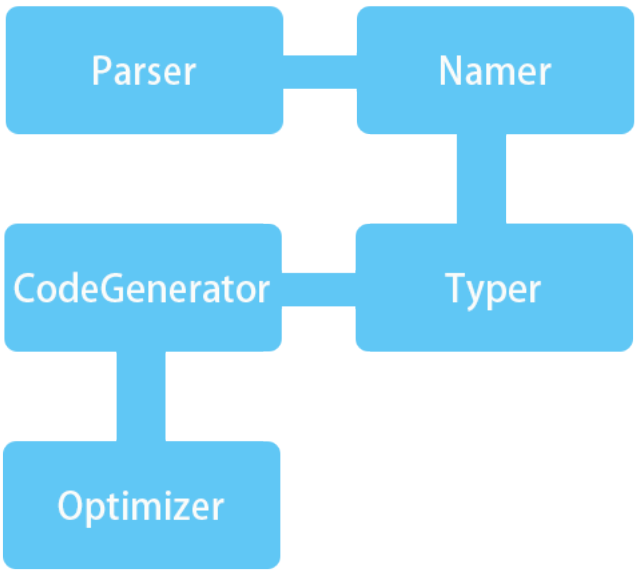
CS215 Compiler Course Project Report

Name: Shi, Yu **Student ID: 5130309117**

1 Introduction

In this project, I implement a simple compiler for SmallC, written in C++. First, let's have an overall view of my compiler.



I divide the compiler into 5 phases

1. Parser

   The parser is generated by yacc in Project 1. Here I link it to other phases, which is written by C++, by using "extern C" key word to include the C header file in my main.cpp.

2. Namer

   This phase attaches symbols to the parse tree. I call the parse tree generated by yacc parser *raw parse tree*. Because it has no symbol and type attached on parse tree nodes. In this phase, symbols are generated using the information from terminals and nonterminals, with the help of a symbol table. Unlike other compilers, my compiler only needs symbol table at this phase, because I attach them directly on the nodes. This is the way *Scala Compiler* deals with symbols. It can reduce a lot of time which would be spent in accessing symbol tables.

3. Typer

   This phase resolves types for every parse tree nodes. Most type checking are done in this phase. Type is also a necessary information for CodeGenerator phase. I implement almost all kinds of type checks, for assignments, arguments, array access, etc. So my compiler is quite type safe. Though we only have *int* in SmallC, I still consider structs as different types, because we can do assignment and other operations between structs, which requires to give them types to create a type safe system.

4. CodeGenerator

   This phase generates *llvm* IR, using names and types accumulated in previous phases. Separating code generator from other phases requires multiple passes through the parse tree, however, this approach makes more sophisticated error detection possible. Since *llvm* is not a machine code language, it is another kind of intermediate representation, we only needs to choose proper *llvm* IR instructions for different structures in the parse tree.

5. Optimizer

   The optimizer eliminates useless instructions in the whole module. It is a global optimizer, not limited by the basic blocks. It checks every instruction to see whether it is used by other instructions, either as operands, or as branch destination. Because in *llvm* IR, every instruction corresponds to an identifier, that is, every identifier in llvm IR can only be the written of at most one instruction. So we can do this optimization in a relatively easy way.

   Now, I will introduce these phases one by one.

2 Parser

The parser is automatically generated by yacc, according to our grammar. It generates a parse tree in the form of a C struct. It is quite inconvenient to implement my compiler in C, because I want more modularization and encapsulation. So I need to link the parser to my c++ main file, and compile them together with g++. Here's the definition of the raw tree struct in common.h file.

```
1 typedef struct tre {
2     char* symbol;
3     char* treeType;
4     int childNum;
5     struct tre** subTrees;
6     int productionId;
7     int lineNumber;
8 } tree;
```

This *tree* struct defines the structure of every parse tree node. The "symbol" field is not the real symbol we used during compiling, it is just the string

representation of contents of terminals and nonterminals. Tree type is the name of nonterminals, for example "PROGRAM". *childNum* represent the number of items in the production body. *subTrees* contains the children. For each nonterminal, I attach a *productionId* in the node, so that I can construct different parse tree nodes in different C++ classes according to the production. *lineNumber* is the line at which this parse tree node locates in the source file, it is used in later phases for providing accurate error report to the users.

3 Namer

Perhaps most compilers doesn't have this phase, I get this idea from Scala compiler. Symbols are important during the whole compiling process. In my compiler, every *identifier* will have a symbol. The symbols primarily contains the name and type of this identifier. It is needed by later phases. For example, when we do type check of a function, we need the information of its parameters and return types, these can be obtained from the symbol of this function. Also, in code generator, if we want to generate an instruction which allocates space for a local variable, we need the name of this variable, again it is in the symbol.

So, how to organize symbols becomes an important problem, we can either choose to use a symbol table and switch the symbol table as the context changes, or we can directly attach symbols to the parse tree. I choose the later one, which is much more convenient and efficient for my compiler. Though I need to create a temporary symbol table during this phase to detect redefinition and to tell local variables from global variables with the same name, it will never be used later, because we can directly get the symbol from the tree node.

