Programming Language Assignment 4

환경 : JAVA 17

2020314193 HWNAG SU YEONG

Expr.g4

```
grammar Expr;

// parser rules
prog : (statement ';' NEWLINE?)*;

statement : decl_list prog_expr      # FunctionExecution
          | prog_expr                # SingleExecution
          ;

decl_list : decl decl_list    # DeclList
          | decl              # LastDecl
          ;

decl : 'def' function_id param_list '=' func_expr 'endef'     # FuncDeclareWithParam
     | 'def' function_id '=' func_expr 'endef'                # FuncDeclare
     ;

param_list : param_id param_list          # ParamList
           | param_id                     # LastParam
           ;

prog_expr : expr                          # ProgramExpr
          ;

func_expr : expr                          # FunctionExpr
          ;
expr : '(' expr ')'                  # Parenthesis
     | 'let' ID '=' expr 'in' expr   # LetIn
     | ID '(' ')'                    # CallFunction
     | ID '(' expr_list ')'          # CallFunctionWithParameter
     | expr ('*'|'/') expr           # Operation
     | expr ('+'|'-') expr           # Operation
     | num                           # Number
     | ID                            # Variable
     | '(' expr ')'                  # Parenthesis
     ;

expr_list : expr ',' expr_list       # ExprList
     | expr                          # LastExpr
     ;
```

prog

- start rule에 해당된다.
- statement 뒤에는 항상 `;`이 와야한다.
- PA1의 기본 구조를 차용하여 여러 줄을 입력받을 수 있도록 하였다. (NEWLINE의 존재)
  (PA4에서는 한 번에 한 줄만 입력되지만, 여러 줄을 입력받을 수 있더라도 문제되지 않음)

statement

- decl_list prog_expr 혹은 prog_expr을 statement로 인식한다.
- 주어진 PA4 템플릿에서는 expr만 단독으로 사용하지만 function body의 expr, let-in 구문의 expr, F1VAE 실행을 위한 expr이 구분되어야 하므로 prog_expr을 따로 정의하였다.

decl_list

- decl_list는 주어진 PA4 템플릿의 grammar를 그대로 사용하였다.

decl

- 주어진 PA 템플릿에서 는 decl를 정의할 때 var를 사용하지만 함수 이름, 파라미터 이름이 구분되어야 하므로 function_id, param_list를 사용하였고 함수의 몸체에 해당하는 expr 역시 func_expr로 정의하였다.
- parameter가 없는 함수 정의, paramter가 있는 함수 정의 둘 다 decl로 인식한다

param_list

- F1VAE에서는 파라미터가 존재하는 경우 1개와 여러개 모두 가능하므로, param_list 여러 파라미터에 해당하는 param_list를 정의하였다.

prog_expr, func_expr

- prog_expr과 func_expr 모두 정의하는 문법 자체는 동일하므로 expr을 사용한다.

expr

- PA 템플릿에 있는 expr을 그대로 반영하였다.

expr_list

- 함수 argument에 해당하는 expr_list는 콤마로 구분된다.

program.java

```java
public class program {

    public static void main(String[] args) throws IOException {

        // Get Lexer
        ExprLexer lexer = new ExprLexer(CharStreams.fromStream(System.in));

        // Get a list of matched tokens
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Pass tokens to parser
        ExprParser parser = new ExprParser(tokens);
        ParseTree antlrAST = parser.prog();

        // Build AST
        // Build the AST with BuildAstVisitor.java
        BuildAstVisitor buildAstVisitor = new BuildAstVisitor();
        buildAstVisitor.visit(antlrAST);

        // Get built AST with AstCall.java
        AstCall astCall = new AstCall();
        String structure = astCall.buildAST(buildAstVisitor.getAstNodesList());

        // Evaluate AST
        Evaluate evaluate = new Evaluate();
        evaluate.buildAST(buildAstVisitor.getAstNodesList());
        String output = evaluate.getOutput();

        // Print Result
        System.out.println(structure);
        System.out.println(output);
    }
}
```

- main 메소드를 갖고있어 프로그램의 entry point를 담당하는 클래스
- antlrAST를 BuildAstVisitor 클래스에 전달하여 AST 구조를 생성한다
- AST 구조는 BuildAstVisitor 클래스의 멤버인 AstNodesList에 저장된다
- AstCall과 Evaluate에서 AST 구조를 전달받아 출력 결과를 저장한다

AstNodes.java

```java
public class AstNodes{
    String functionId;
    List<String> paramList;
    AstNodes functionExpr;

    String id;
    String paramId;
    Double value;
    AstNodes left;
    AstNodes right;

    AstNodes x1;
    AstNodes x2;

    AstNodes programExpr;
    AstType type;
    List<AstNodes> exprList;
    List<AstNodes> declList;
    ...
}
```

