Project: Smallc Compiler

Kangwei Ling 5140219295

January 19, 2017

Contents

1 Lexical Analysis

In this project, I use *flex* to do lexical analysis, which scans through the source code text file and generates token stream. With the language specification, it's very easy to write down the lexical rules for flex to work on. In addition, the token type can be found in the header file generated from *yacc / bison*.

1.1 Keywords

The rules for keywords should be put before identifiers, so that keywords will not be parsed as identifiers.

```
"int"         return(TYPE);
";"         return(SEMI);
","         return(COMMA);
"("         return(LP);
")"         return(RP);
"["         return(LB);
"]"         return(RB);
"{"         return(LC);
"}"         return(RC);
"struct"      return(STRUCT);
"return"      return(RETURN);
"if"          return(IF);
"else"        return(ELSE);
"break"       return(BREAK);
"continue"    return(CONT);
"for"         return(FOR);
```

1.2 Identifiers and Integers

An identifier is a character string consisting of alphabetic characters, digits and the underscore. Digits can't be the first character. Use the notation of lex, it can be write down as,

```
id      [_a-zA-Z][_a-zA-Z0-9]*
```

For numbers, to support 3 format (decimal, octal, hexadecimal), the following notation is used,

```
integer ([0-9]+|0[xX][0-9a-f]+)
```

Thus the rules for identifiers and integers can be written as,

```
{integer}       { save_num(); return(INT); }
{id}        { save_str(); return(ID);  }
```

The two `save_` function is defined to save the `yytext` as token value. As for integers, a transformation from string to int is done with the limit checked (-2^{31}, 2^{31}).

```
void save_num()
{
    try {
      yylval.ival = stoi(string(yytext), 0, 0); // autodetect base
    } catch (std::invalid_argument& e) {
      std::cerr << "Error token: "<< yytext << " found at line: " << curr_lineno
            << ": invalid number."<< endl;
      yylval.ival = 0;
    } catch (std::out_of_range& e) {
    std::cerr << "Error token: "<< yytext << " found at line: " << curr_lineno
          << ": out of range( -2^(31), 2^(31) )."<< endl;
    yylval.ival = 0;
    }
}
```

1.3 Operators

There are many valid operators for *smallc* language. A careful naming approach must be adopted. I use a **L** prefix for logical operators and **B** prefix for binary operators. A suffix of **ASSIGN** is used to denote *op and assign* operators. Below is a small snippet of this part.

```
"."    { save_str(); return(DOT); }
"+"    { save_str(); return(ADD); }
">>"   { save_str(); return(SHR); }
">"    { save_str(); return(GT); }
">="   { save_str(); return(GTE); }
"&"    { save_str(); return(B_AND); }
"&&"   { save_str(); return(L_AND); }
"="    { save_str(); return(ASSIGN); }
"&="   { save_str(); return(AND_ASSIGN); }
">>="   { save_str(); return(SHR_ASSIGN); }
"!"     { save_str(); return(L_NOT); }
"--"    { save_str(); return(P_SUB); }
```

It's important to pay attention to binary minus and unary minus, but in lexical analysis, there is no way to tell whether it's a binary one or unary one. Therefore, this job is delayed to syntax analysis.

```
    /* Minus(-) need to be processed in parser */
"-"     { return(SUB);}
```

1.4 Line Number

Line number is very important for developing and debugging, thus it's necessary to save the line number when scanning for tokens.

```
\n  ++curr_lineno;
```

Whenever a newline symbol is detected, the variable `curr_lineno` is updated, this value is also used when creating abstract syntax tree nodes in syntax analysis.

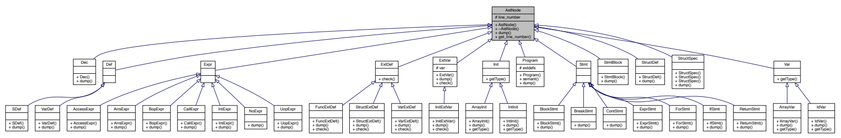
2 Syntax Analysis

2.1 Abstract Syntax Tree

I create a abstract syntax node class for every possible production. All classes inherit from `AstNode` class,

```
class AstNode {
protected:
    int line_number;
public:
    AstNode();
    virtual ~AstNode() { }
    virtual void dump(ostream& os, int n) = 0;
    int get_line_number();
};
```

Each node must have a line number for debugging, and a `dump` method to print out necessary information of this node. I implemented `dump` method as to print out the source code that generate this node, to verify the correctness of the parsing process.Below is a simple inherit class diagram.



The detail of each class is in the source code.

2.2 Bison and Grammar

Bison is the tool used to generate a parser from given grammars. In this project, I associate each production with semantic actions so that the abstract syntax tree is created during the parsing process.

