Compiler Experiment : Simple Compiler

Author：软件81金之航

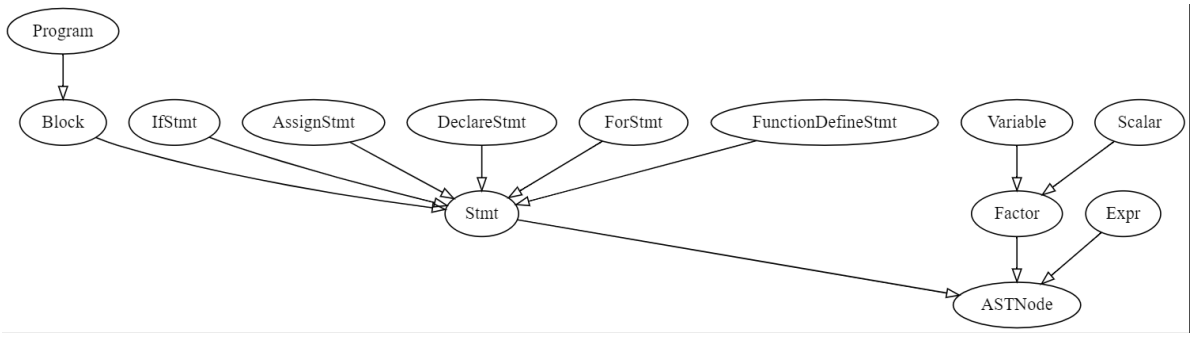
StuId：2183411101

二.语法分析 Parser

语法分析器(parser)：根据语法规则,将符号(词法单元,lexeme,token)流，转换成抽象语法树
类比：中文分析句子成分。

和自然语言不通，编译器只能识别上下文无关文法。一个文法是上下文无关，也就是说,不需要理解这个语言，给定任意这个语言的句子，可以得到一个合理的抽象语法树AST(Abstract Syntax Tree)。

抽象语法树：源代码结构的抽象
抽象：隐藏细节(比如右边表达式的括号被隐藏了，因为和思考无关)
树：每个节点是源代码中的一种结构，每个节点都携带了源代码中的一些关键信息，每个节点代表语言上的关系。




```
定义语句块和语句：
Program --> Stmts --> Stmt Stmts | ϵ
Stmt --> IfStmt | WhileStmt | ForStmt | FunctionDefineStmt | Block
Block --> { Stmts }
IfStmt -> If(Expr) Block Tail
Tail -> else {Block} | else IFStmt | ϵ
DeclareStmt --> var Variable = Expr
AssignStmt --> Variable = Expr
Function --> func(Args) Type Block
Args --> Type Variable, Args | Type Variable | ϵ
ReturnType --> Type | ϵ
Type --> int | string | void | bool | ...
```

1.PeekTokenIterator

准备工作，读取Lexer输出的符号流Token Stream，封装的PeekIterator, 输入流为符号流。

```
public class PeekTokenIterator extends PeekIterator<Token> {

    public PeekTokenIterator(Stream<Token> stream) {
```

```
        super(stream);
    }

    public Token nextMatch(String value) throws ParseException {
        var token = this.next();
        if (!token.get_value().equals(value)) {
            throw new ParseException(token);
        }
        return token;
    }

    public Token nextMatch(TokenType type) throws ParseException {
        var token = this.next();
        if (!token.get_type().equals(type)) {
            throw new ParseException(token);
        }
        return token;
    }
}
```

2.ASTNodeType

一个枚举类，包含了ASTNode的类型。

```
public enum ASTNodeTypes {
    BLOCK,  //代码块
    BINARY_EXPR, // 二元表达式 eg:1+1
    UNARY_EXPR, // 一元表达式 eg:++i
    CALL_EXPR,  //调用语句
    VARIABLE,  //变量
    SCALAR, // 标量 eg:1.0, true
    IF_STMT,  //If语句
    WHILE_STMT,  //While语句
    FOR_STMT,  //For语句
    RETURN_STMT,  //返回语句
    ASSIGN_STMT,  //赋值语句
    FUNCTION_DECLARE_STMT,  //函数定义语句
    DECLARE_STMT  //定义语句
}
```