AstNodes 클래스의 멤버

- Expr.g4에서 각 non-terminal과 terminal을 멤버변수로 갖고 있다.

```java
    public AstNodes visitFunctionExecution(List<AstNodes> declList, AstNodes programExpr) {
        this.declList = declList;
        this.programExpr = programExpr;
        this.type = AstType.FunctionExecution;
        return this;
    }

    public AstNodes visitSingleExecution(AstNodes programExpr) {
        this.programExpr = programExpr;
        this.type = AstType.SingleExecution;
        return this;
    }

    public AstNodes visitDeclList(List<AstNodes> declList) {
        this.declList = declList;
        this.type = AstType.DeclList;
        return this;
    }

    public AstNodes visitLastDecl(List<AstNodes> declList) {
        this.declList = declList;
        this.type = AstType.LastDecl;
        return this;
    }

    public AstNodes visitFuncDeclareWithParam(String functionId, List<String> paramList, AstNodes functionExpr) {
        this.functionId = functionId;
        this.paramList = paramList;
        this.functionExpr = functionExpr;
        this.type = AstType.FuncDeclareWithParam;
        return this;
    }
```

```java
public AstNodes visitFuncDeclare(String functionId,  AstNodes functionExpr) {
    this.functionId = functionId;
    this.functionExpr = functionExpr;
    this.type = AstType.FuncDeclare;
    return this;
}

public AstNodes visitParamList(List<String> paramList) {
    this.paramList = paramList;
    this.type = AstType.ParamList;
    return this;
}


public AstNodes visitLastParam(List<String> paramList) {
    this.paramList = paramList;
    this.type = AstType.LastParam;
    return this;
}

public AstNodes visitProgramExpr(AstNodes programExpr) {
    this.programExpr = programExpr;
    this.type = AstType.ProgramExpr;
    return this;
}

public AstNodes visitFunctionExpr(AstNodes functionExpr) {
    this.functionExpr = functionExpr;
    this.type = AstType.FunctionExpr;
    return this;
}
public AstNodes visitVariable(String variableId) {
    this.id = variableId;
    this.type = AstType.Variable;
    return this;
}


public AstNodes visitNumber(Double value) {
    this.value = value;
    this.type = AstType.Number;
    return this;
}


public AstNodes visitCallFunctionWithParameter(String functionId, List<AstNodes> exprList) {
    this.type = AstType.CallFunctionWithParameter;
    this.functionId = functionId;
    this.exprList = exprList;
    return this;
}

public AstNodes visitOperation(AstNodes left, AstNodes right, AstType type) {
    this.left = left;
    this.right = right;
    this.type = type;
    return this;
}
public AstNodes visitCallFunction(String functionId) {
    this.functionId = functionId;
    this.type = AstType.CallFunction;
    return this;
}
```

```java
public AstNodes visitLetIn(String variableId, AstNodes x1, AstNodes x2) {
    this.id = variableId;
    this.x1 = x1;
    this.x2 = x2;
    this.type = AstType.LetIn;
    return this;
}

public AstNodes visitExprList(List<AstNodes> exprList) {
    this.exprList = exprList;
    this.type = AstType.ExprList;
    return this;
}

public AstNodes visitLastExpr(List<AstNodes> exprList) {
    this.exprList = exprList;
    this.type = AstType.LastExpr;
    return this;
}


public AstNodes visitInteger(double value) {
    this.value = value;
    this.type = AstType.Integer;
    return this;
}

public AstNodes visitFunctionID(String functionId) {
    this.id = functionId;
    this.type = AstType.FunctionID;
    return this;
}
public AstNodes visitParamID(String paramId) {
    this.paramId = paramId;
    this.type = AstType.ParamID;
    return this;
}
```

AstNodes 클래스의 메서드

- 위 메서드는 BuildAstVisitor 클래스에서 AstNodes 인스턴스 생성시 같이 호출되는 메서드이다. 생성자 역할을 한다.
- 생성자를 사용하지 않고 생성자 역할을 하는 메서드를 사용한 이유:
  생성자 오버로딩으로 대체하려고 했지만, 매개변수의 타입과 개수가 동일한 경우가 많이 존재하여 메서드 호출 방식으로 대체하였다.
  또한 Enum을 사용한 type을 통해 각 parser rule이 적용될 때 어떠한 rule이 사용되었는지 명시하도록 하였다.
- AstNodes 대신 Java의 상속을 이용한다면 각 클래스는 본인의 parser rule에 알맞은 필드만 가질 수 있고, 생성자를 사용할 수 있고 Enum을 통한 type을 명시하지 않아도 된다.
  그러나, 각 parser rule에 해당하는 클래스를 전부 만드는 것과, 이후 instanceof와 downcasting에 해당하는 노력과 시간이 많이 발생하므로, 메서드, type을 통한 방식을 사용하였다.