Here's the definition of a symbol table, it is made up with symbol table frames, each frame is a collection of symbols in a statement block or a struct body. Frames are organized as linked list, every frame has an inner frame pointer, pointing to symbols in the outer layer of context. *EnterScope* and *ExitScope* are called whenever we enter or exit a statement block or a struct body. They exchange the *currFrame* with its inner frame to avoid symbol conflict in different context.

```
1  class SymbolTable {
2  private:
3      SymbolTableFrame* currFrame;
4  public:
5      void Add(Symbol* symbol);
6      Symbol* Get(std::string name);
7      SymbolTable();
8      void EnterScope();
9      void ExitScope();
10     void DumpCurrFrameTo(Symbol* structSymbol);
11     Symbol* RegisterSymbol(std::string name, bool isFunction,
12         bool isType, int dimension);
13     SymbolTableFrame* GetCurrFrame();
14 };
```

The structure of C++ parse tree nodes is also important. I choose to define a abstract class for every nonterminal. Every production has a class inherited from its nonterminal class. They are defined in ParseTree.h file. Productions of a nonterminal share the same declaration of some universal methods like *Typer()* and *Gen()*. Here's the interface of the nonterminal *EXTDEFS* and its

first production.

```
class EXTDEFS : public ParseTree {
public:
    static EXTDEFS* createEXTDEFS(tree* rawTree);
    virtual void Typer() = 0;
    virtual void Gen() = 0;
};

//EXTDEFS : EXTDEF EXTDEFS
class EXTDEF;
class EXTDEFS1 : public EXTDEFS {
private:
    EXTDEF* extdef;
    EXTDEFS* extdefs;
public:
    EXTDEFS1(tree* rawTree);
    void Typer();
    void Gen();
};
```

For each node, its class only contains the object of nonterminal of its children, that is, it doesn't know which production its child nonterminal uses. This is an idea of encapsulation. In this way, I can provide a universal interface for all the productions belongs to the same nonterminal. It makes the organization of parse tree much clearer, and tells each production class what it needs to implement.

4 Typer

After the *Namer* phase, now we have symbols for every identifier. We need to give these symbols types. Resolving type of symbols is a little complicated compared with attaching symbols to identifiers. Only types of identifiers need to be known during code generation. But to know the type of an identifier, we also need to know how this identifier is defined in the parse tree structure. Also, to do type check, we need to find out what is assigned to the identifier and what operations are done on them. So we need to type every node in the parse tree, not only nodes contains identifiers.

```
class Type {
public:
    std::string typeName;
    bool isStruct;
    bool isArray;
    virtual void Emit(ofstream& os) = 0;
    bool isInt;
protected:
    Type(std::string typeName, bool isStruct, bool isArray)
    : typeName(typeName), isStruct(isStruct), isArray(isArray) {}
    Type() : isStruct(false), isArray(false), isInt(false) {}
};

class Int : public Type {
public:
    Int() {isInt = true; typeName = "i32";};
    void Emit(ofstream& os);
};

class Array : public Type {
public:
    Type* elementType;
    int length;
    Array(Type* elementType, int length);
    void Emit(ofstream& os);
};
```

First I defined classes for different types. I show the interface of int type and

array type above. Every type requires a *Emit()* method, so that we can print the type in corresponding *llvm* IR form when generating code. Besides two type classes shown above, I also define type classes for functions and structures.

Types are inherited attributes. When defining an identifier, we need to pass the type to the lowest ID terminal level, which contains the symbol for identifier, so that we can set the type for this identifier. Here's the code for typing a function definition.

```
//EXTDEF : SPEC FUNC STMTBLOCK
void EXTDEF2::Typer() {
    func->Typer(spec->Typer());
    stmtblock->Typer();
}
```

As we can see, the *Typer()* function of a function definition calls the typer function of the type specification, the function and the statement block. Meanwhile, it passes the result of type specification to the function typer, so that in the function typer the return type can be available.

Most type checks are done in this phase, for example, when typing a binary operation expression, we can check whether both its operands are of int type. When typing a array access, check the type of the accessed identifier to see whether it is an array. Here's an example.

```
//EXP : ID ARRS
Type* EXP6::Typer() {
    Type* idType = name->Typer();
    if(!idType->isArray && !arrs->isEmpty()) {
        cout << "Line " << lineNumber << " Try to index non-array"
        << name->symbol->name << endl;
    }
    return arrs->Typer(idType);
}
```

5 CodeGenerator

Before printing code into .ll files, we need to do some optimization. So the IR instructions must be stored in date structures. For each IR instruction used in my compiler, I define a class.

```
class BinaryOperation : public Instruction {
private:
    const char* binaryOp;
    Instruction* leftOperand;
    Instruction* rightOperand;
public:
    BinaryOperation(const char* bop, Instruction* lop,
        Instruction* rop, Type* type);
    void Emit(ofstream& os);
    void PassForBranch();
    void CheckReferences();
};
```

As mentioned before, in *llvm* IR, every identifier (or register), represents a result of an instruction. I make full use of this feature. For each instruction, its operands are also instructions. That means the operands are results of these instructions. In this way the data dependences between instructions are maintained right in the instruction structure itself. No extra searches are needed to detect these dependences, making the optimization easier. *PassForBranch()* method above resolves the label issue. *CheckReferences()* are used in

optimization.

