编译器设计与实现 实验报告

07111306
1120131962
赵文天

一、 实验内容与实验环境

1. 实验内容

实验要求在 BITMiniCC 框架的基础上，自己实现预处理、词法分析、语法分析、中间代码生成模块。本实验中，在预处理阶段，实现了去除注释以及多余的空行；在词法分析阶段，根据预先定义的 DFA 将预处理后的源程序转换为属性字流；在语法分析阶段，对属性字流进行处理，使用预先定义的 LL(1)文法生成了语法树；在中间代码生成阶段，根据语法分析生成的语法树以及用到的产生式生成四元式序列。

2. 实验环境

操作系统：Windows 10 专业版

Java 运行环境：Java(TM) SE Runtime Environment (build 1.8.0_91-b14)

Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)

集成开发环境：Eclipse Mars.2 Release 4.5.2

二、 预处理

在预处理阶段，实现了一个简单的自动机，去掉源程序中的注释以及多余的空行。预处理阶段没有对任何预编译指令进行处理，包括#define，#ifdef，#include 等。

三、 词法分析

在词法分析阶段，构造了一个描述 C 语言中单词的有限确定自动机，根据该自动机对单词进行识别。

该自动机可以处理的单词类型如下：

- 整形常量：

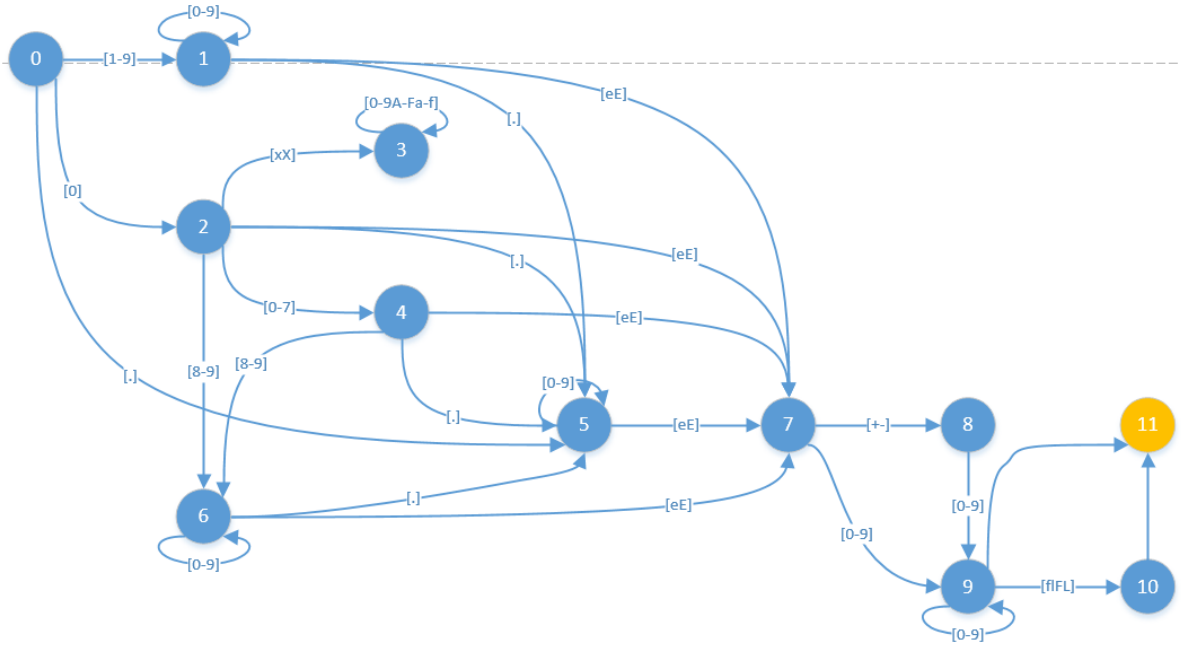
类型	示例
十进制整形	1234
八进制整形	0127
十六进制整形	0x3F3e, 0X3f3E
无符号整形	1234u, 1234U
长整形	1234l, 1234L