BuildAstVisitor.java

```java
public class BuildAstVisitor extends ExprBaseVisitor<AstNodes> {
    List<AstNodes> astNodesList = new ArrayList<>();

    public List<AstNodes> getAstNodesList() { return astNodesList; }

    @Override
    public AstNodes visitFunctionExecution(ExprParser.FunctionExecutionContext ctx) {
        AstNodes functionDecls = visit(ctx.getChild( i: 0));
        AstNodes programExpr = visit(ctx.getChild( i: 1));
        List<AstNodes> declList = new ArrayList<>(functionDecls.declList);
        AstNodes functionExecution = new AstNodes().visitFunctionExecution(declList, programExpr);
        astNodesList.add(functionExecution);
        return functionExecution;
    }

    @Override
    public AstNodes visitSingleExecution(ExprParser.SingleExecutionContext ctx) {
        AstNodes programExpr = visit(ctx.getChild( i: 0));
        AstNodes singleExecution = new AstNodes().visitSingleExecution(programExpr);
        astNodesList.add(singleExecution);
        return singleExecution;
    }

    @Override
    public AstNodes visitDeclList(ExprParser.DeclListContext ctx) {
        ArrayList<AstNodes> declList = new ArrayList<>();
        AstNodes firstDecl = visit(ctx.getChild( i: 0));
        AstNodes secondDecl = visit(ctx.getChild( i: 1));
        declList.add(firstDecl);
        declList.addAll(secondDecl.declList);
        return new AstNodes().visitDeclList(declList);
    }
    @Override
    public AstNodes visitLastDecl(ExprParser.LastDeclContext ctx) {
        AstNodes functionDecl = visit(ctx.getChild( i: 0));
        List<AstNodes> declList = new ArrayList<>();
        declList.add((functionDecl));
        return new AstNodes().visitLastDecl(declList);
    }

    @Override
    public AstNodes visitFuncDeclareWithParam(ExprParser.FuncDeclareWithParamContext ctx) {
        String functionID = visit(ctx.getChild( i: 1)).id;
        List<String> paramList = visit(ctx.getChild( i: 2)).paramList;
        AstNodes functionExpr = visit(ctx.getChild( i: 4)).functionExpr;
        return new AstNodes().visitFuncDeclareWithParam(functionID, paramList, functionExpr);
    }

    @Override
    public AstNodes visitFuncDeclare(ExprParser.FuncDeclareContext ctx) {
        String functionID = visit(ctx.getChild( i: 1)).id;
        AstNodes functionExpr = visit(ctx.getChild( i: 3)).functionExpr;
        return new AstNodes().visitFuncDeclare(functionID, functionExpr);
    }

    @Override
    public AstNodes visitParamList(ExprParser.ParamListContext ctx) {
        List<String> paramList = new ArrayList<>();
        AstNodes firstParam = visit(ctx.getChild( i: 0));
        AstNodes secondParam = visit(ctx.getChild( i: 1));
        paramList.add(firstParam.paramId);
        paramList.addAll(secondParam.paramList);
        return new AstNodes().visitParamList(paramList);
    }
```

```java
@Override
public AstNodes visitLastParam(ExprParser.LastParamContext ctx) {
    String paramId = visit(ctx.getChild( i: 0)).paramId;
    List<String> paramList = new ArrayList<>();
    paramList.add(paramId);
    return new AstNodes().visitLastParam(paramList);
}

@Override
public AstNodes visitProgramExpr(ExprParser.ProgramExprContext ctx) { return visit(ctx.getChild( i: 0)); }

@Override
public AstNodes visitFunctionExpr(ExprParser.FunctionExprContext ctx) {
    AstNodes functionExpr = visit(ctx.getChild( i: 0));
    return new AstNodes().visitFunctionExpr(functionExpr);
}

@Override
public AstNodes visitVariable(ExprParser.VariableContext ctx) {
    String variableId = ctx.getChild( i: 0).getText();
    return new AstNodes().visitVariable(variableId);
}

@Override
public AstNodes visitNumber(ExprParser.NumberContext ctx) {
    Double value = visit(ctx.getChild( i: 0)).value;
    return new AstNodes().visitNumber(value);
}
@Override
public AstNodes visitCallFunctionWithParameter(ExprParser.CallFunctionWithParameterContext ctx) {
    String functionId = ctx.getChild( i: 0).getText();
    List<AstNodes> exprList = visit(ctx.getChild( i: 2)).exprList;
    return new AstNodes().visitCallFunctionWithParameter(functionId, exprList);
}

@Override
public AstNodes visitOperation(ExprParser.OperationContext ctx) {
    AstNodes left = visit(ctx.getChild( i: 0));
    AstNodes right = visit(ctx.getChild( i: 2));
    String operator = ctx.getChild( i: 1).getText();
    return switch (operator) {
        case "*" -> new AstNodes().visitOperation(left, right, AstType.MUL);
        case "/" -> new AstNodes().visitOperation(left, right, AstType.DIV);
        case "+" -> new AstNodes().visitOperation(left, right, AstType.ADD);
        case "-" -> new AstNodes().visitOperation(left, right, AstType.SUB);
        default -> super.visitOperation(ctx);
    };
}

@Override
public AstNodes visitCallFunction(ExprParser.CallFunctionContext ctx) {
    String functionId = ctx.getChild( i: 0).getText();
    return new AstNodes().visitCallFunction(functionId);
}
@Override
public AstNodes visitLetIn(ExprParser.LetInContext ctx) {
    String id = ctx.getChild( i: 1).getText();
    AstNodes x1 = visit(ctx.getChild( i: 3));
    AstNodes x2 = visit(ctx.getChild( i: 5));
    return new AstNodes().visitLetIn(id, x1, x2);
}
```

```java
    @Override
    public AstNodes visitExprList(ExprParser.ExprListContext ctx) {
        ArrayList<AstNodes> exprList = new ArrayList<>();
        AstNodes firstExpr = visit(ctx.getChild( i: 0));
        AstNodes secondExpr = visit(ctx.getChild( i: 2));
        exprList.add(firstExpr);
        exprList.addAll(secondExpr.exprList);
        return new AstNodes().visitExprList(exprList);
    }

    @Override
    public AstNodes visitLastExpr(ExprParser.LastExprContext ctx) {
        AstNodes expr = visit(ctx.getChild( i: 0));
        List<AstNodes> exprList = new ArrayList<>();
        exprList.add(expr);
        return new AstNodes().visitLastExpr(exprList);
    }

    @Override
    public AstNodes visitInteger(ExprParser.IntegerContext ctx) {
        String numText = ctx.getChild( i: 0).getText();
        double num = Double.parseDouble(numText);
        return new AstNodes().visitInteger(num);
    }

    @Override
    public AstNodes visitFunctionID(ExprParser.FunctionIDContext ctx) {
        String functionId = ctx.getChild( i: 0).getText();
        return new AstNodes().visitFunctionID(functionId);
    }
    @Override
    public AstNodes visitParamID(ExprParser.ParamIDContext ctx) {
        String paramId = ctx.getChild( i: 0).getText();
        return new AstNodes().visitParamID(paramId);
    }
}
```

