

# 编译原理实验教程

(草稿)

张昱 陈意云

中国科学技术大学 合肥

2008 年 4 月 9 日

# 目 录

(草稿) .....	I
<b>第 1 章 概述</b> .....	<b>1</b>
1.1 本书的目标 .....	1
1.2 课程设计结构 .....	2
1.2.1 要实现的源语言.....	2
1.2.2 抽象语法树.....	2
1.2.3 课程设计选题.....	3
1.2.4 小结.....	4
1.3 实验形式及评价 .....	4
1.4 实验环境与工具 .....	5
1.4.1 简介.....	5
1.4.2 环境变量设置.....	7
1.4.3 Eclipse的安装和使用.....	9
1.4.4 XML与Ant简介.....	19
<b>第 2 章 MINIJOOOL语言</b> .....	<b>23</b>
2.1 MINIJOOOL语言简介 .....	23
2.1.1 MiniJOOOL语言的特点.....	23
2.1.2 一个MiniJOOOL程序示例.....	24
2.2 MINIJOOOL语言的词法 .....	24
2.3 MINIJOOOL语言的语法 .....	26
2.3.1 类型、值和变量.....	27
2.3.2 类.....	32
2.3.3 语句块和语句.....	38
2.3.4 表达式.....	40
2.4 SIMPLEMINIJOOOL语言简介 .....	42
2.5 SKIPOOMINIJOOOL语言简介 .....	43
2.5.1 一些注意事项.....	44
2.5.2 一个SkipOOMiniJOOOL程序示例.....	46
2.6 SKIPOOMINIJOOOL语言的静态语义 .....	46
2.6.1 抽象语法.....	47
2.6.2 抽象类型、定型环境、断言与定型规则.....	48
2.6.3 良形的类型以及类型兼容.....	50

2.6.4 定型环境的扩展.....	51
2.6.5 表达式.....	51
2.6.6 语句和语句块.....	53
2.6.7 函数的定义.....	55
2.6.8 程序.....	56
<b>第3章 一个简单的程序解释器.....</b>	<b>58</b>
3.1 实验软件包的结构.....	58
3.2 课程设计 1：一个简单的程序解释器.....	59
3.3 实验平台介绍.....	61
3.3.1 实验平台接口.....	61
3.3.2 实验运行平台.....	65
3.4 课程设计 1 开发和测试指南.....	71
3.4.1 在Eclipse 下开发.....	72
3.4.2 在控制台下编译和运行.....	74
3.4.3 测试要求.....	75
3.5 抽象语法树（AST）.....	75
3.5.1 Eclipse AST的总体结构.....	75
3.5.2 org.eclipse.jdt.core.dom.AST.....	76
3.5.3 org.eclipse.jdt.core.dom.ASTNode及其派生类.....	77
3.5.4 org.eclipse.jdt.core.dom.ASTVisitor.....	77
3.5.5 Eclipse AST使用示例.....	77
3.5.6 AST的图形化显示包——ASTView.....	78
3.6 设计模式.....	81
3.6.1 工厂方法模式.....	81
3.6.2 访问者模式.....	82
<b>第4章 词法分析.....</b>	<b>85</b>
4.1 本章课程设计概述.....	85
4.2 课程设计 2-1：用JFlex为MiniJOL语言生成一个词法分析器.....	87
4.2.1 示例.....	87
4.2.2 用JFlex为MiniJOL进行词法分析.....	92
4.3 课程设计 2-2：手工编写一个简单的词法分析器.....	93
4.3.1 示例.....	93
4.3.2 课程设计任务.....	96
4.3.3 编译和运行指南.....	96
4.4 课程设计 2-3：编写一个NFA生成器.....	96
4.4.1 MLex 词法规范文件的格式.....	97

4.4.2 课程设计指导.....	99
4.4.3 课程设计任务.....	102
4.5 课程设计 2-4: 编写一个词法分析器的生成器 .....	103
4.6 JFlex词法规范 .....	104
4.6.1 用户代码.....	104
4.6.2 选项和声明.....	104
4.6.3 词法规则.....	107
<b>第 5 章 语法分析 .....</b>	<b>111</b>
5.1 本章课程设计概述 .....	111
5.2 课程设计 3-1: 手工编写一个语法分析器 .....	113
5.2.1 如何引用Lab2 项目中的类.....	113
5.2.2 要分析的语法结构.....	114
5.2.3 课程设计指导.....	115
5.2.4 课程设计要求.....	116
5.3 课程设计 3-2: 用CUP生成一个语法分析器.....	116
5.3.1 示例 1: 不带错误恢复的语法分析器.....	117
5.3.2 示例 2: 带错误恢复的语法分析器.....	124
5.3.3 课程设计任务.....	125
5.4 课程设计 3-3: 用JAVACC生成一个语法分析器 .....	126
5.4.1 示例 1: 不带错误恢复的语法分析器.....	126
5.4.2 示例 2: 带错误恢复的语法分析器.....	131
5.4.3 课程设计任务.....	133
5.5 课程设计 3-4: 编写一个语法分析器的生成器 .....	133
5.6 CUP与YACC.....	133
5.6.1 YACC简介.....	134
5.6.2 CUP与YACC的文法规范文件的结构.....	134
5.6.3 文法符号.....	135
5.6.4 一个简单的例子.....	136
5.6.5 错误恢复.....	138
5.7 JAVACC .....	138
5.7.1 JavaCC文法规范文件的结构.....	138
5.7.2 JavaCC选项.....	139
5.7.3 产生式.....	140
<b>第 6 章 语义分析 .....</b>	<b>144</b>
6.1 本章课程设计概述 .....	144
6.2 课程设计指导 .....	145

6.2.1 符号表的设计与组织.....	145
6.2.2 对AST的语义分析.....	145
6.2.3 在语法分析的同时进行语义分析.....	145
<b>第7章 MIPS汇编代码生成.....</b>	<b>146</b>
7.1 本章课程设计概述 .....	146
7.2 汇编代码的内部表示 .....	147
7.2.1 设计概述.....	147
7.2.2 示例.....	148
7.3 MIPS R2000/R3000 架构、汇编语言及SPIM.....	151
7.3.1 MIPS R2000/R3000 架构概述.....	151
7.3.2 汇编语言的语法.....	152
7.3.3 寄存器.....	153
7.3.4 汇编指令.....	155
7.3.5 过程调用.....	159
7.4 MIPS汇编语言与SKIP00MINIJOOL语言中部分结构的对应 .....	160
7.4.1 程序.....	160
7.4.2 全局变量的定义和引用.....	161
7.4.3 方法的定义.....	161
7.4.4 方法的调用.....	162
7.4.5 if语句.....	163
7.4.6 while 语句.....	163
7.4.7 return 语句.....	164
7.5 SPIM.....	164
7.5.1 XSpim 和PCSpim .....	164
7.6 SPIM提供的系统调用 .....	165
7.7 全局寄存器分配器 .....	166
7.7.1 与汇编代码生成器的关系.....	166
7.7.2 使用方法.....	166
7.8 课程设计 5-1 .....	167
7.8.1 课程设计指导.....	167
7.8.2 汇编代码生成器的编译和运行.....	168
7.9 课程设计 5-2 .....	168
7.9.1 课程设计指导.....	169
7.9.2 汇编代码生成器的编译和运行.....	169
<b>第8章 X86 汇编代码生成.....</b>	<b>170</b>
8.1 本章课程设计概述 .....	170

8.2 x86 架构、汇编语言及工具 .....	170
8.2.1 数据类型与指令 .....	170
8.2.2 寄存器 .....	171
8.2.3 操作数的格式 .....	171
8.2.4 程序栈 .....	172
8.2.5 一些常用的指令 .....	172
8.2.6 gcc 与 gdb .....	176
8.3 一些语句到x86 汇编代码的映射 .....	177
8.3.1 总体生成的汇编代码结构 .....	177
8.4 x86 寄存器分配器 .....	180
8.5 课程设计 6-1 .....	182
8.5.1 课程设计指导 .....	182
8.5.2 汇编代码生成器的编译和运行 .....	183
8.6 课程设计 6-2 .....	183
<b>第 9 章 综合课程设计 .....</b>	<b>185</b>
9.1 课程设计内容 .....	185
9.2 课程设计开发的目录结构 .....	185
9.3 课程设计提交的目录结构 .....	185
9.4 课程设计的时间节点 .....	186
9.5 课程设计的考评方法 .....	186
<b>参考文献 .....</b>	<b>188</b>
<b>附 录 .....</b>	<b>189</b>
附录 1 MINIJOOL 语言的词法记号ID .....	189
附录 2 算符的优先级与结合性 .....	189
附录 3 MINIJOOL 语言语法的EBNF表示 .....	191
附录 4 SIMPLEMINIJOOL 语言语法的EBNF表示 .....	193
附录 5 SKIPOOMINIJOOL 语言语法的EBNF表示 .....	194
附录 6 语法结构与AST节点的对应关系 .....	196
附录 7 MIPS 汇编语言的EBNF定义 .....	198
附录 8 x86 的AT&T汇编汇编语言的EBNF定义 .....	200

# 第1章 概述

## 1.1 本书的目标

编译原理是构造编译器的重要理论和技术基础。随着计算机技术和社会应用需求的发展,编译原理及技术也越来越多地运用在诸如编辑器、排版系统、数据处理等更广阔的领域,因此《编译原理》这门课程对于计算机及相关专业的本科生来说也越来越显得重要。

在实际的《编译原理》教学和学习中,大家普遍认为这门课程非常抽象而难学,剖析其中的主要原因是实践环节比较薄弱。一方面是缺少系统的编译原理实验教材,另一方面是学生很少实践或实践的深度不够。

十几年来,中国科学技术大学计算机专业学生的编译原理实验一直以阅读和扩展 PL/0 语言的编译器为基础。PL/0 语言过于简单,甚至没有函数参数,这就限制了以这个语言为基础的编译原理课程实践的深度和意义。另外,实验主要停留在阅读 PL/0 编译器已有的源代码层次上,学生实际动手很少。而要掌握编译原理的知识,实践是非常重要的。很显然,这种实验设置已经不能适应教学的需要和对不断发展的编译技术的学习理解。

为给计算机及相关专业的学生设计合适的编译原理课程实验,笔者调研了一些国外知名大学的编译原理实验设置。他们的编译实验已经非常成熟,并且与课程很好地搭配,覆盖了编译原理的主要知识点,这些经验值得我们借鉴。例如,加州大学伯克利分校<sup>1</sup>、加州理工学院<sup>2</sup>等学校的编译原理课程实验,尽管它们在实验语言的定义上有所差别,但是都要求在一个学期内实现一个简单的高级语言,也就是要完成一个功能完备的编译器;此外,加州大学伯克利分校还要求学生实现词法分析器和语法分析器的生成器,在难度上就更高一些。

这些编译实验的设计对当前的编译技术考虑得很周到,定义的语言一般具有面向对象的特性,符合现代高级语言的特征。从学生的角度来看,实验的难度较大,但实验的设计遵循了循序渐进的原则,在必要的时候给予学生足够的帮助。这使得学生能保持对这种技术的兴趣,并且学生成功完成一个实验所获得的成就感会激发他们挑战更高难度的实验任务。学生在完成一个学期的实验后,对编译原理与技术的理论知识可以理解得更加透彻;而实验中涉及到的对一些编程语言和工具环境的使用,更是为学生积累了宝贵的实践经验,学生的动手能力和科研能力都会有大幅度的提升。

在加州大学伯克利分校的编译实验设置基础上,笔者和曾经的一些学生(他们是吕博海、赵雷、周清博、王伟、张昊中、李勋浩)一起尝试做这些实验,然后设计适合国内学校学生的编译原理课程实验。目前,整个课程实验设计尚未全部完成,但是主体部分已成系统。这本书即是反映我们当前的主要成果,其中一部分的内容还很粗糙,需要进一步的完善。

笔者热忱欢迎大家对本书提出宝贵的意见,并将吸纳其中的精髓,来继续完善这本书!

<sup>1</sup> Ras Bodik. UC Berkeley CS164 Programming Languages and Compilers, Fall 2003.  
<http://www.cs.berkeley.edu/~bodik/cs164-fall-2003/>

<sup>2</sup> Jason Hickey. Caltech Compiler Design Laboratory. <http://www.cs.caltech.edu/courses/cs134/cs134b/>

## 1.2 课程设计结构

这一节将介绍综合的编译原理课程设计任务。我们将从要实现的源语言、使用的抽象语法树(Abstract Syntax Tree, AST)和课程设计的选题等来介绍。

### 1.2.1 要实现的源语言

这本书中引入了三种课程设计用的源语言，在第 2 章将详细描述这些语言的特点。

- **MiniJOOL 语言**：一种类似 Java 的小型面向对象语言。在 2.1~2.3 节给出了这种语言的描述，在附录 3 中给出了这种语言的文法描述。
- **SimpleMiniJOOL 语言**：它是 MiniJOOL 语言的一个简单子集，不具有面向对象特征。一个 SimpleMiniJOOL 程序只有一个名为 Program 的类，且类中只有一个静态的、名为 main 的函数。在这种语言中，只有整型类型，因此变量无须定义即可使用。在 2.4 节给出了这种语言的简要描述，在附录 4 中给出了这种语言的文法描述。
- **SkipOOMiniJOOL 语言**：它扩展了 SimpleMiniJOOL 语言，但是程序中只有一个名为 Program 的类，不过类中仅支持多个静态域和方法。它仍然是 MiniJOOL 语言的子集，包含了 MiniJOOL 语言的所有非面向对象特征。在这个语言中，有 int、boolean 和 String 类型以及一维数组类型。在 2.5 节中给出了这种语言的简要描述，2.6 节给出这种语言静态语义的形式描述，附录 5 给出了这种语言的文法描述。

为简便起见，我们将用上述三种语言编写的程序的扩展名都定义为 mj。在附录 1 中给出了三种语言涉及的终结符（记号）名及对应的串值。

在下面的课程设计任务的各个选题中，我们要求你应该实现 SimpleMiniJOOL 语言，然后再扩展来实现 SkipOOMiniJOOL 语言。对于面向对象部分，即完整的 MiniJOOL 语言，我们则不做要求。

### 1.2.2 抽象语法树

在本书的所有课程设计中，抽象语法树（Abstract Syntax Tree, AST）是其中关键的数据结构之一，它是编译器中常用的中间表示形式之一，能够清楚地反映源程序的语法结构。本书以这个结构为接口来分解一个完整编译器的各个任务，从而使你只需做一个完整编译器的部分工作，而这些工作同时又能和其他学生或者本书提供的工作装配到一起，形成一个完整的编译器。

为了简少编程的工作量，本书采用 Eclipse 的 JDT（Java Development Tools）提供的 AST 类层次结构。关于 Eclipse AST 结构和使用说明将在 3.5 节中详细介绍。

你需要在课程设计前期熟悉 Eclipse AST，你还要学会怎么编写 AST 的访问者类（参见 3.6.2 节）。你可以从第 3 章的课程设计入手来学习和了解它们。不论你选择 1.2.3 节中的哪一个选题，你都需要首先了解这个 AST，它是课程设计的基础。



### 1.2.3 课程设计选题

下面列出几个课程设计的选题，我们给出每个选题的输入和输出，以及这个选题的功能要求。其中，每个选题要实现的源语言在 1.2.1 节中已做规定，即你所做的部分必须能实现 SimpleMiniJOOL 语言，这是一个基本要求。你可以在此基础上进行扩展，直至实现 SkipOOMiniJOOL 语言。

#### 1.2.3.1 选题一：实现一个语言的解释器

输入：源语言程序对应的 AST

输出：输出对该 AST 解释执行的结果

功能要求：在解释执行中，要对不合法的 AST 节点进行异常检查和处理。

你需要用文字描述你的解释器中所处理的异常情况。你也需要构造一些测试用例来测试你的程序，你所构造的测试用例有两类：一类是自己手工编码构造的 AST；另一类是编写 mj 程序，再利用其他的分析器来生成所需要的 AST。你也需要用文字说明你的设计实现关键以及你的测试用例的设计意图。

#### 1.2.3.2 选题二：语法分析器的生成

输入：源语言程序

输出：该源语言程序对应的 AST

功能要求：你将使用分析器的生成工具 JFlex 和 CUP 来完成对源语言程序的词法分析和语法分析。你要对输入的不合法程序进行异常处理。

你需要用文字描述你所处理的异常情况。你也需要编写 mj 程序，再利用我们提供的 ASTViewer（见第 3 章）来图形化显示你所构造的 AST，你还需要和其他同学实现的解释器或代码生成器集成来形成一个完整的编译器。你需要用文字说明你的设计实现关键以及你的测试用例的设计意图。

#### 1.2.3.3 选题三：类型检查器

输入：源语言程序对应的 AST

输出：是否满足类型要求

功能要求：你需要对不合法的 AST 进行错误定位以及必要的错误恢复处理。

你需要用文字描述你所处理的异常情况，你也需要参照选题一来编写测试用例。你还需要和其他同学实现的分析器、解释器或代码生成器集成来形成一个完整的编译器。你需要用文字说明你的设计实现关键以及你的测试用例的设计意图。

#### 1.2.3.4 选题四：x86 汇编代码生成器

输入：源语言程序对应的中间表示（如 AST）

输出：对应的 x86 汇编代码文件

功能要求：输出的 x86 汇编代码应能用 gcc 汇编而得到可执行文件，运行可执行文件可以得到正确的执行结果。

你需要用文字描述你的设计实现关键，说明不同的语法结构与汇编代码的映射关系。你需要考虑寄存器分配、错误检查与处理等问题。你还需要和其他同学实现的分析器集成来形成一个完整的编译器。你需要用文字说明你的测试用例的设计意图。

#### **1.2.3.5 选题五：MIPS32 汇编代码生成器**

**输入：**源语言程序对应的中间表示（如 AST）

**输出：**对应的 MIPS32 汇编代码文件

**功能要求：**输出的 MIPS32 汇编代码应能在 SPIM 模拟器上运行并得到正确的执行结果。

你需要用文字描述你的设计实现关键，说明不同的语法结构与汇编代码的映射关系。你需要考虑寄存器分配、错误检查与处理等问题。你还需要和其他同学实现的分析器集成来形成一个完整的编译器。你需要用文字说明你的测试用例的设计意图。

### **1.2.4 小结**

本节所述的课程设计内容是一个综合的编译原理课程实验。你可以参考后面各章介绍的局部而循序渐进的课程设计及指导，从而帮助你打开你要完成的课程设计的思路，你可以超越这些指导，提出更好的设计实现方法。

## **1.3 实验形式及评价**

注意：PB0511 学生的编译实验要求参见第 9 章。

## 1.4 实验环境与工具

### 1.4.1 简介

为完成本书的各个课程设计，你将使用到以下的开发环境和工具：

#### 1、Java 集成开发环境

由于实验中需要用到Eclipse的JDT工具包，故我们推荐使用Eclipse集成开发环境(IDE, Integrated Development Environment)，你可以从 <http://www.eclipse.org/>获得。它是一个广泛使用的开源集成开发环境，在各种平台上都有发布的版本。你既可以用它在Windows操作系统下完成Java程序的开发，也可以用它在Linux等操作系统下完成。

#### 2、Java SDK

你可以从SUN的网站(<http://java.sun.com>)上获得Java软件开发包(SDK)。你可以安装J2SE 5.0(即JDK1.5)或Java SE 6.0(即JDK1.6)中的任一版本，并且将所安装的JRE(Java Runtime Environment)添加到Eclipse的系统库中，这样便于你跟踪调试Java源程序代码。

#### 3、Eclipse 的 Java 开发工具

表 1-1 Eclipse JDT AST 相关的 jar 文件

Eclipse 版本号	相关的 jar 文件
3.1.2	org.eclipse.core.resources_3.1.2.jar org.eclipse.core.runtime_3.1.2.jar org.eclipse.jdt.core_3.1.2.jar org.eclipse.jdt.ui_3.1.2.jar
3.2	org.eclipse.core.resources_3.2.2.jar org.eclipse.core.runtime_3.2.0.jar org.eclipse.equinox.common_3.2.0.jar org.eclipse.jdt.core_3.2.3.jar org.eclipse.jdt.ui_3.2.2.jar
3.3	org.eclipse.core.resources_3.3.0.jar org.eclipse.core.runtime_3.3.100.jar org.eclipse.equinox.common_3.3.0.jar org.eclipse.jdt.core_3.3.1.jar org.eclipse.jdt.ui_3.3.1.jar

我们使用Eclipse的Java开发工具（JDT, Java Development Tools）提供的抽象语法树类层次结构来表示我们的AST。当用Eclipse编译依赖JDT的程序时，需要把Eclipse安装目录中plugins子目录下的几个与AST实现有关的jar文件(表 1-1 列出了几个Eclipse版本中相关的jar

文件列表) 导入到Eclipse的classpath中; 当用JDK的javac和java编译、运行时, 需要在“-classpath”选项中将这些jar文件包含进来。

#### 4、JFlex

这是词法分析器的生成器(Generator), 它能根据jflex词法规范文件自动生成词法分析器的Java源代码。你可以从JFlex主页 <http://jflex.de/> (或 <http://sourceforge.net/projects/jflex/>) 下载JFlex的压缩包, 解压在某个目录下, 这里记为JFLEX\_HOME。切记按照JFlex根目录下的bin/jflex.bat中的注释修改它, 以适应机器的Java虚拟机等配置, 否则不能正常工作。在本书所附的软件包的tools/jflex目录下, 有 1.4.1 版本的jflex.jar。

#### 5、Ant

这是一个基于Java的编译工具, 它与Eclipse集成得很好。它的机制与GNU Make比较相像, 但是由于编译文件是基于XML(eXtensible Markup Language)数据格式来描述的, 因此比makefile更清晰易懂。更多的信息请参考Ant网站: <http://ant.apache.org>。1.4.4 节给出了XML和Ant的简要介绍。

#### 6、CUP

CUP(Constructor of Useful Parsers)是一个以LALR(1)文法为基础的语法分析器的生成器。其主要功能是读入一个文法规范(Grammar specification)文件, 然后生成可识别符合这种文法输入流的Java源代码。所读入的文法文件的扩展名约定为cup。你可以从 <http://www2.cs.tum.edu/projects/cup/> 下载CUP工具并获得更多的信息。在本书所附的软件包的tools/java-cup目录下, 有v11a版本的java-cup-v11a.jar以及java-cup-11a-runtime.jar文件。

#### 7、JavaCC

JavaCC(Java Compiler Compiler)是一种采用递归下降分析的、支持LL(k)文法的编译器的编译器。它不仅可以为输入的文法规范文件(文件扩展名为jj)生成对应分析器的Java源代码, 还提供其他与分析器生成有关的能力, 如树构造(通过包含在JavaCC中的JJTree工具来完成)、动作(actions)、调试(debugging)等。利用JavaCC中的JJDoc工具还可以将文法规范文件转换成文档(HTML格式)。你可以从 <https://javacc.dev.java.net/> 下载JavaCC工具并获得更多的信息。在我们提供的工具箱中的tools/javacc目录下, 有 4.0 版本的javacc.jar。

#### 8、GCC

在代码生成部分的实验中, 要求为抽象语法树生成x86 汇编代码。针对所生成的汇编代码, 本书推荐用GCC(GNU Compiler Collection)编译器将其编译连接成目标文件或可执行文件。GCC是开源的多语言、多目标、多平台的编译器集合, 是Linux系统的官方编译器。你可以从 <http://gcc.gnu.org> 下载。当前GCC已经发展到了 4.3.0 版(截至 2008 年 3 月 5 日), 它因为公开、易用受到众多开源爱好者的喜好。在Windows系统中, 无法直接使用GCC。你可以使用Windows下的Linux平台环境, 如CygWin, 或者使用GCC的Windows版本, 即MinGW(Minimalist GNU for Windows)。你可以从 <http://www.mingw.org/> 获得MinGW工具。

## 9、CygWin

CygWin是Windows下的Linux平台环境。有了CygWin，我们就可以在Windows下使用Linux下的BASH以及其它GNU工具。你可以从<http://www.cygwin.com/>或<http://www.cygwin.cn/>获得CygWin。

## 10、SPIM

在代码生成部分的实验中，还要求为抽象语法树生成MIPS32汇编代码。为了在普通的PC机上生成MIPS RISC架构代码并运行，需要一个能在x86架构上运行MIPS汇编程序的模拟器。SPIM就是这样的一个开源模拟器，你可以从<http://pages.cs.wisc.edu/~larus/spim.html>下载。它可以运行在Linux和MS Windows下。在Linux下，SPIM除提供一个字符界面外，还提供了Xwindow下的图形界面XSpim。在MS Windows下，SPIM提供了一个类似XSpim的图形界面PCSpim。

### 1.4.2 环境变量设置

在运行以上各种开发工具之前，首先介绍环境变量及其设置方法。环境变量是一个具有特定名字的对象，它包含一个或者多个应用程序所将用到的信息，用户可以修改相应的环境变量来对自己的运行环境进行定制。

例如，在MS Windows的命令提示控制台（也称dos控制台）下，输入echo %PATH%（在Linux或Unix的bash控制台下，可以输入echo \$PATH），你会看到系统列出不少用分号隔开的路径。PATH环境变量的作用是指定命令的搜索路径。也就是说，当输入一条控制台命令时，系统会主动在当前路径和PATH环境变量中所列出的各路径中寻找相应的命令来执行。比如，如果PATH中有C:\windows\system32，那么在任意路径下输入winmine命令，都可以打开C:\windows\system32下的扫雷游戏。

环境变量的引用方法在dos下和bash下略有不同。前者将环境变量名放在两个“%”之间来表示对环境变量的引用，比如之前的%PATH%；而后者则在环境变量名前加上“\$”来表示引用，如\$PATH。对于所出现的环境变量引用，系统将用环境变量的值来代替它们。比如，如果环境变量windir的值是"C:\WINDOWS"，那么%windir%\system32表示"C:\WINDOWS\system32"。

下面重点介绍环境变量的设置方法。

#### 1.4.2.1 设置用户或系统环境变量

##### 1、Windows平台下的设置

在Windows操作系统下，可以为单个用户配置环境变量，也可以为所有的用户统一配置环境变量，即系统环境变量。

这里以Windows XP为例来说明环境变量的设置方法。在Windows XP下，用鼠标单击“开始”→“控制面板”→“性能和维护”→“系统”，或者对“我的电脑”单击鼠标右键而后点击“属性”，将弹出“系统属性”窗口；选择窗口的“高级”标签中下方的“环境变量”，

可得如图 1-1 所示的窗口。



图 1-1 环境变量窗口

环境变量窗口上方是用户变量，其中设置的变量只能由当前用户名的用户使用；下方是系统变量，其中设置的变量可以被所有的计算机用户使用。设置系统变量需要 Administrator 权限。单击“新建”按钮，在弹出的对话框中输入变量名和变量值并按“确定”按钮之后，即可完成对环境变量的新增设置。你也可以选中一个现有的变量，单击“编辑”或“删除”按钮，来修改或删除该变量。

## 2、Linux 或 Unix 下的设置

bash 是大多数 Linux 或 Unix 版本的标准 Shell 程序，它具有非常强大的功能。这里给出为 bash 设置环境变量的方法。

在用户主目录下有一个 .bashrc 文件，每一次该用户运行 bash 时，bash 都会自动执行这个脚本。如果在该 .bashrc 文件中加入 `export envname="envval"` 后保存并退出，再在 bash 控制台下执行 `source .bashrc` 命令，即可为该用户设置名为“envname”，值为 envval 的环境变量。

如果要为所有用户都设置同一个环境变量 envname，则可以让 root 用户在 /etc/profile 中加入如下两行：

```
envname="envval"
```

export envname

### 1.4.2.2 设置会话环境变量

在控制台中，可以对当前会话设置环境变量，当会话结束，环境变量就不再存在。你可以用这种方法在批处理文件(即 dos 下的 bat 文件,或 bash 下的 sh 文件)中临时建立环境变量完成某些任务。

在 dos 控制台下，环境变量的设置方法是：set envname=envval。

在 bash 下，变量的设置方法是：envname="envval"或者 export envname="envval"。两者的区别在于：前者仅存在于当前的 shell，是本地变量，当前 shell 的子进程不会意识到这个变量的存在。为了把变量传递到子 shell，需要用 export 命令（即后者）把它们输出出来，被输出出来的变量就像环境变量一样。

## 1.4.3 Eclipse 的安装和使用

Eclipse 是一个广泛使用的开源 IDE，在各种平台上都有版本发布。在本书的课程设计中，可以选用它作为 Java 程序的开发环境，并且用到它附带的 JDT 之一：AST 类层次结构。

考虑到大多数学生之前接触的多为 Visual Studio 之类的 IDE，而且 Eclipse 中许多操作方式与之不同，因此在这里简单介绍一些 Eclipse 中的基本概念，以帮助学生快速熟悉它。

### 1.4.3.1 Eclipse 的获得与安装

**获得Eclipse:** 你可以从 <http://download.eclipse.org/eclipse/downloads/>上获得Eclipse。单击该页面Latest Release标签栏中Build Name标识符下的版本号的链接，选择Eclipse SDK栏中相应操作系统平台对应的Eclipse SDK压缩文件进行下载。

**解压和安装:** 将 Eclipse 的压缩文件解压到一个新文件夹，这里记作 ECLIPSE\_HOME，双击其中的 eclipse.exe（在 Windows 下）或 eclipse（在 Linux 或 Unix 下）即可启动 Eclipse。

**注意:** 在运行Eclipse之前，请确认已经安装JDK 1.5 以上的版本。如果你没有事先安装JDK，或者在环境变量PATH中没有包含JRE或JDK可执行文件的路径，那么Eclipse将无法启动，并给出提示信息（如图 1-2 是在Windows下的信息提示）。

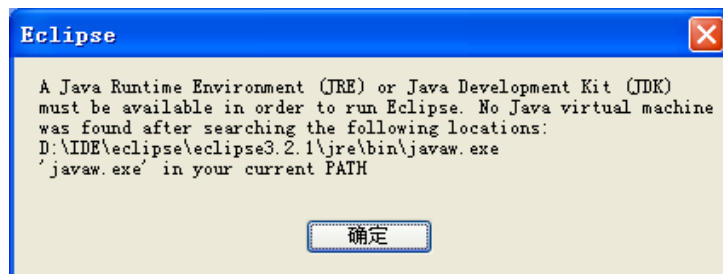


图 1-2 “找不到 Java 虚拟机”的信息提示框

### 1.4.3.2 首次运行 Eclipse

第一次运行Eclipse时，Eclipse会弹出如图 1-3 所示的窗口，要求我们指定一个Eclipse的Workspace(工作区)。



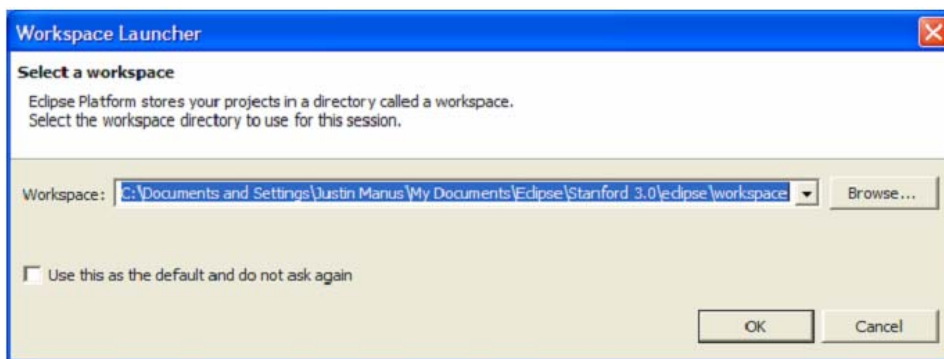


图 1-3 选择工作区

Eclipse 中的 Workspace 与 Visual C++ 6.0 的 Workspace、Visual Studio .NET 的 Solution 类似，一个 Workspace 本身只是一个文件目录，Eclipse 会把新的工程(Project)放在这个目录下。一个 Workspace 可以包含一个或多个工程，工程之间允许有依赖关系。不同之处在于，Eclipse 每次只能打开一个 Workspace，它默认会在启动 Eclipse 时询问打开哪个 Workspace，也可以在启动 Eclipse 后选择菜单项 File 中的 Switch workspace 来切换 Workspace。

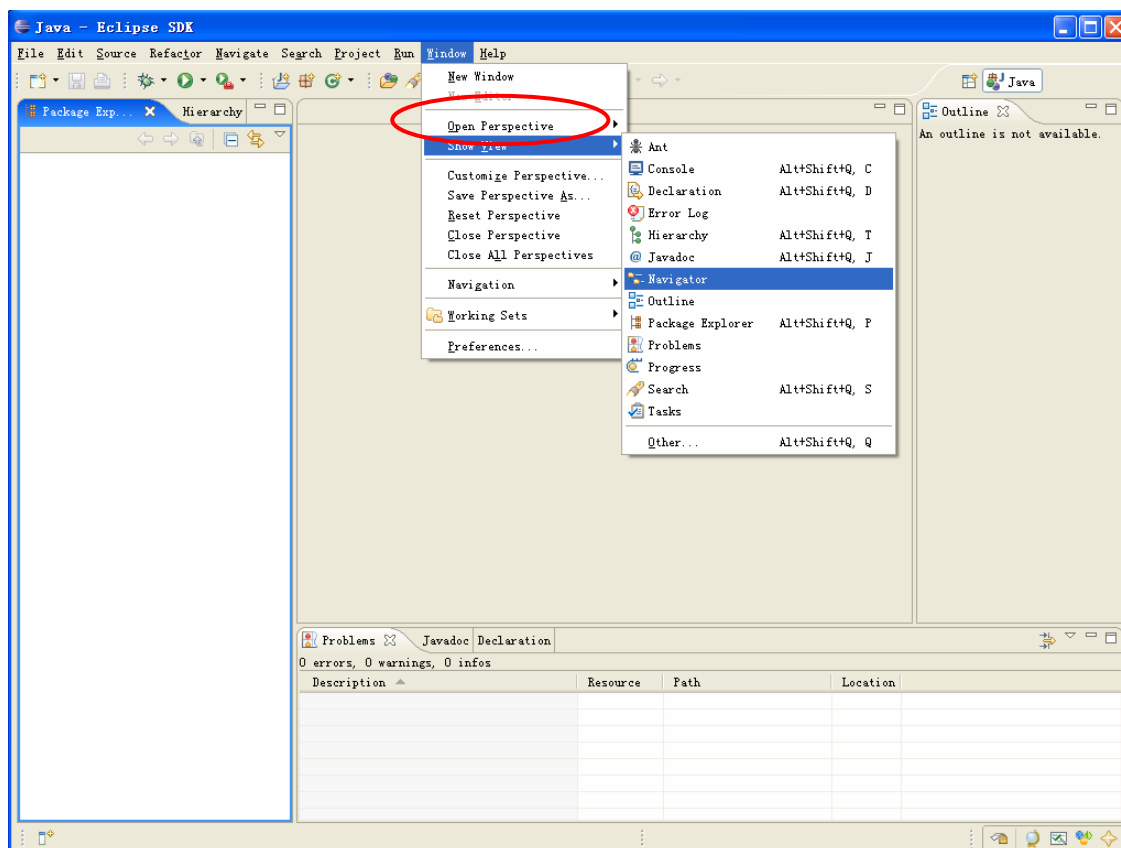


图 1-4 Eclipse 工作台

在这里，建议选择一个独立的、与系统无关的目录保存你的工作区，如 D:\workspace。在选择了合适的工作区之后，即进入Eclipse的欢迎界面。在这里，有对Eclipse简单的介绍。单击Go to Workbench可以跳过教程，进入如图 1-4 所示的工作台(Workbench)。

在Eclipse工作台中，菜单行和工具按钮行下方的大片区域被称为Perspective，它表示在



工作台中有哪视图(View)，这些View是如何分布的。View即Perspective中的那些小窗口，如Console(控制台)、Hierarchy (类层次)等(详见图1-4中菜单项“Show view”所展开的各个菜单项)。Eclipse有默认的Java、Debug等perspective，你可以通过单击菜单项“Window”→“Open Perspective”，选择所期望的perspective。在Eclipse顶端的标题栏上，会显示当前的perspective名称，格式是perspectiveName-Eclipse SDK。你也可以在一个perspective中开启或关闭一些View来改变它，并且可以通过菜单项“Window”→“Save Perspective As”将自己配置的perspective保存起来以方便切换。另外，你可以通过菜单项“Window”→“Preferences”，在弹出的Preferences窗口中，点击“General”→“Perspectives”选项，删除无用的perspective，并可以设置缺省值。

### 1.4.3.3 建立一个工程

你可以新建一个全新的工程，也可以从现有的存档文件或目录创建一个工程。下面分别简要介绍。

#### 1、建立一个新的工程

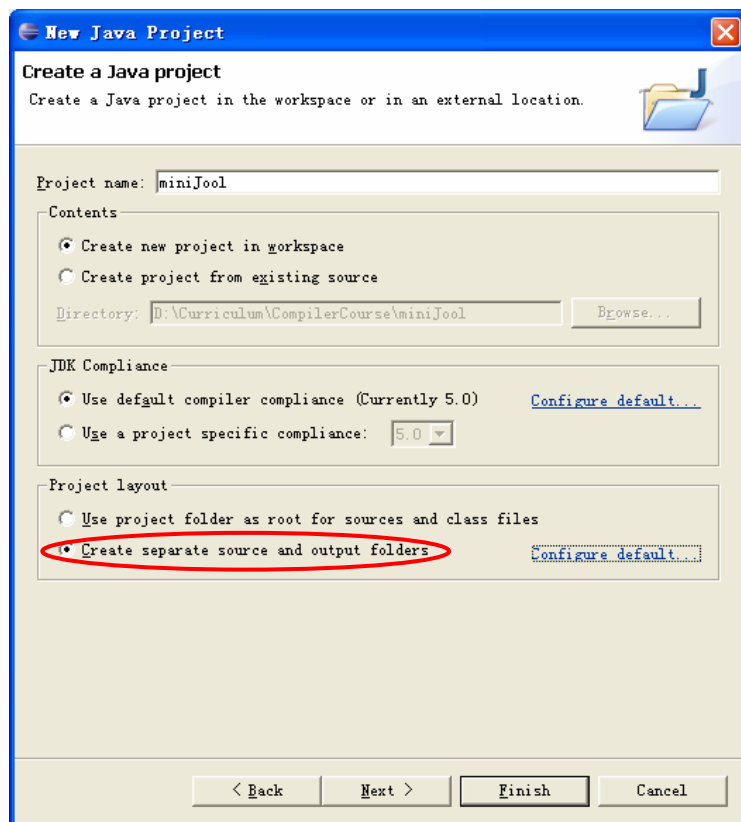


图 1-5 建立一个新的 Eclipse 工程

在选择一个 Workspace 目录之后，如果要在该 Workspace 下面展开工程(Project)，就需要建立新的工程。启动 Eclipse，按如下步骤可以创建一个新的工程：

选择菜单项“File”→“New”→“Project”，在弹出的窗口中选择“Java Project”，再点击“Next”，即可得到如图1-5所示的对话框。

在“Project Name”栏中填上工程名，如 miniJool。在“Project layout”栏中选择“Create

separate source and out put folders”选项，可以将源文件和输出文件用不同的目录分开存放；进一步点击该选项右边的“Configure default...”，可以在弹出的窗口中设置所希望的源文件夹和输出文件夹的名称，缺省情况下它们分别为 src 和 bin。你还可以通过“JDK Compliance”栏来设置该工程所使用的 JDK 的兼容性。在本课程设计中，你需要将 JDK 兼容性设置为 5.0。在设置完上述选项之后单击“Finish”按钮，即完成一个工程的创建，你将在工作台的“Package Explorer” View 中看到你所创建的工程。

我们为各个课程设计提供了程序框架，你可以以这些程序框架为基础创建自己的工程。下面介绍几种从现有源中创建工程的方法。

## 2、从现有的文件目录创建工程

按照上面介绍的方法，当弹出图 1-5 所示的窗口时，在“Contents”栏中选中“Create project from existing source”选项，并设置 Directory 项（如图 1-6），可以将 Project 创建在指定的目录下，此时该目录下原有的文件将直接被导入到新建的工程中。

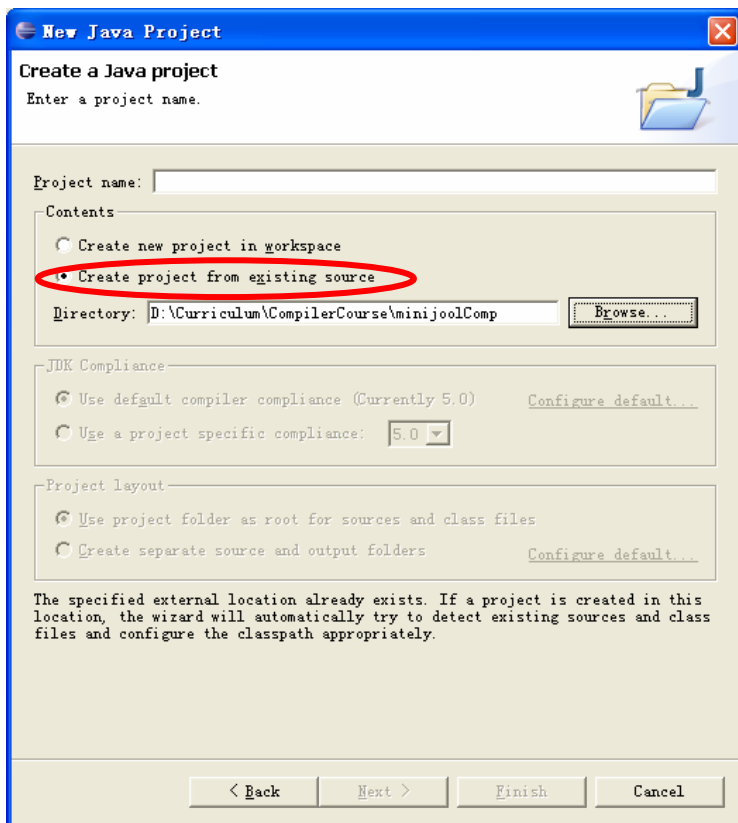


图 1-6 从现有的文件目录中创建工程

## 3、从现有的 Ant 编译文件中创建工程

选择菜单项“File”→“New”→“Project”，在弹出的窗口中选择“Java Project from Existing Ant Buildfile”，再点击“Next”，即可得到如图 1-7 所示的对话框。

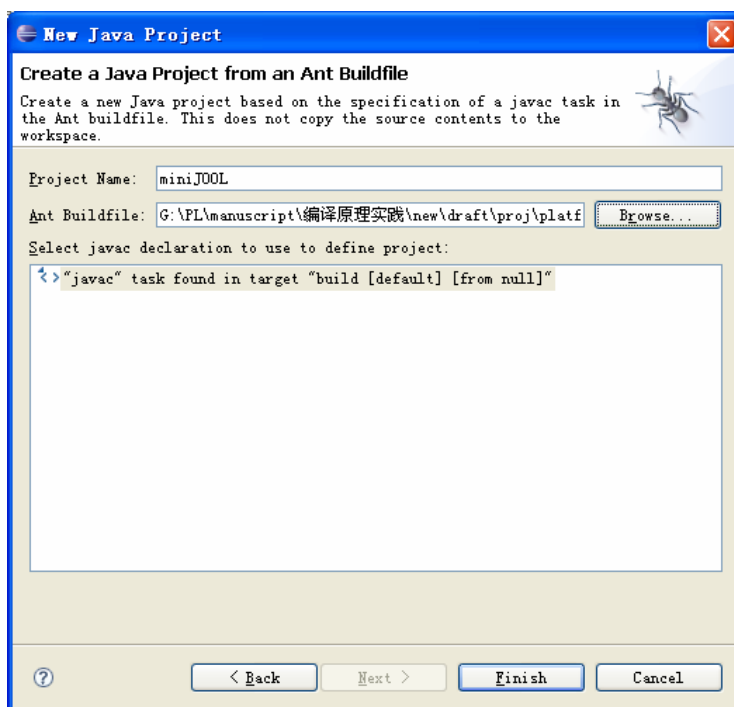


图 1-7 从 Ant 编译文件中创建工程

你可以通过“Browse...”选择 Ant Buildfile，设置“Project Name:”项中的工程名称，再点击“Finish”完成工程的创建。这时，Eclipse 将在 Workspace 目录下建立以工程名命名的子目录，并在该子目录中建立 3 个文件，即 .project、.classpath 和 build.xml。其中，.project 和 .classpath 是每个 Eclipse 工程的工程文件，在这两个文件中分别配置有工程以及 classpath 方面的信息，而 build.xml 则是根据创建工程时提供的 ant 编译文件生成的。你可以查看这 3 个文件的内容来加深你对 eclipse 开发的理解。

#### 4、将存档文件导入到现有工程中

选择菜单项“File”→“Import”，或者在“Package Explorer”View 中相应的工程上单击鼠标右键选择菜单项“Import”；在弹出的 Import 窗口中选择“General”下的“Archive file”，再点击 Next，接着在弹出窗口中的“From archive file”项中选择要导入的压缩包，如 Lab1.zip；在 Into folder 中填 Lab1，勾选“Overwrite existing resources without warning”，然后点击“Finish”，即可将存档文件导入到现有工程中。

#### 5、将现有工程导入到工作区中

选择菜单项“File”→“Import”，在弹出的 Import 窗口中选择“Existing Projects into Workspace”，再点击 Next；接着在弹出窗口中的“Select root directory”项中选择要导入的工程的根目录，如 E:\CompilerProj\student\lab\lab1；这时会在“Projects:”文本框中列出该目录下的工程名，缺省时该工程名左边的检查框是被勾选的（如图 1-8 所示），你直接点击“Finish”，即可将这个工程导入到工作区中。

如果你指定的工程根目录下没有 .project 和 .classpath 文件，这时“Projects:”文本框中将没有列出任何工程，并且“Finish”按钮也是不可以操纵的。

为了便于你快速建立 Eclipse 工程，在本书的各个课程设计软件包的根目录下，我们都给出了该课程设计工程的.project 和.classpath 文件。由于 Eclipse 不支持在配置文件中出现任意形式的相对路径，故在给出的.project 和.classpath 文件中使用绝对路径表示该工程所需的资源位置，同时假设提供给你的编译实验软件包都放在 E:\CompilerProj\student 目录下，记为 ROOT\_DIR。

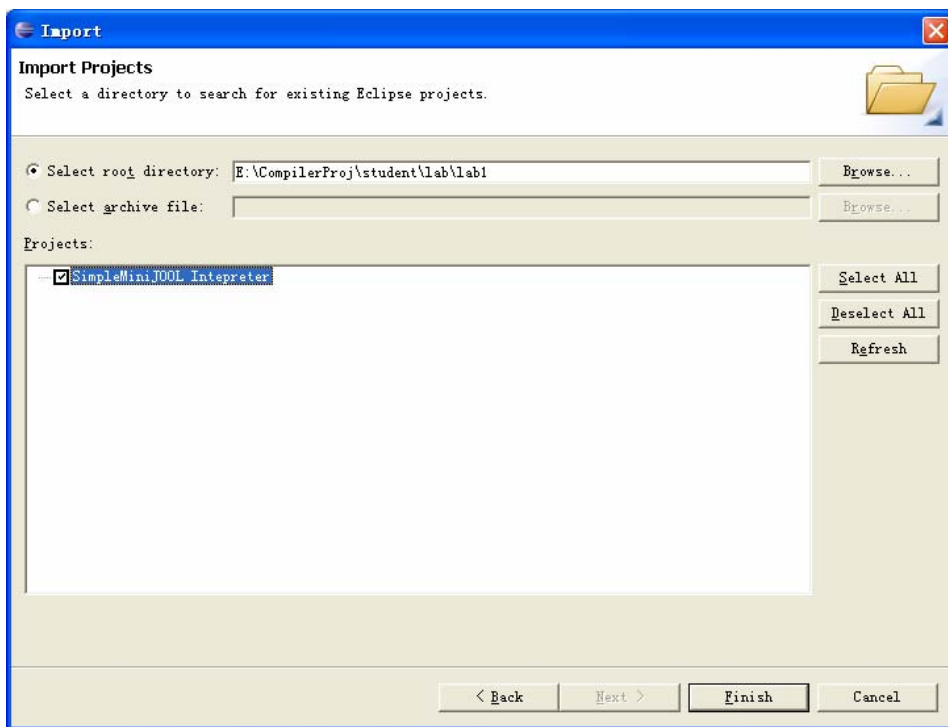


图 1-8 将现有工程导入到工作区

## 6、解决不同系统间的中文支持问题

使用 Eclipse 时遇到的常见问题之一是：在 Windows 下编写的代码中，若包含有中文或者其它宽字符，则这些代码可能会在现有的 Linux（Unix）系统中显示为乱码。这是因为两个不同的操作系统使用了不同的编码格式。为避免这个问题的发生，建议你在编程时统一使用 UTF-8 格式的编码。

如果你在 Eclipse 编程中遇到了上述问题，你可以按下面的方法修改 Eclipse 编程环境中的编码方式：打开工程的 Properties（如图 1-9 所示），修改“Info”页中的“Text File encoding”栏，选择其中的“Other”，并将内容更改为“UTF-8”（或 GBK）即可。

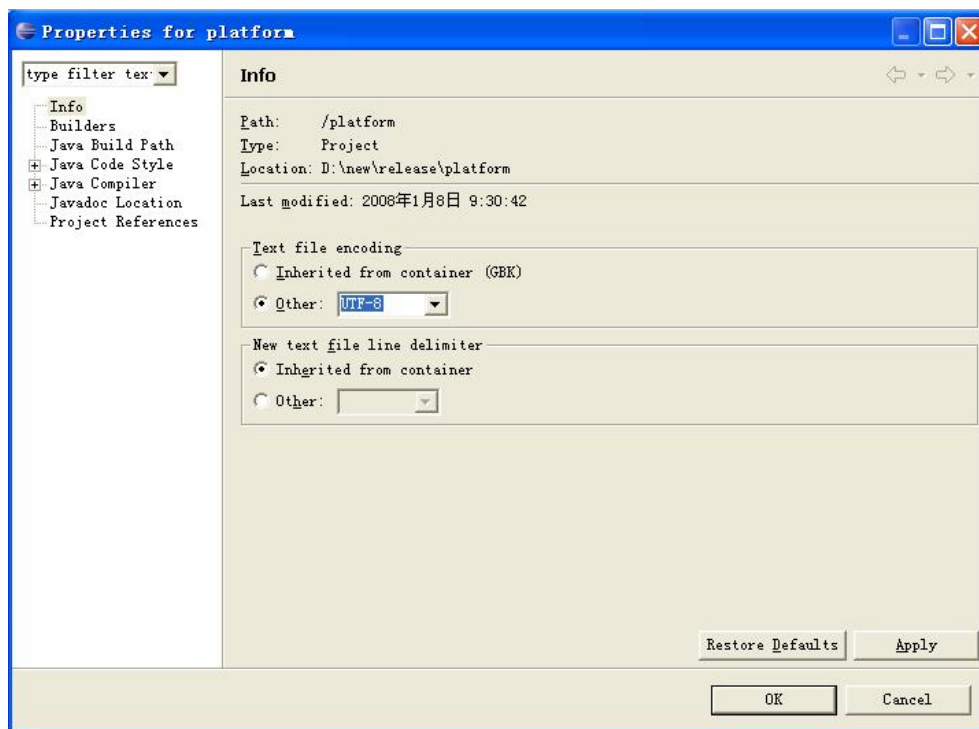


图 1-9 字符编码方式设置

#### 1.4.3.4 配置 CLASSPATH 环境变量

就像 C、C++ 程序在编译运行时需要包含各种各样的头文件或/和库文件一样，Java 程序在编译运行中不仅需要 JDK 或 JRE 的支持，也可能会需要用到各种各样的类库文件（class 或 jar 文件），如本课程设计将依赖 Eclipse 的 JDT 中的部分类库。当 Java 程序在编译运行时需要用到某个类时，JDK 或 JRE 会在名为“CLASSPATH”的环境变量中所列的路径里面按顺序寻找第一个符合的类库文件并使用它。在 1.4.2 节中，已经介绍了在不同操作系统或 Shell 下设置环境变量的方法。这里进一步介绍在 Eclipse 下设置 CLASSPATH 的方法。

在编译依赖 Eclipse JDT 的程序时，由于 Eclipse 版本的不同，可能会出现找不到 JDT 的 class/jar 文件的错误，如“The type org.eclipse.jdt.core.dom.ASTNode cannot be resolved.”，这时就需要自己设置 CLASSPATH，加入缺失的 jar 包。下面介绍两种设置方法。

##### 1、通过 Eclipse 工作台来设置

设置的方法是：选择菜单项“Project”→“Properties”，在弹出的对话框（如图 1-10）中的“Java Build Path”的“Libraries”选项页里，通过点击“Add External Jars”按钮，把 Eclipse 安装目录中 plugins 子目录下的相应 jar 加到列表中，并把原有的版本号不对的去掉（如果有的话）。

