



**软件工程专业综合拓展（论文）**

**少儿编程平台**

**学 院 计算机学院**

**专 业 软件工程**

**年级班别 2016级（3）班**

**学 号 3116004824**

**学生姓名 林楷羽**

**指导教师 欧毓毅**

**2019年11月**

**目 录**

[1. 系统设计 1](#_Toc25073722)

[1.1 界面设计 1](#_Toc25073723)

[1.2 词法解析器设计 3](#_Toc25073724)

[1.2.1 词法解析器功能 3](#_Toc25073725)

[1.2.2 词法解析器的遍历 4](#_Toc25073726)

[1.2.3 词法解析器逻辑设计 5](#_Toc25073727)

[1.2.4 词法解析器核心：nextToken() 7](#_Toc25073728)

[1.3 语法解析器设计 8](#_Toc25073729)

[1.3.1 语法解析器功能 8](#_Toc25073730)

[1.3.2 语法解析树 AST 9](#_Toc25073731)

[1.3.3 语法解析器的遍历与Token匹配 11](#_Toc25073732)

[1.3.4 语法解析器的逻辑设计 13](#_Toc25073733)

[1.4 程序执行器设计 15](#_Toc25073734)

[1.4.1 程序执行器功能 15](#_Toc25073735)

[1.4.2 程序执行器逻辑设计 16](#_Toc25073736)

[1.5 本章小结 17](#_Toc25073737)

# 

# 1. 系统设计

## 1.1 界面设计

本系统的用户界面为编程平台界面。

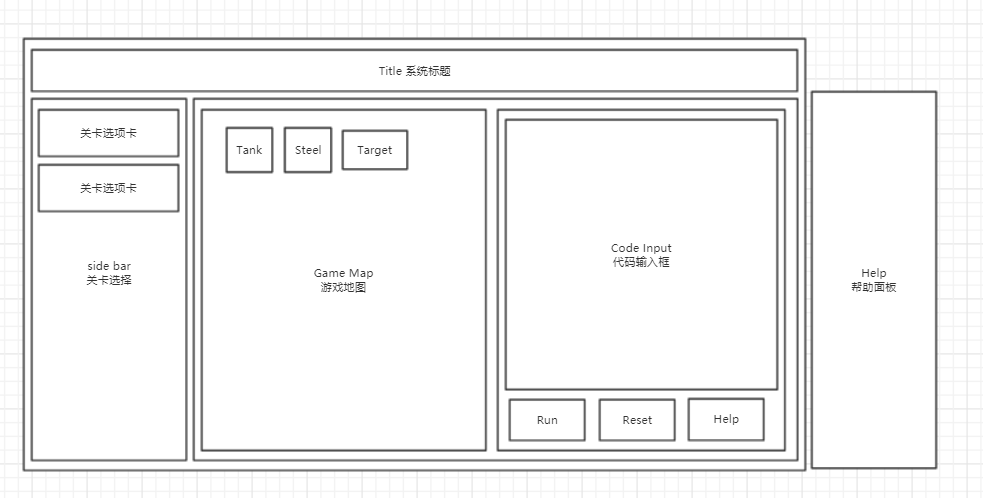


图1.1 编程平台界面设计系统原型图

本系统的用户界面按功能可主要分为：顶部的标题区、左侧的关卡选择区、中间的游戏地图区、右侧的代码交互区。

1. 顶部标题区：显示系统名称。
2. 左侧关卡选择区： 用户可以通过点击关卡选项卡，进行关卡选择。
3. 中部游戏UI区：主要分为四种元素
4. Tank用户通过代码操作的坦克；
5. Steel 路障；
6. Load 坦克可以行驶的路；
7. Target 坦克需要到达的目的地。
8. 右侧代码交互区
9. 代码输入框：用户可以在该区域编辑代码，进行对坦克的控制。
10. 编译按钮：用户输入代码后，点击该按钮进行代码编译、执行，同时，会有代码执行相关提示。
11. 重置按钮：用户执行完代码可点击该按钮使坦克复位。
12. 帮助按钮：用户可以通过点击该按钮，展开API文档帮助界面。