- 浮点型常量：

类型	示例
双精度浮点型	3.1415, 01289
单精度浮点型	3.14f
指数表示	0.314e1, 31.4e-1

整形以及浮点型常量对应的 DFA：

浮点形&整形：

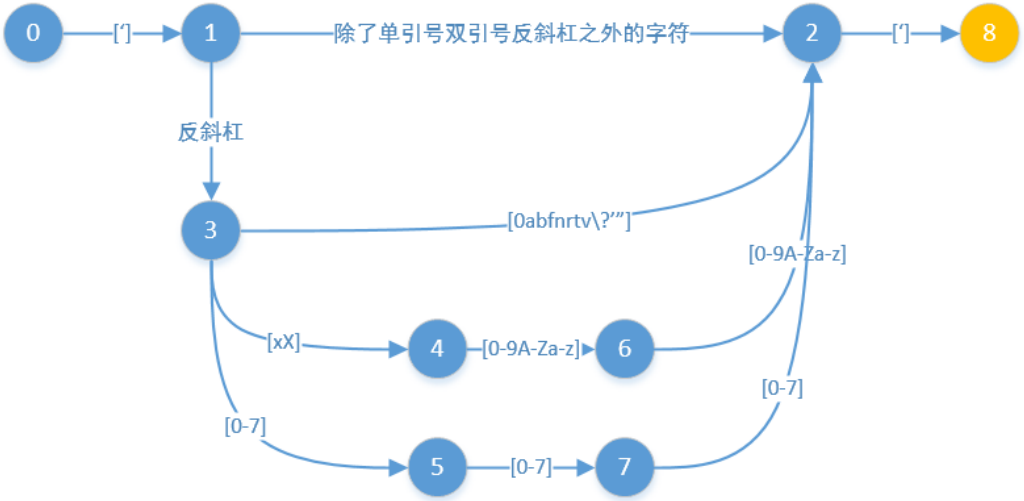


- 字符型常量：

类型	示例
普通字符	’a’
转义字符	’\n’,’\t’,’\0’
十六进制表示的字符	’\x41’（与’A’等价）
八进制表示的字符	’\101’（与’A’等价）

处理字符常量的自动机：

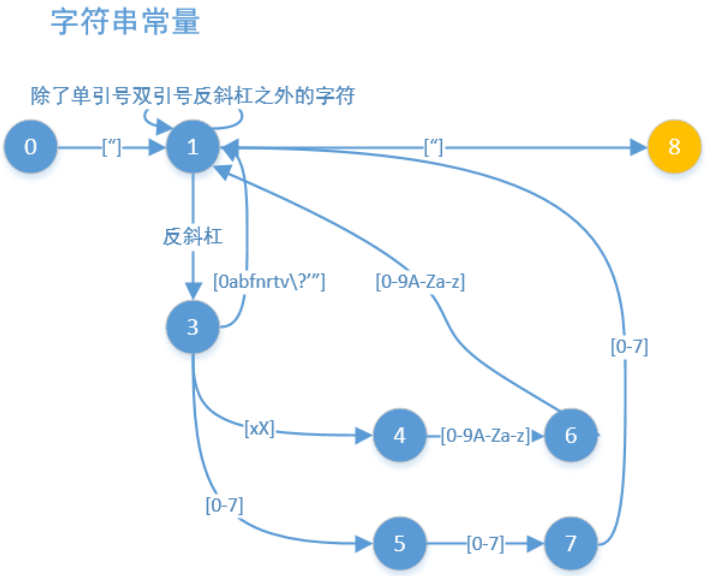
字符常量



- 字符串常量：

类型	示例
字符串常量	”\x41s soon \101s possible.\n”

