antlr error handling

1. `Expr.g4` (ANTLR Grammar)

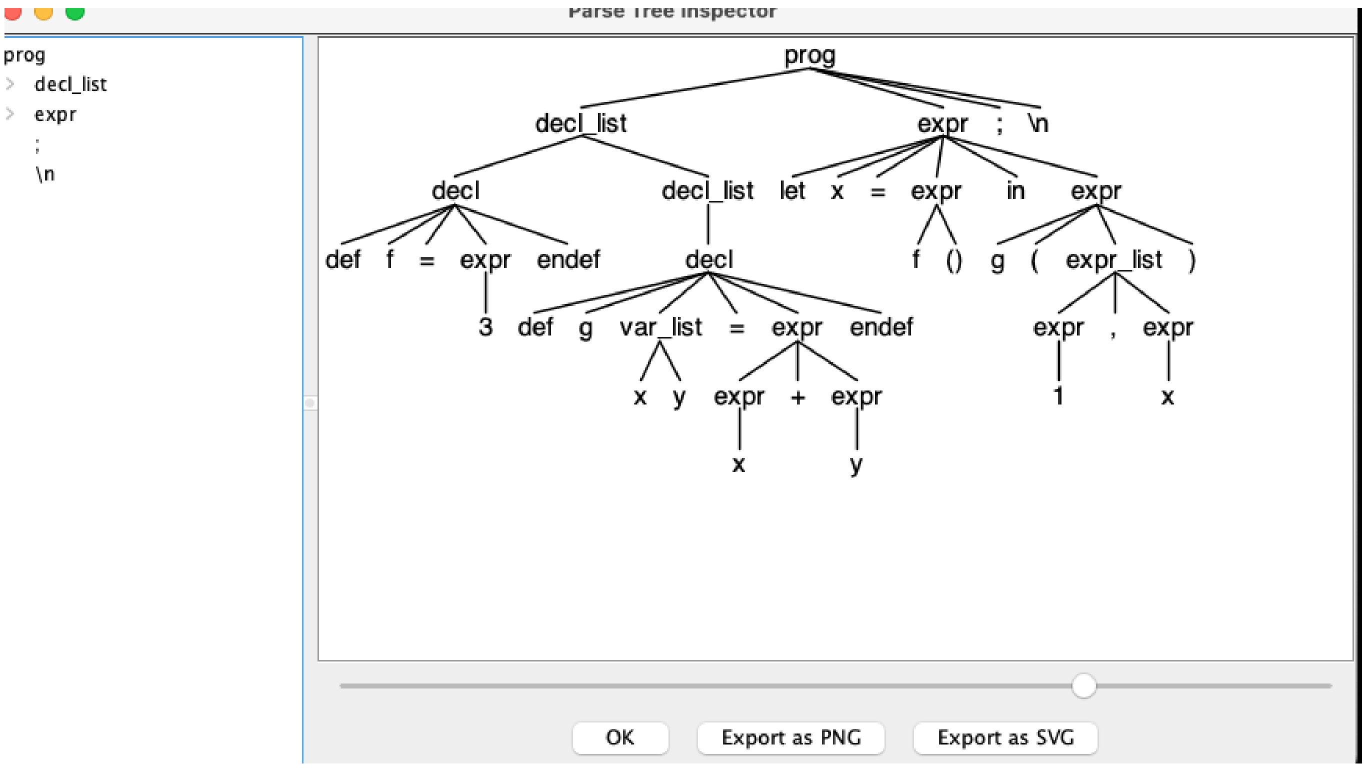
Modify the grammar to include the new syntax elements of F1VAE. This includes support for function definitions (`def`), function calls, and the `let` expression.

```
grammar Expr;

@header {

package antlr; // import antlr package (every generated java file)

}

// parser rules

prog: (decl_list expr | expr) (';' (decl_list expr | expr))* ';' NEWLINE?;

decl_list: decl+;

decl: 'def' ID var_list '=' expr 'endef' # functionDecl
| 'def' ID '=' expr 'endef' # simpleFunctionDecl;

var_list: ID+ ;

expr: ID '()' # functionCallNoArg
        | ID '(' expr_list ')' # functionCall
        | expr op=('*'|'/') expr # infixExpr
        | expr op=('+'|'-') expr # infixExpr
        | 'let' ID '=' expr 'in' expr # letExpr
        | ID op='=' expr # assignExpr
        | num # numberExpr
        | '(' expr ')' # parensExpr
        | '~' '(' expr ')' # negationExpr
        | ID # idExpr;

expr_list: expr (',' expr)* ;

num : NUMBER ;

// lexer rules
```

```
NUMBER : '-'? INT | '-'? REAL ;
INT: [0-9]+ ;
REAL: [0-9]+'.'[0-9]* ;
ID: [a-zA-Z0-9_\-]+; //
NEWLINE: [\r\n]+ ;
WS: [ \t\r\n]+ -> skip ;
```