The types of terminals and non-terminals are declared in yylval union.

```
%union {
    int ival;
    std::string* sval;
    Program* program;
    ExtDefList* extdefs;
    ExtDef* extdef;
    ExtVar* extvar;
    ExtVarList* extvars;
    SExtVar* sextvar;
    SExtVarList* sextvars;
    StructSpec* stspec;
    Paras* paras;
    StmtBlock* stmtblock;
    StmtList* stmts;
    Stmt* stmt;
    DefList* defs;
    Def* def;
    StructDef* stdef;
    StructDefList* stdefs;
    SDec* sdec;
    SDecList* sdecs;
    Dec* dec;
    DecList* decs;
    Var* var;
    Init* init;
    Arrs* arrs;
    Args* args;
    Expr* exp;
}
```

All operator terminals have `sval` (pointer to string) value to be stored in `BopExpr` node class. For non-terminals, they have corresponding class type.

For simplicity, I made a little tweak to `EXTDEF` production for functions.

```
EXTDEF -> TYPE EXTVARS SEMI
        | STSPEC SEXTVARS SEMI
        | TYPE ID LP PARAS RP STBLOCK     (substitute FUNC in)
```

For operator precedences, just implement them as the specification says.

```
%right ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSI
%left L_OR
%left L_AND
%left B_OR
%left B_XOR
%left B_AND
%left EQ NEQ
%left GT GTE LT LTE
%left SHL SHR
%left ADD SUB
%left MUL DIV MOD
```

```
%right UMINUS L_NOT P_ADD P_SUB B_NOT
%left DOT LP RP LB RB

%right THEN ELSE
```

1. binary minus and unary minus

   To solve this problem, just set the precedence over the specific production of UMINUS

   ```
   | SUB   EXPS %prec UMINUS    { $$ = new UopExpr(*$1, $2); }
   ```

2. if else

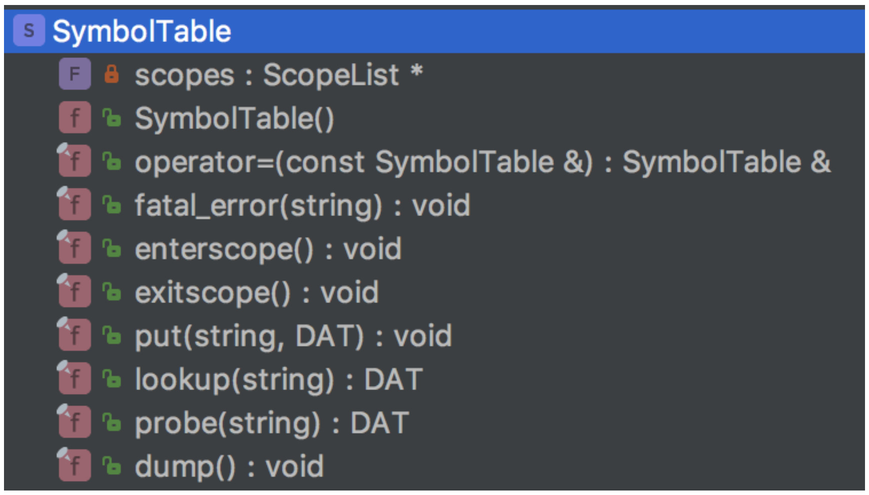
   Note that the `else` should be bond with the last if, using `%right THEN ELSE` associativity and precedence, the rule below solve this problem.

   ```
   | IF LP EXP RP STMT %prec THEN { $$ = new IfStmt($3, $5); }
   | IF LP EXP RP STMT ELSE STMT { $$ = new IfStmt($3, $5, $7); }
   ```

3 Semantic Analysis

3.1 Symbol Table

Symbol Table is a mapping from symbols to their symbol values. In my implementation, symbol table is just a wrapper of mapping from string to template type, meanwhile support scope actions (enter a scope, exit a scope, lookup a symbol, etc).



```
SymbolTable
    scopes : ScopeList *
    SymbolTable()
    operator=(const SymbolTable &) : SymbolTable &
    fatal_error(string) : void
    enterscope() : void
    exitscope() : void
    put(string, DAT) : void
    lookup(string) : DAT
    probe(string) : DAT
    dump() : void
```

The underlying scope is defined as follows.

```
using Scope = std::unordered_map<string, DAT>;
using ScopeList = std::list<Scope>;
```

lookup lookup the symbol in the symbol table, if it's not found in the current scope, search in the previous scope (next entry of the list). This method is used when we using an identifier in places other than declarations.

probe only look for the symbol in the current scope. As an identifier can be redeclared in different scope, so we only search in the current scope when do declarations.