BuildAstVisitor 클래스의 메서드

- BuildAstVisitor 클래스는 ExprBaseVisitor를 상속한 클래스이다.
- 각 visit* 메서드에서는 ExprBaseVisitor의 메서드를 오버라이딩하였다.
- 각 visit* 메서드에서는 AstNodes 클래스에서 정의한 메서드를 사용하여 AstNodes 인스턴스를 생성하고 리턴한다.

메서드 설명

대부분 메서드는 본인의 production rule에 맞는 AstNodes를 생성하고 리턴하는 것이 전부이기에 몇몇 특수한 메서드만 설명하도록 한다. (Concisely explain)

```java
AstNodes visitFunctionExecution(ExprParser.FunctionExecutionContext ctx)
AstNodes visitSingleExecution(ExprParser.SingleExecutionContext ctx)
```

visitFunctionExecution, visitSingleExecution 메서드는 production rule에서 최상위 production rule에 속하는 statement에 해당하는 메서드이다. 따라서 해당 메서드에서 생성되는 AstNodes는 program을 실행하기 위한 모든 AstNodes(트리구조)를 포함하기 때문에 program class에 전달될 필요가 있다. 따라서 위 두 메서드에서는 본인의 production rule에 맞는 AstNodes를 생성할 뿐만 아니라 해당 AstNodes를 BuildAstVisitor의 멤버변수인 astNodesList에 저장한다. 이렇게 하여 getAstNodesList를 통해 생성한 AstNode를

program을 클래스에서 사용할 수 있도록 한다.

AstCall.java

AstCall 클래스는 Ast구조를 출력하기 위한 파일이다.
Ast구조를 String output이라는 멤버 변수에 담아서 리턴할 것이다.

전반적인 로직은 다음과 같다 :
트리 구조의 AstNodes를 자식 AstNodes로 나누고 자식 AstNodes가 처리할 수 있는 단위가 되면 output에 Ast 구조를 추가한다.
AstCall.java 파일은 Evaluate.java 와 매우 유사한 구조와 로직을 갖고 있으므로 상세한 설명은 Evaluate.java에서 설명한다.

```
public class AstCall {

    String output = "";

    public String buildAST(List<AstNodes> astNodesList) {
        for (AstNodes astNode : astNodesList) {
            switch (astNode.type) {
                case FunctionExecution -> processFunctionExecution(astNode);
                case SingleExecution -> processProgramExpr(astNode.programExpr, depth: 0);
            }
        }
        return output;
    }
```

최상위 parser rule에 해당하는 FunctionExecution, SingleExecution을 처리하는 메서드이다. 만약 어느 parer rule에 해당하는지 확인하는 것이 필요하다면 해당 AstNodes의 type을 확인한다.

같은 방식으로 process*메서드를 통해 아직 처리할 수 없는 Ast이면 자식 AstNodes로 분리하여 해당되는 process*메서드를 호출하고. 처리할 수 있는 Ast이면 output에 String을 추가한다.

```
    private void processFunctionExecution(AstNodes functionExecution) {
        List<AstNodes> functionList = functionExecution.declList;
        for (AstNodes functionDecl : functionList) {
            processFunctionDecl(functionDecl);
        }
        AstNodes programExpr = functionExecution.programExpr;
        processProgramExpr(programExpr, depth: 0);
    }

    private void processFunctionDecl(AstNodes functionDecl) {
        AstType type = functionDecl.type;
        switch (type) {
            case FuncDeclareWithParam -> processFuncDeclareWithParam(functionDecl);
            case FuncDeclare -> processFuncDeclare(functionDecl);
        }
    }
```

처리할 수 없는 메서드 -> process자식Ast 호출

```java
private void processFuncDeclareWithParam(AstNodes funcDeclareWithParam) {
    String functionId = funcDeclareWithParam.functionId;
    List<String> paramList = funcDeclareWithParam.paramList;
    AstNodes functionExpr = funcDeclareWithParam.functionExpr;

    output += "DECL\n";
    output += "\t" + functionId + "\n";
    for (String param : paramList) {
        output += "\t" + param + "\n";
    }
    processFuncionExpr(functionExpr, depth: 1);
}

private void processFuncDeclare(AstNodes funcDeclare) {
    String functionId = funcDeclare.functionId;
    AstNodes functionExpr = funcDeclare.functionExpr;

    output += "DECL\n";
    output += "\t" + functionId + "\n";
    processFuncionExpr(functionExpr, depth: 1);
}
```

