Compiler Experiment : Simple Compiler

Author：软件81金之航

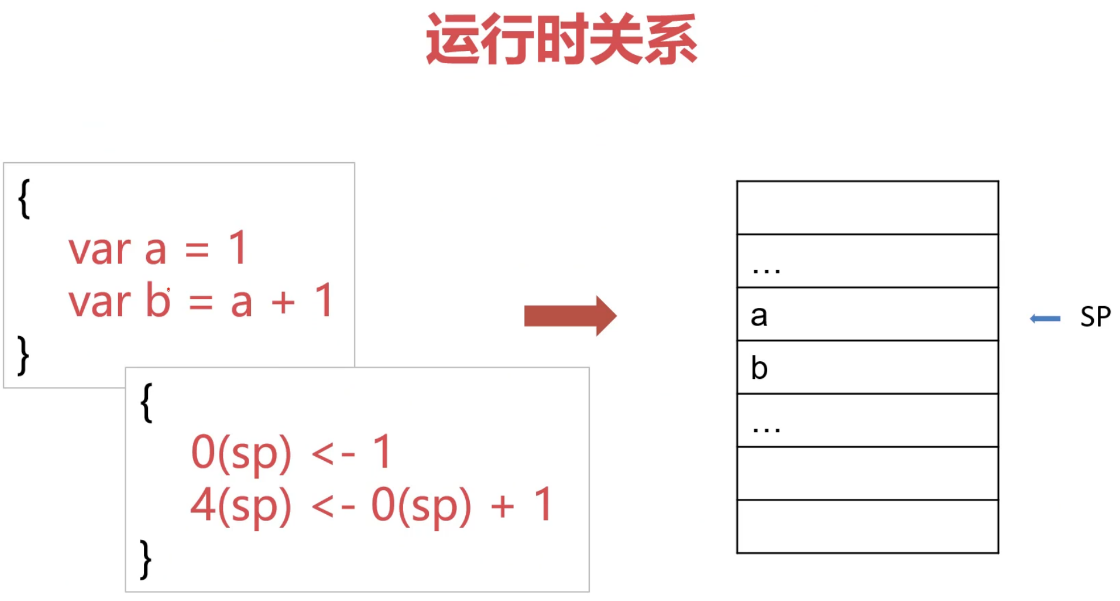
StuId：2183411101

三.语义分析和中间代码生成 Translator

语法制导定义 SDD(Syntax Directed Definition)：定义抽象语法树如何被翻译，文法（如何组织翻译程序? ），属性（用于存储结果和中间值），规则（描述属性如何被计算)。

词法作用域(Lexical Scope)：一个符号的可见范围称之为它的作用域，符号作用域和源代码的书写相关（词法)，并在运行时（实例）生效。

```
错误的作用域：                          正确的作用域：
b=100{                                 var b = 100{
  var b = a + 1                          b = a +1
}                                      }
```