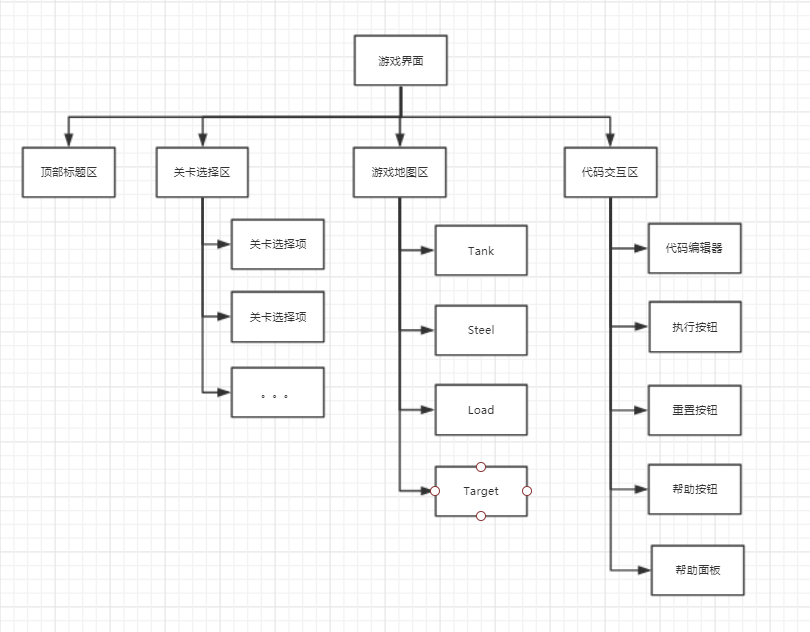


图1.2 编程平台界面设计系统结构图

## 1.2 词法解析器设计

### 1.2.1 词法解析器功能

本次选在需求分析中明确了词法分析这一步需要支持的token类型，如下：

1. 常量： integer整形 string字符串 true/false 布尔类型；

2. 标识符：即变量名，本项目中定义为以连续字母组成，因为需要区分大小写的，所以字母可以是从“A”到“Z”的大写字母和从“a”到“z”的小写字母。；

3. 界符：分号; 逗号, 左右小括号() 左右大括号{}；

4. 操作符：减号- 加号+ 乘号\* 除号/ 赋值= 全等比较== 不等比较！= 大于> 小于<；

5. 关键字、保留字：let if else null return fn while。

可以看到，在词法分析阶段，我们的 Tokens 不需要关心关键词是什么，只要识别是不是关键词即可，因为关键词的辨认会留到语法分析时处理。涉及到语意处理就要考虑上下文，而这都不是词法分析阶段要考虑的。

为此，需要设计一个Lexer词法分析类，该Lexer词法分析类将一段代码段作为输入，并按以上Token分类，将代码分割成一个个合法的Token，输出给Parser语法解析器。

如：let num = 1;

将被解析成：

1. Let(关键字)；
2. num(变量名) ；
3. =(操作符) ；
4. 1(整形) ；
5. ;(界符)。

### 1.2.2 词法解析器的遍历

像大多数语言一样，上述编程语言为了方便人类阅读，采用从左到右的书写方式，因此分词方向也从左到右。自然地，Lexer需要一个字符串遍历器，对输入的代码段进行逐个字符逐个字符的遍历，并按要求进行分割。

