湖南大學
HUNAN UNIVERSITY

课程实验报告

课　程　名　称：编译技术

实验项目名称：TINY 语言编译器的实现

专　业　班　级：软件 2105 班

姓　　名：马小梅

学　　号：202126010530

指　导　教　师：杨金民

完　成　时　间：2024 年 6 月 5 日

信息科学与工程学院

实验题目：

（1）词法分析器构造工具的实现

实验目的：

基于第 5 章的课程知识，以及前面 3 个实验的结果，写出一个完整的 TINY 语言编译器，该编译器的输入为一文本文件，即 TINY 源程序，其输出也是一个文本文件，即以 tm 中间语言写出的源程序，其后缀名为 tm。TM.exe 为一个虚拟机，通过它来运行 tm 程序，得出计算结果。运行 TM.exe 的时候，输入是一个文本文件，即 tm 程序，输出为输入程序的运行结果。TINY 源程序的样例文件为 sample.tny，对其编译后得到 sample.tm 文件。使用 TM.exe 运行 sample.tm，便能得到 sample.tm 的运行结果。

实验环境：

Windows 系统，DevC++

实验任务：

1. 写出 TINY 语言的语义分析和中间代码生成的 SDT。
2. 基于 TINY 语言的语义分析和中间代码生成的 SDT，得出 TINY 语言编译器的完整源代码。
3. 以 sample.tny 作为输入，得出输出 sample.tm，验证所写 TINY 语言编译器的正确性。
4. 用 TM.exe 程序运行 sample.tm 程序，得出 sample.tm 程序的运算结果。

实验步骤：

一、 前提了解

阅读 *TM* 虚拟机的有关文档，了解 *TM* 机的指令结构与寻址方式等相关内容

Tiny Machine 由只读指令存储区、数据存储区、8 个通用寄存器

```
/******* const *******/
#define   IADDR_SIZE  1024 /* increase for large programs */
#define   DADDR_SIZE  1024 /* increase for large programs */
#define   NO_REGS 8
#define   PC_REG  7
```

```
INSTRUCTION iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

NO_REGS 表示通用寄存器的个数，PC_REG 表示程序计数器
iMem[IADDR_SIZE]定义了只读指令存储区大小
dMem[DADDR_SIZE]定义了数据存储区大小
reg[NO-REGS]定义了通用寄存器

下图为 TM 的指令集。基本指令格式有两种：寄存器，即 RO 指令。寄存器-存储器，即 RM 指令。

RO 指令

格式	*opcode r,s,t*
操作码	效果
HALT	停止执行(忽略操作数)
IN	reg[r] ← 从标准读入整形值(s和t忽略)
OUT	reg[r] → 标准输出(s和t忽略)
ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t](可能产生ZERO_DIV)

RM 指令

格式 *opcode r,d(s)*

(*a=d+reg[s]*; 任何对dmem[a]的引用在a<0或a≥DADDR-SIZE时产生DMEM-ERR)

操作码	效果
LD	reg[r] = dMem[a](将a中的值装入r)
LDA	reg[r] = a (将地址a直接装入r)
LDC	reg[r] = d (将常数d直接装入r, 忽略s)
ST	dMem[a] = reg[r](将r的值存入位置a)
JLT	if (reg[r]<0) reg [PC_REG] =a (如果r小于零转移到a，以下类似)
JLE	if (reg[r]<=0 reg [PC_REG] =a
JGE	if (reg[r]>0) reg [PC_REG] =a
JGT	if (reg[r]>0) reg [PC_REG] =a
TEQ	if (reg[r]==0) reg [PC_REG] =a
JNE	if (reg[r]!=0) reg [PC_REG] =a

TM 执行一个常用的取指令-执行循环。

在开始点,TM 将所有寄存器和数据区设为 0，然后将最高正规地址的值（名为 DADDR_SIZE-1)装入到 dMem[0]中。(由于程序可以知道在执行时可用内存的数量，所以它允许将存储器很便利地添加到 TM 上)。TM 然后开始执行 iMem[0]指令。机器在执行到 HALT 指令时停止。可能的错误条件包括 IMEM_ERR(它发生在取指步骤中若 reg[PC_REG]<0 或 reg[PC_REG]≥IADDR_SIZE 时)，以及两个条件 DMEM_ERR 和 ZERO-DIV。

二、 基本变量

（1）Tiny 的 token 类型以及内容

```
public enum TokenType {
    3 个用法
    IF, ELSE, THEN, END, REPEAT, UNTIL, READ, WRITE,
    2 个用法
    ASSIGN, LESS, EQ, PLUS, MINUS, MULTI, DIV,ADD,SUB,MUL,
    3 个用法
    NUM, ID, ERR, LPAR, RPAR, COLON
}
```

```java
public class Token {
    3 个用法
    private TokenType type;    // Token的类型，由TokenType枚举定义
    3 个用法
    private String text;       // Token的文本内容
    3 个用法
    private int line;          // Token出现在源代码中的行号，用于错误报告