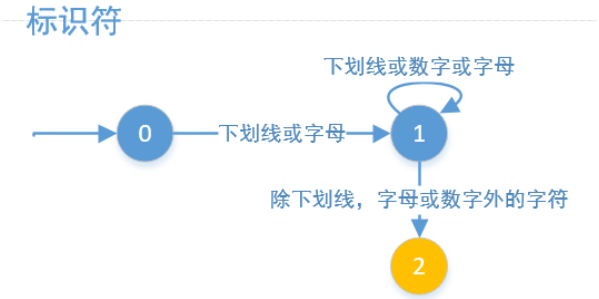
识别字符串常量的自动机：



· 标识符与关键字：

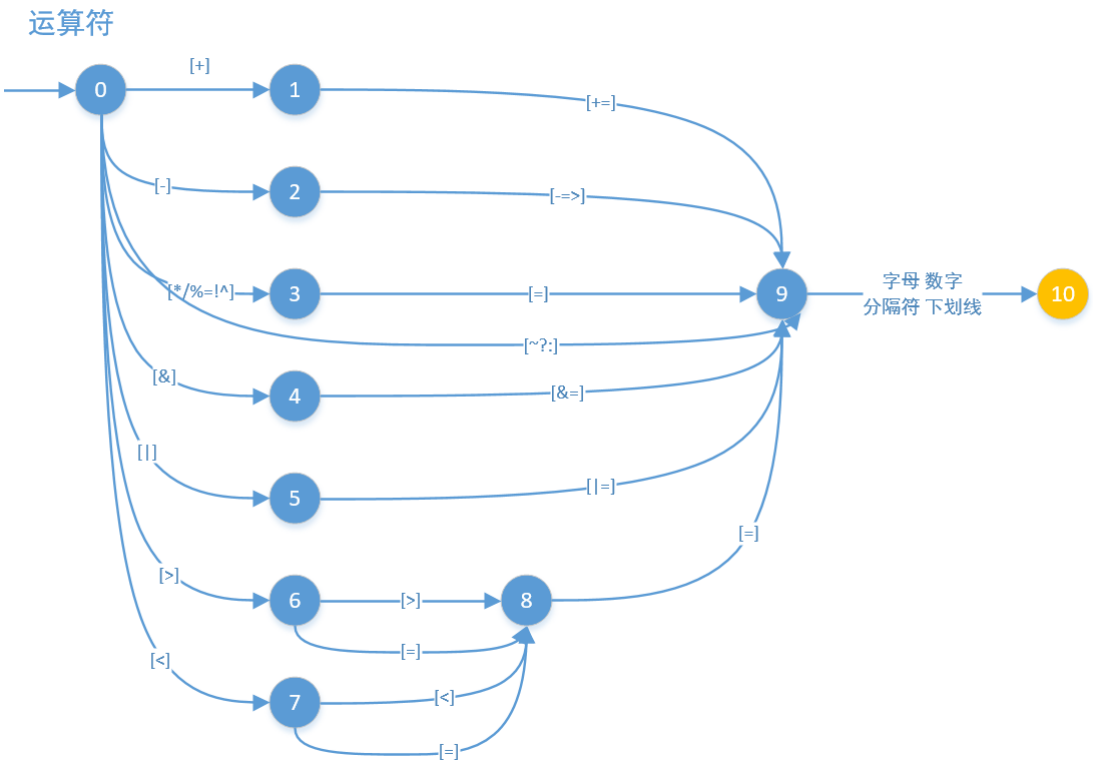
在词法分析阶段，将标识符与关键字同等看待，因为在 C 语言中关键字（保留字）也属于标识符。在输出属性字流时，根据标识符的内容判断标识符是否为关键字。

识别标识符的自动机：



· 运算符：

识别运算符的自动机可以处理除 sizeof 外的所有运算符。



词法分析用到的类在包 me.entalent.minicc.scanner 下。

其中，Scanner 类为词法分析器，实现了 IMiniCCScanner 接口。其中定义的 Token 类表示一个属性字。

词法分析的部分输出：

```
<token>
    <number>1</number>
    <value>int</value>
    <type>keyword</type>
    <line>1</line>
    <valid>true</valid>
</token>
<token>
    <number>2</number>
    <value>main</value>
    <type>identifier</type>
    <line>1</line>
    <valid>true</valid>
</token>
<token>
    <number>3</number>
    <value>(</value>
    <type>separator</type>
    <line>1</line>
    <valid>true</valid>
</token>
```

四、 语法分析

本实验中，采用 LL(1)方法进行语法分析。

1. 用到的文法：

由于实验要求中没有明确规定要使用的文法，这里将所用的文法子集进行说明。

支持的关键字：

int, float, if, while, break, continue, return

支持的运算符：

=（赋值）

== != < > <= >=（相等，不等，小于，大于，小于等于，大于等于）

+ -（加法，减法）

/ *（除法，乘法）

支持的语句：

定义语句，如 int a; float b;（定义时不支持赋值）

赋值语句，如 a = 2;

if 语句，如 if(a > b) { /*其他语句*/ }

while 语句，如 while(a < b) { /*其他语句*/ }

return 语句，如 return a; return a + b;