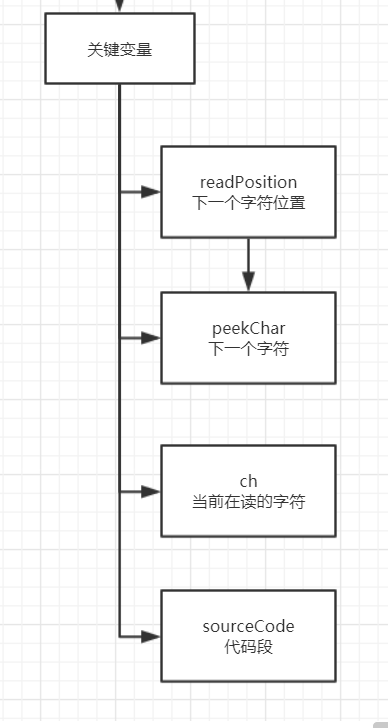


图1.3 Lexer关键变量设计

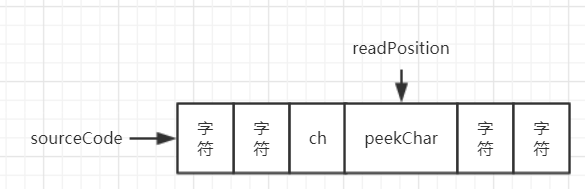


图1.4 readPosition示意图

遍历器的遍历操作：读取当前位置的字符，然后向后移动一位，直至末尾。

private readChar(): string {

if (this.readPosition >= this.sourceCode.length) {

this.ch = EOF;

} else {

this.ch = this.sourceCode[this.readPosition];

}

this.readPosition++;

return this.ch;

}

### 1.2.3 词法解析器逻辑设计

Lexer词法编译器的设计如下：

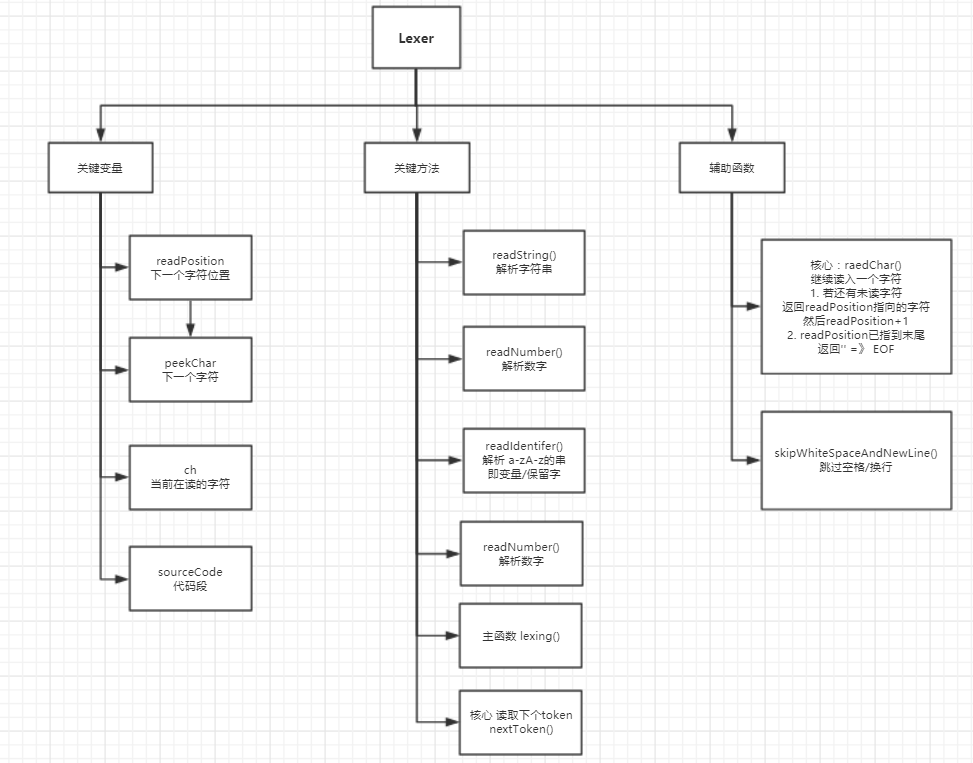


图1.5 Lexer 词法解析器的概要设计

主函数lexing()循环调用核心函数nextToken()，即为逐个读取字符后进行相应的Token切割。

为了更优雅地实现此功能，还需要封装相应Token的解析函数：

1. readString()用来解析字符串；
2. readNumber()用来解析数字；
3. readIdentifer()用来解析a-zA-z的字符串 即变量或保留字。