    4 个用法
    public Token(TokenType type, String text, int line) {
        this.type = type;
        this.text = text;
        this.line = line;
    }
```

（2） Tiny 状态

```java
public enum StateType {
    4 个用法
    START, DONE, ERROR, INID, INNUM, INASSIGN, INCOMMENT
}
```

（3） Tiny 类型判断

```java
2 个用法
static boolean isID(char c) { return Character.isLetter(c); }

2 个用法
static boolean isNUM(char c) { return Character.isDigit(c); }

1 个用法
static boolean isOperator(char c) {...}

1 个用法
static boolean isWhiteSpace(char c) { return c == ' ' || c == '\t'; }

2 个用法
static TokenType identifyReserved(String s) {...}
```

（4） TM 指令

```java
    public Instruction(String op, int r, int d, int s, String comment) {
        this.op = op;
        this.r = r;
        this.d = d;
        this.s = s;
        this.comment = comment;
    }
```

```
0 个用法
public enum TMOpcode {
    0 个用法
    LOAD, STORE, ADD, SUB, MUL, DIV, READ, WRITE, BR, BRPOS, BRZERO, HALT
}
```

三、实验思路

（一）扫描 sample 文件进行 token 识别和状态设置

通过扫描每一行 识别字符串进行判断：数字、关键词等设为相应的 token 并设置对应的初始状态。DONE 状态进行每一类 token 的识别输出。

```
static List<Token> scanToken(String line) {
    List<Token> tokens = new ArrayList<>();
    res = "";
    linepos = 0;
    linesize = line.length();
    System.out.printf("Scanning line: %s%n", line);
    while (linepos < linesize) {
        saveflag = true;
        char ch = line.charAt(linepos);
        System.out.printf("State: %s, Char: '%c', LinePos: %d%n", stat
        switch (state) {
            case START:
                if (isWhiteSpace(ch)) {
                    saveflag = false;
                } else if (isID(ch)) {
                    state = StateType.INID;
                } else if (isNUM(ch)) {
                    state = StateType.INNUM;
                } else if (ch == ':') {
                    state = StateType.INASSIGN;
                } else if (ch == '{') {
                    saveflag = false;
                    state = StateType.INCOMMENT;
                } else if (isOperator(ch)) {
                    state = StateType.DONE;
                    token = identifyOperator(ch);
                } else {
                    state = StateType.ERROR;
                }
                break;
            case INID:
                if (!isID(ch)) {
                    state = StateType.DONE;
                    linepos--;
                    saveflag = false;
                    token = identifyReserved(res); // 检查是否是保留字
                }
                break;
            case INNUM:
                if (!isNUM(ch)) {
                    state = StateType.DONE;
                    linepos--;
                    saveflag = false;
                    token = TokenType.NUM;
                }
                break;
            case INASSIGN:
                if (ch == '=') {
                    state = StateType.DONE;
                    token = TokenType.ASSIGN;
                } else {
                    state = StateType.ERROR;
                    linepos--;
                    saveflag = false;
                }
                break;
            case INCOMMENT:
                if (ch == '}') {
                    state = StateType.START;
                    saveflag = false;
                } else {
                    saveflag = false;
                }
                break;
            case ERROR:
                state = StateType.DONE;
                linepos--;
                saveflag = false;
                token = TokenType.ERR;
                break;
            default:
                break;
        }
        linepos++;
        if (saveflag) {
            res += ch;
        }
```

```
            if (state == StateType.DONE) {
                if (!res.trim().isEmpty()) {
                    System.out.printf("Token identified: Type=%s, Text='%s'%n", token, res.trim());
                    tokens.add(new Token(token, res.trim(), lineno));
                }
                res = "";
                state = StateType.START;
            }
        }
        // 处理行末的token
        if (state == StateType.INID) {
            token = identifyReserved(res);
            tokens.add(new Token(token, res.trim(), lineno));
            System.out.printf("End of line token: Type=%s, Text='%s'%n", token, res.trim());
        } else if (state == StateType.INNUM) {
            token = TokenType.NUM;
            tokens.add(new Token(token, res.trim(), lineno));
            System.out.printf("End of line token: Type=%s, Text='%s'%n", token, res.trim());
        } else if (state == StateType.INASSIGN) {
            token = TokenType.ERR;
            tokens.add(new Token(token, res.trim(), lineno));
            System.out.printf("End of line token: Type=%s, Text='%s'%n", token, res.trim());
        }
        return tokens;
    }
```

识别结果如下：