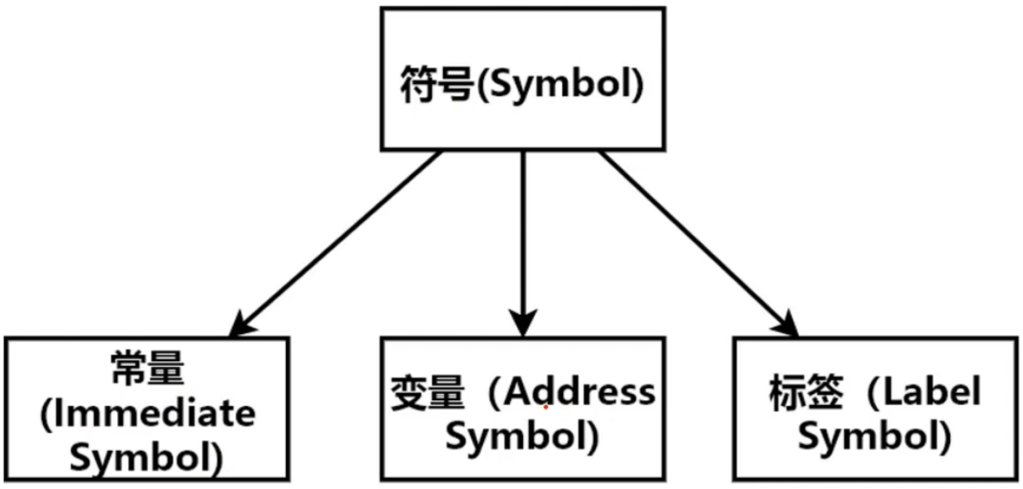


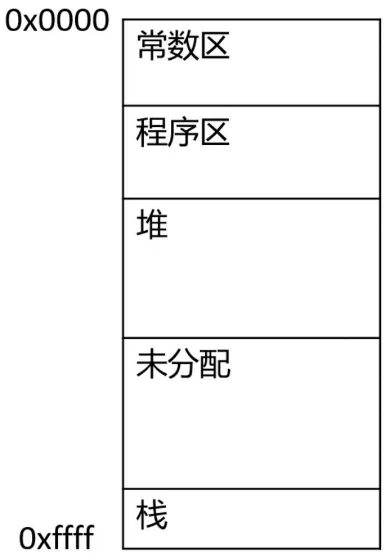
```
{
    var a = 1
    var b = a + 1
}
```

```
{
    0(sp) <- 1
    4(sp) <- 0(sp) + 1
}
```

一个变量的编译过程：
符号（词法） --> ASTNode --> 地址（三地址代码） --> 操作符（运行时环境）

符号表：用于存储符号（变量、常量、标签）在源代码中的位置、数据类型，以及位置信息决定的词法作用域和运行时的相对内存地址。eg：符号（变量、常量、标签），常量表，变量表。



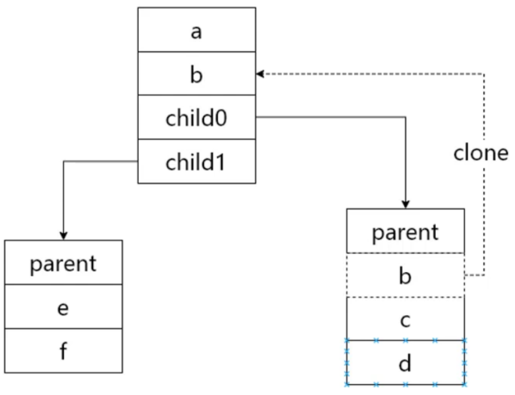


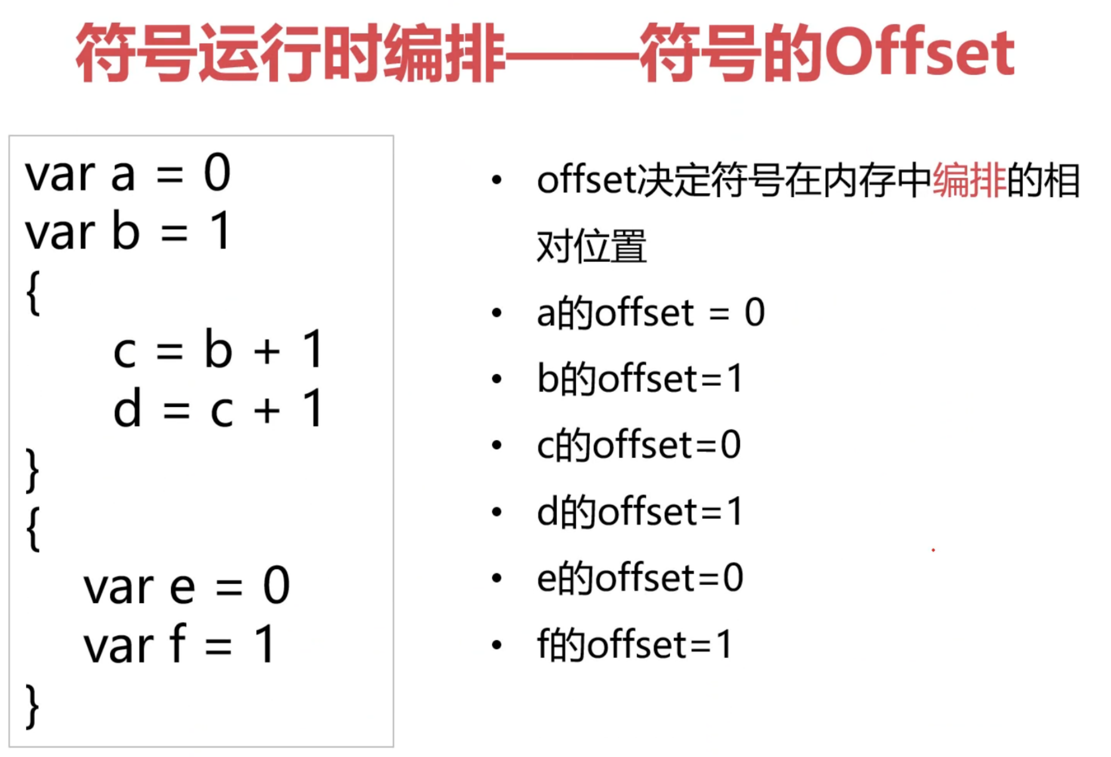
- 常数区：存储常数
- 程序区：字节码
- 堆：存储不规则数据（操作系统部分介绍）
- 未分配：堆自上而下分配，栈自下而上分配。
- 栈：存储变量等规则数据（每个32位）

