Our Pascal Compiler

Yangyi Huang
huangyangyi@zju.edu.cn

Yuhan Wang
wang_yuhan@zju.edu.cn

Shibiao Jiang
3170102587@zju.edu.cn

Zhejiang University

Date: May 31, 2020

Abstract

In our Compiling Principle course during Spring & Summer Semester 2020, we implemente a simple pascal compiler and name it as *Our-Pascal-Compiler*. We use lex & yacc to complete the lexical analysis and choose llvm as our backend. Our development language for this project is **C++**, and the classical Visitor Design Pattern is applied. Besides the fundamental Pascal grammar, We retain some fancy features like *nested functions* or *nested record types*, which make our compiler more useful and funny. This is the report & documentation for our project, and you can find the source code on our Github.

Keywords: Our Pascal Compiler YACC LLVM Visitor Pattern Nested Function

1 Introduction

In this part, we first introduce Pascal briefly, and then the grammar and function our compiler supports.

1.1 Why Pascal?

From `Wikipedia`, Pascal is an imperative and procedural programming language, designed by *Niklaus Wirth* as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. It is named in honour of the French mathematician, philosopher and physicist *Blaise Pascal*.

Pascal has strong typing on all objects, which means that one type of data cannot be converted or interpreted as another without explicit conversions. Unlike most languages in the C-family, Pascal allows nested procedure definitions to any level of depth, and also allows most kinds of definitions and declarations inside subroutines (procedures and functions).

Due to the its rigorous structure and powerful functions, Pascal is widely used in program language teaching. All three of us have learned about Pascal Programming in middle school, so we choose this language as the source language in our compiler.

1.2 Highlights of Our Project

As a programming language, Pascal grammar can be represented as Context Free Grammar (**CFG**). We first make use of Lex and Yacc, which will complete the lexical analysis by scanning the whole program and use the technology of **LALR(1)** to execute the Syntax Analysis. Then we construct the Absract Syntax Tree (**AST**) nodes for each non-terminal in CFG. After that, we do Semantic Analysis to check semantic errors and finally generate Intermediate Code with the help of LLVM. We can also operate LLVM to build the Target Code. And here are the highlights of our project.

1. **The Coverage of Most Pascal Language Basic Features** Unlike the "toy languages" which are often used to write simple compilers, Pascal itself is a powerful and widely used structural programming language. The language features include variable declaration, variable scope management, nested function/procedure declaration and usage, which have a certain degree of complexity. Based on the ISO7185[1] Pascal language standard, we have implemented most but not limited to Pascal basic language features:
   (a) Support most data types in Pascal, including array, string and record types
   (b) Support most statements in Pascal, including if, while, repeat, for, break and exit, etc.
   (c) Support assignment, expression calculation, etc.
   (d) Support some of the important built-in functions/procedure in Pascal
   (e) Support function and procedure features in Pascal, including nested declaration
   (f) Correct implementations of local variable scope and value/ variable parameters passing
2. **Visitor Pattern: A Wise Practice of Building Compilers** Instead of the naive pattern of writing methods directly related to various actions in the AST nodes, we use the visitor pattern as a design pattern for compiler construction. Its support for double-dispatch makes it a wise choice for traversing AST operations.
3. **Using LLVM[2]: A Powerful Compiler Backend** We choose the LLVM C++ API for code generation when generating intermediate code and assembly code. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies, the mature architecture and excellent modular design of its C/C++ library enable us to write the compiler back-end more clearly and efficiently.
4. **The Mechanism of Error Detection and Report**: To make our compiler more useful, we store all the errors and execute error recovery during syntax analysis process and semantic analysis process. After compiling, all the errors will be reported together, or the IR code will be generated if no error finds.

[1]ISO 7185:1983 Programming languages PASCAL https://www.iso.org/standard/13801.html

[2]The LLVM Compiler Infrastructure llvm

2 Lexical Analysis

2.1 Tokens Recognizing

We use the term **token** to represent each meaningful word composed of consecutive characters. Tokens are the basic units in lexical analysis, and they're usually separated by space or tab. In `lexer/pascal.l`, we construct several rules to detect different types of tokens, such like *keywords*, *identifiers* and *literals*.

Besides, we also use lex to locate each token. The positions of tokens will be really useful in the next step (i.e. Semantic Analysis), because we can also provide the programmer with specific error location when some error occurs. The technique of token locations is as follow.

```
#define YY_USER_ACTION \
    yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn + yyleng - 1; \
    yycolumn += yyleng;
```

2.2 Keywords and Symbols

From *Free Pascal Reference guide 3.0*, we select all Turbo Pascal reserved words as our keywords.

```
  and, array, asm, begin, break, case, const, constructor, destructor, div, do,
  downto, else, end, exit, file, for, function, goto, if, implementation, in,
  inherited, inline, interface, label, mod, nil, not, object, of, operator, or,
  packed, procedure, program, record, reintroduce, repeat, self, set, shl, shr,
  then, to, type, unit, until, uses, var, while, with, xor
```

And there are five primitive types in Pascal, including:

```
  integer, real, boolean(or bool), character(or char), string
```

Symbols should be put into consideration, too. { } are used to enclose comments, := is used to represent assignments, and the rest are all involved in expressions.

```
  +  -  *  /  =  <  >  <>  <=  >= := [  ]  .  ,  (  )  : ;  ..  {  }
```

2.3 Literals and Identifiers

Literals are notations for representing the fixed value in source code, so we must distinguish them in lexical analysis one by one. **Note:** No matter what type of literal we find, we return the same type `char*` and won't convert it until the next step.

- *integer* is the simplest type to recognize: It always consists of several numbers. For convenience, we **do not** handle with the negative symbol during the lexical analysis process (i.e. Consider all the literal positive, and regard *the negative symbol* as *the minus operator*).

```
LITERAL_INT [0-9]+
{LITERAL_INT} {
    yylval = strdup(yytext);
    RETURN_TOKEN(LITERAL_INT)
}
```

- *real* type is more complex. We should take the dot and scientific notation into consideration.

```
SIGN "+"|"-"
LITERAL_FLOAT ([0-9]+\.[0-9]+)|([0-9]+\.[0-9]+e{SIGN}?[0-9]+)|([0-9]+e{
    SIGN}?[0-9]+)
```

- As for *char* and *string* type, the existence of escape characters will make the interpretation difficult.

```
LITERAL_CHAR \'.\'
LITERAL_ESC_CHAR '\'#\'
LITERAL_STR \'([^']|{LITERAL_ESC_CHAR})*\'
```

3 Syntax Analysis

3.1 The Design of AST

Using the idea of object-oriented programming, we design an abstract class `ASTNode` to represent each node in Abstract Syntax Tree, as well as many other classes inherited from `ASTNode`. There are **5** types of `ASTNode` from the perspective of coarse granularity division.

`AST/ast_type.cpp` records each class served for type definition.

- **ASTType**: Stores the keyword related to the type. (i.e. `integer`, `char`...)
- **ASTTypeDecl**: There are three patterns of type declaration: *simple-type*, *array-type*, *record-type*. And this is an abstract class to operate all of them representing *the universal type declaration.*
- **ASTSimpleTypeDecl**: This class represents all **pure** type declarations which do not have nested structure. Besides the system type, notice that we also have *Custom type* (Define by the programmer), *range type* (use integer, char or enumerated type to describe a range) and *enumerated types* (Whose legal values consist of a fixed set of constants).

```
simple_type_decl: SYS_TYPE  |  IDENTIFIER  |  (namelist)
|  const_value .. const_value  |  IDENTIFIER  ..  IDENTIFIER
```

- **ASTArrayTypeDecl**: This class represents the array declaration. Recall that the type of array range is `ASTSimpleTypeDecl` and the type of array elements is `ASTTypeDecl`.

- **ASTFieldDecl**: This class represents the record declaration. In a nutshell, it consists of several fields (identifiers) and their corresponding type (`ASTTypeDecl`). We design some extra classes like `ASTTypeDefinition`, `ASTTypeDeclList` and `ASTTypePart` to achieve this target.

`AST/ast_expr.cpp` records each class served for expression calculation. Each complex expression can be split hierarchically to the following (relatively) simple expression.

- **ASTExpressionList**: Stores a list of expressions (`ASTExpr`).
- **ASTExpr**: An abstract class to describe and operate each type of expression. We can use this class to describe any expression entity. Following classes are all inherited from it.
- **ASTUnaryExpr**: represents the unary expression.
- **ASTBinaryExpr**: represents the binary expression.
- **ASTPropExpr**: represents the expression of calling the field of one record type.
- **ASTConstValueExpr**: represents the expression made up by constant value.
- **ASTFuncCall**: represents the expression of function calls.
- **ASTArrayExpr**: represents the expression of array or string calls.

`AST/ast_value.cpp` is designed to to deal with values, such like *constant value*, *var list* and *const list*.

- **ASTConstValue**: This class stores **literals** from lexical analysis. Each literal will be passed to the constructor as an argument along with its type. Here is the code for building `ASTConstValue`.

```
// AST/ast_value.cpp
class ASTConstValue : public ASTNode {
  public: enum class ValueType {INTEGER, FLOAT, CHAR, STRING, BOOL};
}
// parser/pascal.y
const_value:
    LITERAL_INT {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::INTEGER);
        SET_LOCATION($$);
    }
    | LITERAL_FLOAT {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::FLOAT);
        SET_LOCATION($$);
    }
    | LITERAL_CHAR {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::CHAR);
        SET_LOCATION($$);
    }
    | LITERAL_STR {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::STRING);
        SET_LOCATION($$);
    }
    | LITERAL_FALSE {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::BOOL);
        SET_LOCATION($$);
    }
    | LITERAL_TRUE {
        $$ = new ASTConstValue($1, ASTConstValue::ValueType::BOOL);
        SET_LOCATION($$);
    }
;
```

- **ASTConstExpr**: This class consists of an identifier and a `ASTExpr` class to record a constant definition.
- **ASTConstExprList**: This class stores a collection of `ASTConstExpr`.
- **ASTConstPart**: This class represents a completed part of constant definitions.
- **ASTVarDecl**: This class is used to record a variable definition, so it consists of a identifier and a `ASTTypeDecl` class, and the latter could represent any type(simple or nested) that can be defined.
- **ASTVarDeclList**: This class stores a collection of `ASTVarDecl`.
- **ASTVarPart**: This class represents a completed part of variable definitions.

`AST/ast_stmt.cpp` is used to define statements. Note that our compiler supports *goto statement* so it's necessary to maintain the labels of statements. (And of course some statements don't have to be labeled.)