Instructions are organized in basic blocks. Every basic block will have either a branch or return instruction as the last instruction. Basic blocks are organized in functions. They all have their own classes.

Now I will introduce how I generate the intermediate code. Every production class has a *Gen()* method. It generates code for the language structure corresponding to this production. Here's an example of return statement.

```
1  //STMT : RETURN EXP SEMI
2  void STMT3::Gen(std::vector<BasicBlock*>* basicBlocks,
3          BasicBlock* next, BasicBlock* cont, BasicBlock* breakTo) {
4      if(basicBlocks->back()->IsClosed()) {
5          basicBlocks->push_back(new BasicBlock());
6      }
7      Instruction* returnExp = exp->Gen(basicBlocks);
8      basicBlocks->back()->Add(new Return(returnExp, returnExp->type));
9      basicBlocks->back()->Close();
10 }
```

Here *new Return()* creates the corresponding IR instruction.

The *Gen()* method needs to know which basic block it is generating code into, so it accepts the first pointer to a basic block. When generating statements and expressions (used as bool expression), we may need to know the basic block to jump to. These informations are provided by the following three basic block pointers in the parameters.

Each time after we generating a branch instruction, we need to close the current basic block. To close a block means that all intermediate code in this basic block has been generated. If more instructions are going to be generated, a new basic block should be create. That's why we have the first "if" in *Gen()* method.

Except for generating code, code generator also do some type check. To typecheck initialization and function call, types of every element in the initialization or every argument are required. To obtain these information is a little expensive in *Typer* phase. When generating intermediate code are these information are calculated, so I the type check for these structures in *CodeGenerator* phase.

6 Optimizer

I implement two kinds of optimizations

1. Eliminate basic blocks that can never be reached. For example, a statement after the return statement.

2. Eliminate instructions that are executed but has no contribution to the return value of a function, and also has no side effects. Here side effects including I/O, memory allocation and memory store.

The first optimization is done when emitting the code into .ll file. During this process, I check whether there's any branch instruction can reach a basic block. If a basic block cannot be reached by any branch instruction and it is not the first entry basic block of a function, there's no need to emit it.

```
void BasicBlock::Emit(ofstream& os) {
    if((instructions.size() > 0 && referenceCounter > 0)
        || name == "entry") {
        if(!lastClosedWithBranch) {
            UncondBranch* branch = new UncondBranch(this);
            branch->Emit(os);
        }
        if(name != "entry")
            os << name + ":" << endl;
        for(int i = 0; i < instructions.size(); ++i) {
            instructions[i]->Emit(os);
        }
        lastClosedWithBranch = instructions.back()->isBranch;
    }
}
```

In the code, *referenceCounter* is the number of branch instructions that use this basic block as destination. *lastClosedWithBranch* is a flag belong to BasicBlock class, indicating whether the last emitted basic block ends with a branch. If not, that means the control flow after the last basic block will fall on the current one, however in *llvm* every basic block must end with a branch instruction or a return. So in this case an unconditional branch is generated to jump from last basic block to current one.

To do the second optimization, we do search on the instructions in a function body. As introduced before, the data structure for instructions naturally contains the dependences among them. For every instruction, it is easy to know the results of which instructions it is using. I call them *usage instructions* of this instruction. For example, in a BinaryOperation instruction

```
class BinaryOperation : public Instruction {
private:
    const char* binaryOp;
    Instruction* leftOperand;
    Instruction* rightOperand;
public:
    BinaryOperation(const char* bop, Instruction* lop,
        Instruction* rop, Type* type);
    void Emit(ofstream& os);
    void PassForBranch();
    void CheckReferences();
};
```

Both instruction pointers *leftOperand* and *rightOperand* are usage instructions of this BinaryOperation instruction.

First I mark the instructions with side effects, like store, and return instructions, function call instructions as useful. Return instructions gives the return value of whole function, so it must be useful. Function calls are marked as useful because we don't know whether it has any side effects.

Define a *CheckReferences()* method for every instruction, this method will set the instruction as useful and continue to check the usage instructions. In this way, not only dependences in a basic block, but dependences in the whole function body are checked.

```
void BinaryOperation::CheckReferences() {
    live = true;
    leftOperand->CheckReferences();
    rightOperand->CheckReferences();
}
```

7 Conclusion

My compiler is equipped with following features

1. Modularization and encapsulation. Everything is object-oriented, easy for expansion.
2. Detailed error detection. Part of the error detection are done in type check, others, such as undefined symbol and redefinition are done when attaching symbols to parse tree.
3. Optimization beyond basic blocks.

I provide some test cases for features above in the folder.

Thanks for reading!