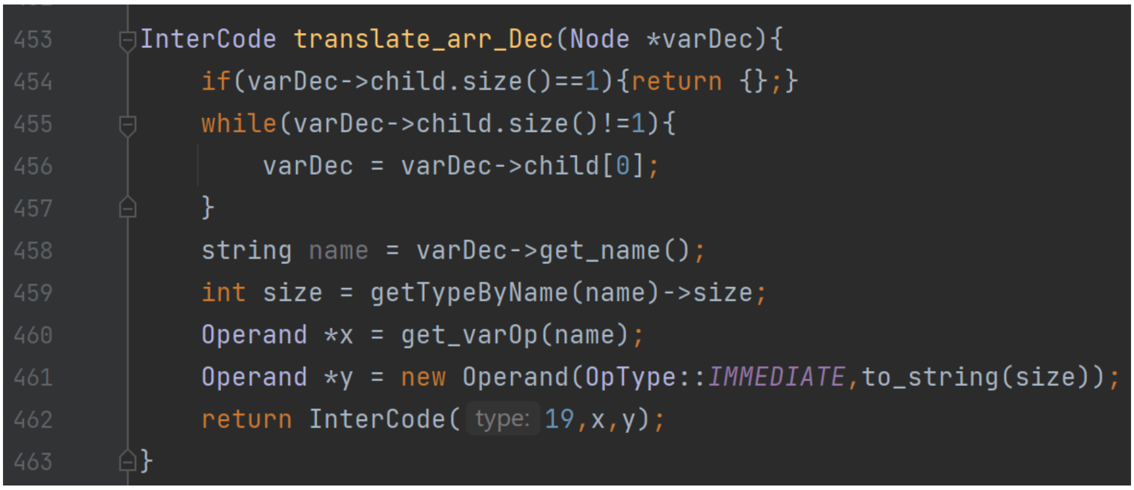


```
if (name.empty()) {
    switch (type) {
        case OpType::PLACE:
            name = to_string(++cnt_place);
            break;
        case OpType::LABEL:
            name = to_string(++cnt_label);
            break;
        case OpType::VAR:
            name = to_string(++cnt_var);
            break;
        default:
            break;
    }
}
```

Figure.1 Using global counters to give names

C. Bonus

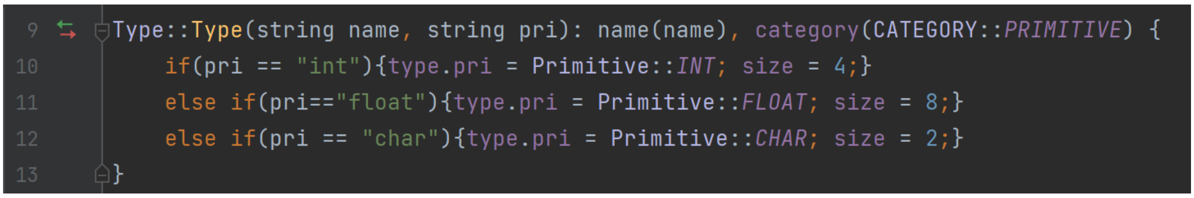
1. What's more, we support translating array declaration which is provided in `translate_arr()` , and `translate_arr_Dec()` functions. Also, we modify the class `spl_type` and add an int member `size` for it to store the space cost for possible usages. `translate_arr_addr()` are used for computing offset when fetching the address.



```
InterCode translate_arr_Dec(Node *varDec){
    if(varDec->child.size()==1){return {};}
    while(varDec->child.size()!=1){
        varDec = varDec->child[0];
    }
    string name = varDec->get_name();
    int size = getTypeByName(name)->size;
    Operand *x = get_varOp(name);
    Operand *y = new Operand(OpType::IMMEDIATE,to_string(size));
    return InterCode( type: 19,x,y);
}
```

Figure.2 `translate_arr_Dec()` function

When translating array or structure, we can start from its primitive type which is set in type constructor. Using them and the array dimensions or struct contents to count for final space cost. We design `translate_offset()` and `get_struct_offset()` for structure and array data which need to address to access. It's useful for TAC-19 which is `DEC x [size]` .