This file defines the grammar of the F1VAE language. It includes rules for parsing programs (`prog`), declarations (`decl_list` and `decl`), expressions (`expr`), and their components. Key additions are the syntax for function definitions (`def` ... `endef`), function calls (with and without arguments), let expressions, and various operations like negation (`~`), infix operations (`+`, `-`, `*`, `/`), and assignment (`=`). The lexer rules define the format for numbers, identifiers, and whitespace, adhering to the language's lexical structure.

My approach to defining the grammar for function declarations with `functionDecl` and `simpleFunctionDecl` is rational and a common practice in language design and parsing. By labeling these two types of declarations separately, we can provide a clear distinction between functions with parameters and functions without parameters. This distinction will simplify the parsing and interpretation process in your `BuildAstVisitor.java`.



```
grun antlr.Expr prog -gui
def f = 3 endef def g x y = x + y endef let x = f() in g(1,x);
^D
```

It works really well.

2. `AstNodes.java` Modifications

```
package expression;

import java.util.ArrayList;
import java.util.List;


class AstNodes {}

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


        public void addExpression(AstNodes e) {
                if(e != null)
                        expressions.add(e);

                }



}


class FunctionDeclNode extends AstNodes {
    String functionName;
    List<String> parameters = new ArrayList<>();
    AstNodes body;

    FunctionDeclNode(String functionName, List<String> parameters, AstNodes
body) {
        this.functionName = functionName;
        this.parameters = parameters;
        this.body = body;
    }
}


class FunctionCallNode extends AstNodes {
    String functionName;
    List<AstNodes> arguments = new ArrayList<>();

    FunctionCallNode(String functionName, List<AstNodes> arguments) {
        this.functionName = functionName;
        this.arguments = arguments;
    }
}

class LetNode extends AstNodes {
    String variableName;
    AstNodes assignedExpr;
    AstNodes inExpr;

    LetNode(String variableName, AstNodes assignedExpr, AstNodes inExpr) {
        this.variableName = variableName;
        this.assignedExpr = assignedExpr;
        this.inExpr = inExpr;
    }
}



class InfixNode extends AstNodes {
    String op; // e.g. ADD, SUB, MUL, DIV
    AstNodes left, right;
}

class NegNode extends AstNodes{
        String op; // eg. NEG
        AstNodes expr;
}

class AssignNode extends AstNodes {
        String idName;
    String op; //ASSIGN
    AstNodes right;
}

class NumberNode extends AstNodes {
    double value;
}

class IdNode extends AstNodes {
    String IdName;
}
```

This file contains classes representing different types of abstract syntax tree (AST) nodes, reflecting the elements of the F1VAE language. It includes `ProgNode` for programs, `FunctionDeclNode` for function declarations, `FunctionCallNode` for function calls, `LetNode` for let expressions, and various other nodes for representing operations (infix, negation, assignment), numbers, and identifiers. Each class is structured to hold relevant data for its respective language element, like function names, parameters, expressions, and operators.