3.ASTNode

一个抽象树的结点，主要属性有词法单元，备注（标签），以及类型。

```
@Data
public abstract class ASTNode {
    /* 树 */
    protected ArrayList<ASTNode> children = new ArrayList<>();
    protected ASTNode parent;
    /* 关键信息 */
    protected Token lexeme; // 词法单元
    protected String label; // 备注(标签)
```

```java
    protected ASTNodeTypes type; // 类型

    private HashMap<String, Object> _props = new HashMap<>();

    public ASTNode() {
    }

    public ASTNode(ASTNodeTypes _type, String _label) {
        this.type = _type;
        this.label = _label;
    }

    public ASTNode getChild(int index) {
        if (index >= this.children.size()) {
            return null;
        }
        return this.children.get(index);
    }

    public void addChild(ASTNode node) {
        node.parent = this;
        children.add(node);
    }

    public List<ASTNode> getChildren() {
        return children;
    }

    public void print(int indent) {
        if (indent == 0) {
            System.out.println("print:" + this);
        }
        System.out.println(StringUtils.leftPad(" ", indent * 2) + label);
        for (var child : children) {
            child.print(indent + 1);
        }
    }

    public void replaceChild(int i, ASTNode node) {
        this.children.set(i, node);
    }

    public HashMap<String, Object> props() {
        return this._props;
    }

    public Object getProp(String key) {
        if (!this._props.containsKey(key)) {
            return null;
        }
        return this._props.get(key);
    }

    public void setProp(String key, Object value) {
        this._props.put(key, value);
    }

    public boolean isValueType() {
```

```
        return this.type == ASTNodeTypes.VARIABLE || this.type ==
ASTNodeTypes.SCALAR;
    }

    public void replace(ASTNode node) {
        if (this.parent != null) {
            var idx = this.parent.children.indexOf(this);
            this.parent.children.set(idx, node);
            //this.parent = null;
            //this.children = null;
        }
    }
}
```

4.Stmt

一个抽象的表达式类，继承ASTNode，被IfStmt，WhileStmt，DeclareStmt等具体的表达式继承。
Program --> Stmts --> Stmt Stmts | ε
Stmt --> IfStmt | WhileStmt | ForStmt | FunctionDefineStmt | Block

```
public abstract class Stmt extends ASTNode {
    public Stmt(ASTNodeTypes _type, String _label) {
        super(_type, _label);
    }

    public static ASTNode parseStmt(PeekTokenIterator it) throws
ParseException {
        if (!it.hasNext()) {
            return null;
        }
        var token = it.next();
        var lookahead = it.peek();
        it.putBack();

        if (token.isVariable() && lookahead != null &&
lookahead.get_value().equals("=")) {
            return AssignStmt.parse(it);
        } else if (token.get_value().equals("var")) {
            return DeclareStmt.parse(it);
        } else if (token.get_value().equals("func")) {
            return FunctionDeclareStmt.parse(it);
        } else if (token.get_value().equals("return")) {
            return ReturnStmt.parse(it);
        } else if (token.get_value().equals("if")) {
            return IfStmt.parse(it);
        } else if (token.get_value().equals("{")) {
            return Block.parse(it);
        } else {
            return Expr.parse(it);
        }
    }
}
```

5.Block

代码块，继承了Stmt，被Program继承。
Block --> { Stmts }

```
ublic class Block extends Stmt {

    public Block() {
        super(ASTNodeTypes.BLOCK, "block");
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        var block = new Block();
        it.nextMatch("{");
        ASTNode stmt = null;
        while( (stmt = Stmt.parseStmt(it)) != null) {
            block.addChild(stmt);
        }
        it.nextMatch("}");
        return block;
    }
}
```

6.Program

Program类，具体表现为一个个程序代码。
Program --> Stmts --> Stmt Stmts | ε

```
public class Program extends Block {
    public Program() {
        super();
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        var block = new Program();
        ASTNode stmt = null;
        while( (stmt = Stmt.parseStmt(it)) != null) {
            block.addChild(stmt);
        }
        return block;
    }
}
```