3.2 Semantic Check

Four different symbol table is used in my implementation.

```
SymbolTable<Var*> IntVarTbl;
SymbolTable<StructSpec*> StVarTbl;
SymbolTable<StructSpec*> StSpecTbl;
SymbolTable<FuncExtDef*> funcTbl;
```

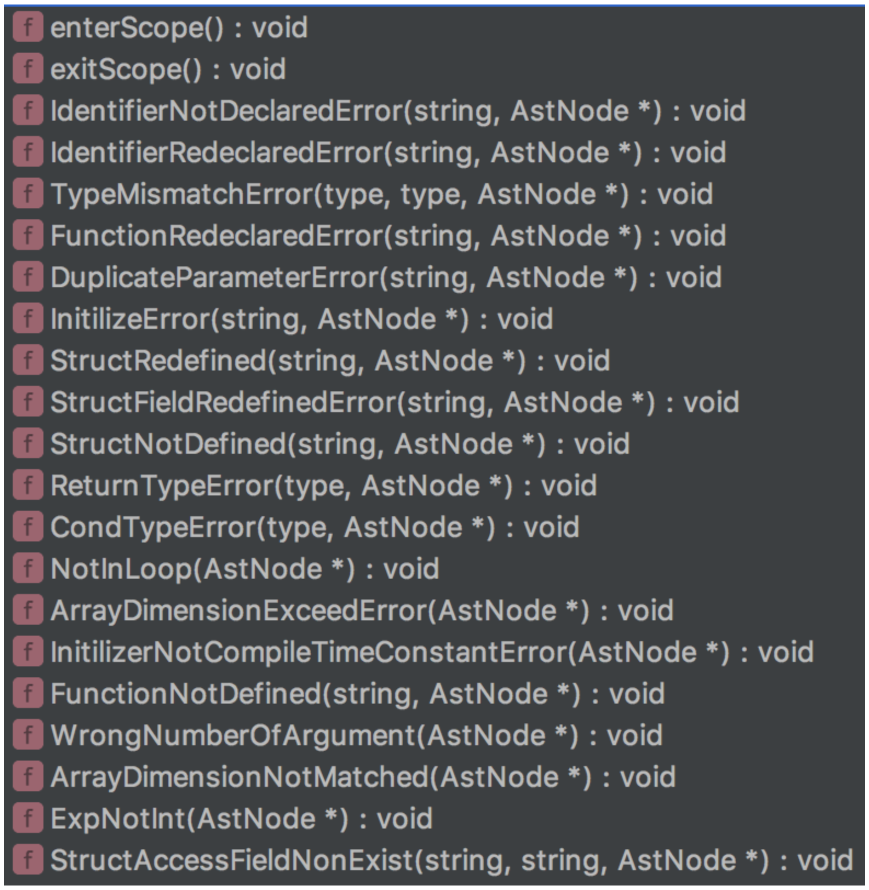
The `IntVarTbl` is used for integer and integer array variables, `StVarTbl` for struct variables, `funcTbl` for function identifiers. In addition, `StSpecTbl` is a mapping from struct tag to struct specification, for example,

```
struct A {
   int a;
};
```

there will be a mapping from *"A"* to this struct specification. Note that, for struct variables (declared ones), their id is also mapping to the corresponding spec.

3.2.1 Helper Functions

I also introduced some help functions to facilitate the semantic analysis process.



```
enterScope() : void
exitScope() : void
IdentifierNotDeclaredError(string, AstNode *) : void
IdentifierRedeclaredError(string, AstNode *) : void
TypeMismatchError(type, type, AstNode *) : void
FunctionRedeclaredError(string, AstNode *) : void
DuplicateParameterError(string, AstNode *) : void
InitilizeError(string, AstNode *) : void
StructRedefined(string, AstNode *) : void
StructFieldRedefinedError(string, AstNode *) : void
StructNotDefined(string, AstNode *) : void
ReturnTypeError(type, AstNode *) : void
CondTypeError(type, AstNode *) : void
NotInLoop(AstNode *) : void
ArrayDimensionExceedError(AstNode *) : void
InitilizerNotCompileTimeConstantError(AstNode *) : void
FunctionNotDefined(string, AstNode *) : void
WrongNumberOfArgument(AstNode *) : void
ArrayDimensionNotMatched(AstNode *) : void
ExpNotInt(AstNode *) : void
StructAccessFieldNonExist(string, string, AstNode *) : void
```

`enterscope` and `exitscope` are just wrapper of symbol table's scope functions, they are used to control the scope of multiple tables. The others are just error functions, which normally takes a string and `AstNode` as arguments and handle errors. The `AstNode` is used to get the line number of the error point. All errors will to put into a `stringstream` and update `semant_errors`.

```
static ostringstream err;
static int semant_errors = 0;
```

Upon finishing semantic checking, if there are more than one semantic errors, then the program will no proceed to code generation.