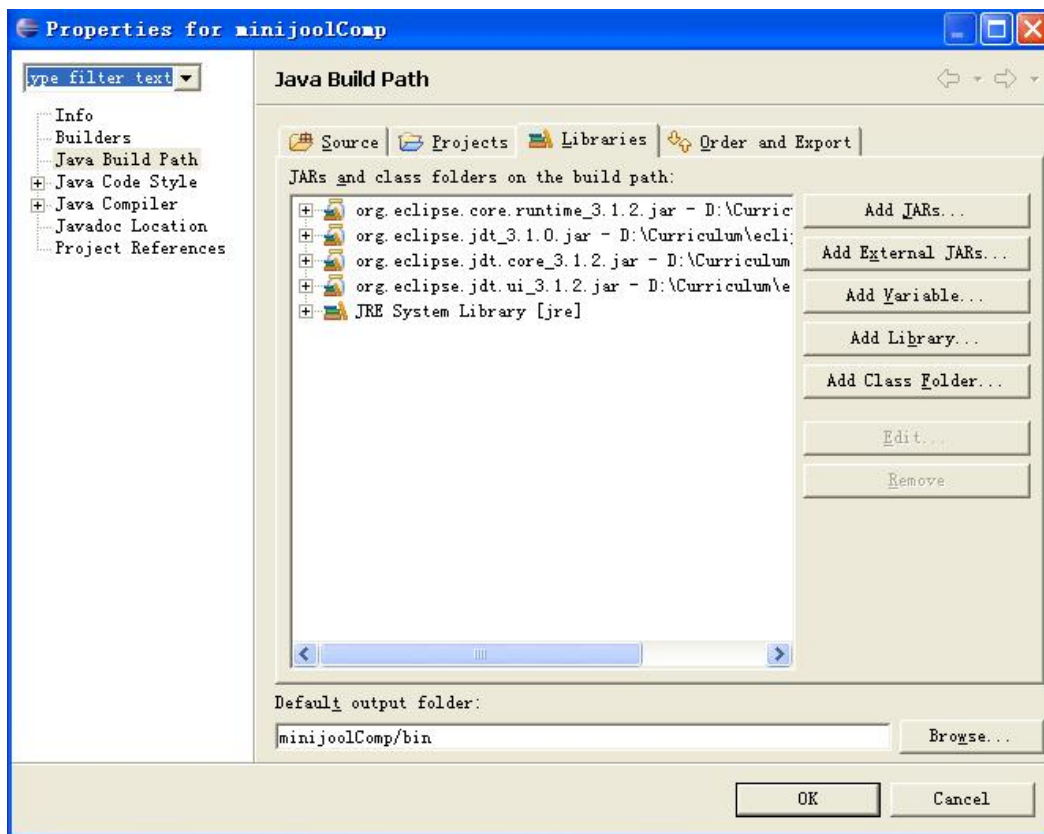


图 1-10 CLASSPATH 的设置

## 2、直接修改工程中的.classpath 文件

你也可以打开你所建工程中的.classpath 文件直接修改，这样更方便。一个普通的.classpath 文件具有如下的结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path="src"/>
  <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="output" path="bin"/>
</classpath>
```

如果要加入一条新的 CLASSPATH 信息，只需要在<classpath></classpath>之间加一条classpathentry 的条目。比如说，如果要加入类库文件，kind 可以设为"lib"，path 设为所要加入的文件的绝对路径或相对于这个.classpath 文件所在目录的相对路径。比如要加入org.eclipse.core.runtime\_3.1.x.jar 这个类库文件，可以在<classpath>...</classpath>中增写一条如下的条目：

```
<classpathentry kind="lib" path="ECLIPSE_HOME\plugins\org.eclipse.core.runtime_3.1.2.jar">
```

### 1.4.3.5 在当前工程中建立新的包、类或接口

在“Package Explorer” View中各工程的任何一个路径上点击右键，选择“New”，此时可以进一步在右键菜单中选择建立包(Package)、类(Class)、接口(Interface)等。图 1-11 是选择新建Java类时弹出的窗口。你可以在该窗口中填入或选择类所在的文件目录(Source

folder)、所属包(Package)、类名(Name)、类的修饰符(Modifiers)、所继承的父类(Superclass)、要实现的接口(Interfaces)等,在单击“Finish”按钮后,Eclipse即会在相应目录下自动生成一个空的类文件,你可以在其中完善类的内容。

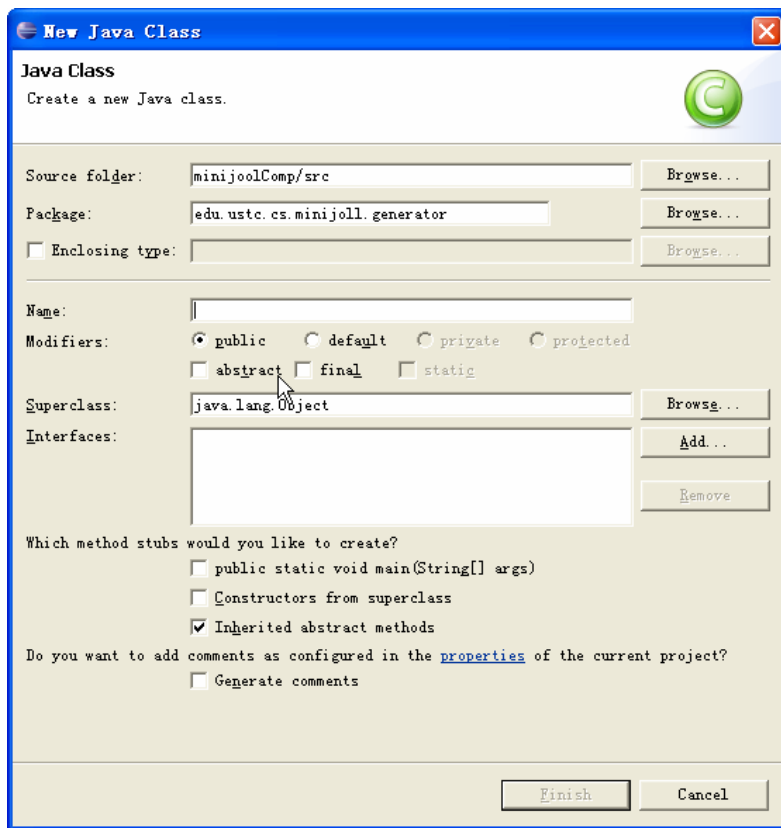


图 1-11 新建一个 Java 类

#### 1.4.3.6 在 Eclipse 中编译 Java 程序

一般地,Eclipse 会缺省地对“Package Explorer”View 中的所有工程进行自动编译,即菜单“Project”中的“Build Automatically”项是打勾的。你可以点击“Build Automatically”来取消打勾。这时,你就可以自主地通过“Project”中的“Build All”或“Build Project”来让系统编译所有的工程或者当前的工程。你可以通过选择“Build Working Set”→“Select Working Set”,在弹出的窗口中选择设置待编译的若干工程。

#### 1.4.3.7 在 Eclipse 中调试或运行 Java 程序

一个 Java 程序允许有一个或者多个入口 Main 函数,在 JVM 上运行某个程序的时候,需要指明所使用的是哪个类中定义的 Main 函数,这个类被称为主类。

Java 程序可以在 Eclipse IDE 下调试或运行。一种简单的方法是在“Package Explorer”View 中选择主类,如 Main.java,单击鼠标右键菜单项“Run as/Debug as”→“Run/Debug”,并在弹出的窗口中配置 Java Application 运行或调试框架,或者直接执行右键菜单项“Run as/Debug as”→“Java Application”,即可以运行或调试从该主类的 Main 函数开始执行的 Java 程序(如图 1-12 所示)。



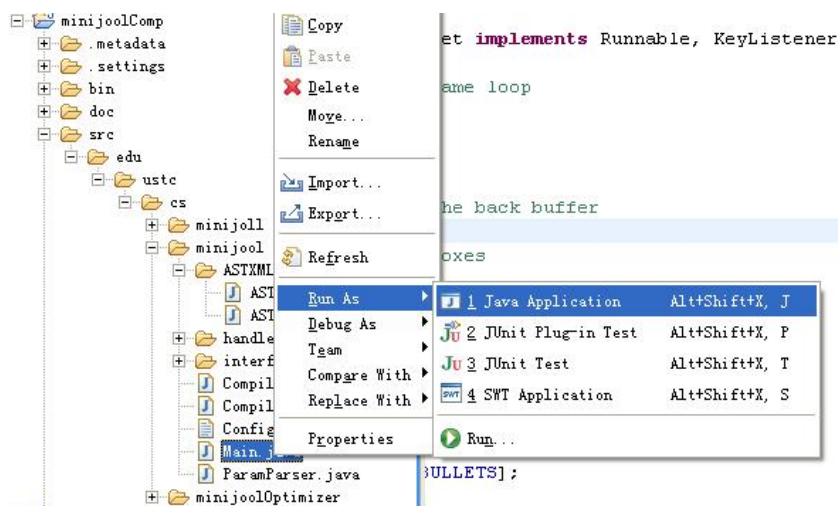




图 1-12 在 Eclipse 中运行 Java 程序

你也可以通过菜单项“Run”下的各个菜单项来控制 Java 程序的运行和调试。在工作台的命令行按钮中有   两个按钮，前者表示 Debug，后者表示 Run。点击按钮右边向下的小箭头，可以在下拉菜单中选择菜单项启动 Debug 或 Run。

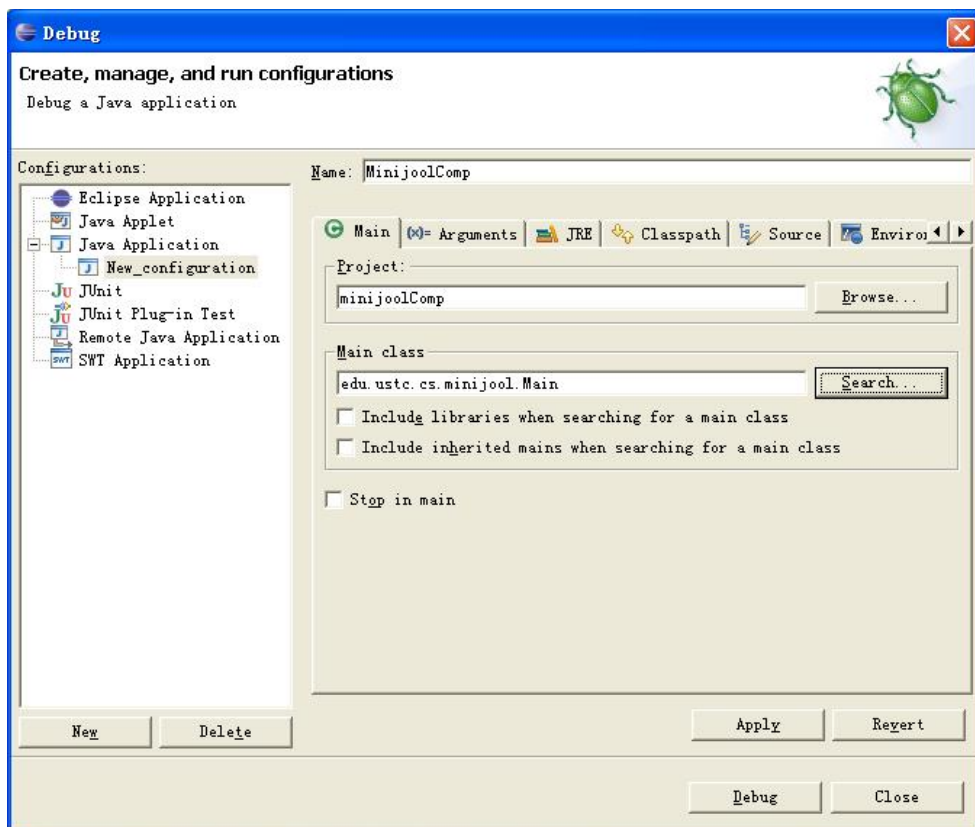


图 1-13 Debug 配置窗口

图 1-13 给出了配置 Java Application 调试框架的窗口。在 Configurations 栏中分类记录了在当前 Workspace 中已经创建的应用程序标签。你可以按下面的方法建立一个 Debug Configuration: 首先, 选择 Configurations 栏中的 Java 程序类型, 它对应需要调试的程序类型, 在此我们选择 “Java Application”, 然后单击鼠标右键选择 “New” 菜单项, 此时会在



“Java Application”下出现一个新的配置项标签，在Configurations栏的右边则以多属性页的形式显示该配置的详细信息，你可以进一步对其进行设置。

下面重点介绍如下几个属性页：

- ✓ Main 属性页：该页用于设置当前配置所针对的工程名（即 Project 栏）以及主类。
- ✓ Arguments 属性页：在该页可以设置 Main 函数的输入参数以及 JVM 的运行参数。
- ✓ Classpath 属性页：在这里可以添加运行该程序所依赖的类库或者所依赖的其他工程。默认会加入 JRE 自带的类库。此时我们不需要对它做改动。

设置完成后，选择主函数所在的类文件，单击右键选择 Debug As -> Java Application 后即可可以进行调试。Run 的配置方法与 Debug 配置类似，这里不再赘述。

## 1.4.4 XML 与 Ant 简介

Ant 是一个基于 Java 的编译工具，它根据指定或默认的 ant 编译文件（build file）中规定的任务来进行编译。Ant 编译文件是一个 XML 文件。下面先简要介绍 XML，然后再介绍 Ant。

### 1.4.4.1 XML

XML（eXtensible Markup Language）是可扩展标记语言，它以树形的形式来组织所描述的数据。一个 XML 文件中有元素、属性、声明、注释等成分。

#### 1、Element(元素)

元素是组成 XML 文档的最小单位。一个元素由一个标签(tag)来定义，包括开始和结束标签以及其中的内容，例如，`<project>...</project>`为一个 XML 元素，“`<project>`”为开始标签，“`</project>`”为结束标签。如果元素开始标签和结束标签之间的内容为空，则可以简写，例如写成`<project />`。

#### 2、Attribute(属性)

属性是对元素标签进一步的描述和说明，一个标签中可以有多个属性，例如，

```
<author age="56" sex="female">An</author>
```

中，author 元素包含两个 age 和 sex 两个属性。

#### 3、Declaration(声明)

在所有 XML 文档的第一行都有一个 XML 声明。这个声明表示该文档是一个 XML 文档，以及它遵循的是哪个 XML 版本的规范。如：`<?xml version="1.0"?>`。

#### 4、Comment(注释)

由`<!--`和`-->`包含起来的字符串是注释字符串。如：`<!--Comment goes here. -->`。

每个 XML 文件只有一个根元素，每个元素可以有自已的属性和子元素。

### 1.4.4.2 Ant

#### 1、安装 ANT

你可以从 <http://ant.apache.org> 下载 Ant 压缩包，然后解压到某个目录，假设记为 D:\IDE\ant，在环境变量中加入名为 ANT\_HOME 的环境变量，值为 ant 所在的解压路径。再在 PATH 环境变量中加入 %ANT\_HOME%\bin（针对 Windows 环境）即可。

#### 2、编写 Ant 的编译文件

Ant 的作用如同 Make，它自动地执行编译文件中所定义的任务。编译文件是用 XML 语言描写的 ant 任务配置文件，它的默认名为 build.xml。

```
<project name="MyProject" default="run" basedir=".">
  <property name="SRC_DIR" location="${basedir}/src"/>
  <property name="DEST_DIR" location="${basedir}/bin"/>
  <property name="DOC_DIR" location="${basedir}/doc"/>
  <path id="project.class.path">
    <pathelement location="${DEST_DIR}"/>
  </path>
  <target name="build">
    <mkdir dir="${DEST_DIR}"/>
    <javac srcdir="${SRC_DIR}" destdir="${DEST_DIR}"/>
    <jar destfile="myproject.jar" basedir="${DEST_DIR}"/>
  </target>
  <target name="doc">
    <mkdir dir="${DOC_DIR}"/>
    <javadoc sourcepath="${SRC_DIR}" destdir="${DOC_DIR}" Locale="en_US"/>
  </target>
  <target name="run" depends="build">
    <java classname="ScopeVariable">
      <classpath refid="project.class.path"></classpath>
      <!--classpath>
        <pathelement location="${DEST_DIR}"/>
      </classpath-->
    </java>
  </target>
</project>
```

图 1-14 一个 ant 编译文件示例

图 1-14 给出了一个 ant 编译文件示例，下面结合这个例子来说明编译文件的内部结构。

**工程(project):** 每个编译文件实际上是一个工程，故在 ant 编译文件中，project 元素是

根元素，其他元素都将在 `project` 元素内声明。`project` 元素允许带有属性，例如 `name` 属性用来指定该工程的名字，`default` 属性用来指定这个工程缺省执行的任务组，`basedir` 属性用来指定执行这个 `ant` 编译文件时的基路径。

**任务组(target):** 在编译文件中，一个任务组是一个名为`target`的元素，它至少有一个属性`name`指明自己的名称，外加可选的一个或多个其它属性（包括`depends`、`refid`等）。一个任务组可以包含多个任务(task)，执行一个任务组就是批量地执行任务组内定义的任务。一个编译文件中可以定义多个任务组，如图 1-14 中包含有名为`build`、`doc`和`run`三个任务组。

任务组的名称是重要的。`Ant` 的作用是成批地执行某些单一的指令，而任务组就是执行中与人交互的最小单位。你可以在控制台输入 `ant build` 命令行来执行当前目录下的 `build.xml` 中定义的 `build` 任务组，或者在任意目录下用 `ant -buildfile=XXX build`（“-buildfile”可以换成“-file”或者“-f”）执行指定的编译文件 `XXX` 中定义的 `build` 任务组。你也可以同时指定运行多个任务组，例如，输入 `ant build doc` 命令行来依次执行 `build.xml` 中定义的 `build` 任务组和 `doc` 任务组。

当编译文件包含多个任务组时，在这些任务组之间允许建立依赖关系。假设在某个编译文件中有两个任务组A和B，其中A有属性`depends="B"`，在执行A的时候，`ant`会先执行A所依赖的任务组B，然后再执行A本身。B也可以依赖于C、D等其它任务组，这样在执行B之前，C和D都会首先被执行。在图 1-14 中，`run`任务组依赖`build`任务组。

**任务(task):** `Ant` 中的任务是指一段能够被执行的代码。任务的通用格式如下所示：

```
<name attribute1="value1" attribute2="value2" ... />
```

`Ant` 通过准确地执行任务中定义的步骤来完成它的使命。下面介绍几个常用的任务声明：

```
<echo message="Message goes here." />
```

在屏幕上输出相应的信息。

```
<javac srcdir="src/main/hello\ant" destdir="build/classes"/>
```

用 `javac` 编译 `src/main/hello\ant` 目录下的\*.java 文件，生成的代码放在 `build/classes` 下。

```
<mkdir dir="c:\workspace"/>
```

建立目录 `c:\workspace`。

图 1-14 中`build`任务组包含`mkdir`、`javac`和`jar`三个任务，分别用来建立目录、编译Java程序、以及将编译得到的class文件打包成jar文件；`doc`任务组包含`mkdir`、`javadoc`两个任务，分别用来建立目录、从指定文件夹中的源程序提取注释生成HTML文档；`run`任务组包含`java`任务来执行指定的主类`ScopeVariable`。

**属性(property):** 属性（元素名为`property`）是任务的一种，它的作用与环境变量和Macro类似，属性可以作为`project`的子元素，也可以作为`target`的子元素。通过定义一个属性，我们可以在编译文件中的其它位置显式或者隐式地引用属性的名称和值。例如，在图 1-14 中声明了 3 个`property`元素：第 1 个`property`元素声明一个名为`SRC_DIR`的属性，其`location`属性指定这个属性所代表的路径，这里用的是相对路径（相对于当前`ant`编译文件所在的路径）。

**路径(path):** `path`元素用来定义一组路径，其属性`id`用来指定该`path`元素的标识，这样其他元素（如图 1-14 中的`classpath`元素）可以通过属性`refid`引用这个`path`元素。`path`元素内可以声明多个`pathelement`元素，每个`pathelement`元素指定一个路径。

上面只列出一些常用的元素，关于 `ant` 更多的使用特征请查询 `ant` 使用手册

(<http://ant.apache.org/manual/index.html>)。

## 第2章 MiniJOOL语言

MiniJOOL (A Mini Java-like Object Oriented Language) 语言是贯穿本书的一个用于实验的程序设计语言。它的大部分语言特性来自于 Java 语言, 但是比 Java 语言要小很多; 同时 MiniJOOL 语言在个别地方又与 Java 语言略有不同, 例如, 它仅支持一维数组, 且数组的长度必须静态指定, 即形如 “int [2] a;” 的形式。引入 MiniJOOL 语言的主要目的是: 帮助和指导你理解编译原理和技术, 开展相关的编译器实现的课程实验, 强化你的软件工程能力; 由于实验的课时和机时有限, 我们并不需要你实现一种强大的语言。

要想为 MiniJOOL 语言实现一个正确的编译器, 就必须先了解这个语言的各种特征。本章将对 MiniJOOL 语言的词法、语法及语义进行详细阐述; 并结合程序例子帮助你了解 MiniJOOL 语言的特征以及需要注意的地方。在详细阐述 MiniJOOL 语言之后, 本章进一步介绍实验中使用的 MiniJOOL 语言的两个子集: SkipOOMiniJOOL 语言和 SimpleMiniJOOL 语言。前者是去掉 MiniJOOL 语言中的面向对象特征后所形成的语言; 后者则是一个程序中仅含一个函数的简单语言。为统一起见, 我们把用 MiniJOOL 语言或其子集编写的源程序文件的扩展名均定义为 mj。

除了非形式地阐述三种语言的特征外, 本章还形式地描述 SkipOOMiniJOOL 语言的静态语义, 这个静态语义是以 SkipOOMiniJOOL 的抽象文法以及类型系统为基础的。通过静态语义的描述, 一方面能严格地定义对语言的约束规则, 另一方面也初步让你了解如何形式地定义一个语言。

### 2.1 MiniJOOL 语言简介

本节先简要介绍 MiniJOOL 语言的特点, 然后通过一个 MiniJOOL 程序示例, 让你初步认识 MiniJOOL 语言。

#### 2.1.1 MiniJOOL 语言的特点

MiniJOOL 语言是一个小型的面向对象语言。为简单起见, 这里仅列出它与 Java 2<sup>[1]</sup> 的主要区别。

- 一个 MiniJOOL 程序的所有类都在同一个文件中, 不支持 Java 语言中的 package 包管理以及 import 导入指令;
- MiniJOOL 语言不支持接口 (interface)、抽象类 (abstract class) 和抽象方法 (abstract method), 支持类的单一继承以及方法的重载 (overload) 和重写 (override);
- MiniJOOL 语言不支持 public、protected、private 访问控制修饰符, 类中所有的成员都是全局可见的;
- MiniJOOL 语言不支持异常 (Exception), 即不提供 try 和 catch 子句;
- MiniJOOL 语言的数据类型可以是类, 可以是 void、32 位整型 int、布尔型 boolean、字符串类型 String, 还可以是以 int、boolean 和 String 为基本元素类型的一维数组

类型，数组的长度必须是常量表达式；

- MiniJOOL 语言支持在类中通过 `final` 修饰符来声明一个常量，如 “`static final int ERROR=-1;`”；
- MiniJOOL 语言不支持 `switch/for/do...while` 语句，但是支持 `if/while/break/continue` 语句；
- MiniJOOL 语言不支持自增和自减运算（即 “++”、“--”）、位运算以及条件运算；
- MiniJOOL 语言提供 “`print(<表达式>;)`” 语句用来支持输出，提供 “`read(<具有左值的整型表达式>;)`” 语句用来支持输入。

### 2.1.2 一个 MiniJOOL 程序示例

图 2-1 是一个完整的 MiniJOOL 程序，其中包含 `Program`、`QuickSort` 和 `Sort` 三个类，`Program` 是主类，`QuickSort` 从 `Sort` 派生。`Sort` 类提供对内置数组的冒泡排序，而 `QuickSort` 提供对内置数组的快速排序。这个程序展示了 MiniJOOL 语言的大部分特征。一个 MiniJOOL 程序可以包含多个类，类之间可以存在继承关系（类 `QuickSort` 继承自类 `Sort`）；第一个类（主类）必须有一个 `static void` 类型的主方法 `main`；程序中可以使用包括整型 `int`、字符串类型 `String`、类类型、一维数组类型在内的多种数据类型等。你可以通过这个程序对 MiniJOOL 语言有一个感性认识。

## 2.2 MiniJOOL 语言的词法

MiniJOOL 语言的词法由两部分规则组成，一部分规定了单词的构成规则，词法分析将根据这些规则从输入的字符流中识别出单词供语法分析使用；另一部分规定了被词法分析过滤掉的符号串，包括注释和空白符号。单词可进一步分为标识符、关键字、分隔符、运算符和常量值。词法分析在识别出单词后，会把该单词对应的词法记号（token）传给语法分析器，这些记号将作为 MiniJOOL 语言语法的终结符。在附录 1 中，给出了 MiniJOOL 语言中各种单词对应的词法记号 ID 的约定。

下面简要说明注释、空白以及各种单词的构成特征。

### 1、注释

MiniJOOL 语言支持单行注释和多行注释，这些和 Java 语言的单行、多行注释一样。单行注释以 “//” 开始并延伸到行末，可以包含任何可打印的字符（ASCII 码为 32~126）。多行注释由 “/\*” 开始并以第一次遇到的 “\*/” 为结尾，内含任何可打印的字符。

**注意：**多行注释不支持嵌套！你需要特别处理 “`/**.../*...***/...*/`” 这样的情况（省略号中不包含 “`/*`” 和 “`*/`” 子串），在这个示例中，“`/**.../*...***/`” 是多行注释，但是 “`...*/`” 不属于注释中的内容。

### 2、空白符号

空格、制表符和换行符一般都视为空白符号，除非这些符号出现在字符串中。

```

// 第一个类是主类，必须有主方法 main
class Program{
    static void main( ) {
        Sort qs = new QuickSort();
        qs.sort( );
        qs = new Sort();
        qs.sort( );
        return;
    }
}

// QuickSort 类继承自 Sort，提供快速排序
class QuickSort extends Sort {
    void sort() {
        qsort(0, data.length - 1);
    }
    int partition(int low, int high) {
        int pivot = data[low];
        int i = low-1;
        int j = high+1;
        while (i < j) {
            i++;
            while (a[i] < pivot) i++;
            j--;
            while (a[j] > pivot) j--;
            if (i < j) swap(i, j);
        }
        return j;
    }
    void qsort(int low, int high) {
        if (low >= high) return;
        int p = partition(low, high);
        qsort(a, low, p);
        qsort(a, p + 1, high);
    }
}

// Sort 类：提供冒泡排序
class Sort{
    // 实例变量
    int[] data = {4,76,5,234,7};
    // 交换 data[i]和 data[j]
    void swap(int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
    // 对 data 中的数进行排序
    void sort (){
        int i = data.length - 1;
        while(i >= 0){
            int j = 0;
            while(j < i){
                if (data[j] > data[j+1]){
                    swap(j, j+1);
                }
                j = j + 1;
            }
            i = i - 1;
        }
    }
    // 打印 data 中的数
    void printArray() {
        int i = 0;
        String str = "\nMember:";
        while (i < data.length) {
            print(str); print(i); print("is ");
            print(data[i]);
        }
    }
}

```

图 2-1 MiniJOOL 程序示例：冒泡排序和快速排序

### 3、常量值

MiniJOOL 语言的值包括整型值、布尔值、空引用和字符串。

**注意：**MiniJOOL 语言中没有字符常量。

整型值有十进制数、八进制数、十六进制数几种形式。MiniJOOL 语言规定，凡是以数字 0 开头的数字序列，一律作为八进制数处理；凡是以 0x 或 0X 开头，后面跟若干位数字的，一律作为十六进制数处理。

布尔值有 true 和 false 两个；空引用为 null。

字符串是由一对双引号括起来的零个或多个字符组成的字符序列。这里的字符指的是一个单独的 ASCII 字符，除去控制字符（ASCII 码为 0~31）、双引号（ASCII 码为 34）、反斜杠（ASCII 码为 92）和 Delete 键（ASCII 码为 127）；或者是一个双字符的合法转义序列，即反斜杠后跟一个双引号、反斜杠、字母 n、字母 r、字母 t 或空格。MiniJOOL 语言不考虑反斜杠后跟数字的转义序列。

### 4、关键字（保留字）

MiniJOOL 语言包含如下关键字，这些关键字都作为保留字使用：

class	static	final	extends	void	int	boolean	String	if	else
while	break	continue	return	print	read	new	this	super	
instanceof		null	true	false					

### 5、标识符

MiniJOOL 语言中的标识符是以字母（'A'~'Z'或'a'~'z'）开始的，由字母、数字（'0'~'9'）、下划线（'\_'）组成的字符串。

### 6、分隔符

( ) { } [ ] ; , .

### 7、运算符

>	<	==	<=	>=	!=	!
&&		+	-	*	/	%
=	+=	-=	*=	/=	%=	

## 2.3 MiniJOOL 语言的语法

一个MiniJOOL程序由一个或多个类声明组成，其中第一个是主类。主类中必须含有一个类型为static void的main方法，它是MiniJOOL程序的入口。在以下各小节中，将从类型、值、变量、类、语句块和语句、表达式几方面，结合EBNF形式<sup>3</sup>来描述MiniJOOL语言的语

<sup>3</sup> 根据 ISO/IEC 14977:1996(E)规定的 EBNF 语法（<http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html>），其中终结符用引号括起，包含在一对方括号内的部分为可选符号，包含在一对花括号内的部分表示该部分重复 0 次或多次。



法，其中除标识符和常量值以外的终结符都直接用其对应的字符串值来表示。

### 2.3.1 类型、值和变量

#### 2.3.1.1 类型和值

除了 void 类型外，MiniJOOL 语言中的类型有 int（32 位）、boolean、String、数组类型以及在当前程序中声明的类类型。与 Java 语言不同的是，这里的数组类型是一维的并且必须静态地确定所声明的数组的长度，但是当数组作为形参时不必指定其长度。下面的 EBNF 式中的 constant\_expression 必须在编译时能求得为一个正整数。

```

type          =  primitive_type | "String" | array_type | class_type
primitive_type =  "int" | "boolean"
array_type     =  (primitive_type | "String") "[" [ constant_expression ] "]"
constant_expression = expression
class_type     =  IDENTIFIER

```

此外，MiniJOOL 语言中还有一个特殊的空引用类型，它是表达式 null 的类型，这个空引用类型没有名字。由于空引用类型没有名字，所以不可能声明一个空引用类型的变量。对一般的程序员来说，可以忽略空引用类型，而只把 null 当成是一个可以为任何引用类型的特殊值。

与上述类型相对应，MiniJOOL 语言中的值有整型值、布尔值、空引用和字符串四类，在 2.2 节中给出了它们的构成特征。

```

literal       =  INTEGER_LITERAL | BOOLEAN_LITERAL
               | STRING_LITERAL | NULL_LITERAL

```

在下面各小节中，将介绍 MiniJOOL 语言中各种类型的取值以及在值上能进行的运算。关于各种类型的变量及其赋值以及函数参数传递机制将在 2.3.1.2 节中说明。

#### 1、整型及其值

MiniJOOL 语言的 int 型占 4 个字节，是有符号整型，其值的取值范围在 -2147483648 到 2147483647 之间。MiniJOOL 语言提供以下在整型值上的运算：

- 比较运算，其结果为 boolean 型值
  - 数值比较运算符：<、<=、>、>=;
  - 数值相等运算符：==和!=;
- 算术运算，其结果为 int 型值
  - 一元的正、负运算符：+和-;
  - 乘除运算符：\*（乘）、/（整除）和%（求余）;
  - 加减运算符：+和-。

#### 2、布尔型及其值

boolean 型表示两种可能的值：true 和 false。boolean 型值在内存中的表示由编译器决定。MiniJOOL 语言提供以下在 boolean 值上的运算：

- 关系运算符：==和!=;
- 逻辑运算符：！（逻辑非）、&&（逻辑与）和||（逻辑或）。

boolean 表达式可以决定 if、while 语句的控制流。

### 3、字符串型及其值

这里要区分String型常量和String型变量。String型常量是由一对双引号括起来的零个或多个字符组成的字符序列。一个长度为 $n$  ( $n>0$ ) 的字符串在内存中将占 $n+1$  个字节，最后一个字节存放的ASCII码值是 0。String型变量的值是一个引用值，它引用一个字符串常量。MiniJOOL语言在字符串值上不提供任何运算，而关于String型变量的运算在 2.3.1.2 节中说明。

### 4、数组类型及其值

MiniJOOL 语言要求在声明数组时静态确定数组的长度，如“int[2] a;”、“int[] a={1,2};”或“int[2] a={1,2};”，并且允许通过“<数组名>.length”获得给定数组的长度，如 a.length 的值是 2。MiniJOOL 语言不提供在数组整体上的运算，只允许对数组元素值进行运算。对数组元素的访问可通过数组名加下标表达式来进行，如 a[1]。MiniJOOL 语言规定，对于一个长度为  $N$  的数组，其元素的下标取值范围为从 0 到  $N-1$  的整数。

### 5、类类型及其值

类类型是引用类型。类的一个实例（instance）称为一个对象（object）。引用值是指向对象的指针，而特殊的空引用不指向任何对象。一个类实例是由 new 表达式显式创建的。

MiniJOOL 语言提供以下在对象引用上的运算符：

- 域访问：使用受限名或域访问表达式来进行；
- 方法调用
- instanceof 运算符
- 引用相等运算符：==和!=。

同一个对象可以有多个引用。对象的状态存储在对象的域中。如果两个变量包含对同一个对象的引用，则可以通过其中一个变量来修改对象的状态，随后另一个变量也可以通过引用观察到改变后的状态。图 2-2 给出了关于类类型及其值使用的程序示例。

```
class Program {
    static void main() {
        A a1 = new A(); // 创建类 A 的一个对象
        A a2 = a1;       // a2 和 a1 指向同一个类 A 的对象
        B b = new B();   // 创建类 B 的一个对象

        print(A.si);    // 域访问：类名 A 作为受限名，输出“2”
        print(a1.si);   // 域访问：类类型变量 a1 作为受限名，输出“2”
        print(a1.i);    // 域访问：类类型变量 a1 作为受限名，输出“4”

        a.f();          // 类方法调用，打印“A”
    }
}
```

```

print(a1 instanceof A); // 判断 a1 引用的对象是否为类 A 的实例。输出 “true”
print(a1 instanceof B); // 判断 a1 引用的对象是否为类 B 的实例。输出 “false”

print(a1==a2); // 判断 a1 和 a2 是否引用同一个对象。输出 “true”
print(a1==b); // 判断 a1 和 b 是否引用同一个对象。输出 “false”

a1.i = 8;      // 通过 a1 修改实例变量 i
print(a2.i);   // 通过 a2 访问同一实例变量 i
    }
}
class A {
    static int si = 2;    // si 是类变量
    int i = 4;           // i 是实例变量
    void f() {
        print("A");
    }
}
class B { }

```

图 2-2 MiniJOOL 程序示例：类类型及其值

### 2.3.1.2 变量

一个变量是一个存储单元，它有关联的类型。变量总是被赋予一个与其类型相兼容的值。关于MiniJOOL语言的类型兼容原则在 2.3.1.3 节说明。

下面结合 图 2-3 中的代码片段来简要说明变量的种类、初始化以及其他使用特征。你可以阅读 2.3.2 和 2.3.4 节获得关于变量在类以及表达式中使用的更详细信息。

```

class VariousVariables{
    static final int n = 5;    // 声明一个带初始化表达式的 final 类变量 n
    static int m;             // 声明一个不带初始化表达式的类变量 m
    int x, y=8;               // x 和 y 是实例变量，声明 y 时带初始化表达式
    int[10] w;                // 声明一个长度为 10 的数组实例变量
    int[10] w1 = w;           // 产生编译时错误，不能将数组赋值给一个数组变量
    VariousVariables(int m1) { // m1 是该构造器的形参
        m = m1;
    }
    int setX(int x){          // x 是方法 setX 的形参
        int oldx = this.x;    // oldx 是局部变量
        this.x = x;           // 通过 this.x 引用实例变量 x
        w[x%10] = x;          // 通过 w 和下标表达式 x%10 来引用数组元素
        return oldx;
    }
}

```

```

    }
    static void main(){
        VariousVariables vv = new VariousVariables(2);
        .....
    }
}

```

图 2-3 MiniJOOL 程序示例：变量的种类

### 1、变量的种类

MiniJOOL 语言中的变量有以下六种：

- **类变量 (class variable)**：指在类声明中用static声明的域（即类中的数据成员），如图 2-3 中的m和n。当类初始化时，会为此类创建类变量并将类变量初始化为缺省值（注意：对于final类变量，必须被显式赋值），类变量的生命期直到程序运行结束为止。
- **实例变量 (instance variable)**：指在类声明中未用static声明的域，如图 2-3 中的x、y和w。在创建类的每一新实例（即对象）时，都要为类中的各实例变量创建一个新变量并按缺省值对变量进行初始化。实例变量的生命期随类实例的消亡而结束。
- **局部变量**：由局部变量声明语句来声明，如图 2-3 中的oldx和vv。当控制流进入一个语句块时，为在该语句块中声明的每一局部变量创建一个新变量；当退出该语句块时，局部变量不再存在。
- **方法的形参**：用来命名传递给方法的实参，如图 2-3 中的x。形参变量在方法被调用时创建，在方法体执行结束时不复存在。
- **构造器的形参**：用来命名传递给构造器的实参，如图 2-3 中的m1。形参变量在用new表达式创建类实例或者显式调用构造器时被创建，在构造器的体执行结束时不再存在。
- **数组元素**：是未命名的变量，程序中可以通过数组引用表达式（类型为数组类型）加下标表达式来使用数组元素，如图 2-3 中的w[x%10]。数组元素变量随数组变量的创建而被创建并被初始化，随数组变量的消亡而结束生命期。

### 2、final 变量

只有类变量和实例变量可以声明为final，final变量只能被赋值一次。如果final变量被赋值多次，将产生编译时错误。声明为final的类变量必须在声明时带初始化表达式，如图 2-3 中的“n = 5”，否则产生编译时错误。

### 3、变量的初值

MiniJOOL 程序中的每一变量在使用前必须有值。

- 类变量、实例变量或数组元素在创建时按缺省值初始化。int型变量的缺省值是0，boolean型变量的缺省值是false，类类型和String型变量的缺省值是null。例如，图 2-3 中m、x以及w的10个元素均缺省为0。

- 方法或构造器的形参被初始化为由调用者提供的对应的实参值。
- 局部变量必须在使用前被显式赋值，否则产生编译时错误。
- 类变量和实例变量在声明时可以带初始化表达式，如 “`static int a=4; int[2] b={a, 2}; String[] ss={"year", "month", "day"};`”。类变量的初始化表达式在类初始化时求值并被赋给类变量，而实例变量的初始化表达式则在实例创建后、构造器执行前被求值并赋给变量。局部变量在声明时也可以带初始化表达式，它在执行这个声明语句时被求值并赋给相应的局部变量。

### 数组声明中的初始化表达式

针对数组变量在声明时的初始化，MiniJOOL 语言中的规定与 Java 语言不太一样。在 Java 语言中，当数组变量在声明时含有初始化表达式（即包含在花括号内、由逗号分隔的表达式序列）时，不能指定数组的长度，数组变量的长度由初始化表达式中的表达式个数决定；此外，花括号中的表达式序列允许以逗号结尾。例如 “`int[2] b={1, 2};`” 将产生编译错误，而可以使用 “`int[] b={1, 2};`” 或 “`int b[]={1, 2};`”。在 MiniJOOL 语言中，数组的初始化表达式同样是包含在花括号内、由逗号分隔的表达式序列，但是它不允许以逗号结尾；当数组变量在声明时含有初始化表达式时，可以同时指定数组的长度，并且要求指定的长度必须与初始化表达式中的表达式个数一样。例如 “`int[2] b={1, 2};`” 和 “`int[] b={1, 2};`” 是正确的，但是 “`int[3] b={1, 2};`” 将产生编译时错误。

## 4、变量所允许的类型和使用特征

类变量、实例变量、局部变量、方法或构造器的形参都可以声明为 `int`、`boolean`、`String`、数组类型或类类型。但是，当类变量、实例变量和局部变量声明为数组类型时，必须指定数组的长度；而当方法或构造器的形参声明为数组类型时则不必指定数组的长度，这时形参将引用传来的实参数组。在 MiniJOOL 语言中，方法或构造器的参数传递方式是单向的值传递。

下面简述各种类型变量的使用特征：

- `int` 型变量只能被赋予 `int` 型值；一旦它被赋值，则可以参加比较运算和算术运算，也可以作为实参传递给 `int` 型形参，还可以传给 `print` 语句输出到标准外设。
- `boolean` 型变量只能被赋予 `boolean` 型值；一旦它被赋值，则可以参加关系运算和逻辑运算，可以作为 `if` 和 `while` 语句的条件，也可以作为实参传递给 `boolean` 型形参，还可以传给 `print` 语句输出到标准外设（注意：输出的是字符串 `true` 或 `false`）。
- `String` 型变量只能被赋予字符串常量或已被赋值的 `String` 型变量；如前所述，`String` 型变量保存的是对字符串的引用值，这样，一个字符串常量就可以被多个字符串变量所引用。此外，`String` 型变量在被赋值（注意：不是 `null`，而是引用一个实际的字符串）之后可以作为实参传递给 `String` 型形参，还可以传给 `print` 语句将所引用的字符串输出到标准外设。
- 数组变量只能作为实参传递给具有相同元素类型的、数组类型的形参，这时数组形参将引用实参数组。数组类型的形参可以接收不同长度的、但元素类型相同的数组类型的实参。不能将一个数组变量赋值给另一个数组变量，否则产生编译时错误（如图 2-3 中的 `w1=w` 是错误的）

- 数组元素在使用前时需要做下标是否越界的检查,这种越界检查要求尽量在编译时完成,对于编译时无法判断的,则应在运行时检查。当数组元素越界时,应产生编译或运行时错误。除此之外,数组元素在使用上与类型和它相同的变量没有差异。
- 类类型的变量缺省为 `null`,它可以通过 `new` 表达式被实例化,也可以引用一个由类型为该类或其子类的变量传来的实例;一旦类类型的变量被赋值,则可以进行域访问、方法调用、检测其运行时类型、判断引用是否相等,还可以传递给具有相同类类型或超类类型的形参。

### 2.3.1.3 类型兼容原则

- 一个类型与其自身相兼容;
- 可以将一个类型为  $\tau[N]$  ( $\tau$  为 `int`、`bool` 或 `String`,  $N$  为正整数) 的值赋值给类型为  $\tau[]$  的变量;
- 若  $A$  是  $B$  的子类,则声明为  $B$  的变量可以引用类型为  $A$  的实例。

## 2.3.2 类

在 MiniJOOL 语言中没有 `public`、`protected` 以及 `private` 这组访问控制修饰符,所有的类及其数据成员(域)和方法成员都是对外可访问的。

### 1、类声明

类声明中的 IDENTIFIER 为类名,如果一个 MiniJOOL 程序内声明了多个同名的类,则产生编译时错误。类声明中可选的 `extends` 子句用来指定该类的直接超类(即父类)。需要指出的是,主类没有父类,如果主类有 `extends` 子句,则产生编译时错误。

```
program = class_declaration { class_declaration }
class_declaration = "class" IDENTIFIER [ "extends" IDENTIFIER ] class_body
```

### 2、类成员

一个类的体由若干个成员声明组成,包括域、方法和构造器。对一个在类  $C$  中或被类  $C$  继承的成员  $m$  来说,其作用域是类  $C$  的整个体。一个类的成员包括从其父类继承而来的所有域成员和方法成员,以及在这个类的体中声明的成员。

```
class_body = "{" class_body_declaration { class_body_declaration } "}"
class_body_declaration = field_declaration | method_declaration | constructor_declaration
```

### 3、域声明

```
field_declaration = [ "static" ] [ "final" ] type variable_declarators ";"
variable_declarators = variable_declarator { "," variable_declarator }
variable_declarator = variable_declarator_id [ "=" variable_initializer ]
variable_declarator_id = IDENTIFIER
variable_initializer = expression | array_initializer
array_initializer = "{" [ expression { "," expression } ] "}"
```

一个类的变量由域声明来引入。如果同一个类体中声明了两个同名的域，则产生编译时错误。方法名、类名和域名可以相互同名，因为它们用在不同的上下文中。

域修饰符有 `static` 和 `final` 两种。如前所述，`static` 域为类变量，而非 `static` 的域为实例变量。前者在类被初始化时创建，一个类只有一个与该域对应的变量；该类的所有实例共享这个变量；而后者在创建类的任一新实例（即对象）时被创建，每个实例各有一个与该域对应的变量。

类变量和实例变量都可以声明为 `final`，这些变量一旦被显式赋值，则禁止对其进行修改。声明为 `final` 的类变量必须在类初始化中被显式赋值，而声明为 `final` 的实例变量必须在每一构造器运行结束前被显式赋值，否则都将产生编译时错误。

一个类中的域声明可以隐藏其超类中的同名域声明，这两个域的类型（包括修饰符）允许不相同。被隐藏的域可以通过受限名来访问，例如，`static`域可以通过“<类名>.<域名>”来访问，而非`static`域可以通过“`super.<域名>`”来访问。与受限名相区分，可称非受限名为简单名。图 2-4 中的代码示意了子类对超类中域声明的隐藏，以及访问子类和超类中域的方法。

```
class Parent {
    int v1 = 2;
    int v2 = 4
    static int v3 = 8;
    static int v4 = 16;
    int v5 = 32;
    static v6 = 64;
}

class Child extends Parent {
    int v1 = 128;           // 隐藏了类 Parent 中的实例变量 v1
    String v2 = "256";     // 隐藏了类 Parent 中的实例变量 v2，即使它们的类型不同
    static int v3 = 512;    // 隐藏了类 Parent 中的类变量 v3
    int v4 = 1024;         // 类 Child 中的实例变量隐藏了类 Parent 中的类变量 v4
    void f() {
        // 以下 3 行代码引用类 Child 中的域
        int local1 = v1;
        String local2 = v2;
        int local3 = v3;
        // 以下两行代码通过受限名访问超类中被覆盖的域
        int super1 = super.v1; // 通过 super 访问超类中的实例变量
        int super3 = Parent.v3; // 通过超类名访问超类中的类变量
        // 超类中没有被隐藏的域在子类中可以直接访问
        int super5 = v5;
    }
}
```

```

// 也可以通过受限名显式访问超类中没有被隐藏的域
int super6 = super.v5;
int super7 = Parent.v6;
}
}

```

图 2-4 MiniJOOL 程序示例：隐藏域及其访问

一个域声明如果含有初始化表达式，如“`int a = 2;`”，则该初始化表达式对类变量来说会在类初始化时被计算赋值；而对实例变量来说会在类实例创建时被计算赋值。类变量的初始化表达式中不允许引用任何实例变量名、`this` 以及 `super` 关键字，否则产生编译时错误。实例变量的初始化表达式可以使用在该类及其超类中声明的任意类变量的简单名，即便这个类变量声明在后；实例变量的初始化表达式也可以通过受限名使用在其他类中声明的类变量，即便这些类声明在后。只有当类 *C* 的一个实例变量 *v1* 出现在类 *C* 的另一个实例变量 *v2* 的初始化表达式中时，*v1* 的声明才必须要求出现在其使用之前。

**注意：**当执行类实例创建表达式（见 2.3.4 节）时，将首先为新的类实例（对象）分配空间，当对象空间分配成功之后，先按缺省值对对象中的各个实例变量赋值，然后再对实例变量声明中的初始化表达式进行计算赋值，最后再查找确定要执行的构造器并执行之。因此，对于图 2-5 中的两段代码来说，当分别执行“`A a=new A();`”后，它们所创建的对象中实例变量 *i* 的值均为 3。

<pre> class A {     int i = 2;     A() {         i = 3;     } } </pre>	<pre> class A {     A() {         i = 3;     }     int i = 2; } </pre>
--	--

图 2-5 MiniJOOL 程序示例：实例变量的初始化

#### 4、方法声明

```

method_declaration=  method_header  method_body
method_header      =  [ “static” ] ( type | “void” ) method_declarator
method_declarator  =  IDENTIFIER “(” [ formal_parameter_list ] “)”
formal_parameter_list =  formal_parameter { “,” formal_parameter }
formal_parameter   =  type variable_declarator_id
method_body        =  block

```

方法用来声明可以被调用执行的代码。一个方法的形参由一组逗号分隔的参数说明列表组成，每个参数说明由类型及表示参数名的标识符组成。方法的形参的作用域是该方法的整个体，在方法体中不能声明与形参同名的局部变量。形参只可以通过简单名来使用，而不能是受限名。方法的形参可以与类中的域名同名，这时在方法体中，外层的域声明将被隐藏，



被隐藏的域名可以通过受限名来访问。即，类变量通过“<类名>.<类变量名>”的方式引用，实例变量通过“this.<实例变量名>”或者“super.<实例变量名>”的方式引用。

在MiniJOOL语言中，方法的形参类型可以是int、boolean、String、一维数组类型或类类型，返回类型可以是void、int、boolean或类类型。方法体是实现该方法的代码块（见 2.3.3 节）。代码块可以不包含任何语句，即只有“{ }”，表示该方法的实现不执行任何代码。如果一个方法声明为void，则方法体中不能含有任何带表达式的return语句。如果一个方法有返回类型，则方法体中的每一个出口都必须有带表达式的return语句且表达式的类型必须与返回类型兼容，否则产生编译时错误。

当形参为数组类型时，在声明时不能指定数组的长度，如果指定了长度，编译器将产生警告信息并忽略这个长度。对于类型为数组的形参，编译器将分配给它一个存放引用值的空间，当虚实结合时，形参单元将保存对实参数组的引用。在方法体中，对于数组类型的形参，可以通过“<形参名>.length”获得所引用的实参数组的长度。在访问数组元素时必须严格遵守下标不得越界。图 2-6 中的代码演示了这些内容。

```
class Program {
    // 输出数组中的所有元素。错误版本。
    void badDump(int[10] src) { // 编译器需要输出一个警告，并忽略这个形参数组长度
        int i = 0;
        /* 当循环体最后一次执行时，会发生数组下标越界。编译器如果不能在
           编译时发现这个错误，那么就要为 src[i] 生成下标越界检查代码，使得
           程序在运行时能够检查到下标越界。*/
        while (i <= src.length) {
            print(src[i]);
            i += 1;
        }
    }

    void goodDump(int[] src) { // 输出数组中的所有元素。正确版本。
        int i = 0;
        while (i < src.length) { // 不会产生数组越界访问
            print(src[i]);
            i += 1;
        }
    }
}
```

图 2-6 MiniJOOL 程序示例：数组型形参以及数组越界检查

一个方法的基调（signature）由方法名、方法的形参个数和各形参的类型组成。一个类中不可以出现两个具有相同基调的方法，否则产生编译时错误。

方法可以声明为 `static`，这时称之为类方法（`class method`）。在类方法的方法体中，总是不能引用这个类或这个类的超类中的非 `static` 成员，否则导致编译时错误。没有声明为 `static` 的方法称为实例方法（`instance method`）。实例方法在调用时总是与一个对象相关联，这个对象在方法体执行期间可以通过 `this` 和 `super` 来引用，称为当前对象。在实例方法的方法体中可以引用这个类或者这个类的超类中的所有成员。

### 方法的重写 (overriding) 与隐藏

一个类不仅可以继承父类中的所有方法，也可以声明一个与超类中的方法基调相同的方法。假设类 *C* 是类 *A* 的子类，*m1* 和 *m2* 分别是类 *C* 和类 *A* 中的方法，它们的基调相同。若 *m1* 是实例方法，则称 *m1* 重写（`override`，或称覆盖）*m2*；若 *m1* 是类方法，则称 *m1* 隐藏（`hide`）类 *C* 的超类中所有与 *m1* 基调相同的方法。被重写的方法或者被隐藏的方法可以通过在方法调用表达式中的方法名前分别加 “`super.`” 或者 “`<超类的类名>.`” 来访问。

需要指出的是，一个实例方法不能重写一个类方法，而一个类方法也不能隐藏一个实例方法，否则都将产生编译时错误。此外，若一个方法的返回类型与被它重写或隐藏的方法的返回类型不兼容，则也产生编译时错误。图 2-7 左部代码中的子类 *Child* 中的方法 *f()* 对超类中的方法 *f()* 的重写是正确的，因为子类中方法 *f()* 的返回类型向上兼容超类中的方法 *f()* 的返回类型。图 2-7 右部代码中的子类 *Child* 中的方法 *f()* 对超类中的方法 *f()* 的重写是错误的，因为子类中方法 *f()* 的返回类型无法向上兼容超类中的方法 *f()* 的返回类型。

<pre>class Parent {     Parent f() {         return new Parent();     } }</pre>	<pre>class Parent {     Child f() {         return new Child();     } }</pre>
<pre>class Child extends Parent {     Child f() {         return new Child();     } }</pre>	<pre>class Child extends Parent {     Parent f() {         return new Parent();     } }</pre>

图 2-7 MiniJOOL 程序示例：重写方法返回类型的兼容性

### 方法的重载 (overloading)

如果一个类的两个方法同名但是基调不同，则称该方法名被重载了。方法重载本身不会导致编译时错误。

需要指出的是，方法的重写是以基调为基础的。方法的基调是在编译时确定的；而由于方法重写，实际被调用的方法将在运行时通过动态查找方法来确定。

图 2-8 中的代码演示了几种合法和非法的方法重写及重载。