```
Scanning line: { Sample program
State: START, Char: '{', LinePos: 0
State: INCOMMENT, Char: ' ', LinePos: 1
State: INCOMMENT, Char: 'S', LinePos: 2
State: INCOMMENT, Char: 'a', LinePos: 3
State: INCOMMENT, Char: 'm', LinePos: 4
State: INCOMMENT, Char: 'p', LinePos: 5
State: INCOMMENT, Char: 'l', LinePos: 6
State: INCOMMENT, Char: 'e', LinePos: 7
State: INCOMMENT, Char: ' ', LinePos: 8
State: INCOMMENT, Char: 'p', LinePos: 9
State: INCOMMENT, Char: 'r', LinePos: 10
State: INCOMMENT, Char: 'o', LinePos: 11
State: INCOMMENT, Char: 'g', LinePos: 12
State: INCOMMENT, Char: 'r', LinePos: 13
State: INCOMMENT, Char: 'a', LinePos: 14
State: INCOMMENT, Char: 'm', LinePos: 15
Scanning line:   in TINY language -
State: INCOMMENT, Char: ' ', LinePos: 0
State: INCOMMENT, Char: ' ', LinePos: 1
State: INCOMMENT, Char: 'i', LinePos: 2
State: INCOMMENT, Char: 'n', LinePos: 3
State: INCOMMENT, Char: ' ', LinePos: 4
State: INCOMMENT, Char: 'T', LinePos: 5
State: INCOMMENT, Char: 'I', LinePos: 6
State: INCOMMENT, Char: 'N', LinePos: 7
State: INCOMMENT, Char: 'Y', LinePos: 8
State: INCOMMENT, Char: ' ', LinePos: 9
State: INCOMMENT, Char: 'l', LinePos: 10
State: INCOMMENT, Char: 'a', LinePos: 11
State: INCOMMENT, Char: 'n', LinePos: 12
State: INCOMMENT, Char: 'g', LinePos: 13
```

```
State: INCOMMENT, Char: 'a', LinePos: 15
State: INCOMMENT, Char: 'g', LinePos: 16
State: INCOMMENT, Char: 'e', LinePos: 17
State: INCOMMENT, Char: ' ', LinePos: 18
State: INCOMMENT, Char: '-', LinePos: 19
Scanning line:   computes factorial
State: INCOMMENT, Char: ' ', LinePos: 0
State: INCOMMENT, Char: ' ', LinePos: 1
State: INCOMMENT, Char: 'c', LinePos: 2
State: INCOMMENT, Char: 'o', LinePos: 3
State: INCOMMENT, Char: 'm', LinePos: 4
State: INCOMMENT, Char: 'p', LinePos: 5
State: INCOMMENT, Char: 'u', LinePos: 6
State: INCOMMENT, Char: 't', LinePos: 7
State: INCOMMENT, Char: 'e', LinePos: 8
State: INCOMMENT, Char: 's', LinePos: 9
State: INCOMMENT, Char: ' ', LinePos: 10
State: INCOMMENT, Char: 'f', LinePos: 11
State: INCOMMENT, Char: 'a', LinePos: 12
State: INCOMMENT, Char: 'c', LinePos: 13
State: INCOMMENT, Char: 't', LinePos: 14
State: INCOMMENT, Char: 'o', LinePos: 15
State: INCOMMENT, Char: 'r', LinePos: 16
State: INCOMMENT, Char: 'i', LinePos: 17
State: INCOMMENT, Char: 'a', LinePos: 18
State: INCOMMENT, Char: 'l', LinePos: 19
Scanning line: }
State: INCOMMENT, Char: '}', LinePos: 0
Scanning line: read x; { input an integer }
State: START, Char: 'r', LinePos: 0
State: INID, Char: 'e', LinePos: 1
State: INID, Char: 'a', LinePos: 2
State: INID, Char: 'd', LinePos: 3
State: INID, Char: ' ', LinePos: 4
```

```
State: START, Char: 'x', LinePos: 5
State: INID, Char: ';', LinePos: 6
Token identified: Type=ID, Text='x'
State: START, Char: ';', LinePos: 6
Token identified: Type=COLON, Text=';'
State: START, Char: ' ', LinePos: 7
State: START, Char: '{', LinePos: 8
State: INCOMMENT, Char: ' ', LinePos: 9
State: INCOMMENT, Char: 'i', LinePos: 10
State: INCOMMENT, Char: 'n', LinePos: 11
State: INCOMMENT, Char: 'p', LinePos: 12
State: INCOMMENT, Char: 'u', LinePos: 13
State: INCOMMENT, Char: 't', LinePos: 14
State: INCOMMENT, Char: ' ', LinePos: 15
State: INCOMMENT, Char: 'a', LinePos: 16
State: INCOMMENT, Char: 'n', LinePos: 17
State: INCOMMENT, Char: ' ', LinePos: 18
State: INCOMMENT, Char: 'i', LinePos: 19
State: INCOMMENT, Char: 'n', LinePos: 20
State: INCOMMENT, Char: 't', LinePos: 21
State: INCOMMENT, Char: 'e', LinePos: 22
State: INCOMMENT, Char: 'g', LinePos: 23
State: INCOMMENT, Char: 'e', LinePos: 24
State: INCOMMENT, Char: 'r', LinePos: 25
State: INCOMMENT, Char: ' ', LinePos: 26
State: INCOMMENT, Char: '}', LinePos: 27
Scanning line: if 0 < x then { don't compute if x <= 0 }
State: START, Char: 'i', LinePos: 0
State: INID, Char: 'f', LinePos: 1
State: INID, Char: ' ', LinePos: 2
Token identified: Type=IF, Text='if'
State: START, Char: ' ', LinePos: 2
State: START, Char: '0', LinePos: 3
State: INNUM, Char: ' ', LinePos: 4
```