처리할 수 있는 메서드 -> output에 Ast 구조 추가

처리할 수 있는 메서드는 다음과 같다

```java
private void processFuncionExpr(AstNodes functionExpr, int depth) {...}

private void processLetIn(AstNodes letIn, int depth) {...}

private void processCallFunctionWithParameter(AstNodes functionCall, int depth) {...}

private void processCallFunction(AstNodes functionCall, int depth) {...}

private void processProgramExpr(AstNodes expr, int depth) {...}
```

Evaluate.java

process*메서드를 통해 아직 처리할 수 없는 Ast이면 자식 AstNodes로 분리하여 해당되는 process*메서드를 호출하고, 처리할 수 있는 Ast이면 expr을 계산한다는 컨셉에서, AstCall.java와 비슷하다.

```java
Map<String, Double> variableStore = new HashMap<>();
Map<String, FunctionDefinition> functionStore = new HashMap<>();

public String getOutput() {
    return output;
}

String output = "";

public String buildAST(List<AstNodes> astNodesList) {
    for (AstNodes astNode : astNodesList) {
        switch (astNode.type) {
            case FunctionExecution -> processFunctionExecution(astNode);
            case SingleExecution -> {
                Double evaluatedValue = processProgramExpr(astNode.programExpr);
                output += evaluatedValue.toString();
            }
        }
    }
    return output;
}
```

Evaluate 클래스는 변수를 저장하는 variableStore와 함수의 정의를 저장하는 functionStore를 Map 자료구조로 갖고 있다. 그리고 expr의 계산 결과는 출력하기 위해 String output 멤버변수에 저장할 것이다.
output 멤버변수에 계산 결과를 저장하는 것은 buildAST 메서드의

```
output +=evaluatedValue.toString();
```

에 존재한다.

```
private void processFunctionExecution(AstNodes functionExecution) {
    List<AstNodes> functionList = functionExecution.declList;
    for (AstNodes functionDecl : functionList) {
        processFunctionDecl(functionDecl);
    }
    AstNodes programExpr = functionExecution.programExpr;
    Double evaluatedValue = processProgramExpr(programExpr);
    output += evaluatedValue.toString();
}
```

FunctionExecution에 해당하는 parser rule을 처리하는 메서드이다.
FunctionExecution은 여러 함수의 정의가 담겨 있는 declList를 순회하며 각 함수의 정의에 해당하는 functionDecl을 processFunctionDecl 메서드를 통해 처리한다.
그리고 함수 정의 후에 계산되어야 할 expression을 processProgramExpr 메소드를 호출하여 계산한다.
계산된 값은

```
output +=evaluatedValue.toString();
```

를 통해 output에 추가된다.

```
private void processFunctionDecl(AstNodes functionDecl) {
    AstType type = functionDecl.type;
    switch (type) {
        case FuncDeclareWithParam -> processFuncDeclareWithParam(functionDecl);
        case FuncDeclare -> processFuncDeclare(functionDecl);
    }
    output += "0.0\n";
}
```

processFunctionDecl 메서드
- 함수 정의는 두 타입으로 나뉠 수 있다.
  1. 파라미터가 존재하는 함수
  2. 파라미터가 없는 함수
- 두 함수 유형에 맞춰 processFuncDeclareWithParam 혹은 processFuncDeclare 메서드를 호출한다
- 함수 정의의 경우 무조건 계산값으로 0.0을 출력하도록 과제에 지시되어 있다.

```
private void processFuncDeclareWithParam(AstNodes funcDeclareWithParam) {
    String functionId = funcDeclareWithParam.functionId;
    List<String> paramList = funcDeclareWithParam.paramList;
    AstNodes functionExpr = funcDeclareWithParam.functionExpr;

    FunctionDefinition functionDefinition = new FunctionDefinition(paramList, functionExpr);
    functionStore.put(functionId, functionDefinition);

    processFunctionExpr(functionExpr, functionId);
}
```

processFuncDeclareWithParam 메서드

- 파라미터가 존재하는 함수를 처리하는 메서드이다.
- 함수의 이름, 파라미터 리스트, 함수의 body 부분을 나눈다.
- 함수의 파라미터 리스트와 body 부분을 FunctionDefinition 인스턴스로 생성한다. (FunctionDefinition 클래스는 이후에 설명하겠다.)
- 그리고 함수의 이름과 정의를 매핑하기 위한 functionStore에 이름을 key로, FunctionDefinition 인스턴스를 value로 저장한다.
- 이후 함수의 몸체 부분을 처리하기 위한 processsFunctionExpr 메서드를 호출한다.

```
private void processFuncDeclare(AstNodes funcDeclare) {
    String functionId = funcDeclare.functionId;
    AstNodes functionExpr = funcDeclare.functionExpr;

    FunctionDefinition functionDefinition = new FunctionDefinition(functionExpr);
    functionStore.put(functionId, functionDefinition);

    processFunctionExpr(functionExpr, functionId);
}
```

