引言

开发此编译器项目的初衷源于我在学校使用 “johnny” 软件来模拟汇编程序时的体验。在使用 johnny 学习时，我意识到它并没有帮助我将高级编程知识融会贯通 —— 特别是我从未真正理解像 Java 这样的语言是如何被转换成机器码的。我设想能有一个工具，把课堂上讲授的 Java 程序翻译成 johnny 的汇编代码，让学生在学习高级编程的同时，也能理解底层机器操作的原理。
基于这个想法，我开始研究编译器的基本原理，并实现了一个可以将 Java 代码编译成 johnny 可识别格式的编译器。由于 Java 本身的复杂度以及 johnny 汇编的局限性，本编译器更注重展示编译原理，而非支持完整的面向对象项目。

1 编译器功能

该编译器由 Java 实现，主要用于编译简易的 Java 程序。功能概述如下：

1.1 总体概览

1.1.1 支持的功能

- **简单变量声明：**
  支持声明 int 类型的变量。
- **声明时初始化：**
  允许在声明变量时直接进行赋值。
- **赋值语句：**
  支持常规的赋值操作。
- **条件语句：**
  支持 if 语句。
- **循环结构：**
  支持 while 循环。
- **函数定义与调用：**
  通过扩展指令集实现对函数的支持。
- **表达式求值：**
  能正确处理操作符优先级。
- **输出：**
  通过扩展指令集支持输出操作。

1.1.2 不支持的功能

- 数字不能超过 999。
- 不支持负数和小数。
- 变量名必须唯一（在 main 和函数作用域之间也不能重复）。
- 除加法与减法外，不支持其他二元运算（如乘法和除法），但可通过函数模拟。

- 不支持类和对象。
- 不支持 string、char、float、double、boolean 等数据类型。
- 不支持在同一行中进行多变量声明。
- 不支持在函数内调用其他函数（包括递归）。
- 不支持复杂数据结构、多文件项目以及高级语法检查。

1.2 开发路线概览

	Lexer	Preprocessor	Parser	Generator	Assembly To Johnny
变量声明	✅	✅	✅	✅	✅
变量计算	✅	✅	✅	✅	✅
运算优先级	✅	✅	✅	✅	✅
作用域（Domain）	✅	✅	✅	✅	✅
输入（Input）	✅	✅	❌	❌	✅
输出（Output）	✅	✅	✅	✅	✅
IF 语句	✅	✅	✅	✅	✅
While 循环	✅	✅	✅	✅	✅
For 循环	✅	✅	✅	✅	✅
函数（Function）	✅	✅	✅	✅	✅
Boolean（布尔类型）	❌	❌	❌	❌	❌
变量名称作用域限制	✅	❌	❌	❌	❌

1.3 扩展指令集

在原有指令的基础上，新增了两个指令：

- **savi addr:**
  对原有 `save` 指令的扩展。将某个寄存器的值保存到由 `addr` 指向的内存地址中。
- **jmpi addr:**
  对原有 `jmp` 指令的扩展。跳转到由 `addr` 指向的内存地址。

1.4 编译日志

编译器在编译过程中会生成详细的日志，主要用于调试。这些日志包括：

- 各阶段的编译信息（如处理的行数、定义的变量、内存分配详情等）。

- 错误信息：在编译失败时会给出失败原因（不含具体行号）。

1.5 SimpleJava 子集设计

1.5.1 整体结构

- **每个程序只有一个类：**
  程序必须写在一个类里，并且包含一个固定签名的 `main` 方法作为入口：

```
public static void main() {
    // ...
}
```

- **静态方法：**
  除了 `main` 之外，还可以定义其他静态方法：
  - 必须返回 int。
  - 只能接收 int 类型参数。
  - 不支持方法重载。

```
public static int multiply(int a, int b) {
    int result = 0;
    while (b > 0) {
        result = result + a;
        b = b - 1;
    }
    return result;
}
```

- **类成员：**
  虽然允许在类中定义成员，但该语言不支持面向对象的特性，如对象、成员变量、继承或多态等。
- **异常处理：**
  不支持 try/catch/finally。
- **递归：**
  不支持递归函数调用。

1.5.2 数据类型

- **仅支持整型：**
  唯一支持的数据类型为 `int` 。
- **布尔值：**
  用 int 来表示布尔值：0 表示 false，非 0 表示 true。
- **不支持其他类型：**
  不支持 `String` 、数组、浮点数、对象或其他数据类型。

1.5.3 表达式与运算符

- **算术运算符：**
  仅支持 `+` 和 `-`。
- **比较运算符：**
  支持 `==`、`!=`、`<`、`>`、`<=`、`>=`。在条件表达式中使用时，其结果会作为 int 存储（0 或 1）。
- **括号：**
  可以使用括号来控制运算优先级。
- **函数调用：**
  在表达式中可以调用静态方法。所有参数必须为 int 类型，返回值也必须是 int 类型。

1.5.4 语句

语言支持以下几种语句类型：

- **变量声明：**
  只能声明 int 变量，可选地进行初始化：

```
int a;
int b = 10;
```

- **赋值语句：**
  例如：

```
a = 5;
a = a + 3;
```

- **条件语句（开发中）：**
  支持 if-else：

```
if (a < 10) {
    a = a + 1;
} else {
    a = a - 1;
}
```

- **循环语句（开发中）：**
  支持 while 循环：

```
while (a != 0) {
    a = a - 1;
}
```

- **函数调用（开发中）：**
  可以调用静态方法：

```
foo(3, a);
```

- **return 语句（开发中）：**
  在静态方法（除了 main 外）中使用 return 返回一个 int 值：

```
    return a + b;
```