静态符号表 SST(Static Symbol Table)：哈希表实现，用于存储常量在常量区的位置。
符号表 ST(Symbol Table)：树+哈希表实现，用于存储每个符号所在的词法作用域，以及它在词法作用域中的相对位置。

符号表示例

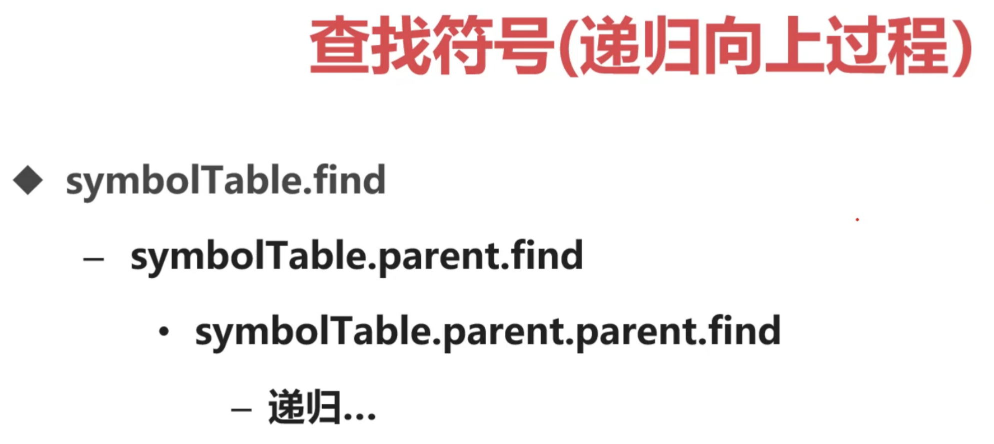
```
var a = 0
var b = 1
{
      c = b + 1
      d = c + 1
}
{
    var e = 0
    var f = 1
}
```




符号运行时编排——符号的Offset

```
var a = 0
var b = 1
{
      c = b + 1
      d = c + 1
}
{
    var e = 0
    var f = 1
}
```

- offset决定符号在内存中编排的相对位置
- a的offset = 0
- b的offset=1
- c的offset=0
- d的offset=1
- e的offset=0
- f的offset=1


查找符号(递归向上过程)

- **symbolTable.find**
  - **symbolTable.parent.find**
    - **symbolTable.parent.parent.find**
      - **递归…**

符号表的实现：

1.SymbolType

枚举类，含有符号Symbol类型：变量AddressSymbol、常量ImmediateSymbol、标签LabelSymbol

```
public enum SymbolType {
    ADDRESS_SYMBOL,
    IMMEDIATE_SYMBOL,
    LABEL_SYMBOL
}
```

2.Symbol

具体为三种符号的实现，“工厂实现”——createAddressSymbol(), createAddressSymbol(), createAddressSymbol

```java
@Data
//一个值或者变量的集合体
public class Symbol {

    SymbolTable parent;
    Token lexeme;
    String label;
    int offset;
    int layerOffset = 0;
    SymbolType type;
    public Symbol(SymbolType type){
        this.type = type;
    }

    public static Symbol createAddressSymbol(Token lexeme, int offset){
        var symbol = new Symbol(SymbolType.ADDRESS_SYMBOL);
        symbol.lexeme = lexeme;
        symbol.offset = offset;
        return symbol;
    }

    public static Symbol createAddressSymbol(Token lexeme){
        var symbol = new Symbol(SymbolType.IMMEDIATE_SYMBOL);
        symbol.lexeme = lexeme;
        return symbol;
    }

    public static Symbol createLabelSymbol(String label, Token lexeme) {
        var symbol = new Symbol(SymbolType.LABEL_SYMBOL);
        symbol.label = label;
        symbol.lexeme = lexeme;
        return symbol;
    }

    public Symbol copy() {
        var symbol = new Symbol(this.type);
        symbol.lexeme = this.lexeme;
        symbol.label = this.label;
        symbol.offset = this.offset;
        symbol.layerOffset = this.layerOffset;
        symbol.type = this.type;
        return symbol;
    }
    ...
}
```