- **ASTStmtList**: It contains a list of statements, which are the main parts of a program.
- **ASTStmt**: This class can represent any kind of statement. Note that it contains a non-labeled statement and an identifier which indicates the **label** of that statement.
- **ASTNonLabelStmt**: This is the base class for the following classes.
- **ASTAssignStmt**: This class contains two `ASTExpr` representing a assignment statement. We know the left expression should be a *left value*, and we will check it in the next (semantic analysis) process.

```
assign_stmt:
    IDENTIFIER SYM_ASSIGN expression{
        ASTExpr *expr = new ASTIDExpr($1);
        $$ = new ASTAssignStmt(expr, $3);
        SET_LOCATION($$);
    }
    | IDENTIFIER SYM_LBRAC expression SYM_RBRAC SYM_ASSIGN expression{
        ASTExpr *expr = new ASTArrayExpr($1, $3);
        $$ = new ASTAssignStmt(expr, $6);
        SET_LOCATION($$);
    }
    | IDENTIFIER SYM_PERIOD IDENTIFIER SYM_ASSIGN expression{
        ASTExpr *expr = new ASTPropExpr($1, $3);
        $$ = new ASTAssignStmt(expr, $5);
        SET_LOCATION($$);
    }
;
```

- **ASTProcStmt**: This class indicates a variable or a function call, which are regarded as valid statements.
- **ASTForStmt**: represents *for statement*, and contains an enumerated class {TO, DOWNTO}.

```
for\_stmt:
    KWD_FOR IDENTIFIER SYM_ASSIGN expression direction expression KWD_DO stmt{
        $$ = new ASTForStmt($2, $4, $5, $6, $8);
        SET_LOCATION($$);
    }
;
direction:
    KWD_TO {
        $$ = ASTForStmt::ForDir::TO;
    }
    | KWD_DOWNTO {
        $$ = ASTForStmt::ForDir::DOWNTO;
    }
;
```

- **ASTRepeatStmt**: represents *repeat statement*, contains an expression(condition) and `ASTStmtList`.
- **ASTWhileStmt**: represents *while statement*.
- **ASTElseClause**: represents *else statement*.
- **ASTIfStmt**: represents *if statement*. `ASTElseClause` belongs to it and can be *nullptr*.

```
if_stmt:
    KWD_IF expression KWD_THEN stmt else_clause {
        $$ = new ASTIfStmt($2, $4, $5);
        SET_LOCATION($$);
    }
;
else_clause:
    KWD_ELSE stmt {
        $$ = new ASTElseClause($2);
        SET_LOCATION($$);
    }
    | %empty {
        $$ = nullptr;
    }
;
```

- **ASTCaseStmt**: This class represents the *case statement*. It consists of an `ASTExpr` indicating the expression and a list of `ASTExpr-ASTStmt` pairs.
- **ASTGotoStmt**: represents the *goto statement*, only contains an identifier indicating the label.
- **ASTBreakStmt**: represents the *break statement*.
- **ASTExitStmt**: represents the *exit statement*. Note the form of exit in function or procedure is different: One is just a literal *"exit"* but the other will nest an expression.

```
// ast/ASTStmt.h
    ASTExitStmt(ASTExpr *expr = nullptr);
// parser/pascal.y
exit_stmt:
    KWD_EXIT SYM_LPAREN expression SYM_RPAREN {
        $$ = new ASTExitStmt($3);
        SET_LOCATION($$);
    } | KWD_EXIT {
        $$ = new ASTExitStmt();
        SET_LOCATION($$);
    }
;
```

`AST/ast_prog.cpp` defines high-level AST classes for a Pascal program. These classes organize the whole program through a rigorous structure. We can learn about it from the following picture.

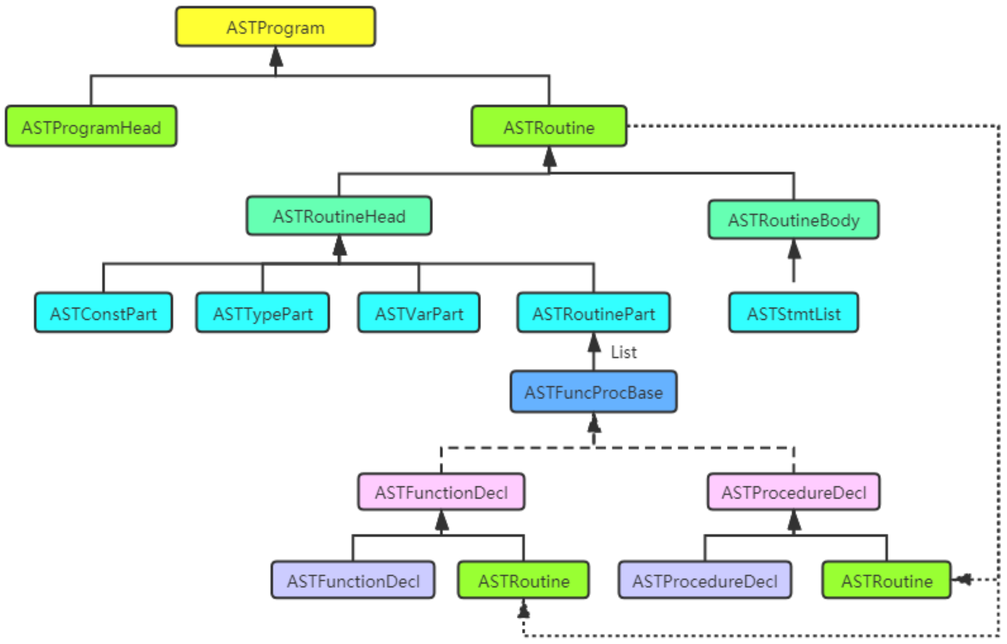


Figure 1: The class structure in `AST/ast_prog.cpp`

3.2 Visualization of AST nodes

In order to test the correctness of lexical analysis and syntax analysis, as well as view the program structure more intuitively, we implement the function of visualizing AST nodes.

To make full use of Graphviz[3], we create a new class called `GraphGenerator` to directly generate the Graphviz source code, and this class is also written in **Visitor Pattern**, where we maintain a data structure which contains nodes and edges in Graphviz. Here are the private members in `GraphGenerator`.

```
class GraphGenerator : public Visitor {
private:
    int id_cnt_ = 0;
    std::stack<int> stk_;
    std::vector<std::string> nodes_;
    std::vector<std::string> edges_;
}
```

`std::stack<int> stk_` is a stack that maintains the hierarchy of AST nodes. More specifically, we will push the current node into the `stk` if we go deeper in the DFS tree. When constructing an edge for Graphviz, we just link the current node to the node in the top of stack. Some functions are as follows.

```
void GraphGenerator::AddNode(std::string label, int line, int col) {
    int id = id_cnt_++;
    std::stringstream ostr("");
    ostr << "node_" << id << "[";
    ostr << "label=\"" << label << "\\n";
    ostr << "[" << line << ",␣" << col << "]";
    ostr << "\"];";
    std::string node_str = ostr.str();
    nodes_.push_back(node_str);
    std::stringstream ostr1("");
    ostr1.clear();
    if (id != 0) {
        ostr1 << "node_" << stk_.top() << "->"
              << "node_" << id << ";";
        std::string edge_str = ostr1.str();
        edges_.push_back(edge_str);
    }
    stk_.push(id);
}

void GraphGenerator::Pop() { stk_.pop(); }
```

[3] Graphviz is a open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks.

And finally we just call GraphGenerator::Save to save the Graphviz source code.

```
void GraphGenerator::Save(std::string path) {
    std::ofstream f(path);
    f << "digraph␣g␣{" << std::endl;
    for (auto e : edges_) f << "\t" << e << std::endl;
    for (auto n : nodes_) f << "\t" << n << std::endl;
    f << "}";
    f.close();
}
```

Build a pure virtual function Accept(GraphGenerator *graph)= 0 in ASTNode and inherit it in other classes. Create one instance GraphGenerator *graph and pass it into all AST nodes. By overloading virtual "Visit" functions, we can define different operations to "draw the pictures".

```
// viz/graph_generator.cpp
shared_ptr<VisitorResult> GraphGenerator::VisitASTRoutineHead(ASTRoutineHead *node
    ) {
    this->AddNode("routine_head", node->line(), node->col());
    if (node->getConstPart() != nullptr) node->getConstPart()->Accept(this);
    if (node->getTypePart() != nullptr) node->getTypePart()->Accept(this);
    if (node->getVarPart() != nullptr) node->getVarPart()->Accept(this);
    if (node->getRoutinePart() != nullptr) node->getRoutinePart()->Accept(this);
    this->Pop();
    return nullptr;
}
shared_ptr<VisitorResult> GraphGenerator::VisitASTBinaryExpr(ASTBinaryExpr *node)
    {
    string op_name = node->getOpName(node->getOp());
    this->AddNode(string("binary_op_expr:␣") + op_name, node->line(),
                  node->col());
    node->getLExpr()->Accept(this);
    node->getRExpr()->Accept(this);
    this->Pop();
    return nullptr;
}
```

Limited by space, we just show a simple Graphviz result: The a+b Problem.

```
program aplusb;
var a,b:integer;
begin
  read(a, b);
  writeln(a+b);
end.
```