以上函数需要依赖于1.2.2的遍历函数readChar()。

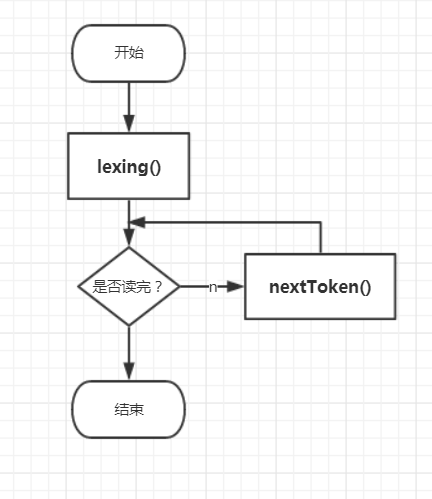


图1.6 词法解析流程图

### 1.2.4 词法解析器核心：nextToken()

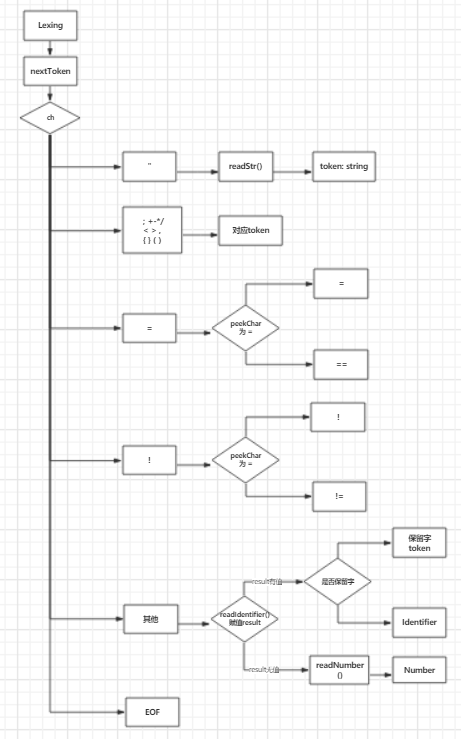


图1.7 nextToken()示意图

NextToken()会将所读取的字符进行判断，如：

switch (this.ch) {

case '=':

if (this.peekChar === '=') {

this.readChar(); // 读出 ==

} else {

// 读出 =

}

break;