3.SymbolTable

符号表的具体实现：

```java
public class SymbolTable {
    private SymbolTable parent = null;
    private ArrayList<SymbolTable> children;  //存放孩子节点
    private ArrayList<Symbol> symbols;  //存放Symbol
    private int tempIndex = 0;  //给临时变量计数
    private int offsetIndex = 0;  //给变量计数
    private int level = 0;  //

    public SymbolTable() {
        this.children = new ArrayList<>();
        this.symbols = new ArrayList<>();
    }

    public void addSymbol(Symbol symbol) {
        this.symbols.add(symbol);
        symbol.setParent(this);
    }

    /*
        var a = 1
        {
            {
                {
                    var b = a
                }
            }
        }作用域
    */
    public Symbol cloneFromSymbolTree(Token lexeme, int layerOffset) {
        var _symbol = this.symbols.stream()
                .filter(x ->
x.lexeme.get_value().equals(lexeme.get_value()))
                .findFirst();
        if (!_symbol.isEmpty()) {
            var symbol = _symbol.get().copy();
            symbol.setLayerOffset(layerOffset);
            return symbol;
        }
        if (this.parent != null) {
            return this.parent.cloneFromSymbolTree(lexeme, layerOffset + 1);
        }
        return null;
    }

    //判断当前符号表是否有Symbol
    public boolean exists(Token lexeme) {
        var _symbol = this.symbols.stream().filter(x ->
x.lexeme.get_value().equals(lexeme.get_value())).findFirst();
        if (!_symbol.isEmpty()) {
            return true;
        }
        if (this.parent != null) {
            return this.parent.exists(lexeme);
        }
        return false;
    }

    public Symbol createSymbolByLexeme(Token lexeme) {
        Symbol symbol = null;
        if (lexeme.isScalar()) {
            symbol = Symbol.createImmediateSymbol(lexeme);
            this.addSymbol(symbol);
        } else {
            var _symbol = this.symbols.stream().filter(x ->
x.getLexeme().get_value().equals(lexeme.get_value())).findFirst();
            if (_symbol.isEmpty()) {
                symbol = cloneFromSymbolTree(lexeme, 0);
                if (symbol == null) {
                    symbol = Symbol.createAddressSymbol(lexeme,
this.offsetIndex++);
                }
                this.addSymbol(symbol);
            } else {
                symbol = _symbol.get();
            }
        }
        return symbol;
    }

    public Symbol createVariable() {
        /*
        * var a = 1 + 2 * 3
        * p0 = 2 * 3
        * p1 = 1 + p0
        * */
        var lexeme = new Token(TokenType.VARIABLE, "p" + this.tempIndex++);
        var symbol = Symbol.createAddressSymbol(lexeme, this.offsetIndex++);
        this.addSymbol(symbol);
        return symbol;
    }
    ...
}
```