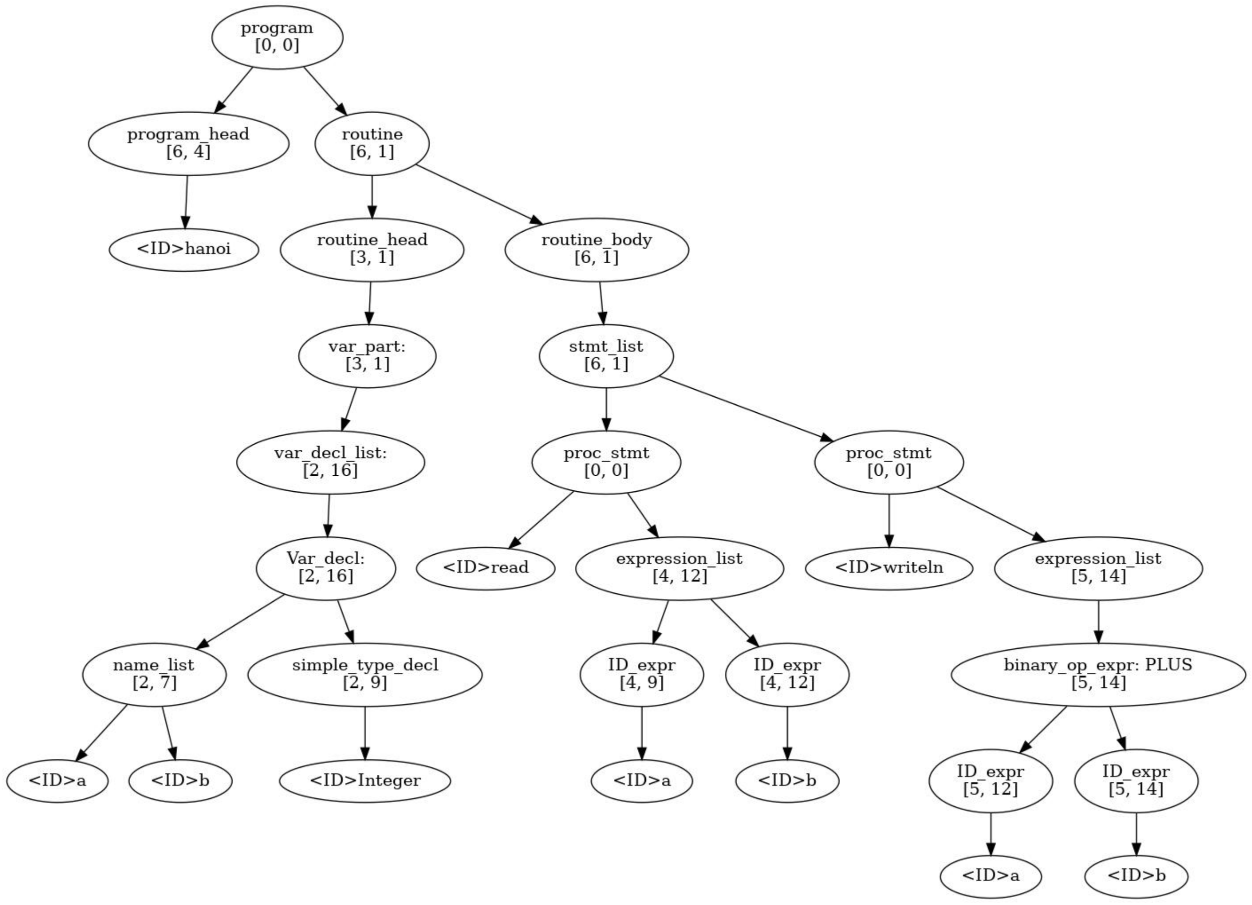


Figure 2: The abstract syntax tree generated by Graphviz.

4 Semantic Analysis

In the semantic analysis phase, we need to maintain some data structures such like *symbol table* and *function table* to do expression type check and find the potential semantic errors.

We will introduce a variety of data structures and semantic environment structures designed for this purpose, combining with our codes and some examples.

Note that semantic analysis is closely related to code generation. To reduce code redundancy and make the structure of the whole project more compact and easy to read, we put codes for these two phases **together**.

4.1 Our-Pascal-Type

As everyone who once built a compiler himself using LLVM knows, the necessity and approach that we design the **types** is highly related to how we maintain the symbol table. If you use the LLVM supported symbol table, you can use its API directly but you cannot extend its given data types. If you build your own symbol table, you can have more extensive data types but will suffer from the burden of coding a data structure and **even more tough problem** that will be introduced in **Chapter 5.3**.

LLVM is an amazing helper when you implement a compiler for C language, which is LLVM initially designed for. However, when it comes to Pascal, a language that has much difference with C, we decided to design our own *class PascalType* as LLVM does not support all Pascal types robustly. We will introduce our own PascalType for semantics representation here, and explain how to transform our type to authentic *llvm::Type* in **Chapter 5.2**.

First of all, we need to construct a base class, we name it `PascalType`. The pointer of this class will be stored in the symbol table, thus it should include the information of which kind of type it belongs to. The types are clearly explained in the `enum class TypeGroup`.

```
class PascalType {
public:
    enum class TypeGroup {
        BUILT_IN, ENUM, SUBRANGE, ARRAY, STR, RECORD
    };
    TypeGroup tg;
};
```

Then we describe each subclass related to the type shown in *TypeGroup*. One thing to be mentioned is that we are describing a **Type**, not a **Value**, so we only record enough imformation to represent it, but do not need to leave space for variable storage.

First is `BuiltinType`, we use another `enum class BasicTypes` in this sub-class to represent it as int, real, char, boolean or void, where void is used for function and procedure declaration only. Then we define a const pointer for each type for easy representation when using our semantics type, since they are frequently used.

```
class BuiltinType : public PascalType {
public:
    enum class BasicTypes {
        INT, REAL, CHAR, BOOLEAN, VOID
    };
    BasicTypes type;

    BuiltinType(BasicTypes type);
};
const BuiltinType INT_TYPE_INST(BuiltinType::BasicTypes::INT);
const BuiltinType REAL_TYPE_INST(BuiltinType::BasicTypes::REAL);
const BuiltinType CHAR_TYPE_INST(BuiltinType::BasicTypes::CHAR);
const BuiltinType BOOLEAN_TYPE_INST(BuiltinType::BasicTypes::BOOLEAN);
const BuiltinType VOID_TYPE_INST(BuiltinType::BasicTypes::VOID);
PascalType *const INT_TYPE = (PascalType *) (&INT_TYPE_INST);
PascalType *const REAL_TYPE = (PascalType *) (&REAL_TYPE_INST);
PascalType *const CHAR_TYPE = (PascalType *) (&CHAR_TYPE_INST);
PascalType *const BOOLEAN_TYPE = (PascalType *) (&BOOLEAN_TYPE_INST);
PascalType *const VOID_TYPE = (PascalType *) (&VOID_TYPE_INST);
```

Then it comes to user-designed types, such as array and record. We only show the class members in the following code. For one thing, our array type is implemented in a nested way, it means that the `element_type` of an array can be another array. When we need to transform our type to an `llvm::Type`, we implement this in a recursive way. For another, the implementation of `EnumType` is a little tricky. In fact, we can consider the definition of an Enum as we firstly name some int constants, then each variable that has an Enum type is an integer actually. Our design for Enum directly reflects this idea.

```
class SubRangeType : public PascalType {
    int low;
    int high;
};

class ArrayType : public PascalType {
    std::pair<int, int> range; // pair(low, high)
    PascalType *element_type;
};

class StrType : public PascalType {
    int dim;
};

class RecordType : public PascalType {
    int size;
    std::vector<std::string> name_vec;
    std::vector<PascalType *> type_vec;
};

class EnumType : public PascalType {
    std::vector<std::string> names_;
};

EnumType::EnumType(std::vector<std::string> names, Generator *g) :
        names_(names), PascalType(PascalType::TypeGroup::ENUM) {
    for (int i = 0; i < names.size(); i++) {
        if (g->named_constants.find(names[i]) != g->named_constants.end()) {
            // multiple constant
            continue;
        } else {
            g->named_constants[names[i]] = llvm::ConstantInt::get(
                    llvm::Type::getInt32Ty(g->context), i, true);
        }
    }
}
```

4.2 Symbol Table and Global Stack

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. It usually exists in memory during the translation process only.

So we create a class called `CodeBlock` to store all the information we need during semantic analysis phase and code generation phase. (Some data structures will be introduced later.)

```
class CodeBlock {
    std::map<std::string, llvm::Value*> named_values;
    std::map<std::string, OurType::PascalType*> named_types;
    std::map<std::string, llvm::Function*> named_functions;
    std::map<std::string, FuncSign*> named_funcsigns;
    std::map<int, llvm::BasicBlock *> labels;
    std::vector<llvm::BasicBlock *> loop_breaks;
    std::string block_name;
    bool is_function;
}
```

`named_types` is the structure that maps from *variable_name* to *type*. To support the record type, `named_types` also stores the mapping from *type_name* to *type*. (Note that *record type* and *variable* can not share the same name, so it's all right.) This structure is really **essential** during the semantic analysis process, because we will check the semantic type according to it, and update it in real time.

In order to maintain variables and functions in different contexts, we then introduce a technology called Global Stack. This structure is the basis of function nesting.

```
std::vector<CodeBlock*> block_stack;
```

`block_stack` is a global stack which stores all the contexts up to now. When we enter a new environment (i.e. enter a function body), a new `CodeBlock*` will be created and pushed into `block_stack`. And when we want to call some variables, just enumerate from the top of the stack[4] to find the first `CodeBlock*` containing that variable name.[5]. When we leave this context, the corresponding `CodeBlock*` will be popped.

[4]To achieve this, we use `std::vector` instead of `std::stack`.

[5]It is just a general description. To deal with the var/val problem in Pascal, we slightly adjust this process. Find more details in **5.4**.

4.3 Maintenance of Function Information

Function maintenance is a complex part in our compiler. To better operate and store functions, we design the following structure for each function header.

```
class FuncSign{
public:
    FuncSign(int n_local, std::vector<std::string> name_list, std::vector<OurType
        ::PascalType*> type_list, std::vector<bool> is_var, OurType::PascalType*
        return_type = nullptr)
        :name_list_(name_list), type_list_(type_list), is_var_(is_var),
            return_type_(return_type), n_local_variables(n_local) {
            if (return_type == nullptr)
                return_type_ = OurType::VOID_TYPE;
        }
private:
    int n_local_variables; // used for parameter passing
    std::vector<OurType::PascalType*> type_list_;
    std::vector<std::string> name_list_;
    std::vector<bool> is_var_;
    OurType::PascalType* return_type_;
};
```

For each function declaration, we should store all the information for its parameters. (So we can check further when somewhere calls it.)

- **name_list** denotes the name of the parameters. When somewhere call this function, the value of parameters brought in will be bound to these names.
- **type_list_** denotes the type of the parameters. We can execute type check according to it.
- **is_var_** signs whether the parameter is variable. Passing by value and passing by reference are both OK, and we will do different operations to implement both of them (Find more details in **5.4**).
- **is_function** indicates whether it is a function or a procedure. We code them together in the base class of them, and this field will help determine the type so that we can change the pointer successfully.

4.4 Semantic Errors

4.4.1 Expression Type Check and Type Conversion

Pascal is a type-safe programming language, so it's important to check the expression type. And some necessary type conversions should be supported, like the calculation between *integer* and *real*.

