ANTLR-F1VAE

Expr.g4

```
grammar Expr;

// parser rules
prog : declList (expr ';' NEWLINE?)*
     | (expr ';' NEWLINE?)* ;

declList: decl+ ;

decl: 'def' VAR varList '=' expr 'endef'  #funDecl
   | 'def' VAR '=' expr 'endef'     #funDecl
   ;

varList : VAR+ ;

expr : expr ('*' | '/') expr    #infixExpr
   | expr ('+' | '-') expr    #infixExpr
     | 'let' VAR '=' expr 'in' expr #letExpr
     | '(' expr ')'             #parenExpr
     | '~' '(' expr')'          #negExpr
     | VAR '=' expr         #varDeclExpr
     | VAR '()'             #funCallExpr
     | VAR '(' exprList ')'     #funCallExpr
     | num              #numExpr
     | VAR              #varExpr
     ;

exprList : expr (',' expr)* ;

num  : INT
     | REAL ;

// lexer rules
```

- program은 함수 선언부와 Expr부로 나뉜다. Decl은 각각의 함수 선언인데, 두 개의 concrete syntax를 하나의 funDecl로 묶었다.
- Function Call 역시 두 개의 syntax를 하나의 funCallExpr로 묶었다.

AstNodes.java

```
public class AstNodes{}

class ProgNode extends AstNodes {
  public List<AstNodes>roots = new ArrayList<AstNodes>();

  ProgNode(List<AstNodes>roots){
    this.roots = roots;
  };
} //list of decl and expr
```

- 하나의 객체로 모든 노드를 관리하기에는 이제는 불필요하게 크기 때문에, 모든 Node들은 기본적으로 빈 AstNode 클래스를 상속한다.
- ProgNode는 각각의 Decl과 Expr를 리스트 형태로 가진다.

```
class funDeclNode extends AstNodes {
  String name;
  AstNodes body;
  List<String>param;
  funDeclNode(String name,AstNodes body, List<String> param){
    this.name = name;
    this.body = body;
```

```
    this.param = param;
  }
}
```

- 모든 함수 선언이 가지는 funDeclNode.

```
class funCallNode extends AstNodes {
  String name;
  List<AstNodes> param;
  funCallNode(String name,List<AstNodes> param){
    this.name = name;
    this.param = param;
  }
}
```

- 모든 함수 호출이 가지는 funCallNode.

```
class LetNode extends AstNodes{
  String var;
  AstNodes body;
  AstNodes expr;
  LetNode(String var, AstNodes body, AstNodes expr){
    this.var = var;
    this.body = body;
    this.expr = expr;
  }
}
```

BuildAstVisitor.java

```
public class BuildAstVisitor extends ExprBaseVisitor<AstNodes>{

  @Override
  public AstNodes visitProg(ExprParser.ProgContext ctx) {

    List<AstNodes>roots = new ArrayList<AstNodes>();
    if(ctx.declList() != null) {
      List<AstNodes>declList = ((declListNode)visit(ctx.declList())).declList;
      for(AstNodes node: declList) {
        roots.add(node);
      }
    }
    for(int i=0;i<ctx.expr().size();i++) {
      roots.add(visit(ctx.expr(i)));
    }
    return new ProgNode(roots);
  }

  @Override
  public AstNodes visitFunDecl(ExprParser.FunDeclContext ctx) {
    String name = ctx.getChild(1).getText();
    AstNodes body = visit(ctx.expr());
    List<String> param = null;
    if(ctx.varList() != null) {
      param = ((varListNode)visit(ctx.varList())).varList;
    }
    return new funDeclNode(name, body, param);
  }

  @Override
  public AstNodes visitInfixExpr(ExprParser.InfixExprContext ctx) {
    String op = "";
    switch(ctx.getChild(1).getText()) {
    case "+":
      op = "ADD";
      break;
    case "-":
      op = "SUB";
      break;
```

```
    case "*":
      op = "MUL";
      break;
    case "/":
      op = "DIV";
      break;
    }
    return new InfixNode( visit(ctx.getChild(0)) ,visit(ctx.getChild(2)),op);
  }


  @Override
  public AstNodes visitFunCallExpr(ExprParser.FunCallExprContext ctx) {
    List<AstNodes> param = null;
    if(ctx.exprList() != null) {
      param = ((exprListNode)visit(ctx.exprList())).exprList;
    }
    return new funCallNode(ctx.getChild(0).getText(), param);
  }

  @Override
  public AstNodes visitLetExpr(ExprParser.LetExprContext ctx) {
    return new LetNode(ctx.getChild(1).getText(),visit(ctx.getChild(3)),visit(ctx.getChild(5)));
  }

...
```

- 각각의 context를 파싱하여 적합한 노드를 생성한다.
- visitFunDecl은 인자로 받은 parameter 리스트 널 체크 후 널 또는 인자에 대한 노드 리스트를 생성한다.