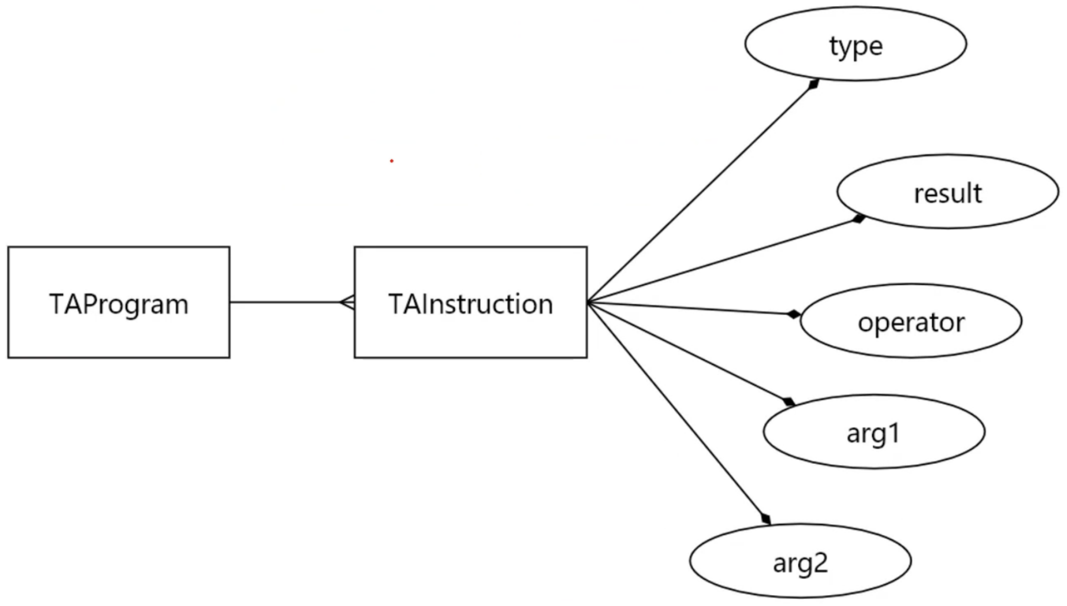
4.StaticSymbolTable

静态符号表

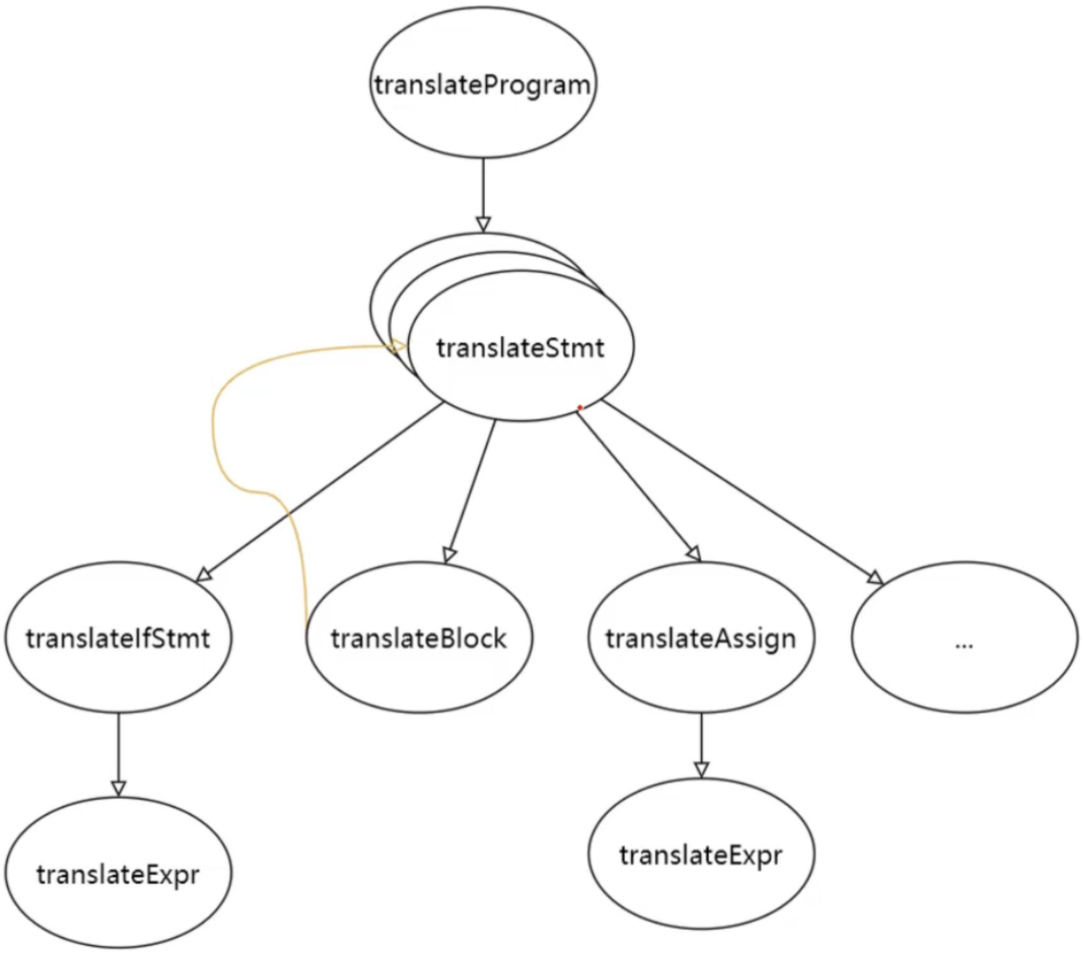
```java
public class StaticSymbolTable {

    private Hashtable<String, Symbol> offsetMap;
    private int offsetCounter = 0;
    private ArrayList<Symbol> symbols;


    public StaticSymbolTable(){
        symbols = new ArrayList<>();
        offsetMap = new Hashtable<>();
    }

    public void add(Symbol symbol){
        var lexval = symbol.getLexeme().get_value();
        if(!offsetMap.containsKey(lexval)) {
            offsetMap.put(lexval, symbol);
            symbol.setOffset(offsetCounter++);
```

```
                symbols.add(symbol);
            } else {
                var sameSymbol = offsetMap.get(lexval);
                symbol.setOffset(sameSymbol.offset);
            }
        }
        ...
}
```