First we show the type check for arithmetic operation. Pascal does not support operator overloading, so the variable type involved in the operation must be *int* or *real* type (Type Check). And if one of the member is *real*, the final result is also *real* (Type Conversion).

```
bool check_arith(PascalType *l, PascalType *r, PascalType *&ret){
    if (!l->isSimple() || !r->isSimple()) return false;
    if (isEqual(l, BOOLEAN_TYPE) || isEqual(r, BOOLEAN_TYPE)) return false;
    if (isEqual(l, CHAR_TYPE) || isEqual(r, CHAR_TYPE)) return false;
    //boolean type and char type can not take part in the arithmetic
    ret = l;
    if (isEqual(l, REAL_TYPE)) ret = l;
    if (isEqual(r, REAL_TYPE)) ret = r;
    return true;
}
```

Comparison operation is sightly different from arithmetic operation. Same types between `char` and `boolean` are also supported. Other type checks have the similar logic so we just omit.

```
bool check_cmp(PascalType *l, PascalType *r, PascalType *&ret) {
    if (!l->isSimple() || !r->isSimple()) return false;
    ret = l;
    if (isEqual(l, BOOLEAN_TYPE) && isEqual(r, BOOLEAN_TYPE)) return true;
    if (isEqual(l, BOOLEAN_TYPE) || isEqual(r, BOOLEAN_TYPE)) return false;
    if (isEqual(l, CHAR_TYPE) && isEqual(r, CHAR_TYPE)) return true;
    if (isEqual(l, CHAR_TYPE) || isEqual(r, CHAR_TYPE)) return false;
    if (isEqual(l, REAL_TYPE)) ret = l;
    if (isEqual(r, REAL_TYPE)) ret = r;;
    return true;
}
```

4.4.2 Assignment Statement Check

Pascal uses the idea of l-values and r-values, deriving from the typical mode of evaluation on the left and right hand side of an assignment statement. An l-value refers to an object that persists beyond a single expression. An r-value is a temporary value that does not persist beyond the expression that uses it.

During the syntax analysis, We don't distinguish the left value from the right value, and directly use the `ASTEexpr` to represent. So now we should recognize them and reject those wrong assignments.Only three types of expressions can be l-value in our pascal compiler: *identifiers (variables)*, *indexes of arrays* and *fields of records*.

In fact, we design a more flexible way to do the assignment statement check. The return value of each `ASTEexpr` contains not only the type and the value in llvm, but also **the memory** in llvm.[6] Expressions that cannot be left will not return the corresponding memory, so we can make a judgement by this.

[6]LLVM will be introduced thoroughly in Section **5**.

```
std::shared_ptr<VisitorResult> VisitASTAssignStmt(ASTAssignStmt *node) {
    ... // left and right are two expressions involved in assignment.
    if (left->getMem() == nullptr)
        return RecordErrorMessage("Invalid␣left␣value.", ...);
    ...
}
```

For example, `VisitASTArrayExpr` returns a value with memory because *indexes of arrays* are valid l-value (and the memory is equal to the start position of array plus the offset) , while `VisitASTBinaryExpr` returns a value without memory.

4.5 Error Recovery

Error Recovery is an important part in our compiler system. Users always want the compiler to point out as many errors as possible at a time, instead of stopping the parsing as soon as an error occurs.

In the lexical analysis stage, lex will report the error when it encounters the wrong token. So we focus on the semantic error.

First, prepare the following two structures globally to record the error messages.

```
    std::vector<std::string> error_message;
    std::vector<std::pair<int, int> > error_position;
```

Image how we should do when meeting an error. The place where the error occurs will pass the messages and locations to an error-handling mechanism. After that, the system needs to be *restored* as much as possible. **Our convention**: If an error is encountered when processing an AST node, the original return value (pointer type) will be assigned to a null pointer.

```
std::shared_ptr<VisitorResult> Generator::RecordErrorMessage(std::string
    cur_error_message, std::pair<int, int> location){
    error_message.push_back(cur_error_message);
    error_position.push_back(location);
    return nullptr;
}
```

The above code is the key function in error recovery. When somewhere goes wrong, we just write `return RecordMessage("...", get_location_pairs())`, then the error messages will be recorded and the null pointer will be passed to the previous layer.

Take some instances about binary operations and the access of record type.

```
// VisitASTBinaryExpr
if (nowOp == Op(AND) || nowOp == Op(OR)){
    if (!check_logic(l->getType(), r->getType()))
```

```
        return RecordErrorMessage("Both␣sides␣of␣the␣binary␣logic␣expression␣need␣
            to␣be␣BOOLEAN␣type.", node->get_location_pairs());
}
// VisitASTPropExpr
    std::string id = node->getId();
    std::string propid = node->getPropId();
    auto record_type_ = val->getType();
    if (!record_type_->isRecordTy())
        return RecordErrorMessage("Non-record␣type␣can␣not␣use␣'.'.", node->
            get_location_pairs());
    auto name_vec = record_type->name_vec;
    int bias = -1;
    for (int i = 0; i < name_vec.size(); i++)
        if (name_vec[i] == propid){
            bias = i;
            break;
        }
    if (bias == -1)
        return RecordErrorMessage(id + "␣do␣not␣have␣property␣" + propid, node->
            get_location_pairs());
```

Another important thing during error handling is **Duplicate Avoidance**. When an error occurs on an AST node, every layer above it will be abnormal but not every layer needs to report this error.

Reconsider the example of assignment. if the left expression or the right expression meets a problem, we **do not need to report the error in this layer** and just `return nullptr`.

```
std::shared_ptr<VisitorResult> VisitASTAssignStmt(ASTAssignStmt *node) {
    ... // left and right are two expressions involved in assignment.
    if (left == nullptr || right == nullptr)
        return nullptr;  //do not need to report the error in this layer
    if (left->getMem() == nullptr)
        return RecordErrorMessage("Invalid␣l-value.", node->get_location_pairs());
    if (!genAssign(...))
        return RecordErrorMessage("Can␣not␣do␣assignment␣between␣different␣types."
            , node->get_location_pairs());
    ...
}
```

And there is a more intuitive example. For the whole programm, if `VarPart` encounters the problem, other parts won't be influenced.

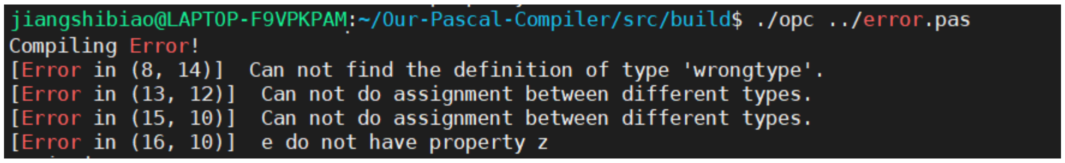
```
std::shared_ptr<VisitorResult> VisitASTRoutineHead(ASTRoutineHead *node) {
    if (node->getConstPart()) node->getConstPart()->Accept(this);
    if (node->getTypePart()) node->getTypePart()->Accept(this);
    if (node->getVarPart()) node->getVarPart()->Accept(this);
    if (node->getRoutinePart()) node->getRoutinePart()->Accept(this);
    ...
}
```

After every error handling is done, generate the target code or print the compiling error.

```
if (gen->hasError()){
    std::cout << "Compiling␣Error!" << std::endl;
    gen->printError();
}
else gen->Save(output_ll_fname);
```

4.6 Test for Error Report

```
program error;
type t = record
  x: real;
  y: integer;
end;
var
  a,b: real;
  c: wrongtype;
  d: integer;
  e: t;
begin
  read(a, b);
  d := a + b;
  writeln(d);
  d := e.x;
  d := e.z;
end.
```



```
jiangshibiao@LAPTOP-F9VPKPAM:~/Our-Pascal-Compiler/src/build$ ./opc ../error.pas
Compiling Error!
[Error in (8, 14)]  Can not find the definition of type 'wrongtype'.
[Error in (13, 12)]  Can not do assignment between different types.
[Error in (15, 10)]  Can not do assignment between different types.
[Error in (16, 10)]  e do not have property z
```

Figure 3: The compiling results for the above code.

5 Code Generation

5.1 Visitor Patterns

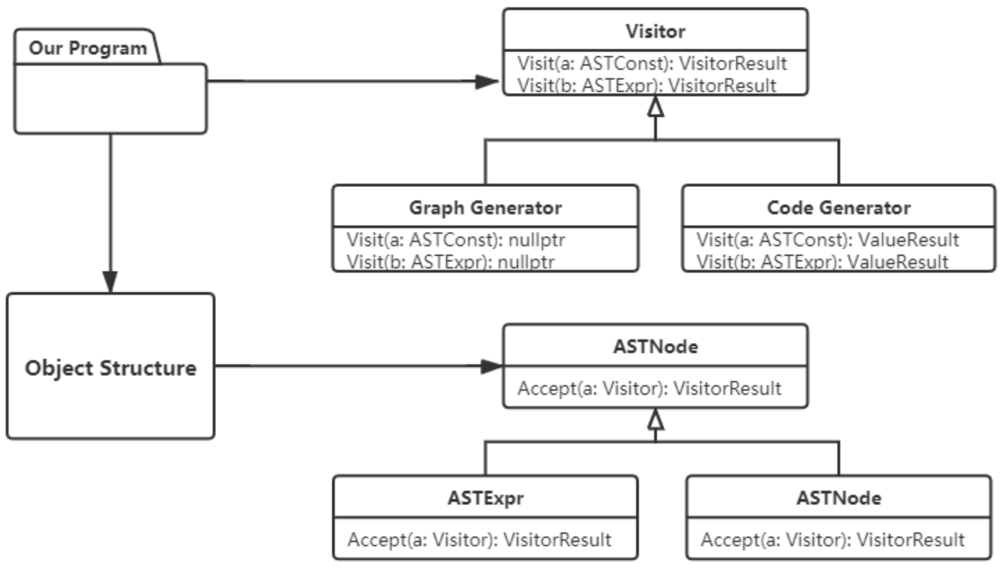

Our Program
Visitor
Visit(a: ASTConst): VisitorResult
Visit(b: ASTExpr): VisitorResult
Graph Generator
Visit(a: ASTConst): nullptr
Visit(b: ASTExpr): nullptr
Code Generator
Visit(a: ASTConst): ValueResult
Visit(b: ASTExpr): ValueResult
Object Structure
ASTNode
Accept(a: Visitor): VisitorResult
ASTExpr
Accept(a: Visitor): VisitorResult
ASTNode
Accept(a: Visitor): VisitorResult

Figure 4: The UML Digram of Visitor Pattern

In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.[7]

Listing 1: Example of Visitor Pattern

```
class Visitor {
    virtual Value VisitClassOne(ClassOne *obj);
    virtual Value VisitClassTwo(ClassTwo *obj);
    ...
}
class ClassOne {
    virtual Value Accept(Visitor *v) {return v->VisitClassOne(this);}
    ...
}
```

[7]wiki: Visitor Pattern

5.1.1 Why Visitor Pattern?

In the design of our compiler system, we need to perform different kinds of operations on different types of nodes in the existing AST object structure; e.g., we have different classes of node `ASTProgram`, `ASTTypedecl` and `ASTExpr`, etc., and we need to perform some operations on them such as AST visualizing, sementic checking and code-generation. A naive design of the system shown below is to add different kinds of methods in the Class `ASTNode` and enable virtual inheritance.

