Report

Name: Jin Zhenming (김진명)

Student ID: 2022315690

This is the report for my PA4.

1. Expr.g4

I added a rule called “decl” to deal with function declarations. Also, since function declarations are always before the main program, I edited the prog rule (adding “decl*” before expr) to make it process both the case with function declarations and not having it (* means 0 or more).

I also added more cases in the expr rule, they are “funcallExpr”, “negateExpr”, and “assignExpr” (Edited). “funcallExpr” used “(expr (',' expr)*)?” to allow it to deal with all possible situations for a function call (no arguments, 1 argument, multiple arguments. And divided by comma). “negateExpr” is a mutated version of “parensExpr” (“~” was added at the start of it). “assignExpr” was also edited to fit the new “LETIN” format.

- Prog
    - “decl*” is placed before “expr” so that it can process both the program with function declarations and not with it.
- Decl

    - It’s used to process function declarations. The first “id” represents the function name.
    - “id*” represents the parameters. ‘*’ is added in the end to handle the case with parameters and not having it.
    - “expr” is the function body
- Expr
    - parensExpr
        - This process the expr with parentheses
    - negateExpr
        - An edited version of “parensExpr”. ‘~’ is added at the start to represent negate operator
    - funcallExpr
        - Function call expression. “id” is the function name
        - “(expr (',' expr)*)?” in the parentheses represents a function call arguments (divided by comma).
    - infixExpr
        - The expression from PA1
        - To handle the infix calculation
        - To handle calculation priority, “*” and “/” are placed before “+” and “-”
    - numberExpr
        - The expression from PA1

        - To handle the number in the program
    - assignExpr
        - Modified version of assignExpr in PA1
        - Can also be called “letinExpr”
        - “id” is the name of the local variable in this letinExpr
        - The first “expr” is the value of “id”
        - The second “expr” is the expression that will be calculated in the end.
    - identifierExpr
        - The expression from PA1
        - To handle identifiers in the program

2. AstNodes.java

Similar to PA1, I modified this file to fit the updated Expr.g4 file.

- All the nodes have a function called “calcExpr” that will call the evaluate method in Evaluate.java to calculate the result.
- Modified the parameters of “AstNode”’s abstract method calcExpr. It now has two maps: one is for variables (“valMem”, Same as in PA1, String, and Double), and the other map is for function declarations (“funMem”, newly added).
    - The datatype of map “funMem” is “Map<String, ArrayList<AstNode>>” which String represents the name of the

function, and the ArrayList of AstNodes that stores the list of parameters and function body (the last element in the ArrayList is function body).

  - The datatype of map “valMem” is “Map<String, Double>” in which the String represents the name of the variable, and Double stores its value of it. (Same as PA1)

- The AsgnNode (AssignNode, or LetInNode) is modified to process the updated “assignExpr (or letinExpr)” in Expr.g4.
  - Added a new attribute called next, which represents the expression after at the end of the expression.

    The letin node is like: “let **id** = **value** in **next**”
- Added three new nodes “funcallNode”, “declNode”, and “negateNode”.
  - “negateNode” is simple, and only has a “value” attribute to represent the expr inside.
  - “funcallNode” represents function calls in the program, which will contain 2 parts, the name of the function, and the arguments of the function call. “id” stores the function name, and “args” is an ArrayList that stores arguments.
  - “declNode” stands for declaration node. It stores the name of the function (“id”), the parameter list (“params”), and the function body (“expr”).
- NormCalcNode

    - Node from PA1
    - It has “left” and “right” two AstNode attributes, also an operator.
    - It’s used to store “infixExpr” expression
- NumNode
    - Node from PA1
    - Stores number value
- IdtNode
    - Node from PA1
    - Stores the name of the identifier

3. AstCall.java

This file is also similar to PA1, but I expanded it to let it fit the requirements of PA4.

- I rewrite the AST output for AsgnNode(AssignNode).
    - The name of the node is changed to “LETIN”.
    - It will first print the local variable name through a recursive call, then the expr assigned to it (“value” attribute), and then the expr after “in” (“next” attribute).
        - The LETIN node is like: ‘let’ id ‘=’ value ‘in’ next
- AST output of “funcallNode” and “declNode” are also added
    - The output of “funcallNode” is like:

        1. Output the name of the node "CALL"
        2. Output the name of the function through a recursive call
        3. Output all the arguments (elements in args) recursively through a for loop.
    - The output of "declNode" is like:
        1. Output the name of the node "DECL"
        2. Output the name of the function through a recursive call
        3. Output all the parameters (all the elements in params, except the last one) recursively through a for loop
        4. Output the function body through a recursive call

           P.S. The last element of "params" is the function body for convenience of evaluation.
- AST output of "negateNode" is added.
    - The output is like:
        1. Output the name of the node "NEG"
        2. Output the expression in it ("value") through a recursive call.
- "node instanceof NormCalcNode"
    - Same as PA1
    - Output the operator of infix expression ("ADD", "SUB", etc.)
    - Output the left-hand side and the right-hand side of the expression recursively.

- “node instanceof NumNode”
    - Same as PA1
    - Output the number (“value” attribute of the node)
- “node instanceof IdtNode”
    - Same as PA1
    - Output the name of the identifier

4. Evaluate.java

This includes a static method called “evaluate” that will calculate the result of the AstNode.

