AST

ASTGeneration.py

```python
from BKITVisitor import BKITVisitor
from BKITParser import BKITParser
from ASTUtils import *

class ASTGeneration(BKITVisitor):
    def visitProgram(self, ctx:BKITParser.ProgramContext):
        '''
            Prog: dataclass in ASTUtils

            We need to visit all children of program. They are list of expressions.
            ctx.expression() returns the list we desire.

            Then we call expression.accept(self) or self.visitExpression(expression),
            function visitExpression() will be triggered.
        '''
        expressions = ctx.expressions().accept(self)
        return Prog(expressions)

    def visitExpressions(self, ctx:BKITParser.ExpressionsContext):
        list_expressions = []
        if ctx.expressions():
            list_expressions.extend(ctx.expressions().accept(self))
            list_expressions.append(ctx.expression().accept(self))

        return list_expressions

    def visitExpression(self, ctx:BKITParser.ExpressionContext):
        '''
            BinOp: dataclass in ASTUtils
        '''
        if ctx.expression():
            sign  = ""
            if ctx.Add():
                sign = ctx.Add().getText()
            elif ctx.Sub():
                sign = ctx.Sub().getText()

            return BinOp(sign, ctx.expression().accept(self), ctx.factor().accept(self))
        else:
            return ctx.factor().accept(self)

    def visitFactor(self, ctx:BKITParser.FactorContext):
        '''
            BinOp: dataclass in ASTUtils
        '''
        if ctx.factor():
            sign  = ""
            if ctx.Mul():
                sign = ctx.Mul().getText()
            elif ctx.Div():
                sign = ctx.Div().getText()

            return BinOp(sign, ctx.factor().accept(self), ctx.term().accept(self))
        else:
            return ctx.term().accept(self)

    def visitTerm(self, ctx:BKITParser.TermContext):
        '''
            As defined in BKIT.g4, term can be either an Integer or an Id,
            so that we need to check if this term contains Integer or Id.

            Because Integer and Id are leaf nodes in AST,
            we must directly call function self.visitInteger(ctx.Integer()) or self.visitId(ctx.Id())
        '''
        if ctx.Integer():
            return self.visitInteger(ctx.Integer())
        elif ctx.Identifier():
            return self.visitIdentifier(ctx.Identifier())
    def visitInteger(self, node:BKITParser.Integer):

        return Int()

    def visitIdentifier(self, node:BKITParser.Identifier):
        return Id()
```

BKIT.g4

```
grammar BKIT;

@lexer::header {
from lexererr import *
}

@lexer::members {
def emit(self):
    tk = self.type
    result = super().emit()
    if tk == self.UNCLOSE_STRING:
        raise UncloseString(result.text)
    elif tk == self.ILLEGAL_ESCAPE:
        raise IllegalEscape(result.text)
    elif tk == self.ERROR_CHAR:
        raise ErrorToken(result.text)
    elif tk == self.UNTERMINATED_COMMENT:
        raise UnterminatedComment()
    else:
        return result;
}

options{
         language=Python3;
}

program
    : expressions EOF
    ;

expressions: expressions expression | ;

expression
    : expression (Add|Sub) factor
    | factor
    ;

factor
    : factor (Mul|Div) term
    | term
    ;

Add : '+';
Sub : '-';
Mul : '*';
Div : '/';

term
    : Integer
    | Identifier
    ;

Integer: [1-9][0-9]* | [0] ;
Identifier: [a-z]+ ;

WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines

ERROR_CHAR: .;
UNCLOSE_STRING: .;
ILLEGAL_ESCAPE: .;
UNTERMINATED_COMMENT: .;
```

AST-1

ASTGeneration.py

```
from BKITVisitor import BKITVisitor
from BKITParser import BKITParser
from ASTUtils import *

class ASTGeneration(BKITVisitor):
    def visitProgram(self, ctx:BKITParser.ProgramContext):
        '''
            Prog: dataclass in ASTUtils
        '''
        if ctx.Add() or ctx.Sub():
            op = ctx.Add().getText() if ctx.Add() else ctx.Sub().getText()
            left = ctx.idTerm().accept(self)
            right = ctx.term().accept(self)
            return Prog(BinOp(op, left, right))
        else:
            _term = ctx.term().accept(self)
            return Prog(_term)

    def visitTerm(self, ctx:BKITParser.TermContext):
        if ctx.idTerm():
            return ctx.idTerm().accept(self)
        else:
            return ctx.intTerm().accept(self)

    def visitIntTerm(self, ctx:BKITParser.IntTermContext):
        '''
            Int: dataclass in ASTUtils
        '''
        return Int(int(ctx.Integer().getText()))

    def visitIdTerm(self, ctx:BKITParser.IdTermContext):
        '''
            Id: dataclass in ASTUtils
        '''
        return Id(ctx.Identifier().getText())
```