运算语句，如 a = b * c + d * e; a = (b > d);

2. 产生式定义及分析表生成：

由于文法中的产生式较多，并且都需要在程序中定义，采用把产生式直接在 Java 代码中初始化不方便调试和扩展。这里以 JSON 格式定义整个文法，将文法的定义储存在文件中。对于文法

$$E = TE',$$
$$E' = +TE'|\varepsilon$$
$$T = FT'$$
$$T' = *FT'|\varepsilon$$
$$F = (E)|\varepsilon$$

其在 JSON 文件中以如下形式进行定义：

```
{
"terminator":
{
"i":"i",
"BL":"(",
"BR":")",
"ADD":"+",
"MUL":"*"
},
"non-terminator":
["E","E1","T","T1","F"],
"begin-symbol":"E",
"grammar-production":
{
"E":"T.E1",
"E1":"ADD.T.E1|$",
"T":"F.T1",
"T1":"MUL.F.T1|$",
"F":"BL.E.BR|i"
}
}
```

其中，terminator 字段定义所有终结符，以及终结符在属性字流中出现的形式。non-terminator 字段的值为一个数组，数组中为所有的非终结符。begin-symbol 字段

的值为字符串，表示文法的开始符号。grammar-production 字段定义了所有产生式，其值为一个 JSON 对象。该对象的每个字段名为产生式左部的非终结符，字段值为产生式右部。产生式右部由字符 '|' 分隔。由于每个终结符或非终结符可能由一个或多个字符表示，一个产生式的符号之间由字符 '.' 分隔。空串由字符 '$' 表示。

通过改变该文件的内容，可以改变词法分析过程中所用的文法，方便地进行扩展与重用。解析 JSON 字符串时使用了 Google 的 gson 库（版本为 2.3.1），其 jar 文件已加入工程的/libs 目录下并加入工程的 Build Path。定义文法的文件名为 syntax.json，该文件需在工程的根目录下或 jar 程序的工作目录下。

在进行 LL(1)分析之前，首先要读取定义了文法的文件中的内容，对其进行解析后求 FIRST 与 FOLLOW 集合，并构造 LL(1)分析表。这里假设文件中定义的文法不会出现直接或间接左递归，并且是 LL(1)文法。在处理文法时，不会消除文法中的左递归或提取左公因式。

Parser 类中的 getAllFirstCollection 方法实现了求 FIRST 集合。该方法中用到的的算法如下：

求一个非终结符的 FIRST 集时，针对产生式$A \to a_1 a_2 \cdots a_n$，有如下算法：

若a_i为非终结符，则先求a_i的 FIRST 集。先将$FIRST(a_i)$中的非ε字符加入$FIRST(A)$。若有$\varepsilon \in FIRST(a_i)$，则令 i=i+1，重复此过程，否则算法结束。若 i=n+1，则有$\varepsilon \in FIRST(A)$。

若a_i为终结符，将a_i加入 $FIRST(A)$，算法结束。

Parser 类中的 getAllFollowCollection 算法实现了求 FOLLOW 集合。该方法根据《编译原理》中的算法 4.3 实现。当没有任何新的符号加入任何非终结符的 FOLLOW 集时，算法结束。