3. `BuildAstVisitor.java` Modifications

```
package expression;

import antlr.ExprBaseVisitor;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

import org.antlr.v4.runtime.tree.ParseTree;

import antlr.ExprParser;
import antlr.ExprParser.AssignExprContext;
import antlr.ExprParser.FunctionCallContext;
import antlr.ExprParser.FunctionCallNoArgContext;
import antlr.ExprParser.FunctionDeclContext;
import antlr.ExprParser.IdExprContext;
import antlr.ExprParser.InfixExprContext;
import antlr.ExprParser.LetExprContext;
import antlr.ExprParser.NegationExprContext;
import antlr.ExprParser.NumberExprContext;
import antlr.ExprParser.ParensExprContext;
import antlr.ExprParser.ProgContext;
import antlr.ExprParser.SimpleFunctionDeclContext;



public class BuildAstVisitor extends ExprBaseVisitor<AstNodes> {

        @Override
        public AstNodes visitProg(ProgContext ctx) {

                ProgNode progNode = new ProgNode();
            int max= ctx.getChildCount()-2;
                for(int i=0; i< ctx.getChildCount()-2; i++ ) {
                        //because of ctrl + d
                    if (ctx.getChild(i) instanceof Decl_listContext) {
                    // Visit each decl in the decl_list and add to
expressions
```

```
                        Decl_listContext declListCtx = (Decl_listContext)
ctx.getChild(i);
                        declListCtx.decl().forEach(declCtx ->
progNode.addExpression(visit(declCtx)));
                        } else if (ctx.getChild(i) instanceof ExprContext) {
                           // It's a single expression
                           ExprContext exprCtx = (ExprContext)
ctx.getChild(i);
                           progNode.addExpression(visit(exprCtx));
                        }

                }
                return progNode;
        }

        @Override
        public AstNodes visitInfixExpr(InfixExprContext ctx) {
                InfixNode infixNode = new InfixNode();
                infixNode.left = visit(ctx.expr(0));
                infixNode.right = visit(ctx.expr(1));
                infixNode.op = ctx.getChild(1).getText();  // the operator
is in the middle
                switch(infixNode.op) {
        case "+":
                infixNode.op = "ADD";
                break;
        case "-":
                infixNode.op = "SUB";
                break;
        case "*":
                infixNode.op = "MUL";
                break;
        default:
                infixNode.op = "DIV";
                break;
        }

        return infixNode;
        }

        @Override
        public AstNodes visitFunctionDecl(FunctionDeclContext ctx) {
                String functionName = ctx.ID().getText(); // First ID is the
function name
                List<String> parameters = new ArrayList<>();
                if (ctx.var_list() != null) {
```

```
            parameters = ctx.var_list().ID()
                    .stream().map(ParseTree::getText)
                    .collect(Collectors.toList());
        }
                 AstNodes body = visit(ctx.expr());
                 return new FunctionDeclNode(functionName, parameters,
body);

        }

        @Override
        public AstNodes visitSimpleFunctionDecl(SimpleFunctionDeclContext
ctx) {
                String functionName = ctx.ID().getText();
//              System.out.print(functionName);
                AstNodes body = visit(ctx.expr());
                return new FunctionDeclNode(functionName, new ArrayList<>(),
body);

        }

        @Override
        public AstNodes visitFunctionCallNoArg(FunctionCallNoArgContext ctx)
{
                return new FunctionCallNode(ctx.ID().getText(), new
ArrayList<>());
        }

        @Override
        public AstNodes visitFunctionCall(FunctionCallContext ctx) {
                String functionName = ctx.ID().getText();
                List<AstNodes> arguments = ctx.expr_list().expr()
                .stream().map(this::visit)
                .collect(Collectors.toList());
                return new FunctionCallNode(functionName, arguments);

        }

        @Override
        public AstNodes visitNegationExpr(NegationExprContext ctx) {
                NegNode negNode = new NegNode();
                negNode.op = "NEGATE";
                negNode.expr = visit(ctx.expr());
                return negNode;

        }
        @Override
        public AstNodes visitLetExpr(LetExprContext ctx) {
                String variableName = ctx.ID().getText();
                AstNodes assignedExpr = visit(ctx.expr(0));
                AstNodes inExpr = visit(ctx.expr(1));
                return new LetNode(variableName, assignedExpr, inExpr);

        }

        @Override
        public AstNodes visitNumberExpr(NumberExprContext ctx) {
                NumberNode numberNode = new NumberNode();
        numberNode.value = Double.parseDouble(ctx.getText());
        return numberNode;
        }

        @Override
        public AstNodes visitParensExpr(ParensExprContext ctx) {
                // Just return the inner expression node directly
            return visit(ctx.expr());
        }

        @Override
        public AstNodes visitAssignExpr(AssignExprContext ctx) {

                AssignNode assignNode = new AssignNode();
            assignNode.idName = ctx.getChild(0).getText();
            assignNode.op = "ASSIGN";
            assignNode.right = visit(ctx.expr());
            return assignNode;
        }



        @Override
        public AstNodes visitIdExpr(IdExprContext ctx) {
            IdNode idNode = new IdNode();
        idNode.IdName = ctx.getText();
        return idNode;
        }



}
```