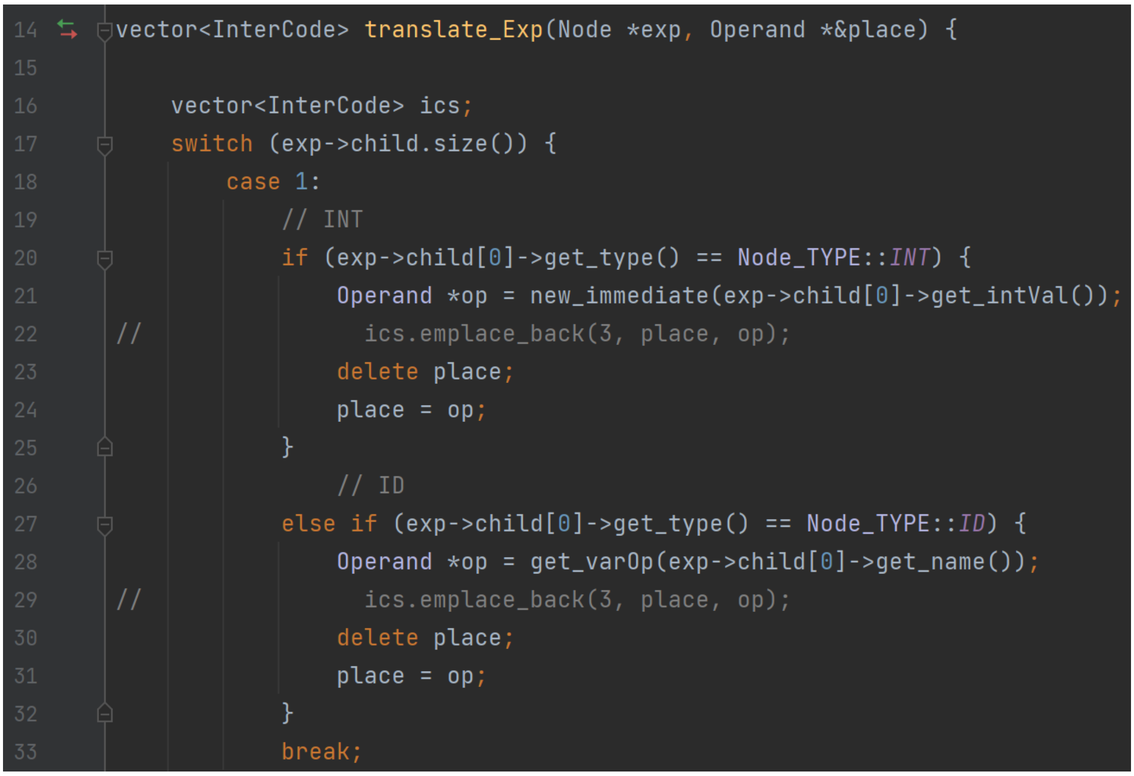


```
Type::Type(string name, string pri): name(name), category(CATEGORY::PRIMITIVE) {
    if(pri == "int"){type.pri = Primitive::INT; size = 4;}
    else if(pri=="float"){type.pri = Primitive::FLOAT; size = 8;}
    else if(pri == "char"){type.pri = Primitive::CHAR; size = 2;}
}
```

Figure.3 Set size for primitive type

D. Optimization

1. When translating expressions, if the expression is a single INT or or ID, we don't need to store the operand in given place. Thus, we will not add the TAC instruction into translation result.



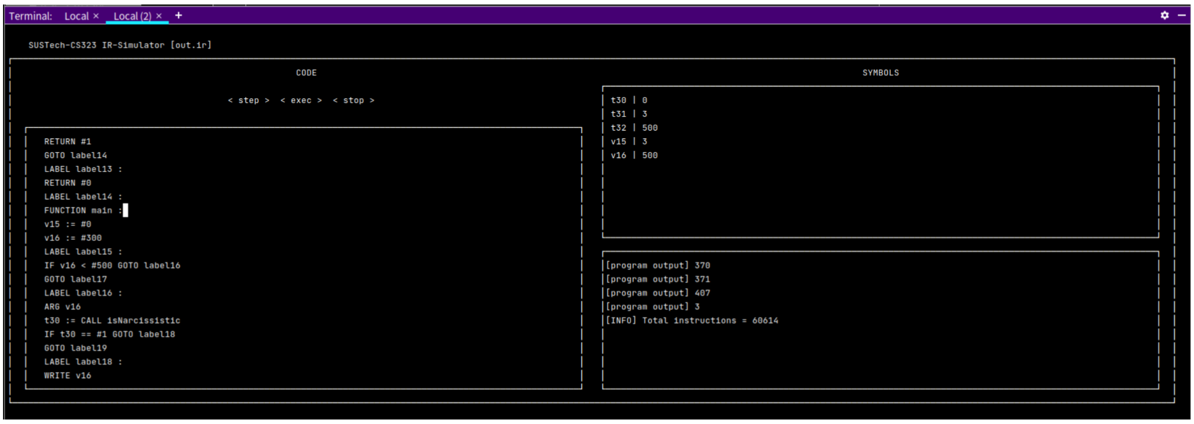
```
vector<InterCode> translate_Exp(Node *exp, Operand *&place) {

    vector<InterCode> ics;
    switch (exp->child.size()) {
        case 1:
            // INT
            if (exp->child[0]->get_type() == Node_TYPE::INT) {
                Operand *op = new_immediate(exp->child[0]->get_intVal());
//                ics.emplace_back(3, place, op);
                delete place;
                place = op;
            }
                // ID
            else if (exp->child[0]->get_type() == Node_TYPE::ID) {
                Operand *op = get_varOp(exp->child[0]->get_name());
//                ics.emplace_back(3, place, op);
                delete place;
                place = op;
            }
            break;
```

Figure.4 Remove unnecessary instructions

III. Test Cases

For give test cases, we generate their TAC results and simulate it with Irsim. All of them can be generated successfully and can be simulated for correct answer.



```
SUSTech-CS323 IR-Simulator [out.ir]

CODE

< step >  < exec >  < stop >

RETURN #1
GOTO label14
LABEL label13 :
RETURN #0
LABEL label14 :
FUNCTION main :
v15 := #0
v16 := #300
LABEL label15 :
IF v16 < #500 GOTO label16
GOTO label17
LABEL label16 :
ARG v16
t30 := CALL isNarcissistic
IF t30 == #1 GOTO label18
GOTO label19
LABEL label18 :
WRITE v16
```

```
SYMBOLS

t30 | 0
t31 | 3
t32 | 500
v15 | 3
v16 | 500
```

```
[program output] 370
[program output] 371
[program output] 407
[program output] 3
[INFO] Total instructions = 60614
```

Figure.5 Irsim Result for Test case 9(60614 instructions)

For extra test cases, we put them in `./test-ex/` folder which contains four test cases. They are used for arrays&structures' translation.

IV. Instructions

Change directory to the root path and using `make splc` to create `splc` in `./bin` root for spl codes' parsing. Then using `bin/splc <test_root>/<test_file_name>` to create immediate code result. And you can use `make clean` to delete all created files.