```
class Parent {
```

```

    int f1() {return 0;}
    static int f2() {return 2;}
}
class Child extends Parent {
    int f1() {return 8;}    // 正确：重写超类 Parent 中的方法 int f1()
    static int f1() {return 16;}// 错误：类方法不能隐藏实例方法
    int f2() {return 32;}  // 错误：实例方法不能重写类方法
    static int f2() {return 64;}// 正确：Child 中的类方法重写了超类 Parent 中的类方法
    int f1(int i) {return i;}// 正确：重载类 Child 中的方法 int f1()
    String f1(String s) {return s;}// 正确：重载类 Child 中的方法 int f1()。
    String f1(int i) {return "128";}// 错误：基调与 int f1(int i)相同，不构成重载。
    int f2(int i) {return i;}// 正确：重载方法 int f2()。int f2()继承自超类 Parent
}

```

图 2-8 MiniJOOL 程序示例：方法的重写与重载

## 5、构造器声明

```

constructor_declaration = constructor_declarator constructor_body
constructor_declarator  = IDENTIFIER "(" [formal_parameter_list] ")"
constructor_body        = "{" [explicit_constructor_invocation] [block_statements] "}"
explicit_constructor_invocation = ("this" | "super") "(" [argument_list] ")" ";"
argument_list           = expression {"," expression }

```

构造器用来创建一个对象，即一个类的实例。构造器的形参在结构和行为上与方法的形参是一样的。一个构造器的基调由其形参个数和各形参的类型组成。如果一个类中声明了两个具有相同基调的构造器，则将产生编译时错误。

一个构造器体的第一条语句可以是对当前类或其父类中另一个构造器的显式调用（如图 2-9）。如果一个构造器直接或间接调用自身，将产生编译时错误。除了显式构造器调用外，构造器的体与方法体类似。在构造器的体中，不允许出现带表达式的return语句，否则产生编译时错误。

<pre> class Point{     int x, y;     Point(int a, int b) {         x=a; y=b;     }     Point(int a) {         this(a, a); // 显式构造器调用     } } </pre>	<pre> class ColorPoint extends Point{     int color;     ColorPoint (int a, int b, int c) {         super(a, b); // 显式构造器调用         color=c;     } } </pre>
---	---

图 2-9 MiniJOOL 程序示例：构造器声明

### 2.3.3 语句块和语句

程序的执行序列由语句控制。MiniJOOL 语言的语句种类是 C 语言或 Java 语言的语句种类的子集，它含有空语句、表达式语句、if 语句、while 语句、break 语句、continue 语句、return 语句以及复合语句（亦称语句块）。此外，MiniJOOL 语言还提供对 int、boolean 和 String 型表达式的打印语句，以及对具有左值的 int 型表达式的输入语句。

#### 1、语句块

语句块是包含在一对花括号中的、由语句和局部变量声明语句组成的语句序列。

```
block      =  "{" [ block_statements ] "}"
block_statements =  block_statement { block_statement }
block_statement =  local_variable_declaration_statement | statement
```

#### 2、局部变量声明语句

局部变量声明语句声明一个或多个局部变量名。每条局部声明语句只被一个语句块包含；在语句块中，局部变量声明语句与其他语句混杂在一起。局部变量声明语句是可执行语句，当它执行时，自左至右依次处理其所含的各个变量声明符（variable\_declarators）。如果一个变量声明符中有初始化表达式，则对表达式求值并将值赋给相应的变量；如果没有初始化表达式，则 MiniJOOL 编译器必须检查该变量在每次使用前是否被赋值；如果局部变量在使用前未被赋值，则产生编译时错误。

```
local_variable_declaration_statement =  type variable_declarators ";"
variable_declarators =  variable_declarator { "," variable_declarator }
variable_declarator   =  variable_declarator_id [ "=" variable_initializer ]
variable_declarator_id = IDENTIFIER
variable_initializer =  expression | array_initializer
array_initializer    =  "{" [ expression { "," expression } ] "}"
```

语句块中一个局部变量声明的作用域是从该声明开始且到这个语句块为止。在一个方法或构造器内，不能重复声明作用域有重叠的同名局部变量，否则产生编译时错误。但是，可以声明与类中域名同名的局部变量，这时在该局部变量的作用域内，外层的域声明将被隐藏，被隐藏的名字可以通过受限名来访问。

```
1) class Base {
2)   static final int i=5;           // 声明一个 final 类变量 i,初始化为 5
3)   void test(){
4)     int i=4;                     // 声明一个与类变量 i 同名的局部变量 i
5)     int y;                       // 声明一个局部变量 y, 它没有被初始化
6)     { int y=Base.i; }            // 在语句块内声明一个同名的局部变量 y, 这将
                                     // 产生编译时错误;
                                     // 可以通过 Base.i 访问被局部变量 i 隐藏的变量
                                     }
}
```

```

}

```

图 2-10 MiniJOOL 程序示例：同名变量

在图 2-10 中的代码中，第 6 行声明的局部变量 `y` 将引起编译时错误。在第 4~6 行，第 2 行声明的域 `i` 将被第 4 行声明的局部变量 `i` 所隐藏，第 6 行通过受限名 `Base.i` 使用域 `i`。

### 3、语句

MiniJOOL 语言有空语句、表达式语句、`break` 语句、`continue` 语句、`return` 语句、语句块、`if` 语句、`while` 语句以及打印语句和输入语句。除打印语句和输入语句外，它们的含义与 C 和 Java 一样。打印语句和输入语句是为了方便程序与用户的交互而提供的，其中打印语句的表达式类型可以是 `int`、`boolean` 或 `String`，而输入语句的表达式类型是 `int` 且具有左值。

```

statement      =  “;” | statement_expression “;” | “break” “;” | “continue” “;”
                  | “return” [expression] “;” | block | “if” “(” expression “)” statement
                  | “if” “(” expression “)” statement “else” statement
                  | “while” “(” expression “)” statement
                  | “print” “(” expression “)” “;”
                  | “read” “(” lvalue “)” “;”

```

上述 EBNF 文法会引起 `if` 语句的“悬空 `else` 问题” (dangling else problem)。针对这种问题，MiniJOOL 语言规定每个 `else` 语句都与其上最近的 `if` 语句相匹配。图 2-11 中的代码展示了 `if-else` 的就近匹配原则。其中相互匹配的 `if-else` 对使用折线连接在一块。

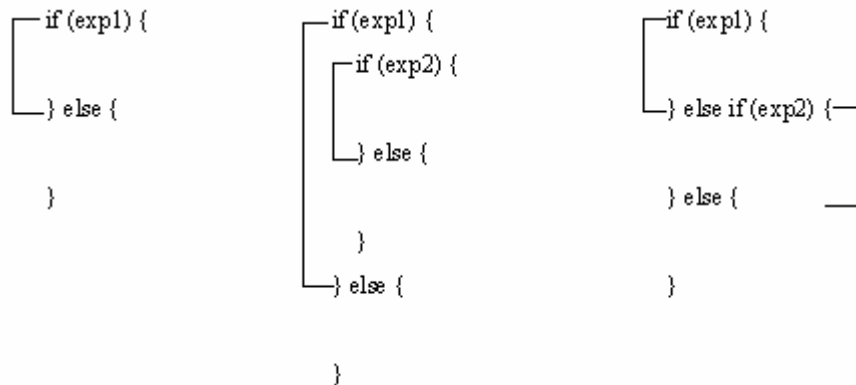


图 2-11 MiniJOOL 程序示例：if-else 的就近匹配

为了解决“悬空 `else` 问题”，可以通过改写文法来保证按 `if-else` 的就近匹配原则来进行语法分析，下面的 EBNF 文法就体现了这种匹配原则。

```

statement      =  statement_without_trailing_substatement
                  | if_then_statement | if_then_else_statement | while_statement
statement_without_trailing_substatement  =  block | “;” | statement_expression “;”
                  | “break” “;” | “continue” “;” | “return” [expression] “;”
                  | “print” “(” expression “)” “;”
                  | “read” “(” lvalue “)” “;”

```

```

if_then_statement = "if" "(" expression ")" statement
if_then_else_statement = "if" "(" expression ")" statement_no_short_if "else" statement
if_then_else_statement_no_short_if = "if" "(" expression ")" statement_no_short_if "else"
statement_no_short_if
statement_no_short_if = statement_without_trailing_substatement
| if_then_else_statement_no_short_if | while_statement_no_short_if
while_statement = "while" "(" expression ")" statement
while_statement_no_short_if = "while" "(" expression ")" statement_no_short_if

```

#### 4、表达式语句

表达式语句由 `statement_expression` 后跟分号组成。可以组成语句的表达式有赋值表达式、方法调用以及类实例创建表达式。

```

statement_expression = assignment_expression | method_invocation
| class_instance_creation_expression

```

### 2.3.4 表达式

#### 1、基本表达式

基本表达式由构成其他表达式的最简单表达式组成，包括常量值、`this`、类实例创建表达式、域访问、方法调用和数组访问。由一对圆括号括起的表达式也视为基本表达式。

```

primary = literal | "this" | "(" expression ")"
| class_instance_creation_expression | field_access | method_invocation
| array_access

```

#### 类实例创建表达式

类实例创建表达式用来创建新的对象。它在运行时的求值次序是：首先为新的类实例分配空间。如果空间不够，则发出“`OutOfMemoryError`”的错误并异常中止；如果分配成功，则新对象将包含在该类及其超类声明的所有域（指实例变量）的新实例。当新的域实例被创建时，域实例被初始化为其缺省值。接着，自左至右计算构造器的实参值，如果其间发生异常，则不再计算右边的实参且异常中止实例创建表达式的求值。实参值计算后则调用相应的构造器。类实例创建表达式的值是对新创建对象的引用。

```

class_instance_creation_expression = "new" class_type "(" [ argument_list ] ")"

```

#### 域访问表达式

```

field_access = ( primary | "super" ) "." IDENTIFIER

```

域访问表达式可以访问一个对象或数组的域、或者超类中的域。当通过 `primary` 进行域访问时，`primary` 的类型必须是引用类型，否则产生编译时错误。

#### 方法调用表达式

```

method_invocation = ( name | ( "super" | primary ) "." IDENTIFIER ) "(" [ argument_list ] ")"

```

方法调用表达式用来调用类方法或者实例方法。由于方法重载，在编译时辨析方法名比域名要复杂得多；同样由于实例方法的重写，在运行时调用一个方法也比访问域要复杂。

编译时处理方法调用的第 1 步是确定要调用的方法名以及相应的类；第 2 步是在所确定的类中查找可用的方法声明（要求方法基调应相符），如果找到多个相符的方法基调，则产生编译时错误。运行时处理方法调用的第 1 步是计算引用目标；第 2 步是计算实参；第 3 步是对实际要调用的方法代码进行定位。

**注意：**当参数为类类型时，实参的类型可以与形参类型一致，或者是形参类型的子类。

### 数组访问表达式

`array_access = (name | field_access) “[” expression “]”`

数组访问表达式将引用一个数组元素变量。数组访问表达式由数组引用表达式和下标表达式组成。数组引用表达式的类型必须是一个数组类型，下标表达式必须是 `int` 型，否则产生编译时错误。此外，需要尽可能地在编译时对下标进行越界检查，如果编译时无法判断，则应在运行时进行越界检查。如果发生越界，则产生编译或运行时错误。

## 2、具有左值的表达式

所有的表达式都具有右值，而只有变量、域访问表达式、数组访问表达式具有左值。由一对圆括号括起的左值表达式也视为左值表达式。

`lvalue = name | field_access | array_access | (“ lvalue “)`

## 3、一般的表达式

一般的表达式由变量名、基本表达式以及各种运算表达式组成。附录 2 列出了 MiniJOOL 语言各种运算符的优先级和结合性。它们的含义与 Java 中的一样，这里不再赘述。需要强调的是，`||`和`&&`运算的求值过程是不完全计算（称为短路计算），即，对表达式 `a && b` 来说，如果 `a` 为 `false`，则无需计算 `b`，即可得到整个表达式的值为 `false`；对表达式 `a || b` 来说，如果 `a` 为 `true`，则无需计算 `b`，即可得到整个表达式的值为 `true`。

```
expression = name | primary
            | unary_operator expression
            | expression binary_operator expression
            | expression “instanceof” class_type
            | assignment_expression

binary_operator = “*” | “/” | “%” | “+” | “-”
                | “==” | “!=” | “<” | “<=” | “>” | “>=” | “||” | “&&”

unary_operator = “+” | “-” | “!”

assignment_expression = lvalue assignment_operator assignment_expression
assignment_operator = “=” | “*=” | “/=” | “%=” | “+=” | “-=”
```

上面对表达式的 EBNF 表示会引起许多表达式求值的二义性。为解决这个问题，可以根据算符的优先级和结合性，利用《编译原理》教材中介绍的方法，改写文法来消除歧义，也可以在 CUP 或 JavaCC 支持的文法描述中用优先权说明语句描述相关终结符的优先级和结合性。

## 2.4 SimpleMiniJOOL 语言简介

SimpleMiniJOOL 语言是 MiniJOOL 语言的简单子集。与 MiniJOOL 程序一样，用这种语言编写的程序的文件扩展名为“mj”。下面列出这种语言的基本特征，其详细的 EBNF 文法定义参见附录 4。

**程序：**每一个 SimpleMiniJOOL 程序有唯一的一个名为 Program 的类，类中有唯一的一个类型为 static void、名为 main 的方法，而没有其它的域和方法。main 方法的方法体不允许出现带表达式的 return 语句。

**数据类型：**SimpleMiniJOOL 语言中只有一种数据类型，即 32 位 int 型。

**常量：**即整型常量，包括十进制整数、以“0”开头的八进制整数、和以“0x”或“0X”开头的十六进制整数。

**变量：**由于 SimpleMiniJOOL 语言只有一种类型，故语言不提供变量声明语句。在 SimpleMiniJOOL 程序中，变量不必声明即可使用。所有的变量都视为全局变量，且缺省值为 0。

**运算：**SimpleMiniJOOL 语言包含 MiniJOOL 语言中的算术运算（正、负、加、减、乘、除、求余）、数值比较运算（>、>=、<、<=、==、!=）、逻辑运算（&&、||、!）和赋值运算（=、+=、-=、\*=、/=、%=）。与 C 语言类似，MiniJOOL 语言的编译器在给出逻辑运算结果时，以数值 1 代表“真”，以 0 代表“假”，但是在判断一个量是否为“真”时，以 0 代表“假”，以非 0 代表“真”。

**表达式：**SimpleMiniJOOL 语言中的表达式包括常量值、变量名、带括号的表达式、算术运算表达式、比较运算表达式、逻辑运算表达式以及赋值表达式。赋值运算左边的表达式必须是变量名。

**语句块和语句：**与 MiniJOOL 语言不同的是，SimpleMiniJOOL 语言没有局部变量声明语句。它的语句包括空语句、表达式语句、if 语句、while 语句、break 语句、continue 语句、return 语句、print 语句和语句块。

图 2-12 给出了一个 SimpleMiniJOOL 程序及其执行结果示例。

<u>一个 SimpleMiniJOOL 程序:</u>	<u>执行结果:</u>
class Program {	10
static void main() {	9
a = 10;	8
while (a > 0) {	7
print(a);	6
a = a - 1;	5
}	4
return;	3
}	2
	1

图 2-12 一个 SimpleMiniJOOL 程序及其执行结果



## 2.5 SkipOOMiniJOOL 语言简介

SkipOOMiniJOOL 语言是去掉 MiniJOOL 语言中的面向对象特征后形成的语言。与 MiniJOOL 程序一样，用 SkipOOMiniJOOL 语言编写的程序的文件扩展名为“mj”。下面列出这种语言的基本特征，其详细的 EBNF 文法定义参见附录 5。

**程序：**一个 SkipOOMiniJOOL 程序只有一个名为 Program 的类，类声明不含 extends 子句，类声明中所有的成员都是 static 的且不含构造器成员，类中有唯一的一个类型为 static void、名为 main 的方法。main 方法的方法体不允许出现带表达式的 return 语句。

**数据类型：**SkipOOMiniJOOL 语言中有 32 位 int 型、boolean 型、String 和一维数组类型。这些类型在含义与使用上与 MiniJOOL 语言中的类型（参见 2.3.1 节）相同。

**常量：**包括整型常量、布尔型常量和字符串常量，其构成特征参见 2.2 节。

**变量：**由于 SkipOOMiniJOOL 程序只有一个类且类中的成员均是 static 的，所以与 MiniJOOL 语言不同的是，SkipOOMiniJOOL 语言没有实例变量、构造器的形参，只有类变量（这里称为全局变量）、方法（称为函数）的形参、局部变量和数组元素。变量名可以是标识符或者是数组访问表达式。全局变量和数组元素在创建时按缺省值初始化。全局变量和局部变量在声明时可以带初始化表达式。全局变量、形参、局部变量的类型可以是 int 型、boolean 型、String 或一维数组类型，数组元素的类型可以是 int 型、boolean 型或 String 变量。有关变量的使用特征与 MiniJOOL 语言类似，参见 2.3.1 节。

**函数：**在 Program 类中声明的方法成员都是 static 的，这些方法也称为函数。函数可以有 0 个或多个参数，参数的类型可以是 int、boolean、String 和一维数组类型，其中数组类型可以指定长度，也可以不指定长度。函数的返回类型可以是 void，也可以是 int 或 boolean，但是不能是 String 或者是数组类型。SkipOOMiniJOOL 语言中的函数参数的传递方式是按值传递。对于 SkipOOMiniJOOL 程序中出现的每一个函数调用，需要在编译时检查这些函数是否在该程序中有定义，并且检查形参与实参的个数是否相等，形参与实参的类型是否兼容，如果有不一致的地方，将产生编译时错误。

**运算：**SkipOOMiniJOOL 语言包含 MiniJOOL 语言中的算术运算（正、负、加、减、乘、除、求余）、数值比较运算（>、>=、<、<=、==、!=）、逻辑运算（&&、||、!）和赋值运算（=、+=、-=、\*=、/=、%=）。SkipOOMiniJOOL 语言在 int 型、boolean 型、String 型以及数组类型上的运算特征与 MiniJOOL 语言相类似。

**表达式：**SkipOOMiniJOOL 语言中的表达式包括常量值、变量名、数组访问表达式、方法调用表达式、带括号的表达式、算术运算表达式、比较运算表达式、逻辑运算表达式以及赋值表达式，不包含域访问表达式以及类实例创建表达式。赋值运算左边的表达式必须是变量名或数组访问表达式。SkipOOMiniJOOL 语言中的名字不含受限名。

**语句块和语句：**SkipMiniJOOL 语言中的语句与 MiniJOOL 语言完全相同，包括局部变量声明语句、空语句、表达式语句、if 语句、while 语句、break 语句、continue 语句、return 语句、print 语句和语句块。

### 2.5.1 一些注意事项

#### 1、变量的种类

- **全局变量**：在Program类中声明的域成员（必须是static），它在执行main前被创建并被初始化，到程序运行结束时消亡。如果SkipOOMiniJOOL程序中的域成员在声明时不含static，则编译器应给出警告信息并将该域视为static。图 2-13 中的代码说明这种情况。

```
class Program {
    int i; // 警告：缺少 static
    static void main() {return;}
}
```

图 2-13 SkipOOMiniJOOL 程序示例：全局变量声明缺少 static

- **局部变量**：在语句块中通过局部变量声明语句来声明的变量。在一个语句块中，对于由直接从属于这个语句块的（不从属于这个语句块中任何内嵌语句块的）局部变量声明语句声明的变量，其生存期从变量的声明点开始直到这个语句块结束为止。
- **方法的形参**：用来命名传递给方法的实参。形参变量在方法被调用时创建，在方法体执行结束时不复存在。
- **数组元素**：是未命名的变量，可以通过数组访问表达式来使用数组元素。数组元素变量随数组变量的创建而被创建和初始化，随数组变量的消亡而结束生命期。

#### 2、final 变量

只有全局变量可以声明为 final。如果 final 变量被赋值多次，将产生编译时错误。声明为 final 的类变量必须在声明时带初始化表达式。

#### 3、程序中的同名问题

SkipOOMiniJOOL 语言中的同名处理遵循以下规则：

- 方法名和变量名可以同名；
- 全局变量名和局部变量名可以同名。当同名的局部变量与同名的全局变量有重叠的作用域时，局部变量将隐藏同名的全局变量；
- 全局变量名和形参可以同名。当同名的形参与同名的全局变量有重叠的作用域时，形参将隐藏同名的全局变量；
- 在一个函数内，不允许重复声明作用域有重叠的同名局部变量，否则产生编译时错误；
- 在一个函数内，不允许与形参同名的局部变量，否则产生编译时错误。

图 2-14 示意了各种同名变量正确与错误使用的情况。

```
class Program {
    static void main() {
        {int j = 8;} //①
        int j = 16; //② 正确：与①中声明的局部变量 j 的作用域不重叠
        {int j = 32;} //③ 错误：与②中声明的局部变量 j 的作用域重叠
    }
}
```

```

    }
    static int i = 2;    // 声明一个全局变量 i
    static int i() {    // 正确：函数名 i 可以与全局变量名同名
        return i;
    }
    static int f() {
        int i = 4;    // 正确：局部变量 i 隐藏了全局变量 i
        return i;    // 返回 4，而不是 2
    }
    static int g(int i) { // 正确：形参 i 隐藏了全局变量 i
        int i = i+1;    // 错误：与形参 i 同名
        return i;    // 返回的是形参 i 的值
    }
}

```

图 2-14 SkipOOMiniJOOL 程序示例：同名变量

#### 4、函数的定义与调用

如前所述，Program 类中的函数必须是 static，如果 SkipOOMiniJOOL 程序中的函数在声明时不含 static，则编译器应给出警告信息并将该函数视为 static。

对于返回类型为 void 的函数，在函数体中不得出现带表达式的 return 语句；对于返回类型不为 void 的函数，在函数体中要保证每一个出口都包含有带表达式的 return 语句，且表达式的类型与返回类型兼容。否则，将产生编译时错误。

对于函数调用，这里只强调参数为数组类型时的处理规则。当参数为数组类型时，在声明时不能指定数组的长度，如果指定了长度，编译器将产生警告信息并忽略这个长度。对于每个函数调用，编译器需要检查实参与形参的类型是否兼容，如果不相同，则产生编译时错误；当参数为数组类型时，编译器只检查实参的元素类型是否与形参的元素类型相同。图 2-15 中的代码演示了这种情况。

```

class Program {
    static void main() {
        int[10] digit = {0,1,2,3,4,5,6,7,8,9};
        int[16] hex = {0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
                       0x8,0x9,0xA,0xB,0xC,0xD,0xE,0xF};
        String[26] letter = {"a","b","c","d","e","f","g","h","i","j","k","l","m","n",
                             "o","p","q","r","s","t","u","v","w","x","y","z"};

        int summary;
        summary = sum(digit);    // 正确
        summary = sum(hex);    // 正确
        summary = sum_fix(digit); // 正确
        summary = sum_fix(hex); // 正确：实参和形参的类型兼容，数组长度被忽略。
        summary = sum(letter);  // 错误：实参和形参的类型不兼容
    }
    static int sum(int[] data) { // sum()可以接受任意长度的整型数组为参数

```

```

        // 省略内部实现
    }
    static int sum_fix(int[10] data) { // 编译器需要产生一个警告，并且忽略数组的长度
        // 省略内部实现
    }
}

```

图 2-15 SkipOOMiniJOOL 程序示例：数组类型的参数

### 2.5.2 一个 SkipOOMiniJOOL 程序示例

图 2-16 中是一个完整的SkipOOMiniJOOL程序。这个程序是图 2-1 中快速排序算法的SkipOOMiniJOOL实现。

```

class Program {
    static void main() {
        int[10] i = {123,52,8,74,62,74,55,44,74,80};
        sort(i);
        printArray(i);
    }
    void sort(int[] data) {
        qsort(data, 0, data.length - 1);
    } // end of sort
    int partition(int[] data, int low, int high) {
        int pivot = data[low];
        int i = low - 1;
        int j = high + 1;
        while (i < j) {
            i++;
            while (data[i] < pivot) i++;
            j--;
            while (data[j] > pivot) j--;
            if (i < j) swap(i, j);
        }
        return j;
    } // end of partition
    void qsort(int[] data, int low, int high) {
        if (low >= high) return;
        int p = partition(data, low, high);
        qsort(data, low, p);
        qsort(data, p + 1, high);
        return;
    }
    void swap(int[] data, int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
        return;
    }
    void printArray(int[] data) {
        int i = 0;
        String str = "\nMember:";
        while (i < data.length) {
            print(str); print(i); print("is ");
            print(data[i]);
        }
    }
} //end of Program

```

图 2-16 SkipOOMiniJOOL 程序示例：快速排序

## 2.6 SkipOOMiniJOOL 语言的静态语义

本节形式地描述 SkipOOMiniJOOL 语言的静态语义，它由一组可以使用的类型、断言

以及推理这些断言的定型规则组成，用来规定良形（well-formed）的 SkipOOMiniJOOL 程序应具有的性质。类型检查将根据这些推理规则在编译时或运行时检查程序的有效性，如果检查通过，则程序是良形的。

### 2.6.1 抽象语法

为便于定义 SkipOOMiniJOOL 语言的静态语义，这里引入语言的抽象语法（abstract syntax），其文法如图 2-17 所示。类型检查将根据静态语义自上而下、自左至右地检查程序。

```

prog    = vardeclist fundeclist fundeflist
vardeclist =  $\varepsilon$  | vardec vardeclist
vardec   = type id ";" | type id "=" initexp ";"
initexp  = exp | "{" aexplist "}"
aexplist = aexp [ ",", aexplist ]
aexp     = exp
fundeclist = fundec ";" [ fundeclist ]
fundec   = type id "(" [ paramlist ] ")"
          | "void" id "(" [ paramlist ] ")"
paramlist = type id { ",", type id }
fundeflist = fundef [ fundeflist ]
fundef    = fundec block
type      = ( "int" | "bool" | "String" ) [ constexp ]
constexp  = exp
block     = "{" [ blkstmts ] "}"
blkstmts = blkstmt [ blkstmts ]
blkstmt   = vardec | stmt
stmt      = ";"
          | lval assignop exp ";"
          | id "(" [ explist ] ")" ";"
          | "break" ";"
          | "continue" ";"
          | "return" [ exp ] ";"
          | block
          | "if" "(" exp ")" stmt
          | "if" "(" exp ")" stmt "else" stmt
          | "while" "(" exp ")" stmt
          | "print" "(" exp ")" ";"
          | "read" "(" lval ")" ";"

lval      = id | id "[" exp "]" | "(" lval ")"
explist   = exp [ ",", explist ]
exp       = lval
          | "(" exp ")"
          | const
          | uop exp
          | exp biop exp

```

$$\begin{aligned}
& | lval \text{ assignop } exp \\
& | id \text{ "(" } [ \text{explist} ] \text{ ")" } \\
\text{assignop} &= \text{"="} | \text{"*="} | \text{"/="} | \text{"\%="} | \text{"+="} | \text{"-="} \\
\text{biop} &= \text{"*"} | \text{" /"} | \text{" \%"} | \text{"+"} | \text{"-"} \\
& | \text{"<"} | \text{"<="} | \text{">="} | \text{">"} | \text{"=="} | \text{"!="} | \text{"\&\&"} | \text{"||"} \\
\text{uop} &= \text{"+"} | \text{"-"} | \text{"!"} \\
\text{const} &= \text{intconst} | \text{strconst} | \text{"true"} | \text{"false"}
\end{aligned}$$

图 2-17 SkipOOMiniJOOL 的抽象文法

抽象语法与具体语法虽然不完全一样，但是基于图 2-17 中的抽象文法定义的静态语义反映 SkipOOMiniJOOL 语言的静态语义。具体语法到抽象语法的变换主要有以下几点：

- 1) 忽略一些由语法直接能检查的符号，如忽略如 “class Program” 的类声明，只包含类体；忽略 “static” 等；
- 2) 将每条声明有  $n$  ( $n>0$ ) 个变量的声明语句按变量自左至右出现的次序拆成  $n$  条声明语句，使得每条声明语句仅声明一个变量。这样拆分的目的是便于定义含有或不含有初始化表达式的变量声明语句的定型规则。
- 3) 将所有全局变量声明按原先的先后次序调整到程序的头部，在全局变量声明之后、所有函数定义之前，增加所有函数的声明。这样做是因为 SkipOOMiniJOOL 语言允许全局变量或函数先使用再定义，而下文的静态语义是遵循变量或函数先定义再使用的原则来定义的。

在依据静态语义对程序进行类型检查时，每遇到一个变量或函数声明就按所声明的类型产生一个符号加到定型环境（在编译器中用符号表来实现定型环境）中；而每遇到对变量或函数的使用则检查其是否在环境中存在，若存在则检查它是否按环境中保存的符号类型来使用，否则程序就不是良形的。

需要强调的是，在实际实现编译器时，并不需要先将 SkipOOMiniJOOL 程序变换成抽象语法所描述的形式，而只需要先对 SkipOOMiniJOOL 程序进行扫描收集所有全局变量和函数的符号，把它们加入到符号表中，然后再对程序按静态语义进行类型检查。

- 4) 改变一些产生式的定义，使得便于写定型规则。
- 5) 引入较简洁的非终结符名。

## 2.6.2 抽象类型、定型环境、断言与定型规则

### 1、抽象类型

SkipOOMiniJOOL 语言的抽象类型的 EBNF 规则定义如下。

$$\begin{aligned}
T^p &= \text{int} | \text{bool} | \text{String} \\
T^b &= T^p | \text{array}[N, T^p] | \text{array}[T^p] \\
T^v &= T^p | \text{array}[N, T^p] \\
T^f &= T^p | \text{array}[T^p] \\
T^d &= T^b | T^b \times T^d
\end{aligned}$$

$$T^a = \text{void} \mid T^d$$

$$T^r = \text{void} \mid \text{int} \mid \text{bool}$$

$$T = \text{unit} \mid \text{void} \mid T^d \mid T^a \rightarrow T^r$$

其中,  $T^b$  为基本类型集合,  $T^v$  为局部变量或全局变量所允许的类型集合,  $T^f$  为形参所允许的类型集合,  $\text{array}[N, T^p]$  表示长度为正整数  $N$  的、元素类型为  $T^p$  的一维数组类型集合,  $\text{array}[T^p]$  表示长度不定、元素类型为  $T^p$  的一维数组类型集合。

**注意:** 如果形参为数组类型的话, 则它只能是  $\text{array}[T^p]$ , 即不指定长度的数组类型。

$T^a$  为参数类型集合, 它可以是代表空积类型的  $\text{void}$  类型, 也可以是由  $n$  ( $n > 0$ ) 个基本类型组成的  $n$  元积类型集合。  $\text{void}$  类型只有一个空元素, 而  $n$  元积类型的元素是  $n$  元元组。  $T^r$  为函数的返回类型集合,  $T^a \rightarrow T^r$  表示参数类型属于  $T^a$ 、返回类型属于  $T^r$  的函数类型集合,  $\text{unit}$  代表语句类型。  $T$  表示 SkipOOMiniJOOL 语言中的所有抽象类型的集合。

在下文中, 符号  $\tau^p$ 、 $\tau^b$ 、 $\tau^v$ 、 $\tau^f$ 、 $\tau^d$ 、 $\tau^a$ 、 $\tau^r$  分别表示属于  $T^p$ 、 $T^b$ 、 $T^v$ 、 $T^f$ 、 $T^d$ 、 $T^a$  或  $T^r$  的任意类型。

## 2、定型环境

类型检查在给定的定型环境 (在编译器中表示为符号表) 中进行。定型环境  $\Gamma$  定义如下:

$$\Gamma = \cdot \mid \Gamma, x : \tau^b \mid \Gamma, f : \tau^a \rightarrow \tau^r \mid \Gamma, \text{rettype} : \tau^r$$

定型环境  $\Gamma$  包含当前作用域中的变量名、函数名、以及当前函数的返回类型 **rettype**, **rettype** 是保留字, “ $\cdot$ ”表示空环境。如果 **rettype** 是  $\text{void}$ , 则函数无返回值。

## 3、断言

SkipOOMiniJOOL 语言的静态语义将由以下断言来描述。

$\Gamma \vdash \circ$	定型环境是良形的
$\Gamma \vdash \tau$	在定型环境 $\Gamma$ 下, 类型 $\tau$ 是良形的
$\Gamma \vdash e : \tau$	在定型环境 $\Gamma$ 下, 表达式 $e$ 的类型为 $\tau$
$\Gamma \vdash S : \text{unit}$	在定型环境 $\Gamma$ 下, 语句 $S$ 是良形的
$\Gamma \vdash \text{rettype} : \tau^r$	在定型环境 $\Gamma$ 下, <b>rettype</b> 的类型为 $\tau^r$

$\Gamma$ 、 $\tau$ 、 $e$ 、 $S$  分别表示定型环境、类型、表达式和语句。一个定型环境  $\Gamma$  是良形的, 是指当  $\Gamma$  出现在断言的左边。在上述断言中, 形如  $\Gamma \vdash M : T$  的断言是**定型断言**, 它表示在定型环境  $\Gamma$  ( $M$  的自由变量都在出现  $\Gamma$  中) 下,  $M$  具有类型  $T$ 。

## 4、推理规则

推理规则是在一组已知的、有效的断言基础上, 声称某个断言的有效性。在本书中采取如下形式来描述:

$$(\text{规则名}) (\text{注释}) \quad \text{推理规则} \quad (\text{注释})$$

推理规则的一般形式是

$$\frac{\Gamma_1 \vdash S_1, \dots, \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

在每条推理规则中, 横线上面是一组称为**前提**的断言, 横线下面是一个称为**结论**的断言。推理规则的含义是, 当所有的前提都得到满足时, 则结论也一定成立。前提个数可以为零, 这样的规则称为**公理**。

针对上述不同的断言形式, 有不同类别的推理规则。例如, 称推理断言  $\Gamma \vdash \circ$  的规则为

环境规则，称推理断言 $\Gamma \vdash \tau$ 的规则为语法规则，称推理形如 $\Gamma \vdash M : T$ 的断言的规则为定型规则。为了保证语言是安全的，SkipOOMiniJOOL 语言将副条件（side condition）引入到传统的定型规则中。这些副条件用来说明在变量值或引用状态（主要指数组类型的形参对实参数组的引用）上的显式需求。为了与一般的定型前提相区分，副条件包含在一对花括号中，写在相应的定型规则的右边。

### 2.6.3 良形的类型以及类型兼容

#### 1、良形的类型

以下规则用来推理 SkipOOMiniJOOL 语言中哪些是良形的类型。TPRODUCT 规则描述函数参数类型的良形性。TFUN 规则描述函数类型的良形性，函数参数类型和返回值类型都可以为 void。

(TINT)	$\frac{}{\Gamma \vdash \text{int}}$
(TBOOL)	$\frac{}{\Gamma \vdash \text{bool}}$
(TSTRING)	$\frac{}{\Gamma \vdash \text{String}}$
(TARRAY1)	$\frac{N \text{ is a positive integer} \quad \Gamma \vdash \tau^p}{\Gamma \vdash \tau^p[N]}$
(TARRAY2)	$\frac{\Gamma \vdash \tau^p}{\Gamma \vdash \tau^p[]}$
(TPRODUCT)	$\frac{\Gamma \vdash \tau^b \quad \Gamma \vdash \tau^d}{\Gamma \vdash \tau^b \times \tau^d}$
(TVOID)	$\frac{}{\Gamma \vdash \text{void}}$
(TFUN)	$\frac{\Gamma \vdash \tau^a \quad \Gamma \vdash \tau^r}{\Gamma \vdash \tau^a \rightarrow \tau^r}$

#### 2、类型兼容

当不同类型的变量用于赋值和参数传递时，需要比较它们的类型兼容性。对任意两个类型 $\tau_1$ 、 $\tau_2$ 来说，如果它们是兼容的，则记为 $\tau_1 \cong \tau_2$ 。以下描述了 SkipOOMiniJOOL 语言的类型兼容规则，这些规则推理形如 $\Gamma \vdash \tau_1 \cong \tau_2$ 的断言。

(TCMP_INT)	$\frac{}{\Gamma \vdash \text{int} \cong \text{int}}$
(TCMP_BOOL)	$\frac{}{\Gamma \vdash \text{bool} \cong \text{bool}}$
(TCMP_STRING)	$\frac{}{\Gamma \vdash \text{string} \cong \text{string}}$
(TCMP_ARRAY1)	$\frac{\Gamma \vdash \tau^p[N_1] \quad \Gamma \vdash \tau^p[N_2] \quad N_1 = N_2}{\Gamma \vdash \tau^p[N_1] \cong \tau^p[N_2]}$



$$(TCMP\_ARRAY2) \quad \frac{\Gamma \vdash \tau_1^p[N] \quad \Gamma \vdash \tau_2^p[] \quad \Gamma \vdash \tau_1^p \cong \tau_2^p}{\Gamma \vdash \tau_1^p[N] \cong \tau_2^p[]}$$

$$(TCMP\_ARRAY3) \quad \frac{}{\Gamma \vdash \tau^p[] \cong \tau^p[]}$$

$$(TCMP\_PRODUCT) \quad \frac{\Gamma \vdash \tau_1^d \cong \tau_2^d \quad \Gamma \vdash \tau_1^b \cong \tau_2^b}{\Gamma \vdash \tau_1^b \times \tau_1^d \cong \tau_2^b \times \tau_2^d}$$

规则(TCMP\_PRODUCT)描述了函数的实参列表与形参列表之间的类型兼容规则。

### 2.6.4 定型环境的扩展

在类型检查期间，定型环境将被扩展，例如将当前作用域中的变量名、函数名、返回类型（用保留字 **rettype** 表示）加入到定型环境中。定型环境的扩展由形如  $\Gamma \vdash \circ$  的断言以及如下推理规则来定义。规则(CEMPTY)指出空环境是良形的定型环境。规则(CADD\_VAR)指出在定型环境  $\Gamma$  上扩展一个不在  $\Gamma$  的论域（用  $dom(\Gamma)$  表示）中的变量  $x$ ， $x$  的类型为  $\tau^b$ 。

$$(CEMPTY) \quad \frac{}{\emptyset \vdash \circ}$$

$$(CADD\_VAR) \quad \frac{\Gamma \vdash \circ \quad \Gamma \vdash \tau^b \quad x \notin dom(\Gamma)}{\Gamma, x : \tau^b \vdash \circ}$$

规则(CADD\_RETTYPE)指出当 **rettype** 不在  $\Gamma$  中时，在  $\Gamma$  上扩展 **rettype**:  $\tau^r$  所形成的环境是良形的。

$$(CADD\_RETTYPE) \quad \frac{\Gamma \vdash \circ \quad \Gamma \vdash \tau^r \quad \text{rettype} \notin dom(\Gamma)}{\Gamma, \text{rettype} : \tau^r \vdash \circ}$$

规则(CADD\_MAIN)是向  $\Gamma$  扩展 **main** 函数的规则，它与扩展其他函数的规则(CADD\_FUN)有些不同，因为其参数和返回值类型都必须为 **void**。对于带参的函数，其每个形参类型是  $\tau^f$ ，即对于形参是数组类型时必须为  $array[\tau^p]$ ，这将在函数声明与定义（2.6.7 节）中介绍。

$$(CADD\_MAIN) \quad \frac{\Gamma \vdash \circ \quad \text{main} \notin dom(\Gamma)}{\Gamma, \text{main} : \text{void} \rightarrow \text{void} \vdash \circ}$$

$$(CADD\_FUN) \quad \frac{\Gamma \vdash \circ \quad \Gamma \vdash \tau^a \rightarrow \tau^r \quad f \neq \text{main} \quad f \notin dom(\Gamma)}{\Gamma, f : \tau^a \rightarrow \tau^r \vdash \circ}$$

### 2.6.5 表达式

为了对表达式  $e$  进行类型检查，必须指出其子表达式是有效的，并且由它们形成的表达式也是有效的。表达式的有效性由定型断言  $\Gamma \vdash e : \tau$  和推理这个断言的定型规则来定义。

#### 1、常量

常量包括布尔值 **true** 和 **false**、整数以及字符串。下面的规则是显然的。

$$(EXP\_TRUE) \quad \frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$(EXP\_FALSE) \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$(EXP\_INT) \quad \frac{}{\Gamma \vdash \text{intconst} : \text{int}}$$

$$\text{(EXP\_STRING)} \quad \frac{}{\Gamma \vdash \text{strconst} : \text{String}}$$

## 2、带括号的表达式

$$\text{(EXP\_BRACKET)} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau}$$

## 3、左值

在 SkipOOMiniJOOL 语言中，所有的表达式都具有右值，而只有变量和数组访问表达式具有左值。赋值运算要求左边的表达式应具有左值。为了便于给出赋值表达式的定型规则，我们引入辅助断言  $\Gamma \vdash e \text{ lval}$  用来断言表达式  $e$  在定型环境  $\Gamma$  下具有左值。规则(LVAL\_VAR)指出在  $\Gamma$  上扩展变量  $x : \tau^b$ ，可以推导出  $x$  是左值。规则(LVAL\_ARRAY1)和(LVAL\_ARRAY2)都是用来推理数组访问表达式是左值的规则。前者针对数组变量  $l$  在声明时指定了长度，这时要求下标表达式  $e$  的值必须是 0 和  $N-1$  之间的整数，由于  $e$  的值不一定能静态确定，故把对  $e$  的约束作为规则的副条件；后者是针对  $l$  在声明时（即充当形参）没有指定长度的情况，这时对下标表达式的约束由传入的实参数组长度  $l.length$  来限定。规则(LVAL\_BRACKET)指出若  $l$  是左值，则带括号的  $l$  也是左值。

$$\begin{aligned} \text{(LVAL\_VAR)} & \quad \frac{}{\Gamma, x : \tau^b \vdash x \text{ lval}} \\ \text{(LVAL\_ARRAY1)} & \quad \frac{\Gamma \vdash l : \tau^p[N] \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l[e] \text{ lval}} \{0 \leq e < N\} \\ \text{(LVAL\_ARRAY2)} & \quad \frac{\Gamma \vdash l : \tau^p[] \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l[e] \text{ lval}} \{0 \leq e < l.length\} \\ \text{(LVAL\_BRACKET)} & \quad \frac{\Gamma \vdash l \text{ lval}}{\Gamma \vdash (l) \text{ lval}} \end{aligned}$$

以下三条是不带括号的左值表达式的定型规则，而带括号的定型规则见上面的 (EXP\_BRACKET)。

$$\begin{aligned} \text{(EXP\_VAR)} & \quad \frac{}{\Gamma, x : \tau^b \vdash x : \tau^b} \\ \text{(EXP\_ARRAY1)} & \quad \frac{\Gamma \vdash l : \tau^p[N] \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l[e] : \tau^p} \{0 \leq e < N\} \\ \text{(EXP\_ARRAY2)} & \quad \frac{\Gamma \vdash l : \tau^p[] \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l[e] : \tau^p} \{0 \leq e < l.length\} \end{aligned}$$

## 4、双目运算

SkipOOMiniJOOL 语言的双目运算分以下三大类：在 `int` 型值上的算术运算，其表达式的定型规则由 (EXP\_INT\_ARITH) 给出；在 `int` 型值上的比较运算，其定型规则由 (EXP\_INT\_COMP) 给出；在 `bool` 值上的运算，其定型规则由 (EXP\_BOOL\_ARITH) 给出。

$$\text{(EXP\_INT\_ARITH)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ biop } e_2 : \text{int}}$$

其中  $\text{biop} = \text{"+"} \mid \text{"-"} \mid \text{"*"} \mid \text{" /"} \mid \text{"\%"} \mid \text{"<"} \mid \text{">"} \mid \text{"<="} \mid \text{">="}$

$$\text{(EXP\_INT\_COMP)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ biop } e_2 : \text{bool}}$$

其中  $biop = "<" \mid "<=" \mid ">=" \mid ">" \mid "==" \mid "!="$

$$(EXP\_BOOL\_ARITH) \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ biop } e_2 : \text{bool}}$$

其中  $biop = "&\&" \mid "||" \mid "==" \mid "!="$

## 5、单目运算

SkipOOMiniJOOL 语言的单目运算分两大类：在 `int` 型值上的一元运算，由规则 (EXP\_INT\_UARITH) 给出；在 `bool` 值上的一元运算，由规则 (EXP\_BOOL\_NOT) 给出。

$$(EXP\_INT\_UARITH) \quad \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash uop \ e : \text{int}}$$

其中  $uop = "+" \mid "-"$

$$(EXP\_BOOL\_NOT) \quad \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash !e : \text{bool}}$$

## 6、实参列表与方法调用

规则 (EXPLIST\_SEQ) 描述了实参列表的定型规则。实参的类型可以为  $\tau^p[]$ ，这时实参是当前函数的形参。规则 (EXP\_FUN\_PARAM) 和 (EXP\_FUN\_NOPARAM) 分别给出了带参的函数调用以及不带参的函数调用的定型规则。

$$(EXPLIST\_SEQ) \quad \frac{\Gamma \vdash exp : \tau^b \quad \Gamma \vdash explist : \tau^d}{\Gamma \vdash exp, explist : \tau^b \times \tau^d}$$

$$(EXP\_FUN\_PARAM) \quad \frac{\Gamma \vdash f : \tau_1^d \rightarrow \tau^r \quad \Gamma \vdash explist : \tau_2^d \quad \Gamma \vdash \tau_1^d \cong \tau_2^d}{\Gamma \vdash f(explist) : \tau^r}$$

$$(EXP\_FUN\_NOPARAM) \quad \frac{\Gamma \vdash f : \text{void} \rightarrow \tau^r}{\Gamma \vdash f() : \tau^r}$$

## 7、赋值

赋值表达式的定型规则有如下两条。它们规定赋值运算符左边的表达式必须是左值，而且左右两边的表达式必须有相同的类型。

$$(EXP\_INT\_ASSIGN) \quad \frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash l \text{ lval} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l \text{ assignop } e : \text{int}}$$

其中  $assignop = "=" \mid "+=" \mid "-=" \mid "*=" \mid "/=" \mid "\%="$

$$(EXP\_OTHER\_ASSIGN) \quad \frac{\Gamma \vdash l : \tau \quad \Gamma \vdash l \text{ lval} \quad \Gamma \vdash e : \tau \quad \tau = \text{bool} \mid \text{string}}{\Gamma \vdash l = e : \tau}$$

### 2.6.6 语句和语句块

语句和语句块的有效性由断言  $\Gamma \vdash S : \text{unit}$  以及推理这个断言的推理规则定义。下面分别给出各种语句的定型规则。

#### 1、表达式语句

表达式语句中的表达式只允许是赋值表达式或函数调用。它们的定型规则如下：

$$(STMT\_ASSIGN) \quad \frac{\Gamma \vdash l \text{ assignop } e : \tau^p}{\Gamma \vdash l \text{ assignop } e ; : \text{unit}}$$

$$\begin{array}{c}
 (\text{STMT\_FUN\_PARAM}) \quad \frac{\Gamma \vdash f(\text{explist}) : \tau^r}{\Gamma \vdash f(\text{explist}); \text{unit}} \\
 (\text{STMT\_FUN\_NOPARAM}) \quad \frac{\Gamma \vdash f() : \tau^r}{\Gamma \vdash f(); \text{unit}}
 \end{array}$$

## 2、分支和循环语句

$$\begin{array}{c}
 (\text{STMT\_IF}) \quad \frac{\Gamma \vdash \text{exp} : \text{bool} \quad \Gamma \vdash \text{stmt} : \text{unit}}{\Gamma \vdash \text{if}(\text{exp}) \text{ stmt} : \text{unit}} \\
 (\text{STMT\_IF\_ELSE}) \quad \frac{\Gamma \vdash \text{exp} : \text{bool} \quad \Gamma \vdash \text{stmt}_1 : \text{unit} \quad \Gamma \vdash \text{stmt}_2 : \text{unit}}{\Gamma \vdash \text{if}(\text{exp}) \text{ stmt}_1 \text{ else } \text{stmt}_2 : \text{unit}} \\
 (\text{STMT\_WHILE}) \quad \frac{\Gamma \vdash \text{exp} : \text{bool} \quad \Gamma \vdash \text{stmt} : \text{unit}}{\Gamma \vdash \text{while}(\text{exp}) \text{ stmt} : \text{unit}}
 \end{array}$$

## 3、return 语句

return 语句的定型规则主要是保证它与当前定义的函数的返回类型（保存在 **rettype** 中）相兼容。对于带表达式的 return 语句，其定型规则为(STMT\_RETVAL)，它要求表达式的类型与 **rettype** 的类型一样；而对于不带表达式的 return 语句，其定型规则为(STMT\_RET)，要求 **rettype** 的类型必须是 void。

$$\begin{array}{c}
 (\text{STMT\_RETVAl}) \quad \frac{\Gamma \vdash e : \tau^r \quad \Gamma \vdash \text{rettype} : \tau^r}{\Gamma \vdash \text{return } e; \text{unit}} \\
 (\text{STMT\_RET}) \quad \frac{\Gamma \vdash \text{rettype} : \text{void}}{\Gamma \vdash \text{return}; \text{unit}}
 \end{array}$$

## 4、其他简单语句

$$\begin{array}{c}
 (\text{STMT\_EMPTY}) \quad \frac{}{\Gamma \vdash ; : \text{unit}} \\
 (\text{STMT\_BREAK}) \quad \frac{}{\Gamma \vdash \text{break}; \text{unit}} \\
 (\text{STMT\_CONT}) \quad \frac{}{\Gamma \vdash \text{continue}; \text{unit}} \\
 (\text{STMT\_PRINT}) \quad \frac{\Gamma \vdash e : \tau^p}{\Gamma \vdash \text{print}(e); \text{unit}} \\
 (\text{STMT\_READ}) \quad \frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash l \text{ lval}}{\Gamma \vdash \text{read}(l); \text{unit}}
 \end{array}$$

## 5、复合语句块

语句块是一个由花括号包含的、0 个或多个局部变量声明语句 *vdec* 或上述 *stmt* 语句组成的序列。局部变量声明语句的推理规则主要在于检查局部变量的类型是否声明成  $\tau^p$  或者  $\tau^p[N]$ ，或者声明成  $\tau^p[]$  且带有数组初始化表达式，此时由初始化表达式中的表达式个数决定所声明的数组变量的长度；如果局部变量在声明时指定了初始化表达式，还要检查局部变量的声明类型是否与初始化表达式的类型兼容；当类型检查通过时，为声明的局部变量产生一个变量名扩展到定型环境中，即 2.6.4 节中的(CADD\_VAR)。

在引入语句块的定型规则前，先考虑初始化表达式 *initexp* 的定型规则，*exp* 的定型规则见 2.3.4 节。这里增加三条对数组初始化表达式的规则，这些规则将花括号中的表达式类型

均限制为  $\tau^p$ , 并且对其中的表达式个数进行计数, 最终将由花括号包含的表达式定型为  $\tau^p[N]$  类型,  $N$  为正整数。

$$\begin{array}{c}
 \text{(VEXP\_AEXP)} \quad \frac{\Gamma \vdash aexp : \tau^p}{\Gamma \vdash aexp : \tau^p[1]} \\
 \text{(VEXP\_AEXPLIST)} \quad \frac{\Gamma \vdash aexp : \tau^p \quad \Gamma \vdash aexplist : \tau^p[n] \quad n \text{ is a positive integer}}{\Gamma \vdash aexplist, aexp : \tau^p[n+1]} \\
 \text{(VEXP\_INIT\_ARRAY)} \quad \frac{\Gamma \vdash aexplist : \tau^p[n] \quad n \text{ is a positive integer}}{\Gamma \vdash \{aexplist\} : \tau^p[n]}
 \end{array}$$

以下是针对语句块花括号中的不同情况给出的定型规则。规则(BSTMTS\_EMPTY)指出空语句序列是良形的。规则(BSTMTS\_VDEC\_NOINIT\_SEQ)和(BSTMTS\_VDEC\_INIT\_SEQ)定义以局部变量声明语句开头的语句序列的良形性, 当局部变量声明有效时, 在定型环境  $\Gamma$  中将扩展这个局部变量作为后续语句序列  $blkstmts$  的定型环境。两条规则中, 前者针对声明时不带初始化表达式的情况, 后者针对带初始化表达式的情况。对于前者, 局部变量必须声明为任意的  $\tau^v$ 。对于后者, 局部变量可以声明为任意的  $\tau^b$  (即允许声明为  $\tau^p[]$ ), 但同时必须与初始化表达式的类型  $\tau^v$  相兼容; 若兼容则以初始化表达式的类型  $\tau^v$  作为该变量的类型加到定型环境中。规则(BSTMTS\_STMT\_SEQ)定义以  $stmt$  开头的语句序列的良形性。

$$\begin{array}{c}
 \text{(BSTMTS\_EMPTY)} \quad \frac{}{\Gamma \vdash \varepsilon : \text{unit}} \\
 \text{(BSTMTS\_VDEC\_NOINIT\_SEQ)} \quad \frac{\Gamma, x : \tau^v \vdash \circ \quad \Gamma, x : \tau^v \vdash blkstmts : \text{unit}}{\Gamma \vdash \tau^v x; blkstmts : \text{unit}} \\
 \text{(BSTMTS\_VDEC\_INIT\_SEQ)} \quad \frac{\Gamma, x : \tau^v \vdash \circ \quad \Gamma \vdash e : \tau^v \quad \Gamma \vdash \tau^v \cong \tau^b \quad \Gamma, x : \tau^v \vdash blkstmts : \text{unit}}{\Gamma \vdash \tau^b x = e; blkstmts : \text{unit}} \\
 \text{(BSTMTS\_STMT\_SEQ)} \quad \frac{\Gamma \vdash stmt : \text{unit} \quad \Gamma \vdash blkstmts : \text{unit}}{\Gamma \vdash stmt blkstmts : \text{unit}}
 \end{array}$$

由语句序列的良形性, 就可以得到复合语句块的良形性, 即定型规则(STMT\_BLOCK)。

$$\text{(STMT\_BLOCK)} \quad \frac{\Gamma \vdash blkstmts : \text{unit}}{\Gamma \vdash \{blkstmts\} : \text{unit}}$$

### 2.6.7 函数的定义

在所有函数定义前增加声明这些函数的结构, 是为了在推理函数定义时定型环境中已含有所有的函数名及其类型。关于函数声明语句的处理见 2.6.8 节。

一个函数定义包括返回类型、函数名、参数列表和函数体。规则(FDEF\_PARAM)是带参函数定义的定型规则, 其前提有 3 个: 1) 函数名  $f$  在  $\Gamma$  中; 2) 对  $\Gamma$  扩展各个形参和函数的返回类型所形成的是良形的定型环境, 3) 在扩展后的定型环境下可以推出函数体  $block$  是良形的; 其结论为在  $\Gamma$  下函数定义是良形的。规则(FDEF\_NOPARAM)是无参函数定义的定型规则。

$$\begin{array}{c}
 \Gamma \vdash f : \tau_1^f \times \tau_2^f \times \dots \times \tau_n^f \rightarrow \tau^r \\
 \Gamma, p_1 : \tau_1^f, p_2 : \tau_2^f, \dots, p_n : \tau_n^f, \text{rettype} : \tau^r \vdash \circ \\
 \text{(FDEF\_PARAM)} \quad \frac{\Gamma, p_1 : \tau_1^f, p_2 : \tau_2^f, \dots, p_n : \tau_n^f, \text{rettype} : \tau^r \vdash \text{block} : \text{unit}}{\Gamma \vdash \tau^r f(\tau_1^f p_1, \tau_2^f p_2, \dots, \tau_n^f p_n) \text{ block} : \text{unit}} \\
 \\
 \Gamma \vdash f : \text{void} \rightarrow \tau^r \quad \Gamma, \text{rettype} : \tau^r \vdash \circ \\
 \text{(FDEF\_NOPARAM)} \quad \frac{\Gamma, \text{rettype} : \tau^r \vdash \text{block} : \text{unit}}{\Gamma \vdash \tau^r f() \text{ block} : \text{unit}}
 \end{array}$$

注意：参数类型是任意的  $\tau^f$ ，即参数类型不能是  $\tau^p[N]$ 。

### 2.6.8 程序

程序中有 0 个或多个全局变量声明，后跟 1 个或多个函数声明，后跟 1 个或多个函数定义。下面三条定型规则分别针对由不带初始化的全局变量声明语句开头的序列、由带初始化的全局变量声明语句开头的序列以及没有 *vardeclist* 时的良形性：

$$\begin{array}{c}
 \Gamma, x : \tau^v \vdash \circ \\
 \text{(PROG\_VDECS\_VDEC1)} \quad \frac{\Gamma, x : \tau^v \vdash \text{vardeclist fundeclist fundeflist} : \text{unit}}{\Gamma \vdash \tau^v x ; \text{vardeclist fundeclist fundeflist} : \text{unit}} \\
 \\
 \Gamma, x : \tau^v \vdash \circ \quad \Gamma \vdash e : \tau^v \quad \Gamma \vdash \tau^b \cong \tau^v \\
 \text{(PROG\_VDECS\_VDEC2)} \quad \frac{\Gamma, x : \tau^v \vdash \text{vardeclist fundeclist fundeflist} : \text{unit}}{\Gamma \vdash \tau^b x = e ; \text{vardeclist fundeclist fundeflist} : \text{unit}} \\
 \\
 \Gamma \vdash \text{fundeclist fundeflist} : \text{unit} \\
 \text{(PROG\_VDECS\_EMPTY)} \quad \frac{\Gamma \vdash \text{fundeclist fundeflist} : \text{unit}}{\Gamma \vdash \varepsilon \text{ fundeclist fundeflist} : \text{unit}}
 \end{array}$$

对函数声明语句的定型，主要包括对形参列表的定型以及在定型环境中扩展函数的规则（见 2.6.4 节）。下面两条是对形参列表的定型规则。

$$\begin{array}{c}
 \text{(FDEC\_PARAM)} \quad \frac{}{\Gamma \vdash \tau^f x : \tau^f} \\
 \\
 \text{(FDEC\_PARAM\_SEQ)} \quad \frac{\Gamma \vdash \tau^f x : \tau^f \quad \Gamma \vdash \text{typeidlist} : \tau^d}{\Gamma \vdash \tau^f x, \text{typeidlist} : \tau^f \times \tau^d}
 \end{array}$$

下面两条规则针对由带参函数声明语句开头的序列的良形性。规则 (PROG\_FDEC1) 指出：若函数声明语句序列的最后一条规则是带参函数声明，则定型环境  $\Gamma$  中必须含有类型为  $\text{void} \rightarrow \text{void}$  的主函数 *main*。

$$\begin{array}{c}
 \Gamma, f : \tau^d \rightarrow \tau^r \vdash \circ \quad \Gamma \vdash \text{typeidlist} : \tau^d \\
 \text{(PROG\_FDECS\_FDEC1)} \quad \frac{\Gamma, f : \tau^d \rightarrow \tau^r \vdash \text{fundeclist fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f(\text{typeidlist}); \text{fundeclist fundeflist} : \text{unit}} \\
 \\
 \Gamma, f : \tau^d \rightarrow \tau^r \vdash \circ \quad \Gamma \vdash \text{typeidlist} : \tau^d \\
 \text{(PROG\_FDEC1)} \quad \frac{\Gamma \vdash \text{main} : \text{void} \rightarrow \text{void} \quad \Gamma, f : \tau^d \rightarrow \tau^r \vdash \text{fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f(\text{typeidlist}); \text{fundeflist} : \text{unit}}
 \end{array}$$

下面两条规则针对由无参函数声明语句开头的序列的良形性。规则(Prog\_FDEC2)指出：若函数声明语句序列的最后一条规则是无参函数声明，则或者  $f$  是类型为  $\text{void} \rightarrow \text{void}$  的主函数  $\text{main}$ ，或者定型环境  $\Gamma$  中含有类型为  $\text{void} \rightarrow \text{void}$  的主函数  $\text{main}$ 。

$$\text{(PROG\_FDECS\_FDEC2)} \quad \frac{\Gamma, f : \text{void} \rightarrow \tau^r \vdash \circ \quad \Gamma, f : \text{void} \rightarrow \tau^r \vdash \text{fundeclist fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f(); \text{fundeclist fundeflist} : \text{unit}}$$

$$\text{(PROG\_FDEC2)} \quad \frac{\Gamma, f : \text{void} \rightarrow \tau^r \vdash \circ \quad \Gamma, f : \text{void} \rightarrow \tau^r \vdash \text{main} : \text{void} \rightarrow \text{void} \quad \Gamma, f : \text{void} \rightarrow \tau^r \vdash \text{fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f(); \text{fundeflist} : \text{unit}}$$

上述四条规则保证了良形程序中的函数声明语句序列不为空。下面两条规则分别定义针对由带参函数定义开头的序列、由无参函数定义开头的序列。

$$\text{(PROG\_FDEFS\_FDEF1)} \quad \frac{\Gamma \vdash \tau^r f(\tau_1^f p_1, \tau_2^f p_2, \dots, \tau_n^f p_n) \text{ block} : \text{unit} \quad \Gamma \vdash \text{fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f(\tau_1^f p_1, \tau_2^f p_2, \dots, \tau_n^f p_n) \text{ block fundeflist} : \text{unit}}$$

$$\text{(PROG\_FDEFS\_FDEF2)} \quad \frac{\Gamma \vdash \tau^r f() \text{ block} : \text{unit} \quad \Gamma \vdash \text{fundeflist} : \text{unit}}{\Gamma \vdash \tau^r f() \text{ block fundeflist} : \text{unit}}$$

规则(Prog\_WF)描述程序的良形性。

$$\text{(PROG\_WF)} \quad \frac{\emptyset_\Gamma \vdash \text{vardeclist fundeclist fundeflist} : \text{unit}}{\emptyset_\Gamma \vdash \text{program} : \text{unit}}$$

## 第3章 一个简单的程序解释器

在本章中，你将学习为 MiniJOOl 语言的一个简单子集 SimpleMiniJOOl 实现一个解释器。实现这个解释器并不要求你有很深入的编译原理知识，而只需要有一定的程序设计基础和编程经验，再加上本章中介绍的一些知识就可以顺利完成。通过实现这个解释器，可以使你对编译技术有大致地了解，增进对编译课程的学习兴趣，并且熟悉课程实验所需要的语言、工具及技术，为后续的实验打下基础。

在下面各节中，我们首先简要介绍本书所配套的实验软件包结构，然后给出本章中课程设计的任务要求，接下来围绕课程设计给出一些指导，包括实验平台的使用、测试环境的搭建、抽象语法树和设计模式的介绍等。

### 3.1 实验软件包的结构

为配合本书的各个课程设计，我们提供实验平台软件以及各个课程设计的框架软件。这些软件包由以下目录组成：

- **lab 目录**

存放各个课程设计的框架软件。在这个目录下，各个课程设计的软件包将分子目录存放。

- **lab1 目录**：有关 SimpleMiniJOOl 解释器的课程设计，参见本章。
- **lab2 目录**：有关 MiniJOOl 语言词法分析的课程设计，参见第 4 章。
- **lab3 目录**：有关 SkipOOMiniJOOl 语言语法分析的课程设计，参见第 5 章。
- **lab4 目录**：有关 SkipOOMiniJOOl 语言语义分析的课程设计，参见第 6 章。
- **lab5 目录**：有关 SkipOOMiniJOOl 语言 MIPS 汇编代码生成的课程设计，参见第 7 章。
- **lab6 目录**：有关 SkipOOMiniJOOl 语言 x86 汇编代码生成的课程设计，参见第 8 章。

- **lib 目录**

存放各个课程设计所依赖的 AST、低级中间表示等的类库文件。

- **AST 目录**

分 Eclipse 版本存放与 Eclipse JDT AST 有关的类库文件。

- **edu.ustc.cs.compile.parser.simpleminijool.jar**：实现 SimpleMiniJOOl 语言分析器的类库文件，在课程设计 1 中需要使用。
- **edu.ustc.cs.compile.parser.skipoominijool.jar**：实现 SkipOOMiniJOOl 语言分析器的类库文件。
- **edu.ustc.cs.compile.parser.mlexspec.jar**：实现 MLex 词法规范描述语言分析器的类库文件，在课程设计 2-3 和 2-4 中需要使用。
- **edu.ustc.cs.compile.arch.jar**：汇编语言内部表示及寄存器分配等的类库文件，



供 x86 或 MIPS 汇编代码生成使用。

- **platform 目录**

存放实验平台软件包，包括实验平台接口、实验运行平台等。

- **lib 目录**

- ◆ **edu.ustc.cs.compile.platform.core.jar**: 实验平台运行时需要的类库文件，实验运行平台的主类 `edu.ustc.cs.compile.platform.core.Main` 打包在这个类库中。

- ◆ **edu.ustc.cs.compile.platform.interfaces.jar**: 提供给各个课程设计的实验平台接口，例如解释器接口 `InterpreterInterface`、语法分析器接口 `ParserInterface`、解释器异常类 `InterpreterException`，等等。

- ◆ **edu.ustc.cs.compile.platform.util.jar**: 提供帮助你完成实验的辅助工具，例如以图形化方式显示AST的ASTView包（见 3.5.6 节）。

- **config 目录**

- ◆ **configure.xml**: 实验平台的配置文件，虽然不是必需的，但是合理使用可以帮助你减少一些工作量。

- ◆ **configure.xsd**: 定义实验平台配置文件的 XML Schema 文件，请不要对它做任何修改。

- **test 目录**

存放用于测试 SimpleMiniJOOL 语言、SkipOOMiniJOOL 语言以及 MiniJOOL 语言编译器的各种测试程序。

- **tools 目录**

存放 JFlex、CUP、JavaCC 等工具包。

- **doc 目录**

存放实验平台等的应用编程接口文档。

在开展下面各个课程设计之前，你需要把上述的 lab、lib、platform 和 tools 目录复制到你本机的指定工作目录下，记为 ROOT\_DIR。

## 3.2 课程设计 1: 一个简单的程序解释器

本课程设计是本书所有课程设计中 simplest 的一个，旨在让你发现实现一个解释器并不是一件很困难的事情，从而使你产生对编译原理和技术的学习兴趣。

在本课程设计中，你将为 SimpleMiniJOOL 语言实现一个解释器。这个解释器接收一个 SimpleMiniJOOL 程序的中间表示（即抽象语法树 AST），通过对这个 AST 进行遍历完成对 SimpleMiniJOOL 程序的解释执行。为了提供对一个 SimpleMiniJOOL 源程序的解释执行，你可以使用我们提供的词法和语法分析模块产生 AST，也可以手工为这个程序编写一个构造 AST 的 Java 类，然后通过实验平台将产生 AST 的模块与自己编写的解释器模块结合起来。由于语言比较简单，你在设计实现这种语言的解释器时，不必考虑复杂的类型系统、符号表

等，而只需要利用已有的编程知识就可以了。

与课程设计 1 相关的文件主要集中在本书提供的软件包的目录 `ROOT_DIR/lab/lab1` 中，该目录包括如下内容：

- **README:** 本课程设计的说明文件。
- **.project 文件和.classpath 文件:** 本课程设计的 Eclipse 工程文件。
- **bin 目录**
  - **build.xml**  
lab1 的 ant 编译文件。在当前目录下执行 ant 或 ant build 时，会执行这个编译文件中的 build 任务组，它将执行 javac 编译../src 目录下的所有文件，并将所生成的 class 文件存放在../classes 目录下。若执行 ant doc，则会执行编译文件中的 doc 任务组，它将执行 javadoc 为../src 目录下的所有源文件生成 html 文档存放在../doc 目录下
  - **run.bat 和 run.sh**  
在当前目录下调用实验平台的脚本文件。这两个文件分别为 MS Windows 和 Linux 下的脚本。
- **src 目录**  
存放你实验用到的一些 Java 源文件，你需要对其中的部分源文件补充代码。这些 Java 源文件的包路径均为 edu.ustc.cs.compile.interpreter，具体包括：
  - **Interpreter.java:** 解释器 Interpreter 类的框架代码，它是编译实验平台提供的解释器接口类 InterpreterInterface 的实现类。InterpreterInterface 接口要求它的实现类必须实现方法 `public void interpret(InterRepresent ir)`。这个方法通过参数 ir 提供待解释执行的一个 SimpleMiniJOOOL 程序的中间表示。
  - **TestCase.java:** 手工构建中间表示的 TestCase 类的框架代码。TestCase 类中的方法 `public CompilationUnit createAST( )`用于手工构建 AST。TestCase 类是实验平台提供的词法语法分析接口类 ParserInterface 的实现类。TestCase 类中的方法 `public CompilationUnit doParse(FileReader src)`对文件 src 不做任何处理，直接调用 `createAST( )`并返回其结果。
  - **2 个异常类的 java 文件:**即表示除零异常的 DivideByZeroException.java 和 void 引用异常的 VoidReferenceException.java。这 2 个类均派生自实验平台提供的解释器异常接口类 InterpreterException。在本课程设计中，你不必对这些类进行修改，但是你需要在 Interpreter 类中使用这些异常类来处理解释执行中遇到的错误。
  - **Main.java:** 为便于你调试本课程设计所引入的主类。你可以修改这个类中 main 方法的代码，通过 fromParser 的值来控制是由 SimpleMiniJOOOL 分析器（在类库 `ROOT_DIR/lib/edu.ustc.cs.compile.parser.simpleminijool.jar` 中）还是由 TestCase 构造 AST，更改输入程序的文件名；通过 viewAST 控制是否调用 ASTViewer 显示 AST。

- **config 目录**

存放本实验所需的实验平台的配置文件。

- **lab1-parser.xml**: 该配置文件将指定用我们提供的 SimpleMiniJOOL 分析器类 edu.ustc.cs.compile.parser.simpleminijool.Parser 生成 AST, 再启动 Interpreter 对这个 AST 解释执行。

- **lab1-testcase.xml**: 该配置文件将指定用 TestCase 类手工构造 AST, 再启动 Interpreter 对这个 AST 解释执行。

- **test 目录**

收集一些测试你所编写的解释器是否有效的 SimpleMiniJOOL 程序。你可以在这个目录下, 补充更多的测试用例。

### 课程设计任务

你可以为 ROOT\_DIR/lab/lab1 中的代码建立 Eclipse 工程 (按 1.4.3.3 节介绍的方法), 以便利用 Eclipse 进行编译和调试; 你也可以直接使用 ant 或者 jdk 来进行编译和运行。在 3.4 节中给出了采用这两种方法的简要开发指南。你需要对 lab1 所提供的框架代码做如下工作:

- 在 Interpreter.java 中提供的框架补全实现 SimpleMiniJOOL 解释器的代码;
- 在 TestCase.java 中定义构建三个或更多的合法 SimpleMiniJOOL 程序的 AST 的方法, 如 createAST(); 然后在 doParse 方法中, 将 AST 构建方法作为参数传给 ir.setIR 方法, 如 ir.setIR(createAST());
- 编写 3 个以上的合法 SimpleMiniJOOL 源程序代码;
- 阅读 3.4 节给出的开发和测试指南, 分别利用 TestCase 和我们提供的分析器模块测试你的解释器。

这似乎听起来不太容易, 不过在了解了后面介绍的实验平台、AST 以及要用到的设计模式之后, 这就是一项很简单的工作了。

## 3.3 实验平台介绍

从本章开始的每个课程设计都需要在本书提供的实验平台软件下开展。实验平台将编译器的基本组成部分分成语法分析器、语义检查器、优化器、解释器、汇编代码生成器等类型的组件, 并定义了各类组件的基本接口。你需要根据这些接口的定义实现编译器的各个组件 (通常是实现这些接口的类), 然后在实验平台上运行这些实现, 检查它们是否正确、有效地工作。

由此可见, 实验平台软件主要包括两个部分: 实验平台接口和实验运行平台。关于实验平台软件的目录和文件结构见 3.1 节, 下面分别介绍实验平台接口以及实验运行平台的使用。

### 3.3.1 实验平台接口

为了使你的编译器组件能够在实验平台上运行, 实验平台定义了编译器的各类组件和实验平台之间的接口。你需要实现这些接口, 或者使用本书所配软件包中提供的特定组件。为了运行某类完整的编译器或解释器, 如分析器+解释器、分析器+x86 代码生成器, 你需要

将相应接口的实现类通过配置文件传递给实验平台，然后执行实验运行平台的主类。实验平台接口将中间表示作为编译器各个组件之间信息传递的接口，这降低了编译器各个组件之间的耦合度，使你能够将精力集中在当前你所关注的组件的设计和实现上，而不必过多地关心这个组件如何与其它组件结合。

在edu.ustc.cs.compile.platform.interfaces包中定义了所有的实验平台接口类，如表3-1所示。如果你要使用实验平台接口，你需要类库edu.ustc.cs.compile.platform.interfaces.jar（在platform/lib下）的支持，以及表3-2中的Eclipse AST库的支持。

表 3-1 实验平台接口清单

接口类名	描述
ParserInterface	语法分析器接口
ParserException	语法分析中导致编译器无法继续正确执行时抛出此类异常
InterpreterInterface	解释器接口
InterpreterException	解释执行中导致编译器无法继续正确执行时抛出此类异常
CheckerInterface	语义检查器接口
CheckerException	语义检查中导致编译器无法继续正确执行时抛出此类异常
OptimizerInterface	优化器接口
OptimizerException	优化中导致编译器无法继续正确执行时抛出此类异常
GeneratorInterface	代码生成器接口
GeneratorException	代码生成中导致编译器无法继续正确执行时抛出此类异常
InterRepresent	中间表示访问接口，作为访问高级中间表示(如 AST)或低级中间表示(如接近汇编代码的中间表示)的公共接口
SymTable	符号表接口

### 3.3.1.1 语法分析器接口

**接口类名称：**public interface ParserInterface

**需要实现的方法：**public InterRepresent doParse(File src) throws ParserException

**说明：**你的语法分析器类需要实现这个接口类并给出方法doParse()的实现。实验平台会调用你的实现类中的doParse()方法，将源程序文件通过形参src传递给你的语法分析器。你的语法分析器需要返回一个中间表示（见3.3.1.6节）实例。如果你的语法分析器在分析中发现源程序有错，并且该错误影响编译器后续部分的正确运行时，就需要抛出ParserException异常（见3.3.1.8节）。实验平台捕获到这个异常后，会终止运行。

### 3.3.1.2 解释器接口

**接口定义：**public interface InterpreterInterface

**需要实现的方法：**public void interpret(InterRepresent ir) throws InterpreterException

**说明：**你的解释器类需要实现这个接口类并给出方法interpret()的实现。实验平台会调用你的实现类中的interpret()方法，将源程序的中间表示（见3.3.1.6节）通过形参ir传递给

你的解释器。如果你的解释器在执行时发现程序的中间表示有错，并且该错误影响编译器后续部分的正确运行时，需要抛出异常`InterpreterException`（见 3.3.1.8 节）。实验平台捕获到这个异常后，会终止运行。

### 3.3.1.3 语义检查器接口

**接口定义：** `public interface CheckerInterface`

**需要实现的方法：** `public boolean check(InterRepresent ir) throws CheckerException`

**说明：** 你的语义检查器类需要实现这个接口类并给出方法`check()`的实现。实验平台会调用你的实现类中的`check()`方法，将源程序的中间表示（见 3.3.1.6 节）通过形参`ir`传递给你的语义检查器。如果你的语义检查器认为程序语义正确，需要返回`true`，此时实验平台会继续执行；否则返回`false`，实验平台终止运行。如果你的语义检查器在执行时发现程序有错，并且该错误影响编译器后续部分的正确运行时，需要抛出异常`CheckerException`（见 3.3.1.8 节）。实验平台捕获到这个异常后，会终止运行。

### 3.3.1.4 优化器接口

**接口定义：** `public interface OptimizerInterface`

**需要实现的方法：**

`public InterRepresent doOpt(InterRepresent ir) throws OptimizerException;`

**说明：** 你的优化器类需要实现这个接口类并给出方法`doOpt()`的实现。实验平台会调用你的实现类中的`doOpt()`方法，将源程序的中间表示（见 3.3.1.6 节）通过形参`ir`传递给你的优化器，你的优化器需要返回优化后的代码的中间表示。如果你的优化器在执行时会发生会影响编译器后续部分正确运行的错误时，需要抛出异常`OptimizerException`（见 3.3.1.8 节）。实验平台捕获到这个异常后，会终止运行。

### 3.3.1.5 汇编代码生成器接口

**接口定义：** `public interface GeneratorInterface`

**需要实现的方法：**

`public void generateAsm(File outFile, InterRepresent ir) throws GeneratorException`

`public InterRepresent generateAsm(InterRepresent ir) throws GeneratorException`

**说明：** 你的汇编代码生成器类需要实现这个接口类并给出`generateAsm()`方法的实现。实验平台会调用你的实现类中的`generateAsm()`方法，将源程序的中间表示（见**错误！未找到引用源。**节）通过形参`ir`传递给你的汇编代码生成器。你的汇编代码生成器需要将对应的汇编代码写入到形参`outFile`表示的文件中。如果你的汇编代码生成器在执行时发现程序有错，并且该错误影响编译器的后续部分的正确运行时，需要抛出异常`GeneratorException`（见 3.3.1.8 节）。实验平台捕获到这个异常后，会终止运行。

### 3.3.1.6 中间表示访问接口

**接口定义：** `public interface InterRepresent`

**需要实现的方法：** `public Object getIR()`