3.2.2 Checking

Semantic check is started by calling `semant` method of the `Program` class, which is the final ast root. Then it will recursively check all extdefs.

1. dimension of arrays will be no more than 2.

   This can be done by checking the `ArrayVar`. The `Var` part of `ArrayVar` must have dimension less than 2, with normal scalar having dimension of 0.

   ```
   if (this->var->getDim() >= 2)
       ArrayDimensionExceedError(this);
   ```

2. initializer compatibility.

   A integer variable can not be initialized with an array (such as {1,1}), vice versa. This is done by checking their dimensions.

   ```
   if (this->var->getType() != this->init->getType() ||
       this->var->getSize() < this->init->getSize()) {
       InitilizeError("Incompatible initializer", this);
    }
   ```

3. variables and functions should be declared before usage.

   variables and functions is only used in expressions. When checking, their identifiers are used to lookup in the symbol table. If not found, an error occurs.

4. variables and functions should not be re-declared.

   just like the previous one, search with their identifier to check if exists, but only search in current scope (`probe`). If non exists, we are all fine. (All symbol table need to be searched).

5. Reserved words.

   This is done in lexical part, as reserved words will be firstly parsed as keyword tokens, not identifiers.

6. Program must contain a function int main() to be the entrance

   Just check the `funcTbl` after checking the extdefs.

```
FuncExtDef* MainFunc = funcTbl.lookup("main");
if (MainFunc == NULL || MainFunc->getParamCount() != 0) {
    ++semant_errors;
    err << "Program must contain a function int main()!" << endl;
}
```

7. The number and type of variable(s) passed should match the definition of the function.

   While checking for `CallExpr`, check if the number of arguments equals to the function's parameter counts, and check if all arguments give a int result.

```
int n_args_prov = this->args->size();
int n_args_need = func->getParamCount();
if (n_args_need != n_args_prov)
    WrongNumberOfArgument(this);
 // check arguments
for (Expr* exp: *this->args) {
    ExprType::type tp = exp->check();
    if (tp != ExprType::INTEGER)
        ExpNotInt(exp);
}
```

8. Use `[]` operator to a non-array variable is not allowed.

   Compare the dimension of variable with the `Arrs` size (number of indices).

```
int dim = v->getDim();
int ac = this->arrs->size();
if (dim != ac) {
    ArrayDimensionNotMatched(this);
```

9. The `.` operator can only be used to a struct variable.

   Because I use different symbol table for struct variable and int variable, this is solved.

10. `break` and `continue` can only be used in a for-loop.

    I keep a global count for loops. Whenever I entered a for-loop, increment that count. Whenever done checking a for-loop, decrement it. If the count is 0 when checking node of `break` or `continue`, there must be an error.

11. Right-value can not be assigned by any value or expression.

    When check for Binary OP nodes, if the operator is associated with assignment, then check if the left expression is an lvalue. Only array access and struct access are lvalues, so this is not so difficult.

```
// check if lval
if (this->op.find("=") != std::string::npos && this->op != "=="
        && this->op != ">=" && this->op != "<="
        && this->op != "!=") {
    if (!this->lexp->isLval()) {
        ++semant_errors;
        err << "line " << this->get_line_number() << " " <<  << "error: "
            << "not an lvalue" << endl;
    }
}
```

12. The condition of `if` statement should be an expression with `int` type.

    Every expression has a type, the semantic checking of expression will return its type.

```
ExprType::type cond_tp = this->cond->check();
if (cond_tp != ExprType::INTEGER)
    CondTypeError(cond_tp, this);
```

13. The condition of `for` should be an expression with `int` type or ϵ.

    The `BLANK` type is returned by empty expression.

```
ExprType::type cond_tp = this->cond->check();
if (cond_tp != ExprType::INTEGER && cond_tp != ExprType::BLANK)
    CondTypeError(cond_tp, this);
```

14. Only expression with type `int` can be involved in arithmetic.

    This one can be checked inside `CallExpr` , `ArrsExpr` and `AccessExpr`. As other than those nodes, we only have integers.

15. global initializer should be compile time constant.

    The initializer of global variables(int and int arrays) can not contain `CallExpr`, `ArrsExpr` or `AccessExpr`.

    ```
    if (!this->init->isConstant())
        InitilizerNotCompileTimeConstantError(this->init);
    ```

3.3 Annotation

3.3.1 Indirect information replacement

When doing semantic checking, I also annotate some ast nodes like struct declarations(with full specification), array access(with var definition), struct access(with full specification) and function calls(with function definition) so that in the following phases I don't need to look up them in symbol table.

3.3.2 Compile time constant folding

The initializer of global variables(int var, int array) should be able to be computed at compile time. And the value is used to generate mips code.