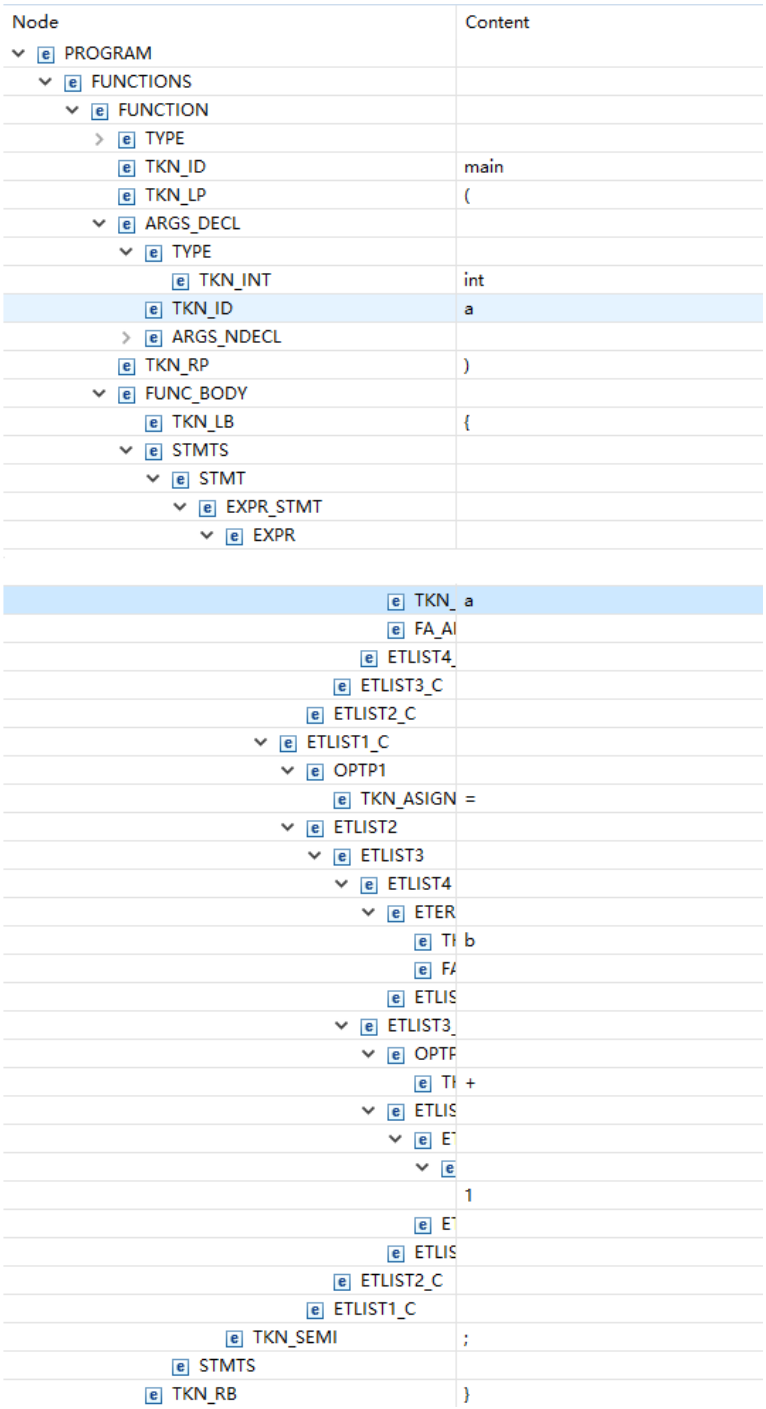
Parser 类中的 buildLL1Table 方法构造 LL(1)分析表。

3. 语法分析的结果：

对于下面的源程序

```
int main(int a, int b){
    a = b + 1;
}
```

可以得到下面的语法树：



五、 语义分析与中间代码生成

在中间代码生成阶段，根据语法分析生成的语法树以及分析过程中所用的产生式生成中间代码。中间代码的形式为四元式。

中间代码生成所用的类位于 me.entalent.minicc.icgen 包中。InternalCodeGen 类进行中间代码生成。

四元式可以用下面的四元组表示：

$$(op, arg1, arg2, res)$$

其中，op 表示四元式进行的操作，即运算符。

arg1 和 arg2 分别为两个操作数，若为单目运算，则 arg2 留空。

res 保存运算结果，或跳转指令要跳转到的语句标号。在程序中，四元式用

Quadruple 类表示。该类中保存了四元式的四个成员，以及四元式的标号。该类的定义如下：

```
public class Quadruple {
    //四元式标号
    public int index;
    //运算符，操作数1，操作数2，运算结果
    public String op, arg1, arg2, res;
}
```