explanation of visitfuntiondecl

This file extends the `ExprBaseVisitor` class, implementing methods to visit different parse tree nodes and construct the corresponding AST nodes. It handles the construction of program nodes (`ProgNode`), function declaration nodes (`FunctionDeclNode`), function call nodes (`FunctionCallNode`), let nodes (`LetNode`), and nodes for various expressions (infix, negation, numbers, identifiers, etc.). The visitor methods extract relevant information from the parse tree and use it to create and populate the appropriate AST nodes.

4. `Evaluate.java` Modifications

```
package expression;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Evaluate {

    private Map<String, Double> variables = new HashMap<>();
    private Map<String, FunctionDeclNode> functions = new HashMap<>();
    public List<String> semanticErrors = new ArrayList<>();

    public double evaluate(AstNodes node) {

        if (node instanceof FunctionDeclNode) {
            // Function declaration doesn't produce a value in this context,
just registers the function.
            FunctionDeclNode funcDecl = (FunctionDeclNode) node;
            functions.put(funcDecl.functionName, funcDecl);
            return 0.0;  // Return 0.0 for the function declaration itself.

        } else if (node instanceof LetNode) {
            LetNode letNode = (LetNode) node;
            double value = evaluate(letNode.assignedExpr);
            // Temporarily store the current value of the variable if it
exists to restore it later.
            Double oldValue = variables.get(letNode.variableName);
            variables.put(letNode.variableName, value);
            double result = evaluate(letNode.inExpr);
            // Restore the old value if it existed.
            if (oldValue != null) {
                variables.put(letNode.variableName, oldValue);
            } else {
                variables.remove(letNode.variableName);
            }
            return result;

        } else if (node instanceof NegNode) {
            NegNode negNode = (NegNode) node;
            return -evaluate(negNode.expr);  // Apply negation.

        } else if (node instanceof FunctionCallNode) {
            FunctionCallNode funcCall = (FunctionCallNode) node;
            FunctionDeclNode funcDecl =
functions.get(funcCall.functionName);
            // Error warning
            if (funcDecl == null) {
                semanticErrors.add("Error: Undefined function " +
funcCall.functionName + " detected.");
                return 0.0;
            }
            // Argument Mismatch Check
            if (funcCall.arguments.size() != funcDecl.parameters.size()) {
                semanticErrors.add("Error: The number of arguments of " +
funcCall.functionName +
                                   " mismatched, Required: " +
funcDecl.parameters.size() +
                                   ", Actual: " +
funcCall.arguments.size());
                return 0.0;
            }
            // Evaluate the arguments in the current scope and store them in
a new map for the function scope.
            Map<String, Double> newScope = new HashMap<>(variables);

            for (int i = 0; i < funcCall.arguments.size(); i++) {
                String param = funcDecl.parameters.get(i);
                double argValue = evaluate(funcCall.arguments.get(i));
                newScope.put(param, argValue);
            }

            // Save the current scope.
            Map<String, Double> oldScope = new HashMap<>(variables);
            // Replace the current scope with the new scope for the function
body evaluation.
            variables = newScope;

            double result = evaluate(funcDecl.body);
            // Restore the old scope after the function call.
            variables = oldScope;
            return result;

        } else if (node instanceof InfixNode) {
            InfixNode infixNode = (InfixNode) node;
            double left = evaluate(infixNode.left);
            double right = evaluate(infixNode.right);
            switch (infixNode.op) {
                case "ADD":
                    return left + right;
                case "SUB":
                    return left - right;
                case "MUL":
                    return left * right;
                default:
                    // DIV case
                    // You don't have to worry about division error.
                    return left / right;
            }

        } else if (node instanceof NumberNode) {
            NumberNode numberNode = (NumberNode) node;
            return numberNode.value;

        } else if (node instanceof IdNode) {
            IdNode idNode = (IdNode) node;
            if (variables.containsKey(idNode.IdName)) {
                return variables.get(idNode.IdName);
            } else {
                // Free Identifier Check
                semanticErrors.add("Error: Free identifier " + idNode.IdName
+ " detected.");
                return 0.0;
            }

        } else if (node instanceof AssignNode) {
            AssignNode assignNode = (AssignNode) node;
            double value = evaluate(assignNode.right);
            String id = assignNode.idName;
            variables.put(id, value);
            return 0.0;
        }

        return 0.0; // Default value for any other node
```

```
    }
}
```