Listing 2: A Naive Design

```
class ASTNode {
    virtual llvm::Value codeGen(/*parameters*/) = 0;
    virtual void visualize(/*parameters*/) = 0;
}
```

However this design have two major drawbacks:

1. Limiting the return type of each methods, requires modifying object structure if the operations needs to use multiple return types.
2. The algorithms (operations) are highly coupled to the object structure, the code structure is mixed and adding operations requires modifying the object structure.

The main reason for this deficiency is that C++ only supports dynamic single-dispatch natively (through virtual inheritance) and not dynamic double-dispatch natively. Therefore we need to introduce visitor pattern to decouple the algorithms and the object structure, implement some sort of double-dispatch.

Morever, visitor pattern is widely used in processing/traversaling AST structure: ANTLR[8] adopts visitor pattern as one of the ways to traversal the abstract syntax tree.

5.1.2 Implementing Visitor Pattern in C++

Because we need to return different types of intermediate results when processing different actions of the AST (even different intermediate results in the processing of the same action), we need to make the implementation of the visitor pattern support the return of multi-type results. Here we can do it in the abstract class-derived way: We define a class `VisitorResult` and let the result classes of other types inherit from it. When using the value of the results, we need to cast the type of their pointer from `shared_ptr< VisitorResult>`.

```
class VisitorResult {};

class NameList : public VisitorResult {
public:
    NameList(const std::vector<std::string> &list) : list_(list) {};
```

[8]ANother Tool for Language Recognition https://www.antlr.org/

```
    ~NameList() = default;
    const std::vector<std::string> &getNameList() const { return this->list_; };
private:
    std::vector<std::string> list_;
};

class ValueResult : public VisitorResult {...};
class class ValueListResult : public VisitorResult {...};

...
```

Then we define the `Visitor` class and add `Accept` methods to each AST class, returning values with type `std::shared_ptr<VisitorResult>`.

```
class Visitor {

public:
    virtual ~Visitor() = default;
    virtual shared_ptr<VisitorResult> VisitASTNode(ASTNode *node) = 0;
    virtual shared_ptr<VisitorResult> VisitASTNameList(ASTNameList *node) = 0;
    ...
};

std::shared_ptr<VisitorResult> ASTStmt::Accept(Visitor *visitor) { return visitor
    ->VisitASTStmt(this); }

std::shared_ptr<VisitorResult> ASTStmtList::Accept(Visitor *visitor) { return
    visitor->VisitASTStmtList(this); }
...
```

Now we have implemented dynamic single-dispatch by virtual-inheriting `Accept` methods. To implement doubel-dispatch, we still need to virtual-inherit all the "Visit" methods by define deriving class of `Visitor` for different purposes:

```
class Generator : public Visitor {
    virtual std::shared_ptr<VisitorResult> VisitASTNode(ASTNode *node);
    virtual std::shared_ptr<VisitorResult> VisitASTNameList(ASTNameList *node);
    virtual std::shared_ptr<VisitorResult> VisitASTExpressionList(
        ASTExpressionList *node);
...
};
```

Then we can implement AST traversal and other operations in the definition of each "Visit" method. We can access properties of each node by predined getters and visit its sub-tree by `Accept` methods.

Listing 3: Example in code generation

```
std::shared_ptr<VisitorResult> Generator::VisitASTExpressionList(ASTExpressionList
     *node) {
    auto expr_list = node->getExprList();
    std::vector<std::shared_ptr<ValueResult>> ret;
    int cnt = 0;
    for (auto expr_node: expr_list){
        //Traverse sub-trees and get return values
        auto val = static_pointer_cast<ValueResult>(expr_node->Accept(this));
        if (val == nullptr) {
            std::cerr << "meet␣nullptr␣at␣VisitASTExpressionList!␣at␣parameter:␣"
                << cnt << std::endl;
            return nullptr;
        }
        ret.push_back(val);
        cnt++;
    }
    return std::make_shared<ValueListResult>(ret);
}
```

5.2 From Our-Pascal-Type to LLVM

In LLVM, types represent the type of a value. Types are associated with a context instance. The context internally deduplicates types so there is only 1 instance of a specific type alive at a time. In other words, a unique type is shared among all consumers within a context.

Therefore, when we start to generate intermediate code using LLVM backend, we ought to use LLVM API to transform our types for semantics representation, to `llvm::Type`, which is frequently required as a parameter when doing Alloc, CreateFunc, etc. Apart from our own type, we also need to provide the LLVMContext used in compilation.

```
llvm::Type *getLLVMType(llvm::LLVMContext &context, PascalType *const p_type);
```

We start from built-in types. For these we only need to return an LLVM pre-defined const pointer.

```
if (p_type->tg == PascalType::TypeGroup::BUILT_IN) {
    if (isEqual(p_type, INT_TYPE))
        return llvm::Type::getInt32Ty(context);
    else if (isEqual(p_type, REAL_TYPE))
        return llvm::Type::getDoubleTy(context);
    else if (isEqual(p_type, CHAR_TYPE))
        return llvm::Type::getInt8Ty(context);
    else if (isEqual(p_type, BOOLEAN_TYPE))
        return llvm::Type::getInt1Ty(context);
    else if (isEqual(p_type, VOID_TYPE))
        return llvm::Type::getVoidTy(context);
    else return nullptr;
}
```

For String and Array, the core work both lies on the creation of an `llvm::Array`, the only difference is that - String cannot be nested while when generating an array type, we have to recursively get its `element_type` first and then return the created `llvm::Array`.

```
else if (p_type->tg == PascalType::TypeGroup::STR) {
    StrType *str = (StrType *) p_type;
    return llvm::ArrayType::get(getLLVMType(context, CHAR_TYPE), (uint64_t) (str->
        dim));
} else if (p_type->tg == PascalType::TypeGroup::ARRAY) {
    ArrayType *array = (ArrayType *) p_type;
    llvm::ArrayType *ret = nullptr;
    int len = array->range.second - array->range.first + 1;
    std::cout << "creating␣array␣of␣length␣" <<len << std::endl;
    ret = llvm::ArrayType::get(getLLVMType(context, array->element_type), (
        uint64_t) len);
    return ret;
}
```

For Record, we need to use the static function `get` provided by llvm::StructType. This type shows why a semantic type class is necessary. As is shown in the code, when we construct a llvm::StrucType, we only provide the llvm::Type of each component, but without their names. However, if we want to load or store a member value, we access them with their names. To this respect, the semantics type and llvm::Type is both required when operating a Pascal Record.

```
else if (p_type->tg == PascalType::TypeGroup::RECORD) {
    RecordType *record = (RecordType *) p_type;
    std::vector<llvm::Type *> llvm_type_vec;
    for (auto t : record->type_vec) {
        llvm_type_vec.push_back(getLLVMType(context, t));
    }
    return llvm::StructType::get(context, llvm_type_vec);
}
```

Finally it comes to the Enum Type mentioned in **Chapter 4.2**, we use a tricky idea and make an Enum Type an Integer Type actually.

```
else if (p_type->tg == PascalType::TypeGroup::ENUM) {
    return llvm::Type::getInt32Ty(context);
}
```

5.3 A Challenge: Sub-Sub-Program and its parameters

We all know what a sub-program is. A sub-program can also be named as a function or a procedure, that can be called by the main process or another sub-program. But what is a sub-sub-program? Actually, it is one of the major features that Pascal differs from C. In Pascal, following code is allowed and similar structure is frequently used.

```
program my;
var a : integer;
procedure f(a : integer);
    procedure g;
    begin
        a := 2;
    end;
begin
    writeln(a);
    g;
    writeln(a);
end;
begin
    a := 3;
    writeln(a);
    f(a);
    writeln(a);
end.
```

The output of the program should be `3 2 3 3` (replace the space with enter). In this example, procedure f is a sub-program, and procedure g is a **sub-sub-program** that is defined in the *routine_head* of procedure f. Actually, we can define another procedure h in the *routine_head* of procedure g and make it a sub-sub-sub-program, but this example is good enough to explain the challenge we encountered.

In this case, procedure f has a *val variable* **a**, whose modification will not affect the caller's parameter. Procedure g assigns value **2** to variable **a**, while it is clear that g does not have this variable in its own function_head or routine_head, thus this **a** is that of function f.

In C language, such kind of function definition is not allowed, we cannot define a function in another function, therefore LLVM does not support such situation. We now show what the major problem that supports this argument. Considering the symbol table mentioned in **Chapter 4.2**, the code block stack represents the hierarchy of function definition, and the local variables are stored in the symbol table that belongs to the current function, or procedure, see the following figure. It means that, in the example above, to implement the assignment of **a** in procedure g, we have to scan from the top to the bottom, find the nearest *llvm::AllocaInst** that corresponds to this **a** and use it to call IRBuilder.CreateLoad.

However, here comes the biggest problem - **We cannot implement *IRBuilder.CreateLoad* to an**

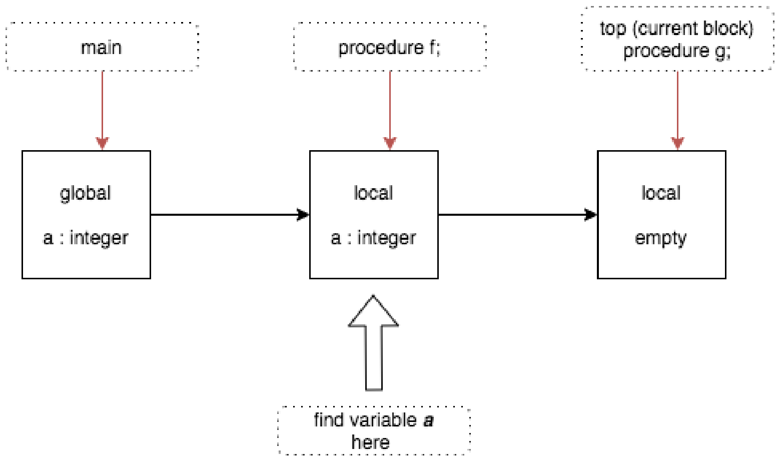


Figure 5: How to find a non-local nor global variable using codeblock stack.