7.IfStmt

If语句，继承Stmt。
IfStmt -> If(Expr) Block Tail
Tail -> else {Block} | else IFStmt | ε

```
If语句的文法：
IfStmt --> If(Expr) Block Tail
Tail --> else { Block } | else IFStmt | ϵ
```

```
public class IfStmt extends Stmt {
    public IfStmt() {
        super(ASTNodeTypes.IF_STMT, "if");
    }

    public static ASTNode parse(PeekTokenIterator iterator) throws
ParseException {
        return parseIF(iterator);
    }

    // IfStmt -> If(Expr) Block Tail
    public static ASTNode parseIF(PeekTokenIterator iterator) throws
ParseException {
        var lexeme = iterator.nextMatch("if");
        iterator.nextMatch("(");
        var ifStmt = new IfStmt();
        ifStmt.setLexeme(lexeme);
        var expr = Expr.parse(iterator);
        ifStmt.addChild(expr);
        iterator.nextMatch(")");
        var block = Block.parse(iterator);
        ifStmt.addChild(block);
        var tail = parseTail(iterator);
        if (tail != null) {
            ifStmt.addChild(tail);
        }
        return ifStmt;

    }

    // Tail -> else {Block} | else IFStmt | ϵ
    public static ASTNode parseTail(PeekTokenIterator iterator) throws
ParseException {
        if (!iterator.hasNext() ||
!iterator.peek().get_value().equals("else")) {
            return null;
        }
        iterator.nextMatch("else");
        var lookahead = iterator.peek();
        if (lookahead.get_value().equals("{")) {
            return Block.parse(iterator);
        } else if (lookahead.get_value().equals("if")) {
            return parseIF(iterator);
        } else {
            return null;
        }
    }
}
```

8.AssignStmt

赋值语句，继承了Stmt
AssignStmt --> Variable = Expr

```
public class AssignStmt extends Stmt {
    public AssignStmt() {
        super(ASTNodeTypes.ASSIGN_STMT, "assign");
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        var stmt = new AssignStmt();
        var tkn = it.peek();
        var factor = Factor.parse(it);
        if (factor == null) {
            throw new ParseException(tkn);  //tkn is not Variable or Scala
        }
        stmt.addChild(factor);
        var lexeme = it.nextMatch("=");
        var expr = Expr.parse(it);
        stmt.addChild(expr);
        stmt.setLexeme(lexeme);
        return stmt;
    }
}
```

9.DeclareStmt

定义语句，继承了Stmt
DeclareStmt --> var Variable = Expr

```
public class DeclareStmt extends Stmt {
    public DeclareStmt() {
        super(ASTNodeTypes.DECLARE_STMT, "declare");
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        var stmt = new DeclareStmt();
        it.nextMatch("var");
        var tkn = it.peek();
        var factor = Factor.parse(it);
        if (factor == null) {
            throw new ParseException(tkn);  //tkn is not Variable or Scala
        }
        stmt.addChild(factor);
        var lexeme = it.nextMatch("=");
        var expr = Expr.parse(it);
        stmt.addChild(expr);
        stmt.setLexeme(lexeme);
        return stmt;
    }
}
```

10.ForStmt

For语句，继承了Stmt

```
public class ForStmt extends Stmt {
    public ForStmt() {
        super(ASTNodeTypes.FOR_STMT, "for");
    }
}
```

11.1.FunctionDefineStmt

函数参数定义，继承了Stmt
Function --> func(Args) Type Block
Args --> Type Variable, Args | Type Variable | ε
ReturnType --> Type | ε
Type --> int | string | void | bool | ...

```
public class FunctionDeclareStmt extends Stmt {

    public FunctionDeclareStmt() {
        super(ASTNodeTypes.FUNCTION_DECLARE_STMT, "func");
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        it.nextMatch("func");

        // func add() int {}
        var func = new FunctionDeclareStmt();
        var lexeme = it.peek();  //func
        var functionVariable = (Variable) Factor.parse(it);  //add
        func.setLexeme(lexeme);
        func.addChild(functionVariable);
        it.nextMatch("(");
        var args = FunctionArgs.parse(it);
        it.nextMatch(")");
        func.addChild(args);
        var keyword = it.nextMatch(TokenType.KEYWORD);  //int
        if (!keyword.isType()) {
            throw new ParseException(keyword);
        }

        functionVariable.setTypeLexeme(keyword);
        var block = Block.parse(it);
        func.addChild(block);
        return func;
    }

    public ASTNode getArgs() {
        return this.getChild(1);
    }

    public Variable getFunctionVariable() {
```

```
        return (Variable) this.getChild(0);
    }

    public String getFuncType() {
        return this.getFunctionVariable().getTypeLexeme().get_value();
    }

    public Block getBlock() {
        return (Block) this.getChild(2);
    }

}
```