三地址代码：



```
TAProgram: 三地址代码程序  1-->n  TAInstruction:三地址指令 --> type | result | oprator | arg1 | arg2
三地址指令五元组表示：(类型      返回值     操作符     操作数1    操作数2)
                eg:  Assign  a         =         b         1
                     Assign  p0        >         a         10
                     IF                          p0        L0(标签)
                     Assign  c                   100
                     GoTo                        L1(标签)
```



5.TAInstructionType

三地址指令类型：

```java
public enum TAInstructionType {
    ASSIGN,  //赋值
    GOTO,  //跳转
    IF,  //条件
    LABEL,  //标签
    CALL,  //函数调用
    RETURN,  //返回
    SP,  //栈指针
    PARAM,  //传参
    FUNC_BEGIN  //函数开始
}
```

6.TAInstruction

三地址指令：

```java
public class TAInstruction {
    private Object arg1;
    private Object arg2;
    private String op;
    private Symbol result;  //返回值，Symbol
    private TAInstructionType type;
    private String label = null;

    //三地址指令五元组表示
    public TAInstruction(TAInstructionType type, Symbol result, String op, Object arg1, Object arg2){
        this.op = op;
        this.type = type;
        this.arg1 = arg1;
        this.arg2 = arg2;
        this.result = result;
    }

    @Override
    public String toString() {
        switch (this.type) {
            case ASSIGN:
                if (arg2 != null) {
                    return String.format("%s = %s %s %s",result,arg1,op,arg2);
                } else {
                    return String.format("%s = %s",result,arg1);
                }
            case IF:
                return String.format("IF %s ELSE %s", this.arg1, this.arg2);
            case GOTO:
                return String.format("GOTO %s", this.arg1);
            case LABEL:
                return String.format(this.arg1 + ":");
            case FUNC_BEGIN:
                return "FUNC_BEGIN";
            case RETURN:
                return "RETURN " + this.arg1;
            case PARAM:
                return "PARAM " + this.arg1 + " " + this.arg2;
            case SP:
                return "SP " + this.arg1;
            case CALL:
                return "CALL " + this.arg1;
        }
        throw new NotImplementedException("Unkonw opcode type:" + this.type);
    }
    ...
}
```

7.TAProgram

三地址程序:

```
public class TAProgram {
    private ArrayList<TAInstruction> instructions = new ArrayList<>();  //存储指令的ArrayList
    private int labelCounter = 0;  //L0: 给label计数
```

```java
    private StaticSymbolTable staticSymbolTable = new StaticSymbolTable();
 //静态符号表

    public void add(TAInstruction code) {
        instructions.add(code);
    }

    public ArrayList<TAInstruction> getInstructions() {
        return instructions;
    }

    @Override
    public String toString() {
        var lines = new ArrayList<String>();
        for (var opcode : instructions) {
            lines.add(opcode.toString());
        }
        return StringUtils.join(lines, "\n");
    }

    public TAInstruction addLabel() {
        var label = "L" + labelCounter++;
        var taCode = new TAInstruction(TAInstructionType.LABEL, null, null,
null, null);
        taCode.setArg1(label);
        instructions.add(taCode);
        return taCode;
    }

    public void setStaticSymbols(SymbolTable symbolTable) {
        for (var symbol : symbolTable.getSymbols()) {
            if (symbol.getType() == SymbolType.IMMEDIATE_SYMBOL) {
                staticSymbolTable.add(symbol);
            }
        }
        for (var child : symbolTable.getChildren()) {
            setStaticSymbols(child);
        }
    }

    public StaticSymbolTable getStaticSymbolTable() {
        return this.staticSymbolTable;
    }
}
```