***llvm::AllocaInst** in the previous codeblock.** The reason is that, the *llvm::AllocaInst** is actually related to the **register** that stores the variable [9]. We all know that, on function calling, there is a caller and callee protection mechanism. That is to say, when you enter the basic block of procedure g, the registers used for variables in procedure f are no longer corresponded. They have been protected and saved to an unknown place, and if you try to load their values, you cannot avoid encountering fatal errors.

You may ask that C language has a far more complicated local variable system but why they do not have such problems? The local variables of C languages are mainly dealt within a function, when you enter another function and create another environment, old variables are no longer allowed to use, because these two functions never follow a definition hierarchy. Those local variables that may come across similar issues are those defined in and out of the **for statement** or **if-else statement**, etc. These statements are not functions, so we do not need a function calling process to enter their environments, thus symbol tables can be constructed in a normal and natural way, while LLVM supports them very well.

Now let us go back to Pascal, how can we solve this issue? The solution we provide is to pass **all local variables** that a sub-sub-program should be able to access as its parameters. As an intuition can tell, such approach will use far more space than the natural symbol table construction way, but we have no other choices, so we make our implementation as economical as possible. First of all, we never pass any global variables, because they can be accessed anywhere themselves. Secondly, we do not pass the *named_types* table, because we only store semantics type in this table and generate the *llvm::Type* on *CreateAlloca*, therefore they can be stored separately in each own code block.

In this approach, it is easy to find that, we only need to scan the first code block for global variables, and the last for any local ones. See the coming figure to check and strengthen your comprehension.

To achieve this pattern, we have to modify the way we define and call a function. We have to define the function args not only considering the user's program, but also the all namespace that currently we can

[9]More precisely maybe a register that stores the start address in the memory and another for offset, but not important

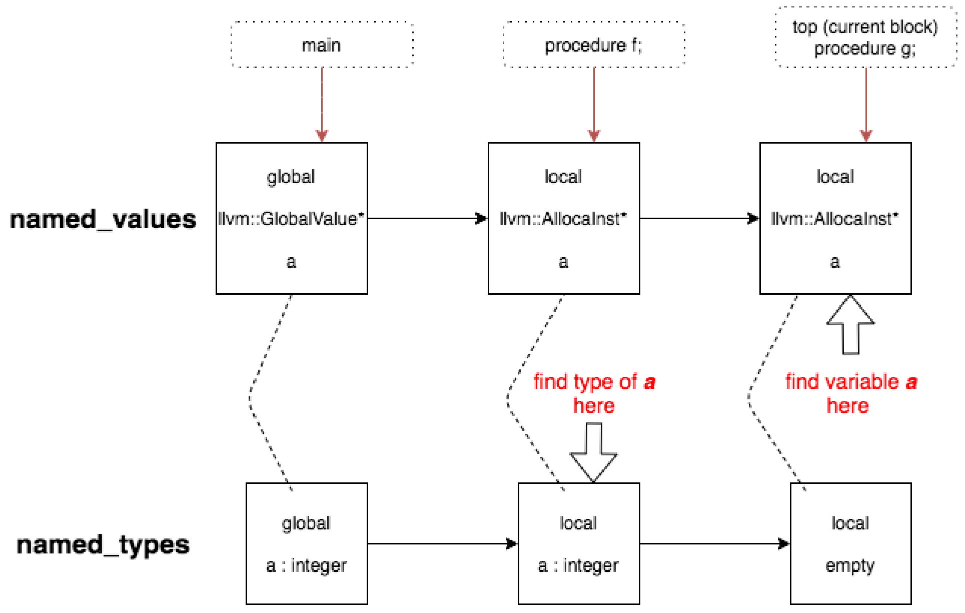


Figure 6: Our approach designed for Pascal Grammar.

access. On calling, we get the names of all local variables that should be passed from the semantics function head, and search them each respectively. The actual work is even more complicated. We have to consider the fact that the arguments and local variables of newly defined function may overwrite those previous ones. So we must carefully design the order of argument-passing. The most proud thing is that we achieve this pattern within 200 line codes, and the total number of lines of this project is about 7000.

Detailed codes can be found in *Generator::VisitASTFuncProcBase* of `src/generator/generator_program.cpp` and in *Generator::VisitASTFuncCall* of `src/generator/generator_expr.cpp`. They are also attached in the appendix of this report, read the codes and check the ideas provided above.

5.4 Operate LLVM Blocks to Interpret Statements

In this part, we will discuss how to interpret and generate codes of several types of statements, and show our general ideas by using part of the code as examples.

5.4.1 Conditional Branch Statements

In pascal, we have two kinds of conditional branch statements : IF statements and SWITCH statements. Since LLVM IR use blocks to organize the structure of the program, here we generate each part of the code of the statements into an `llvm::BasicBlock`, and use `llvm::CreateBr` and `llvm::CreateCondBr` to generate branch instruction and conditional-branch instruction, which are `br` instructions in LLVM IR.

After analysing the logic structure of `IF` and `SWITCH` (**Figure 7**), it's not difficult for us to finish this part of code generation. The code of `Generator::VisitIfStmt` is shown below as an example:

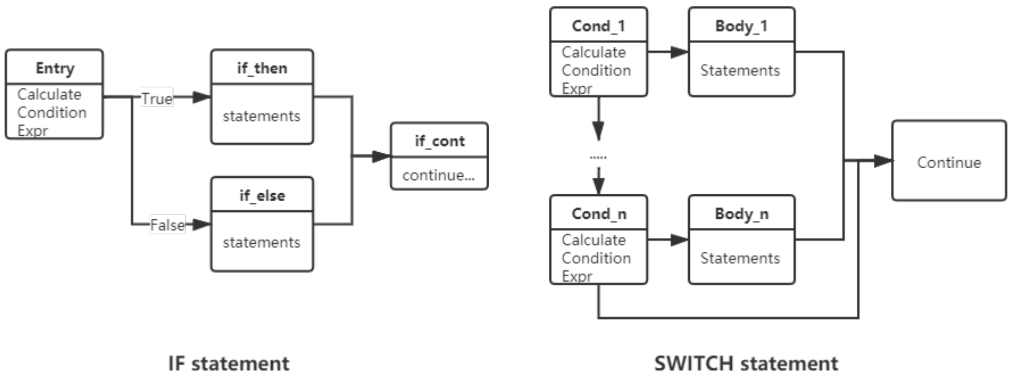


Figure 7: Logic Digrams of the Two Pascal Conditional Statements

Listing 4: Generator::VisitIfStmt

```
std::shared_ptr<VisitorResult> Generator::VisitASTIfStmt(ASTIfStmt *node) {
    llvm::Function *func = this->builder.GetInsertBlock()->getParent();
    llvm::BasicBlock *then_block = llvm::BasicBlock::Create(this->context, "
        if_then", func);
    llvm::BasicBlock *else_block = llvm::BasicBlock::Create(this->context, "
        if_else", func);
    llvm::BasicBlock *cont_block = llvm::BasicBlock::Create(this->context, "
        if_cont", func);
    auto cond_res = std::static_pointer_cast<ValueResult>(node->getExpr()->Accept(
        this));
    this->builder.CreateCondBr(cond_res->getValue(), then_block, else_block);
    this->builder.SetInsertPoint(then_block);
    node->getStmt()->Accept(this);
    this->builder.CreateBr(cont_block);
    this->builder.SetInsertPoint(else_block);
    if (node->getElseClause() != nullptr) {
        node->getElseClause()->Accept(this);
    }
    this->builder.CreateBr(cont_block);
    this->builder.SetInsertPoint(cont_block);
    return nullptr;
}
```

5.4.2 Loop Statements

Loop statements can be considered as special forms of conditional branch statements which contain loop logical structure. Here we will describe the logical structure of the three kinds of loop statements literally.

REPEAT..UNTIL Statement: Can be divided into three blocks, **repeat_body**, **repeat_cond**, **repeat_cont**.

1. **repeat_body**: Whatever the conditional expression is, the body of REPEAT statement will run once at the entry point;
2. **repeat_cond**: Corresponding to the part of UNTIL, calculate the conditional expression, and conditionally branch to **repeat_body** (if False) or **repeat_cont** (if True).
3. **repeat_cont**: The end of the loop, continue for other statements.

WHILE Statement: Approximately, it can be divided into three blocks, **while_cond**, **while_body**, **while_cont**.

1. **while_cond**: After the entry point, we need to check the condition first; If true, branch to **while_body**, otherwise branch to **while_cont**.
2. **while_body**: Run the statements in the block, and branch to **while_cond**. Here is where the looping happens.
3. **while_cont**: The end of the loop, continue for other statements.

FOR Statement: Actually **FOR** has the most complicated logical structure in these statements, here we only consider **TO** and **DOWNTO** "For expressions". We need to do assignments and calculations on the iterator variable and use that as our branch conditions.

We have divided **FOR** into 5 parts:

1. **for_start**: The entry point of **FOR**, assign the initial value to the iterator, and check whether the initial value is in the loop range, branch to **for_body** if True, otherwise the **for_end**;
2. **for_body**: Generates the statements in the body part of **FOR**, then branch to **for_cond**.
3. **for_cond**: Step one and update the value of the iterator, check whether the iterator exceeds the range of for loop, if True, branch to **for_step_back**, otherwise **for_body**;
4. **for_step_back**: Since after the loop the iterator value should be the last legal value in the range, when the iterator exceeds the range, it should step one back to recover the value. After that, branch to **for_end**;
5. **for_end**: The end of the loop, continue for other statements.

It is worth mentioning that for the assignments and calculations in **FOR** statements, we can implement semantic expansion by creating temporary `ASTAssignStmt` and `ASTExpr` nodes, and then use the `Accept` method to generate corresponding code, which greatly increases the reuse rate of our code.

BREAK statement: Pascal support **BREAK** statement to exit current loop. To implement this feature, we need to record the "continue" block of current loop in the stack-like structure. Here we store them in `CodeBlock::loop_breaks`. Creating a Br to this block enable us to end the loop imediately:

Listing 5: Generator::VisitASTBreakStmt

```
this->builder.CreateBr(this->getCurrentBlock()->loop_breaks.back());
```

```
llvm::Function *func = this->builder.GetInsertBlock()->getParent();
llvm::BasicBlock *cont_block = llvm::BasicBlock::Create(this->context, "
    break_cont", func);
this->builder.SetInsertPoint(cont_block);
```

At last we show part of the code in `Generator::VisitASTWhileStmt` to demonstrate our idea.