Evaluate.java

```
public class Evaluate {
  //Define methods to calculate the expression we get as input
  //The name of the method should be "evaluate"
    static  Map<String, Double> store = new HashMap<>();
    static  Map<String, funDeclNode> fstore = new HashMap<>();
    static  List<String>errors = new ArrayList<>();

  public double evaluate(AstNodes node) {

    else if(node instanceof funDeclNode) {
      funDeclNode Node = (funDeclNode)node;

      String name = Node.name;
      fstore.put(name,Node);

      Map<String, Double> backup = new HashMap<>(store);
      Map<String, Double> closure = new HashMap<>(store);
      if( Node.param != null){
      for(String p: Node.param) {
        closure.put(p, 0.0);
      }
}
      store = closure;
      evaluate(Node.body);
      store = backup;
    }
```

- 변수 저장을 위한 store, 함수 저장을 위한 fstore, 에러 메시지 저장을 위한 errors는 static으로 Evaluate 클래스에 저장했다. 단일한 저장소를 관리하기 위함이다.
- body에서의 변수들이 parameter에 존재하는 변수들을 제외하고 free identifier 검사를 해야 하므로 새로운 closure 환경 안에서 evaluate한다.
- 임시적으로 free identifier 검사를 한 후 다시 store를 복구시킨다.

```
else if(node instanceof funCallNode) {
  funCallNode Node = (funCallNode)node;
  String functionName = Node.name;
  List<AstNodes>param = Node.param;
  if(param != null){
    for(AstNodes n: param){
      evaluate(n);
    }
  }

  if(!fstore.containsKey(functionName)) {
    errors.add("Undefined function "+functionName+" detected.");
    return 0.0;
  }

  List<String>fparam = fstore.get(functionName).param;
  AstNodes functionBody = fstore.get(functionName).body;

  if(fparam != null && param ==null){

    errors.add("The number of arguments of "+functionName+" mismatched, Required: "+fparam.size()+", Actual: 0");
      return 0.0;
  }

  if(fparam == null && param !=null){

    errors.add("The number of arguments of "+functionName+" mismatched, Required: 0, Actual: "+param.size());
      return 0.0;
  }
  if(fparam != null && param != null){
    if(fparam.size() != param.size()){

    errors.add("The number of arguments of "+functionName+" mismatched, Required: "+fparam.size()+", Actual: "+param.size());
      return 0.0;
    }
  }

  Map<String, Double> backup = new HashMap<>(store);
  Map<String, Double> closure = new HashMap<>(store);

  if(fparam != null) {
    for(int i=0;i<fparam.size();i++) {
      String key = fparam.get(i);
      double val = evaluate(param.get(i));
      closure.put(key, val);
    }
  }
  store = closure;
  double result = evaluate(functionBody);
  store = backup;
  return result;
}
```

- 가장 먼저 인자로 들어온 AstNodes의 유효성 검사를 한 후 인자 개수 검사를 한다.
- body 평가는 역시 closure 안에서 해야 하기 때문에 새로 만들어 준 store 안에 들어온 인자를 대입하여 저장한 후, 해당 환경 안에서 evaluate한다. 그 후 store을 복구시킨다.

```
else if(node instanceof LetNode) {
  LetNode Node = (LetNode)node;
  Double backup = store.get(Node.var);

  String key = Node.var;
  double val = evaluate(Node.body);
  store.put(key,val);
  double result = evaluate(Node.expr);
  if(backup != null) {
    store.put(Node.var, backup);
  }
  else{
    store.remove(Node.var);
```

```
      }
      return result;
    }

    else if(node instanceof VARNode) {
       VARNode Node = (VARNode)node;
      if(!store.containsKey(Node.ID)) {
        errors.add("Free identifier "+ Node.ID+" detected");
        return 0.0;
      }
      return store.get(Node.ID);

    }
    return 0.0;


  }
}
```

- LetNode에서는 새로 선언된 문자와 evaluate된 double을 넣어준다. 이 body는 현재 온전히 evaluate될 수 있어야 한다.
- VARNode evaluate 시 Free identifier 를 체크한다.

AstCall.java

```
public class AstCall {

  public static void Call(AstNodes node, int depth) {
    Evaluate Evaluator = new Evaluate();
    for(int i=0;i<depth;i++) {System.out.print("\t");}

    if(node instanceof InfixNode) {
      String op = ((InfixNode) node).op;
      AstNodes LHS = ((InfixNode) node).LHS;
      AstNodes RHS = ((InfixNode) node).RHS;
      System.out.printf("%s\n",op);

      Call(LHS,++depth);
      Call(RHS,depth--);

    }
    else if(node instanceof funDeclNode) {
      funDeclNode Node = (funDeclNode)node;
      System.out.print("DECL\n");

      Call(new VARNode(Node.name),++depth);
      if(Node.param != null) {
        for(String s : Node.param) {
          Call(new VARNode(s),depth);
        }
      }
      Call(Node.body,depth--);

    }

    else if(node instanceof funCallNode) {
      funCallNode Node = (funCallNode)node;
      System.out.print("CAll\n");
      Call(new VARNode(Node.name),depth+1);
      if(Node.param != null) {
        for(AstNodes s :Node.param) {
          Call(s,depth+1);
        }
      }
    }

    else if(node instanceof LetNode) {
      LetNode Node = (LetNode)node;
      System.out.print("LETIN\n");
      Call(new VARNode(Node.var),++depth);
      Call(Node.body,depth);
      Call(Node.expr,depth--);

    }
```

```
    else if(node instanceof NegNode) {
      AstNodes body = ((NegNode) node).body;
      System.out.print("NEGATE\n");
      Call(body,depth+1);
    }
    else if(node instanceof ParenNode) {
      AstNodes body = ((ParenNode) node).body;
      Call(body,depth);
    }
    else if(node instanceof VARNode) {
      VARNode Node = (VARNode)node;
      System.out.printf("%s\n",Node.ID);
    }
    else if(node instanceof NUMNode) {
      System.out.printf("%.1f\n", Evaluator.evaluate(node));
    }

    else if(node instanceof varDeclNode) {
      varDeclNode Node = (varDeclNode)node;
      System.out.print("ASSIGN\n");
      Call(new VARNode(Node.name),++depth);
      Call(Node.body,depth--);

    }
    else return;
  }


}
```

- depth를 전달하는 방식으로 indent를 구현한다.