

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
};
class Operand {
public:
    OpType type;
    string name;
    Operand(OpType type);
    Operand(OpType type, string name);
    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`).

```
15     if (name.empty()) {
16         switch (type) {
17             case OpType::PLACE:
18                 name = to_string(++cnt_place);
19                 break;
20             case OpType::LABEL:
21                 name = to_string(++cnt_label);
22                 break;
23             case OpType::VAR:
24                 name = to_string(++cnt_var);
25                 break;
26             default:
27                 break;
28         }
29     }
```

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.

```
453 InterCode translate_arr_Dec(Node *varDec){
454     if(varDec->child.size()==1){return {};}
455     while(varDec->child.size()!=1){
456         varDec = varDec->child[0];
457     }
458     string name = varDec->get_name();
459     int size = getTypeByName(name)->size;
460     Operand *x = get_varOp(name);
461     Operand *y = new Operand(OpType::IMMEDIATE,to_string(size));
462     return InterCode( type: 19,x,y);
463 }
```

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. It's useful for TAC-19 which is `DEC x [size]`.

```
9  Type::Type(string name, string pri): name(name), category(CATEGORY::PRIMITIVE) {
10      if(pri == "int"){type.pri = Primitive::INT; size = 4;}
11      else if(pri=="float"){type.pri = Primitive::FLOAT; size = 8;}
12      else if(pri == "char"){type.pri = Primitive::CHAR; size = 2;}
13  }
```

Figure.3 Set size for primitive type

2. We support `For` statements' translation which is implemented in `vector<InterCode> translate_stmt(Node *stmt)`. Its actions are similar as `while` 's actions.

```
250      // FOR LP Def Exp SEMI Exp RP Stmt
251      else if (stmt->child.size() == 8) {
252          // Def
253          vector<InterCode> def = translate_Def(stmt->child[2]);
254          translate.insert(translate.end(), def.begin(), def.end());
255          // WHILE Exp_2 Stmt + Exp_3
256          Operand *lb1 = new_label();
257          Operand *lb2 = new_label();
258          Operand *lb3 = new_label();
259          vector<InterCode> exp2 = translate_cond_Exp(stmt->child[4], lb2, lb3);
260          vector<InterCode> code1 = translate_stmt(stmt->child[8]);
261          Operand *tp2 = new_place();
262          vector<InterCode> code2 = translate_Exp(stmt->child[6], tp2);
263          translate.emplace_back(1, lb1);
264          translate.insert(translate.end(), exp2.begin(), exp2.end());
265          translate.emplace_back(1, lb2);
266          translate.insert(translate.end(), code1.begin(), code1.end());
267          translate.insert(translate.end(), code2.begin(), code2.end());
268          translate.emplace_back(11, lb1);
269          translate.emplace_back(1, lb3);
270      }
```

Figure.4 Translation For Statements

- 3.

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.

```

14  vector<InterCode> translate_Exp(Node *exp, Operand *&place) {
15
16      vector<InterCode> ics;
17      switch (exp->child.size()) {
18          case 1:
19              // INT
20              if (exp->child[0]->get_type() == Node_TYPE::INT) {
21                  Operand *op = new_immediate(exp->child[0]->get_intVal());
22                  ics.emplace_back(3, place, op);
23                  delete place;
24                  place = op;
25              }
26              // ID
27              else if (exp->child[0]->get_type() == Node_TYPE::ID) {
28                  Operand *op = get_varOp(exp->child[0]->get_name());
29                  ics.emplace_back(3, place, op);
30                  delete place;
31                  place = op;
32              }
33              break;

```

Figure.5 Delete unnecessary instructions

2.

III. Test Cases

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

- Test case with Array Translation

```

// TBD
-----
// TBD

```

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.