Listing 6: Generator::VisitASTWhileStmt

```
this->getCurrentBlock()->loop_breaks.push_back(end_block);

this->builder.CreateBr(cond_block);
this->builder.SetInsertPoint(cond_block);

auto cond_res = std::static_pointer_cast<ValueResult>(node->getExpr()->Accept(
    this));
if (cond_res == nullptr || !isEqual(cond_res->getType(), BOOLEAN_TYPE))
    return RecordErrorMessage("Invalid␣condition␣in␣while␣statement.", node->
        get_location_pairs());

this->builder.CreateCondBr(cond_res->getValue(), body_block, end_block);
this->builder.SetInsertPoint(body_block);
node->getStmt()->Accept(this);

this->builder.CreateBr(cond_block);
this->builder.SetInsertPoint(end_block);

this->getCurrentBlock()->loop_breaks.pop_back();
```

5.4.3 Assignment Statement

Since we can process types, variables, values correctly, we can always get the pointer and the value by `ValueResult::getMem` and `ValueResult::getValue` from a legal left expression and a legal right expression; Besides LLVM supports assignment of struct and arrays natively; We can directly use `LLVM::CreateStore` to generate assignment statements.

5.4.4 Statements about Functions and Procedures

For statements of procedure calling (`ASTProcStmt`), since we consider procedures as void-type functions, we can create temporary `ASTFuncCall` to generate its code.

More importantly, we also implement the **EXIT** statement to exit the current procedure or function immediately. This operation is created through `LLVM::CreateRet`; Because in Pascal, an exit statement can be called without arguments to return a result variable named after the function in the current scope, or to exit the current procedure, we need to make an appropriate assessment of the current environment.

```
shared_ptr<VisitorResult> Generator::VisitASTExitStmt(ASTExitStmt *node) {
    std::shared_ptr<ValueResult> res;
    if (node->getExpr() == nullptr) {
        if (this->block_stack.size() == 1) {//is global
            this->builder.CreateRet(llvm::ConstantInt::get(llvm::Type::getInt32Ty(
                this->context), 0));
        }
        else if (!this->getCurrentBlock()->is_function) {
            this->builder.CreateRetVoid();
        }
        else {
            llvm::Value* ret = this->builder.CreateLoad(this->getCurrentBlock()->
                named_values[this->getCurrentBlock()->block_name], "ret");
            this->builder.CreateRet(ret);
        }
    }
    else {
        res = static_pointer_cast<ValueResult>(node->getExpr()->Accept(this));
        this->builder.CreateRet(res->getValue());
    }
    llvm::Function *func = this->builder.GetInsertBlock()->getParent();
    llvm::BasicBlock *cont_block = llvm::BasicBlock::Create(this->context, "
       exit_cont", func);
    this->builder.SetInsertPoint(cont_block);
    return nullptr;
}
```

5.5 System Functions

Implementing necessary system built-in functions is important for the functionality of a compiler. Here we have implemented two most frequently use procedures: `WRITE/WRITELN`, `READ/READLN`.

Since LLVM supports C-like `printf` and `scanf` functions, we choose to implement our I/O functionalities based on them. In the previous sections, we have introduce how we maintain type information and memory/value information of constants and variables; So we can construct the format string of `printf` and `scanf`, passing corresponding values or pointer values as parameters to implement `WRITE/WRITELN` and `READ/READLN`.

In the first time of generation of system functions, we need to register them and get corresponding `llvm::Funtion*` reference.

Here we show part of the code of our WRITE/WRITELN procedure generation:

```
llvm::Value* GenSysWrite(const std::vector<std::shared_ptr<ValueResult>> &
    args_list, bool new_line, Generator *generator) {
    static llvm::Function *llvm_printf = nullptr;
    if (llvm_printf == nullptr) {
        //register printf
        ...
    }
    std::string format;
    std::vector<llvm::Value *> printf_args;
    printf_args.emplace_back(nullptr);
    for (auto arg: args_list) {
        OurType::PascalType *tp = arg->getType();
        if (tp->isIntegerTy()) {
            format += "%d";
            printf_args.emplace_back(arg->getValue());
        }
        ...
        else if (tp->isStringTy()) {
            format += "%s";
            printf_args.emplace_back(arg->getMem());
        }
    }
    if (new_line) {
        format += "\n";
    }
    printf_args[0] = generator->builder.CreateGlobalStringPtr(format, "
        printf_format");
    return generator->builder.CreateCall(llvm_printf, printf_args, "call_printf");
}
```

6 Tests and Results

6.1 Recursive Function Test

```
program test_recursive_gcd;
var
    tcase, a, b: integer;
function gcd(a: integer; b: integer): integer;
    begin
        if b = 0 then
            gcd := a
        else
            gcd := gcd(b, a mod b);
    end;
begin
    read(tcase);
    while tcase > 0 do
    begin
        tcase := tcase - 1;
        read(a, b);
        writeln(gcd(a, b));
    end;
end.
```

Compiling successfully, and the IR Code is as follows.

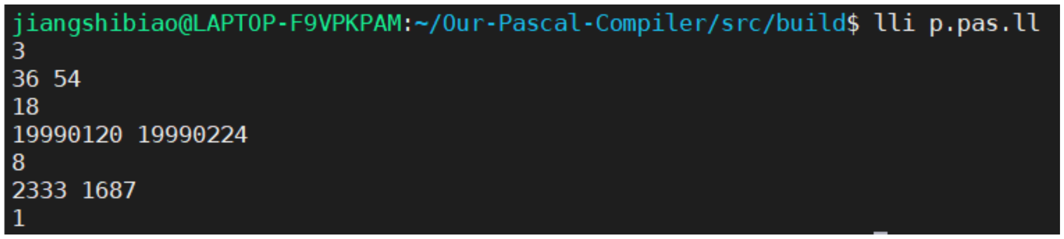
```
...
define i32 @gcd(i32*, i32*) {
entry:
  %2 = load i32, i32* %0
  %a = alloca i32
  store i32 %2, i32* %a
  %3 = load i32, i32* %1
  %b = alloca i32
  store i32 %3, i32* %b
  %gcd = alloca i32
  %4 = load i32, i32* %b
  %cmptmp = icmp eq i32 %4, 0
  br i1 %cmptmp, label %if_then, label %if_else

if_then:                                          ; preds = %entry
  %5 = load i32, i32* %gcd
  %6 = load i32, i32* %a
  store i32 %6, i32* %gcd
  br label %if_cont

if_else:                                          ; preds = %entry
  %7 = load i32, i32* %gcd
  %8 = load i32, i32* %b
  %9 = load i32, i32* %a
  %10 = load i32, i32* %b
  %modtmp = srem i32 %9, %10
  %"0_1" = alloca i32
  store i32 %modtmp, i32* %"0_1"
  %11 = call i32 @gcd(i32* %b, i32* %"0_1")
  store i32 %11, i32* %gcd
  br label %if_cont

if_cont:                                          ; preds = %if_else, %if_then
  %12 = load i32, i32* %gcd
  ret i32 %12
}
...
```

Execute the target code successfully.

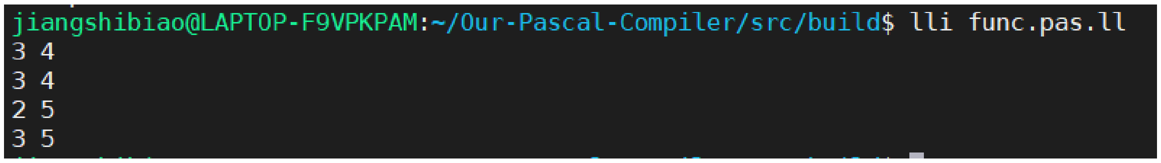


```
jiangshibiao@LAPTOP-F9VPKPAM:~/Our-Pascal-Compiler/src/build$ lli p.pas.ll
3
36 54
18
19990120 19990224
8
2333 1687
1
```

Figure 8: The result for recursive function test

6.2 Nested Function Test

```
program myfunc;
var a, b : integer;
procedure f(a : integer; var b: integer);
    procedure g;
    begin
        a := 2;
    end;
begin
    writeln(a, '␣', b);
    g;
    b := 5;
    writeln(a, '␣', b);
end;
begin
    a := 3;
    b := 4;
    writeln(a, '␣', b);
    f(a, b);
    writeln(a, '␣', b);
end.
```

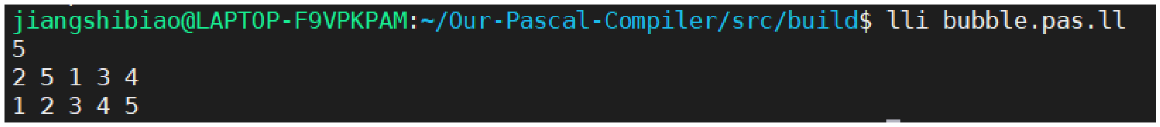


```
jiangshibiao@LAPTOP-F9VPKPAM:~/Our-Pascal-Compiler/src/build$ lli func.pas.ll
3 4
3 4
2 5
3 5
```

Figure 9: The result for nested function test

6.3 Bubble Sort Test

```
program test_bubble;
var
  r:array[1..100] of integer;
  i, j, n, temp: integer;
  flag: boolean;
begin
  readln(n);
  for i:=1 to n do
    read(r[i]);
  for i:=1 to n-1 do
    begin
      flag:=true;
      for j:=n-1 downto i do
        if r[j+1]<r[j] then
          begin
            temp:=r[j+1];
            r[j+1]:=r[j];
            r[j]:=temp;
            flag:=false;
          end;
        if flag then break;
    end;
  for i:=1 to n do
    if i < n then write(r[i], '␣') else writeln(r[i]);
end.
```



```
jiangshibiao@LAPTOP-F9VPKPAM:~/Our-Pascal-Compiler/src/build$ lli bubble.pas.ll
5
2 5 1 3 4
1 2 3 4 5
```

Figure 10: The result for bubble sort

7 Group Work Division

To better learn about the whole structure of our compiler, we work through the whole process as three full-stack engineers together instead of dividing our work by stages. Thanks to the VSCode Live Share, we can develop the codes together.