```

public void setIR(Object ir)
public SymTable getSymTable()
public void setSymTable(SymTable symTable)

```

**说明：**实验平台降低编译器各个组件耦合度的方法之一就是：各个组件之间的信息传递是以中间表示为基础。为此，实验平台定义了统一的中间表示访问接口 `InterRepresent`。

在本书的课程设计中，将中间表示分成 2 种类别：高级中间表示和低级中间表示。高级中间表示接近源程序，如 Eclipse AST，这类中间表示主要用于在语法分析器、解释器、语义检查器、优化器、汇编代码生成器之间传递信息。低级中间表示更加接近低级语言（例如 gcc 中使用的 RTL），主要用于优化器和汇编代码生成器中。

不论采用什么中间表示，中间表示访问接口都包含方法 `getIR()`、`setIR()`、`getSymTable()` 和 `setSymTable()`，分别用来返回中间表示、设置中间表示、返回符号表、设置符号表。符号表接口参见 3.3.1.7 节。

在本书配套的实验平台类库 `edu.ustc.cs.compile.platform.util.jar` 中提供了两个中间表示访问接口的实现类，即 `HIR` 和 `LIR`（均在 `edu.ustc.cs.compile.platform.util.ir` 包中），分别表示高级中间表示的访问类和低级中间表示的访问类。

#### **`edu.ustc.cs.compile.platform.util.ir.HIR`**

`HIR` 类以 Eclipse AST 为中间表示，它要求传给方法 `setIR()` 的参数必须是 `CompilationUnit` 的实例，表示一个程序的 AST 的根；在 `HIR` 类中，没有定义符号表域，因此其 `getSymTable()` 方法只是简单地返回 `null`，而 `setSymTable()` 方法则什么也不做。如果你需要以 Eclipse AST 为基础扩展符号表，则你可以定义一个继承自 `HIR` 的类。

#### **`edu.ustc.cs.compile.platform.util.ir.LIR`**

`LIR` 类中没有定义低级中间表示和符号表域，你可以定义继承自 `LIR` 的类来表示你所希望的低级中间表示访问类。

### **3.3.1.7 符号表接口**

**接口定义：** `public interface SymTable`

**需要实现的方法：** 无

**说明：**实验平台并不直接调用这个符号表。定义这个符号表接口是为了使中间表示访问接口更加完整，使得编译器的若干组件之间可以复用符号表结构或信息。例如，当语法分析器将中间表示传递给语义检查器时，如果同时提供了符号表，那么语义检查器就无须重新扫描 AST 构造符号表。实验平台没有规定符号表需要实现的方法，关于符号表类的具体结构和做法可以由你自由设计和实现。

### **3.3.1.8 导致编译器终止执行的异常类**

为了使你的编译器组件可以及时向实验平台报告影响编译器后续部分正确运行的各种错误，实验平台的接口部分还提供了以下直接派生自类 `Exception` 的异常类，当实验平台捕获到这些异常时，将终止编译器的运行。这些异常类中没有再定义新的方法。

- **`ParserException`：**语法分析器抛出的异常类。

- **InterpreterException**: 解释器抛出的异常类。
- **CheckerException**: 语义检查器抛出的异常类。
- **OptimizerException**: 优化器抛出的异常类。
- **GeneratorException**: 汇编代码生成器抛出的异常类。

### 3.3.2 实验运行平台

本节将依次介绍实验运行平台所需要的类库，以及实验运行平台的使用方法、命令行选项和配置文件。

#### 3.3.2.1 运行所需的类库

实验运行平台需要表 3-2 中的类库文件的支持。你需要保证这些类库能够在自己的Java运行环境中被找到。

表 3-2 实验平台需要的类库

文件名	说明
edu.ustc.cs.compile.platform.core.jar	这 3 个由实验平台提供
edu.ustc.cs.compile.platform.interfaces.jar	
edu.ustc.cs.compile.platform.util.jar	
org.eclipse.core.resources_*.jar	提供对 Eclipse AST 的支持。你可以在 Eclipse 的安装目录下的 plugins 目录中找到它们，文件名中的*是文件通配符。
org.eclipse.core.runtime_*.jar	
org.eclipse.jdt.core_*.jar	
org.eclipse.jdt.ui_*.jar	
org.eclipse.equinox.common_*.jar	如果前面的四个文件来自 Eclipse3.2 及之后的版本，那么这个文件也是必须的。你可以在 Eclipse 的安装目录下的 plugins 目录中找到它，文件名中的*是文件通配符。

#### 3.3.2.2 实验运行平台的使用

你主要在控制台上运行实验平台，如 Linux 或 Unix 的 bash 控制台、Windows 的 dos 控制台。你可以输入 java 命令行或者输入各个课程设计软件包中 bin 目录下的批命令脚本（Windows 下为 run.bat，Linux 或 Unix 下为 run.sh）来启动实验平台的运行。

##### 1、java 命令行

运行实验平台的 java 命令行如下所示：

```
java -classpath <classpath> edu.ustc.cs.compile.platform.Main [options] [source-file]
```

其中，

- <>表示必选的参数，[]表示可选的参数。<、>、[、]不是参数的组成内容。
- <classpath>中至少需要指定运行实验平台需要的类库文件（见表 3-2）的位置。如

如果你的代码中用到了其它第三方类库（即不是JRE提供的类库），你也需要在这里指定它的位置。例如，在课程设计 1 中你可能需要使用SimpleMiniJOOOL语法分析器，这时你就需要在<classpath>中添加edu.ustc.cs.compile.parser.simpleminijool.jar的位置。

- [options]指定实验平台的各种选项。你可以根据各个课程设计的需要适当选择。如果你使用实验平台的配置文件，也需要在这里指定配置文件的位置。3.3.2.3 节详细介绍实验平台的命令行选项，3.3.2.4 节详细介绍实验平台的配置文件。
- [source-file]指定待编译的源程序文件的位置。如果你使用配置文件，可以忽略这个参数；否则，你必须指定一个源程序文件。

## 2、批命令脚本

由于运行实验平台时需要有许多类库的支持，因此需要在系统环境变量 CLASSPATH 中或者用来运行实验平台的 java 命令行的-classpath 选项中指定这些类库的位置。虽然在系统环境中设置 CLASSPATH 能避免你每次运行 java 时输入 classpath 选项，但是一旦重装系统或者在别的机器上运行，就需要重新设置环境变量。为避免你重复配置 classpath，我们在每个课程设计软件包的bin目录下都提供了调用实验平台的脚本文件run.sh(用于Linux或Unix平台)和run.bat(用于MS Windows平台)。

这样，你可以在放有run.bat或run.sh的目录下用以下命令运行实验平台：

*在Linux和类Linux平台中*

```
./run.sh [options] [source-file]
```

*在MS Windows平台中*

```
run.bat [options] [source-file]
```

其中，[options]和[source-file]与前面出现的命令中的完全一致。

如果你使用了其它第三方类库，就需要修改脚本文件中的变量CLASSPATH。例如要添加对第三方类库文件third.jar（假设在../lib下）的支持，你需要对脚本做如下修改：

- **run.sh**：在“CLASSPATH=...\${PLATFORM\_DIR}/”之后（不要换行）添加“: ../lib/third.jar”；
- **run.bat**：在“set CLASS\_PATH=...%PLATFORM\_DIR%”之后（不要换行）添加“; ..\lib\third.jar”。

### 3.3.2.3 实验平台的命令行选项

- **--help 或 -h**：获取实验平台命令行参数的说明。使用这个参数时，其它参数会被自动忽略。
- **--cfg-file <file>或 -cf <file>**：指定配置文件的位置。当实验平台看到这个参数后，会从配置文件中读取参数，并且忽略除--help和-h以外的所有参数。
- **--debug 或 -d**：打开调试模式。使用这个参数可以使实验平台输出更多的内容。这些内容可能对你调试自己的代码有所帮助。
- **--parser-class <classname>或 -P <classname>**：指定你的语法分析器的全称类名，类



名缺省为 `edu.ustc.cs.compile.parser.skipoominijool.Parser`。在课程设计 1 中，你可以指定 `<classname>` 为我们提供的 `SimpleMiniJOOL` 分析器类，即 `edu.ustc.cs.compile.parser.simpleminijool.Parser`，你也可以指定 `<classname>` 为你编写的实现手工构造 AST 的 `edu.ustc.cs.compile.interpreter.TestCase` 类。在使用这个参数和以下的两个参数时，你必须保证这些类可以在 `classpath` 中可见。

- **--disp-ast=[yes|no]或-da=[yes|no]**: 指定是否调用 `ASTView` 查看语法分析得到的 AST。其中 `[yes|no]` 表示从 `yes` 和 `no` 中选择一个。“[”、“]”、“|”不是参数的组成部分。这个参数缺省设为 `no`。
- **--do-interp=[yes|no]或-i=[yes|no]**: 指定是否调用解释器，缺省设为 `no`。在课程设计 1 中，你需要将其设为 `yes`。
- **--interp-class <classname>或-I <classname>**: 指定你的解释器的全称类名，缺省类名为 `edu.ustc.cs.compile.interpreter.Interpreter`。该选项在 **--do-interp=yes** 或 **-i=yes** 时生效。
- **--do-check=[yes|no]或-c=[yes|no]**: 指定是否调用语义解析器。缺省设为 `no`。
- **--checker-class <classname>或-C <classname>**: 指定你的语义检查器的全称类名，缺省类名为 `edu.ustc.cs.compile.checker.skipoominijool.Checker`。该选项在 **--do-check=yes** 时生效。
- **--do-opt =[yes|no]（或-opt=[yes|no]）**: 指定是否调用第一阶段优化器。缺省设为 `no`。如果该参数为 `yes`，实验平台将在语法分析器和语义检查器执行结束后调用第一阶段优化器。
- **--opt-class <classname>（或-O <classname>）**: 指定平台调用的优化器的全称类名，缺省为 `edu.ustc.cs.compile.optimizer.Optimizer`。该参数在 **--do-opt =yes（或-opt=yes）** 时生效。
- **--disp-opt-ast=[yes|no]（或-doa=[yes|no]）**: 指定是否调用 `ASTView` 来查看经过优化后的 AST 树。打开这个选项可以对比优化前的 AST 树，手工检查针对 AST 树的优化是否正确。
- **--dump-class <classname>（或-D <classname>）**: 指定定制 `ASTView` 的输出属性（详见 3.5.6.2 节）的类的全称类名。缺省使用 `ASTView` 自带的属性定制类。该参数在 **--disp-ast=yes（或-da=yes）** 或 **--disp-opt-ast=yes（或-doa=yes）** 时有效。
- **--gen-asm=[yes|no]或-g=[yes|no]**: 指定是否调用汇编代码生成器，缺省设为 `yes`。在课程设计 1 中，你需要将其设为 `no`。
- **--gen-class <classname>或-G <classname>**: 指定你的汇编代码生成器的全称类名，缺省类名为 `edu.ustc.cs.compile.generator.Generator`。该选项在 **--gen-asm=yes（或-g=yes）** 时生效。
- 用 **--arch=[x86|mips] 或 -a=[x86|mips]**: 指定生成 Intel X86 架构还是 MIPS R2000/R3000 架构上的汇编代码。默认为 `x86`。该选项在 **--gen-asm=yes（或-g=yes）** 时生效。

- **-S <file>**: 指定汇编代码文件名 file, 使得汇编代码生成器将生成的汇编代码输出到该文件中。默认 file 为 a.s。该选项在 **--gen-asm=yes** (或 **-g=yes**) 时生效。
- **--exec=[yes|no]** 或 **-e=[yes|no]**: 指定是否运行生成的汇编代码。如果生成的是 x86 架构上的汇编代码, 实验平台会先调用 gcc 将汇编代码编译成可执行文件, 然后再执行。如果生成的是 mips 架构上的汇编代码, 实验平台会调用 spim 执行这些汇编代码。默认为 yes。该选项在 **--gen-asm=yes** (或 **-g=yes**) 时生效。
- **--gcc-path <path>** 或 **-gcc <path>**: 指定你的系统中 gcc 的绝对路径。默认为 /usr/bin/gcc。该选项在 **--gen-asm=yes** (或 **-g=yes**) 且 **--arch=x86** (或 **-a=x86**) 时生效。
- **-o <file>**: 指定可执行文件名 file, 使得 gcc 将汇编代码编译链接成可执行代码输出到 file 中。如果指定 **--arch=yes**, 那么实验平台忽略这个参数。默认为 a.out。该选项在 **--gen-asm=yes** (或 **-g=yes**) 且 **--arch=x86** (或 **-a=x86**) 时生效。
- **--spim-path <path>** 或 **-spim <path>**: 指定你的系统中 spim 的绝对路径。默认为 /usr/local/bin/spim。该选项在 **--gen-asm=yes** (或 **-g=yes**) 且 **--arch=mips** (或 **-a=mips**) 时生效。

### 3.3.2.4 实验平台的配置文件

使用命令行参数的优点是灵活, 缺点是重复输入过多 (每次运行都需要输入较长的、与上次输入几乎相同的参数)。因此, 我们提供另一种指定实验平台运行参数的方法——配置文件, 并在各个课程设计软件包的 config 目录下提供用于该课程设计的配置文件样本, 如 lab1/config/lab1-parser.xml。要使用配置文件, 你需要使用命令行参数 **--cfg-file <file>** (或 **-cf <file>**) 指定配置文件。下面先介绍配置文件的格式, 然后给出配置文件的示例。

#### 1、配置文件的格式

配置文件采用 XML 文件格式来描述。在 ROOT\_DIR/platform/config 或者各个课程设计软件包的 config 目录下都有一个 configure.xsd 文件, 这个 XML Schema 文件是用来约束配置文件的格式和内容的。下面结合 configure.xsd 来说明配置文件的格式。

配置文件的根元素是 configs, 在 configure.xsd 的第 4~12 给出这个元素的类型定义:

```

4  <xs:element name="configs" type="Config"/>
5
6  <xs:complexType name="Config">
7    <xs:sequence>
8      <xs:element ref="boolCfgs" minOccurs="0" maxOccurs="1"/>
9      <xs:element ref="strCfgs" minOccurs="0" maxOccurs="1"/>
10     <xs:element ref="archCfgs" minOccurs="0" maxOccurs="1"/>
11   </xs:sequence>
12</xs:complexType>

```

上述类型定义指出该元素中可以依次含有 0 个或 1 个 boolCfgs 子元素 (用来配置布尔

型参数)、0 个或 1 个 strCfgs 子元素 (用来配置字符串型参数) 和 0 个或 1 个 archCfgs 子元素 (用来配置汇编代码架构参数)。

### 布尔型参数

布尔型参数在 boolCfgs 元素中配置, 该元素的类型定义在 configure.xsd 的第 14~20 行:

```

14 <xs:element name="boolCfgs">
15   <xs:complexType>
16     <xs:sequence>
17       <xs:element ref="boolCfg" minOccurs="0" maxOccurs="unbounded"/>
18     </xs:sequence>
19   </xs:complexType>
20 </xs:element>

```

它指出 boolCfgs 元素内含有 0 个或多个 boolCfg 子元素, 每个 boolCfg 子元素用来配置一个布尔型参数。boolCfg 元素的类型定义见 configure.xsd 的第 22~27 行:

```

22 <xs:element name="boolCfg">
23   <xs:complexType>
24     <xs:attribute name="name" type="BoolCfgName" use="required"/>
25     <xs:attribute name="value" type="xs:boolean" use="required"/>
26   </xs:complexType>
27 </xs:element>

```

上述类型定义指出每个 boolCfg 元素包含两个必须的 (required) 属性, 即 name 和 value。value 属性的取值范围限制为布尔型, 即值为 "true" 或 "false"。name 属性的取值范围由 configure.xsd 的第 29~40 行的类型定义 BoolCfgName 来限定:

```

29 <xs:simpleType name="BoolCfgName">
30   <xs:restriction base="xs:string">
31     <xs:enumeration value="debug"/>
32     <xs:enumeration value="dispAST"/>
33     <xs:enumeration value="dispOptAST"/>
34     <xs:enumeration value="doInterp"/>
35     <xs:enumeration value="doCheck"/>
36     <xs:enumeration value="genAsm"/>
37     <xs:enumeration value="doOpt"/>
38     <xs:enumeration value="exec"/>
39   </xs:restriction>
40 </xs:simpleType>

```

BoolCfgName 类型定义指出属于该类型的值只能取 "debug"、"dispAST"、"dispOptAST"、"doInterp"、"doCheck"、"genAsm"、"doOpt" 和 "exec" 之一, 这些值分别与命令行选项 --debug (或 -d)、--disp-ast (或 -da)、--disp-opt-ast (或 -doa)、--do-interp (或 -i)、--do-check (或 -c)、

--gen-asm（或-g）、--do-opt（或-opt）、--exec（或-e）相对应。

### 字符串型参数

字符串型参数在 strCfgs 元素中配置, 该元素的类型定义在 configure.xsd 的第 41~47 行。与 boolCfgs 元素类似, strCfgs 元素内含 0 个或多个 strCfg 子元素, 每个 strCfg 子元素用来配置一个字符串型参数。strCfg 元素同样由 name 和 value 两个必须的属性组成 (类型定义见 configure.xsd 的第 50~55 行)。value 属性的取值范围限制为字符串型, 即"xs:string"。

```

50 <xs:element name="strCfg">
51   <xs:complexType>
52     <xs:attribute name="name" type="StrCfgName" use="required"/>
53     <xs:attribute name="value" type="xs:string" use="required"/>
54   </xs:complexType>
55 </xs:element>

```

name 属性的取值范围由 configure.xsd 的第 57~71 行的类型定义 StrCfgName 来限定:

```

57 <xs:simpleType name="StrCfgName">
58   <xs:restriction base="xs:string">
59     <xs:enumeration value="srcFile"/>
60     <xs:enumeration value="asmFile"/>
61     <xs:enumeration value="exeFile"/>
62     <xs:enumeration value="parserClass"/>
63     <xs:enumeration value="checkerClass"/>
64     <xs:enumeration value="interpClass"/>
65     <xs:enumeration value="optClass"/>
66     <xs:enumeration value="genClass"/>
67     <xs:enumeration value="dumpClass"/>
68     <xs:enumeration value="spimPath"/>
69     <xs:enumeration value="gccPath"/>
70   </xs:restriction>
71 </xs:simpleType>

```

StrCfgName 类型定义指出属于该类型的值只能取"srcFile"、"asmFile"、"exeFile"、"parserClass"、"checkerClass"、"interpClass"、"optClass"、"genClass"、"dumpClass"、"spimPath"、和"gccPath"之一, 这些值分别对应于命令行选项 source-file、-S、-o、--parser-class（或-P）、--checker-class（或-C）、--interp-class（或-I）、--opt-class（或-O）、--gen-class（或-G）、--dump-class（或-D）、--spim-path（或-spim）、--gcc-path（或-gcc）相对应。

### 汇编代码架构参数

汇编代码架构参数在 archCfgs 元素中配置, 它内含 1 个 arch 子元素。一个 arch 元素只含一个必须的 value 属性, 该属性的值限定为取"x86"或"mips"之一。arch 元素与命令行选项 --arch（或-a）相对应。

## 2、配置文件示例

图 3-1 是课程设计 1 软件包中的一个实验平台配置文件lab1-parser.xml。其中， configs元素中的属性用于指明约束该配置文件的XML Schema文件的位置。在boolCfgs元素中，依次设置实验平台将选用debug模式、显示AST、调用解释器组件、不产生汇编代码也不执行汇编代码；在strCfgs元素中，依次指定SimpleMiniJOOl源程序的位置、分析器的类名、解释器的类名、以及ASTView属性定制类的类名。

```
<?xml version="1.0"?>

<configs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="./configure.xsd">
  <boolCfgs>
    <boolCfg name="debug" value="true"/>
    <boolCfg name="dispAST" value="true"/>
    <boolCfg name="doInterp" value="true"/>
    <boolCfg name="genAsm" value="false"/>
    <boolCfg name="exec" value="false"/>
  </boolCfgs>

  <strCfgs>
    <strCfg name="srcFile" value="test/syntax.mj"/>
    <strCfg name="parserClass"
      value="edu.ustc.cs.compile.parser.simpleminijool.Parser"/>
    <strCfg name="interpClass"
      value="edu.ustc.cs.compile.interpreter.Interpreter"/>
    <strCfg name="dumpClass"
      value="edu.ustc.cs.compile.platform.utils.ASTView.plugin.GenericPropertyDump"/>
  </strCfgs>
</configs>
```

图 3-1 一个实验平台配置文件：lab1-parser.xml

## 3.4 课程设计 1 开发和测试指南

在确认把本书配套的软件包（包括 lab、lib、platform 和 tools 子目录）复制到你的编译实验根目录（假设为 E:/CompilerProj/student，记为 ROOT\_DIR）中之后，你可以开始课程设计 1 的开发实践了。你可以选择在 Eclipse 下编辑、编译、调试你的程序，也可以利用文本编辑器编辑、用控制台命令来编译和运行。下面分别简述这两种情况下的开发方法。

### 3.4.1 在 Eclipse 下开发

#### 1、建立工程

在 1.4.3.3 节介绍了多种建立Eclipse工程的方法，不同的方法导致所建立的工程的工程文件（即.project和.classpath文件）存放路径以及工程的输出路径不太一样，这给Eclipse初学者造成了混乱。为便于Eclipse初学者能轻松建立和控制工程，在每个课程设计软件包的根目录（如ROOT\_DIR/lab/lab1）下都提供有.project和.classpath。图 3-2 和 图 3-3 分别给出了课程设计 1 的.project文件和.classpath文件。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <projectDescription>
3   <name>SimpleMiniJOOL Intepreter</name>
4   <comment></comment>
5   <projects>
6   </projects>
7   <buildSpec>
8     <buildCommand>
9       <name>org.eclipse.jdt.core.javabuilder</name>
10      <arguments>
11      </arguments>
12    </buildCommand>
13  </buildSpec>
14  <natures>
15    <nature>org.eclipse.jdt.core.javanature</nature>
16  </natures>
17  <linkedResources>
18    <link>
19      <name>classes</name>
20      <type>2</type>
21      <location>E:/CompilerProj/student/lab/lab1/classes</location>
22    </link>
23    <link>
24      <name>src</name>
25      <type>2</type>
26      <location>E:/CompilerProj/student/lab/lab1/src</location>
27    </link>
28  </linkedResources>
29</projectDescription>

```

图 3-2 lab1/.project 文件

你需要注意.project 文件中的第 21、26 行，这两行分别设置 classes 和 src 的位置。如果你的 ROOT\_DIR 不是 E:/CompilerProj/student，则你需要修改这两行，保证 Java 源程序的位

置和输出的 class 文件位置被正确地设置。

.classpath 文件中主要设置编译运行工程所需的 classpath。你需要修改并确认这个文件中的各个位置是否被正确设置。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <classpath>
3   <classpathentry output="classes" kind="src" path="src"/>
4   <classpathentry sourcepath="JRE_SRC" kind="var" rootpath="JRE_SRCROOT"
      path="JRE_LIB"/>
5   <classpathentry kind="lib" path=
      "E:/CompilerProj/student/platform/lib/edu.ustc.cs.compile.platform.interfaces.jar"/>
6   <classpathentry kind="lib" path=
      "E:/CompilerProj/student/lib/edu.ustc.cs.compile.parser.simpleminijool.jar"/>
7   <classpathentry kind="lib" path=
      "E:/CompilerProj/student/tools/java-cup/java-cup-11a-runtime.jar"/>
8   <classpathentry kind="lib" path=
      "E:/CompilerProj/student/lib/AST/3.1.2/org.eclipse.core.resources_3.1.2.jar"/>
9   <classpathentry kind="lib" path=
      "E:/CompilerProj/student/lib/AST/3.1.2/org.eclipse.core.runtime_3.1.2.jar"/>
10  <classpathentry kind="lib" path=
      "E:/CompilerProj/student/lib/AST/3.1.2/org.eclipse.jdt.core_3.1.2.jar"/>
11  <classpathentry kind="lib" path=
      "E:/CompilerProj/student/lib/AST/3.1.2/org.eclipse.jdt.ui_3.1.2.jar"/>
12  <classpathentry kind="output" path="classes"/>
13</classpath>

```

图 3-3 lab1/.classpath 文件

在确认.project 和.classpath 正确设置之后，你可以用 1.4.3.3 节中第 5 点“将现有工程导入到工作区中”介绍的方法建立工程。这样，你就可以在 Eclipse 下对这个工程进行开发了。

## 2、编译与调试

你可以选择 Eclipse 的“Project”菜单中的 Build 项进行编译（见 1.4.3.6 节），也可以选择“Package Explorer”View 中该工程的 bin 目录下的 build.xml，单击鼠标右键菜单项“Run as/Debug as”→“Ant Build”来启动 ant 进行编译。

为调试你所编写的 Interpreter 程序，你可以在“Package Explorer”View 中选择 src 目录下 edu.ustc.cs.compile.interpreter 包中的主类 Main.java，单击鼠标右键菜单项“Run as/Debug as”→“Java Application”启动运行或调试。

## 3、用更多的 AST 测试解释器

作为输入给解释器的中间表示 AST 可以通过两种渠道获得：一种是利用我们提供的

SimpleMiniJOOl 分析器分析 SimpleMiniJOOl 程序得到，一种是执行你写的 TestCase 类中的手工构建 AST 方法来获得。

为使用 TestCase 类中构造的 AST 进行测试，你需要将类 Main 的 main 方法里的 fromParser 置为 false，并且修改类 TestCase 中的 doParse 方法里的 ir.setIR() 中的参数，使之为你所希望执行的 AST 构建方法，如

```
ir.setIR(createAST());
```

为使用我们提供的分析器对输入的源程序文件进行分析构造 AST，再以该 AST 进行测试，你需要将类 Main 的 main 方法里的 fromParser 置为 true，并且在 main 方法中设置你所希望测试的源程序文件名，如：

```
inFile = new File("test/syntax.mj");
```

### 3.4.2 在控制台下编译和运行

在用编辑器补全课程设计 1 的所有源代码后，你可以启动 dos 或 bash 控制台。在启动控制台前，你需要确认 Java 虚拟机、实验平台的 classpath 以及所用操作系统的文件搜索路径已被正确地配置。接着，启动控制台并按下面的步骤对课程设计 1 进行编译和运行：

- 1) 进入 lab1/bin 目录；
- 2) 执行 ant 或 ant build，编译所有的源代码；
- 3) 执行 run.bat 或 run.sh（前者针对 dos 控制台，后者针对 bash 控制台）来运行完整的解释器，你需要正确地设置配置文件或命令行选项。

**注意：**在 bash 控制台下，你需要通过下面的命令将 run.sh 变为可执行：

```
chmod +x run.sh
```

然后执行 run.sh 或 ./run.sh 来启动实验平台运行完整的解释器。

下面重点说明如何用 run.bat 或 run.sh（以下统称为 run）来运行解释器进行测试，我们将按 AST 的不同构建渠道来分别说明。

#### 使用 SimpleMiniJOOl 分析器组件

你可以在命令行输入以下命令启动实验平台，测试对 SimpleMiniJOOl 程序的解释执行：

```
run -P edu.ustc.cs.compile.parser.simpleminijool.Parser -da=yes
-i=yes -g=no -I edu.ustc.cs.compile.interpreter.Interpreter
<SimpleMiniJOOl 程序文件名>
```

你也可以使用配置文件 lab1/config/lab1-parser.xml 来启动实验平台并测试：

```
run --cfg-file ../config/lab1-parser.xml
```

你可以修改 lab1/config/lab1-parser.xml 中如下元素的 value 值，以设置待测试的 SimpleMiniJOOl 程序文件名：

```
<strCfg name="srcFile" value="../test/syntax.mj"/>
```

#### 使用 TestCase 类手工构建 AST

你可以在命令行输入以下命令启动实验平台，测试对你在 TestCase 中构建的 AST 的解



释执行：

```
run -P edu.ustc.cs.compile.interpreter.TestCase -da=yes -i=yes
-g=no -I edu.ustc.cs.compile.interpreter.Interpreter
```

你也可以使用配置文件 `lab1/config/lab1-testcase.xml` 来启动实验平台并测试：

```
run --cfg-file ../config/lab1-testcase.xml
```

这些命令的具体解释可以参考 3.3.1。

### 3.4.3 测试要求

在课程设计 1 中，要求你手工创建 AST 的目的在于使你能尽快熟悉构造 AST 的方法，了解一个 mj 程序与 AST 之间的对应关系。这对于你顺利完成后面的课程设计是有必要的。

当然，仅仅用一个测试用例来测试解释器的功能和正确性是不行的，而像 TestCase 中那样手工构造 AST 又非常繁琐。所以，更多的测试用例是采用前面提到的直接编写一个 SimpleMiniJOOl 程序的源代码，然后用我们提供的 SimpleMiniJOOl 分析器来分析它得到对应的 AST。

我们在 lab1/test 中包含了一些简单的测试用例，你可以使用它们来测试，指导老师也可以用它们来检查学生的成果。实现的细节问题可以适度把握，测试的侧重点在：

- 中缀表达式、前缀表达式；
- print 和 read 语句；
- 赋值语句；
- if/while 语句和它们的条件；
- 当 print/read 语句结果被引用或者 0 作为除数时，需要报错。

## 3.5 抽象语法树（AST）

抽象语法树（AST，Abstract Syntax Tree）是编译器中常用的中间表示形式之一。相对其它一些中间表示形式，它更接近源程序语言，能够更清楚地反映源程序的语法结构。在本书的课程设计中，都将使用 AST 作为表示 SimpleMiniJOOl 语言以及后面出现的 SkipOOMiniJOOl 语言、MiniJOOl 语言的中间形式。此外，AST 将和符号表结合在一起，在编译器的各个组件之间传递信息。

在课程设计中，你不需要自己实现 AST，而是使用 Eclipse JDT 中的 AST 实现（以下简称 Eclipse AST）。下面是对 Eclipse AST 结构和使用的说明。

### 3.5.1 Eclipse AST 的总体结构

Eclipse AST 是 Eclipse JDT 的一个重要组成部分，可以表示 Java 语言中的所有语法结构；它采用工厂方法模式和访问者模式（参见 3.6 节）来设计，可以减轻程序员需要深入了解其结构的压力。

需要特别注意的是，使用 Eclipse AST 构建的抽象语法树在拓扑结构上必须是无环的。如果构建抽象语法树的程序违反了这个原则，那么在运行这个程序时会抛出异常。

Eclipse AST 包括多个部分，其中与本书的课程设计有关的主要是以下三个部分：

- `org.eclipse.jdt.core.dom.AST`：Eclipse AST 的工厂类，用于创建表示各种语法结构的节点；
- `org.eclipse.jdt.core.dom.ASTNode` 及其派生类：用于表示 Java 语言中的所有语法结构，在实际使用中常作为 AST 上的节点出现；
- `org.eclipse.jdt.core.dom.ASTVisitor`：Eclipse AST 的访问者类，定义了统一的访问 AST 中各个节点的方法。

### 3.5.2 `org.eclipse.jdt.core.dom.AST`

Eclipse AST 的工厂类，提供一系列形如 `newTYPE()` 的方法，用来构造名为 `TYPE` 的 Eclipse AST 节点。要使用这些方法，首先需要创建 `org.eclipse.jdt.core.dom.AST` 的实例：