8.Translator

完整的语义分析以及三地址转换程序:

```java
public class Translator {
    public TAProgram translate(ASTNode astNode) throws ParseException {
        var program = new TAProgram();
        var symbolTable = new SymbolTable();
        for (var child : astNode.getChildren()) {
            translateStmt(program, child, symbolTable);
        }
        program.setStaticSymbols(symbolTable);
        var main = new Token(TokenType.VARIABLE, "main");
        if (symbolTable.exists(main)) {
            symbolTable.createVariable(); // 返回值
            program.add(new TAInstruction(TAInstructionType.SP, null, null,
                    -symbolTable.localSize(), null));
            program.add(new TAInstruction(
                    TAInstructionType.CALL, null, null,
                    symbolTable.cloneFromSymbolTree(main, 0), null));
            program.add(new TAInstruction(TAInstructionType.SP, null, null,
                    symbolTable.localSize(), null));
        }
        return program;
    }

    //语句块翻译
    public void translateBlock(TAProgram program, Block block, SymbolTable parent) throws ParseException {
        var symbolTable = new SymbolTable();
        parent.addChild(symbolTable);
        //每个Block增加一个作用域链
        var parentOffset = symbolTable.createVariable();
        parentOffset.setLexeme(new Token(TokenType.INTEGER, symbolTable.localSize() + ""));

        for (var child : block.getChildren()) {
            translateStmt(program, child, symbolTable);
        }
    }

    //翻译各种语句
    public void translateStmt(TAProgram program, ASTNode node, SymbolTable symbolTable) throws ParseException {
        switch (node.getType()) {
            case BLOCK:
                translateBlock(program, (Block) node, symbolTable);
                return;
            case IF_STMT:
                translateIfStmt(program, (IfStmt) node, symbolTable);
                return;
            case ASSIGN_STMT:
                translateAssignStmt(program, node, symbolTable);
                return;
            case DECLARE_STMT:
                translateDeclareStmt(program, node, symbolTable);
                return;
            case FUNCTION_DECLARE_STMT:
                translateFunctionDeclareStmt(program, node, symbolTable);
                return;
            case RETURN_STMT:
                translateReturnStmt(program, node, symbolTable);
                return;
            case CALL_EXPR:
                translateCallExpr(program, node, symbolTable);
                return;
        }
        throw new NotImplementedException("Translator not impl. for " + node.getType());
    }


    /**
     * IF语句翻译成三地址代码
     * 1. 表达式
     * 2. 语句块
     * 3. else Tail处理
     */
    public void translateIfStmt(TAProgram program, IfStmt node, SymbolTable symbolTable) throws ParseException {
        var expr = node.getExpr();
        var exprAddr = translateExpr(program, expr, symbolTable);
        var ifOpCode = new TAInstruction(TAInstructionType.IF, null, null, exprAddr, null);
        program.add(ifOpCode);

        translateBlock(program, (Block) node.getBlock(), symbolTable);

        TAInstruction gotoInstruction = null;

        //if(expr) {...} else {...} | if(expr) {...} else if(expr) {...}
        if (node.getChild(2) != null) {
            gotoInstruction = new TAInstruction(TAInstructionType.GOTO, null, null, null, null);
            program.add(gotoInstruction);
            var labelEndIf = program.addLabel();
            ifOpCode.setArg2(labelEndIf.getArg1());
        }

        if (node.getElseBlock() != null) {
            translateBlock(program, (Block) node.getElseBlock(), symbolTable);
        } else if (node.getElseIfStmt() != null) {
            translateIfStmt(program, (IfStmt) node.getElseIfStmt(), symbolTable);
        }

        var labelEnd = program.addLabel();
        if (node.getChild(2) == null) {
            ifOpCode.setArg2(labelEnd.getArg1());
        } else {
            gotoInstruction.setArg1(labelEnd.getArg1());
        }
    }

    //翻译返回语句
    private void translateReturnStmt(TAProgram program, ASTNode node, SymbolTable symbolTable) throws ParseException {
        ...
    }

    //翻译函数定义语句
    private void translateFunctionDeclareStmt(TAProgram program, ASTNode node, SymbolTable parent) throws ParseException {
        ...
    }

    //翻译定义语句
    private void translateDeclareStmt(TAProgram program, ASTNode node,
SymbolTable symbolTable) throws ParseException {
    ...
    }

    //翻译表达式
    public Symbol translateExpr(
        ...
    }

    //翻译调用语句
    private Symbol translateCallExpr(TAProgram program, ASTNode node,
SymbolTable symbolTable) throws ParseException {
          ...
    }
}
```