8 Appendix

Listing 7: Code for defining a function or procedure

```
std::shared_ptr<VisitorResult> Generator::VisitASTFuncProcBase(ASTFuncProcBase *
    node) {
    bool is_function = node->getIam() == ASTFuncProcBase::FuncType::FUNCTION;
    auto parameters = std::static_pointer_cast<TypeListResult>(
        is_function ? ((ASTFunctionDecl*)node)->getFunctionHead()->getParameters()
            ->Accept(this)
                    : ((ASTProcedureDecl*)node)->getProcedureHead()->getParameters
                        ()->Accept(this));

    if (parameters == nullptr)
        return RecordErrorMessage("Can␣not␣recognize␣the␣parameters␣for␣function/
            procedure␣definition.", node->get_location_pairs());

    OurType::PascalType *return_type = OurType::VOID_TYPE;
    std::string func_name;
    if (is_function){
        func_name = ((ASTFunctionDecl*)node)->getFunctionHead()->getFuncName();
        auto return_type_result = std::static_pointer_cast<TypeResult>(((
            ASTFunctionDecl*)node)->getFunctionHead()->getSimpleTypeDecl()->Accept(
            this));
        if (return_type_result == nullptr)
            return RecordErrorMessage("Can␣not␣recognize␣the␣return␣type␣for␣the␣
                function␣definition.", node->get_location_pairs());
        return_type = return_type_result->getType();
    }else{
        func_name = ((ASTProcedureDecl*)node)->getProcedureHead()->getProcName();
    }
    llvm::Type *llvm_return_type = OurType::getLLVMType(context, return_type);
    auto name_list = parameters->getNameList();
    auto type_var_list = parameters->getTypeList();
    std::vector<llvm::Type*> llvm_type_list;
    std::vector<OurType::PascalType*> type_list;
    std::vector<bool> var_list;
```

```
for (int i = 0; i < name_list.size(); i++)
    for (int j = i+1; j < name_list.size(); j++)
        if (name_list[i] == name_list[j])
            return RecordErrorMessage("The␣parameters␣in␣the␣function/
                procedure␣definition␣are␣duplicated.", node->get_location_pairs
                ());

// Adding local variables
// we must put local variables first
// because after we create this function,
// we have to add the variables to the next CodeBlock
// in this step, we must add the function parameters later
// so as to overwrite the older local variables
auto local_vars = this->getAllLocalVarNameType();
std::vector<std::string> local_name_list = local_vars.first;
std::vector<OurType::PascalType *> local_type_list = local_vars.second;
for(int i = 0; i < local_name_list.size(); i++) {
    name_list.push_back(local_name_list[i]);
    type_list.push_back(local_type_list[i]);
    var_list.push_back(true);
    llvm_type_list.push_back(llvm::PointerType::getUnqual(OurType::getLLVMType
        (context, local_type_list[i])));
}

// adding function parameters
for (auto type: type_var_list){
    type_list.push_back(type->getType());
    var_list.push_back(type->is_var());
    llvm_type_list.push_back(llvm::PointerType::getUnqual(OurType::getLLVMType
        (context, type->getType())));
}

FuncSign *funcsign = new FuncSign((int)(local_name_list.size()), name_list,
    type_list, var_list, return_type);
llvm::FunctionType *functionType = llvm::FunctionType::get(llvm_return_type,
    llvm_type_list, false);
llvm::Function *function = llvm::Function::Create(functionType, llvm::
    GlobalValue::ExternalLinkage, func_name, module.get());

this->getCurrentBlock()->set_function(func_name, function, funcsign);

llvm::BasicBlock* oldBlock = this->builder.GetInsertBlock();
```

```
    llvm::BasicBlock* basicBlock = llvm::BasicBlock::Create(context, "entry",
        function, nullptr);
    this->builder.SetInsertPoint(basicBlock);

    // MODIFY PARAMETERS PASSING
    block_stack.push_back(new CodeBlock());
    this->getCurrentBlock()->block_name = func_name;
    this->getCurrentBlock()->is_function = is_function;
    int iter_i = 0;
    for(llvm::Function::arg_iterator arg_it = function->arg_begin(); arg_it !=
        function->arg_end(); arg_it++, iter_i++) {
        if (var_list[iter_i]) {
            this->getCurrentBlock()->named_values[name_list[iter_i]] = (llvm::
                Value *)arg_it;
            if (iter_i >= local_name_list.size())
                this->getCurrentBlock()->named_types[name_list[iter_i]] =
                    type_list[iter_i];
            std::cout << "Inserted␣var␣param␣" << name_list[iter_i] << std::endl;
        } else {
            llvm::Value *value = this->builder.CreateLoad((llvm::Value *)arg_it);
            llvm::AllocaInst *mem = this->builder.CreateAlloca(
                OurType::getLLVMType(this->context, type_list[iter_i]),
                nullptr,
                name_list[iter_i]
            );
            this->builder.CreateStore(value, mem);
            this->getCurrentBlock()->named_values[name_list[iter_i]] = mem;
            if (iter_i >= local_name_list.size())
                this->getCurrentBlock()->named_types[name_list[iter_i]] =
                    type_list[iter_i];
            std::cout << "Inserted␣val␣param␣" << name_list[iter_i] << std::endl;
        }
    }

    // add function to named_value for itself
    if (is_function) {
        llvm::AllocaInst *mem = this->builder.CreateAlloca(
            OurType::getLLVMType(this->context, return_type),
            nullptr,
            func_name
        );
        this->getCurrentBlock()->named_values[func_name] = mem;
        this->getCurrentBlock()->named_types[func_name] = return_type;
        std::cout << "Inserted␣val␣param␣" << func_name << std::endl;
    }

    // add return mechanism
    if (is_function) {
        ((ASTFunctionDecl*)node)->getRoutine()->Accept(this);
        if (this->block_stack.size() == 1) {
            this->builder.CreateRet(llvm::ConstantInt::get(llvm::Type::getInt32Ty(
               this->context), 0, true));
        } else {
            llvm::Value *ret = this->builder.CreateLoad(this->getCurrentBlock()->
               named_values[func_name]);
            this->builder.CreateRet(ret);
        }
    } else {
        ((ASTProcedureDecl*)node)->getRoutine()->Accept(this);
        this->builder.CreateRetVoid();
    }

    this->builder.SetInsertPoint(oldBlock);
    this->block_stack.pop_back();
    return nullptr;
}
```

Listing 8: Code for calling a function or procedure

```
std::shared_ptr<VisitorResult> Generator::VisitASTFuncCall(ASTFuncCall *node) {
    ASTExpressionList *argList = node->getArgList();
    std::shared_ptr<ValueListResult> value_list;
    std::vector<std::shared_ptr<ValueResult>> value_vector;
    bool have_args = true;
    if (argList == nullptr) {
        have_args = false;
    } else {
        value_list = std::static_pointer_cast<ValueListResult>(node->getArgList()
            ->Accept(this));
        value_vector = value_list ->getValueList();
        have_args = true;
    }

    std::string func_name = node->getFuncId();
    for (int i = block_stack.size() - 1; i >= 0; i--){
        FuncSign *funcsign = block_stack[i]->find_funcsign(func_name);
        if (funcsign == nullptr ) continue;
        // Note the function/procedure can not be overridden in pascal, so the
```

```
            function is matched iff the name is matched.
        // VERY IMPORTANT !!!
        // NameList().size include all local variables that require to be passed
        // we should compare NameList.size() - n_local
        // which is the actual arg size
        if (funcsign->getNameList().size() - funcsign->getLocalVariablesNum() !=
            value_vector.size())
            return RecordErrorMessage("Can't␣find␣function␣" + func_name + ":␣you␣
                have␣" + std::to_string(value_vector.size()) + "parameters,␣but␣the
                ␣defined␣one␣has␣"
              + std::to_string(funcsign->getNameList().size() - funcsign->
                getLocalVariablesNum()) + "parameters.", node->get_location_pairs
                ());

        auto name_list = funcsign->getNameList();
        auto type_list = funcsign->getTypeList();
        auto var_list = funcsign->getVarList();
        auto return_type = funcsign->getReturnType();
        llvm::Function *callee = block_stack[i]->find_function(func_name);
        std::vector<llvm::Value*> parameters;

        // adding local variables
        // in generator_program.cpp, we define all locals at the head of the para
            list
        int cur;
        int n_local = funcsign->getLocalVariablesNum();
        for(cur = 0; cur < n_local; cur++) {
            std::string local_name = name_list[cur];
            if (this->getCurrentBlock()->named_values.find(local_name) == this->
                getCurrentBlock()->named_values.end()) {
                std::cout << node->get_location() << "local␣variable␣" <<
                    local_name << "␣need␣to␣be␣passed,␣but␣not␣found." << std::endl
                    ;
                parameters.push_back(nullptr);
            } else {
                parameters.push_back(this->getCurrentBlock()->named_values[
                    local_name]);
            }
        }

        // PASSING function args
        for (auto value: value_vector){
            if (!isEqual(value->getType(), type_list[cur]))
                return RecordErrorMessage("Type␣does␣not␣match␣on␣function␣" +
                    func_name + "␣calling.", node->get_location_pairs());
            if (value->getMem() != nullptr) {
                parameters.push_back(value->getMem());
            } else {
                this->temp_variable_count++;
                // here we encounter a literally const value as a parameter
                // we add a local variable to the IRBuilder
                // but do not reflect it in Current_CodeBlock->named_values
                // thus we do not add abnormal local variables when we declare
                    another function/procedure
                llvm::AllocaInst *mem = this->builder.CreateAlloca(
                    getLLVMType(this->context, type_list[cur]),
                    nullptr,
                    "0_" + std::to_string(this->temp_variable_count)
                );
                this->builder.CreateStore(value->getValue(), mem);
                parameters.push_back(mem);
            }
            cur++;
        }
        auto ret = builder.CreateCall(callee, parameters);
        if (funcsign->getReturnType()->tg == OurType::PascalType::TypeGroup::STR)
            {
            // to return a str type for writeln to print
            // we have to use its pointer
            // to achieve this, we add a never used variable here
            // we do this shit only to the str type return value
            // VERY BAD CODING STYLE
            // NEED TO BE MODIFIED ASAP
            this->temp_variable_count++;
            std::cout << ((OurType::StrType *)funcsign->getReturnType())->dim <<
                std::endl;
            llvm::AllocaInst *mem = this->builder.CreateAlloca(
                OurType::getLLVMType(this->context, funcsign->getReturnType()),
                nullptr,
                "0_" + func_name + std::to_string(this->temp_variable_count)
            );
            this->builder.CreateStore(ret, mem);
            llvm::Value *value = this->builder.CreateLoad(mem);
            return std::make_shared<ValueResult>(funcsign->getReturnType(), value,
                 mem); //, ret->getPointerOperand()); //, "call_"+ node->getFuncId
                ()
        } else {
            return std::make_shared<ValueResult>(funcsign->getReturnType(), ret);
        }
    }
    // Currently, sys_function will use no local variables that has cascade
       relation
    // So we do not need to deal with the locals and do it simply
    if (isSysFunc(node->getFuncId())) {
        return std::make_shared<ValueResult>(OurType::VOID_TYPE, genSysFunc(node->
            getFuncId(), value_vector));
    }
    return RecordErrorMessage("Function␣" + func_name + "␣not␣found.", node->
       get_location_pairs());
}
```