```
AST ast = AST.newAST(AST.JLS3);
```

其中，参数 `AST.JLS3` 指示所生成的 `ast` 包含处理 JLS3(Java 语言规范第3版)的 AST API。JLS3 是 Java 语言所有早期版本的超集，JLS3 API 可以用来处理直到 Java SE 6(即 JDK1.6)的 Java 程序。

表 3-3 SimpleMiniJOOl 使用的 AST 节点类型

AST 节点类	用处	说明
<code>CompilationUnit</code>	编译单元	在 Java 中表示整个 Java 源文件，允许包含多个类声明。
<code>TypeDeclaration</code>	类声明	在 Java 中被用作类声明和接口声明，但 SimpleMiniJOOl 中仅有一个类声明。
<code>MethodDeclaration</code>	方法声明	在 SimpleMiniJOOl 中只有一个 Main 方法声明。
<code>Block</code>	语句列表	花括号括起来的复合语句
<code>ExpressionStatement</code>	表达式语句	由表达式组成的合法语句，称为表达式语句。在 SimpleMiniJOOl 中可以表示 <code>print/read</code> 语句或赋值表达式语句。
<code>IfStatement</code>	if 语句	
<code>WhileStatement</code>	while 语句	
<code>MethodInvocation</code>	方法调用	用来表示 SimpleMiniJOOl 中的 <code>print/read(&lt;参数&gt;)</code> ，方法的名字总是“ <code>print</code> ”或“ <code>read</code> ”，并且只有一个参数。
<code>Assignment</code>	赋值表达式	在 SimpleMiniJOOl 中，包括 <code>=</code> 以及 <code>+=</code> 、 <code>-=</code> 、 <code>*=</code> 、 <code>/=</code> 、 <code>%=</code> 这样的复合赋值运算符。
<code>InfixExpression</code>	中缀表达式	在 SimpleMiniJOOl 中，包括 <code>+</code> 、 <code>-</code> 、 <code>*</code> 、 <code>/</code> 、 <code>%</code> 、 <code>==</code> 、 <code>!=</code> 、 <code>&lt;</code> 、 <code>&lt;=</code> 、 <code>&gt;</code> 、 <code>&gt;=</code> 、 <code>&amp;&amp;</code> 、 <code>  </code> 运算符。
<code>PrefixExpression</code>	前缀表达式	在 SimpleMiniJOOl 中，包括 <code>-</code> 、 <code>+</code> 、 <code>!</code> 运算符。
<code>SimpleName</code>	变量名	
<code>NumberLiteral</code>	整数数字	

### 3.5.3 org.eclipse.jdt.core.dom.ASTNode 及其派生类

org.eclipse.jdt.core.dom.ASTNode及其派生类可以表示Java语言中的所有语法结构，而且这些类通过组合形成一个树形结构，方便利用语法规则进行归约。虽然这些类为数众多，但是SimpleMiniJOOL语言只会用到如表3-3所示的一小部分。

### 3.5.4 org.eclipse.jdt.core.dom.ASTVisitor

org.eclipse.jdt.core.dom.ASTVisitor 是 Eclipse AST 的访问者类，提供了统一的访问方法用于访问 AST 中的各个节点。

课程设计中设计和使用的各个 AST 访问者类都是由 ASTVisitor 派生。在 ASTVisitor 中提供对各类 AST 节点的 visit 方法，其中与某个特定的 ASTNode 的派生类 T 对应的 visit 方法为

```
public boolean visit(T n)
```

这个方法首先访问节点 n。如果方法返回 true，Eclipse AST 会继续访问 n 的子孙节点；否则，Eclipse AST 不再向下访问。在 ASTVisitor 中，所有的 visit 方法的方法体对传入的参数节点 n 不做任何处理，直接返回 true。

在课程设计1中，你需要定义自己的访问者类，即 Interpreter.java 中的 InterpVisitor，以实现对各种语法结构的解释执行。具体而言，你需要从 ASTVisitor 派生自己的访问者类 InterpVisitor，重写与需要解释的语法结构相对应的 visit() 方法。

要想对 AST 节点 n 调用你的访问者类实例 visitor 中的 visit() 方法，可以使用 AST 节点的方法 accept()，即 n.accept(visitor)。

### 3.5.5 Eclipse AST 使用示例

在这一节中，我们将演示如何利用 Eclipse AST 手工构建如下的 SimpleMiniJOOL 程序的 AST 中间表示。

```
class Program {
    static void main() {
        i = 10;
    }
}
```

首先，你需要通过 Eclipse AST 工厂类中的方法 newAST() 建立一个 AST 实例：

```
AST ast = AST.newAST(JLS3);
```

利用这个 AST 实例，就可以按如下的方法创建各种 AST 节点，并构建完整的抽象语法树。

然后，利用 Eclipse AST 工厂类中的各种创建方法按如下步骤创建所需要的 AST 节点：

1) 整个 SimpleMiniJOOL 程序构成一个 CompilationUnit：

```
CompilationUnit cu = ast.newCompilationUnit();
```

2) 在 CompilationUnit 实例中包含一个 TypeDeclaration，表示程序中的类 Program：

```
TypeDeclaration type = ast.newTypeDeclaration( );
```

```

type.setName(ast.newSimpleName("Program"));    // 定义类的名字
3) 在这个 TypeDeclaration 实例中添加类 Program 中的方法 main():
MethodDeclaration method = ast.newMethodDeclaration( );
method.setName(ast.newSimpleName("main"));
type.bodyDeclarations().add(method);
// 设置方法 main()的 modifier 为 static
method.modifiers().add(
    ast.newModifier(Modifier.ModifierKeyword.STATIC_KEYWORD));
// 设置方法 main()的返回类型为 void
method.setReturnType2(ast.newPrimitiveType(PrimitiveType.VOID));
4) 构造 main 函数的函数体 mainBody
Block mainBody = ast.newBlock();
method.setBody(mainBody);
5) 向方法 main 函数体 mainBody 中添加语句
// 构建赋值表达式
Assignment assign = ast.newAssignment();
// 设置赋值表达式的左值为 i
assign.setLeftHandSide(ast.newSimpleName("i"));
// 设置赋值表达式的赋值算符为=
assign.setOperator(Assignment.Operator.ASSIGN);
// 设置赋值表达式的右值为数字 10
assign.setRightHandSide(ast.newNumberLiteral("10"));
// 由赋值表达式构建语句，并把这个语句加入方法 Main()的函数体
ExpressionStatement statement = ast.newExpressionStatement(assign);
mainBody.statements().add(statement);

```

至此，用 Eclipse AST 表示的 SimpleMiniJOOOL 程序的抽象语法树就构建完毕了。

在 lab1/src/edu/ustc/cs/compile/interpreter/TestCase.java 中的 createSampleAST()方法给出了构建一个简单 SimpleMiniJOOOL 程序对应的 AST 的完整示例。

需要再次强调的是，使用 Eclipse AST 构建的抽象语法树在拓扑结构上必须是无环的。无论是手工构建 AST 还是自动构建 AST，你都需要小心的检查自己的代码，避免违反这个原则。

### 3.5.6 AST 的图形化显示包——ASTView

为了便于你直观地查看以Eclipse AST为基础构建的AST，我们提供AST的图形化显示包 edu.ustc.cs.compile.platform.utils.ASTView（简称ASTView包）。ASTView包不是Eclipse AST 的组成部分，而是实验平台的一部分。在ASTView包中，core子包中的类ASTViewer能接收一个mj程序的AST实例的根（类型为CompilationUnit），然后以图 3-4 所示的图形界面形式

显示所接收的AST实例的树形结构，并提供对树中节点的各项属性以及节点对应的代码的浏览功能。在这个图形界面中，左部为AST结构显示区，右部分为上下两部分，上部是AST节点属性显示区，下部是该AST对应的代码的显示区。

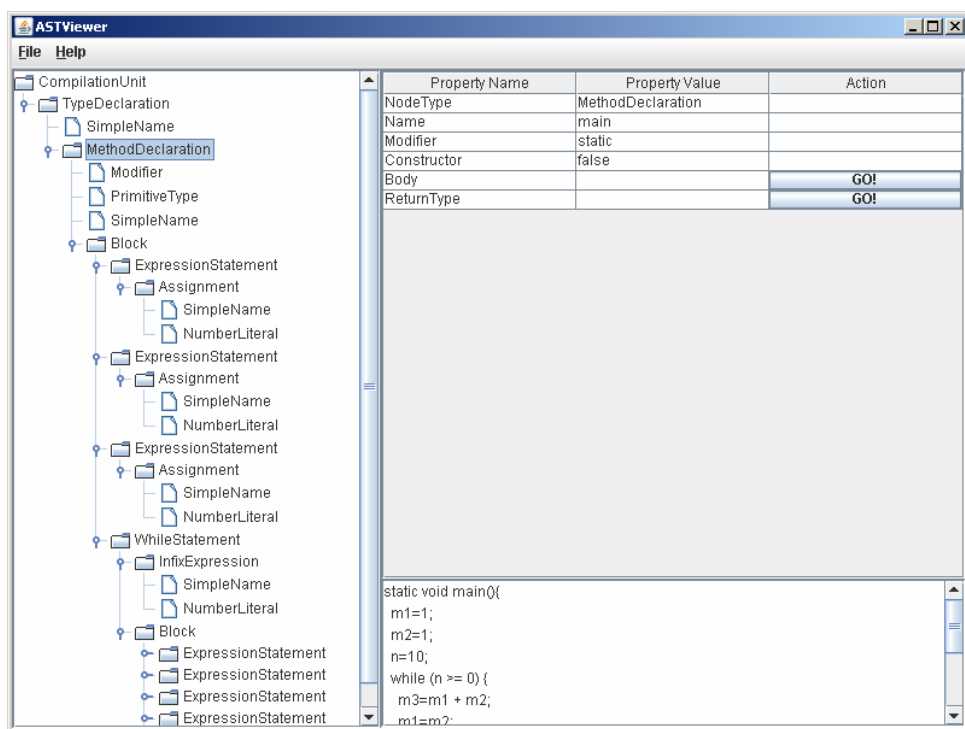


图 3-4 ASTViewer 的图形界面

实验平台缺省地给出各类节点在 ASTViewer 中显示的属性及内容。如果你对所显示的属性及其内容不满意，那么你可以通过 ASTView 包中提供的属性定制接口提交自己定制的属性列表。

在以下各小节中，将依次介绍 ASTViewer 的使用以及 ASTView 的属性定制等。

### 3.5.6.1 ASTViewer 的使用

大多数情况下，你无需在自己的代码中显式调用 ASTViewer，而是通过实验平台的命令行参数 `--disp-ast=yes`（或 `-da=yes`）或 `--disp-opt-ast=yes`（或 `-doa=yes`）调用 ASTViewer 显示 AST。前者告诉实验平台在语法分析器组件执行结束后，显示该组件产生的 AST；后者告诉实验平台在优化器组件执行结束后，显示该组件优化后得到的 AST。

如果你希望在自己的代码中调用 ASTViewer，那么你需要在代码中显式创建 ASTViewer 对象，并将要显示的 AST 的根节点（请确认其类型为 `CompilationUnit`）传递给 ASTViewer 对象，然后调用 ASTViewer 对象的方法 `show()` 来图形化显示 AST。你可以参考以下代码：

```
// cu 为要显示的 AST 的根节点，类型为 CompilationUnit
ASTViewer astviewer = new ASTViewer(cu, null);
astviewer.show();
```

要使用 ASTViewer，在你的 Java 程序中必须包含以下两条 import 声明：

```
import edu.ustc.cs.compile.platform.utils.ASTView.core;
```

```
import edu.ustc.cs.compile.platform.utils.ASTView.plugin;
```

### 3.5.6.2 ASTView 的输出属性定制

如果你忽略实验平台的命令行选项`--dump-class`（或`-D`），或者你使用带一个参数的 `ASTViewer` 构造器或带两个参数的 `ASTViewer` 构造器但第二个参数置为 `null`，那么 `ASTView` 包将根据类 `plugin.GenericPropertyDump` 中的方法 `dump()` 获得要为各种 `AST` 节点显示的属性。类 `GenericPropertyDump` 派生自类 `ASTViewPropertyDump`（在 `core` 子包中）。

类 `ASTViewPropertyDump` 为每种 `AST` 节点类型 `T` 定义了方法

```
public List dump(T n)
```

方法 `dump()` 返回一个要显示的属性列表。列表中的每个元素的类型为类 `core.ASTViewPropertyItem`，这个类中定义了一个三元组<属性名称，属性值，属性指向的节点>，并提供如表3-4所示的公共方法修改和获得这些值。其中，属性名称将显示在 `ASTViewer` 图形界面（图 3-4）的属性显示区的 `Property Name` 一栏中；属性值将显示在属性显示区的 `Property Value` 一栏中；属性指向的节点是通过 `ASTViewer` 属性显示区的 `Action` 一栏的按钮可以跳转到的 `AST` 节点，这个节点必须是当前 `AST` 节点的子节点。

表 3-4 类 `ASTViewPropertyItem` 中的方法

方法	说明
<code>ASTViewPropertyItem(String name, String value)</code>	构造器，创建形如<name, value, null>的属性条目
<code>ASTViewPropertyItem(String name, ASTNode astnode)</code>	构造器，创建形如<name, null, astnode>的属性条目
<code>ASTViewPropertyItem(String name, String value, ASTNode astnode)</code>	构造器，创建形如<name, value, astnode>的属性条目
<code>String getName()</code>	返回属性名称
<code>String getValue()</code>	返回属性值
<code>ASTNode getASTNode()</code>	返回属性指向的节点

你可以通过从类 `ASTViewPropertyDump` 派生新的类，如 `MyDump`，并且复写这个类中某些 `AST` 节点类型的 `dump` 方法，然后创建 `MyDump` 对象传递给 `ASTViewer` 对象来实现属性定制。例如，你希望 `ASTViewer` 能够显示 `CompilationUnit` 节点中的属性“author”（假设为 `String` 类型），可以定制如下的类

```
public class MyDump extends ASTViewPropertyDump {
    public List dump(CompilationUnit n) {
        String author = (String)n.getProperty("author");
        ASTViewPropertyItem att = new ASTViewProperty("Author", author);
```

```

        ArrayList attList = new ArrayList();
        attList.add(att);
        return attList;
    }
}

```

编译这个类，然后通过实验平台的命令行选项`--dump-class`（或`-D`）将它传递给 `ASTViewer`。如果你是在自己的代码中调用 `ASTViewer`，则可以创建 `MyDump` 的一个实例，将它通过 `ASTViewer` 构造器的第二个参数传递给 `ASTViewer`。

类 `ASTViewPropertyDump` 中的所有方法 `dump()` 都返回一个空的 `List`。如果你仅仅想对 `ASTViewer` 的缺省属性显示进行个别的修改，可以从类 `plugin.GenericPropertyDump` 派生自己的类。

## 3.6 设计模式

设计模式是用来帮助程序员交流面向对象软件设计中的经验。程序员一次又一次遇到相似的问题，相似的问题通常有相似的解决方案，尽管会有细节上的不同。因此，有经验的程序员可以识别一个熟悉的问题并且将一个熟悉的解决方案加以变化应用到这个问题上；而设计模式则是用来帮助新的程序员从这些经验中获利。

在本书的课程设计中，用到了两种设计模式：工厂方法(`Factory method`)和访问者(`Visitor`)。两者均出现在 `Eclipse AST` 中，只是你不直接使用工厂方法这种模式进行编程，而必须熟悉访问者模式的特征并掌握用它编程的方法。

根据 `Erich Gamma` 等编著的经典参考书《设计模式》，一个设计模式有四部分：

- 一个名字，我们可以方便地讨论它；
- 一个问题描述，让我们了解应该在什么时候使用它；
- 一个一般性的解决方案；
- 应用这个模式的效果。

下面我们将从这四个方面分别解释工厂方法和访问者这两个模式。当然，我们的描述是简单的，你可以参考设计模式书籍或者从互联网中找到更多的信息。不过，如果仅仅是为了完成课程设计，了解下面的介绍也就足够了，但是这需要你首先对面向对象语言的特性有一定的理解。

### 3.6.1 工厂方法模式

我们以 `Eclipse AST` 提供的工厂方法为例来解释这种模式。

#### 模式名

工厂方法。

#### 问题

设想一个庞大而复杂的产品类体系，比如各种 `Eclipse AST` 节点类。又设想我们需要创

建属于这个类体系的各种节点对象。传统的面向对象解决方案是，使用相应类的构造器来创建对象实例。为此，你需要事先了解这个复杂的类体系，识别出哪些是抽象的类，哪些是具体的类，类之间的继承关系是什么；你也需要将这些类绑定到你的代码中。例如，在 Eclipse AST 中，ASTNode 是所有节点类的抽象基类，Statement 是各种语句的抽象类，由它派生了 IfStatement、WhileStatement、ExpressionStatement 等具体的语句类。如果你需要创建这些语句的对象实例，你就需要将它们绑定到你的代码中。

### 解决方案

与其让使用者事先熟悉完整的产品类体系，不如在一个类中提供各种具体产品的创建方法。这个类称为**工厂类**，每一个创建具体产品的方法称为**工厂方法**。在 Eclipse AST 中，类 AST 就是这样的工厂类，它里面包含了许多工厂方法，如 newIfStatement( )、newWhileStatement( )、newExpressionStatement(*Expression*)等等。这样，当你需要为一个用类 Java 的语言编写的源程序创建 AST 时，你可以直接选择这个 AST 类中所提供的工厂方法来创建你所希望构造的 AST 节点，并且你可以将你的节点对象类型统一声明为 ASTNode。在 JDT AST 中，为了便于使用者将工厂方法和对应的节点类关联起来，工厂方法的命名采取“new+节点类名”的形式，这样使用者可以很快找到关联的节点类并利用这个节点提供的 set 方法来设置节点中的域。

### 效果

工厂方法不再将与各种 AST 节点有关的类绑定到你的代码中，代码仅处理 ASTNode 抽象类，因此它可以与任何具体的节点类一起使用。由于将创建节点对象的工厂方法集中在一个工厂类 AST 中，减轻了你事先了解完整的 AST 节点类体系的压力。

## 3.6.2 访问者模式

我们以 Eclipse AST 提供的访问者为例来解释这种模式。

### 模式名

访问者。

### 问题

设想一个庞大而复杂的类层次结构，比如 Eclipse AST 节点的层次。又设想我们需要实现这个层次结构中的一些操作，在我们的例子中，操作是对节点的解释或者是对节点的某种打印输出。传统的面向对象解决方案是，为每个操作在每个节点类中添加一个方法。例如，每个节点类都将会会有一个 interpret 方法。不幸的是，这种方法将各个操作的代码分散到许多类中，使得读者难以理解这些代码，并且也使不同操作的代码交织在一起。另一方面，由于 AST 类层次是由 Eclipse JDT 提供的，我们不想直接修改这些类，因为它们可能会随 Eclipse 版本的更新而被更改。如果我们直接修改这些类，那么每次 Eclipse 发布一个新版本时，我们将不得不把这些修改重做一遍。



## 解决方案

与其往类层次中的各个类中添加方法，不如为每个操作创建一个类。这个类叫做**访问者类**，在这个类中，类层次中的每一个类都有一个对应的方法。例如，我们可以建一个解释器类 `Interpreter`，其中每一种 AST 节点类型都有一个 `visit` 方法，这样 `Interpreter` 类看起来就是：

```
class Interpreter extends ASTVisitor {
    public boolean visit(Assignment n) {
        ...
    }
    public boolean visit(WhileStatement n) {
        ...
    }
    ...
}
```

这个类知道怎样用合适的方法来解释 AST 层次中的每个类。同时，所有的 AST 类层次的访问者类都从 `ASTVisitor` 派生，所以它们有统一的 `visit` 接口。现在，我们只需向 AST 层次中的每个类中添加一个 `accept` 方法，如下所示：

```
class WhileStatement {
    ...
    public void accept(ASTVisitor n) {
        n->visit(this);
    }
}
```

这样，我们就可以对 AST 类层次应用任何访问者了，并且我们不需要修改类层次中的各个类就可以创建新的访问者。

## 效果

访问者模式主要有两个好处：它使添加新操作变得容易，并且把每个操作封闭在单独的一块代码中；另一个较小的好处是访问者对象为在某种操作中维护状态提供了一个方便的场所。

访问者模式的主要缺点是向类层次中添加新类越来越难，添加一个新类需要向每个访问者添加一个新的方法。另一个缺点是为支持访问者，类层次中的类必须暴露足够的功能，这破坏了封闭性。例如，在一个标准的面向对象解决方案中，Eclipse 的开发者可能把一个语句块中的语句列表保持为私有数据。既然使用了访问者模式，他们必须创建一个 `public` 的 `statements()` 方法，把这些信息提供给访问者。

在我们的例子中，主要的缺点“访问者使添加新类变得困难”并不存在。因为 Java 语言（MiniJOOL 语言及其子语言也一样）有一个几乎不变的众所周知的规范，所以很少需要加入新的 AST 类。

Eclipse 的开发者预见到了 AST 访问者的需要，所以他们提供了一个访问者接口 `ASTVisitor`，并且为每个节点类型编写了 `accept` 方法。

使用 Eclipse AST 的访问者时应当意识到一些细节。

首先，对于访问者，`visit` 方法都返回 `void`。在 Eclipse 中，它们返回一个布尔值，用来控制 AST 的遍历。如果 `visit` 方法返回 `true`，`accept` 将会自动地访问当前节点的子节点。你需要对遍历顺序进行更细致的控制，所以你所有的 `visit` 方法都应返回 `false`。你的 `visit` 方法将通过调用子节点的 `accept` 函数来遍历 AST。

第二，`ASTVisitor` 是一个抽象类，所以如果你直接由它派生你自己的访问者类，你将需要为每个 AST 节点类型写一个 `visit` 函数，包括许多你在本课程设计中不需要的。你应该从 `GenericVisitor` 派生你自己的访问者类，它为每个节点类型实现了一个“什么也不做”的方法。

## 第4章 词法分析

词法分析的主要作用是根据语言的词法规则对输入的源程序字符流进行分析,识别出一个个的单词 (lexeme), 再将各单词对应的词法记号 (token) 依次提供给语法分析器, 这些记号将作为语言语法的终结符。

把词法分析从编译过程中独立出来可以大大降低编译器设计的复杂性, 减少编译程序的错误。词法分析器需要与语法分析器相协调, 它们需要有统一的词法记号类别编号, 并且有一致的单词获取以及信息访问接口, 等等。词法分析器应该提供给语法分析器足够多的单词信息, 以便语法分析器能够顺利进行分析和错误定位与处理。例如, 在我们的课程实验中, 要求把每个记号的位置信息传递给语法分析器。

在本章中, 你将学习为 MiniJOOOL 语言实现词法分析。MiniJOOOL 语言尽管比较小, 但是为它手工构造词法分析器的工作量已经比较大了。为了便于你掌握词法分析的各个方面的知识, 我们安排了四个独立的课程设计, 你可以根据实际情况, 选做其中的部分或全部。

### 4.1 本章课程设计概述

在本章中, 我们安排如下四个课程设计:

#### 课程设计 2-1 用 JFlex 为 MiniJOOOL 语言构造一个词法分析器。

在现代编译技术中, 有专门的词法分析器的生成器, 如 lex 及其各种变形, 它们可以从高级的词法规范文件生成词法分析器的源程序代码。由于我们使用 Java 作为课程实验的编程语言, 因此我们选择能生成 Java 源代码的词法分析器的生成器, 即 JFlex。通过这个实验, 你可以熟悉 JFlex 的使用, 了解 JFlex 词法规范文件的格式, 掌握为特定词法规则编写词法规范文件的方法。

#### 课程设计 2-2 手工编写一个简单的词法分析器。

在这个课程设计中, 你将学到词法分析的手工实现过程: 首先给出待词法分析的语言的正规定义; 然后由正规式画出状态转换图 (又称转换图); 再根据转换图编写词法分析程序。你需要在这个课程设计中掌握词法分析程序的框架, 注意词法状态的定义、识别和转换变迁, 以及它们是如何映射到完成词法分析的代码中。

#### 课程设计 2-3 编写一个非确定有限自动机(NFA)的生成器。

在这个课程设计中, 你将学到 AST 不仅可以表示程序, 还可以表示某些高级规范, 比如描述要词法分析器做什么的词法规范。你将学习用 AST 表示正规式, 再将这些正规式的 AST 翻译成 NFA。通过这个实验, 让你了解一个 NFA 生成器的构造方法, 为**课程设计 2-4**打下基础。

#### 课程设计 2-4 编写一个词法分析器的生成器。

课程设计 2-3 的不足是所生成的 NFA 没有被保存下来。课程设计 2-4 则要求实现由输入的词法规范文件生成词法分析器的源代码。这个词法分析器的生成器的功能几乎可以与诸

如 JFlex 这些著名的词法分析器的生成器相比较。在完成这样的一个工具后，可以使你对有限自动机和词法分析器原理的理解达到非常深的程度。

编写这样一个词法分析器的生成器是有些难度的，在我们给的实验指导中，会尽可能地帮助你理解其中的一些关键概念和技术，同时给出一些合适的例子，来减少你在实验中遇到的困难。

与上述课程设计相关的文件主要集中在本书提供的软件包的目录 `ROOT_DIR/lab/lab2` 中，该目录包括如下内容：

- **README:** 本章课程设计的说明文件。
- **.project 文件和.classpath 文件:** 本章课程设计的 Eclipse 工程文件。
- **bin 目录**
  - **build.xml**

管理 lab2 的 ant 编译文件，缺省执行的任务组是 `all`。`all` 任务组分别用 `lab2-1.xml`、`lab2-2.xml`、`lab2-3.xml`、`lab2-4.xml` 四个编译文件来运行 ant，从而编译并运行课程设计 2-1、2-2、2-3 和 2-4。此外，编译文件中还包括独立编译并运行每个课程设计的任务组 `lab2-1`、`lab2-2`、`lab2-3`、`lab2-4`，以及 1 个由注释生成 HTML 文档的 `doc` 任务组。
  - **lab2-1.xml:** 配合 build.xml 管理课程设计 2-1 的 ant 编译文件。
  - **lab2-2.xml:** 配合 build.xml 管理课程设计 2-2 的 ant 编译文件。
  - **lab2-3.xml:** 配合 build.xml 管理课程设计 2-3 的 ant 编译文件。
  - **lab2-4.xml:** 配合 build.xml 管理课程设计 2-4 的 ant 编译文件。
- **src 目录**

存放你实验用到的一些 Java 源文件，你需要对其中的部分源文件补充代码。这些 Java 源文件的包路径均为 `edu.ustc.cs.compile.lexer`，具体包括：

  - **Symbol.java:** 课程设计 2-1 到 2-4 中使用的词法记号类，你需要补充记号来完成你所实现的词法分析器。
  - **Lexer.java:** 课程设计 2-2 中词法分析器的抽象基类，其中声明有抽象方法 `nextToken`。
  - **ExpressionLexer.java:** 手工编写的简单词法分析器，从类 `Lexer` 继承，你需要在这个类中补充代码完成课程设计 2-2。
  - **UnmatchedException.java:** 课程设计 2-2 中使用的异常类。
  - **NFAGenerator.java:** 一个由词法规范对应的 AST 生成 NFA 的访问者类。你需要在这个类中补充代码完成课程设计 2-3。
  - **NFAState.java:** 课程设计 2-3 和 2-4 中使用的 NFA 状态类，支持 NFA 状态及转换。
  - **NFASimulator.java:** 课程设计 2-3 和 2-4 中运行 NFA 的模拟器。它接收一个 NFA 和一个测试文件，运行该 NFA 对测试文件进行词法分析。你不需要修改这个文件。

- **LexerCodeGenerator.java:** 词法分析器的生成器类。你需要在这个类中补充代码使其根据 AST 来生成对应的词法分析器，从而完成课程设计 2-4。
- **LexerCode.java.sample:** 在课程设计 2-4 中词法分析器的生成器生成的文件的示例。
- **Main.java:** 本章课程设计的测试总控类，在 `main` 方法中调用 `runPart2()`、`runPart3()`、`runPart4()`可以分别启动运行课程设计 2-2、2-3 和 2-4。
- **config 目录**  
存放本章课程设计所需的词法规范文件。
  - **JFlex/sample.flex:** 课程设计 2-1 中使用的描述英文单词和整数的 JFlex 词法规范文件。
  - **MLex/MiniJOOL.mlex:** 课程设计 2-3 和 2-4 中使用的描述 MiniJOOL 语言的 MLex 词法规范文件。
- **test 目录**  
收集一些测试你所编写的词法分析器是否有效的程序。你可以在这个目录下，补充更多的测试用例。

MiniJOOL语言的词法描述见 2.2 节。在以下各节中，我们先依次介绍四个课程设计的任务并给出相应的提示和指导，最后在 4.6 节介绍JFlex工具。

## 4.2 课程设计 2-1: 用 JFlex 为 MiniJOOL 语言生成一个词法分析器

在这个课程设计中，你将用 JFlex 为 MiniJOOL 语言生成一个词法分析器。这是选做内容，可以跳过，不过完成这个课程设计有助于加深对词法分析器及其生成器工作原理的理解。

### 4.2.1 示例

我们先介绍用 JFlex 生成一个简单的词法分析器的例子，这个词法分析器可以识别由英文字母组成的单词和由数字组成的整数。这个示例帮助你熟悉 JFlex，为开展自己的课程实验做准备。

先看一下这个示例相关联的文件：

- **config/JFlex/sample.flex:** JFlex 词法规范文件，描述单词和整数的词法构成。
- **bin/build.xml 和 bin/lab2-1.xml:** 总控的 ant 编译文件以及管理课程设计 2-1 的子编译文件。编译文件中描述了对课程设计 2-1 的相关代码进行 jflex 生成、编译、连接、运行的过程。针对这个示例，lab2-1.xml 是用来控制由 sample.flex 生成词法分析器的源代码 SampleLexer.java (存放在 `src/edu/ustc/cs/compile/lexer` 目录下)，并编译、运行所生成的词法分析器。
- **src/edu/ustc/cs/compile/lexer/Symbol.java:** 一个包含词法记号定义的文件。生成的词法分析器要用到这个词法记号类。

在这个示例实验中，需要用到 JFlex 工具，你需要确认它是否已经安装和配置好(参见

1.4.1 节)。

接下来的工作分两步走：一是编写词法规范文件 `sample.flex`；二是对 `sample.flex` 运行 JFlex，产生词法分析器的 Java 源代码，接着编译这个词法分析器，再运行这个词法分析器进行测试。我们将在下面两小节中分别叙述。

#### 4.2.1.1 编写 JFlex 词法规范文件

按照 JFlex 的词法规范文件格式，可以编写英文字母序列和整数的词法规范文件 `sample.flex`。其内容如下：

```
package edu.ustc.cs.compile.jflex;

%%

%line

%column

%public

%class SampleLexer

%type Symbol

%eofval{
return Symbol.EOF;
%eofval}

%{
    public static void main(String argv[]) {
        if (argv.length != 1) System.out.println("Usage: java Lexer inputfile");
        else {
            SampleLexer l=null;
            try {
                l = new SampleLexer(new java.io.FileReader(argv[0]));
                Symbol s = l.yylex();
                while (s != Symbol.EOF) {
                    System.out.println(s);
                    s = l.yylex();
                }
            } catch (Exception e) {
                System.out.println("Unexpected exception:");
                e.printStackTrace();
            }
        }
    }
}
```

```

%%
[:digit:]+ { return new Symbol(Symbol.INT, yytext(), yyline, yycolumn); }
[:letter:]+ { return new Symbol(Symbol.WORD, yytext(), yyline, yycolumn); }
.\n      { }

```

图 4-1 sample.flex 文件中的内容

JFlex的词法规范文件格式详见 4.6 节，这里结合sample.flex简要说明。整个词法规范文件分为三个部分，各部分之间用“%%”隔开。

第一部分是**用户代码**。JFlex 将把这部分代码复制到生成的类文件的最开始。在这个示例中，声明了一个 package；如果有必要的话，还可以加上 import 导入语句。

第二部分是**选项与声明**。其中，选项用来定制你希望生成的词法分析器，而声明则是一些能在第三部分使用的词法状态和宏(Macro)定义的声明。在这个示例中，只包含了一些选项。下面分别对示例中出现的各个选项进行解释：

在 JFlex 生成的词法分析器类中，最重要的方法是 `yylex()`，它对源文件进行扫描并返回下一个词法记号。`%type` 选项用来指定这个 `yylex()` 方法所返回的记号类型，在本例中为 `Symbol`。`Symbol` 是我们提供的描述一个词法记号的类，在这个类中包含 `type`、`token`、`line` 和 `column` 四个数据成员，依次表示记号的类型、记号对应的串值、记号在输入文件中的行、列位置。在 `Symbol` 中还定义了各种可能记号类型常量，如 `PLUS`、`IDENTIFIER` 等。

`%class` 指定生成的词法分析器类的名字，本例中为 `SampleLexer`。

`%public` 声明生成的类为 `public` 访问权限。

`%line` 和 `%column` 分别把行和列的计数打开，这样就可以通过 `yyline` 和 `yycolumn` 这两个变量获得当前记号在输入文件中的行、列位置了。例如，你可以用 `new Symbol(Symbol.INT, yytext(), yyline, yycolumn)` 来为当前识别出的整型记号产生一个 `Symbol` 对象，其中记号的串值可以通过调用 `yytext()` 获得。

`%eofval{}`和`%eofval{}`之间的代码是在分析器遇到文件末尾的时候执行的。在本例中，直接返回一个在 `Symbol` 类中定义的 EOF 值。

`%{}`和`%}`之间的代码将被直接复制到生成的词法分析器类中。在本例中，我们为词法分析器添加一个 `main` 方法，它通过参数 `argv` 来获取希望被词法分析的输入文件名，然后对这个文件进行词法分析，再逐个打印输出词法分析器所返回的记号。

第三部分是**词法规则**。本例中只有三条规则，正规式“`[:digit:]+`”表示 1 个或多个十进制数字组成的串，正规式“`[:letter:]+`”表示 1 个或多个字母组成的串，正规式“`.\n`”表示所有的字符。每个正规式右边由一对花括号括起的部分表示词法匹配后要执行的动作（Java 代码）。在本例中，前两条规则都是简单地当前识别出的词法记号构造一个 `Symbol` 对象再将该对象返回。

你可以参考 4.6 节或者 JFlex 用户手册，获得更多关于 JFlex 词法规范的说明。在 JFlex 软件包中的 `examples` 子目录下有更多的示例，如果想深入了解 JFlex 可以作为参考。

### 4.2.1.2 词法分析器的生成、编译和运行

在编写完 sample.flex 后, 就可以运行 JFlex 生成这个词法规范文件所对应的词法分析器源代码了。

为了便于运行 JFlex, 我们提供了一个 build.xml 和 lab2-1.xml。你可以在 Eclipse 中用鼠标选中 bin/build.xml 并按右键, 然后执行“Run as”→“Ant Build...”菜单项。在弹出的如图 4-2 的窗口中, 你可以选择要执行的任务组, 这里你需要选中 lab2-1, 然后按“Run”按钮执行 ant。

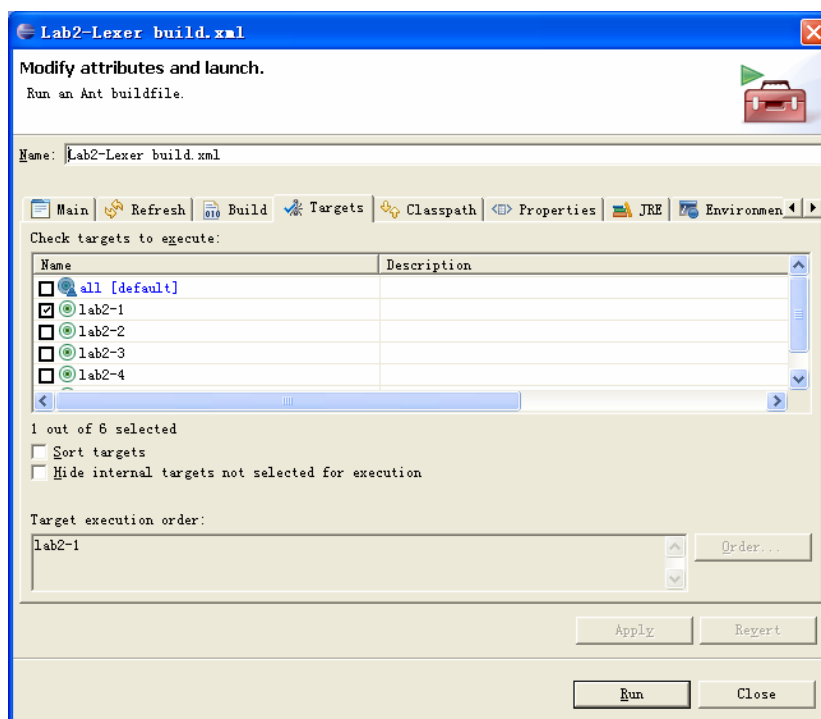


图 4-2 sample.flex 文件中的内容

如果配置正确, 在 Eclipse 的“Console”视窗中就可以看到如下的结果了。

Buildfile: E:\CompilerProj\student\lab\lab2\bin\build.xml

lab2-1:

all:

jflex:

[java] Reading

"E:\CompilerProj\student\lab\lab2\config\JFlex\sample.flex"

[java] Constructing NFA : 12 states in NFA

[java] Converting NFA to DFA :

[java] .....

[java] 7 states before minimization, 4 states in minimized DFA

[java] Old file

"E:\CompilerProj\student\lab\lab2\src\edu\ustc\cs\compile\lexer\SampleLexer.java" saved as

"E:\CompilerProj\student\lab\lab2\src\edu\ustc\cs\compile\lexer\SampleLexer.java~"



```

[java] Writing code to
"E:\CompilerProj\student\lab\lab2\src\edu\ustc\cs\compile\lexer\SampleLexer.java"
build:
[javac] Compiling 1 source file to
E:\CompilerProj\student\lab\lab2\classes
run:
[java] 2 3    (0, 0)
[java] 2 52342490  (1, 0)
[java] 2 234    (2, 2)
BUILD SUCCESSFUL
Total time: 6 seconds

```

你也可以在命令行方式下进入 ROOT\_DIR/lab/lab2/bin 目录，键入如下命令来运行：

```
ant lab2-1
```

或 ant -buildfile build.xml lab2-1

其中，“-buildfile”选项可以简写成“-file”或者“-f”。不过，你需要确保 ant 所在的目录和 classpath 路径等已被正确地配置到系统的环境变量中。

#### 4.2.1.3 使用的 ant 编译文件

下面对示例中使用的 build.xml 以及 lab2-1.xml 中的相关内容进行简要说明。

##### 1、build.xml

build.xml 是本章课程设计的总控编译文件。其中定义了一组表示各种路径信息的属性，如 BIN\_DIR 等，这些属性可以在诸如 lab2-1.xml 的子编译文件中使用。all 任务组是 build.xml 的缺省任务组，它依次执行 lab2-1、lab2-2、lab2-3、lab2-4 任务组。其中，lab2-1 任务组的定义如下：

```

<target name="lab2-1">
    <ant antfile="{BIN_DIR}/lab2-1.xml" inheritAll="true"/>
</target>

```

ant 元素表示执行 ant 命令，其中的 antfile 属性用来设置执行所依赖的编译文件，inheritAll 属性设置为 true 表示在用 ant 执行 lab2-1.xml 时继承在 build.xml 中定义的所有属性。这样，在 lab2-1.xml 中就没有必要重复声明属性；但是由于依赖 build.xml 中的属性，在单独执行“ant -f lab2-1.xml”命令时，也会发生一些错误。

##### 2、lab2-1.xml

第 1 行是 project 元素的开始标签，其中的 name、default 和 basedir 属性分别指定该 ant 编译项目的名字、缺省要执行的任务组以及工作的根目录。

```
<project name="lab2-1" default="all" basedir="..">
```

all 任务组会通过 antcall 命令依次调用名为 jflex、build 和 run 的任务组：

```
<target name="all">
```

```

<antcall target="jflex"/>
<antcall target="build"/>
<antcall target="run"/>
</target>

```

jflex 任务组利用 java 命令执行 jflex，其中在 java 元素内可以添加若干 arg 子元素设置运行 jflex 时的命令行参数。在执行这个任务组后，将在指定的\${SRC\_DIR}/\${LEXER\_DIR}目录下，生成 SampleLexer.java。

```

<target name="jflex">
  <java fork="yes" jar="${JFLEX}">
    <arg value="-v"/>
    <arg value="-d"/>
    <arg value="${SRC_DIR}/${LEXER_DIR}"/>
    <arg value="${JFLEX_CFG}"/>
  </java>

```

build 任务组利用 javac 命令编译词法分析器源文件，即 Symbol.java 和 SampleLexer.java。

```

<target name="build" description="编译">
  <mkdir dir="${DEST_DIR}"/>
  <javac debug="on" destdir="${DEST_DIR}" srcdir="${SRC_DIR}"
    includes="${LEXER_DIR}/Symbol.java, ${LEXER_DIR}/SampleLexer.java"/>
</target>

```

run 任务组利用 java 命令运行词法分析器，其中的 arg 子元素的 value 属性设置待词法分析的文件名为\${TEST\_DIR}/expr.txt。你可以修改这个 value 属性值来设置你希望词法分析的文件。

```

<target name="run" description="运行示例分析器">
  <java classpath="${CLASSPATH}"
    classname="edu.ustc.cs.compile.lexer.SampleLexer">
    <arg value="${TEST_DIR}/expr.txt"/>
  </java>
</target>

```

除了用 ant 来运行以外，你还可以直接执行由 jflex 生成的词法分析器类 SampleLexer 来对输入的文件进行词法分析，依次得到识别出的各个词法记号。

#### 4.2.2 用 JFlex 为 MiniJOOL 进行词法分析

参考 4.2.1 节所讲的例子和 4.6 节的 JFlex 词法规范或者 JFlex 的用户手册，你就可以着手为 MiniJOOL 语言生成一个词法分析器。分析器类的名字规定为 MJLexer，与 SampleLexer 在同样的 package 中，即为 edu.ustc.cs.compile.lexer。Symbol 类仍为我们所提供的，但是你需要对它进行扩充，使之能够表示 MiniJOOL 语言中的所有记号。

你需要为 MiniJOOL 语言编写一个 JFlex 词法规范文件 MiniJOOL.flex, 并将这个文件放在 lab2/config/JFlex 目录下。之后, 你需要修改 lab2-1.xml, 为 MiniJOOL.flex 编写相应的词法分析器生成、编译和运行的任务组, 并在 all 任务组中调用这些任务组。

你还需要在 test 目录中增加一些测试用例来测试你的 MiniJOOL 词法分析器。在设计这些测试用例时, 你应该对照 2.2 节所描述的词法特点进行。需要注意的是对注释的测试, 你要认真检查词法分析器的输出, 注意注释有没有分析完整, 有没有“吃掉”非注释代码。另外, 你还要检查对字符和字符串的分析, 看转义字符是否能正确地分析出来。

关于在 Eclipse 或控制台下如何生成、编译和运行 MiniJOOL 词法分析器, 可以参见 4.2.1.2 节。

### 4.3 课程设计 2-2: 手工编写一个简单的词法分析器

在这个课程设计中, 你将要手工编写一个识别表达式的词法分析器。课程设计的目的在于让你通过实现这个简单的分析器, 来了解如何将转换图编写成一个词法分析程序, 从而对词法分析器的运行机理有更深入的理解, 为后面的实验打下基础。

#### 4.3.1 示例

我们同样先给出一个简单的例子来说明手工实现词法分析器的思路。在这个例子里, 要识别的词法记号是整数和加号。

首先看一下这个示例关联的文件 (均在 src/edu/ustc/cs/compile/lexer 目录) 下:

- **Lexer.java** 词法分析器的抽象基类, 其中有抽象方法 nextToken;
- **ExpressionLexer.java** 手工实现能识别整数和加号的词法分析器类, 它由 Lexer 派生, 需要实现 nextToken 方法;
- **Symbol.java** 词法分析器类 ExpressionLexer 中要用到这个词法记号类文件。
- **Main.java** 测试总控类, 在这个类的 main 方法里调用 runPart2( ) 可以运行 ExpressionLexer 词法分析器。
- **UnmatchedException.java** 一个异常类, 表示不匹配的异常。

接下来, 我们来说明怎样手工实现能识别整数和加号的词法分析器 ExpressionLexer。

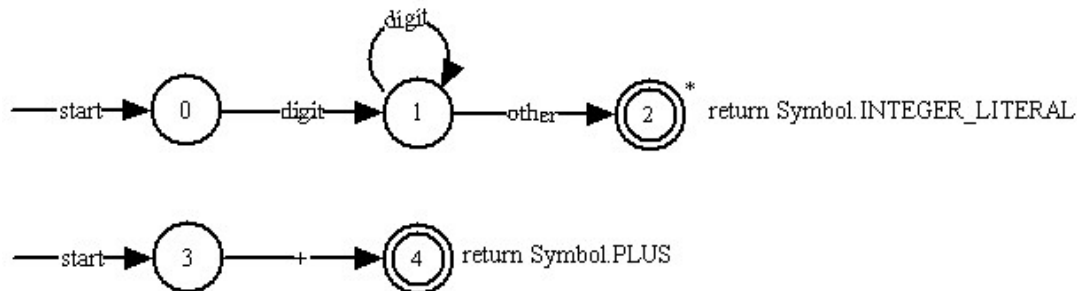


图 4-3 识别整数和加号的转换图

第一步, 分别写出整数和加号的正规式定义:

int → digit+

plus → '+'

第二步，根据正规式画出对应的转换图，如图 4-3 所示。其中双圈表示接受状态，双圈右上角的星号表示到达该接受状态之后需要回退一个字符到输入流中去。

第三步，根据转换图编写词法分析器类 `ExpresssionLexer.java`，其中的关键是 `nextToken` 方法(如图 4-4 所示)的实现。

由图 4-3 的转换图可知，应该根据当前的状态和输入的字符来决定下一个状态。在 `nextToken` 方法中，有一个无限的 `while` 循环，该循环在分析到匹配转换图的单词或找不到匹配时退出。对于转换图中的每一个状态，在 `while` 循环中都对应有一段代码，即 `switch` 语句中的一个 `case`。在该段代码中，首先获取输入中的下一个字符(即调用方法 `nextChar` 而把下一个字符保存在成员变量 `c` 中)；然后对字符变量 `c` 进行判断：如果它与转换图中当前状态的某一出边上的字符相匹配，则在代码中将当前状态(即 `state` 变量)设置为该出边所到达的状态，接着执行下一次循环；如果字符变量 `c` 在转换图上没有相匹配的出边，则调用方法 `fail`，得到一个新的开始状态，再继续进行分析。如果到达接受状态，则词法分析器将返回该接受状态对应的记号，在返回前需要把词法分析器类中的相关变量重置(见 `getToken` 方法)，必要时回退一个字符到输入流中(即调用 `pushbackChar` 方法)。

```
public Symbol nextToken() {
    while (true) {
        switch (state) {
            case 0:
                // 获取下一个字符
                nextChar();

                // 判断是否空白
                if (Character.isWhitespace(c)) {
                    if (c == '\n') {
                        line++;
                        column = 0;
                    }
                    dropChar();
                } else if (Character.isDigit(c)) {
                    state = 1;
                } else if ((c & 0xff) == 0xff) {
                    return getToken(Symbol.EOF);
                } else {
                    state = fail();
                }
                break;
            case 1:
                nextChar();
                if (Character.isDigit(c)) {
```

```

        state = 1;
    } else {
        state = 2;
    }
    break;
case 2:
    pushbackChar();
    return getToken(Symbol.INTEGER_LITERAL);
case 3:
    nextChar();
    if (c == '+') {
        state = 4;
    } else {
        state = fail();
    }
    break;
case 4:
    return getToken(Symbol.PLUS);
}
}
}

```

图 4-4 ExpressionLexer.java 中的 nextToken 方法

图 4-5 给出的fail方法只为每个转换图的开始状态准备了一段代码，该代码根据词法分析器类的成员变量start（指示词法分析器初始应该在哪个状态中）的值，将其设置为下一个转换图的开始状态。如果当前的start已经是最后一个开始状态，这时就需要调用错误恢复方法来处理，在这个示例中，只是简单地抛出一个异常。

```

private int fail() {
    while (lexeme.length() > 0)
        pushbackChar();
    switch (start) {
    case 0:
        start = 3;
        break;
    case 3:
        recover();
        break;
    default:
        throw new UnmatchedException();
    }
    return start;
}

```

图 4-5 ExpressionLexer.java 中的 fail 方法

### 4.3.2 课程设计任务

在理解 4.3.1 中的例子的基础上，你可以着手继续完善 `ExpresssionLexer.java`，使它能够分析一个或多个表达式。要分析的记号有：数字、赋值号、变量（以字母开头，后跟 0 个或多个字母、数字或下划线）、运算符（算术四则运算）、括号、分号等，你需要能跳过输入中的单行注释或多行注释。

要求你应该：

- 先给出这些记号的正规定义；
- 然后作出转换图；
- 编写 `ExpresssionLexer` 代码并调试；
- 编写 6 个以上的典型测试用例测试你的词法分析器，要求测试用例中不仅有正确的表达式，而且还有错误的表达式，说明你的测试用例的设计考虑。

### 4.3.3 编译和运行指南

在 `Main.runPart2()` 方法中，缺省打开源文件 `"e:/CompilerProj/student/lab/lab2/test/expr.txt"` 并对它进行词法分析。你可以修改使之变成你所希望分析的源文件名。

下面分别简述在 Eclipse 和命令控制台上编译、运行本课程设计的方法。

#### 1、Eclipse 环境

在 Eclipse 中，用鼠标选中 Lab2 工程的 `Main.java`，按鼠标右键并选择菜单项“Run As”→“Java Application”，即可以执行词法分析。

**expr.txt 的内容:**

```
3 +
52342490
+ 234
```

**对 expr.txt 的词法分析结果:**

```
2 3    (1, 1)
3 +    (1, 3)
2 52342490 (2, 1)
3 +    (3, 1)
2 234  (3, 3)
```

图 4-6 expr.txt 及其词法分析结果

图 4-6 显示了 `expr.txt` 的内容以及词法分析器对这个文件进行词法分析所识别出的各个记号。针对每个记号，将输出该记号所属的类别编号、对应的文本串、以及所处的行号和列号；其中记号的类别编号值与你当前使用的 `Symbol` 类中的定义有关，未必是这里的 2(表示整数)或 3(表示加号)。

#### 2、在命令控制台下

你可以在命令控制台下进入 `ROOT_DIR/lab/lab2/bin` 目录，键入如下命令来编译和运行：

```
ant lab2-2
```

## 4.4 课程设计 2-3: 编写一个 NFA 生成器

课程设计 2-3 是编写一个 NFA 生成器。为了简化课程设计的负担，我们将这个实验分

成四步：

- 1) 由一个词法规范文件产生一个抽象语法树 AST。这个词法规范文件的格式由我们定义，记为 MLex 词法规范描述语言，文件的扩展名约定为 `mlex`。
- 2) 编写一个 AST 的访问者类 `NFAGenerator`，为 1) 中得到的 AST 创建 NFA。
- 3) 编写一个 `NFASimulator` 类，它负责用给定的 NFA 分析输入文件，识别出词法记号。
- 4) 编写一个总控程序，用它来控制上面 3 步的顺次执行，最终测试所生成的 NFA 是否能正确识别词法记号。

你只需要自己实现第 2) 步，而表示 NFA 状态的 `NFAState` 类以及其余三步均由我们来提供实现。其中第 1) 步的 MLex 词法规范分析器为 `edu.ustc.cs.compile.parser.mlexspec.Parser`，由类库 `ROOT_DIR/lib/edu.ustc.cs.compile.parser.mlexspec.jar` 提供，其余关联的文件有：

- **Main.java** 测试总控类，在 `main` 方法中调用 `runPart3()` 可以运行这个实验。
- **Symbol.java** 词法分析器类中要用到这个词法记号类文件。
- **NFAGenerator.java** 由 AST 生成 NFA 的访问者类，我们提供了基本的框架，你需要在这个类中补充代码使其产生 NFA。
- **NFAState.java** NFA 状态类。支持 NFA 状态的管理及状态转换。
- **NFASimulator.java** 用于测试运行 NFA 的模拟器，它以一个 NFA 和一个用于测试的源文件为输入。
- **config/MLex/MiniJOOL.mlex** 一个 MiniJOOL 语言的 `mlex` 词法规范文件。

你可以修改类 `Main` 中的成员变量 `file` 的值，设置要词法分析的源文件名。

为了理解要开展的课程设计，我们首先介绍这里使用的 `Mex` 词法规范文件的格式，然后明确课程设计的任务。

#### 4.4.1 MLex 词法规范文件的格式

在这个课程设计中，我们将词法规范文件转换成用 Eclipse AST 类层次描述的 AST，来简化我们的编码工作。因此，为了实现和理解上的方便，我们采用与 Java 非常相似的语法来定义词法规范文件的格式，这个词法规范文件的扩展名约定为 `mlex`。尽管如此，由这个词法规范文件生成的 AST 还是与第二章中根据 `SimpleMiniJOOL` 源代码生成的 AST 在含义上是有区别的。

下面通过一个例子来了解 MLex 词法规范文件的格式：