}

## 1.3 语法解析器设计

### 1.3.1 语法解析器功能

在需求分析中明确了语法解析器需要支持的内容解析功能，如下：

**语句的解析**：

1. 声明语句的语法解析

2. 赋值语句的语法解析

3. 表达式语句的语法解析

4. 返回值语句的语法解析

5. 语句块的语法解析

6. 条件判断语句的语法解析

7. 循环语句的语法解析

**表达式的解析：**

1. 变量表达式的语法解析

2. 整形表达式的语法解析

3. 字符串表达式的语法解析

4. 布尔值表达式的语法解析

5. 函数表达式的语法解析

6. 前缀表达式的语法解析

7. 中缀表达式的语法解析

8. 函数调用表达式的语法解析

对于表达式的解析，还需要注意**运算符的优先级**

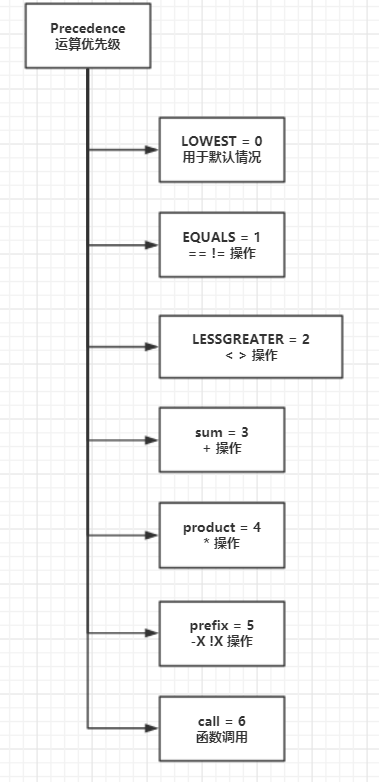


图 1.8运算符优先级表

### 1.3.2 语法解析树 AST

以解析顺序为角度，语法分析分为两种，自顶而下与自底而上。

本项目使用自顶向下的思路实现Parser语法解析器类。

Parser语法解析器类将Lexer词法解析器的输出：Token串作为输入，读取Token串并构建AST（语法解析树）。

AST的节点简单分为两大类：

1. 语句节点（Statement）、
2. 表达式节点（Expression）。

AST可以形象的表示程序执行的结构形式，比如

let a = 1;

let b = 2;

let add = fn(x, y) {

return x + y;

};

add(a, b);

表示为AST形式为：

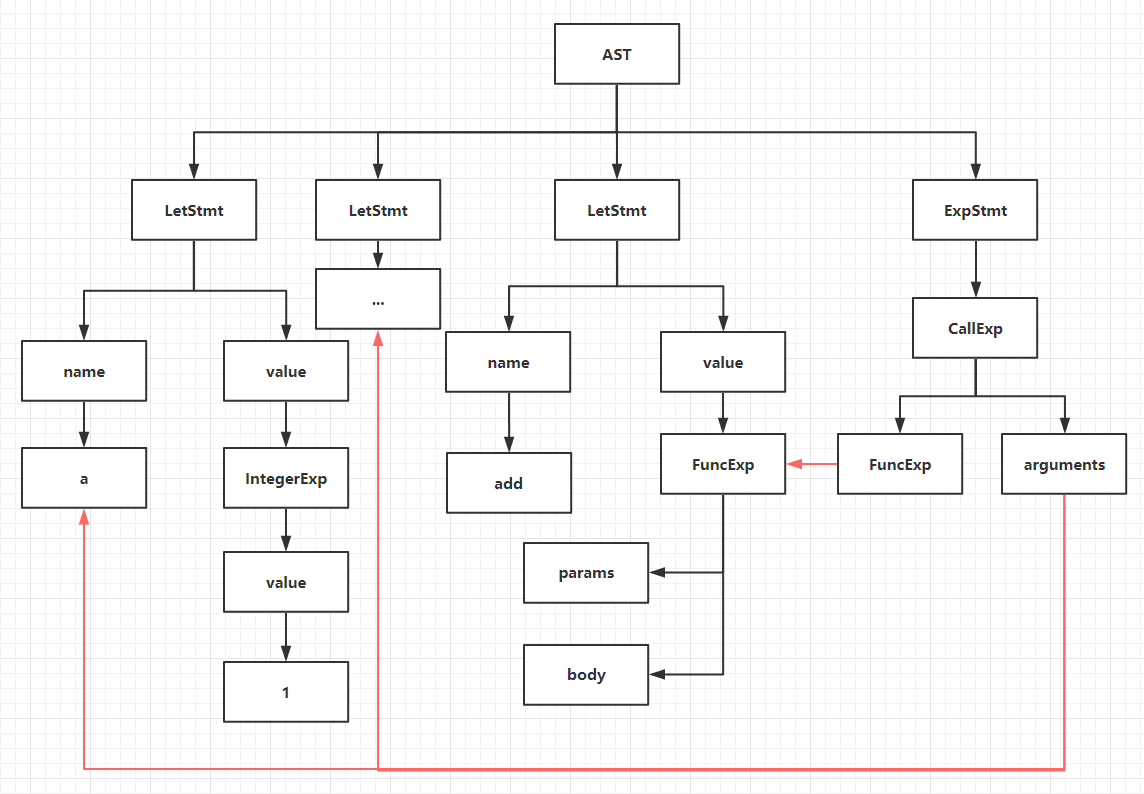


图 1.9AST示意图

1. letStatement语句：变量名为 a 值为 1

2. letStatement语句：变量名为 b 值为 2

3. letStatement语句：变量名为 add 值为函数表达式

4. ExpressionStatement语句：具体为函数调用表达式(CallExpression)，调用函数add 参数为a = 1, b= 2

### 1.3.3 语法解析器的遍历与Token匹配

与词法解析器Lexer类似，语法解析器Parser需要对Token串进行相应分割，因此需要一个对于Token串的遍历操作。所以需要定义一个curToken指向正在进行读取的Token。