11.2.FunctionArgs

函数参数语句，继承了ASTNode，主要用于函数入口中的参数处理，比如func(String a, int b)
识别到（，接下来type = String, 把 a 交给Factor.parse处理，
然后判断接下来是，还是)
如果是，则继续type和Factor.parse处理
如果是），则返回args。

```
public class FunctionArgs extends ASTNode {
    public FunctionArgs() {
        super();
        this.label = "args";
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        var args = new FunctionArgs();
        while (it.peek().isType()) {
            var type = it.next();
            var variable = (Variable) Factor.parse(it);
            variable.setTypeLexeme(type);
            args.addChild(variable);
            if (!it.peek().get_value().equals(")")) {
                it.nextMatch(",");
            }
        }
        return args;
    }
}
```

12.Factor

因子抽象类，继承了ASTNode，主要用于处理ASTNode Type中的标量Scalar和变量Variable

```
public class Factor extends ASTNode {
    public Factor(Token token) {
        super();
        this.lexeme = token;
        this.label = token.get_value();
    }
```

```
7
8       public static ASTNode parse(PeekTokenIterator it) {
9           var token = it.peek();
10          var type = token.get_type();
11          if(type == TokenType.VARIABLE) {
12              it.next();
13              return new Variable(token);
14          } else if(token.isScalar()){
15              it.next();
16              return new Scalar(token);
17          }
18          return null;
19      }
20  }
```

13.Scalar

标量类，继承Factor

```
1  public class Scalar extends Factor{
2      public Scalar(Token token) {
3          super(token);
4          this.type = ASTNodeTypes.SCALAR;
5      }
6  }
```

14.Variable

变量类，继承Factor

```
1  @Data
2  public class Variable extends Factor {
3      private Token typeLexeme = null;
4
5      public Variable(Token token) {
6          super(token);
7          this.type = ASTNodeTypes.VARIABLE;
8      }
9  }
```