```
class Lexer {
    static void main() {
        //宏定义
        ws = ' ' | '\n' | '\r' | '\t';
        digit = '0' - '9';
        alpha = 'a' - 'z' | 'A' - 'Z';
        id = (alpha) + (digit | alpha) * 0;
```

```

// 规则
if("class")    { return CLASS; }
if("static")   { return STATIC; }
if(id)         { return IDENTIFIER; }
if(digit * 1)  { return INTEGER_LITERAL; }
}
}

```

一个MLex词法规范描述文件由一个只含一个方法的类组成。方法体由两部分构成，第一部分是宏定义，它通过赋值号定义常用的模式，例如，十进制数字的集合`digit = '0' - '9'`，表示把'0'到'9'的范围赋给宏名"digit"，可以利用宏构造更复杂的正规式。第二部分是词法规则，使用if语句来定义，要匹配的正规式写在if的条件中，词法记号的类型用if语句的then块中的return语句来指定，你可以假定在if语句体内只有一个return语句而没有别的语句。正规式可以使用表4-1所示的形式。

表 4-1 正规式的形式及其含义

正规式	含义
'a'	匹配单个字符
'a'-'z'	匹配 ASCII 码值在两个字符之间的字符范围，包括这两个字符
"string"	匹配一个字符串
rx1   rx2	" "是或运算符，匹配 rx1 或者 rx2
rx1 + rx2	"+"是连接运算符，匹配 rx1 后紧跟 rx2
rx * 0	匹配零个或多个 rx
rx * 1	匹配一个或多个 rx
macro	匹配 macro 定义的正规式

**注意：**正规式定义的顺序是有影响的。定义在前面的正规式应该比后面的优先级高。

另外，规则部分中的 return 语句返回的是一个记号，如“CLASS”，返回的这些记号需要在词法记号类 Symbol 中添加定义。例如针对上述的词法规范，Symbol.java 中应该有如下常量成员定义：

```

public static final int INTEGER_LITERAL = counter++;
public static final int IDENTIFIER = counter++;
public static final int CLASS = counter++;
public static final int STATIC = counter++;

```

在 Symbol.java 中使用一个计数器变量 counter 为这些记号赋值，这样可以方便地把记号归类，并且容易扩展增加新的记号。



### 4.4.2 课程设计指导

类 `edu.ustc.cs.compile.parser.mlexspec.Parser` 是我们提供的用于分析 `MLex` 词法规范文件的分析器，其实例方法 `doParse()` 方法可以把输入的词法规范文件转换成 AST，如：

```
File in = new File("config/MLex/MiniJOOL.mlex");
HIR ir = null;
try {    ir = (HIR)(parser.doParse(in));
} catch (ParserException e) {
    .....
}
```

接下来你可以考虑 `NFAGenerator.java` 的实现了。首先你需要了解 NFA 的表示，然后你要解决对 AST 中宏定义和词法规则的处理，其中涉及到 NFA 状态以及状态转换的构造、宏和记号的处理等。

#### 4.4.2.1 NFA 的表示

我们将一个 NFA 表示成一个图，并且 NFA 中的每一个状态用一个 `NFAState` 对象表示。在 `NFAState` 类中，主要有以下成员变量：

```
private boolean isFinal; /** 该 NFA 状态是否是接受状态 */
private int priority;    /** 接受状态的优先级，数越大则优先级越高 */
private int tokenId;     /** 接受状态的记号 id，即 Symbol.java 中对应的编号值 */
private int id = idCounter++; /** 这个 NFA 状态的 id */
private Set[] transitions = new Set[256]; /** 状态转换表 */
private static ArrayList stateList = new ArrayList(); /** 所有 NFA 状态的列表 */
```

需要说明的是，为便于快速处理状态转换，这里将转换表 `transitions` 定义为一个包含 256 个元素的数组，数组的下标即为面临输入的字符的 ASCII 码值，而对应的元素则是一个集合（即一个 `LinkedHashSet` 容器对象），表示该状态在面临输入的字符时可以转换到的状态集合。

在 `NFAState` 类中，还定义了一个 `static` 类型的 `ArrayList` 对象 `stateList`，这个对象保存当前 NFA 的所有状态。

#### 4.4.2.2 NFA 生成的一般方法

在 `MLex` 词法规范所对应的 AST 中，主要包括以下几类 AST 节点：`TypeDeclaration`、`MethodDeclaration`、`IfStatement`、`Assignment`、`ReturnStatement`、`InfixExpression`、`StringLiteral`、`CharacterLiteral`、`SimpleName` 和 `Block`。为了构造 NFA，你需要在 `NFAGenerator` 类中为每类 AST 节点实现一个 `visit` 方法。

词法记号及其正规式分别在 `if` 语句的 `then` 分支以及表达式中描述。NFA 就是根据 `if` 语句中的表达式来生成的。在遍历表达式对应的 AST 时，应该采用后序遍历的策略，即对某一个节点，先访问子节点生成子节点对应的 NFA，然后再根据该节点的类型来生成包含子

节点 NFA 的 NFA。

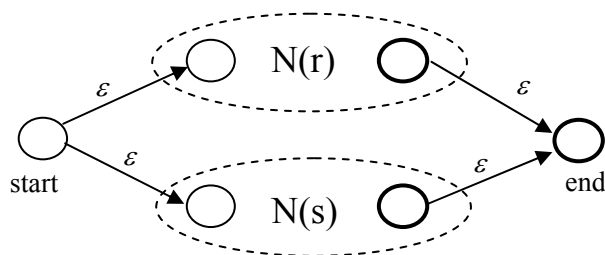


图 4-7  $r | s$  对应的 NFA

例如，某节点为正规式  $r | s$  对应的 AST 的根节点，则该节点为 `InfixExpression` 对象，运算类型是 `InfixExpression.Operator.OR`，假设遍历左子树  $r$  时生成  $r$  的 NFA 为  $N(r)$ ，遍历  $s$  时生成  $s$  的 NFA 为  $N(s)$ ，则为  $r | s$  生成的 NFA 如图 4-7 所示。由图可见，需要为正规式  $r | s$  创建一个开始状态 `start` 和结束状态 `end`，然后在 `start` 状态的  $\epsilon$ （定义为 `NFAState.EPSILON`）转换中分别添加  $N(r)$  和  $N(s)$  的开始状态，再分别在  $N(r)$  和  $N(s)$  的结束状态的  $\epsilon$  转换中添加 `end` 状态。

在 `NFAGenerator` 类中，可以定义两个成员变量来保存当前节点对应的 NFA 的开始状态和结束状态，如 `tempStart` 和 `tempEnd`，这样在子节点访问结束后，就可以对子节点对应的 NFA 进行操作，构造当前节点的 NFA，然后把这两个成员设置为新 NFA 的开始与结束状态。例如，针对前面的  $r | s$  正规式，在访问它对应的 `InfixExpression` 节点时，可以有如图 4-8 的处理：

```
public boolean visit(InfixExpression s) {
    NFAState ls, le, rs, re;
    InfixExpression.Operator op = s.getOperator(); // 取得运算类型
    if (op.equals(InfixExpression.Operator.OR)) { // 或运算
        s.getLeftOperand().accept(this);           // 访问左操作数
        ls = tempStart;                             // 取得左操作数 NFA 的开始状态
        le = tempEnd;                                // 取得左操作数 NFA 的结束状态
        s.getRightOperand().accept(this);           // 访问右操作数
        rs = tempStart;                             // 取得右操作数 NFA 的开始状态
        re = tempEnd;                                // 取得右操作数 NFA 的结束状态

        tempStart = new NFAState();                // 为当前节点创建开始状态
        tempEnd = new NFAState();                  // 为当前节点创建结束状态
        tempStart.addTransition(NFAState.EPSILON, ls); // 添加状态转换
        tempStart.addTransition(NFAState.EPSILON, rs);
        le.addTransition(NFAState.EPSILON, tempEnd);
        re.addTransition(NFAState.EPSILON, tempEnd);
    }
}
```

```

.....    // 其他运算类型的处理
}

```

图 4-8 对 InfixExpression 节点的访问处理

#### 4.4.2.3 宏对 NFA 生成的影响

在生成 NFA 时，比较麻烦的地方是对宏的处理。一种错误的处理思想是，当访问 AST 遇到宏定义（即遇到 Assignment 节点）时，就为这个宏生成对应的 NFA，并把这个宏及其 NFA 的开始状态与结束状态保存到映射表（Map）中。之后，如果某 AST 节点是对宏的引用（即遇到 SimpleName 节点），则直接从映射表中取出该宏对应的 NFA 的开始状态与结束状态。这种方案乍看没有什么问题，但是仔细思考就会明白这其中含有错误。

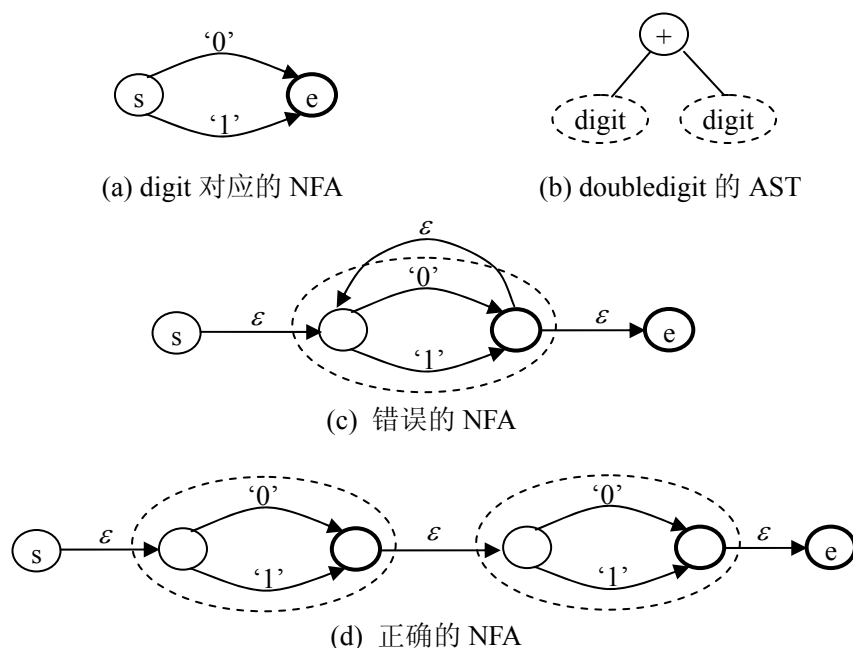


图 4-9 宏引用的处理

设想我们定义了一个宏：

`digit = '0' | '1';`

来表示二进制数字并为它生成了 NFA，如图 4-9(a)所示。又设想要定义一个新的宏表示两位二进制数，即

`doubledigit = digit + digit;`

那么 `doubledigit` 对应的 AST 如图 4-9(b)所示。对于连接运算，要从左操作数对应的 NFA 的结束状态引出一条  $\epsilon$  边到右操作数对应的 NFA 的开始状态。为此，首先访问两个子节点，分别得到它们对应的 NFA 的开始和结束状态。注意，由于两个子节点引用了同一个宏，因此取得的开始和结束状态分别是相同的。当做连接操作后，得到对应的 NFA 如图 4-9(c)，这显然是错误的，它实际上是正规式 “`digit * 1`” 的 NFA。

为了避免上面的错误，需要另外寻找一个正确的解决办法。一种较易理解的方案是遇到宏定义时先不构造 NFA，而是在映射表中保存宏名到它对应的正规式 AST 的映射。当引用

该宏时，则从映射表获得对应的 AST，然后遍历这个 AST，构造 NFA。这样，针对同一个宏的不同位置的引用，就可以构造出多个 NFA 对象，从而避免了前面所提到的问题。图 4-9(d) 给出了采用这种方案构造出的正确 NFA。

在 NFAGenerator 类中，需要引入一个宏映射表来管理词法规范中所定义的各个宏。根据上一小节所提的解决方案，宏映射表的每一条目应至少包含宏名、该宏对应的正规式 AST。此外，由于 mlex 词法规范规定正规式是有优先级的，即定义在前面的正规式应该比后面的优先级高，因此宏映射表的每一条目还需要记录该宏的优先级别。

#### 4.4.2.4 记号的处理

当访问 AST 遇到 IfStatement 节点时，由 if 表达式子节点的访问可以获得记号对应的 NFA，由 then 分支中的 ReturnStatement 语句可以获得这个 NFA 的所识别出的记号对应的名称。这时，你需要将当前的 NFA 的结束状态标记为接受状态，并且由记号名称获得它在 Symbol 类中的编号值。假设 ReturnStatement 节点为 stat，则你可以用下面的代码获得记号对应的编号值：

```
String tokenName = ((SimpleName)stat.getExpression()).getIdentifier();
int tokenId = Class.forName(
    "edu.ustc.cs.compile.lexer.Symbol").getField(tokenName).getInt(null);
```

#### 4.4.2.5 NFASimulator

为了测试你编写的 NFAGenerator 类所生成的 NFA，我们提供 NFASimulator 类，它可以用生成的 NFA 对一个输入文件进行词法分析。你可以调用 NFASimulator 的 nextToken 方法来取得每一个记号。代码片段如下：

```
NFASimulator sim = new NFASimulator(reader, nfaGen.getNFA());
Symbol s = sim.nextToken();
while (s.getType() != Symbol.EOF) {
    System.out.println(s);
    s = sim.nextToken();
}
```

### 4.4.3 课程设计任务

在熟悉了上面的内容之后，你可以着手进行本课程设计了。课程设计的主要任务是：

1. 按规定的 MLex 词法规范格式，编写 MiniJOOL 语言的词法规范文件 config/MLex/MiniJOOL.mlex；并测试你所编写的 mlex 词法规范文件能否得到正确的 AST。
2. 在 NFAGenerator 类中补充代码使其产生 NFA。在 NFAGenerator 类中已经给出了几个 visit 方法的实现，可以作为参考；你需要注意对宏、优先级等的处理。
3. 写 6 个以上的合法 MiniJOOL 词汇文件作为测试用例，说明你的测试用例的设计依

据;

4. 修改 `edu.ustc.cs.compile.lexer.Main` 类中的 `main` 方法和 `file` 成员, 并运行这个测试控制类, 测试你所做的课程设计。

你可以按与 4.3.3 节类似的方法来编译和运行本课程设计。在命令控制台下需要执行以下命令:

```
ant lab2-3
```

## 4.5 课程设计 2-4: 编写一个词法分析器的生成器

课程设计 2-3 中实现的 NFA 生成器在使用上有很大的不方便, 亦即, 对同一个词法规范, 每次都要重新从词法规范文件生成 NFA, 在运行结束之后 NFA 就随之销毁, 而没有被保存下来。

在本课程设计中, 你将实现一个真正意义的词法分析器, 它根据 `MLex` 词法规范文件生成词法分析器的 Java 源代码, 这就是与课程设计 2-3 的直接区别。尽管这个代码仍然是生成 NFA 的, 但是它不再依赖于词法规范文件。这个词法分析器的生成器就与 `lex/JFlex` 之类的工具很相似了。

生成产生 NFA 的代码比直接生成 NFA 在思维能力的要求上要更高一些, 但是一旦你理解了它的规律 (原理), 参照 `NFAGenerator`, 实现起来就很容易了。

课程设计 2-4 关联的文件有:

- **config/MJLex/MiniJOOL.mlex:** 一个 MiniJOOL 词法规范文件。
- **LexerCodeGenerator.java:** 由 AST 生成对应的词法分析器代码, 我们提供了基本的框架, 你需要补充代码完善这个类。在这个类中提供了一个 `main` 方法, 你可以编译运行这个类, 来为 `MJLex/MiniJOOL.mlex` 生成一个词法分析器类 `LexerCode`。
- **LexerCode.java.sample:** 这是 `LexerCodeGenerator.java` 需要生成的词法分析器的样例, 供你在完善 `LexerCodeGenerator.java` 时参考。
- **NFAState.java** NFA 状态类。支持 NFA 状态的管理及状态转换。
- **NFASimulator.java** 用于测试运行 NFA 的模拟器, 它以一个 NFA 和一个用于测试的单词源文件为输入。
- **Main.java** 测试总控类, 在 `main` 方法中调用 `runPart4()` 可以通过 `NFASimulator` 用生成的 `LexerCode` 对 `MiniJOOL` 源文件进行词法分析。

你可以修改类 `Main` 中成员 `file` 的值, 设置要词法分析的输入文件名。

在熟悉了上面的内容之后, 你可以着手进行本课程设计了。课程设计的主要任务是:

1. 在 `LexerCodeGenerator` 类中已经给出了几个 `visit` 方法的实现, 你应该参考它们以及我们给出的 `LexerCode` 类来补充剩余的代码。
2. 由于生成词法分析器的源代码和用生成的词法分析器分析程序需要两个步骤运行, 你应该编写一个 Ant 编译文件来辅助你编译运行。

3. 你可以采用课程设计 2-3 的测试用例来进行测试。

## 4.6 JFlex 词法规范

本节以JFlex 1.4.1 版本来说明。正如在 4.2.1.1 节中看到的，JFlex的词法规范文件由三部分组成，各部分之间通过由“%%”开始的单行来分隔，即：

```
用户代码
%%
选项和声明
%%
词法规则
```

在词法规范的各个部分中，允许出现以“/\*”开头和“\*/”结尾的多行注释，以及以“/”开头的单行注释。不过 JFlex 的注释允许嵌套，因此你在使用时应保证“/\*”和“\*/”的数目要配对。

### 4.6.1 用户代码

第一部分所包含的用户代码将被直接复制到所生成的词法分析器源文件中的开头部分。这一部分可以放 package 声明和 import 语句。在这一部分中，也可以放自己的辅助类，如记号类，但是这并不是一个好的 Java 编程风格，应该将这些辅助类单独放在自己的 .java 文件中。

### 4.6.2 选项和声明

词法规范文件的第二部分所包含的选项是用来定制你希望生成的词法分析器；而声明则是一些词法状态和宏定义的声明，它们将用于词法规范文件的第三部分，即词法规则中。

每一条 JFlex 指令必须放在一行的开始，并且由“%”字符引导，如：

```
%class SampleLexer
```

下面分节对常用的选项和声明作简要说明。此外，JFlex 还提供与 CUP 协作的选项，即 %cup、%cupsym、%cupdebug，我们将在 5.3 节介绍这些选项。

#### 4.6.2.1 常用的选项

##### 1、%class “classname”

告诉 JFlex 将生成的词法分析器类命名为“classname”，并且将生成的代码写到文件“classname.java”中。如果执行 JFlex 命令时，没有使用“-d <directory>”命令行选项，则生成的代码将被写到词法规范文件所在的目录中。如果词法规范中没有出现%class 指令，则生成的类将命名为“Yylex”，并且生成的代码将被写到文件“Yylex.java”中。在一个词法规范中只应有一条%class 指令。

##### 2、%extends “classname”

这条指令使生成的类成为类“classname”的子类。在一个词法规范中只应有一条%extends

指令。

### 3、%public

这条指令使生成的类为 `public` 类型。类似地，还有 `%final`、`%abstract` 这些指令，它们分别使生成的类为 `final` 或 `abstract` 类型。

### 4、%{...}% 类代码指令

该指令的格式是：

```
%{
    用户代码
}%
```

其中的用户代码将被直接复制到生成的类中。在这里，你可以定义自己的词法分析器类中的成员变量和函数。如果词法规范中出现有多个类代码指令，则 JFlex 依据这些指令在词法规范中出现的先后次序而将这些指令中的代码连接起来，再复制到生成的类中。

### 5、%init{...%init} 初始化代码指令

该指令的格式是：

```
%init{
    用户代码
}%init}
```

其中的用户代码将被直接复制到生成的类的构造器中。在这里，可以初始化由 `%{...}%` 指令声明的成员变量。如果词法规范中出现有多个初始化代码指令，则这些代码按它们在词法规范中出现的先后次序而被连接起来。

### 6、%buffer "size"

这条指令用于设置扫描缓冲区的初始大小，缺省为 16384。

### 7、%function "name"

这条指令用于设置词法扫描函数(或称为方法)的名称。如果在词法规范中没有出现这条指令，则扫描函数为“`yylex`”。

### 8、%integer 或 %int

这两条指令均使扫描函数声明为 `int` 类型，这样扫描函数就会返回 `int` 值当作记号。在这种设置下，文件结束的缺省值为 `YYEOF`，`YYEOF` 是生成的类中的一个 `public static final int` 型成员变量。

### 9、%intwrap

这条指令使扫描函数声明为 Java 的包装类 `Integer` 类型。这时，文件结束的缺省值为 `null`。

### 10、%type "typename"

这条指令使扫描函数声明为指定的类型。例如，在 4.2.1.1 节的例子中，扫描函数声明为 `Symbol` 类型。在这种设置下，文件结束的缺省值为 `null`。如果 `typename` 不是 `java.lang.Object` 的子类，则你应该使用 `%eofval{...%eofval}` 指令或 `<<EOF>>` 规则来规定其他的文件结束值。

### 11、%eofval{...%eofval} 文件结束值指令

该指令的格式是：

```
%eofval{  
    用户代码  
%eofval}
```

其中的用户代码将被直接复制到扫描函数中，并且将在每次遇到文件结束时执行。这个用户代码应该返回表示文件结束的值，如 4.2.1.1 节的例子中的 `return Symbol.EOF`。在词法规范中只能有一个 `%eofval{...%eofval}` 指令。

### 12、%eof{...%eof} 文件结束代码指令

该指令的格式是：

```
%eof{  
    用户代码  
%eof}
```

其中的用户代码将在遇到文件结束时只执行一次。这个代码将被放在 `void yy_do_eof()` 方法中，并且不应返回任何值；如果要返回值，则应使用 `%eofval{...%eofval}` 指令或者 `<<EOF>>` 规则。如果词法规范中出现多个文件结束代码指令，则这些代码应依照在规范中的出现次序而被连接在一起。

### 13、%eofclose

这条指令用来使 JFlex 在文件结束处关闭输入流。代码 `yyclose()` 被追加到方法 `yy_do_eof()` 中，并且在这个方法的 `throws` 子句中声明 `java.io.IOException` 异常。

### 14、%debug

在生成的类中创建一个 `main` 函数，它将从命令行中获得输入文件名，然后对这个输入文件运行词法分析器，并向 Java 控制台打印输出每个返回记号的信息，直到遇到文件结束为止。所输出的信息包括：行号(如果设置了 `%line`)、列号(如果设置了 `%column`)、匹配的文本、执行的动作(含词法规范中的行号)。

### 15、%char

这条指令用来使 JFlex 返回字符计数。`int` 型成员变量 `yycchar` 包含从输入开始到当前记号开始处的字符数(从 0 开始计数)。

### 16、%line

这条指令用来使 JFlex 返回行计数。`int` 型成员变量 `yyline` 包含从输入开始到当前记号开始处的行数(从 0 开始计数)。

### 17、%column

这条指令用来使 JFlex 返回列计数。`int` 型成员变量 `yycolumn` 包含从当前行开始到当前记号开始处的字符数(从 0 开始计数)。



### 4.6.2.2 声明

声明包括状态声明和宏定义两类。

#### 1、状态声明

状态声明有以下格式：

```
%s[tate] "状态标识符" [, "状态标识符", ...]
```

```
%x[state] "状态标识符" [, "状态标识符", ...]
```

前一种用来包含状态，后一种则用来去掉状态。在词法规范中可以包含多行状态声明，每一行都从%state 或%xstate 开始(或者是从%s 或%x 开始)。状态标识符是以字母开头的、后跟字母、数字或下划线组成的字符序列。状态标识符可以用逗号或空白分隔。

#### 2、宏定义

宏定义的格式如下：

```
宏标识符 = 正规式
```

按照这种形式定义的宏标识符稍后可以被引用。右边(RHS, right hand side)的正规式必须是合式的(well formed)，并且不能包含“^”、“/”或“\$”运算符。

### 4.6.3 词法规则

词法规则部分包含一组正规式和词法匹配后要执行的动作(Java 代码)。在词法分析器读取输入时，它会查看所有正规式，按最长匹配选择匹配的正规式，并执行该正规式对应的动作。

#### 4.6.3.1 词法规则的语法和语义

在 JFlex 用户手册的 4.3.1 节中，用 BNF 文法给出了词法规则部分的语法定义。这里列出部分重要的产生式。

```
Rule      ::= [StateList] ['^'] RegExp [LookAhead] Action
           | [StateList] '<<EOF>>' Action
           | StateGroup

StateList ::= '<' Identifier (',' Identifier)* '>'

LookAhead ::= '$' | '/' RegExp

Action    ::= '{' JavaCode '}' | '|'

RegExp   ::= RegExp '|' RegExp           // 两个正规式的或运算
           | RegExp RegExp               // 两个正规式的连接运算
           | '(' RegExp ')'               // 即匹配 RegExp
           | '!' RegExp                   // 正规式的否定运算
           | '~' RegExp                   // 匹配任何文本直到第一次匹配 RegExp
           | RegExp '{' Number [',' Number] '}' // RegExp 的重复次数
           | '[' ['^'] (Character | Character '-' Character)* ']'
           | PredefinedClass
```

```
| '{' Identifier '}'
| "'" StringCharacter+ "'"
| Character
```

其中:

- 在词法规则中,除了用正规式来描述外,还可以用一个可选的 `StateList` 来进一步细化词法规则。`StateList` 是一个词法状态列表,它就像是一个开始条件。例如,如果词法分析器处在 `STRING` 词法状态,则只有那些由开始条件`<STRING>`引导的正规式可以被匹配。词法状态 `YYINITIAL` 是预定义的,也是词法分析器启动扫描时所处的状态。如果一个正规式没有指定 `StateList`,则它将在所有词法状态上匹配。
- `Number` 是非负的十进制整数。
- `Identifier` 是一个字母`[a-zA-Z]`后跟 0 个或多个字母、数字或下划线`[a-zA-Z0-9_]`。
- 转义序列包括: 1) `\n`、`\r`、`\t`、`\f` 和 `\b`; 2) `\x` 后跟两个十六进制数字`[a-fA-F0-9]`,或者反斜杠后跟从 000 到 377 的三个八进制数字,表示标准的 ASCII 转义序列; 3) `u` 后跟四个十六进制数字`[a-fA-F0-9]`,表示 unicode 转义序列; 4) 反斜杠后跟其他任何 unicode 字符,代表这个字符。
- `Character` 是不包含下面字符之一的转义序列或任何 unicode 字符:  
| ( ) { } [ ] < > \ . \* + ? ^ \$ / "
- `StringCharacter` 是不包含下面字符之一的转义序列或任何 unicode 字符:  
\ "
- 正规式 `r` 前面加上`^`运算符,表示 `r` 只在输入的每行开始进行匹配。
- 正规式 `r` 后跟上`$`运算符,表示 `r` 只在输入的每行结尾进行匹配。
- 假设 `r1` 和 `r2` 是正规式,则 `r1/r2` 表示 `r1` 匹配的文本必须是定长的,或者 `r2` 的内容的开始不匹配 `r1` 的尾部。例如, `"abc" / "a"|"b"` 是合法的,因为`"abc"`是定长的;  
`"a"|"ab" / "x"*` 也是合法的,因为`"x"*`的前缀不匹配`"a"|"ab"`的后缀; `"x"|"xy" / "yx"`是非法的,因为`"x"|"xy"`的后缀`"y"`也是`"yx"`的前缀。
- `[StateList] <<EOF>> { 动作代码}`  
这条`<<EOF>>`规则与`%eofval`指令十分类似,其不同在于`<<EOF>>`规则前可以放可选的 `StateList`。在遇到文件结束并且词法分析器当前处于 `StateList` 中的某一词法状态时,执行动作代码。
- `PredefinedClass` 规定 JFlex 中的以下预定义的字符类:
 

<code>"."</code>	包含除 <code>\n</code> 外的所有字符
<code>"[:jletter:]"</code>	由 <code>java.lang.Character.isJavaIdentifierStart()</code> 决定的字符类
<code>"[:jletterdigit:]"</code>	由 <code>java.lang.Character.isJavaIdentifierPart()</code> 决定的字符类
<code>"[:letter:]"</code>	由 <code>java.lang.Character.isLetter()</code> 决定的字符类
<code>"[:digit:]"</code>	由 <code>java.lang.Character.isDigit()</code> 决定的字符类
<code>"[:uppercase:]"</code>	由 <code>java.lang.Character.isUpperCase()</code> 决定的字符类
<code>"[:lowercase:]"</code>	由 <code>java.lang.Character.isLowerCase()</code> 决定的字符类

### 4.6.3.2 在动作代码中可以访问的应用编程接口

生成的词法分析器类中的方法和成员变量名以“yy”为前缀，表示它们是自动生成的，避免与复制到这个类中的用户代码有名字冲突。由于用户代码也是类中的一部分，JFlex 没有像 `private` 修饰符这样的语言手段来指示哪些方法和成员是内部的，哪些属于应用编程接口 (API)。取而代之，JFlex 遵循一种命名约定：以“zz”为名字前缀的方法或成员将被认为是内部使用的，在 JFlex 的各发布版本之间不会通告对这些方法或成员的变化；生成类中不以“zz”为名字前缀的方法或成员就属于提供给用户在动作代码中使用的 API，在 JFlex 的各发布版本之间会尽可能地支持它们并保持稳定不变。

当前，API 由以下方法和成员变量组成：

- **String yytext( )**  
返回所匹配的输入文本串。
- **int yylength( )**  
返回所匹配的输入文本串的长度(不需要创建一个 String 对象)。
- **char yycharat(int pos)**  
返回位于匹配的文本中第 `pos` 个字符，这等价于 `yytext( ).charAt(pos)`，但是执行得会更快一些。`pos` 的取值范围是 0 到 `yylength( )-1`。
- **void yyclose( )**  
关闭输入流。
- **void yyreset(java.io.Reader reader)**  
关闭当前的输入流，并复位词法分析器，使之读取一个新的输入流。所有的内部变量将被复位，原先的输入流不能被重用(内部缓冲区的内容被丢弃)。词法状态被设置为 `YY_INITIAL`。
- **void yypushStream(java.io.Reader reader)**  
将当前的输入流保存到一个栈中，并从一个新的输入流中读取。词法状态以及行、字符和列的计数信息保持不变。可以用 `yypopstream`(通常放在 `<<EOF>>` 动作中)恢复当前的输入流。
- **void yypopStream( )**  
关闭当前的输入流，从输入流栈出栈，并从弹出的输入流中继续读取。
- **boolean yymoreStream( )**  
如果输入流栈中还有输入流，则返回 `true`。
- **int yystate( )**  
返回当前的词法状态。
- **void yybegin( int lexicalState )**  
进入词法状态 `lexicalState`。
- **void yypushback( int number )**  
将所匹配的文本中 **number** 个字符退回到输入流中。这些被退回的字符将在下次调用扫描方法时被再次读入。在调用 `yypushback` 后，被退回的字符将不会包含在

yylength 和 yytext( )中。

- **int yyline**

包含输入文件的当前行数(从 0 开始, 只有在设置了%line 指令时才被激活)。

- **int yychar**

包含输入文件的当前字符数(从 0 开始, 只有在设置了%char 指令时才被激活)。

- **int yycolumn**

包含输入文件的当前列数(从 0 开始, 只有在设置了%column 指令时才被激活)。

## 第5章 语法分析

在本章中，你将学习为 MiniJOOL 语言的一个非面向对象子集 SkipOOMiniJOOL 语言实现一个语法分析器。语法分析器按 SkipOOMiniJOOL 语言的语法规则检查词法分析输出的记号流是否符合这些规则，并依据这些规则所体现出的语言中各种语法结构的层次性，以 Eclipse JDT 中的 AST 为基础构建描述记号流的抽象语法树 AST。

为了便于你掌握语法分析的各方面知识，我们在本章安排了三个构造语法分析器的课程设计以及一个构造语法分析器的生成器的课程设计，你可以根据实际情况，选做其中的部分或全部。

### 5.1 本章课程设计概述

在本章中，我们安排如下四个课程设计：

#### 课程设计 3-1 手工编写一个语法分析器。

在这个课程设计中，你可以使用有回退的递归下降方法来构造一个语法分析器，这个分析器能分析赋值语句序列并生成对应的 AST。你需要知道递归下降方法所适用的文法范围，以及如何将一个不适用的文法改写为一个适用的文法。

在现代编译技术中，同样也有专门的语法分析器的生成工具，如以 LALR 文法为基础的 YACC (Yet Another Compiler Compiler)及其各种变形，如 CUP (Constructor of Useful Parsers)，以 LL(k)文法为基础的 JavaCC (Java Compiler Compiler)等。这些工具可以从输入的高级文法规范文件生成语法分析器的源程序代码。下面两个课程设计就是针对这两类生成器进行设计的。

#### 课程设计 3-2 用 CUP 生成一个语法分析器。

在这个课程设计中，你将熟悉 CUP 的使用以及 cup 文法规范文件的格式，体会 LALR 文法的编写特点。你需要为 SkipOOMiniJOOL 语言以至 MiniJOOL 语言编写 cup 文法规范文件，在这个文件中，你需要考虑如何添加构造 AST 的代码，如何进行错误恢复。

#### 课程设计 3-3 用 JavaCC 生成一个语法分析器。

在 JavaCC 的文法规范文件中，不仅可以描述语言的语法规则，而且可以描述词法规则。你将在这个课程设计中学习用 JavaCC 为 SkipOOMiniJOOL 语言以至 MiniJOOL 语言构造一个不含语义分析的编译器前端，包括词法分析、语法分析和 AST 的生成，你需要考虑语法分析中的错误恢复问题。通过这个课程设计，你可以熟悉 JavaCC 的使用以及它的文法规范文件的格式，你还可以体会 LL(k)文法的编写特点，掌握编写 JavaCC 文法规范文件的方法。

#### 课程设计 3-4 编写一个语法分析器的生成器。

在这个课程设计里，你将编写一个能产生递归下降有回退的语法分析器的生成器。在完成这样的工具后，可以使你对递归下降有回退的语法分析器原理的理解达到非常深的程度。

通过这一章的课程设计练习，你可以对递归下降的语法分析器有深入的理解，这种语法分析器比较容易理解，并且容易实现，不过在为它写文法的时候，要求比较严格。语法分析器的生成器又涉及到了从高级的文法规范生成代码的工作，这也会为最后的代码生成步骤打下基础。另外，通过使用 CUP 和 JavaCC 来为语言构造语法分析器，可以让你进一步理解 LALR 文法和 LL 文法的特点。

在开始做本章的课程设计前，你需要将我们提供的ROOT\_DIR/lab/lab3 导入到Eclipse中，建立Lab3 项目。在课程设计 3-1 中使用了课程设计 2-2 的词法分析器，在 5.2 节将介绍如何在Eclipse中引用Lab2 的结果。与本章课程设计相关的文件主要集中在本书提供的软件包的目录ROOT\_DIR/lab/lab3 中，该目录包括如下内容：

- **README:** 本章课程设计的说明文件。
- **.project 文件和.classpath 文件:** 本章课程设计的 Eclipse 工程文件。
- **bin 目录**
  - **run.bat 和 run.sh:** 运行实验平台的批处理文件。
  - **lab3-1.xml、lab3-2-expr.xml 和 lab3-2-expr-err.xml、lab3-3-expr.xml 和 lab3-3-expr-err.xml:** 一组 ant 编译文件，分别用于课程设计 3-1、课程设计 3-2 中的示例、课程设计 3-3 中的示例。
- **config 目录**

存放本章课程设计所需的词法、语法规规范文件以及实验平台配置文件。

  - **lab3-1.xml、lab3-2-expr.xml 和 lab3-2-expr-err.xml、lab3-3-expr.xml 和 lab3-3-expr-err.xml:** 一组实验平台配置文件，依次用于课程设计 3-1、课程设计 3-2 中的示例、课程设计 3-3 中的示例。
  - **CUP 目录**
    - ◆ **expr.cup:** 课程设计 3-2 中的示例 1（不带错误恢复的分析器）所需的 CUP 文法规范文件。
    - ◆ **expr\_err.cup:** 课程设计 3-2 中的示例 2（带错误恢复的分析器）所需的 CUP 文法规范文件。
  - **JFlex 目录**
    - ◆ **expr.flex:** 课程设计 3-2 中的示例所需的 JFlex 词法规范文件。
  - **JJ 目录**
    - ◆ **expr.jj:** 课程设计 3-3 中的示例 1（不带错误恢复的分析器）所需的 JavaCC 文法规范文件。
    - ◆ **expr\_err.jj:** 课程设计 3-3 中的示例 2（带错误恢复的分析器）所需的 JavaCC 文法规范文件。
- **src 目录**

存放你实验用到的一些 Java 源文件，你需要对其中的部分源文件补充代码。这些 Java 源文件的包路径前缀均为 edu.ustc.cs.compile.parser，具体包括：

  - **edu/ustc/cs/compile/parser 目录:**

- ◆ **RDParser.java**: 用于课程设计 3-1, 使用的语法分析器的抽象基类;
- ◆ **ExpressionParser.java**: 用于课程设计 3-1, 分析赋值语句序列的语法分析器, 从 RDParser 继承, 并实现实验平台的 ParserInterface。
- ◆ **Main.java**: 测试总控类, 在 main 方法中调用 runPart1( )、runPart2( )、runPart3( )可以分别运行课程设计 3-1、课程设计 3-2、课程设计 3-3。
- **test 目录**  
存放你实验用到的一些测试程序, 你可以在其中补充更多的测试程序。

## 5.2 课程设计 3-1: 手工编写一个语法分析器

4.4 节要求你手工编写一个能识别表达式的词法分析器 ExpressionLexer。在这个课程设计中, 要求你手工编写一个能分析赋值语句序列并构造相应的 AST 的语法分析器。你需要使用在上一章中实现的 ExpressionLexer 来识别输入文件中的记号, 然后用所编写的语法分析器检查这些记号流是否符合赋值语句序列的语法。

本课程设计旨在让你通过实现一个简单的语法分析器, 来了解如何将一个语法描述编写成一个语法分析程序, 你将使用容易理解和实现的递归下降法来编写语法分析程序。

课程设计 3-1 关联的 Java 源文件 (相对于路径 src/edu/ustc/cs/compile/parser.) 有:

- **Main.java**: 测试总控类, 在 main 方法中调用 runPart1( )可以运行这个实验;
- **expr/RDParser.java**: 语法分析器的抽象基类, 其中的抽象方法 parse 是用于语法分析的, 每个由 RDParser 派生的类需要实现这个抽象方法;
- **expr/ExpressionParser.java**: 实现对赋值语句序列分析的语法分析器, 这个类从 RDParser 继承, 并实现实验平台的 ParserInterface, 你需要在里面补充代码, 实现 parse 方法, 使之能分析输入的程序并产生对应的 AST。

关联的其他文件有:

- **bin 目录**
  - **lab3-1.xml**: 课程设计 3-1 的 ant 编译文件。
- **config 目录**
  - **lab3-1.xml**: 用实验平台运行课程设计 3-1 的配置文件。

这个课程设计中引用了 lab2-2 课程设计中的如下类 (包名为 edu.ustc.cs.compile.lexer):

- **Lexer**: ExpressionParser 分析器需要用到的抽象词法分析器类;
- **ExpressionLexer**: ExpressionParser 分析器需要用到的具体词法分析器类;
- **Symbol**: 分析器要用到这个词法记号类文件。

### 5.2.1 如何引用 Lab2 项目中的类

这里将分别介绍在 Eclipse 中以及在命令行上的处理方法。

#### 1、用我们提供的 lab3 直接导入到 Eclipse 建立 lab3 工程

在我们提供的 lab3 目录下的工程文件.project 和.classpath 中, 已经反映对 lab2 的代码的

引用。你可以按 1.4.3.3 节中第 5 点的方法，利用它们来导入建立 lab3 工程，这时所建立的工程中已经包含 lab2 中的代码，即 lab2-src。（此方法仍需要改进，待修订）

下面说明为引用 lab2 中的代码需要对 .project 和 .classpath 做哪些修改。首先，在 .project 文件中的 linkedResources 元素里增加如下两个名为 lab2-classes 和 lab2-src 的链接资源，以分别引用 lab2 中的 classes 和 src 位置：

```
<link>
  <name>lab2-classes</name>
  <type>2</type>
  <location>E:/CompilerProj/student/lab/lab2/classes</location>
</link>
<link>
  <name>lab2-src</name>
  <type>2</type>
  <location>E:/CompilerProj/student/lab/lab2/src</location>
</link>
```

然后在 .classpath 中的 classpath 元素里增加如下一条子元素

```
<classpathentry output="lab2-classes" kind="src" path="lab2-src"/>
```

该子元素表示，源文件位于由名为 lab2-src 指定的链接位置，而生成的 class 文件则输出到由名为 lab2-classes 指定的链接位置。

## 2、在 Eclipse 中将 lab2 工程加入到 lab3 工程中

如果你建立的 lab3 工程中尚没有引用 lab2 工程，且 lab2 工程也已经在当前的 Eclipse 工作区中，这时你需要修改 lab3 工程的“Properties”设置中的“Java Build Path”，在其中的“Projects”页面，点击“Add”命令，在弹出的窗口勾选“Lab2-Lexer”，表示 lab3 工程将引用 lab2 工程。

## 3、在命令行上的处理方法

在命令行上处理的方法比较简单，只需要在用 javac 编译或用 java 运行课程设计 3-1 时，保证 lab2 的 classes 路径被加在命令所需的 classpath 中。

### 5.2.2 要分析的语法结构

赋值语句序列由一系列的赋值语句组成，每一个赋值语句由赋值表达式后跟分号组成。整个赋值语句序列的文法如下：

```
sequence      =  assignment sequence | ε
assignment    =  IDENTIFIER EQ expression SEMICOLON
expression    =  expression PLUS term
               | expression MINUS term
               | term
```



```

term      = term MULT factor | term DIV factor | factor
factor    = LPAREN expression RPAREN
           | IDENTIFIER
           | INTEGER_LITERAL

```

其中，由大写字母组成的符号代表终结符(记号)，上面出现的各个终结符的含义可以在附录 1 中查到；由若干小写字母组成的符号代表非终结符，而非终结符 `sequence` 是这个语言的开始符。

上面的文法不适用于递归下降的分析方法，因为这个文法是左递归的。为了使用递归下降方法分析，需要将文法改写如下：

```

sequence  = assignment sequence | ε
assignment = IDENTIFIER EQ expression SEMICOLON
expression = term expression_1
expression_1 = PLUS term expression_1
              | MINUS term expression_1
              | ε
term        = factor term_1
term_1      = MULT factor term_1 | DIV factor term_1 | ε
factor      = LPAREN expression RPAREN
              | IDENTIFIER
              | INTEGER_LITERAL

```

### 5.2.3 课程设计指导

在进行语法分析时，需要维护一个栈，这个栈已经在我们提供的语法分析器抽象基类 `RDParse` 中实现，`RDParse` 类中的 `peek`、`push`、`pop` 等方法可以用于完成取栈中的元素、入栈和出栈等操作。你只需要将自己的语法分析器类从类 `RDParse` 继承即可使用，如本课程设计要实现的 `ExpressionParser` 类。`RDParse` 中有一个抽象的 `parse` 方法，任何从 `RDParse` 派生的类需要实现这个方法来完成语法分析并返回输入程序所对应的 AST 的根节点。

在 `ExpressionParser` 类中，将采用递归下降的分析方法实现语法分析，因此针对赋值语句序列文法中的每一个非终结符，在类 `ExpressionParser` 中有一个对应的、以该非终结符命名的方法。这样，在 `parse` 方法中，就只要调用开始符号对应的方法并返回其结果（表示分析成功与否）。

在文法中，一个产生式的右部(RHS, right-hand-side)可能对应多个分支，每一分支定义了产生式左部(LHS, left-hand-side)非终结符所代表的串的一种可能。在类 `ExpressionParser` 中，可以为每一分支定义一个方法，该方法用 LHS 符号名加序号来命名。例如，对产生式

```
sequence = assignment sequence | ε
```

LHS 符号为 `sequence`，它有 `assignment sequence` 和 `ε` 两个分支，则在 `ExpressionParser` 类中定义三个方法，即 `sequence`、`sequence1` 和 `sequence2`，分别用来分析整个 RHS、和 RHS 的

两个分支。

在开始分析 RHS 前，首先应记住当前的状态，包括下一记号的位置、栈顶的位置等等；如果 RHS 有多个分支，则按分支出现的先后次序来依次调用分支对应的方法，如果某一分支分析成功，则整个 RHS 分析成功并返回；否则把状态设置为上一分支分析前所记录的状态，再继续调用下一分支对应的方法。

在每一个 RHS 分支对应的方法中，将顺次分析该分支中的每个终结符和非终结符。如果是终结符，则调用 RDParse 类的 token 方法判断下一个记号是否与终结符相匹配（一个例外是对于终结符  $\epsilon$ ，遇见它的时候，调用 tokenEpsilon 方法）。如果是非终结符，则调用它对应的方法。仅当每个调用都返回 true 时，该 RHS 分支对应的方法才返回 true。

需要注意的是，针对正确的分析，需要构造各文法符号对应的AST节点，并根据RHS的构成将这些AST节点连接起来形成一棵AST。语法分析器类中的栈在这里就可以用来缓存当前分析出的文法符号对应的AST节点。你可以参考 5.3.1 节中的示例来了解每一种语法结构所对应的AST节点类型。

#### 5.2.4 课程设计要求

我们已经在 ExpressionParser 类中为所有的非终结符添加了对应的方法，并且给出了这些方法的分析框架，你需要在这些方法中补充构造 AST 的代码，为了做到这一点，你需要利用栈来传递符号对应的 AST 节点。

你应该至少写 6 个稍复杂的测试用例来测试自己的分析器。在 Main 类中，我们已经编写了测试这个赋值语句序列语法分析器所需要的代码（包括对词法分析器的调用），即方法 runPart1。你可以将 runPart1 中的“test/exp\_list.txt”替换成你自己的测试用例文件名来进行测试。

在编译、运行本课程设计前，你需要确认你已经正确地完成 lab2-2 中的词法分析器 ExpressionLexer，使之能识别本课程设计中的所有词法记号。

在 Eclipse 环境下，以 lab3 工程的 src/edu.ustc.cs.compile.parser.Main 为主类进行调试和运行；或者以 lab3 工程 bin/lab3-1.xml 为 ant 编译文件来用 ant 调试和运行。

在命令控制台下，进入 lab3/bin 目录下，执行 ant 编译 lab3-1：

```
ant -f lab3-1.xml
```

再通过实验平台运行语法分析器 ExpressionParser

```
./run.sh -cf ../config/lab3-1.xml
```

或 run -cf ../config/lab3-1.xml

你可以修改 config/lab3-1.xml 设置运行实验平台的命令行选项。

### 5.3 课程设计 3-2: 用 CUP 生成一个语法分析器

CUP 是一个类似 YACC 的工具，它能从 cup 文法规范文件生成对应的 LALR 语法分析器。与 YACC 生成 C 语言代码不同，它生成的是 Java 语言代码。在这个课程设计中，要求

你用 CUP 为 MiniJOOL 语言或其子集生成一个语法分析器,词法分析器则用 JFlex 来生成(即课程设计 2-1)。

### 5.3.1 示例 1: 不带错误恢复的语法分析器

我们先介绍如何用 CUP 生成一个简单的不带错误恢复的语法分析器的例子,这个语法分析器可以分析上一节所说的赋值语句序列。这个示例是帮助你熟悉 CUP,了解如何让 JFlex 和 CUP 协作,为开展自己的课程设计做准备。

先看一下这个示例关联的文件:

- **config/JFlex/expr.flex:** JFlex 词法规范文件,描述赋值语句块的词法构成。
- **config/CUP/expr.cup:** CUP 语法规则文件,描述赋值语句块的语法。
- **bin/lab3-2-expr.xml:** 一个 ant 编译文件,用来控制由 expr.flex 生成词法分析器类 ExprLexer.java,由 expr.cup 生成词法记号类 ExprSymbol.java 和语法分析器类 ExprParser.java,这些文件均被输出到 src/edu/ustc/cs/compile/parser/expr 目录下;接下来,ant 会继续编译、运行所生成的分析器。
- **config/lab3-2-expr.xml:** 用实验平台运行本示例所需的配置文件。

在这个示例中用到 JFlex 和 CUP 工具,你需要确认它们是否已经安装和配置好(参见 1.5.1 节)。

接下来的工作分三步走:一是编写词法规范文件 expr.flex;二是编写语法规则文件 expr.cup;三是对 cup.xml 进行 ant build,产生分析器源代码,接着编译分析器,再运行这个分析器进行测试。我们将在下面三小节中分别叙述。

#### 5.3.1.1 编写 jflex 词法规范文件

在上一章中,我们已经熟悉如何使用 JFlex 以及如何编写 jflex 词法规范文件。本节不再赘述,而是侧重说明 JFlex 与 CUP 协同工作时需要注意的地方。

在我们提供的 expr.flex 中,使用了如下三个与 CUP 有关的选项,即:

```
%cupsym ExprSymbol
%cup
%cupdebug
```

下面依次解释这三个选项。

##### 1、%cup

使用这条指令表示将与 CUP 协同工作。它等价于如下的一组指令:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
    return new java_cup.runtime.Symbol(<CUPSYM>.EOF);
%eofval}
```

%eofclose

<CUPSYM>的缺省值为 sym，你可以通过%cupsym 指令来修改。

## 2、%cupsym “classname”

用于设置 CUP 生成的包含终结符号(词法记号)名的类或接口。缺省为 sym。这条指令应该放在%cup 之前。

## 3、%cupdebug

在生成的类中创建一个 main 函数，它期望从命令行上获得输入文件的名字，然后对这个文件运行词法分析器(扫描器)，它将打印每一个由词法分析器返回的记号的行号、列号、匹配的文本串、CUP 符号名。

在 expr.flex 中定义了以下几个宏分别表示可打印的字符、空白符、注释、标识符、整数：

```
Printable = [ ~\ ]
WhiteSpace = [ \t\f\n\r ] \r\n
Comment = "/" ({Printable} | [\t])*
Identifier = [:letter:] ([:letter:] | [:digit:])*
IntegerLiteral = [:digit:]+
```

其中，这里的 IntegerLiteral 定义比较简单，你可以进一步扩展来支持更多的整数文本。

另外，定义了以下以 YYINITIAL 词法状态为开始条件的词法规则：

```
<YYINITIAL> {
    "("                { return symbol(ExprSymbol.LPAREN); }
    ")"                { return symbol(ExprSymbol.RPAREN); }
    ";"                { return symbol(ExprSymbol.SEMICOLON); }
    "+"                { return symbol(ExprSymbol.PLUS); }
    "-"                { return symbol(ExprSymbol.MINUS); }
    "*"                { return symbol(ExprSymbol.MULT); }
    "/"                { return symbol(ExprSymbol.DIV); }
    "="                { return symbol(ExprSymbol.EQ); }
    {IntegerLiteral}   { return symbol(
        ExprSymbol.INTEGER_LITERAL, new Integer(yytext())); }
    {Comment}          { /* ignore */ }
    {WhiteSpace}        { /* ignore */ }
    {Identifier}        { return symbol(
        ExprSymbol.IDENTIFIER, yytext()); }
}
```

下面两条规则分别表示遇到其它字符的错误处理，以及遇到文件结束符的处理。

```
.\n                { throw new RuntimeException(
    "Illegal character \"\""+yytext()+"\" at line "
    +yyline+", column "+yycolumn); }
<<EOF>>           { return symbol(ExprSymbol.EOF); }
```

上面的许多词法规则的动作中，都包含有对 symbol 方法的调用，该方法在 expr.flex 中的用户代码中定义，包含如下两种形式：

```
private Symbol symbol(int type) {
    return new Symbol(type, yyline+1, yycolumn+1);
}

private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline+1, yycolumn+1, value);
}
```

需要注意的是，这里的 Symbol 类是 CUP 运行时库中的 `java_cup.runtime.Symbol`。

### 5.3.1.2 编写 cup 语法规范文件

`expr.cup` 是描述赋值语句序列的 CUP 语法规范文件，将它输入到 CUP 可以生成能分析赋值语句序列的语法分析器，这个分析器将在分析的过程中构造相应的 AST。

从 `expr.cup` 中可以看到，一个 CUP 语法规范文件分用户代码、终结符和非终结符的声明、算符优先级声明和规则几部分。文法由一组语法规则组成，在规则中可以嵌入用户希望执行的动作代码。

#### 1、用户代码

这一部分是一些 Java 代码，通常有 `package` 声明、`import` 声明，还有其他的一些代码。在 `expr.cup` 中，使用了 `action code { :用户代码: }` 和 `parser code { :用户代码: }` 两类代码块。