processFuncDeclare 메서드

- 파라미터가 없는 함수를 처리하는 메서드이다.
- 함수의 이름, 함수의 body 부분을 나눈다.
- 함수의 body 부분을 FunctionDefinition 인스턴스로 생성한다.
- 그리고 함수의 이름과 정의를 매핑하기 위한 functionStore에 이름을 key로, FunctionDefinition 인스턴스를 value로 저장한다.
- 이후 함수의 몸체 부분을 처리하기 위한 processsFunctionExpr 메서드를 호출한다.

```
private void processFunctionExpr(AstNodes functionExpr, String functionId) {
    AstType type = functionExpr.type;

    if (type == AstType.Variable) {
        String paramId = functionExpr.id;
        FunctionDefinition functionDefinition = functionStore.get(functionId);
        List<String> paramList = functionDefinition.paramList;
        if(paramList.contains(paramId)) return;
        System.out.println("Error: Free identifier " + paramId + " detected.");
        System.exit( status: 0);
    }

    switch (type) {
        case ADD, SUB, MUL, DIV -> {
            processFunctionExpr(functionExpr.left, functionId);
            processFunctionExpr(functionExpr.right, functionId);
        }
    }
}
```

processsFunctionExpr 메서드

- 함수의 몸체부분은 그 구조가 다른 expr과 그렇게 다르지 않다.
- 함수의 몸체 부분에서는 함수 몸체를 정의할 때 사용한 id가 free identifier인지 검사한다.
- 만약 free identifier라면, 즉시 Error message를 출력하고, 프로그램이 더 이상 진행될 필요가 없으므로 system.exit을 통해 프로그램을 종료한다

```
private Double processLetIn(AstNodes letIn) {
    String id = letIn.id;
    Double evaluatedId = processProgramExpr(letIn.x1);
    variableStore.put(id, evaluatedId);
    return processProgramExpr(letIn.x2);
}
```

processLetIn 메서드

- let in 구문을 처리하는 메서드이다
- let id = x1 in x2 라는 구문에서, x1을 계산해서 id에 매핑하고, varaibleStore에 그 정보를 저장한 다음, x2를 계산한 값이 let id = x1 in x2의 계산 결과이다.
- 마찬가지로, x1의 계산을 위해 processProgramExpr(letIn.x1);을 호출하고, 그 값과 id를 매핑하여 variableStore에 업데이트 한다
- 마지막으로 processProgramExpr(letIn.x1)의 값을 계산하여 리턴한다

```
private Double processCallFunctionWithParameter(AstNodes functionCall) {
    String functionId = functionCall.functionId;

    if (!functionStore.containsKey(functionId)) {
        System.out.println("Error: Undefined function " + functionId + " detected.");
        System.exit( status: 0);
    }

    FunctionDefinition functionDefinition = functionStore.get(functionId);
    AstNodes functionExpr = functionDefinition.functionExpr;

    List<String> paramList = functionDefinition.paramList;
    int paramSize = paramList.size();

    List<AstNodes> exprList = functionCall.exprList;
    int argSize = exprList.size();

    if (paramSize != argSize) {
        System.out.println("Error: The number of arguments of " + functionId + " mismatched, Required "
                + paramSize + ", Actual: " + argSize);
        System.exit( status: 0);
    }
    functionDefinition.clearArgMap();
    for(int i = 0; i < paramSize; i++) {
        String parameter = paramList.get(i);
        AstNodes expr = exprList.get(i);
        Double evaluatedExpr = processProgramExpr(expr);
        functionDefinition.mappingArg(parameter, evaluatedExpr);
    }
    Map<String, Double> argMap = functionDefinition.argMap;
    return processFunctionExpr(functionExpr, argMap);
}
```

processCallFunctionWithParamter 메서드

- 파라미터가 있는 함수가 program_expr에서 호출되었을 때, 그 함수를 처리하는 메서드이다.
- 먼저, 정의되지 않은 함수 호출이라면 즉시 에러 메시지를 출력하고 프로그램을 곧바로 종료한다.
- 마찬가지로, parameter 개수와 argument 개수를 비교하여 다르다면 즉시 에러 메시지를 출력하고 프로그램을 곧바로 종료한다.
- 함수가 이전에 사용되어 paramter와 argument가 매핑된 기록이 남아있을 수 있어 clearArgMap을 통해 매핑 정보를 초기화한다.
- 정상적인 argument가 함수 호출시 사용되었다면, 먼저 argument를 하나씩 계산한 다음, functionDefinition.mappingArg를 통해 함수 몸체의 parameter와 계산된 argument를 매핑한다. 이를 통해 함수 몸체 내부에서 parameter를 활용한 식이 있다면 argument로 치환되어 해당 식이 계산될 수 있다.