`Evaluate` is a class that traverses an AST and computes the result of the F1VAE program. It maintains a symbol table (`variables`) for variable values and a function table (`functions`) for defined functions. The `evaluate` method is overloaded for different AST node types, handling their specific evaluation logic. It also captures semantic errors, such as undefined functions, argument mismatches, and free identifiers, and adds these errors to a list (`semanticErrors`).

Handling Scopes in Function Calls

When a function call is made, the interpreter needs to create a new scope for the function's execution. This is because the function might have its own local variables (parameters) that should not interfere with the variables in the global scope or the scope of the caller. Here's how it's done:

1. **Creating a New Scope:**
   - When a `FunctionCallNode` is encountered, a new scope is created as a copy of the current scope (`variables`). This is done by initializing `newScope` as a new `HashMap` with the contents of `variables` .
2. **Setting Argument Values:**
   - The arguments passed to the function are evaluated in the current scope. This means that their values are computed considering the variables and functions available outside the function.
   - For each argument, its value is then mapped to the corresponding parameter name in `newScope` .
3. **Switching to the New Scope:**
   - The current scope (`variables`) is saved in `oldScope` for later restoration.
   - `variables` is then set to `newScope` . This means that from this point onwards, any variable lookup or modification within the function body will be done in the context of the function's scope.
4. **Evaluating the Function Body:**
   - The function's body is then evaluated with `variables` now pointing to `newScope` . This confines the function execution to its local scope.
5. **Restoring the Old Scope:**
   - Once the function body is evaluated, the original scope (`oldScope`) is restored to `variables` . This ensures that after the function call, the execution continues with the original set of variables.

Handling Scopes in `let` Expressions

The `let` expression creates a temporary scope where a variable is bound to a value only within the expression.

1. **Evaluating the Bound Expression:**
   - The expression to the right of the `=` in the `let` statement is evaluated, and its result is bound to the variable specified in the `let`.
2. **Temporary Variable Binding:**
   - If the variable already exists in `variables`, its current value is saved in `oldValue`.
   - The new value is then put in `variables`.
3. **Evaluating the `in` Expression:**
   - The expression following the `in` keyword is evaluated. During this evaluation, the variable introduced by the `let` is available in `variables`.
4. **Restoring the Original Value:**
   - After the evaluation of the `in` expression, if `oldValue` was not `null` (i.e., the variable existed before the `let`), its original value is restored.
   - If the variable did not exist before the `let`, it is removed from `variables`, effectively limiting its scope to the `let` expression.