15.1.Expr

表达式类，继承ASTNode，对应的是非终结符，而终结符对应的是词法单元， 需要实现消除左递归，主要实现了combine(), race() 两个方法。

```
1  public class Expr extends ASTNode {
2
3      private static PriorityTable table = new PriorityTable();
4
```

```
    public Expr() {
        super();
    }

    public Expr(ASTNodeTypes type, Token lexeme) {
        super();
        this.type = type;
        this.label = lexeme.get_value();
        this.lexeme = lexeme;
    }

    // left:E(k) -> E(k) op(k) E(k+1) | E(k+1)
    // right:
    //    E(k) -> E(k+1) E_(k)
    //       var e = new Expr(); e.left = E(k+1); e.op = op(k); e.right = E(k+1) E_(k)
    //    E_(k) -> op(k) E(k+1) E_(k) | ϵ
    // 最高优先级处理:
    //    E(t) -> F E_(k) | U E_(k)
    //    E_(t) -> op(t) E(t) E_(t) | ϵ
    private static ASTNode E(int k, PeekTokenIterator it) throws ParseException {
        if (k < table.size() - 1) {
            return combine(it, () -> E(k + 1, it), () -> E_(k, it));
        } else {
            return race(
                    it,
                    () -> combine(it, () -> F(it), () -> E_(k, it)),
                    () -> combine(it, () -> U(it), () -> E_(k, it))
            );
        }
    }

    //E_(k) -> op(k) E(k+1) E_(k) | ϵ
    private static ASTNode E_(int k, PeekTokenIterator it) throws ParseException {
        var token = it.peek();
        var value = token.get_value();

        if (table.get(k).contains(value)) {
            var expr = new Expr(ASTNodeTypes.BINARY_EXPR, it.nextMatch(value));
            expr.addChild(Objects.requireNonNull(combine(it,
                    () -> E(k + 1, it),
                    () -> E_(k, it)
            )));
            return expr;
        }
        return null;
    }

    //E(t) -> F E_(k) | U E_(k)  最高优先级处理
    private static ASTNode U(PeekTokenIterator it) throws ParseException {
        var token = it.peek();
        var value = token.get_value();

        if (value.equals("(")) {
            it.nextMatch("(");
```

```
            var expr = E(0, it);
            it.nextMatch(")");
            return expr;
        } else if (value.equals("++") || value.equals("--") ||
value.equals("!")) {
            var t = it.peek();
            it.nextMatch(value);
            Expr unaryExpr = new Expr(ASTNodeTypes.UNARY_EXPR, t);
            unaryExpr.addChild(E(0, it));
            return unaryExpr;
        }
        return null;
    }

    //E(t) -> F E_(k) | U E_(k)  最高优先级处理
    private static ASTNode F(PeekTokenIterator it) throws ParseException {
        var factor = Factor.parse(it);
        if (factor == null) {
            return null;
        }
        if (it.hasNext() && it.peek().get_value().equals("(")) {
            return CallExpr.parse(factor, it);
        }
        return factor;
    }

    //E(k) -> E(k+1) E_(k) comibine E(k+1) E_(k)
    private static ASTNode combine(PeekTokenIterator it, ExprHOF aFunc,
ExprHOF bFunc) throws ParseException {
        var a = aFunc.hoc();
        if (a == null) {
            return it.hasNext() ? bFunc.hoc() : null;
        }
        var b = it.hasNext() ? bFunc.hoc() : null;
        if (b == null) {
            return a;
        }
        Expr expr = new Expr(ASTNodeTypes.BINARY_EXPR, b.lexeme);
        expr.addChild(a);
        expr.addChild(b.getChild(0));
        return expr;
    }

    private static ASTNode race(PeekTokenIterator it, ExprHOF aFunc,
ExprHOF bFunc) throws ParseException {
        if (!it.hasNext()) {
            return null;
        }
        var a = aFunc.hoc();
        if (a != null) {
            return a;
        }
        return bFunc.hoc(); //a == null
    }

    public static ASTNode parse(PeekTokenIterator it) throws ParseException
{
        return E(0, it);
```

```
    }
}
```

15.2ExprHOF

函数式编程思想：HOF: High order function 高阶函数

```
@FunctionalInterface
public interface ExprHOF {
    ASTNode hoc() throws ParseException;
}
```

15.3PriorityTable

优先级表

```
public class PriorityTable {
    private List<List<String>> table = new ArrayList<>();
    public PriorityTable() {
        table.add(Arrays.asList("&", "|", "^"));
        table.add(Arrays.asList("==", "!=", ">", "<", ">=", "<="));
        table.add(Arrays.asList("+", "-"));
        table.add(Arrays.asList("*", "/"));
        table.add(Arrays.asList("<<", ">>"));
        //为什么没有() ++ -- ! 第一优先级操作符号在构造算符优先文法阶段处理，最高优先级处理:
        //    E(t) -> F E_(k) | U E_(k)
        //    E_(t) -> op(t) E(t) E_(t) | ϵU
    }

    public int size(){
        return table.size();
    }

    public List<String> get(int level) {
        return table.get(level);
    }
}
```

16.ParserUtils

Parser的工具类，有将前缀表达式转换至后缀表达式的方法toPostfixExpression() 和 宽度优先遍历的方法toBFSString()

```
public class ParserUtils {
    // Prefix
    // Postfix
    public static String toPostfixExpression(ASTNode node) {
        if (node instanceof Factor) {
            return node.getLexeme().get_value();
        }
        var prts = new ArrayList<String>();
        for (var child : node.getChildren()) {
```