- 마지막으로 processFunctionExpr을 통해 함수 몸체를 계산한 값을 리턴한다

```
private Double processFunctionExpr(AstNodes functionExpr, Map<String, Double> argMap) {
    AstType type = functionExpr.type;

    if (type == AstType.Number) return functionExpr.value;

    if (type == AstType.Variable) {
        String id = functionExpr.id;
        return argMap.get(id);
    }
    if (type == AstType.LetIn) return processFunctionExpr(functionExpr, argMap);
    if (type == AstType.CallFunctionWithParameter) return processFunctionExpr(functionExpr, argMap);
    if (type == AstType.CallFunction) return processFunctionExpr(functionExpr, argMap);

    switch (type) {
        case ADD -> {
            return processFunctionExpr(functionExpr.left, argMap) + processFunctionExpr(functionExpr.right, argMap);
        }

        case SUB -> {
            return processFunctionExpr(functionExpr.left, argMap) - processFunctionExpr(functionExpr.right, argMap);
        }

        case MUL -> {
            return processFunctionExpr(functionExpr.left, argMap) * processFunctionExpr(functionExpr.right, argMap);
        }

        case DIV -> {
            return processFunctionExpr(functionExpr.left, argMap) / processFunctionExpr(functionExpr.right, argMap);
        }
    }
    return -999.0;
}
```

processFunctionExpr 메서드

- 함수 몸체를 계산하는 메서드이다.
- 만약 함수 몸체 내부에서 파라미터를 사용한다면, 매핑된 argument의 값을 사용한다
- 나머지 계산하는 방법은 일반적인 expr 계산과 동일하다.

```
private Double processCallFunction(AstNodes functionCall) {
    String functionId = functionCall.functionId;
    if (!functionStore.containsKey(functionId)) {
        System.out.println("Error: Undefined function " + functionId + " detected.");
        System.exit( status: 0);
    }
    FunctionDefinition functionDefinition = functionStore.get(functionId);
    int paramSize = functionDefinition.paramList.size();
    if(paramSize != 0) {
        System.out.println("Error: The number of arguments of " + functionId + " mismatched, Required: "
                + paramSize + ", Actual: 0");
        System.exit( status: 0);
    }
    AstNodes functionExpr = functionDefinition.functionExpr;
    Double value = processProgramExpr(functionExpr);
    return value;
}
```

processCallFunction 메서드

- 파라미터가 없는 함수를 처리하는 메서드이다

- 먼저, 정의되지 않은 함수 호출이라면 즉시 에러 메시지를 출력하고 프로그램을 곧바로 종료한다.
- 마찬가지로, 파라미터가 없는 함수인데 대입된 argument가 있따면 즉시 에러 메시지를 출력하고 프로그램을 곧바로 종료한다.
- 파라미터가 없는 함수이므로 매핑할 argument가 없으므로 processFunctionExpr을 통해 함수 몸체를 계산한 값을 리턴한다

```
private Double processProgramExpr(AstNodes expr) {
    AstType type = expr.type;

    if (type == AstType.Number) return expr.value;

    if (type == AstType.Variable) {
        String id = expr.id;
        if (variableStore.containsKey(id)) return variableStore.get(id);
        System.out.println("Error: Free identifier " + id + " detected.");
        System.exit( status: 0);
    }
    if (type == AstType.LetIn) return processLetIn(expr);
    if (type == AstType.CallFunctionWithParameter) return processCallFunctionWithParameter(expr);
    if (type == AstType.CallFunction) return processCallFunction(expr);

    switch (type) {
        case ADD -> {
            return processProgramExpr(expr.left) + processProgramExpr(expr.right);
        }

        case SUB -> {
            return processProgramExpr(expr.left) - processProgramExpr(expr.right);
        }

        case MUL -> {
            return processProgramExpr(expr.left) * processProgramExpr(expr.right);
        }

        case DIV -> {
            return processProgramExpr(expr.left) / processProgramExpr(expr.right);
        }
    }
    return -999.0;
```

- program_expr을 계산하는 메서드이다
- variableStore를 사용해 정의되지 않은 variable이 사용되는 경우 에러 메시지를 출력하고 프로그램을 종료한다.
- 정의된 varaible인 경우 해당 variable의 값을 variableStore에서 꺼내와 사용한다.
- Let in, 함수 호출인 경우 해당 파서 룰에 맞는 메서드를 호출하여 처리한다.
- ADD, SUB, MUL, DIV와 같은 operation인 경우 left, right를 통해 재귀적으로 처리한다.

FunctionDefinition.java

```java
public class FunctionDefinition {
    List<String> paramList;
    AstNodes functionExpr;

    Map<String, Double> argMap = new HashMap<>();

    String type;

    public FunctionDefinition(List<String> paramList, AstNodes functionExpr) {
        this.paramList = paramList;
        this.functionExpr = functionExpr;
        this.type = "FunctionWithParam";
    }

    public FunctionDefinition(AstNodes functionExpr) {
        this.paramList = new ArrayList<>();
        this.functionExpr = functionExpr;
        this.type = "FunctionNoParam";
    }

    public void mappingArg(String id, Double value) { argMap.put(id, value); }

    public void clearArgMap() {
        argMap.clear();
    }
}
```

- FunctionDefinition 클래스를 정의한다
- paramList를 생성자 인수로 받는 생성자의 경우 파라미터가 있는 함수의 정의에 해당하는 생성자이다
- fuctionExpr만 받는 생성자의 경우 파라미터가 없는 함수의 정의에 해당하는 생성자이다.
- mappingArg 메서드의 경우 program_expr에서 함수를 호출할 때, argument를 전달하는 경우 해당 argument와 paramter를 매핑하기 위한 메서드이다.
- clearArgMap 메서드의 경우, 함수를 호출하기 전에 호출되어 argument와 paramter를 매핑한 정보를 삭제한다.