```
Token identified: Type=NUM, Text='0'
State: START, Char: ' ', LinePos: 4
State: START, Char: '<', LinePos: 5
Token identified: Type=LESS, Text='<'
State: START, Char: ' ', LinePos: 6
State: START, Char: 'x', LinePos: 7
State: INID, Char: ' ', LinePos: 8
Token identified: Type=ID, Text='x'
State: START, Char: ' ', LinePos: 8
State: START, Char: 't', LinePos: 9
State: INID, Char: 'h', LinePos: 10
State: INID, Char: 'e', LinePos: 11
State: INID, Char: 'n', LinePos: 12
State: INID, Char: ' ', LinePos: 13
Token identified: Type=THEN, Text='then'
State: START, Char: ' ', LinePos: 13
State: START, Char: '{', LinePos: 14
State: INCOMMENT, Char: ' ', LinePos: 15
State: INCOMMENT, Char: 'd', LinePos: 16
State: INCOMMENT, Char: 'o', LinePos: 17
State: INCOMMENT, Char: 'n', LinePos: 18
State: INCOMMENT, Char: ''', LinePos: 19
State: INCOMMENT, Char: 't', LinePos: 20
State: INCOMMENT, Char: ' ', LinePos: 21
State: INCOMMENT, Char: 'c', LinePos: 22
State: INCOMMENT, Char: 'o', LinePos: 23
State: INCOMMENT, Char: 'm', LinePos: 24
State: INCOMMENT, Char: 'p', LinePos: 25
State: INCOMMENT, Char: 'u', LinePos: 26
State: INCOMMENT, Char: 't', LinePos: 27
State: INCOMMENT, Char: 'e', LinePos: 28
State: INCOMMENT, Char: ' ', LinePos: 29
State: INCOMMENT, Char: 'i', LinePos: 30
State: INCOMMENT, Char: 'f', LinePos: 31
```

```
Scanning line:   fact := 1;
State: START, Char: ' ', LinePos: 0
State: START, Char: ' ', LinePos: 1
State: START, Char: 'f', LinePos: 2
State: INID, Char: 'a', LinePos: 3
State: INID, Char: 'c', LinePos: 4
State: INID, Char: 't', LinePos: 5
State: INID, Char: ' ', LinePos: 6
Token identified: Type=ID, Text='fact'
State: START, Char: ' ', LinePos: 6
State: START, Char: ':', LinePos: 7
State: INASSIGN, Char: '=', LinePos: 8
Token identified: Type=ASSIGN, Text=':='
State: START, Char: ' ', LinePos: 9
State: START, Char: '1', LinePos: 10
State: INNUM, Char: ';', LinePos: 11
Token identified: Type=NUM, Text='1'
State: START, Char: ';', LinePos: 11
Token identified: Type=COLON, Text=';'
Scanning line:   repeat
State: START, Char: ' ', LinePos: 0
State: START, Char: ' ', LinePos: 1
State: START, Char: 'r', LinePos: 2
State: INID, Char: 'e', LinePos: 3
```

```
Scanning line:     fact := fact * x;
State: INID, Char: ' ', LinePos: 0
State: START, Char: ' ', LinePos: 0
State: START, Char: ' ', LinePos: 1
State: START, Char: ' ', LinePos: 2
State: START, Char: ' ', LinePos: 3
State: START, Char: 'f', LinePos: 4
State: INID, Char: 'a', LinePos: 5
State: INID, Char: 'c', LinePos: 6
State: INID, Char: 't', LinePos: 7
State: INID, Char: ' ', LinePos: 8
Token identified: Type=ID, Text='fact'
State: START, Char: ' ', LinePos: 8
State: START, Char: ':', LinePos: 9
State: INASSIGN, Char: '=', LinePos: 10
Token identified: Type=ASSIGN, Text=':='
State: START, Char: ' ', LinePos: 11
State: START, Char: 'f', LinePos: 12
State: INID, Char: 'a', LinePos: 13
State: INID, Char: 'c', LinePos: 14
State: INID, Char: 't', LinePos: 15
State: INID, Char: ' ', LinePos: 16
Token identified: Type=ID, Text='fact'
State: START, Char: ' ', LinePos: 16
State: START, Char: '*', LinePos: 17
Token identified: Type=MULTI, Text='*'
```

```
State: INID, Char: 'p', LinePos: 4
State: INID, Char: 'e', LinePos: 5
State: INID, Char: 'a', LinePos: 6
State: INID, Char: 't', LinePos: 7
End of line token: Type=REPEAT, Text='repeat'
Scanning line:     fact := fact * x;
State: INID, Char: ' ', LinePos: 0
```