1.5.5 示例

```
class SimpleProgram {
  public int mul(int a, int b) {
    int result = 0;
    while (b > 0) {
      result += a;
      b--;
    }
    return result;
  }
  public static void main(String[] args) {
    int x = 10;

    for(int i=0;i<x;i++){
      System.out.println(mul(i,2));
    }
  }
}
```

2 使用方法

本章节将结合图片和截图，介绍编译器的具体使用步骤：

- **代码生成：**
  分步说明如何编译 Java 程序。
- **图形界面：**
  介绍如何在编译器的图形界面中进行操作。
- **RAM 数据编辑器：**
  如何使用内置的 RAM 数据编辑器。
- **导入到 johnny：**
  说明如何将编译器生成的汇编代码导入到 johnny 软件中。

另外还提供了若干示例程序：

- 展示函数用法的示例。
- 演示更复杂算法的示例。
- 用于触发错误的示例，以说明不支持的功能。

3 编译器内部原理

编译器通过多阶段的转换过程，将源程序逐级分解为更低层次的形式，最终生成 johnny 能识别的机器码。

3.1 编译流程概览

源程序将经过一系列转换器，每个阶段都将程序化简至更基本的形式。

3.2 词法分析（Lexer）

Lexer 将源码拆解为一个个词法符号（如关键字、运算符、标识符等），并忽略空格与换行符等不必要元素。
简要说明： Lexer 把源代码分解成最基本的记号，为后续分析奠定基础。
（稍后会插入 Lexer 输出示例。）

3.3 预处理（Preprocessor）

预处理器对词法符号流进行初步清理。由于编译器不支持类结构，它会移除类的相关构造，并识别程序的入口点（main 函数）。

3.4 语法分析（Parser）

Parser 根据词法符号流构建抽象语法树（AST）。
AST 说明： 抽象语法树是程序结构的层次化表示，能清晰反映代码元素之间的语法关系，便于进一步分析和生成代码。对于函数，编译器会进行第二次解析来定位并处理函数定义 —— 这一特殊步骤可确保所有与函数相关的结构被正确识别。Parser 采用递归方式实现，因此能够自然地处理嵌套结构。
（稍后将插入 AST 节点的继承关系图以及示例输出。）

3.5 代码生成（Generator）

生成器（Generator）会遍历 AST 并输出汇编代码。主要工作点包括：

- **标签（Label）的使用：**
  为了便于阅读和调试，生成器使用标签代替直接的内存地址，最终再将这些标签解析为具体地址。
- **递归生成：**
  每个 AST 节点都实现了一个 `generate()` 方法，用来递归地产生代码。
- **常量管理：**
  在语法分析时遇到的常量会记录下来，最终存放在特定的内存区域。
- **函数生成：**
  - 函数定义的汇编代码会生成在 main 函数之后。
  - 调用函数时，先将当前行号保存在某个固定寄存器，再跳转到对应的函数。
  - 函数通过将返回值写到指定寄存器，然后使用 `jmpi` 跳转回去的方式返回。

    - 函数参数存储在专门的内存区域里。
- **临时变量：**
  表达式求值过程中需要的临时变量会从特定区域申请地址，计算结束后，如果这些中间结果不再使用，对应地址可被重复利用。

3.6 AsmToJohnny 转换器

在生成汇编代码后，AsmToJohnny 转换器会将其翻译成 johnny 识别的机器码：

- **标签解析：**
  转换器会多次扫描汇编代码，确保所有标签被正确解析。
- **特殊标签处理：**
    - 如果某行只有一个标签而没有其他指令，则需要将此标签移动到下一行的行尾，以保证行号对齐。
    - 对于像 `THIS_LINE_PLUS_THREE` （表示当前行号加三）或指向固定寄存器的标签，需要做专门的逻辑处理。

4 内存管理

编译器将内存划分为若干区域，每个区域对应不同用途，布局如下：

- **0–600：程序区域**
  存放编译生成的程序代码。
- **600–800：常量区域**
  存放编译过程发现、运行时需用的数字常量。
- **801–850：变量区域**
  用于存储源程序中定义的变量。
- **851–870：临时变量区域**
  在表达式求值时，用于保存中间结果。
- **871–890：函数参数区域**
  专门用来保存函数调用时的参数。
- **891–899：固定指针/寄存器区域**
  为函数调用和固定操作中需用的指针及寄存器预留的区域。
  本区域中定义的重要地址包括：
    - `OUTPUT_PointerAddress = 895`
    - `PARAMETER_PointerAddress = 896`
    - `RETURN_PointerAddress = 897`
    - `FUNCTION_Result_Address = 898`
    - `ERROR_DefaultAddress = 899`
- **900–949：输入区域**
  用于存储输入数据。
- **950–999：输出区域**
  用于存储输出数据。

5 技术挑战

标签处理

在生成的汇编代码中处理标签时，需要注意：

- **多次扫描：**
  为了正确解析所有标签，转换器要多次遍历汇编代码。
- **空行标签：**
  如果某一行只有标签而没有其他指令，该标签会被移动到下一行的行尾，以保持行号正确。
- **特殊标签：**
  像 `THIS_LINE_PLUS_THREE` （表示当前行号加三）或表示固定寄存器的标签，需要专门的处理逻辑。

函数调用的限制

在有限的内存环境中实现完整的函数调用栈相当复杂。因此，该编译器不支持嵌套或递归的函数调用：

- **简化方案：**
  仅使用一个简单的、基于固定寄存器的跳转机制，而非完整的调用栈。

6 项目统计

- 代码总行数：3543
- 源代码行所占比例：72%
- 注释行所占比例：11%
- 空行所占比例：17%