BKIT.g4

```
grammar BKIT;

@lexer::header {
from lexererr import *
}

@lexer::members {
def emit(self):
    tk = self.type
    result = super().emit()
    if tk == self.UNCLOSE_STRING:
        raise UncloseString(result.text)
    elif tk == self.ILLEGAL_ESCAPE:
        raise IllegalEscape(result.text)
    elif tk == self.ERROR_CHAR:
        raise ErrorToken(result.text)
    elif tk == self.UNTERMINATED_COMMENT:
        raise UnterminatedComment()
    else:
        return result;
}

options{
         language=Python3;
}

program
```

```
    : (idTerm Add term | idTerm Sub term | term) EOF
    ;

term: idTerm | intTerm;

intTerm : Integer;
idTerm : Identifier;

Add: '+';
Sub: '-';
Integer: [1-9][0-9]* | [0];
Identifier: [a-z]+ ;

WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines

ERROR_CHAR: .;
UNCLOSE_STRING: .;
ILLEGAL_ESCAPE: .;
UNTERMINATED_COMMENT: .;
```

CodeRunner3

CodeRunner.py

```python
from ASTUtils import *
from utils import lookup

"""
In utils.py, we have a function lookup(string) that returns the value of a variable in the environment.
Usage: lookup("x") ==> value of variable x
"""


class CodeRunner:
    def visitProgram(self, ctx: Prog):
        return str(ctx.expr.accept(self))

    def visitBinaryOp(self, ctx: BinOp):
        left = ctx.left.accept(self)
        right = ctx.right.accept(self)

        if ctx.op == "+":
            return left + right
        elif ctx.op == "-":
            return left - right
        elif ctx.op == "*":
            return left * right
        elif ctx.op == "/":
            return left / right
        elif ctx.op == "%":
            return left % right

    def visitInteger(self, node: Int):
        return node.value

    def visitId(self, node: Id):
        return lookup(node.name)
```

ASTGeneration.py

```python
from BKITVisitor import BKITVisitor
from BKITParser import BKITParser
from ASTUtils import *


class ASTGeneration(BKITVisitor):
    def visitProgram(self, ctx: BKITParser.ProgramContext):
        return Prog(ctx.expression().accept(self))
    def visitExpression(self, ctx: BKITParser.ExpressionContext):
        if ctx.expression():
            op = ctx.Add().getText() if ctx.Add() else ctx.Sub().getText()
            left = ctx.term().accept(self)
            right = ctx.expression().accept(self)

            return BinOp(op, left, right)

        else:
            return ctx.term().accept(self)

    def visitTerm(self, ctx: BKITParser.TermContext):
        if ctx.term():
            op = ""
            if ctx.Mul():
                op = ctx.Mul().getText()
            elif ctx.Div():
                op = ctx.Div().getText()
            else:
                op = ctx.Mod().getText()

            left = ctx.factor().accept(self)
            right = ctx.term().accept(self)

            return BinOp(op, left, right)

        else:
            return ctx.factor().accept(self)

    def visitFactor(self, ctx: BKITParser.FactorContext):
        if ctx.intTerm():
            return ctx.intTerm().accept(self)
        elif ctx.idTerm():
            return ctx.idTerm().accept(self)

    def visitIntTerm(self, ctx: BKITParser.IntTermContext):
        return Int(int(ctx.Integer().getText()))

    def visitIdTerm(self, ctx: BKITParser.IdTermContext):
        return Id(ctx.Identifier().getText())
```

BKIT.g4

```
grammar BKIT;

@lexer::header {
from lexererr import *
}

@lexer::members {
def emit(self):
    tk = self.type
    result = super().emit()
    if tk == self.UNCLOSE_STRING:
        raise UncloseString(result.text)
    elif tk == self.ILLEGAL_ESCAPE:
        raise IllegalEscape(result.text)
    elif tk == self.ERROR_CHAR:
        raise ErrorToken(result.text)
    elif tk == self.UNTERMINATED_COMMENT:
        raise UnterminatedComment()
    else:
        return result;
}

options{
         language=Python3;
}


program
```

```
    : expression EOF
    ;

expression
    : term (Add | Sub) expression
    | term
    ;

term
    : factor (Mul | Div | Mod) term
    | factor
    ;

factor
    : intTerm | idTerm
    ;

intTerm: Integer;

idTerm: Identifier;

Add : '+';
Sub : '-';
Mul : '*';
Div : '/';
Mod : '%';

Integer: [0-9]+ ;
Identifier: [a-zA-Z_][a-zA-Z0-9_]*;

WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines

ERROR_CHAR: .;
UNCLOSE_STRING: .;
ILLEGAL_ESCAPE: .;
UNTERMINATED_COMMENT: .;
```