5. `Program.java` Modifications

```java
package expression;
import java.io.IOException;

import org.antlr.v4.runtime.*;

import antlr.ExprLexer;
import antlr.ExprParser;

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);
        // Make AST from prog and print the tree
        ExprParser.ProgContext ctx = parser.prog();
        ProgNode AST = (ProgNode)new BuildAstVisitor().visitProg(ctx);
        // Semantic error detection
        Evaluate Evaluator1 = new Evaluate();
        AST.expressions.forEach(node -> Evaluator1.evaluate(node));
        if (!Evaluator1.semanticErrors.isEmpty()) {
            Evaluator1.semanticErrors.forEach(System.out::println);
            return;  // Terminate if there are semantic errors
        }

        AST.expressions.forEach(node -> new AstCall().Call(node, 0));
        // Evaluate AST result
        Evaluate Evaluator2 = new Evaluate();
        AST.expressions.forEach(node ->
System.out.println(Evaluator2.evaluate(node)));
    }
}
```

The `program` class contains the `main` method, acting as the entry point for the F1VAE interpreter. It sets up the ANTLR lexer and parser to read and parse input, then uses `BuildAstVisitor` to create an AST from the parsed program (`ProgContext`). It performs a first pass to detect semantic errors using an instance of `Evaluate`, printing any errors found. If no semantic errors are detected, it proceeds to evaluate the program, printing the results of each expression in the AST.

semantic error detection stage

Comparative Analysis : Implementing the F1VAE Interpreter in OCaml and Java with ANTLR

I had the opportunity to work on two separate projects involving the implementation of an interpreter for the F1VAE language. The first project was done in OCaml, a functional programming language known for its powerful pattern matching and concise syntax. The second project involved using Java in combination with ANTLR, a robust tool for generating parsers. This report aims to compare and contrast my experiences with both projects, highlighting the strengths and challenges of each approach.

Language and Tool Selection

- **OCaml Experience:**
    - The functional nature of OCaml made handling recursive data structures, like abstract syntax trees (ASTs), quite intuitive. Pattern matching in OCaml is a standout feature,

    making the interpreter logic clearer and more maintainable.
  - However, the manual effort required in parsing and the steep learning curve of functional programming posed significant challenges, especially for someone more accustomed to imperative programming languages.
- **Java with ANTLR Experience:**
  - ANTLR significantly simplified the parsing process in the Java project. Writing a grammar and having a parser and lexer generated automatically saved considerable time and effort.
  - The object-oriented approach of Java provided a structured way to build the interpreter, though it did require more boilerplate code compared to OCaml. The visitor pattern, used for traversing the AST, facilitated a clean separation of logic and operations on the tree nodes.

Parsing and AST Generation

- **In OCaml:**
  - Crafting the parser manually in OCaml was a challenging yet rewarding experience. It provided a deep understanding of the parsing process but at the cost of increased complexity and potential for errors.
  - Direct manipulation of ASTs in OCaml was more straightforward and required less code, aligning well with the language's strengths in handling such structures.
- **In Java with ANTLR:**
  - ANTLR's automation in generating a parser was a game-changer. It abstracted the complexities of parsing and allowed me to focus more on the interpreter logic.
  - Implementing the visitor pattern in Java was initially challenging but proved to be a powerful approach to separate the concerns of tree traversal from node processing.

Error Handling

- **OCaml's Approach:**
  - Error handling in OCaml was an elegant affair. Utilizing option types and pattern matching, I could manage errors gracefully within the functional flow.
  - The expressiveness of OCaml made the error-handling code less verbose and more readable.
- **Java with ANTLR's Approach:**
  - Java's exception handling mechanism provided a clear and explicit way to manage errors. This made the control flow for handling exceptions distinct from the main program logic.
  - The ability to create detailed error objects in Java was advantageous, offering more information for debugging purposes.