```
State: START, Char: ' ', LinePos: 18
State: START, Char: 'x', LinePos: 19
State: INID, Char: ';', LinePos: 20
Token identified: Type=ID, Text='x'
State: START, Char: ';', LinePos: 20
Token identified: Type=COLON, Text=';'
Scanning line:     x := x - 1;
State: START, Char: ' ', LinePos: 0
```

（二）语法分析

```java
public Program parse() {
    System.out.println("Starting parse......");
    List<Statement> statements = new ArrayList<>();
    while (!isAtEnd()) {
        statements.add(parseStatement());
    }
    System.out.println("Completed parse.");
    return new Program(statements);
}
```

```java
private Statement parseStatement() {
    Token token = peek();
    System.out.println();
    System.out.println("Parsing statement starting with token:  " + token);
    switch (token.getType()) {
        case IF:
            return parseIfStatement();
        case REPEAT:
            return parseRepeatStatement();
        case ID:
            return parseAssignmentStatement();
        case READ:
            return parseReadStatement();
        case WRITE:
            return parseWriteStatement();
        default:
            throw new RuntimeException("Unexpected token: " + token.getType() + " at line " + token.getLine());
    }
}
```

思路：对每一个 statement 根据类型进行语法分析。

if 语句：保存表达式的左右操作数，进行表达式语法分析，然后对 THEN 和 ELSE 语句进行分析。

```java
private IfStatement parseIfStatement() {
    System.out.println("Parsing if statement...");
    Token ifToken = consume(TokenType.IF, errorMessage: "Expect 'if'");
    Expression condition = parseExpression();
    consume(TokenType.THEN, errorMessage: "Expect 'then'");
    List<Statement> thenStatements = new ArrayList<>();
    while (!check(TokenType.ELSE) && !check(TokenType.END) && !isAtEnd()) {
        thenStatements.add(parseStatement());
    }
    List<Statement> elseStatements = new ArrayList<>();
    if (check(TokenType.ELSE)) {
        advance();
        while (!check(TokenType.END) && !isAtEnd()) {
            elseStatements.add(parseStatement());
        }
    }
    consume(TokenType.END, errorMessage: "Expect 'end'");
    System.out.println("Completed if statement.");
    return new IfStatement(condition, thenStatements, elseStatements, ifToken.getLine());
}
```

分析结果：

```
Parsing statement starting with token:  Token(IF, 'if', line 6)
Parsing if statement...
Consuming token: Token(IF, 'if', line 6)
Parsing expression...
Parsing simple expression...
Parsing term...
Parsing factor...
Number: 0
Completed term.
Completed simple expression.
Parsing simple expression...
Parsing term...
Parsing factor...
Variable: x
Completed term.
Completed simple expression.
Completed expression.
Consuming token: Token(THEN, 'then', line 6)
```

READ 和 WRITE 语句：

```
private ReadStatement parseReadStatement() {
    System.out.println("Parsing read statement...");
    Token readToken = consume(TokenType.READ,  errorMessage: "Expect 'read'");
    Token idToken = consume(TokenType.ID,  errorMessage: "Expect variable name");
    consume(TokenType.COLON,  errorMessage: "Expect ';'");
    System.out.println("Completed read statement.");
    return new ReadStatement(idToken.getText(), readToken.getLine());
}

1 个用法
private WriteStatement parseWriteStatement() {
    System.out.println("Parsing write statement...");
    Token writeToken = consume(TokenType.WRITE,  errorMessage: "Expect 'write'");
    Expression expression = parseExpression();
    System.out.println("Completed write statement.");
    return new WriteStatement(expression, writeToken.getLine());
}
```

分析结果：

```
Parsing statement starting with token:  Token(READ, 'read', line 5)
Parsing read statement...
Consuming token: Token(READ, 'read', line 5)
Consuming token: Token(ID, 'x', line 5)
Consuming token: Token(COLON, ';', line 5)
Completed read statement.
```

```
Parsing statement starting with token:  Token(WRITE, 'write', line 12)
Parsing write statement...
Consuming token: Token(WRITE, 'write', line 12)
Parsing expression...
Parsing simple expression...
Parsing term...
Parsing factor...
Variable: fact
Completed term.
Completed simple expression.
Completed expression.
Completed write statement.
Consuming token: Token(END, 'end', line 13)
Completed if statement.
Completed parse.
```

Repeat 语句：

```
    private RepeatStatement parseRepeatStatement() {
        System.out.println("Parsing repeat statement...");
        Token repeatToken = consume(TokenType.REPEAT, errorMessage: "Expect 'repeat'");
        List<Statement> statements = new ArrayList<>();
        while (!check(TokenType.UNTIL) && !isAtEnd()) {
            statements.add(parseStatement());
        }
        consume(TokenType.UNTIL, errorMessage: "Expect 'until'");
        Expression condition = parseExpression();
        consume(TokenType.COLON, errorMessage: "Expect ';'"); // Ensure the ';' after the condition
        System.out.println("Completed repeat statement.");
        return new RepeatStatement(statements, condition, repeatToken.getLine());
    }
```

获取 token 和 repeat 语句所在行号，从当前位置进行循环直到结束。