在语义分析以及中间代码生成的过程中，规定了如下的综合属性：

bid 和 eid

若某个节点或其子节点在处理的过程中生成了四元式，则 bid 表示这组四元式的第一个的标号，eid 表示最后一个四元式的标号。四元式的标号均从 1 开始。若某个节点没有对应的四元式，则这两个值均为 0。该属性向上传递的过程中，父节点需要对子节点所表示的标号的区间进行合并。

result

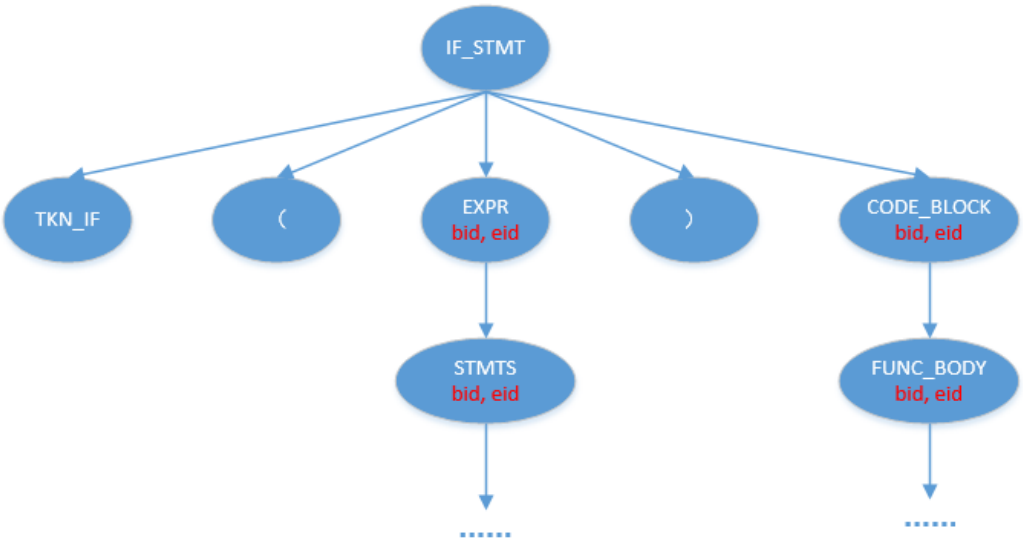
当一个节点在处理过程中生成了四元式（如非终结符 EXPR 对应的节点），result 属性表示四元式的运算结果。若生成的四元式为(+, a, b, T1)，则父节点的 result 值为 T1。

以 if 语句的语义分析为例，说明语义分析的过程：

在文法规则中，if 语句对应如下产生式：

$$IF\_STMT \rightarrow TKN\_IF(EXPR)CODE\_BLOCK$$

对应的语法树：



首先，对 EXPR 节点进行处理，生成条件判断语句对应的四元式，并获取 EXPR 节点的 result 属性 T1。先生成跳转语句的四元式$(j_F, T_1, _, _)$，设其标号为 index。之后生成 CODE_BLOCK 节点，即 if 语句对应的代码块的四元式，获取节点的 bid 与 eid 属性。显然，eid+1 为之前标号为 index 的四元式要跳转到的节点，将其改为$(j_F, _, _, eid+1)$。对 while 语句的处理与对 if 语句的处理类似。

输入下面的源程序：

```
int main(int a, int b){
```

```
        if(a > 0) {
            b = b + 1;
        }
        while(b < 0) {
            a = a * 2;
        }
    }
```

得到的四元式序列为

```
1 (>, a, 0, T1)
2 (jf, T1, _, 5)
3 (+, b, 1, T2)
4 (=, b, T2, T3)
5 (<, b, 0, T4)
6 (jf, T4, _, 10)
7 (*, a, 2, T5)
8 (=, a, T5, T6)
9 (j, _, _, 5)
```

其中 jf 表示条件不成立跳转，j 表示无条件跳转。

六、 附加说明

1. 对编译器框架的一些改进

在阅读 BITMiniCC 框架的源代码时，发现 MiniCCompiler 中对 Java 代码进行调用的过程中直接调用了 MiniCCPreProcessor，MiniCCScanner 这些框架自带的类，在调用自己的类时还需要更改这里的 Java 代码。这意味着此处的代码无法隐藏或进行混淆处理，并且允许被调用的其他类不实现相应的接口，调用方式不够规范。为避免这些不足，对 MiniCCompiler 类进行了修改，修改后的类为 me.entalent.minicc.MiniCCompiler2。