I introduced an `eval` method for expressions, this method will get the value of a constant expression during semantic analysis.

The `eval` will only evaluate for binary OP and unary OP and int constant expression. For `ArrsExpr`, if it is an scalar var access(there is no subscript access to array), then return the value of `var` (which I already put it here during indirect information replacement), the default value of `var` is 0.

The value of an `Init` (either an int init or array init) will be stored in a vector (for int init, there is only one element), the value is computed by a method `eval`,

```
void IntInit::eval() {
    int v = this->exp->eval();
    //cout << "# compute: " << v << endl;
```

```
    value->push_back(v);
}

void ArrayInit::eval() {
    for (Expr* exp: *this->args) {
        int v = exp->eval();
        //cout << "# compute: " << v << endl;
        value->push_back(v);
    }
}
```

Note that, after compute compile time constant for global variables, the value of a global var should be updated with its init value, so that the future usage of this value during eval is correct.

```
this->init->eval();

// code below is associate value with int var.
// array value value will never been accesses
// during eval expression
IntInit* intinit = dynamic_cast<IntInit*>(this->init);
if (intinit != NULL) {
    (dynamic_cast<IdVar*>(this->var))->value = (*(intinit->value))[0];
    return;
}
```

3.3.3 Struct Specification flatten

Because in the grammar, specification for struct is wrapped in two layers, it will be very useful for calculating offset if the fields are in a flatten list.

```
for (StructDef* structdef : *this->sdefs) {
    structdef->check();
    fields.insert(fields.end(),
           structdef->sdecs->begin(), structdef->sdecs->end());
}
```

3.3.4 read and write

These two function is added into `funcTbl` at the beginning.

```
Paras* pp = new Paras;
pp->push_front("x");
FuncExtDef* read = new FuncExtDef("read", pp, nullptr);
FuncExtDef* write = new FuncExtDef("write", pp, nullptr);
funcTbl.put("read", read);
funcTbl.put("write", write);
```

When checking for function call, `read` call must be called with lvalue argument!

```
if (this->id == "read" && !exp->isLval()) {
    ++semant_errors;
    err << "line " << exp->get_line_number() << " " << "error: "
    << "not an lvalue" << endl;
}
```

4 IR Generation

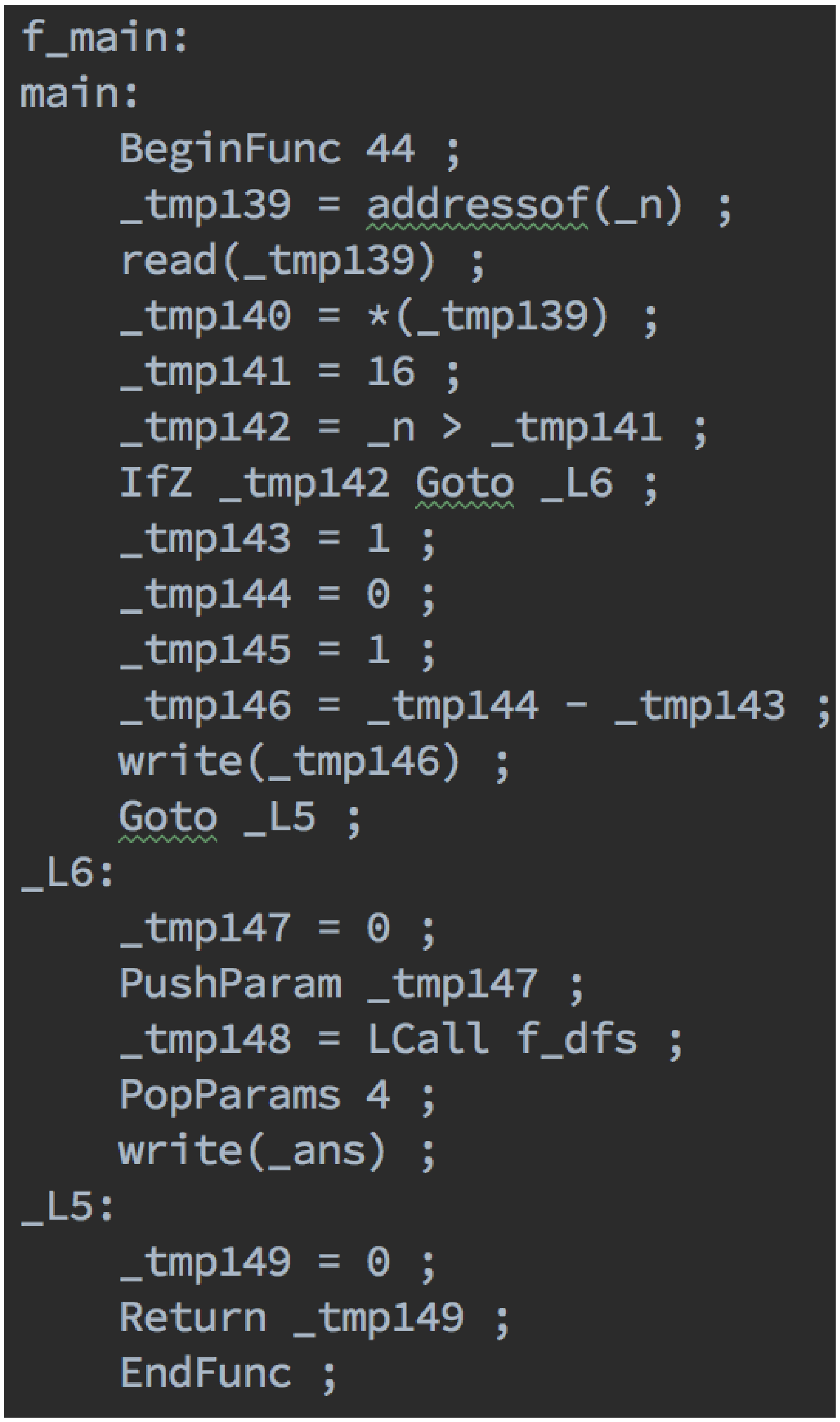
For this project, I use three address code as intermediate representation, this form of IR is very close to the assembly language so that it will be very easy to transform into final assembly code.