Conclusion

Both OCaml and Java with ANTLR have distinct advantages for building an interpreter. OCaml, with its functional paradigm, offers succinctness and a natural fit for language interpretation. In contrast, Java combined with ANTLR brings powerful tooling and a more structured approach to the table, suitable for larger-scale projects. My experience with both languages has been immensely educational, highlighting the importance of choosing the right tool for the task at hand. As I reflect on these projects, I realize that the choice between OCaml and Java with ANTLR is not just about language capabilities but also about personal preference and the specific demands of the project.

Test cases with below examples

```
// OCaml: let ast = (Prog ([], (Add (Id "x", Num 1))))
// x + 1;
// "Error: Free identifier x detected."

// OCaml: let ast = (Prog ([(FunDef ("x", [], (Add (Id "x", Num 1))))],
(Call ("x", []))))
// def x = x + 1 endef x();
// "Error: Free identifier x detected."

// OCaml: let ast = (Prog ([(FunDef ("x", ["x";"y"], (Add (Id "x", Num
1))))], (Call ("x", []))))
// def x x y = x + 1 endef x();
// "Error: The number of arguments of x mismatched, Required: 2, Actual: 0"

// OCaml: let ast = (Prog ([(FunDef ("x", ["x";"y"], (Add (Id "x", Num
1))))], (Call ("x", [Num 1]))))
// def x x y = x + 1 endef x(1);
//"Error: The number of arguments of x mismatched, Required: 2, Actual: 1"
//체크 한번 해야함 -> 0 아니면 error 나옴

// OCaml: let ast = (Prog ([(FunDef ("x", [], (Add (Num 5, Num 9))))], (Call
("x", []))))
// def x = 5 + 9 endef x();
// 14

// OCaml: `let ast = (Prog ([(FunDef ("x", ["x"], (Add (Id "x", Num 1))))],
(Call ("x", [Num 3]))))`
// def x x = x + 1 endef x(3);
// 4
// OCaml: `let ast = (Prog ([(FunDef ("x", ["x"; "y"], (Add (Id "x", Id
"y"))))], (Call ("x", [Num 3; Num 4]))))`
// def x x y = x + y endef x(3, 4);
// 7

// OCaml: `let ast = (Prog ([(FunDef ("x", ["x"; "y"], (Sub (Id "x", Id
"y"))))], (Call ("x", [Num 10; Num 5]))))`
// def x x y = x - y endef x(10, 5);
// 5

OCaml: `let ast = (Prog ([(FunDef ("x", ["x"; "y"], (Add (Id "x", Id
"y")))); (FunDef ("x2", ["x"; "y"], (Sub(Id "x", Id "y"))))], (Call ("x",
[Num 3; Num 4]))))`
// def x x y = x + y endef def x2 x y = x - y endef x(3, 4);
// 7

OCaml: `let ast = (Prog ([], (Add (Add ((Num 5, Num 18)), Num 30))))`
// (5 + 18) + 30;
// 53

OCaml: `let ast = (Prog ([], (Sub (Add ((Num 50, Num 18)), Num 3))))`
//(50 + 18) - 3;
// 65

OCaml: `let ast = (Prog ([], (Add (Add ((Num 1, Num 2)), Num (-3)))))`
// (1 + 2) + -3;
// 0


OCaml: `let ast = (Prog ([], (LetIn (("x"), (Num 5), (Add (Num 3, Id "x")))
)))`
// let x = 5 in 3 + x;
// 8

OCaml: `let ast = (Prog ([], (LetIn (("x"), (Num 5), (Add (Num 3, Id "z")))
)))`
// let x = 5 in 3 + z;
//  "Error: Free identifier z detected."

OCaml: `let ast = (Prog ([(FunDef ("x", ["x";"y"], (Add (Id "x", Num 1))))],
(Call ("y", []))))`
//def x x y = x + 1 endef y();
// "Error: Undefined function y detected."
```

```
//additional test case shown in ppt

def f =3 endef let x= f() in y =3 ;
def f =3 endef def g x y = x + y endef let x = f() in g(1,x);
let x = 5 in 3 + x
def x = x+1 endef x();
def x x y = x + 1 endef y ();
def x x y = x + 1 endef x();
```