Most parts are the same as in PA1

These are the changes I made (Compare to PA1, rests are the same):

- To be able to process the new structure of “AsgnNode (LetInNode)”, I rewrote the “node instance of AsgnNode” case.
    - “valMem” is the map to store variables. It will store the variable and its value in the map and return the result of the recursive evaluation of “next” attribute of the node (with updated “valMem”).
- In “node instanceof IdtNode” case, I added an exception handling. By checking whether the return value of accessing the map is null or not to identify if the identifier is a free identifier.

- In “node instanceof negateNode” case, I let it return the negated evaluation result of the “value” attribute.
- I added “node instanceof declNode”

  It will store the function declaration to “funMem” (a map that will store all the function declarations). The expression is stored at the last position of params (it’s an ArrayList that stores AstNodes, polymorphism is used to make it possible)
- I added “node instanceof funcallNode”.
  - It will get the name of the function being called in the node (“id”). Get the arguments of the function call from the node and the ArrayList that stores the parameters of the function in “funMem”, which is the map that stores the function declarations (the last element in params is the function body for convenience).
  - If the parameter list returned from the map is empty, it means the function is not defined, then it will throw an exception.
  - If the number of arguments does not match the number of parameters, then it will throw an exception.
  - After checking for errors, it will declare a new map that stores local variables (parameters). A for loop is used to match the parameter name and the argument value and put them into the map (the local memory called “localValMem”).

    - After matching parameters and arguments, it will evaluate the result through a recursive function call with local variable memory and function memory as parameters, and the expression it’s calculating is the function body.
- “node instanceof NormCalcNode”
    - Same as PA1
    - Return different calculations result recursively according to the operator
- “node instanceof NumNode”
    - Same as PA1
    - Return the value of the number (“value” attribute of the node)
- “node instanceof IdtNode”
    - Similar to PA1
    - Before returning the corresponding value stored in the memory, it will check if the value exists first, then return the value.
    - If the value does not exist (the return value from the map is null), then exit with an error message.

5. BuildAstVisitor.java

Most methods remain the same as my PA1, but I made several modifications to be able to process new grammar.

- “visitAssignExpr”
    - I modified this method to let it fit the new “letin” expression. It will read the name of the identifier into a String, visit the first expr to get the expression of the identifier (the part between ‘=’ and ‘in’), and visit the second expr to get the expression of “next” (the part after ‘in’).
    - Then it will return a new “AsgnNode” with arguments of “id” (name of the identifier), “value” (expression of the identifier), and “next” (the last expression).
- “visitFuncallExpr”
    - A new method to process function calls.
    - The name of the function being called is being read into a String.
    - The arguments will be stored in an ArrayList that stores AstNode (ArrayList<AstNode>)
    - Before reading the arguments of the function call, it will first check if it has arguments. Only when it has arguments, it will start reading arguments in the function call.
    - In the end, it will return a new funcallNode with arguments “id” (the name of the function) and “args” (expressions of arguments).
- “visitNormDecl”
    - A new method to process function declarations
    - The name of the function is being read into a String

    - The parameters are stored in an ArrayList (ArrayList<AstNode>).
    - There might be several ‘id’s in the expression, the first ‘id’ is always the function's name, and the rest are parameters. Therefore the for loop that visits the context and adds the IdtNode to the array starts from “i = 1”.
    - In the end, it will add the function body (expr) to the parameter ArrayList, and return a new “declNode” with parameters “id”, “param”, and “expr”.
- “visitNegateExpr”
    - Return a new “negateNode” with the argument “visit(ctx.expr())”, which is the expression in the parentheses.
- “visitInfixExpr”
    - Same as PA1
    - Visit 2 “exprs” (left-hand side and right-hand side) and the “operator” in the content and store them as AstNodes and String.
    - Return a new NormCalcNode
- “visitParensExpr”
    - Same as PA1
    - Visit and return the “expr” in the parentheses
- “visitIdentifierExpr”
    - Same as PA1

    - Visit the content and get the text of the “id” in the context. Return a new “IdtNode” with the text we got.

6. program.java

- I declared 2 HashMap. One is to store the variable, the other one stores function declarations.
- Two ArrayLists are used to store the results of the expressions, one is for declarations, and the other is for expressions.
- The result will be calculated before printing AST, and the result will be stored in the corresponding ArrayList. I let the program do the evaluation first to detect the exceptions.
    - For each loop is used to calculate the result
    - Evaluate function declaration first, then expressions
- After calculating the result, it will output AST through “AstCall.Call” function. The initial indent number is 0.
    - For loop is used to output the AST
    - Output AST of function declaration first, then expressions
- In the end, it will print the result of the expressions. Since the results are stored in ArrayLists, the results being printed are from these two ArrayLists.
    - For loop is used to output the AST

    - Output the result of function declaration first, then expressions

My experience comparing implementation on OCaml and ANTLR

1. Implementing OCaml can write less code compared to ANTLR.
2. In PA3 (and interpreter written in OCaml), there wasn't any user input but received AST directly during compiling (which means we only let it run the AST program, and we didn't implement the lexer and the parser). In ANTLR, we make it able to receive user input from the terminal, which means we'll have to write lexer rules in the g4 file.
3. ANTLR is based on Java and has better readability than OCaml. (not really a big difference on it, maybe this difference in readability is because I feel more familiar with Java)
4. I experienced a new form of programming language, called functional programming language, different from what I have learned. For instance, I never thought a function could be a parameter so that it could implement some functions like Summation (∑ operation) easily. In Java, it will be more complicated.