Every TAC class inherit from `Instruction` class.

```
class Instruction {
protected:
    char printed[128];
public:
    virtual ~Instruction() {}
    virtual void Print();
    virtual void EmitSpecific(Mips *mips) = 0;
    virtual void Emit(Mips *mips);
};
```

The printed array is the string representation of the instruction(for printing three address code out).

Below is a small IR snippet:



```
f_main:
main:
    BeginFunc 44 ;
    _tmp139 = addressof(_n) ;
    read(_tmp139) ;
    _tmp140 = *(_tmp139) ;
    _tmp141 = 16 ;
    _tmp142 = _n > _tmp141 ;
    IfZ _tmp142 Goto _L6 ;
    _tmp143 = 1 ;
    _tmp144 = 0 ;
    _tmp145 = 1 ;
    _tmp146 = _tmp144 - _tmp143 ;
    write(_tmp146) ;
    Goto _L5 ;
_L6:
    _tmp147 = 0 ;
    PushParam _tmp147 ;
    _tmp148 = LCall f_dfs ;
    PopParams 4 ;
    write(_ans) ;
_L5:
    _tmp149 = 0 ;
    Return _tmp149 ;
    EndFunc ;
```

4.1 Three Address Code

The following three address code classes are used in this project.

```
class LoadConstant; // load immediate
class Assign;
class Load;
class Store;
class BinaryOp;
class Label;
class Goto;
class IfZ;
class BeginFunc;
class EndFunc;
class Return;
class PushParam;
class RemoveParams;
class LCall;
class GlobalData;
class ReadInt;
class WriteInt;
class LoadAddress;
```

It's very clear what does each one means, you can tell by its name. The detail of each class is in file `tac.h/.cpp`.

4.2 Location

A Location object is used to identify the operands to the various TAC instructions. A Location is either `fp` or `gp` relative (depending on whether in stack or global segment) and has an offset relative to the base of that segment. For example, a declaration for integer num as the first local variable in a function would be assigned a Location object with name "num", segment fpRelative, and offset -8.

```
typedef enum {fpRelative, gpRelative} Segment;

class Location
{
protected:
    const string variableName;
    Segment segment;
    int offset;

public:
    Location(Segment seg, int offset, const string name);

    const string GetName() const     { return variableName; }
    Segment GetSegment() const      { return segment; }
    int GetOffset() const           { return offset; }

    bool locationRef;

    bool operator==(const Location& that) const {
        return this->GetName() == that.GetName() &&
       this->GetSegment() == that.GetSegment() &&
       this->GetOffset() == that.GetOffset();
    }
};
```

Each TAC class is associated with one or more `Location` (i.e. src and dst)

4.3 IR generator

The CodeGenerator class is used to build TAC instructions (using the Tac class and its subclasses) and store the instructions in a sequential list when traversing the abstract syntax tree for IR generation, ready for further processing or translation to MIPS as part of final code generation.