```
Parsing statement starting with token:  Token(REPEAT, 'repeat', line 8)
Parsing repeat statement...
Consuming token: Token(REPEAT, 'repeat', line 8)
```

赋值语句：

获取赋值的 ID，进行表达式的分析：是吧分析左右操作数并打印输出

```
private AssignStatement parseAssignmentStatement() {
    System.out.println("Parsing assignment statement...");
    Token idToken = consume(TokenType.ID, errorMessage: "Expect variable name");
    consume(TokenType.ASSIGN, errorMessage: "Expect ':='");
    Expression expression = parseExpression();
    consume(TokenType.COLON, errorMessage: "Expect ';'");
    System.out.println("Completed assignment statement.");
    return new AssignStatement(idToken.getText(), expression, idToken.getLine());
}
```

分析结果：

```
Parsing statement starting with token:  Token(ID, 'fact', line 9)
Parsing assignment statement...
Consuming token: Token(ID, 'fact', line 9)
Consuming token: Token(ASSIGN, ':=', line 9)
Parsing expression...
Parsing simple expression...
Parsing term...
Parsing factor...
Variable: fact
Parsing factor...
Variable: x
Completed term.
Completed simple expression.
Completed expression.
Consuming token: Token(COLON, ';', line 9)
Completed assignment statement.
```

（三）机器代码生成

初始化以及根据类型判断进行每一行对应机器代码的生成：

```
private void generateProgram(Program program) {
    System.out.println("开始生成程序代码");
    emitRM( op: "LD", mp,  d: 0,  s: 0,  comment: "load max address from location 0");
    emitRM( op: "ST", ac,  d: 0,  s: 0,  comment: "clear location 0");

    for (Statement stmt : program.getStatements()) {
        generateStatement(stmt);
    }
    emitRO( op: "HALT",  r: 0,  s: 0,  t: 0,  comment: "halt");
    System.out.println("程序代码生成完成");
}
```

```
private void generateStatement(Statement stmt) {
    System.out.println("生成语句: " + stmt.getClass().getSimpleName());
    if (stmt instanceof IfStatement) {
        generateIfStatement((IfStatement) stmt);
    } else if (stmt instanceof RepeatStatement) {
        generateRepeatStatement((RepeatStatement) stmt);
    } else if (stmt instanceof AssignStatement) {
        generateAssignment((AssignStatement) stmt);
    } else if (stmt instanceof ReadStatement) {
        generateReadStatement((ReadStatement) stmt);
    } else if (stmt instanceof WriteStatement) {
        generateWriteStatement((WriteStatement) stmt);
    }
}
```

（1）READ 语句

```
private void generateReadStatement(ReadStatement stmt) {
    System.out.println("开始生成读取语句: " + stmt.getVariable());
    emitRO( op: "IN", ac, s: 0, t: 0, comment: "read integer value");
    int memLoc = getMemoryLocation(stmt.getVariable());
    emitRM( op: "ST", ac, memLoc, gp, comment: "read: store value");
    System.out.println("读取完成，输入值存储于地址: " + memLoc);
}
```

生成结果：

```
开始生成程序代码
  0:     LD  6,0(0)  load max address from location 0
  1:     ST  0,0(0)  clear location 0
生成语句: ReadStatement
开始生成读取语句: x
  2:     IN  0,0,0  read integer value
  3:     ST  0,0(5)  read: store value
读取完成，输入值存储于地址: 0
```

（2）IF 语句：判断表达式的真假进行地址跳转。表达式通过保存左右操作数并进行操作符的判断选择相应指令。表达式为假进行地址跳转到 ELSE 语句并进行修正。

```
private void generateIfStatement(IfStatement stmt) {
    System.out.println("开始生成 If 语句");
    generateExpression(stmt.getCondition());
    int jumpToElse = emitSkip( howMany: 1); // 占位符，需要后续修补
    System.out.println("If 条件为假时跳转到 Else: 地址 " + jumpToElse);

    for (Statement thenStmt : stmt.getThenStatements()) {
        generateStatement(thenStmt);
    }
    int skipElse = emitSkip( howMany: 1); // 占位符，需要后续修补
    System.out.println("跳过 Else 的跳转地址: " + skipElse);

    if (!stmt.getElseStatements().isEmpty()) {
        backpatch(jumpToElse, op: "JEQ", value: currentLine() - jumpToElse); // 修正到 Else 开始的地址
        for (Statement elseStmt : stmt.getElseStatements()) {
            generateStatement(elseStmt);
        }
    } else {
        backpatch(jumpToElse, op: "LDA", value: currentLine() - jumpToElse); // 如果没有 Else，直接跳到 If 语句结束
    }

    backpatch(skipElse, op: "LDA", value: currentLine() - skipElse); // 修正到 If 语句结束的地址
    System.out.println("If 语句结束于: " + currentLine());
}
```