AstType.java

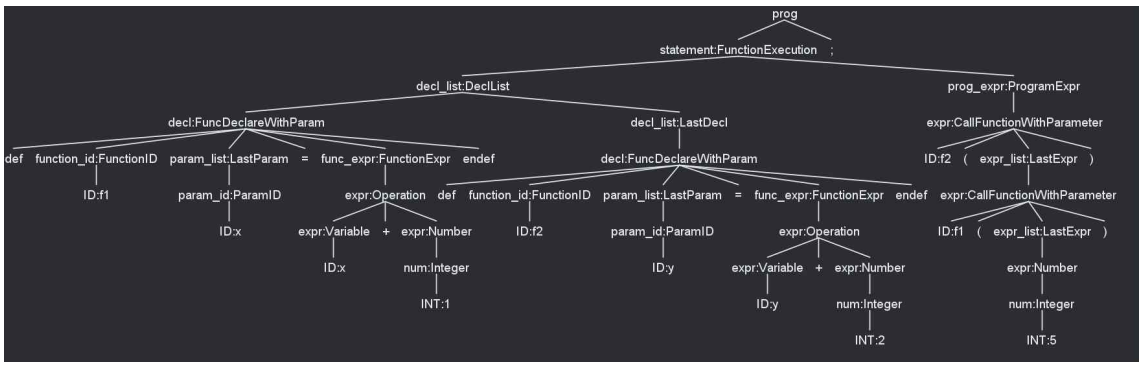
```java
public enum AstType {
    FunctionExecution, SingleExecution,
    DeclList, LastDecl,
    FuncDeclareWithParam, FuncDeclare,
    ParamList, LastParam,

    ProgramExpr, FunctionExpr,
    Parenthesis, LetIn, CallFunction, CallFunctionWithParameter,
    Operation, Number, Variable,
    ADD, SUB, MUL, DIV,
    ExprList, LastExpr,
    Integer, FunctionID, ParamID
}
```

enum에 해당하는 파일이며, AstNodes의 타입을 정의하기 위한 파일이다.

My experience comparing implementation on OCaml and ANTLR

OCaml에 비해서 ANTRLR의 구현 방식은 상당히 복잡하고 그 양이 많았다.
가장 큰 이유는 다음과 같다.
1. OCaml은 재귀적으로 코드를 작성하는 것이 일반적이고 쉽다.
내가 구현한 Expr.g4 파일을 바탕으로
def f1 x = x + 1 endef def f2 y = y + 2 endef f2(f1(5));
에 대해서 AST tree를 표현하면 다음과 같다.



위 사진에서 볼 수 있듯이 트리구조로 이루어져있으며, 반복되는 패턴이 많이 등장한다.
Ocaml에서는 재귀적으로 위 트리구조를 효과적으로 처리할 수 있다.
그러나 내가 JAVA와 ANTLR를 사용해 인터프리터를 구현했을 때는 재귀적으로 처리하는 것이 쉽지 않았다. 가장 큰 이유는 타입이 너무나 다양했기 때문이다.
다양한 타입에 대해서 재귀적으로 프로그래밍을 하게 되는 경우 하나의 재귀 함수가 크고 복잡해지는 문제가 발생했다. 일반적으로 재귀함수는 간단한 구현을 통해 효과적인 코딩을 하는 것이 목적인데 재귀함수가 타입에 따라 케이스가 많아지고 복잡해져서 재귀함수를 통한 구현을 포기하고, 각 타입에 맞는 메서드를 구현하는 것으로 대신하였다.
결론적으로, 재귀적으로 프로그래밍하지 않았던 java에서 훨씬 코드를 작성하는 것이 복잡하였다.

2. OCaml의 패턴 매칭을 통한 강력한 타입 매칭
나는 이번 과제에서 AstNodes 하나로 각 AST를 구현하였다. 상속이나 인터페이스의 구현을 통해 해결하려고도 시도했지만 무엇을 사용하든 각 클래스를 따로 정의해야 하며 자식 클래스를 확인하기 위한 instanceof, 다형성을 사용하는 과정에서 발생하는 다운캐스팅과 같은 추가적인 작업이 발생하여 하나의 클래스를 사용하였다. 이 부작용으로, 같은 AstNodes일지라도 어떤 parser rule에 해당하는지 확인하기 위한 type 필드를 추가적으로 부여하게 되었다.
결국 많은 메서드에서 type 필드를 부여하거나 체크하는 일이 많아져 과제의 구현 복잡성이 증가하였다.
그러나 OCaml에서는 이러한 작업이 필요하지 않았는데, 가장 큰 이유는 패턴 매칭이었다.
패턴 매칭을 통해 각 AST를 효과적으로 처리할 수 있기 때문에, 내가 이번 과제에서 겪었던 복잡성이 많이 줄어들었다.

따라서, 이번 과제를 수행하면서 OCaml의 언어적 특성이 인터프리터를 구현하는 데 많은 장점이 있음을 알게 되었다.