This class has two code lists, one for global data, another for code segment.

```
std::list<Instruction*> *code;
std::list<Instruction*> *data;
```

Also, this class holds a symbol table mapping from variable name to their locations (including temp vars). Besides, it provides utility functions to use the stack and global area, as well as generate label and temp vars.

```
string NewLabel();
Location *GenGlobalVar(const string name, unsigned int size, vector<int> *init);
Location *GetNewLocationOnStack(const string name);
Location *GetNewBulkLocationOnStack(const string name, unsigned int size);
Location *GenTempVar();
```

Apart from all these, other methods are used to generate corresponding TAC instruction. The details of the implementation is in file `IRgen.h/cpp`.

4.4 emit

The TAC instruction list is built up upon traversing the abstract syntax tree. The `emit` method of ast classes does this job. The implementation can be found in file `ast_emit.h`.

4.4.1 Global Variables

Upon start, it'will firstly generate tac for global variables(int, int array, struct). Actually these three are basically the same because they only use a sequence of 4-bytes memory. With enough information provided in the ast node, use `IRGenerator`'s `GenGlobalVar` method is very straightforward.

4.4.2 Function

For function defs, firstly a label tac will be generated for that function, locations of its parameters will be set, then its body will be emit.

```
void FuncExtDef::emit(IRGenerator *irg)
{
    irg->NewScope();

    string tmp = "f_" + this->id;
    irg->GenLabel(tmp);

    if (this->id == "main")
        irg->GenLabel("main");

    BeginFunc* bf = irg->GenBeginFunc();

    // gen stack space for parameter
    int onStackParam = 0;
    for (const string &p : *params) {
        irg->InsertLocation(p, new Location(fpRelative, 4 + onStackParam * 4, p));
        ++onStackParam;
    }

    this->stmtBlock->emit(irg);

    bf->SetFrameSize(irg->currentStackSize * 4);
    irg->GenEndFunc();

    irg->RemoveScope();
}
```

4.4.3 Statement

For statement, just follow the definition of the statement and generate correct tac. For loops to work with `break` and `continue`, I kept two global lists of exit labels and for-loop's update label. Thus `break` and `continue` will be jump to the correct label.

4.4.4 Expression

The `emit` method for expressions are a little bit difficult, because I was torn between reference(addresses) and values.

1. Binary Operators

   For normal binary operators(corresponding mips op exists), just generate BinaryOp TAC. For logical and "&&" and logical or "||", I expand them into normal operations.

   ```
   // for logical and &&
    Location* zero = irg->GenLoadConstant(0);

    Location* llres = irg->GenBinaryOp("!=", zero, lres);
    Location* rrres = irg->GenBinaryOp("!=", zero, rres);
    return irg->GenBinaryOp("&", llres, rrres);
   ```

   For operator associated with assignment, firstly the result is calculated as normal operations, and then an reference location(the address) is generated via `emitMemoryLocation`, then a store tac is generated.

2. Unary Operators

   transformed into binary operators. Bitwise negation is transformed into an XOR with -1.

3. Call function

   The arguments will be emit first and PushParam is genrated.(The last argument is pushed firstly).

4. ArrsExpr (array access)

   The `emitMemoryLocation` method will generate a `LoadAddress` TAC and the location contains that address will returned. For normal `emit` method, a `Load` TAC is generated from the memory location. Because arrays are not allowed to have dimension more than 2, therefore calculation of the offset is hardcoded

```
Location* ArrsExpr::emitMemoryLocation(IRGenerator *irg)
{
    Location* base = irg->GetLocation(var->getId());
    Location* baseLoc = irg->GenLoadAddress(base);
    int dim = var->getDim();
    if (dim == 0) {
        baseLoc->locationRef = true;
        return baseLoc;
    } else {
        Location* fourL = irg->GenLoadConstant(4);
        Location* arg0 = arrs->front()->emit(irg);
        Location* baseLoc = irg->GenLoadAddress(base);

        if (dim == 1) {
            Location* fl = irg->GenBinaryOp("+", baseLoc,
                                    irg->GenBinaryOp("*", arg0, fourL));
            fl->locationRef = true;
            return fl;
        } else {
            // two dimension
            int numInRow = var->getSize();
            Location* col = irg->GenLoadConstant(numInRow);
            Location* arg1 = arrs->back()->emit(irg);
```

```
                    Location* tmp = irg->GenBinaryOp("+",
                                        irg->GenBinaryOp("*", arg0, col), arg1);

                    Location* fl = irg->GenBinaryOp("+", baseLoc,
                                         irg->GenBinaryOp("*", tmp, fourL));
                    fl->locationRef = true;
                    return fl;
                }
            }
        }
```

5. AccessExpr struct field access

   This part is very similar to array access (one dimension array);