```java
private void generateExpression(Expression expr) {
    if (expr instanceof BinaryExpression) {
        BinaryExpression binExpr = (BinaryExpression) expr;
        generateExpression(binExpr.getLeft());
        emitRM( op: "ST", ac, tmpOffset--, mp, comment: "save left operand");
        generateExpression(binExpr.getRight());
        emitRM( op: "LD", ac1, ++tmpOffset, mp, comment: "load left operand");
        switch (binExpr.getOperator()) {
            case PLUS:
                emitRO( op: "ADD", ac, ac1, ac, comment: "op +");
                break;
            case MINUS:
                emitRO( op: "SUB", ac, ac1, ac, comment: "op -");
                break;
            case MULTI:
                emitRO( op: "MUL", ac, ac1, ac, comment: "op *");
                break;
            case DIV:
                emitRO( op: "DIV", ac, ac1, ac, comment: "op /");
                break;
            case LESS:
                emitRO( op: "SUB", ac, ac1, ac, comment: "op <");
                emitRM( op: "JLT", ac, d: 2, pc, comment: "jump if less");
                emitRM( op: "LDC", ac, d: 0, s: 0, comment: "false case");
                emitRM( op: "LDA", pc, d: 1, pc, comment: "skip true case");
                emitRM( op: "LDC", ac, d: 1, s: 0, comment: "true case");
                break;
            case EQ:
                emitRO( op: "SUB", ac, ac1, ac, comment: "op ==");
                emitRM( op: "JEQ", ac, d: 2, pc, comment: "jump if equal");
                emitRM( op: "LDC", ac, d: 0, s: 0, comment: "false case");
                emitRM( op: "LDA", pc, d: 1, pc, comment: "skip true case");
                emitRM( op: "LDC", ac, d: 1, s: 0, comment: "true case");
                break;
        }
```

生成结果：

```
生成语句: IfStatement
开始生成 If 语句
   4:     LDC  0,0(0)  load constant
   5:      ST  0,0(6)  save left operand
   6:      LD  0,0(5)  load variable
   7:      LD  1,0(6)  load left operand
   8:     SUB  0,1,0   op <
   9:     JLT  0,2(7)  jump if less
  10:     LDC  0,0(0)  false case
  11:     LDA  7,1(7)  skip true case
  12:     LDC  0,1(0)  true case
If 条件为假时跳转到 Else: 地址 13
```

（3）赋值语句

```java
private void generateAssignment(AssignStatement stmt) {
    System.out.println("开始生成赋值语句: " + stmt.getVariable());
    generateExpression(stmt.getExpression());
    int memLoc = getMemoryLocation(stmt.getVariable());
    emitRM( op: "ST", ac, memLoc, gp, comment: "assign: store value");
    System.out.println("赋值完成，变量 " + stmt.getVariable() + " 存储于地址: " + memLoc);
}
```

生成结果：

```
开始生成赋值语句: fact
 14:     LDC  0,1(0)  load constant
 15:      ST  0,1(5)  assign: store value
赋值完成，变量 fact 存储于地址: 1
```

（4）repeat 语句：保存循环开始位置并从此位置进行循环生成语句，结束位置进行跳转。

```java
private void generateRepeatStatement(RepeatStatement stmt) {
    System.out.println("开始生成 Repeat 语句");
    int loopStart = currentLine();
    System.out.println("循环开始于: " + loopStart);
    for (Statement s : stmt.getStatements()) {
        generateStatement(s);
    }
    generateExpression(stmt.getCondition());
    emitRM( op: "JEQ", ac, d: loopStart - currentLine() - 1, pc, comment: "repeat until x = 0");
    System.out.println("Repeat 语句生成完成，跳回到: " + loopStart);
}
```

生成结果：

```
开始生成 Repeat 语句
循环开始于: 16
生成语句: AssignStatement
开始生成赋值语句: fact
 16:     LD  0,1(5)  load variable
 17:     ST  0,0(6)  save left operand
 18:     LD  0,0(5)  load variable
 19:     LD  1,0(6)  load left operand
 20:    MUL  0,1,0  op *
 21:     ST  0,1(5)  assign: store value
赋值完成，变量 fact 存储于地址: 1
```

```
生成语句: AssignStatement
开始生成赋值语句: x
 22:     LD  0,0(5)  load variable
 23:     ST  0,0(6)  save left operand
 24:    LDC  0,1(0)  load constant
 25:     LD  1,0(6)  load left operand
 26:    SUB  0,1,0  op -
 27:     ST  0,0(5)  assign: store value
赋值完成，变量 x 存储于地址: 0
 28:     LD  0,0(5)  load variable
 29:     ST  0,0(6)  save left operand
 30:    LDC  0,0(0)  load constant
 31:     LD  1,0(6)  load left operand
 32:    SUB  0,1,0  op ==
 33:    JEQ  0,2(7)  jump if equal
 34:    LDC  0,0(0)  false case
 35:    LDA  7,1(7)  skip true case
 36:    LDC  0,1(0)  true case
 37:    JEQ  0,-22(7)  repeat until x = 0
Repeat 语句生成完成，跳回到: 16
```

（5）WRITE 语句