修改后的 MiniCCompiler2 中，需要将被调用的 Java 类的名称写入配置文件 config.xml，并以反射的方式对自己实现的其他类进行调用。修改后，config.xml 的格式：

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
  <phases>
    <phase>
        <!--
        若某个阶段调用Java代码，path的值应为该阶段要用到的类的完整类名
        若某个阶段调用可执行文件，path的值应为可执行文件的路径
         -->
      <phase skip="false" type="java" path="bit.minisys.minicc.pp.MiniCCPreProcessor" name="pp" />
      <phase skip="false" type="java" path="me.entalent.minicc.scanner.Scanner" name="scanning" />
      <phase skip="false" type="java" path="me.entalent.minicc.parser.Parser" name="parsing" />
      <phase skip="true" type="java" path="bit.minisys.minicc.semantic.MiniCCSemantic" name="semantic" />
      <phase skip="false" type="java" path="me.entalent.minicc.icgen.InternalCodeGen" name="icgen" />
      <phase skip="true" type="java" path="bit.minisys.minicc.optimizer.MiniCCOptimizer" name="optimizing" />
      <phase skip="true" type="java" path="bit.minisys.minicc.codegen.MiniCCCodeGen" name="codegen" />
      <phase skip="true" type="java" path="" name="simulating" />
    </phase>
  </phases>
</config>
```

其中，每个 phase 标签的 type 属性的值为"java"时，其 path 属性的值应为该阶段要调用的类的完整名称，包括包名和类名。type 属性为其他值时，相应的格式不变。

在 MiniCCompiler2 的 run 方法中，首先通过 ClassLoader 加载相应的类，之后调用 Class 的 newInstance 方法构造类的实例，这里要求被调用的类定义了被 public 修饰的无参构造方法，且该方法能正确构造出所需对象。之后调用方法签名为 void run(String input, String output)的方法，进行编译过程相应阶段的动作。调用过程

中分别对加载类、获取被调用的方法、方法调用等步骤可能抛出的异常进行了处理。修改后，无需再更改 MiniCCompiler2 的代码，只需要更改配置文件即可更改要调用的类。因此可以隐藏编译器框架的实现或对其进行混淆。

对其他语言，如 C++语言编译成的二进制文件进行调用的方法未做改变，仍为执行系统命令。

2. 解析/生成 XML 的工具类

在实验过程中，发现没有使用过 Java 的 JDOM API 或对 XML 语言不够熟悉的同学在解析或生成 XML 文件时遇到了一些困难。并且在编译的某个阶段的直接调用 JDOM 读取文件，对 XML 进行解析或生成，也使代码不够简洁。这里根据编译过程中可能用到的 XML 文件的格式，实现了一个可以对 XML 进行解析以及生成的工具类 me.entalent.minicc.util.xml.XmlNode，对 DOM 进行了封装，且调用方式足够简单。

该类的一个实例表示一个 XML 节点，该节点的属性，内容，以及指向所有子节点的引用。XmlNode 类使用 DOM 方式解析或生成 XML，提供从文件读入，输出到文件或字符串等方法。该类的一个使用示例如下：

```
//构造根节点
XmlNode root = new XmlNode("root");
//构造一个子节点
XmlNode quaternion = new XmlNode("quaternion")
                .attribute("op", "+")          //设置节点的属性
                .attribute("arg1", "a")
                .attribute("arg2", "1")
                .attribute("res", "T1")
                .textContent("a+1的四元式"); //设置节点的文本内容

root.addChild(quaternion);                     //添加子节点
System.out.println(root.toXmlString());        //输出到字符串
root.writeXmlToFile(new File("1.xml"));//输出到文件
```

该工具类的不足之处：由于使用 DOM 方式解析 XML，只能一次将整个文档的所有标签载入，内存消耗较大；设计 XmlNode 类时未考虑多线程场景，该类可能是线程不安全的；由于要保证输出的格式，输出 XML 文档的过程中使用了只有 SUN JDK 中提供的 XMLSerializer 类，因此无法用 OpenJDK 进行编译。