同时，需要一个peekPos指针，指向下一个准备读取的Token位置。

如：

private get curToken(): Token {

if (!this.tokens.length) {

return new Token(TokenType.EOF, "", 0, "");

}

return this.tokens[this.peekPos - 1];

}

private nextToken() {

this.peekPos++;

}

因为在进行语法解析的过程中，不可避免的，需要向前看一位进行逻辑判断，所以，同时还需要peekToken指向下一个准备读取的Token。

private get peekToken(): Token {

if (this.tokens.length < 2) {

return new Token(TokenType.EOF, "", 0, "");

}

return this.tokens[this.peekPos];

}

在进行相关Token匹配时，也需要封装函数进行Token匹配。

private curTokenIs(tokenType: TokenType) {

return this.curToken.type() === tokenType;

}

// 匹配后 不会调用nextToken

private peekTokenIs(tokenType: TokenType) {

return this.peekToken.type() === tokenType;

}

// 匹配后 会自动调用nextToken

private expectPeek(tokenType: TokenType) {

if (this.peekTokenIs(tokenType)) {

this.nextToken();

return true;

} else {

return false;

}

}

### 1.3.4 语法解析器的逻辑设计

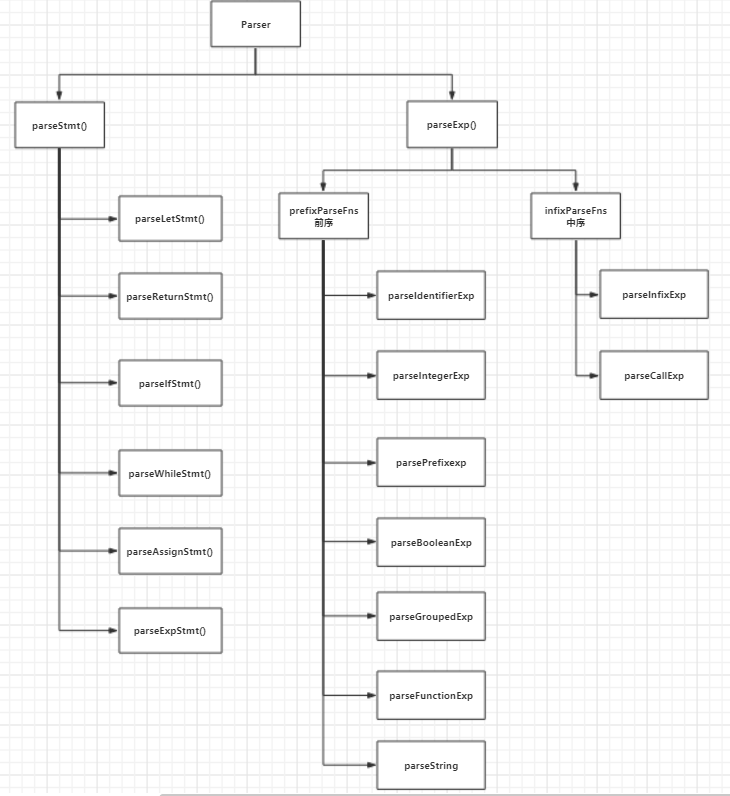


图 1.10语法解析器概要设计

基于经过1.2.1的介绍，词法解析将会输出如下Token串：

1. Let(关键字)
2. num(变量名)
3. =(操作符)
4. 1(整形)
5. ;(界符)

而语法解析，就是将这些Token串按照语法规则，组合成AST。

如：以上Token串将组合成一个letStmt节点：

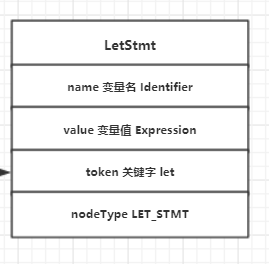


图 1.11 letStmt示意图

1. name变量名-> num

2. value变量值-> 1

语句的解析较为简单，先由统一的入口parseStmt()函数进行逻辑分发：

private parseStatement() {

switch 当前token类型 {

case token为let:

return 解析LetStmt语句;

case token为return

return 解析return语句;

case token为if:

return 解析if语句;