```
private void generateWriteStatement(WriteStatement stmt) {
    System.out.println("开始生成写入语句");
    generateExpression(stmt.getExpression());
    emitRO( op: "OUT", ac,  s: 0,  t: 0,  comment: "write integer value");
    System.out.println("写入完成");
}
```

生成结果：

```
生成语句: WriteStatement
开始生成写入语句
 38:     LD  0,1(5)  load variable
 39:    OUT  0,0,0  write integer value
写入完成
```

程序结束：

```
If 语句结束于: 41
 41:    HALT  0,0,0  halt
程序代码生成完成
开始保存输出到文件: output.tm
输出保存完成
编译完成，输出保存至：output.tm
TM 源代码生成成功!
```

（四）运行测试

```
C:\Windows\System
Microsoft Windows [版本 10.0.22631.3672]
(c) Microsoft Corporation。保留所有权利。

E:\code\JAVA\BY\BY-4>tm output.tm
TM  simulation (enter h for help)...
Enter command: g
Enter value for IN instruction: 5
OUT instruction prints: 120
HALT: 0,0,0
Halted
Enter command:
```

Output.tm 文件：

```
 0:    LD  6,0(0)
 1:    ST  0,0(0)
 2:    IN  0,0,0
 3:    ST  0,0(5)
 4:   LDC  0,0(0)
 5:    ST  0,0(6)
 6:    LD  0,0(5)
 7:    LD  1,0(6)
 8:   SUB  0,1,0
 9:   JLT  0,2(7)
10:   LDC  0,0(0)
11:   LDA  7,1(7)
12:   LDC  0,1(0)
14:   LDC  0,1(0)
15:    ST  0,1(5)
16:    LD  0,1(5)
17:    ST  0,0(6)
18:    LD  0,0(5)
19:    LD  1,0(6)
20:   MUL  0,1,0
21:    ST  0,1(5)
22:    LD  0,0(5)
23:    ST  0,0(6)
24:   LDC  0,1(0)
25:    LD  1,0(6)
26:   SUB  0,1,0
27:    ST  0,0(5)
28:    LD  0,0(5)
29:    ST  0,0(6)
30:   LDC  0,0(0)
31:    LD  1,0(6)
32:   SUB  0,1,0
33:   JEQ  0,2(7)
34:   LDC  0,0(0)
35:   LDA  7,1(7)
36:   LDC  0,1(0)
37:   JEQ  0,-22(7)
38:    LD  0,1(5)
39:   OUT  0,0,0
13:   JEQ  0,27(7)
40:   LDA  7,0(7)
41:  HALT  0,0,0
```

思考题：

阅读 *TM.exe* 程序的源程序 *TM.c*，阅读时要知道 *TM.c* 是一个虚拟机，也可说是一个解释执行器，也可以说是一个编译器，为什么？。通过这个例子，就知道虚拟机，解释执行器，编译器，这三者其实同义。

答：

TM.c 这个程序源代码的确可以让你理解虚拟机、解释执行器和编译器之间的关系

1. 虚拟机 (Virtual Machine)：在计算机科学中，虚拟机是一种软件实现，它模拟了硬件的功能，使得可以在其上运行软件。在这个例子中，TM.c 实现了一个虚拟机，它模拟了一台虚拟的计算机，可以执行一系列指令。

2. 解释执行器 (Interpreter)：解释执行器是一种程序，它能够直接执行源代码，逐行解释并执行代码。TM.c 可以被看作是一个解释执行器，因为它直接读取源代码，并解释执行其中的指令，而不需要将源代码转换成另一种形式。

3. 编译器 (Compiler)：编译器是将源代码转换成目标代码的程序，目标代码通常是机器语言或者字节码。尽管 TM.c 没有将源代码转换成另一种形式，但是它可以被看作是一个简化的编译器，因为它将源代码转换成了虚拟机可以执行的指令序列。

综上所述，TM.c 实际上具有虚拟机、解释执行器和编译器的特征，因为它可以解释执行源代码，并且将源代码转换成虚拟机可以执行的指令序列。因此，通过这个例子，我们可以看到虚拟机、解释执行器和编译器在某种程度上是同义的，它们都涉及将源代码转换成可执行的形式。

收获与体会：

通过实现 TINY 语言编译器，我深刻理解了编译原理中的词法分析、语法分析和代码生成等概念。了解到 TM 虚拟机如何进行代码生成以及 TM 机器指令和计算机系统的指令有很多相似，都是通过汇编语言生成代码，其次这个实验让我领悟到编译器是如何将高级语言转换为机器代码的，提升了我对编程语言底层运行机制的理解。同时对虚拟机以及编译器和解释执行器都有很多的理解。最后，通过实践，我锻炼了自己的编程能力和解决问题的能力，收获颇丰。

实验成绩	