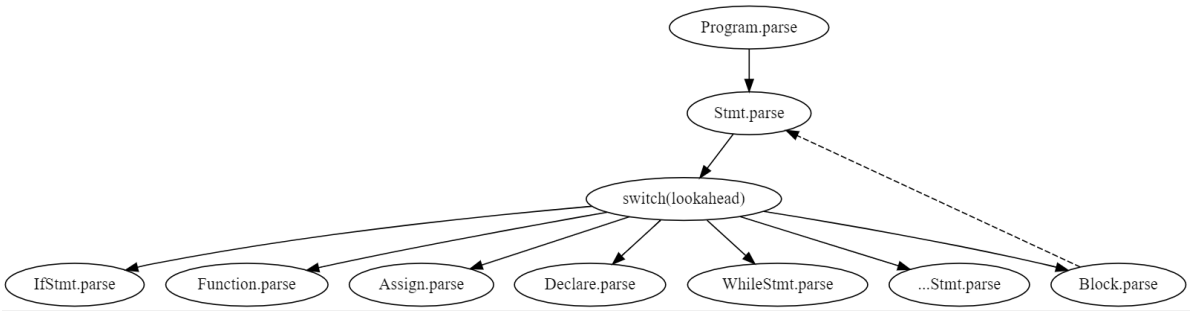
```
            prts.add(toPostfixExpression(child));
        }
        var lexemeStr = node.getLexeme() != null ?
node.getLexeme().get_value() : "";
        if (lexemeStr.length() > 0) {
            return StringUtils.join(prts, " ") + " " + lexemeStr;
        } else {
            return StringUtils.join(prts, " ");
        }
    }

    public static String toBFSString(ASTNode root, int max) {
        var queue = new LinkedList<ASTNode>();
        var list = new ArrayList<String>();
        queue.add(root);

        int c = 0;
        while (queue.size() > 0 && c++ < max) {
            var node = queue.poll();
            list.add(node.getLabel());
            for (var child : node.getChildren()) {
                queue.add(child);
            }
        }
        return StringUtils.join(list, " ");
    }
}
```

17.Parser

语法分析器整体结构



完整的语法分析程序构建完成：可以从分析来自String或者来自file的输入流。

```
public class Parser {
    public static ASTNode parse(String source) throws LexicalException, ParseException {
        var lexer = new Lexer();
        var tokens = lexer.analyse(new PeekIterator<>(source.chars().mapToObj(c ->(char)c), '\0'));
        return Program.parse(new PeekTokenIterator(tokens.stream()));
    }

    public static ASTNode fromFile(String file) throws FileNotFoundException, UnsupportedEncodingException, LexicalException, ParseException {
        var tokens = Lexer.fromFile(file);
        return Program.parse(new PeekTokenIterator(tokens.stream()));
    }
}
```

语法分析Example1：

输入文件：recursion.ts

```
func fact(int n)  int {
  if(n == 0) {
    return 1
  }
  return fact(n - 1) * n
}
```

具体方法：

```
@Test
public void function1() throws FileNotFoundException, UnsupportedEncodingException, LexicalException, ParseException {
    var tokens = Lexer.fromFile("./example/recursion.ts");
    var functionStmt = (FunctionDeclareStmt) Stmt.parseStmt(new PeekTokenIterator(tokens.stream()));
    functionStmt.print(0);

    assertEquals("func fact args block", ParserUtils.toBFSString(functionStmt, 4));
    assertEquals("args n", ParserUtils.toBFSString(functionStmt.getArgs(), 2));
    assertEquals("block if return", ParserUtils.toBFSString(functionStmt.getBlock(), 3));
}
```

Output：

```
print:parser.ast.FunctionDeclareStmt@516a3485

func fact args block
args n
block if return
fact n n 0 == 1 return if fact n 1 - n * return fact
```

语法分析Example2：

输入文件：

```
func add(int a, int b) int {
  return a + b
}
```

具体方法：

```
@Test
public void function() throws FileNotFoundException,
UnsupportedEncodingException, LexicalException, ParseException {
    var tokens = Lexer.fromFile("./example/function.ts");
    var functionStmt = (FunctionDeclareStmt) Stmt.parseStmt(new
PeekTokenIterator(tokens.stream()));
    functionStmt.print(0);

    var args = functionStmt.getArgs();
    assertEquals("a", args.getChild(0).getLexeme().get_value());
    assertEquals("b", args.getChild(1).getLexeme().get_value());

    var type = functionStmt.getFuncType();
    assertEquals("int", type);

    var functionVariable = functionStmt.getFunctionVariable();
    assertEquals("add", functionVariable.getLexeme().get_value());

    var block = functionStmt.getBlock();
    assertEquals(true, block.getChild(0) instanceof ReturnStmt);

}
```

Output:

```
print:parser.ast.FunctionDeclareStmt@5a1d3d8f

add a b a b + return add
func add args block
args a b
block return
```