6. IntExpr

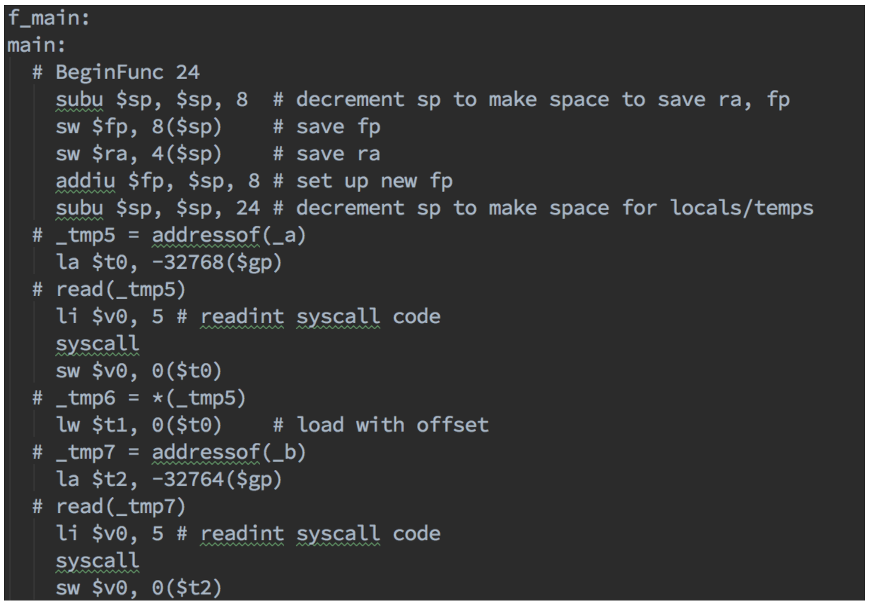
   A `LoadConstant` TAC is genrated with the int value associated with the node.

5 Code Generation

Once the IR is generated, each TAC will be translated into mips code. If `-tac` is in the end of command line arguments, then the TAC codes will be written to the file, otherwise mips code will be generated.

The code generation work is done with the `Mips` class. The Mips class defines an object capable of emitting MIPS instructions and managing the allocation and use of registers. It is used by the Tac instruction classes to convert each instruction to the appropriate MIPS equivalent.

Below is a small generated mips snippet



```
f_main:
main:
  # BeginFunc 24
    subu $sp, $sp, 8  # decrement sp to make space to save ra, fp
    sw $fp, 8($sp)    # save fp
    sw $ra, 4($sp)    # save ra
    addiu $fp, $sp, 8 # set up new fp
    subu $sp, $sp, 24 # decrement sp to make space for locals/temps
  # _tmp5 = addressof(_a)
    la $t0, -32768($gp)
  # read(_tmp5)
    li $v0, 5 # readint syscall code
    syscall
    sw $v0, 0($t0)
  # _tmp6 = *(_tmp5)
    lw $t1, 0($t0)    # load with offset
  # _tmp7 = addressof(_b)
    la $t2, -32764($gp)
  # read(_tmp7)
    li $v0, 5 # readint syscall code
    syscall
    sw $v0, 0($t2)
```

5.1 Instruction Selection

1. Binary Operation

   For operator: "*", "/", "%", "+", "-", ", ">", ">=", "<", "<=", "=", "!", "&", "^", "|", corresponding mips instructions exist, a mapping works well.

2. Load & Store

   `li` instruction is used for `LoadConstant`, `la` is used for `LoadAddress`, `lw` is used for `Load`, `sw` is used for `Store`

3. Others `move` instruction for `Assign`, `b` for `Goto`, `beqz` for `IfZ`. Details of functions call are listed in the file `mips.h/cpp`

4. Read & Write

   syscall of mips is used.

```
void Mips::EmitRead(Location *dst) {
    Emit("li $v0, 5 # readint syscall code");
    Emit("syscall");
    Register rd = GetRegister(dst);
    Emit("sw $v0, %d(%s)", 0,regs[rd].name);
}

void Mips::EmitWrite(Location *src) {
    Emit("li $v0, 1 # readint syscall code");
    Register r = GetRegister(src);
    Emit("move $a0, %s", regs[r].name);
    Emit("syscall");
}
```

5.2 Register allocation

I used a very basic register policy in this project. A method `GetRegister` is used to get a register for use.

Given a location for a current var, a reason (ForRead or ForWrite) and up to two registers to avoid, this method will assign to a var to register trying these alternatives in order:

1. if that var is already in a register ("same" is determined by matching name and same scope), we use that one
2. find an empty register to use for the var
3. find an in-use register that is not dirty. We don't need to write value back to memory since it's clean, so we just replace with the new var
4. spill an in-use, dirty register, by writing its contents to memory and then replace with the new var For steps 3 & 4, we respect the registers given to avoid (ie the other operands in this operation). The register descriptors are updated to show the new state of the world. If for read, we load the current value from memory into the register. If for write, we mark the register as dirty (since it is getting a new value).