//...其余语句解析

default:

return 解析表达式语句

}

}

当前token类型为let时，进入到letStmt语句解析函数parseLetStmt()中:

private parseLetStatement() {

if (尝试匹配下一个token 若为变量类型token则遍历器移位) {

不为变量类型token 发生错误

}

变量名 = token信息

if (尝试匹配下一个token 若为赋值Token则遍历器移位) {

不为赋值token 发生错误

}

变量值 = 解析表达式

if (尝试匹配下一个token 若为分号token则遍历器移位) {

不为分号token 发生错误

}

根据变量名、变量值，组成letStmt节点返回

}

## 1.4 程序执行器设计

### 1.4.1 程序执行器功能

程序执行器是本“微型编译器”的最终阶段，他将接收语法解析后的AST抽象语法树，遍历AST节点并执行相应的代码逻辑，在此阶段，还需要执行内置函数，用于对游戏中坦克的控制。

在语句的执行过程中，程序执行器将支持产生如下结果：

1. 基础类型节点：整型节点、布尔值节点、字符串节点、null（空）节点。

2. 返回值类型节点。

3. 函数表达式节点。

同时，为了支持游戏编程模式，需要支持内置函数的调用：

1. Move(step): 控制坦克向前移动。

2. turnLeft()、turnRight()：控制坦克向左、向右转向。

3. canMove(): 用于判断坦克是否可以向前移动。

### 1.4.2 程序执行器逻辑设计

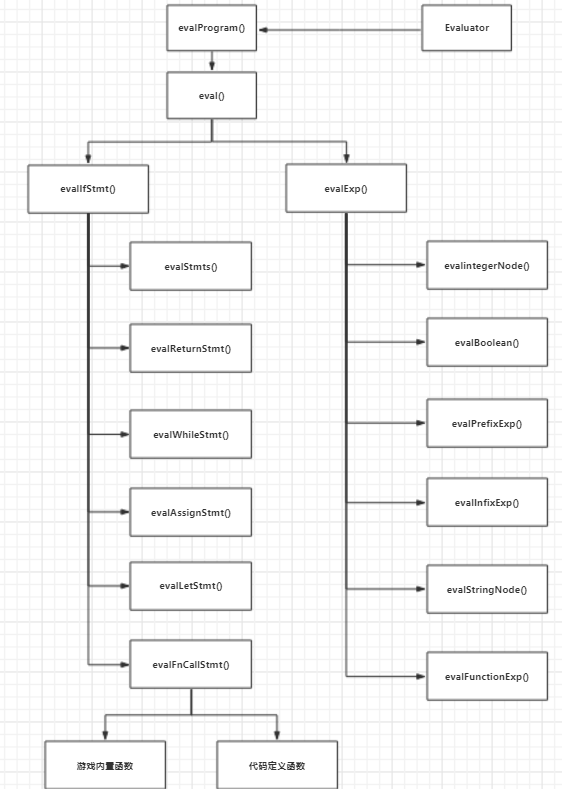


图 1.12程序执行器概要设计

## 1.5 本章小结

本章分为四大部分，界面设计中介绍该系统的界面元素，用户交互功能设计；在词法解析器设计中介绍了词法解析器的功能及实现思路并给出部分实现代码；在语法解析器设计中介绍了语法解析器的功能并通过具体例子描述了语法解析树(AST)的结构；在程序执行器设计中介绍了程序执行器的功能以及简单的设计思路。