`action code { :用户代码: }` 用来封装动作代码中用到的数据成员和方法，CUP 将把这些代码复制到生成的语法分析器类中的内部类中，你可以通过语法分析器类中的 `action_obj` 对象来访问这些数据成员和方法。在 `expr.cup` 的 `action code` 块中包含如下的代码：

```
action code {
    private ASTNode root = null;
    private AST ast = AST.newAST(AST.JLS3);
    public ASTNode getAST() {
        return root;
    }
};
```

其中，`ast` 和 `root` 为动作代码中可以访问的 AST 对象和需要构造的 AST 根节点(这里为 `Block` 类型)，而方法 `getAST` 则用来供外部访问语法分析所得到的 AST 根节点。

`parser code { :用户代码: }` 中的用户代码将被 CUP 复制到生成的语法分析器类中，你可以为生成的分析器类添加一些成员，如 `main` 方法等。`expr.cup` 中的 `parser code` 块定义有 `doParse` 和 `main` 两个 `static` 方法。`doParse` 方法是为了实现实验平台分析器接口 `ParserInterface` 中的方法，这样这个分析器就可以作为分析器组件在实验平台上使用了。下面是 `expr.cup` 中的 `parser code` 块的内容：

```
parser code {
    private boolean success = false;
    private boolean debug = false;
```

```
public InterRepresent doParse(File src) throws ParseException {
    ExprLexer lexer = null;
    ExprParser parser = null;
    try {
        lexer = new ExprLexer( new FileReader(src) );
    } catch (FileNotFoundException e) {
        throw new ParseException();
    }

    parser = new ExprParser(lexer);
    try {
        parser.parse();
    } catch (Exception e) {
        System.err.println("Parser Exception.");
        if (debug) {
            e.printStackTrace();
        }
        throw new ParseException();
    }

    HIR ir = new HIR();
    ir.setIR(parser.action_obj.getAST());

    success = true;
    return ir;
}

public static void main(String[] argv) {
    String srcFileName = null; // source file name
    File srcFile = new File(srcFileName);

    ExprParser parser = new ExprParser();
    HIR ir = new HIR();
    try {
        ir = (HIR)parser.doParse(srcFile);
    } catch (ParseException e) {
        System.err.println("ParseException");
        e.printStackTrace();
        System.exit(-1);
    }

    ASTViewer astviewer = new ASTViewer((ASTNode)ir.getIR(),
        new GenericPropertyDump());
    astviewer.show();
}
```

```

    }
  :}

```

其中, ExprParser 是所生成的语法分析器类, 这个名字是在 CUP 命令行参数中设置的; 语法分析所构造的 AST 可以通过 parser.action\_obj.getAST() 获得并返回。

## 2、终结符和非终结符声明

终结符由 terminal 声明, 非终结符由 non terminal 声明。你可以指定这些符号所具有值的类型; 如果未指定, 则说明这些符号没有值。

在 expr.cup 中定义了如下的终结符, 其中只有 INTEGER\_LITERAL 和 IDENTIFIER 有值, 类型分别是 Integer 和 String。

```

terminal          SEMICOLON, PLUS, MINUS, MULT, DIV, EQ;
terminal          LPAREN, RPAREN;
terminal Integer  INTEGER_LITERAL;
terminal String   IDENTIFIER;

```

下面是 expr.cup 中定义的非终结符, 除了 goal, 其余符号都有值, 值的类型为 List 或者是某类 AST 节点类型。

```

non terminal goal;
non terminal List sequence, statements;
non terminal ExpressionStatement statement;
non terminal Assignment assignment;
non terminal Expression expression, term, factor;

```

## 3、算符优先级声明

算符优先级声明由 precedence 来指明, 出现在同一 precedence 语句中的终结符具有相同的优先级。在一个 cup 文件中, 可以有多条 precedence 语句, 出现在前面的 precedence 语句中的终结符比出现在后面的优先级低。如在下面的优先级声明中, LPAREN(左括号)的优先级最高, PLUS(加号)和 MINUS(减号)的优先级最低。另外, 同一优先级的符号缺省采用自左向右的结合。

```

precedence left PLUS, MINUS;
precedence left MULT, DIV;
precedence left LPAREN;

```

**注意:** 上面三条优先级声明在 expr.cup 中并不一定需要, 因为语法规则的定义已经蕴涵了这些符号的优先关系。

## 4、规则

规则由一组包含有语义动作的产生式以及归约后要执行的语义动作代码组成。你可以用 start with 开始符; 来指定开始符。如果没有指定开始符, 则默认的开始符是第一个规则的 LHS。语义动作代码包含在 { : 和 ; } 之间。下面是 expr.cup 中的部分文法定义:

```

goal          ::=
sequence:seq
{

```

```

        root = ast.newBlock();
        ((Block)root).statements().addAll(seq);
    :}
    ;
sequence ::=
    {: RESULT = new LinkedList ();
    :}
    | statements:bs
    {: RESULT = bs;
    :}
    ;

```

其中,产生式的 LHS 由 RESULT 引用,用户可以为 RHS 中的符号引入标号,如 sequence:seq 表示为符号 sequence 引入 seq 标号,这样就可以通过标号来引用相应的 RHS 符号的值了。需要指出的是,如果希望 LHS 有值,必须在语义动作中显式地将 RHS 符号的值按所需的要求计算并赋值给 RESULT。

你可以参考 CUP 用户手册<sup>[2]</sup>了解更多的 CUP 用法。

### 5.3.1.3 分析器的生成、编译和运行

在编写完 expr.flex 和 expr.cup 后,就可以运行 JFlex 和 CUP 生成分析器的源代码了。为了便于分析器的生成、编译和运行,我们提供了一个 ant 编译文件 lab3-2-expr.xml。下面先说明这个文件,再说明如何开发和测试。

#### 1、lab3-2-expr.xml

在 lab3-2-expr.xml 文件中定义了 jflex、java-cup、config、build 四个任务组,缺省的任务组为 build。

jflex 任务组利用 jflex 为 config/JFlex/expr.flex 生成词法分析器 ExprLexer.java,并将生成的文件置于 src/edu/ustc/cs/compile/parser/expr 下。

```

<target name="jflex" description="生成词法分析器">
    <mkdir dir="${SRC_DIR}/${PARSER_DIR}"/>
    <jjava fork="yes" jar="${JFLEX}">
        <arg value="-v"/>
        <arg value="-d"/>
        <arg value="${SRC_DIR}/${PARSER_DIR}"/>
        <arg value="${CFG_DIR}/JFlex/expr.flex"/>
    </jjava>
</target>

```

java-cup 任务组依赖于 jflex 任务组,它利用 CUP 根据 config/CUP/expr.cup 生成语法分析器 ExprParser.java 和词法记号类 ExprSymbol.java,并置于 src/edu/ustc/cs/compile/parser/expr 目录下。



```

<target name="java-cup" depends="jflex" description="生成语法分析器">
  <java fork="yes" jar="${JAVACUP}">
    <arg value="-package"/>
    <arg value="${PARSER_PACKAGE}"/>
    <arg value="-parser"/>
    <arg value="${PARSER_CLASS}"/>
    <arg value="-symbols"/>
    <arg value="${SYM_CLASS}"/>
    <arg value="-progress"/>
    <arg value="${CFG_DIR}/CUP/expr.cup"/>
  </java>
  <move file="${PARSER_CLASS}.java"
    todir="${SRC_DIR}/${PARSER_DIR}"/>
  <move file="${SYM_CLASS}.java" todir="${SRC_DIR}/${PARSER_DIR}"/>
</target>

```

config 任务组依赖于 java-cup 任务组，它主要是在生成分析器后，提示用户修改所生成的 ExprParser.java，使得这个类实现 ParserInterface 接口。即将

```
public class ExprParser extends java_cup.runtime.lr_parser
```

换成

```
public class ExprParser extends java_cup.runtime.lr_parser implements ParserInterface
```

build 任务组对 src/edu/ustc/cs/compile/parser/expr 下的三个文件进行编译。

```

<target name="build" description="编译语法分析器">
  <mkdir dir="${DEST_DIR}"/>
  <javac debug="on" classpath="${CLASSPATH}"
    srcdir="${SRC_DIR}" destdir="${DEST_DIR}"
    includes="${PARSER_DIR}/ExprLexer.java,
      ${PARSER_DIR}/${PARSER_CLASS}.java,
      ${PARSER_DIR}/${SYM_CLASS}.java"/>
</target>

```

## 2、开发和测试

在命令控制台下，进入 ROOT\_DIR/lab/lab3/bin 下，执行

```
ant -f lab3-2-expr.xml config
```

将由词法和语法规规范生成分析器，然后修改 ExprParser.java 使之实现 ParserInterface。然后执行

```
ant -f lab3-2-expr.xml build
```

对生成的分析器进行编译。

为运行所生成的分析器，你可以用在 `bin` 下提供的 `run.bat` 或 `run.sh` 和配置文件 `config/lab3-2-expr.xml` 来启动实验平台进行运行，即

```
run -cf ..\config\lab3-2-expr.xml
```

```
或 ./run.sh -cf ../config/lab3-2-expr.xml
```

你也可以直接执行类 `ExprParser` 来对输入的文件进行语法分析并查看生成的 AST 示意图。你还可以修改类 `Main` 中的 `main` 方法使之调用 `runPart2` 方法，再执行这个 `Main` 类来对指定的文件进行语法分析并查看生成的 AST 示意图。

### 5.3.2 示例 2：带错误恢复的语法分析器

当输入给语法分析器的源程序文件中存在多个语法错误时，5.3.1 节中的不带错误恢复的语法分析器往往会在遇到第一个语法错误时就停止。然而，在实际使用中，我们总是希望语法分析器能够尽可能多地发现输入的源程序中存在的语法错误。这时，就要求语法分析器具备错误恢复的能力。在这一节中，你需要在示例 1 的基础上，为语法分析器添加错误恢复能力。

先看一下这个示例关联的文件：

- **config/JFlex/expr.flex**: JFlex 词法规范文件，描述赋值语句块的词法构成。
- **config/CUP/expr\_err.cup**: CUP 语法规则文件，描述赋值语句块的语法。
- **bin/lab3-2-expr-err.xml**: 一个 ant 编译文件，用来控制由 `expr.flex` 生成词法分析器类 `ExprLexer.java`，由 `expr_err.cup` 生成词法记号类 `ExprSymbol.java` 和语法分析器类 `ExprParser.java`，这些文件均被输出到 `src/edu/ustc/cs/compile/parser/expr` 目录下；接下来，ant 会继续编译、运行所生成的分析器。
- **config/lab3-2-expr-err.xml**: 用实验平台运行本示例所需的配置文件。

#### 5.3.2.1 CUP 的错误恢复机制

CUP 提供一种和 YACC 类似的错误恢复机制。它提供了一个特殊的非终结符 `error` 用于匹配错误的语法结构。非终结符 `error` 无需在 CUP 文法规范文件中声明即可使用。

为了利用 CUP 的错误恢复机制，你需要在 `cup` 文件中添加一些包含非终结符 `error` 的规则，用于匹配可能出现的错误的语法结构。例如，我们以示例 1 中的 `expr.cup` 文件为基础，改进得到一个能支持错误恢复的 `cup` 文件 `expr_err.cup`，其中 `assignment` 的产生式规则定义比 `expr.cup` 中的多增加了一条 RHS 分支，即：

```
assignment ::=
    IDENTIFIER EQ expression
    | error EQ expression
;
```

这样当输入流中的赋值表达式的左值不是标识符时，CUP 的错误恢复机制会跳过若干输入流中的记号（用 `error` 表示它们），直到遇到一个等号和表达式为止，然后根据这条规则得到一个 `assignment`。你可以完善这条规则的语义动作：输出更加有用的提示信息，比如告知用

户赋值表达式的左值只能为标识符；令 RESULT 为一个 AST 节点，比如表示 “error=error” 的 AST 节点。

需要注意的是，一个错误被恢复，当且仅当错误的记号后有足够多的能够被正确解析的记号。CUP 缺省指定在错误记号之后需要有至少 3 个记号才能正确解析。你可以通过在 parser code 块中定义变量 `int _error_sync_size` 改变这个缺省值。

### 5.3.2.2 分析器的生成、编译和运行

带错误恢复的语法分析器的生成、编译和运行与示例 1 基本相同。只要将 `ant` 配置文件改为 `bin/lab3-2-exp-err.xml`，实验平台配置文件改为 `config/lab3-2-exp-err.xml`。

### 5.3.3 课程设计任务

参考 5.3.1 节所讲的例子和 5.6 节的 CUP 语法规则或者 CUP 的用户手册，你就可以着手为 MiniJOOL 语言或其子集生成一个语法分析器。这个语法分析器不对输入的程序进行语义分析，并且在分析过程中要构造相应的 AST。你需要理解 Eclipse JDT 中的 AST 节点种类及结构，为 MiniJOOL 语言或其子集的语法结构选择合适的 ASTNode 类型或 List 类型。

在本课程设计中，你必须能为 SkipOO MiniJOOL 语言建立正确的能生成 AST 的语法分析器，当然你可以进一步扩展得到 MiniJOOL 语言的语法分析器。针对一种语言，你需要依次完成以下任务：

- 1) 编写描述语言语法的不含任何语义动作代码的 `cup` 文法，并将它保存在 `config/CUP/MiniJOOL.cup`；把在上章课程设计完成的 MiniJOOL 词法规范文件 `MiniJOOL.flex` 置于 `config/JFlex` 下。

你不必从零开始写这个 `cup` 文件。由于 MiniJOOL 是 Java 语言的一个子集，所以你可以下载 Java 的 `cup` 文法（<http://www2.cs.tum.edu/projects/cup/javagrm.tar.gz>），对它进行删减，从而改变成 MiniJOOL 语言或其子集的 `cup` 文法规范文件。

- 2) 参照 `bin/lab3-2-exp.xml`，写一个 `ant` 编译文件，假设为 `bin/lab3-2-mj.xml`，使之能根据 `MiniJOOL.flex`、`MiniJOOL.cup` 生成词法分析器类 `MJLexer`、词法记号类 `MJSymbol` 和语法分析器类 `MJParser` 的源代码，并将这三个文件存放在目录 `src/edu/ustc/cs/compile/parser/minijool` 中。
- 3) 按 5.3.1 节介绍的方法用 `ant` 运行 `lab3-2-exp.xml`，生成上述三个类文件，修改其中的分析器类使之实现 `ParserInterface` 接口。再用 `ant` 对修改后的分析器类进行编译。
- 4) 如果编译有错，则修改你所编写的 `MiniJOOL.flex` 和 `MiniJOOL.cup`，直至能得到编译正确的分析器。
- 5) 按 5.3.1 节介绍的方法运行分析器分析给定的源文件，如果运行时有错，则调试、修改你所编写的 `MiniJOOL.cup`，直至由它能得到正确的语法分析器。
- 6) 在 `MiniJOOL.cup` 中添加语义动作，使之能边分析边构造 AST。
- 7) 再按上述 3)~5) 的方法进行反复生成、修改、编译和调试，直至你所编写的 `MiniJOOL.cup` 能为一个源程序得到正确的 AST。

- 8) 在你的文法规范文件中增加错误恢复部分的规则或代码，再按上述 3)~5)的方法进行反复生成、修改、编译和调试，直至你所编写的 MiniJOOL.cup 能为一个源程序得到正确的 AST。
- 9) 编写 6 个以上合法的源文件作为测试用例，来测试你的语法分析器。

你可以在本章课程设计的测试总控类 Main 中增加一个 runPart2mj()方法，该方法的方法体与 runPart2 类似，只是其中调用的是 MiniJOOL 分析器；然后让 main 方法调用 runPart2mj。之后，你就可以在 Eclipse 中进行调试了。

## 5.4 课程设计 3-3: 用 JavaCC 生成一个语法分析器

相对于 CUP、YACC 这些生成自底向上分析器的工具，JavaCC 能生成自顶向下(递归下降)的分析器。JavaCC 支持 LL(k)文法，它比 CUP 和 YACC 能支持更多形式的语法，不过它要求文法是非左递归的。在 JavaCC 的文法规范文件中，不仅可以描述语言的语法规则，而且可以描述词法规则，这便于语法规则读取和维护词法记号。

在这个课程设计中，你将学习用 JavaCC 为 MiniJOOL 语言或其子集描述词法和语法规则，生成相应的分析器。

### 5.4.1 示例 1: 不带错误恢复的语法分析器

我们先介绍如何用 JavaCC 生成一个能分析 5.2.2 节的赋值语句序列的分析器。这个示例帮助你熟悉 JavaCC，了解在 JavaCC 文法规范文件中如何描述词法和语法以及要执行的动作，为开展自己的课程设计做准备。

先看一下这个示例关联的文件：

- **config/JJ/expr.jj**: JavaCC 文法规范文件，描述赋值语句序列的词法和语法；
- **bin/lab3-3-expr.xml**: 一个 ant 编译文件，用来控制由 expr.jj 生成分析器类 ExprParser.java、记号管理器类 ExprParserTokenManager.java、常量定义类 ExprParserConstants.java，以及 Token.java、SimpleCharStream.java、ParseException.java、TokenMgrError.java 这些对任何 JavaCC 文法规范文件都相同的类，所有这些文件被输出到 src/edu/ustc/cs/compile/parser/jjexpr 目录下；接下来，ant 会继续编译、运行所生成的分析器。
- **config/lab3-3-expr.xml**: 用实验平台运行本示例所需的配置文件。

在这个示例中使用 JavaCC 工具，你需要确认它是否已经安装和配置好（参见 1.4.1 节）。

接下来的工作分两步走：一是编写文法规范文件 expr.jj；二是对 jj.xml 进行 ant build，产生分析器源代码，接着编译分析器，再运行这个分析器进行测试。我们将在下面两小节中分别叙述。

#### 5.4.1.1 编写 jj 文法规范文件

在我们提供的 expr.jj 文件中，涉及到 JavaCC 文法规范文件中的两部分内容：一是 Java

编译单元的定义；二是产生式定义部分。后者又依次包括描述词法的正规式产生式和描述语法的BNF产生式。下面将分别对它们进行介绍。

### 1、Java 编译单元

这一部分包含在“PARSER\_BEGIN(ExprParser)”和“PARSER\_END(ExprParser)”之间。所谓Java编译单元是Java语言的方言，它是指一个Java文件的全部内容，包括0个或1个package声明、0个或多个import声明以及分析器类的声明，等等。在expr.jj文件中，这部分代码示意如下：

```
PARSER_BEGIN(ExprParser)
package edu.ustc.cs.compiler.parser.jjexpr;
import java.util.*;
.....
public class ExprParser{
    // 数据成员的声明
    private AST ast = AST.newAST(AST.JLS3);
    private Block root = null;
    .....
    // 方法成员的定义
    .....
}
PARSER_END(ExprParser)
```

上面的Java编译单元声明了待生成的分析器类ExprParser，其中的数据成员ast和root分别为分析器构造AST所需的AST类的实例以及生成的AST树的根节点。parse方法是进行分析的总控方法，其中会调用开始符对应的方法，即sequence方法。需要注意的是，分析时可能抛出ParseException异常，因此需要在parse方法中捕获这类异常并作处理。

JavaCC将根据expr.jj生成与该文法密切相关的三个类文件：

- ExprParser.java：能分析赋值语句序列的分析器；
- ExprParserTokenManager.java：赋值语句序列分析器中的记号管理器(即词法分析器)；
- ExprParserConstants.java：包含一串有用的常量，主要是记号的编号及值等信息；

以及四个不依赖于具体文法的类文件，即：

- Token.java：描述一个记号的类；
- ParseException.java：语法分析的异常类；
- TokenMgrError.java：词法分析的异常类；
- SimpleCharStream.java：字符流的处理类。

在生成的分析器类ExprJJ.java中将包含上面的Java编译单元中的一切，并且ExprJJ类声明中的后部将包含生成的分析器代码。

### 2、正规式产生式

正规式产生式用来定义语言的词法。在expr.jj中，含有SKIP和TOKEN两种正规式产生

式(如图 5-1)。前者定义词法分析器要跳过的字符或字符串；后者定义在描述语言语法时要使用的词法记号，即终结符。除了SKIP和TOKEN外，JavaCC还提供SPECIAL\_TOKEN和MORE两种正规式产生式，你可以查阅 5.7 节或JavaCC的说明文档来了解它们。

<pre> SKIP : {     " "     "\t"     "\n"     "\r" } </pre>	<pre> TOKEN : {     &lt; IDENTIFIER: ["a"-"z", "A"-"Z"] (         ["a"-"z", "A"-"Z", "_", "0"-"9"] ) * &gt;     &lt; INTEGER_LITERAL: ( ["0"-"9"] ) + &gt;     &lt; LPAREN: "(" &gt;     &lt; RPAREN: ")" &gt;     &lt; SEMICOLON: ";" &gt;     &lt; PLUS: "+" &gt;     &lt; MINUS: "-" &gt;     &lt; MULT: "*" &gt;     &lt; DIV: "/" &gt;     &lt; EQ: "=" &gt; } </pre>
--	--

图 5-1 expr.jj 文件中的正规式产生式

图 5-1 左边描述了词法分析器需要跳过的空白符，而右边则描述了赋值语句序列所涉及的词法记号。这里，每一记号定义包含在一对尖括号中；冒号左边是记号名，如IDENTIFIER；冒号右边是描述这个记号的正规式，如描述IDENTIFIER是以字母开头并由字母、数字或下划线组成的串。

### 3、BNF 产生式

在 expr.jj 文件中，TOKEN 正规式产生式之后都是 BNF 产生式，它们定义赋值语句序列的语法以及构造 AST 的动作代码。

以表示赋值语句序列的产生式“sequence ::= ( assignment ";" ) \*”为例，它在 expr.jj 中对应 BNF 产生式如下所示：

```

1. Block sequence() :
2. {
3.     Block block;
4.     ArrayList seq;
5.     Assignment as;
6. }
7. {
8.     {
9.         block = ast.newBlock();
10.        seq = new ArrayList();
11.    }
12.    (
13.        as = assignment( ) ";"

```

```

14.      {
15.          seq.add(ast.newExpressionStatement(as));
16.      }
17.  )*
18.  <EOF>
19.  {
20.      block.statements().addAll(seq);
21.      success = true;
22.      return block;
23.  }
24. }

```

第1行是BNF产生式的LHS,后跟冒号。JavaCC采用方法声明的形式来描述非终结符。故LHS符号名sequence即为方法名,Block为返回值类型,这里sequence方法不带参数。

第2~6行是由花括号包含的Java代码块,这里放了3个局部变量声明。

第7~24行是由花括号包含的各个可选的展开式,这里只有一个展开式。第8~11、14~16、19~23行都是Java代码块,用来表示动作。第12、13、17、18行体现了RHS的组成。“as = assignment( )”表示RHS中的非终结符号assignment,它写成方法调用形式,方法的返回值被赋给变量as。第17行的“\*”说明将重复第13行若干次。第18行是终结符EOF,所有的终结符名都由一对尖括号包含。

JavaCC将根据各个Java代码块在BNF产生式中的位置,将它们依次复制到生成的非终结符所对应的方法的方法体中。

在JavaCC中,BNF产生式的“|”运算不仅可以选择各个展开式,而且可以出现在一个展开式里,用来描述该展开式中一个局部组成的选择关系(或关系)。这可以从expr.jj中expression对应的BNF产生式看到,这里expression的RHS被改造成如下结构:

```
e1=term() ( <PLUS> | <MINUS> ) e2=term() )*
```

#### 4、小结

从expr.jj中,你可以看到jj文件与cup文件的如下一些不同:

1) 在jj文件中,不需要先声明非终结符名及其类型,而直接以类似Java方法声明的形式定义它对应的产生式;你也不需要声明终结符的类型,它统一为Token类型。

2) 在jj文件中,没有提供优先级的声明机制,因此你需要在定义文法的产生式时处理终结符的优先关系。

3) 在jj文件中,对每个BNF产生式定义,你可以根据需要定义LHS符号对应的方法的参数和返回值类型,并在RHS符号中,以方法调用的形式调用符号对应的方法。非终结符的参数和返回值以及终结符的返回值(Token对象)为在分析中上下传值提供了灵活的渠道。然而在cup文件中,要想在分析中上下传值,就只能通过非终结符或终结符的返回值、和action\_obj对象来进行。

在expr.jj中只使用了JavaCC文法规范的部分特征,你可以参考5.7节或JavaCC说明文档来了解更多的JavaCC的用法。

### 5.4.1.2 分析器的生成、编译和运行

在编写完 `expr.jj` 后，就可以运行 JavaCC 生成分析器的源代码了。为了便于分析器的生成、编译和运行，我们提供了一个 ant 编译文件 `lab3-3-expr.xml`。

由于 JavaCC 生成的类文件较多，所以在 `lab3-3-expr.xml` 文件中的第 15~17 行分别定义了宏 `PARSER_DIR`、`PARSER_PACKAGE` 和 `PARSER_CLASS`，用来指定生成的类文件相对于 `src` 的输出路径、包名以及分析器类名。

```
<property name="PARSER_DIR" value="edu/ustc/cs/compile/parser/jjexpr"/>
<property name="PARSER_PACKAGE" value="edu.ustc.cs.compile.parser.jjexpr"/>
<property name="PARSER_CLASS" value="ExprParser"/>
```

在 `lab3-3-expr.xml` 中定义有 `javacc` 和 `build` 两个任务组，分别表示“生成分析器”、“编译分析器”。下面是名为 `javacc` 的任务描述。

```
<target name="javacc" description="生成分析器">
  <mkdir dir="${SRC_DIR}/${PARSER_DIR}"/>
  <java fork="yes" classpath="${CLASSPATH}" classname="javacc">
    <arg value="-STATIC=false"/>
    <arg value="-OUTPUT_DIRECTORY=${SRC_DIR}/${PARSER_DIR}"/>
    <arg value="${CFG_DIR}/JJ/expr.jj"/>
  </java>
</target>
```

其中，`java` 元素中的属性 `classname` 用来指定要执行的主类为 `javacc`。`classpath` 子元素用来说明要用到的类路由 `CLASSPATH` 指定。`arg` 子元素设置执行所用的命令行参数，这里 `STATIC` 选项被设置为 `false`，表示生成的分析器和记号管理器中的所有方法和类变量都不指定为 `static`；`OUTPUT_DIRECTORY` 选项设置生成的类文件的输出路径，最后的 `JJ/expr.jj` 是要处理的 `jj` 文件。

`build` 任务组描述用 `javac` 编译生成的类文件。

```
<target name="build" description="编译语法分析器">
  <mkdir dir="${DEST_DIR}"/>
  <javac debug="on" classpath="${CLASSPATH}"
    srcdir="${SRC_DIR}" destdir="${DEST_DIR}"
    includes="${PARSER_DIR}/*.java"/>
</target>

<target name="parse" depends="compile" description="运行分析器">
  <java classpath="${CLASSPATH}"
    classname="edu.ustc.cs.compile.parser.jjexpr.ExprParser">
    <classpath refid="CLASSPATH"/>
    <arg path="${basedir}/test/exp_list.txt"/>
  </java>
```



</target>

你也可以直接执行类 ExprParser 来对输入的文件进行语法分析并查看生成的 AST 示意图。你还可以修改类 Main 中的 main 方法使之调用 runPart3 方法，再执行这个 Main 类来对指定的文件进行语法分析并查看生成的 AST 示意图。

## 5.4.2 示例 2：带错误恢复的语法分析器

在这一节中，你需要在上一节的示例 1 基础上，为语法分析器添加错误恢复能力。先看一下这个示例关联的文件：

- **config/JJ/expr\_err.jj**: JavaCC 文法规范文件，描述赋值语句序列的词法和语法；
- **bin/lab3-3-expr-err.xml**: 一个 ant 编译文件，用来控制由 expr\_err.jj 生成分析器类 ExprParser.java、记号管理器类 ExprParserTokenManager.java、常量定义类 ExprParserConstants.java，以及 Token.java、SimpleCharStream.java、ParseException.java、TokenMgrError.java 这些对任何 JavaCC 文法规范文件都相同的类，所有这些文件被输出到 src/edu/ustc/cs/compile/parser/jjexpr 目录下；接下来，ant 会继续编译、运行所生成的分析器。
- **config/lab3-3-expr-err.xml**: 用实验平台运行本示例所需的配置文件。

由于 JavaCC 基于 LL(k)文法构造分析器，它所提供的错误恢复功能强大且使用简单。在这里我们通过改进 expr.jj 得到 expr\_err.jj，后者加入了简单的错误恢复机制，使得分析器能够跳过不正确的表达式，进而继续分析输入的文件。

JavaCC 提供两种错误恢复：浅恢复(Shallow Error Recovery)和深恢复(Deep Error Recovery)的错误恢复机制。它们有不同的应用范围，其中浅恢复用于恢复在分析中不能匹配当前非终结符的所有产生式 RHS 分支时的情况；深恢复则是指当前已选择某产生式 RHS 分支进行匹配，但是在按该分支进行分析起见遇到错误的情况。

### 5.4.2.1 浅恢复(Shallow Error Recovery)

浅恢复是 JavaCC 提供的最简单的错误恢复机制。如下是 expr.jj 中处理赋值表达式的规则，这里略去语义动作部分的内容：

```
Assignment assignment() :
{.....}
{ <IDENTIFIER> <EQ> expression() }
```

如果不加任何错误恢复机制的话，在碰见一个错误的 assignment 时，JavaCC 生成的分析器会抛出一个错误并终止对后续程序的分析。

我们在文法规范文件的最后可以添加如下的一个 JAVACODE 产生式，它以“JAVACODE”为引导，后跟一个错误恢复函数 error\_skipto，这个函数会创建一个 ParseException 异常，然后抛弃所遇到的每一个 token 直到遇到第一个分号(即 SEMICOLON)为止。

```
JAVACODE
```

```

void error_skipto(int kind) {
    ParseException e = generateParseException(); // generate the exception object.
    System.out.println(e.toString()); // print the error message
    Token t;
    do {
        t = getNextToken();
    } while (t.kind != kind);
}

```

然后，可以修改 assignment 的规则定义，增加一条 “error\_skipto(SEMICOLON)” 产生式 RHS 分支：

```

Assignment assignment() :
{
{
    <IDENTIFIER> <EQ> expression()
    | error_skipto(SEMICOLON)
}
}

```

这样当生成的分析器在分析时如果遇到不能匹配 “<IDENTIFIER> <EQ> expression()” 的错误，就会调用 error\_skipto(int kind)函数来进行错误恢复。你可以进一步修改 error\_skipto 函数，在其中增加错误定位信息的输出等。

#### 5.4.2.2 深恢复(Deep Error Recovery)

浅恢复有它的局限之处，例如当赋值语句序列语言的分析器分析形如 “id 34;” 这样的语句时，当遇到 id 会选择 “<IDENTIFIER> <EQ> expression()” RHS 分支进行分析，但是之后由于没有赋值号，会导致语法分析出错。

在这种情况下，可以用深恢复来解决，例如，进一步在 assignment 的规则中增加 try-catch-finally 形式的深恢复：

```

Assignment assignment() :
{
{
    try{
        <IDENTIFIER> <EQ> expression()
    }catch (ParseException pe){
        error_skipto(SEMICOLON);
    }
    | error_skipto(SEMICOLON)
    ...
}
}

```

这样，当用生成的分析器分析错误的赋值语句时，如果产生错误，就会向上层结构抛出异常。这里使用 try-catch 结构正是为了捕获这样的异常，使之不再向上抛出，并跳过下一

个 SEMICOLON。

### 5.4.2.3 分析器的生成、编译和运行

带错误恢复的语法分析器的生成、编译和运行与示例 1 基本相同。只要将 ant 配置文件改为 bin/lab3-3-exp-err.xml，实验平台配置文件改为 config/lab3-3-exp-err.xml。

### 5.4.3 课程设计任务

参考 5.4.1 节所讲的例子和 5.7 节的JavaCC语法规则或者JavaCC的说明文档，你就可以着手为MiniJOOL语言或其子集生成一个语法分析器。这个语法分析器不对输入的程序进行语义分析，并且在分析过程中要构造相应的AST。和课程设计 3-2 一样，你需要理解Eclipse JDT中的AST节点种类及结构，为MiniJOOL语言或其子集的语法结构选择合适的ASTNode类型或List类型。针对一种语言，你需要依次完成以下任务：

- 1) 编写描述语言语法的不含任何语义动作代码的 jj 文件，并将它保存在 config/JJ/MiniJOOL.jj。

你不必从零开始写这个 jj 文件。由于 MiniJOOL 是 Java 语言的一个子集，所以你可以从下载的 javacc 软件包中的 examples\JavaGrammars 下选取某一 Java 版本的 jj 文法，对它进行删减，从而修改成 MiniJOOL 语言或其子集的 jj 文法规范文件。

- 2) 参照 lab3-3-expr.xml，写一个 ant 编译文件，假设为 lab3-3-mj.xml，使之能根据 MiniJOOL.jj 生成分析器类的源代码，然后编译这些类文件，再运行测试所生成的分析器。
- 3) 用 bin/lab3-3-mj.xml 对所编写的 MiniJOOL.jj 文件进行分析器生成、编译、修改和调试，直至由 MiniJOOL.jj 能得到正确的语法分析器。
- 4) 在 MiniJOOL.jj 中添加构造 AST 的语义动作，使之能边分析边构造 AST。
- 5) 再重复 3)对所编写的 MiniJOOL.jj 进行修改和调试，使之能为一个合法的源程序构造正确的 AST。
- 6) 在 MiniJOOL.jj 中添加错误恢复并进行修改和调试，使得分析器具有错误定位与恢复能力。
- 7) 你可以使用在课程设计 3-2 中编写的 MiniJOOL 源文件作为测试用例，来测试你的语法分析器；你还可以另外再重新编写一些 MiniJOOL 源文件作为测试用例。

## 5.5 课程设计 3-4: 编写一个语法分析器的生成器

## 5.6 CUP 与 YACC

CUP<sup>[2]</sup>是一个类 YACC 的、以 LALR(1)文法为基础的语法分析器的生成工具，它能由一个语法规则文件(扩展名为 cup)转换成可分析输入文件是否符合该语法规则的 Java 程序。

为了便于熟悉 YACC 的读者了解 CUP 与 YACC 之间的差异，我们在下面的叙述中，将

会对这两种工具进行一些比较。

### 5.6.1 YACC 简介

最初的 YACC<sup>[3]</sup>是由 Bell 实验室于上世纪 70 年代中期推出的，称为 AT&T YACC。随后出现了许多变种，如 Berkeley 的 YACC（简称 BYACC）<sup>[4]</sup>和 GNU（GNU's Not Unix）的 Bison<sup>[5]</sup>。comp.compilers 新闻组中的讨论认为这三者之间的差异主要表现在政治上，而不是功能上<sup>[6]</sup>。AT&T YACC 属于那些拥有 Bell 实验室 Unix 源码的人，它迄今仍有所有权归属问题。BYACC 在用户遵循 Berkeley 的“*just don't sue us*”的许可下可以被自由地使用。而 Bison 源自 BYACC，由 GNU 自由软件组织开发和维护，它增加了许多额外的特征。

三者输入的文法格式是一样的。一般来说，它们生成的语法分析器在功能上看不出什么明显的差异。但是笔者通过深入分析 BYACC v1.8 和 Bison v1.35，它们在生成的 LALR(1) 分析表和冲突的处理上还是略有差别的：

(1)BYACC 比 Bison 多加了一个非终结符 \$accept 和一个编号为 0 的规则“\$accept → S”；

(2)为使规则 RHS 内嵌的语义动作都转换到只出现在 RHS 的最右端，即用综合属性的计算模拟继承属性的计算，YACC 会自动加入产生  $\cdot$  的标记非终结符，这些标记非终结符的前缀名在 BYACC 中是“\$”，而在 Bison 中则是“@”；

(3)BYACC 生成的语法分析器中状态 0 和 1 分别固定为起始状态和接受状态，而 Bison 生成的语法分析器中状态 0 为起始状态，接受状态编号则在最后；

(4)当文法中存在移进动作和多个归约动作相互冲突时，BYACC 和 Bison 均只报出这些冲突的一部分，二者在所报信息的选取上不完全一样。

### 5.6.2 CUP 与 YACC 的文法规范文件的结构

在 YACC 中，文法规范文件的扩展名是 y。图 5-2 示意了 .y 和 .cup 文件的结构。其中，**声明部分**声明符号及其类型、终结符的优先级和结合性。**规则**包括语法规则的定义以及归约后要执行的语义动作(程序代码)。默认的开始符是第一个规则的 LHS。**用户代码**是任何合法的程序代码，在 .cup 中它们被包在 { : } 块中，在 .y 中则无须特别的括号，这些用户代码将直接被复制到生成的语法分析器代码中。

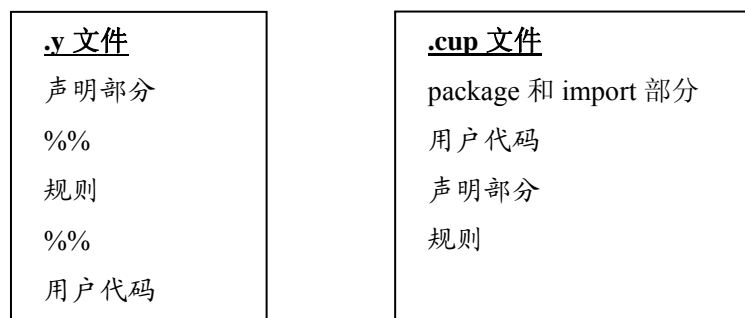


图 5-2 YACC 与 CUP 文法规范文件的结构

在 .cup 文件中允许出现以下几种可选的用户代码块：

- 1) `action code{ : 代码 :}` CUP 将在生成的语法分析器类中生成单独的、非 `public` 的内部类来包含该块中的代码，用以封装为内嵌在语法规则中的语义动作代码所使用的变量和方法。
- 2) `parser code{ : 代码 :}` 这类代码块与 `action code` 非常类似，不过 `parser code` 中的代码内容将被复制到生成的语法分析器类中。
- 3) `init code{ : 代码 :}` 该块中的代码是在语法分析器请求第一个记号之前执行的。一般是用来初始化词法分析器以及语义动作所需要的各种表及其它数据结构。
- 4) `scan code{ : 代码 :}` 该块中的代码指出语法分析器怎样取得下一个记号。如果词法分析器返回的是一个 `java_cup.runtime.Symbol` 对象，则 `scan with` 中的代码返回的也应该是一个 `Symbol`。

### 5.6.3 文法符号

本节将介绍YACC和CUP中文法符号的定义与使用的差异。你可以结合 5.6.4 节给出的例子来消化理解。

#### 5.6.3.1 类型定义

一般地，在.y文件的“**声明部分**”需要用“`%union{...}`”定义一个共用体类型，来说明各种符号的类型。例如，在图 5-3 中的.y文件里，这个共用体类型包含两种成员，一是ival表示整数，一是sval表示字符指针。

在.cup 中无须专门定义类型来说明符号的类型，而是可以直接使用已有的 Java 类型、类库中的类或者是用户自定义的类。

#### 5.6.3.2 终结符的定义与使用

在.y 文件的“**声明部分**”中，可以用“`%token [类型] 终结符名[, ..., 终结符名]`”来声明终结符名。例如，“`%token PLUS MINUS EQUALS`”和“`%token <sval> NAME`”，后者声明了一个名为 NAME 的终结符，其值将取共用体类型中的 sval 成员。在.y 文件中，允许规则 RHS 直接使用终结符对应的串表示，如‘`{`’。

在.cup 文件的“**声明部分**”中，可以用“`terminal [类型] 终结符名[, ..., 终结符名];`”为终结符声明名字和类型。例如，“`terminal PLUS, MINUS, EQUALS;`”和“`terminal String NAME;`”。不过，cup 文法规范不允许在规则 RHS 中使用终结符对应的串表示，而必须使用终结符对应的名字。

“**声明部分**”还可以指定终结符的优先级和结合性。在.y 文件中，可以使用以关键字`%left`、`%right`或`%nonassoc`为开始、后跟一组终结符列表的若干行来说明终结符的优先级和结合性。出现在同一行的终结符有相同的优先级和结合性，结合性由这行的关键字决定，`left`表示左结合，`right`表示右结合，`nonassoc`表示不具有结合性。终结符间的优先级取决于它所在的行，越在后的行则优先级越高。

在.cup 文件中，可以类似地使用以下声明来说明终结符的优先级和结合性，其中优先级与结合性的确定方式与 YACC 类似。所有没有用 `precedence` 声明的终结符被处理成具有最

低的优先级。当有移进-归约(shift/reduce)错误时，如果涉及的两个终结符都没有指定优先级，则无法根据优先级来解决冲突。

```
precedence left   终结符名[,..., 终结符名];
precedence right  终结符名[,..., 终结符名];
precedence nonassoc  终结符名[,..., 终结符名];
```

### 5.6.3.3 非终结符的定义

在.y 文件中无须在“**声明部分**”事先声明规则中将使用的所有非终结符，而只需要使用“%type [类型] 非终结符名[,..., 非终结符名]”来为某些需要返回值的非终结符指定其值的类型。你可以用“%start 开始符名”来指定语法分析的开始符。

在.cup 文件中必须在“**声明部分**”声明规则中将使用的所有非终结符，其声明的格式是“non terminal [类型] 非终结符名[,..., 非终结符名];”。来为某些需要返回值的非终结符指定其值的类型。你可以用“start with 开始符名”来指定语法分析的开始符。

### 5.6.3.4 符号的值

.y 文件中的每个符号都可以有值，符号的类型可以不一样。你可以在语义动作中用“\$”引用规则的 LHS，用“\$1”、“\$2”等自左至右依次引用 RHS 中的符号。在缺省情况下，BYACC 会把“\$1”的值传给“\$”。

在.cup 中，你需要在语义动作中用“RESULT”引用 LHS。如果你希望 LHS 有值，就必须在语义动作中显式地将 RHS 符号的值按所需的要求计算并赋值给 RESULT。对于 RHS 符号，如果要使用符号的值，则需要为这些符号引入标号，如 exp:e1 表示为符号 exp 引入 e1 标号，这样就可以通过标号 e 来使用符号 exp 的值了。

## 5.6.4 一个简单的例子

### 表达式的词法规范—flex 文件

```
%{ #include "y.tab.h" %}
%%
[0-9]+ { yyval.ival = atoi(yytext);
        return NUMBER; }
[ \t] /* ignore whitespace */
\n    return 0; /* logical EOF */
"+"   return PLUS;
"-"   return MINUS;
...
%%
```

### 表达式的词法规范—jflex 文件

```
import java_cup.runtime.Symbol;
%%
%cup
%%
[0-9]+ { return new Symbol ( sym.NUMBER,
                          new Integer (yytext())); }
[ \t] { /* ignore whitespace */ }
\n { return new Symbol (sym.EOF); }
"+" { return new Symbol (sym.PLUS); }
"-" { return new Symbol (sym.MINUS); }
...
```

图 5-3 表达式的词法描述：flex 文件和 jflex 文件

我们以一个简单的表达式分析器为例，让你理解 CUP 文法规范与 YACC 文法规范的区别。这里，我们仍然用工具来生成词法分析器，并让它与生成的语法分析器一起协同工作，来完成对输入文件的分析。

在前面，我们已经学习了用 JFlex 与 CUP 协同工作来为语言生成词法分析器和语法分析器。而针对 YACC，你就需要一个能生成 C 语言代码的词法分析器的生成工具，例如 Flex。图 5-3 给出了用 Flex 和 JFlex 的词法规范格式描述的表达式的词法规范。图 5-4 给出了用 YACC 和 CUP 来描述的表达式文法规范。

#### 表达式的语法描述：expr1.y 文件

```
%union {
    int ival;
    char *sval;
}
%token PLUS MINUS EQUALS
%token <sval> NAME
%token <ival> NUMBER
%type <ival> expression
%%
statement : NAME EQUALS expression { }
    | expression { printf (" = %d\n", $1); }
    ;
expression : expression PLUS NUMBER
    { $$ = $1 + $3; }
    | expression MINUS NUMBER
    { $$ = $1 - $3; }
    | NUMBER { $$ = $1; }
    ;
extern FILE *yyin;
int yyerror (char *s)
{
    fprintf (stderr, "%s\n", s);
}
int main ()
{
    if (yyin == NULL) yyin = stdin;
    while (!feof(yyin)) yyparse();
}
```

#### 表达式的语法描述：expr1.cup 文件

```
parser code
{
    public static void main (String argv[])
        throws Exception
    { new parser(new Yylex(System.in)).parse(); }
}
terminal PLUS, MINUS, EQUALS;
terminal String NAME;
terminal Integer NUMBER;
non terminal statement;
non terminal Integer expression;
statement ::= NAME EQUALS expression:e
    | expression:e
    { : System.out.println(" = " + e); : }
    ;
expression ::= expression:e PLUS NUMBER:n
    { : RESULT = new Integer(e.intValue()
        + n.intValue()); : }
    | expression:e MINUS NUMBER:n
    { : RESULT = new Integer(e.intValue()
        - n.intValue()); : }
    | NUMBER:n
    { : RESULT = n; : }
    ;
```

图 5-4 表达式的语法描述：y 文件和 cup 文件

运行命令：

bison -d -y expr1.y

或 byacc -d expr1.y

可以根据 expr1.y 生成分析器源码，它们将默认地输出到 y.tab.c 和 y.tab.h 中，其中 y.tab.c 中

的函数 `yyparse` 负责控制对输入串的扫描、移进和归约处理。

运行命令

```
java java_cup.Main expr1.cup
```

将生成语法分析器类文件 `parser.java` 和词法记号类文件 `sym.java`。`parser` 对象中的 `parse()` 方法负责完成对输入串的扫描、移进和归约。

### 5.6.5 错误恢复

CUP 使用与 YACC 一样的错误恢复机制。CUP 提供一个特殊的非终结符 `error` 来支持错误恢复，这个 `error` 可以匹配一个错误的输入序列。

## 5.7 JavaCC

JavaCC 是一种采用递归下降分析的、支持 LL(k)文法的编译器的编译器。它不仅可以为输入的文法规范文件（文件扩展名为 `jj`）生成对应分析器的 Java 源代码，还提供其他与分析器生成有关的能力，如树构造（通过包含在 JavaCC 中的 JJTree 工具）、动作（actions）、调试等。

在下面各小节中，将依次介绍 JavaCC 文法规范文件的结构与定义，以及 JJTree 和 JJDoc 等工具。

### 5.7.1 JavaCC 文法规范文件的结构

一个 JavaCC 文法规范文件（扩展名为 `jj`）开始于可选的选项清单，后跟包含在“`PARSER_BEGIN(name)`”和“`PARSER_END(name)`”之间的 Java 编译单元，之后是产生式清单。它形式地描述为：

```
javacc_input ::=  javacc_options
                  "PARSER_BEGIN" "(" <Identifier> ")"
                  java_compilation_unit
                  "PARSER_END"  "(" <Identifier> ")"
                  (production)*
                  <EOF>
```

“`PARSER_BEGIN`”和“`PARSER_END`”后的 `name` 必须是一样的，这个名字用来确定生成的分析器的名字。如果 `name` 是“`MyParser`”，则将生成三个依赖于文法的类文件 `MyParser.java`、`MyParserTokenManager.java`、`MyParserConstants.java`，以及四个独立于文法的类文件 `Token.java`、`ParseException.java`、`TokenMgrError.java`、`SimpleCharStream.java`。后面四个文件可以在任何文法之间复用。

“`PARSER_BEGIN`”和“`PARSER_END`”之间是一个正规的 Java 编译单元（在 Java 方言中，一个编译单元是一个 Java 文件的全部内容）。它可以是任意的 Java 编译单元，只要它包含要生成的分析器的类声明。



如果编译单元包含一个 `package` 声明, 则在所有生成的类中都将包含这个声明。如果编译单元包含一些 `import` 声明, 则在生成的分析器类和记号管理器类中将包含这些声明。

生成的分析器文件包含编译单元中的一切, 此外, 在分析器类结尾还包含生成的分析器代码。在生成的分析器类中, 对于文法规范文件中的每一非终结符都有一个对应的 `public` 方法声明。通过调用与非终结符对应的方法, 可以完成关于该非终结符的分析。与 CUP、YACC 不同的是, 在 JavaCC 中没有单一的开始符, 分析可以从文法中的任意非终结符开始。

生成的记号管理器提供 `public` 的 `getNextToken` 方法, 用来返回下一个记号(Token), 这个方法会抛出 `TokenMgrError` 异常。

### 5.7.2 JavaCC 选项

JavaCC选项由保留字“options”开始, 后跟由一对花括号包含的一个或多个选项绑定。每个选项绑定指定一个选项的设置。同一个选项不能被设置多次。表 5-1 列出了JavaCC中部分选项的名字、值类型及含义。

表 5-1 JavaCC 部分选项及含义

选项名	值类型	含义
LOOKAHEAD	整型	分析时对选择点做决定前向前看的记号数。缺省值为 1。这个值越小, 则分析器速度越快。
CHOICE_AMBIGUITY_CHECK	整型	缺省值为 2。当一个非终结符的规则中含有多个前缀相同的形如 “A   B  ...” 的 RHS 分支时, 为消除歧义最多向前查看的记号数。
OTHER_AMBIGUITY_CHECK	整型	缺省值为 1。指其他形式(如“(A)*”、“(A)+”、“(A)?”)的 RHS 分支存在歧义时, 为消除歧义最多向前查看的记号数。
STATIC	布尔型	缺省值为 <code>true</code> , 表示生成的分析器和记号管理器中的所有方法和类变量都指定为 <code>static</code> 。
DEBUG_PARSER	布尔型	缺省值为 <code>false</code> 。该选项用于从生成的分析器中获取调试信息。
DEBUG_LOOKAHEAD	布尔型	缺省值为 <code>false</code> 。该选项用于从生成的分析器中获取调试信息。
USER_TOKEN_MANAGER	布尔型	缺省值为 <code>false</code> , 表示生成在指定文法记号上工作的记号管理器。如果设为 <code>true</code> , 则生成的分析器接受来自任意 <code>TokenManager</code> 类型的记号管理器。

### 5.7.3 产生式

在 JavaCC 文法规范文件中，可以出现有四类产生式，即：

```
production ::= javacode_production
           | regular_expr_production
           | bnf_production
           | token_manager_decls
```

其中，JAVACODE 产生式 `javacode_production` 和 BNF 产生式 `bnf_production` 用于定义文法，分析器将由这些文法而生成；正规式产生式 `regular_expr_production` 用于定义文法的记号，记号管理器将根据这些信息以及内联在分析器文法中的记号规范来生成；记号管理器声明 `token_manager_decls` 用于定义一些声明，这些声明将插入到生成的记号管理器中。

#### 5.7.3.1 JAVACODE 产生式

JAVACODE 产生式提供一种为某些产生式写 Java 代码的方式，用来取代通常的 EBNF 扩展。其形式定义如下：

```
javacode_production ::= "JAVACODE"
                    java_access_modifier java_return_type java_identifier "("
                    java_parameter_list ")"
                    java_block
```

#### 5.7.3.2 BNF 产生式

BNF 产生式是用于描述文法的标准产生式。每个 BNF 产生式有一个非终结符充当 LHS，然后在 RHS 用 BNF 扩展来定义该非终结符。其形式定义如下：

```
bnf_production ::= java_access_modifier java_return_type java_identifier "("
                  java_parameter_list ")" ":"
                  java_block "{" expansion_choices "}"
```

由于每一非终结符被翻译成生成的分析器中的一个方法，故 JavaCC 采用类似 Java 方法声明的风格来描述非终结符。非终结符的名字是方法名，所声明的参数和返回值是在分析树中上上传值的手段，方法缺省的访问修饰符为 `public`。

BNF 产生式的 RHS 有两部分。第一部分是 `java_block`，即一组任意的 Java 声明和代码 (Java 块)，这些代码将复制到 LHS 非终结符对应的方法的开始处，从而在分析处理中每次使用这个非终结符，就会执行这些声明和代码。这部分的声明对 BNF 展开式中的动作代码是可见的。

第二部分是包含在一对花括号中的 BNF 展开式 `expansion_choices`。`expansion_choices` 是一个或多个由 `"|"` 分隔的展开式，每个展开式为一个展开式单元序列。展开式单元 `expansion_unit` 有如下 6 种形式：

```

expansion_unit ::= local_lookahead
                | java_block
                | "(" expansion_choices ")" [ "+" | "*" | "?" ]
                | "[" expansion_choices "]"
                | [ java_assignment_lhs "=" ] regular_expression
                | [ java_assignment_lhs "=" ] java_identifier "(" java_expression_list ")"

```

其中, `local_lookahead` 指定一组向前看的约束, 它由"LOOKAHEAD"引导, 后跟一组包含在圆括号内的 `lookahead` 约束。`lookahead` 约束有三种: 向前看的记号数、向前看的语法结构(由 `expansion_choices` 指定)、向前看的语义(由包含在花括号内的表达式指定)。`java_block` 是由花括号括起的一组 Java 声明和代码, 称为分析器动作。`regular_expression` 即正规式, 当正规式被匹配时, 会创建 `Token` 对象, 可以将这个对象赋值给由 `java_assignment_lhs` 代表的变量。最后一个分支是一个非终结符, 它被写成方法调用, 方法的返回值可以被赋给变量。不过在 `lookahead` 计算期间, 不会执行上述两种赋值。

### 5.7.3.3 正规式产生式

正规式产生式用于定义被生成的记号管理器所处理的词法实体。其形式定义如下:

```

regular_expr_production ::= [ lexical_state_list ]
                        regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
                        "{" regexpr_spec ( "|" regexpr_spec )* "}"