9.Example

中间代码生成Example1(计算表达式):

```
@Test
    public void transExpr() throws LexicalException, ParseException {
        var source = "a+(b-c)+d*(b-c)*2";
        var p = Parser.parse(source);
        p.print(0);
        var exprNode = p.getChild(0);

        var translator = new Translator();
        var symbolTable = new SymbolTable();
        var program = new TAProgram();
        translator.translateExpr(program, exprNode, symbolTable);
        System.out.println(program.toString());
        var expectedResults = new String[]{
                "p0 = b - c",
                "p1 = b - c",
                "p2 = p1 * 2",
                "p3 = d * p2",
                "p4 = p0 + p3",
                "p5 = a + p4"
        };
        assertOpcodes(expectedResults, program.getInstructions());
    }
```

输出Output1：

```
print:parser.ast.Program@213c7a36

p0 = b - c
p1 = b - c
p2 = p1 * 2
p3 = d * p2
p4 = p0 + p3
p5 = a + p4
```

中间代码生成Example2(If语句):

输入:

```
if(a == 1) {
  b = 100
} else if(a == 2) {
  b = 500
} else if(a == 3) {
  b = a * 1000
} else {
  b = -1
}
```

测试方法:

```
    @Test
    public void testIfElseIf() throws FileNotFoundException, ParseException, LexicalException, UnsupportedEncodingException {
        var astNode = Parser.fromFile("./example/complex-if.ts");
        var translator = new Translator();
        var program = translator.translate(astNode);
        System.out.println(program.toString());

        var expected = "p0 = a == 1\n" +
                "IF p0 ELSE L0\n" +
                "b = 100\n" +
                "GOTO L5\n" +
                "L0:\n" +
                "p1 = a == 2\n" +
                "IF p1 ELSE L1\n" +
                "b = 500\n" +
                "GOTO L4\n" +
                "L1:\n" +
                "p2 = a == 3\n" +
                "IF p2 ELSE L2\n" +
                "p1 = a * 1000\n" +
                "b = p1\n" +
                "GOTO L3\n" +
                "L2:\n" +
                "b = -1\n" +
                "L3:\n" +
                "L4:\n" +
                "L5:";
        assertEquals(expected, program.toString());
    }
```

输出Output2：

```
print:parser.ast.Program@184d2ac2

p0 = a == 1
IF p0 ELSE L0
b = 100
GOTO L5
L0:
p1 = a == 2
IF p1 ELSE L1
b = 500
GOTO L4
L1:
p2 = a == 3
IF p2 ELSE L2
p1 = a * 1000
b = p1
GOTO L3
L2:
b = -1
L3:
L4:
L5:
```

中间代码生成Example3(函数)：

输入：

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

测试方法：

```
    @Test
    public void testRecursiveFunction() throws FileNotFoundException, ParseException, LexicalException, UnsupportedEncodingException {
        var astNode = Parser.fromFile("./example/recursion.ts");
        var translator = new Translator();
        var program = translator.translate(astNode);
        System.out.println(program.toString());

        var expect = "L0:\n" +
                "FUNC_BEGIN\n" +
                "p1 = n == 0\n" +
                "IF p1 ELSE L1\n" +
                "RETURN 1\n" +
                "L1:\n" +
                "p2 = n - 1\n" +
                "PARAM p2 6\n" +
                "SP -5\n" +
                "CALL L0\n" +
                "SP 5\n" +
                "p4 = p3 * n\n" +
                "RETURN p4";
        assertEquals(expect, program.toString());
    }
```

输出Output2：

```
print:parser.ast.Program@184d2ac2

L0:
FUNC_BEGIN
p1 = n == 0
IF p1 ELSE L1
RETURN 1
L1:
p2 = n - 1
PARAM p2 6
SP -5
CALL L0
SP 5
p4 = p3 * n
RETURN p4
```