```

`lexical_state_list` 指定该产生式被应用的一组词法状态, 如果缺省, 则表示应用于称为“DEFAULT”的标准词法状态; 如果为“<\*>”, 则表示应用于所有的词法状态; 否则, 就应用于所指定的、包含在尖括号内的所有词法状态。

`regexpr_kind` 指定该正规式产生式的种类, 可以为以下四种之一:

- **TOKEN**: 说明产生式中的正规式是描述文法中的记号, 记号管理器会为该正规式的每个匹配创建一个 `Token` 对象, 并返回给分析器。
- **SPECIAL\_TOKEN**: 说明产生式中的正规式是描述特殊记号。这些记号在分析中并不重要, 即 BNF 产生式会忽略它们。不过, 记号管理器仍会把特殊记号通过其相邻的实在记号对应的 `Token` 对象里的 `specialToken` 域传递给分析器, 使得分析器的动作代码可以访问它们。特殊记号对处理诸如注释等词法实体很有用。
- **SKIP**: 记号管理器将简单地跳过(忽略)与产生式中的正规式相匹配的串。
- **MORE**: 与这类正规式匹配的串将被缓存, 直到匹配下一个 **TOKEN** 或 **SPECIAL\_TOKEN**; 接着, 缓冲区中的所有匹配将与最后的 **TOKEN** 或 **SPECIAL\_TOKEN** 匹配一起连接形成一个 **TOKEN** 或 **SPECIAL\_TOKEN**, 再传递给分析器。

"IGNORE\_CASE"在正规式产生式中是可选的, 如果有这个选项, 则该产生式是大小写不敏感的。

`regexpr_spec` 是词法实体的实际描述, 它包含一个正规式(见下面的 `regular_expression` 形式定义), 后面可选地跟有一个 Java 代码块(词法动作), 然后是词法状态标识符(这也是可选的)。在词法分析中, 只要匹配这个正规式, 就会执行词法动作, 然后执行一些公共的记号动作。如果在 `regexpr_spec` 中指定了词法状态, 则记号管理器会迁移到该词法状态再进一步地处理(记号管理器初始地从“DEFAULT”状态启动)。

```
regular_expression ::= java_string_literal
                    | "<" [ [ "#" ] java_identifier ":" ] complex_regular_expression_choices ">"
                    | "<" java_identifier ">"
                    | "<" "EOF" ">"
```

如上所示, JavaCC 中的正规式有四种。第一种是 Java 中的字符串文字, 如“++”。第二种是由尖括号括起的复杂正规式, 正规式中可以有“|”、“+”、“\*”、“?”等算符, 分别表示或运算、1 个或多个、0 个或多个、0 个或 1 个; 还可以通过形如“<正规式名>”的形式来引用其他命名的正规式。在正规式中, 包含在方括号内的、由逗号分隔的一个字符(如‘z’)或字符范围(如‘a’-‘z’)用来描述一个字符集合。如果方括号前有“~”符号, 则所表示的字符集合是任意不出现在方括号中的字符。第三种是对其他命名正规式的引用。第四种是预定义的正规式“<EOF>”。需要注意的是, 如果某正规式名以“#”开头, 则这个正规式是私有的, 不能被文法产生式使用。

#### 5.7.3.4 记号管理器声明

记号管理器声明是由“TOKEN\_MGR\_DECL:”引导的一个 Java 代码块。代码块中可以包含一组 Java 声明和语句, 这些代码将被复制到生成的记号管理器中, 可以在词法动作内被访问。

从中可以看出, jj 文件包括以下三个部分

1) **选项(Options):** 包括 LL(k)文法中向前看的 k 值的设置(LOOKAHEAD)、是否将解析器中的方法设置为静态、是否设置为 Debug 状态、是否设置优化、输出文件管理等。这些选项也可以在 javacc 的命令行中输入。

2) **Java 编译单元(Compilation unit):** 在“PARSER\_BEGIN”与“PARSER\_END”之间定义。它包含一个与指定的标识符(Identifier)同名的类声明, 还可以包含 package、import 声明。JavaCC 对此不做语法检查。

一般地, 若标识符为 MyParser, 则 javacc 根据所读入的文法文件产生相符的 MyParser.java(分析器)、MyParserTokenManager.java(记号管理器, Token Manager, 也称扫描器或词法分析器)、MyParserConstants.java(常量定义), 以及对任意文法都相同的 Token.java、JavaCharStream.java、ParseException.java(分析异常)、TokenMgrError.java(记号错误)。生成的解析器中包含与文法文件中每个非终结符相关的 public 方法, 对某非终结符(Nonterminal)相关的解析通过调用与该非终结符相应的方法来完成。与 YACC 不同的是, JavaCC 的开始符不局限于一个, 文法的设计者可根据需要指定多个开始符进行不同语法结构的句子的识别。记号管理器中提供取下一记号的 public 方法: Token getNextToken() throws ParseError。

3) **产生式(Production)**: 它是整个文法的核心, 用来说明语言中的记号以及各记号如何组成合法的句子。

(1) 正规式产生式: 用正规式定义 javacc 生成的记号管理器所处理的词汇实体, 可分为要跳过的符号(SKIP)、前缀符号(MORE)、需要参与语法分析的记号(TOKEN)或称为终结符(Terminal)、不参与解析的专用记号(SPECIAL\_TOKEN)四种。

(2) BNF 产生式: 是定义文法的标准产生式。它描述非终结符在生成的解析器中对应的方法、非终结符的 BNF 展开式(BNF expansions)定义等, 还可嵌入 Java 块增加各种附加的语义以及异常与错误处理。

(3) javacode 产生式: 提供直接写 Java 代码定义非终结符的手段, 从而取代 BNF 产生式。这对文法的设计者要求较高, 他必须十分清楚 javacc 对产生式中选择点以及向前看等的处理机制。

(4) 记号管理器声明: 是以"TOKEN\_MGR\_DECLS:"开始的 Java 块, 这些声明和语句将被写到生成的记号管理器中, 以增添词法分析时的动作。每个 JavaCC 文法文件中只能有一个记号管理器声明。

## 第6章 语义分析

在本章中，你将学习为 SkipOOMiniJOOB 语言完成语义分析。语义分析将源程序中变量和函数的定义与它们的各个使用联系起来，依据语言定义来检查源程序的语义一致性，以保证程序各部分能有意义地结合在一起。语义分析是以语法分析和符号表为基础，重点是类型检查，即检查每个算符的运算对象，看它们的类型是否适当；检查实参与形参的类型是否兼容；根据函数类型，检查函数体中的 `return` 语句是否完整、是否与函数类型相兼容，等等。你需要根据 SkipOOMiniJOOB 语言的静态语义定义，来设计实现语义分析器。

为了便于你掌握语义分析的各个方面的知识，我们安排了四个课程设计，你可以根据实际情况，选做其中的部分或全部。

### 6.1 本章课程设计概述

在本章的课程设计中，我们将循序渐进地进行如下课程设计：

#### 课程设计 4-1 为 AST 构造符号表。

在课程设计 3-1、3-2 或 3-3 中，你学会为一个语言构造一个能生成 AST 的语法分析器。在这些语法分析器实现中，你没有考虑符号表的构造与维护，而符号表是语义分析等的基础。在这个课程设计中，你将为输入的 AST 构造符号表并将其输出。你需要考虑 `int`、`boolean` 和数组类型等的符号要维护的信息，需要考虑全局符号表与局部符号表的组织结构，等等。

#### 课程设计 4-2 对 AST 进行语义分析。

在课程设计 4-1 的基础上，你将对输入的 AST 进行语义分析，这个分析器将利用符号表中的信息来进行类型传递和检查，以便发现 AST 所对应的源程序中更多的语义错误，等等。你可以先构造完整的符号表，再进行语义分析，即分成两遍来实现。你也可以在遍历 AST 时一边构造符号表、一边进行语义分析，即放在一遍中来实现。

#### 课程设计 4-3 在语法分析的同时构造符号表。

你将在课程设计 3-2 或 3-3 的基础上，结合课程设计 4-1 的符号表设计，来修改 `cup` 或 `jj` 文法规范文件，使之能在语法分析的同时能构造符号表，输出 AST 和符号表供后继的语义分析器使用。

#### 课程设计 4-4 在语法分析的同时做语义分析。

在课程设计 4-3 的基础上，你将在语法分析的同时添加更多的源程序语义错误检查代码，从而完善这个分析器。这个分析器的输出结果同样可以有两种方式，一种是只输出合法程序对应的 AST，另一种则不仅输出合法程序对应的 AST，还输出完整的符号表。

在 `ROOT_DIR/lab/lab4` 中给出了这些课程设计的框架代码，对于课程设计 4-2 中需要完成的语义分析器类必须要实现实验平台的 `CheckerInterface` 接口；而对于课程设计 4-4，则直接修改 `cup` 文件或 `jj` 文件，使得由它生成的分析器能进行语义检查。

## 6.2 课程设计指导

### 6.2.1 符号表的设计与组织

在实验平台接口中提供了符号表接口 `SymTable` (见 3.3.1.7 节), 这是一个在类体中没有任何成员的空接口。你需要自行设计、组织符号表, 并使得你实现的符号表类实现这个接口。

在为 `SkipOOMiniJOOl` 语言以及 `MiniJOOl` 语言设计符号表时, 需要注意处理全局变量(类变量)、块中局部变量声明语句所声明的变量以及形参变量的作用域以及同名问题。

由于每个语句块都可能有变量声明语句, 所以可以为每个语句块建立一个独立的符号表, 这个符号表可以使用以符号名为 `key`、以符号对应的描述信息为值(如符号的类型编号; 对于一维数组类型的符号, 还需要记录数组的长度和元素的类型, 等等)的哈希表。你需要考虑各类符号需要在符号表中记载的信息, 使得这些信息能帮助你进行语义分析以至代码生成。

各个语句块(包括类体)都有各自的符号表, 如果要将这些符号表保存并输出给后继的编译器组件的话, 你同样可以采用哈希表来保存这些符号表, 哈希表的 `key` 可以是每个符号表所关联的 `AST` 节点, 哈希表的值则是对应的符号表的引用。当然, 你也可以采取其他的方法来设计、组织这些符号表。

如果符号表只是为当前的编译器组件服务而无需传给后继的组件, 则你没有必要把所有的符号表都保存下来, 而可以采用栈的方式来保存当前编译器组件在工作中作用域有效的所有符号。这时, 每进入一个语句块, 则向符号表栈中增加符号, 而每退出一个语句块, 则需要清除该语句块中声明的符号。

### 6.2.2 对 AST 的语义分析

对 `AST` 的语义分析类同样可以定义为一个访问者类, 它为每种 `AST` 节点定义 `visit` 方法, 通过 `visit` 方法来检查当前传入的 `AST` 节点是否满足语言的静态语义。在语义分析中, 需要使用符号表中的信息, 包括变量以及函数的类型信息等。

在 2.6 节给出了 `SkipOOMiniJOOl` 语言的静态语义, 你需要根据其中的定型规则来设计实现语义分析器。

### 6.2.3 在语法分析的同时进行语义分析

在语法分析的同时进行语义分析需要你: 在语法分析的同时维护符号表, 然后在各个语法结构的语义动作中增加对该类语法结构的语义检查。你需要进一步考虑语义检查发生错误时的错误定位和恢复机制。

## 第7章 MIPS 汇编代码生成

在这一章和下一章中,你将学习为 SkipOOMiniJOOl 程序生成 MIPS 架构以及 x86 架构下的汇编代码。汇编代码生成器是以前端产生的中间表示(如 AST 和符号表信息)为输入的。无论是生成 MIPS 汇编代码还是 x86 汇编代码,在设计汇编代码生成器时都需要考虑存储管理、指令选择、寄存器分配、计算次序选择等问题。

为便于设计实现针对不同架构下的汇编代码生成,我们引入一组汇编代码的统一内部表示类及接口(见 7.2 节),每一种特定的汇编代码(如 MIPS 或 x86)都以此为基础并实现其中的接口。此外,我们还针对 MIPS 和 x86 分别设计实现了寄存器分配器,你可以在汇编代码生成中使用我们提供的寄存器分配器来分配寄存器,也可以自己实现寄存器分配。由于 MIPS 架构有丰富的寄存器且指令相比 x86 要简单得多,为此我们引入全局的寄存器分配器,这个分配器每次接收表示一个函数的指令序列,输出这个指令序列中的寄存器分配前后的映射关系。

本章将重点介绍 MIPS 架构下的汇编代码生成。MIPS CPU 是一种高效简洁的 RISC 结构 CPU。它最早是美国斯坦福大学的一个研究性项目,之后由 MIPS 计算机系统公司获得,并一直发展到今天。MIPS CPU 在高性能计算、信号处理、图像处理、嵌入式系统等多个领域有着广泛的应用。此外, MIPS CPU 的优良结构对许多 CPU 的设计有着重要的影响,比如 DEC 的 Alpha 和 HP 的 Precision。在本书中,我们选用 MIPS 家族中较早的 R2000/R3000 作为代码生成的 RISC 体系的平台。选择 R2000/R3000 主要有以下几点考虑: MIPS 家族之后的大多数成员都衍生自 R2000/R3000; R2000/R3000 的所有寄存器和指令几乎可以不加修改地使用在其后的 MIPS CPU(64 位 CPU 必须进行 64 位扩展)上; R2000/R3000 结构更加简单和易于理解,比较适合在课程实验中使用。

为在普通的计算机上运行所生成的 MIPS 汇编代码,我们选择 SPIM 作为 MIPS R2000/R3000 RISC 架构的目标机的模拟器。SPIM 可以运行在 Linux 和 MS Windows 下。在 Linux 下, SPIM 除提供一个字符界面外,还提供了一个 Xwindow 下的图形界面 XSpim。在 MS Windows 下, SPIM 提供了一个类似 XSpim 的图形界面 PCSpim。

我们将在以下各节依次介绍课程设计的任务、汇编代码的统一表示、MIPS 汇编代码的说明、SPIM 的使用、以及我们提供的 MIPS 寄存器分配器。

### 7.1 本章课程设计概述

在本章的课程设计中,我们将循序渐进地进行如下两个课程设计:

#### 课程设计 5-1 利用现有的寄存器分配器生成 MIPS 汇编代码。

在这个课程设计中,你将使用我们提供的 MIPS 寄存器分配器来设计实现 MIPS 汇编代码生成器。为此,你需要理解我们提供的 MIPS 寄存器分配器的接口和特点,你还需要保证你的汇编代码的内部表示采用我们所提供的形式。



**课程设计 5-2 独立实现完整的 MIPS 汇编代码生成器。**

在这个课程设计中，你可以自行设计和实现汇编代码生成器中遇到的各种问题。

要完成本章所述的课程设计，除了需要使用 Eclipse AST、实验平台库外，还需要使用存放在 ROOT\_DIR/lib 下的 edu.ustc.cs.compile.arch.jar。

## 7.2 汇编代码的内部表示

### 7.2.1 设计概述

一个汇编程序是一组语句序列，其中的语句分为三种类型：伪指令（Directive）、标签（Label）和指令（Instruction）。不同的汇编代码在具体的指令和伪指令的种类上存在差异，而在格式上存在一些共性之处。例如：

- 伪指令是以“.”（不包括引号）开始，后跟一个伪指令码（Director），每种伪指令码可以有 0 个或多个参数。参数可以是字符串、整数。
- 标签是由标签名后跟“:”（不包括引号）组成，表示程序中的某个地址。
- 指令是以一个操作码开始，可以有 0 个或多个操作数。操作数可以是寄存器、寄存器变量、标签及整数。

由于不同汇编语言上存在的上述共性，我们设计以下几个不依赖于具体汇编语言的类（均在 edu.ustc.cs.compile.arch 包下）：

- AssemblyElement：是一个抽象类，表示汇编程序中的一条语句；
- Directive：从 AssemblyElement 继承，表示一条伪指令；
- Instruct：从 AssemblyElement 继承，表示一条指令；
- Label：从 AssemblyElement 继承，表示一条标签语句；
- AssemblySequence：由 LinkedList<AssemblyElement> 派生，表示一组元素类型为 AssemblyElement 的序列。
- RegisterVariable：表示一个寄存器变量。类中包含一个整型实例成员，表示这个寄存器变量的 id，用来唯一识别一个寄存器变量。

伪指令码、指令操作码以及寄存器的种类与具体汇编语言密切相关的，为了使得在上述类表示中不必考虑具体的汇编语言特征，我们定义如下三个接口：

- Director：伪指令码的公共接口；
- OpCode：指令操作码的公共接口；
- Register：寄存器的公共接口。

不同汇编语言的伪指令码、指令操作码以及寄存器表示类将分别实现相应的接口。例如，针对 MIPS 架构，我们在 edu.ustc.cs.compile.arch.mips 包下提供有 MIPSDirector、MIPSOpcode 和 MIPSRegister 三个类。在这三个类中，每个类中都声明了属于这个类的若干个 static final 实例，分别表示这个类中可以取得的各类实例。如：

```
public class MIPSDirector implements Director {
```

```

    /** director ".data" */
    public static final MIPSDDirector DATA = new MIPSDDirector();
    /** director ".text" */
    public static final MIPSDDirector TEXT = new MIPSDDirector();
    .....
}

```

这样，就可以利用这些 `final` 类变量来创建特定类别的伪指令实例，如：

```
direct = new Directive(MIPSDDirector.DATA, null);
```

所有的汇编代码表示类中都提供有 `toString()` 方法，用来输出该类的实例所对应的字符串，为代码发射奠定基础。

## 7.2.2 示例

本节通过例子说明如何使用上述内部表示表达 MIPS 汇编代码。由于我们提供的汇编代码的中间表示分离了与特定汇编语言相关和无关的部分，所以只要替换以下例子中的特定汇编语言相关的部分（伪指令码，操作码，寄存器），即可以使用于其它平台。

以下的例子并没有涵盖我们提供的汇编代码中间表示的所有方面。更加详细的内容可以参考汇编代码内部表示的 API 文档。

### 7.2.2.1 表达没有参数的伪指令

表达伪指令

```
.data
```

可以使用如下代码

```
Directive direct = new Directive(MIPSDDirector.DATA, null);
```

其中，

- 使用类 `Directive` 的对象表示一个伪指令，这个类中有唯一的构造方法。
- 类 `Directive` 的构造方法拥有 2 个参数，第一个参数指明伪指令码，第二个参数提供伪指令需要的参数。
- 伪指令码使用类 `MIPSDDirector` 中定义的 `static final MIPSDDirector` 对象表示，这些对象的名称是伪指令码的大写形式（省略“.”）。例如，`.data` 用 `MIPSDDirector.DATA` 表示，`.text` 用 `MIPSDDirector.TEXT` 表示。
- 如果伪指令不需要参数，将 `null` 作为类 `Directive` 的构造方法的第二个参数。

### 7.2.2.2 表达带有参数的伪指令

表达伪指令

```
.align 2
```

可以使用如下代码

```
ArrayList<Directive.Argument> args = new ArrayList<Directive.Argument>();
```

```
args.add(new Directive.Argument(new Integer(2)));
Directive direct = new Directive(MIPSDirector.ALIGN, args);
```

其中,

- 类 Directive 的构造方法的第二个参数的类型是 List<Directive.Argument>。List 中依次存放伪指令需要的参数。
- 每个参数的类型是 Directive.Argument。类 Directive.Argument 中提供了多个构造方法创建不同类型内容的参数。Directive.Argument(Integer)创建内容是立即数的参数, Directive.Argument(Label)创建内容是标签的引用的参数 (例如 .ent main), Directive.Argument(String)创建内容是字符串的参数 (例如 .ascii "Hello,world!")。

### 7.2.2.3 表达标签

表达标签

```
main:
```

可以使用如下代码

```
Label label = new Label("main");
```

其中,

- 使用类 Label 的对象表示一个标签, 这个类中有唯一的构造方法。
- 类 Label 的构造方法只有一个 String 类型的参数, 接受标签的名称。

### 7.2.2.4 表达没有操作数的指令

表达指令

```
syscall
```

可以使用如下代码

```
Instruct inst = new Instruct(MIPSOpcode.SYSCALL, null);
```

其中,

- 使用类 Instruct 的对象表示一个指令, 这个类中有唯一的构造方法。
- 类 Instruct 的构造方法拥有 2 个参数, 第一个参数指明指令的操作码, 第二个参数提供指令的操作数。
- 指令的操作码用 MIPSOpcode 中定义的 static final MIPSOpcode 对象表示, 这些对象的名称是操作码的大写形式。例如, .syscall 用 MIPSOpcode.SYSCALL 表示, .nop 用 MIPSOpcode.NOP 表示。
- 如果指令不需要操作数, 将 null 作为类 Instruct 的构造方法的第二个参数。

### 7.2.2.5 表达操作数包含标签的指令

表达指令

```
j.func #func is a label
```

可以只用如下代码

```
ArrayList<Instruct.Operand> operands = new ArrayList<Instruct.Operand>();
```

```
operands.add(new Instruct.Operand(new Label("func")));
Instruct inst = new Instruct(MIPSOpcode.J, operands);
```

其中,

- 类 `Instruct` 的构造方法的第二个参数的类型是 `List<Instruct.Operand>`。`List` 中依次存放指令需要的操作码。
- 每个参数的类型是 `Instruct.Operand`。类 `Instruct.Operand` 中提供了多个构造方法创建不同类型内容的操作数。`Instruct.Operand(Integer)` 创建内容是立即数的操作数, `Instruct.Operand(Label)` 创建内容是标签的引用的操作数, `Instruct.Operand(Register)` 创建内容是寄存器的操作数, `Instruct.Operand(Register, Integer)` 创建内容是寄存器+偏移的操作数 (例如 `12($fp)`), `Instruct.Operand(RegisterVariable)` 创建内容是寄存器变量的操作数, `Instruct.Operand(RegisterVariable, Integer)` 创建内容是寄存器变量+偏移的操作数 (例如 `12($R0)`, `R0` 是寄存器变量)。

### 7.2.2.6 表达操作数包含寄存器和立即数的指令

表达指令

```
li $ra, 1
```

可以使用如下代码

```
ArrayList<Instruct.Operand> operands = new ArrayList<Instruct.Operand>();
operands.add(new Instruct.Operand(MIPSRegister.RA)); # register
operands.add(new Instruct.Operand(new Integer(1))); # immediate digit
Instruct inst = new Instruct(MIPSOpcode.LI, operands);
```

其中,

- 内容是寄存器的操作数使用类 `Instruct.Operand` 中的构造方法 `Instruct.Operand(Register)` 创建。
- `Register` 是一个结构无关的接口, 结构相关的类 `MIPSRegister` 是这个接口的一个实现。`MIPSRegister` 中定义了一系列 `static final MIPSRegister` 对象来表示 MIPS CPU 中的通用寄存器。这些对象的名字是相应寄存器名称的大写形式。例如, `$31` 或者 `$ra` 用 `MIPSRegister.RA` 表示, `$0` 或者 `$zero` 用 `MIPSRegister.ZERO` 表示。
- 内容是立即数的操作数使用类 `Instruct.Operand` 中的构造方法 `Instruct.Operand(Integer)` 创建。

### 7.2.2.7 表达操作数包含寄存器变量的指令

表达指令

```
li $R0, 1
```

可以使用如下代码

```
ArrayList<Instruct.Operand> operands = new ArrayList<Instruct.Operand>();
operands.add(new Instruct.Operand(new RegisterVariable(0))); # register variable
operands.add(new Instruct.Operand(new Integer(1)));
```

```
Instruct inst = new Instruct(MIPSOpcode.LI, operands);
```

其中,

- 在生成真正的汇编代码之前, 可以使用寄存器变量代替寄存器的出现, 然后用我们提供的 MIPS 寄存器分配器 (仅限于 MIPS 汇编代码) 或者你自己的寄存器分配器, 将这些寄存器变量替换成真正寄存器或者是栈上的某个位置。
- 每个寄存器变量使用一个非负的整数作为自己的标号, 以和其它的寄存器变量区别。
- 寄存器变量使用类 RegisterVariable 的对象表示, 这个类拥有唯一的构造方法。

## 7.3 MIPS R2000/R3000 架构、汇编语言及 SPIM

### 7.3.1 MIPS R2000/R3000 架构概述

MIPS R2000/R3000 架构既支持大尾端 (big-endian) 字节序, 也支持小尾端 (little-endian) 字节序, 但是在一个系统上, 必须统一使用一种字节序, 否则会出现混乱。

一个运行在 MIPS R2000/R3000 上的程序的地址空间如图 7-1 所示。

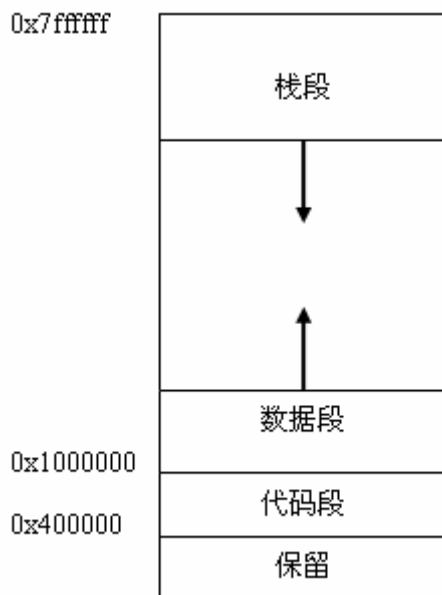


图 7-1 程序地址空间

0x0~0x400000 是保留空间 (reserved), 程序不可用。

0x400000~0x1000000 是代码段 (text segment), 存放程序的指令。

0x1000000 以上的地址空间分为两部分: 从低地址向高地址增长的数据段 (data segment) 和从高地址向低地址增长的栈段 (stack segment)。数据段中位于低地址的部分是静态数据 (static data)。静态数据包括在编译时大小已知的, 并且在程序的整个执行过程中都可见的对象。例如, C 语言中静态分配的全局变量。静态数据之上是动态数据 (dynamic data)。动态数据包括在程序执行期间动态分配的对象。例如, C 语言中动态分配的变量。

在硬件层次上, MIPS R2000/R3000 仅提供一种寻址方式: 基址+偏移。其中基址保存在

通用寄存器中，偏移为 16 位。虽然在硬件层次上，MIPS R2000/R3000 仅提供了“基址+偏移”的寻址方式，但是通过汇编器，你可以使用以下方式的寻址方式。在本章的课程设计中，你可能更多的使用的是前两种寻址方式。

- **基址+偏移**: 与硬件层次上的寻址方式相同。形式为“offset(\$register)”。例如: 10(\$3), 8(\$sp), 0(\$4)。
- **Direct**: 使用标签 (label) 或者外部变量名作为地址。例如: jal f 中的 f。
- **Direct+index**: 形式为 offset(label)。其中 offset 为 16 位。例如: 16(L1), L1 是汇编程序中出现的一个 label。
- **Constant**: 使用一个立即数作为地址。这个立即数会被做为一个 32 位绝对地址使用。
- **Register indirect**: 直接使用寄存器的值作为地址，即偏移为 0 的“基址+偏移”寻址方式。

### 7.3.2 汇编语言的语法

一段MIPS汇编程序大致如图 7-2 所示。以下将介绍其中的各个部分。

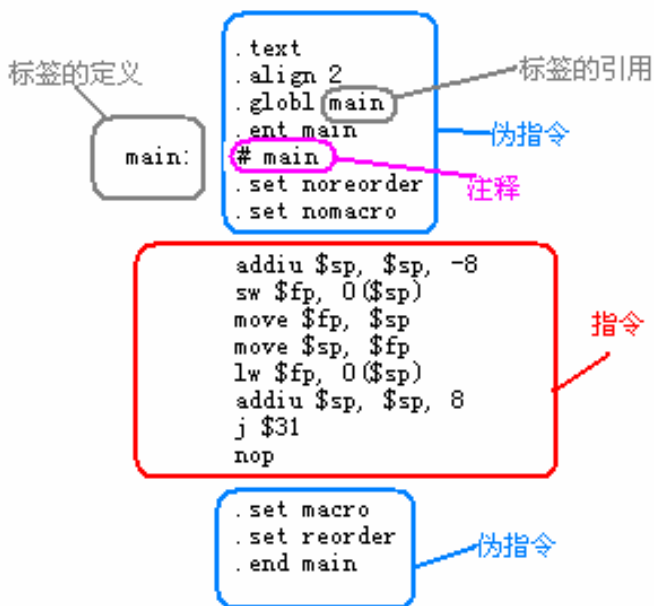


图 7-2 MIPS 汇编程序

#### 1、注释 (Comment)

注释以符号#开始，直到该行的末尾。例如，

```
.global main # Must be global
```

#### 2、标识符 (Identifier)

标识符是不以阿拉伯数字作为前缀的，由英文字母 (A-Za-z)、阿拉伯数字 (0-9)、下划线 (\_)、点 (.) 组成的字符串 (不是 4、所说的字符串，不用包括在引号”中)。汇编指令

的作业码 (opcode) 是保留字, 不能作为标识符。

### 3、数字 (Number)

默认情况下, 数字均表示十进制数。如果数字以 0x 为前缀, 则表示十六进制数。

### 4、字符串 (String)

字符串必须包括在一对双引号中。字符串中的一些特殊字符必须用转义字符来表示。如“\n”、“\t”等等。

### 5、标签 (Label)

标签用来表示汇编程序中的某个位置。例如表示某个过程的开始, 或者跳转指令的跳转目的地。标签的出现分为两种, 一种是标签的定义, 一种是标签的使用。标签的定义在程序中单独占一行或者在一个语句的开始, 由标签名称加一个: 组成, 表示程序中的这个位置。标签的引用即标签的名称, 表示引用标签表示的位置。

### 6、伪指令 (Directive)

表 7-1 中仅列出课程实验中可能会遇到的伪指令。表中[]不是伪指令的组成部分, 仅表示可选内容。

表 7-1 MIPS 汇编中的部分伪指令

伪指令	说明
.align n	使下一个数据按照 $2^n$ byte 对齐。如果 n 为 0, 则取消自动数据对齐, 直到下一个 .data 的出现。
.ascii str	将字符串 str 存储在内存中。字符串不会自动以 null 结尾。
.asciiz str	将字符串 str 存储在内存中。字符串自动以 null 结尾。
.byte b1,...,bn	将 n 个值存储在连续的字节上。
.data [addr]	该伪指令之后的条目都将存储在数据段上。如果给出参数 addr, 那么数据段从地址 addr 开始。
.global sym	声明一个全局符号。
.space n	在当前段中分配 n 字节空间。
.text [addr]	该伪指令之后的条目都将存储在代码段上。如果给出参数 addr, 那么代码段从地址 addr 开始。
.word w1,...,wn	将 n 个值存储在连续的字上。
.ent sym	表示名为 sym 的方法的开始
.end sym	表示名为 sym 的方法的结束

## 7.3.3 寄存器

相对于 CISC 体系结构的 CPU, RISC 体系结构的 CPU 的一个显著特点是: 除加载/存

储指令外，其它所有指令的操作对象都是寄存器或立即数。

表 7-2 R2000/R3000 中的通用寄存器

寄存器别名	标号	用途
zero	0	常数 0。
at	1	保留，供汇编器使用。
v0	2	存放子过程的返回值。
v1	3	
a0	4	存放子过程的头几个参数。
a1	5	
a2	6	
a3	7	
t0	8	子过程可以自由使用而不需事先备份旧值的寄存器。旧值的备份由调用者完成。
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	子过程使用这些寄存器之前需要备份它们的旧值；在返回之前必须恢复它们的旧值。
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	与 t0~t7 相同
t9	25	
k0	26	保留，供操作系统处理中断和陷阱
k1	27	
gp	28	全局指针，指向静态数据区的低 64K 内存块（可以用于存放全局变量和常数）的中间。通过 gp，可以快速访问这个内存块中的数据。
sp	29	栈指针，指向栈顶。
s8 或 fp	30	帧指针。
ra	31	子过程的返回地址。

MIPS R2000/R3000 内建 32 个 32 位通用寄存器，标号 0~31。第*i*个（*i*从 0 开始）寄存器即可以用\$*i*表示，也可以用它们的别名表示。各个寄存器的别名、标号、用途如表 7-2 所



示。

### 7.3.4 汇编指令

以下仅介绍课程设计中可能会用到的汇编指令。

#### 1、空指令

空指令不做任何事情，仅仅占用一个指令周期。

- **nop**: 空指令。实际上，在 MIPS 中任何目的寄存器为 zero 的指令都可以作为空指令使用。

#### 2、寄存器移动

- **move d, s**: 将寄存器 s 中的数据赋给寄存器 d。

#### 3、常数加载

- **la d, addr**: 加载地址 addr 到寄存器 d。
- **li d, j**: 加载立即数到寄存器 d，其中  $-32768 \leq j \leq 65535$ 。

#### 4、整数加法

以下 2 条指令在加法结果溢出时会产生异常 (exception)。

- **add d, s, t**: 将寄存器 s 和 t 中的值相加，结果存入寄存器 d ( $d=s+t$ )。
- **add d, s, j**: 将寄存器 s 的值和有符号立即数 j 相加，结果存入寄存器 d ( $d=s+(\text{signed})j$ )。

以下 2 条指令在加法结果溢出时不会产生异常。

- **addu d, s, t**: 将寄存器 s 和 t 中的值相加，结果存入寄存器 d ( $d=s+t$ )。
- **addiu d, s, j**: 将寄存器 s 的值和有符号立即数 j 相加，结果存入寄存器 d ( $d=s+(\text{signed})j$ )。

#### 5、整数减法

以下 1 条指令在减法溢出时会产生异常。

- **sub d, s, t**: 将寄存器 s 和 t 中的值相减，结果存入寄存器 d ( $d=s-t$ )。

以下 1 条指令在减法溢出时不会产生异常。

- **subu d, s, t**: 将寄存器 s 和 t 中的值相减，结果存入寄存器 d ( $d=s-t$ )。

#### 6、数学函数

以下 1 条指令实现求整数绝对值的功能。

- **abs d, s**: 将寄存器 s 中的值得绝对值存入寄存器 d ( $d=|s|$ )。

以下 2 条指令实现求整数的相反数的功能。

- **neg d, s**: 将寄存器 s 中的值得相反数存入寄存器 d ( $d=-s$ )。如果结果溢出，产生异常。

- **negu d, s:** 将寄存器 s 中的值得相反数存入寄存器 d ( $d=-s$ )。如果结果溢出, 不会产生异常。

## 7、位逻辑运算

以下 8 条指令分别实现整数的按位与、按位或、按位异或、按位非或、按位取反运算。

- **and d, s, t:** 将寄存器 s 和 t 中的值按位与, 结果存入寄存器 d ( $d=s\&t$ )。
- **andi d, s, j:** 将寄存器 s 中的值和无符号整数 j 按位与, 结果存入寄存器 d ( $d=s\&(\text{unsigned})j$ )。
- **or d, s, t:** 将寄存器 s 和 t 中的值按位或, 结果存入寄存器 d ( $d=s|t$ )。
- **ori d, s, j:** 将寄存器 s 中的值和无符号整数 j 按位或, 结果存入寄存器 d ( $d=s|(\text{unsigned})j$ )。
- **xor d, s, t:** 将寄存器 s 和 t 中的值按位异或, 结果存入寄存器 d ( $d=s\wedge t$ )。
- **xori d, s, j:** 将寄存器 s 中的值和无符号整数 j 按位异或, 结果存入寄存器 d ( $d=s\wedge(\text{unsigned})j$ )。
- **nor d, s, t:** 将寄存器 s 和 t 中的值按位非或, 结果存入寄存器 d ( $d=\sim(s|t)$ )。
- **not d, s:** 将寄存器 s 中的值按位取反, 结果存入寄存器 d ( $d=\sim s$ )。

## 8、位移运算

以下 2 条指令分别实现整数的循环左移和循环右移运算。

- **rol d, s, t:** 将寄存器 s 中的值循环左移 t (寄存器 t 中的值) 位, 结果存入寄存器 d ( $d=(s\ll t)|((\text{unsigned})s\gg(32-t))$ )。
- **rord, s, t:** 将寄存器 s 中的值循环右移 t (寄存器 t 中的值) 位, 结果存入寄存器 d ( $d=((\text{unsigned})s\gg t)|(s\ll(32-t))$ )。

以下 2 条指令实现整数的算术左移运算。

- **sll d, s, t:** 将寄存器 s 中的值算术左移 t (寄存器 t 中的值) 位, 结果存入寄存器 d ( $d=s\ll(t\%32)$ )。
- **sll d, s, j:** 将寄存器 s 中的值算术左移 j 位 (j 是立即数,  $0\leq j<31$ ), 结果存入寄存器 d ( $d=s\ll j$ )。

以下 2 条指令实现整数的算术右移运算。

- **sra d, s, t:** 将寄存器 s 中的值算术右移 t (寄存器 t 中的值) 位, 结果存入寄存器 d ( $d=(\text{signed})s\gg t$ )。
- **sra d, s, j:** 将寄存器 s 中的值算术右移 j 位 (j 是立即数), 结果存入寄存器 d ( $d=(\text{signed})s\gg j$ )。

以下 2 条指令实现整数的逻辑右移运算。

- **srl d, s, t:** 将寄存器 s 中的值算术右移 t (寄存器 t 中的值) 位, 结果存入寄存器 d ( $d=(\text{unsigned})s\gg t$ )。
- **srl d, s, j:** 将寄存器 s 中的值算术右移 j 位 (j 是立即数), 结果存入寄存器 d ( $d=(\text{unsigned})s\gg j$ )。

### 9、Set if...

Set if...指令都是 3 操作数指令，比较后 2 个操作数（倒数第 1 个操作数可以是寄存器，也可以是 16bit 立即数；倒数第 2 个操作数是寄存器），当它们满足一定的逻辑关系时，将第一个操作数置为 1，否则置为 0。以 u 为后缀的 Set if...指令将后 2 个操作数作为无符号整数处理，其它的 Set if...指令将后 2 个操作数作为有符号整数处理。

以下仅列出各个指令用于比较后 2 个操作数的逻辑关系（省略以 u 为后缀的指令，它们的逻辑关系与不带后缀的指令相同）。

- **slt**: 小于关系。
- **seq**: 等于关系。
- **sge**: 大于等于关系。
- **sgt**: 大于关系。
- **sle**: 小于等于关系。
- **slt**: 小于关系。
- **sne**: 不等于关系。

### 10、整数乘、除、求余运算

MIPS R2000/R3000 中的乘、除、求余运算有单独的单元进行，比一般的整数运算耗费更多的时间，而且不会检查结果溢出和除数为 0，也不会将结果放入通用寄存器中。运算结果的高 32 位（对于除法运算是余数）和低 32 位（对于除法运算是整数商）分别存放在内部寄存器 hi 和 lo 中，可以通过指令 mfhi 和 mflo 获得。

但是 MIPS 汇编语言中提供乘、除、求余指令可以检查溢出和除数为 0，以及将结果存入通用寄存器。

以下 2 条指令实现 32 位整数的乘法运算，结果保留低 32 位。如果发生溢出，会产生异常。

- **mulo d, s, t**: 有符号整数乘法。将寄存器 s 和 t 中的值相乘，结果存入寄存器 d ( $d=(\text{signed})s*(\text{signed})t$ )。
- **mulou d, s, t**: 无符号整数乘法。将寄存器 s 和 t 中的值相乘，结果存入寄存器 d ( $d=(\text{unsigned})s*(\text{unsigned})t$ )。

以下 2 条指令实现 32 位整数的除法运算。如果发生溢出或除数为 0，会产生异常。

- **div d, s, t**: 有符号整数除法。将寄存器 s 和 t 中的值相除，整数商存入寄存器 d ( $d=(\text{signed})s/(\text{signed})t$ )。
- **divu d, s, t**: 无符号整数除法。将寄存器 s 和 t 中的值相除，整数商存入寄存器 d ( $d=(\text{unsigned})s/(\text{unsigned})t$ )。

以下 2 条指令实现 32 位整数的求余运算。如果发生溢出或除数为 0，会产生异常。

- **rem d, s, t**: 有符号整数求余。将寄存器 s 和 t 中的值相除，余数存入寄存器 d ( $d=(\text{signed})s\%(\text{signed})t$ )。
- **remu d, s, t**: 无符号整数求余。将寄存器 s 和 t 中的值相除，余数存入寄存器 d

( $d = (\text{unsigned})s \% (\text{unsigned})t$ )。

## 11、数据加载/存储

数据加载/存储指令是 MIPS 汇编指令中唯一以内存地址作为操作数的一类指令。

数据加载/存储指令支持多种数据长度：8bit，16bit，32bit，64bit。

在硬件层次上，数据加载指令要求地址对齐。即 8bit 数据出现在内存地址（以 bit 为单位）中 8 的倍数的位置，16bit 数据出现在内存地址中 16 的倍数的位置，32bit 数据出现在内存地址中 32 的倍数的位置，64bit 数据出现在内存地址中 64 的倍数的位置。

对于长度小于寄存器位数（32bit）的数据，数据加载指令会以两种方式对数据进行扩展：

- **补 0**：将数据的高位用 0 填充，这类指令以 **u** 为后缀；
- **按符号扩展**：将数据的高位用数据的符号位填充，这类指令不以 **u** 为后缀。

如前所述，所有的数据加载指令的结果至少一个指令周期后才能得到。因此，数据加载指令后的一条指令必须与载入的数据无关（例如 **nop** 指令）。

以下 11 条指令分别实现各种数据长度的数据加载和存储。

- **lb d, addr**：从地址 **addr** 加载 8bit 数据到寄存器 **d**。按符号扩展。
- **lbu d, addr**：从地址 **addr** 加载 8bit 数据到寄存器 **d**。补 0 扩展。
- **lh d, addr**：从地址 **addr** 加载 16bit 数据到寄存器 **d**。按符号扩展。
- **lhu d, addr**：从地址 **addr** 加载 16bit 数据到寄存器 **d**。补 0 扩展。
- **lw d, addr**：从地址 **addr** 加载 32bit 数据到寄存器 **d**。按符号扩展。
- **lwu d, addr**：从地址 **addr** 加载 32bit 数据到寄存器 **d**。补 0 扩展。
- **ld d, addr**：从地址 **addr** 加载 64bit 数据到寄存器 **d**。按符号扩展。
- **ldu d, addr**：从地址 **addr** 加载 64bit 数据到寄存器 **d**。补 0 扩展。
- **sb t, addr**：将寄存器 **t** 中的低 8bit 数据存储到地址 **addr**。
- **sh t, addr**：将寄存器 **t** 中的低 16bit 数据存储到地址 **addr**。
- **sw t, addr**：将寄存器 **t** 中的 32bit 数据存储到地址 **addr**。

## 12、跳转/分支

以下指令中以 **b** 为前缀的为分支指令，以 **j** 为前缀的为跳转指令。分支指令使用 16bit 的偏移，因此最多可以跳转  $2^{15}-1$  条指令。跳转指令使用 26bit 的地址空间。

如前所述，跳转/分支指令存在分支延迟的现象。可以通过在跳转/分支指令后添加 1 条 **nop** 指令避免过多的关注分支延迟。

以下 4 条指令是跳转指令。

- **j label**：无条件跳转到标签 **label** 处。
- **jr r**：无条件跳转到寄存器 **r** 中值表示的地址处。
- **jal label**：无条件跳转到标签 **label** 处，并将跳转指令的下一条指令的地址存入寄存器 **\$ra**。
- **jalr r**：无条件跳转到寄存器 **r** 中值表示的地址处，并将跳转指令的下一条指令的

地址存入寄存器 \$ra。

以下指令是分支指令。其中 `src` 既可以是寄存器，也可以是立即数。

- **b label:** 无条件分支到标签 `label` 处。
- **beq/bge/bgt/ble/blt/bne s, src, label:** 比较寄存器 `s` 中的值和 `src`，如果某种关系成立，就分支到标签 `label` 处。对于这些指令，比较的关系分别为等于、大于等于、大于、小于等于、小于、不等于。

### 7.3.5 过程调用

在 MIPS 汇编程序中，可以有多种方式调用一个过程和从被调过程中返回。其中一种简单的方法是，直接用跳转指令跳转到被调过程的方式调用过程，以及直接从被调过程中跳转到调用点的下一条指令的方式返回。例如下面的这段代码中，第 3 行的代码通过跳转指令调用过程 `f`，第 8 行的代码提供跳转指令返回调用点的下一条指令。

```

1  main:
2      ..... # codes before procedure call
3      j f   # call procedure f
4  L1:
5      ..... # codes after procedure call
6  f:
7      ..... # codes before return
8      j L1  # return

```

无论使用哪种方法，汇编程序员都需要小心的处理被调用者和调用者之间的寄存器冲突、参数传递、返回值传递等问题。以下将介绍 MIPS O32 Compiling and Performance Tuning Guide（以下简称 `o32`）中提到的一种现在广泛使用的过程调用和过程返回的方法。

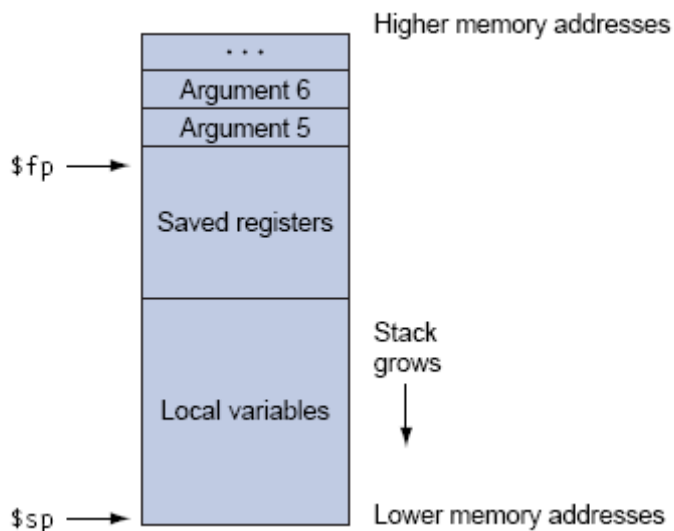


图 7-3 过程的帧栈结构

一个正在执行的过程的内存空间是如图 7-3 所示的帧（`frame`）。帧指针 `fp` 和栈指针 `sp` 之间的部分就是一个帧。帧指针 `fp` 指向当前过程的最后一个参数之后的位置。栈指针 `sp` 指向当

前过程的栈顶。o32 中要求栈指针sp按 8 字节对齐。

当前过程调用另一个过程时，必须按照以下步骤：

- 1) 传递参数。将被调用过程的参数倒序存放在栈上，即第一个参数存放在内存的低地址，最后一个参数存放在内存高地址。另外，即使被调用过程的参数少于 4 个，你也要在栈上分配至少存放 4 个参数的空间(16bytes)。被调用过程可以使用 $(n*4)(\$fp)$ 的形式访问第  $n$  个参数。实际上，前 4 个参数是通过寄存器\$ $a0 \sim a3$ 传递的，但你仍然需要在栈上为这些参数保留空间（即前面所说的 16bytes）。被调用过程使用寄存器访问前 4 个参数，而不是通过栈。
- 2) 保存寄存器。如果在被调用过程的调用点寄存器\$ $t0 \sim t9$ 中有活跃数据，那么当前过程必须将它们的值保存在栈上。保存位置的地址必须比参数位置的地址高。
- 3) 执行 jal 指令。跳转到被调用过程。

被调用过程在执行初始，必须按照以下步骤：

- 1) 为被调用过程的分配栈帧空间，即将栈指针\$ $sp$ 减去帧的大小作为栈指针\$ $sp$ 的值。帧的大小需要根据被调用过程中定义的局部变量的数目和类型、调用的其它过程的数目等决定。
- 2) 将需要被调用过程保存的寄存器存放在栈上。帧指针\$ $fp$ 必须保存在栈上。如果调用其它过程，那么寄存器\$ $ra$ 也必须保存在栈上。
- 3) 获取帧指针\$ $fp$ 。即栈指针\$ $sp$ 加上帧的大小作为帧指针\$ $fp$ 的值。

当被调用过程执行完毕，需要返回时，必须按照以下步骤：

- 1) 恢复需要被调用过程保存的寄存器在过程执行初始的值。
- 2) 释放栈帧空间，即将栈指针\$ $sp$ 加上帧的大小作为栈指针\$ $sp$ 的值。
- 3) 返回调用者，即跳转到寄存器\$ $ra$ 所保存的地址。

## 7.4 MIPS 汇编语言与 SkipOOMiniJOOl 语言中部分结构的对应

对于 SkipOOMiniJOOl 语言中的许多结构，都有多种形式的汇编代码与之对应。下面介绍的是汇编语言结构相对固定的几种语法结构。

### 7.4.1 程序

与一个SkipOOMiniJOOl语言程序对应的汇编程序至少由 2 部分组成（如图 7-4 所示）：数据段和代码段。

```
.data
# definitions of global objects

.text
#definitions of methods
```

图 7-4 总体程序结构

数据段以.data开始，代码段以.text开始。这里的数据段和代码段不是图 7-4 中提到的数

据段和代码段，仅表示汇编程序的结构。数据段中定义各种全局变量，代码段中定义程序中出现的所有的的方法。

### 7.4.2 全局变量的定义和引用

全局变量定义在数据段中。

SkipOOMiniJOOOL语言中整型全局变量的定义形式如图 7-5 所示。其中, `global_var_name` 是全局变量的名称, `var_value` 是全局变量的初始值。

```
.align 2
global_var_name:
.word var_value
```

图 7-5 整型全局变量的定义

图 7-6 展示了定义字符串型全局变量的汇编代码。其中, `str` 是字符串变量的名称, `string` 是字符串的内容。

```
.align 2
str:
.asciiz "string"
```

图 7-6 字符串型全局变量的定义

使用全局变量时直接使用它的名称即可。

### 7.4.3 方法的定义

方法定义在代码段中。方法定义中需要注意的事项可以参考 7.3.5。

SkipOOMiniJOOOL语言中方法的定义显示如图 7-7 所示。其中,

- `method_name` 是方法的名称。
- 方法的返回地址存放在寄存器\$31 中, 不需程序员显式赋值, 只要在方法结束时跳转到\$31 指定的地址。
- 方法的返回值约定存储在寄存器\$2 和\$3 中。

```
.align 2
.global method_name
.ent method_name
method_name:
    # codes of the method body
j $31    # return
.end main
```

图 7-7 方法的定义

### 7.4.4 方法的调用

方法调用需要注意的事项可以参考 7.3.5。图 7-8 展示调用一个只有一个整型参数的方法 `f` 的汇编代码。注意，方法 `f` 需要的参数并未放在栈上为其预留的空间中，而是使用寄存器 `$4`。一些约定俗成的规定是：方法的前 4 个参数（如果寄存器长度允许）通过寄存器 `$4~$7` 传递，但是栈上仍然要预留它们的空间（即使什么都不存）；更多的参数通过栈传递。图 7-9 展示调用拥有 5 个整型参数的方法 `g` 的汇编代码，其中第 5 个参数通过栈传递。

Main:

```

addiu $sp, $sp, -32  # allocate stack space
                        # NOTE: at least 16 bytes for arguments of f
sw $31, 28($sp)      # backup return address
sw $fp, 24($sp)       # backup frame pointer
move $fp, $sp

lw $4, addr          # the value of argument is in addr
jal f                 # call method f
nop                  # for delayed branches

move $sp, $fp
lw $31, 28($sp)       # restore return address
lw $fp, 24($sp)       # restore frame pointer
addiu $sp, $sp, -32   # free stack space
j $31
nop
.end main

```

图 7-8 方法调用 1

```

# ignore the allocation of stack space and the backup of registers,
# which are somehow like previous example
sw $2, 16($sp)        # pass 5th argument via stack
                        # suppose the value of 5th argument is in $2

lw $4, addr1          # 1st argument
lw $5, addr2          # 2nd argument
lw $6, addr3          # 3rd argument
lw $7, addr4          # 4th argument
jal g                 # call method g
nop

# ignore the free of stack space and restore of registers,
# which are somehow like previous example

```

图 7-9 方法调用 2



### 7.4.5 if 语句

图 7-10 展示了if-then-else语句对应的汇编代码。

```
# calculate condition expression and suppose store the bool result in $2
beq $2, $0, $L2
nop
$L1:
# codes of then branch
j $L3
nop
$L2:
# codes of else branch
$L3:
# codes of statements after if-then-else
```

图 7-10 if-then-else 语句

图 7-11 展示了if-then语句对应的汇编代码。

```
#calculate condition expression and suppose store the bool result in $2
beq $2, $0, $L1
nop
# codes of then branch
$L1:
# codes of statements after if-then
```

图 7-11 if-then 语句

### 7.4.6 while 语句

图 7-12 展示了while语句对应的汇编代码。

```
j $L1
nop
$L2:
# codes of loop body
$L1:
# calculate condition expression and suppose store the bool result in $2
bne $2, $0, $L2
nop
```

图 7-12 while 语句

### 7.4.7 return 语句

图 7-13 展示return语句对应的汇编代码。MIPS汇编中通过跳转到\$31 中存放的当前方法返回地址从一个方法返回它的调用者。返回值通过寄存器\$2~\$3 返回。

```
lw $2, addr
# codes to free stack space and restore callee-saved registers
j $31
nop
```

图 7-13 return 语句

## 7.5 SPIM

spim是SPIM的字符界面。spim运行后会提供一个类似shell的界面，根据用户输入的命令执行相应的功能。常用的命令如 表 7-3 所示。

表 7-3 spim 常用命令

命令	功能
?	获取 spim 的帮助。
load “src_file”	载入汇编语言文件 src_file，注意引号”不可省略。在载入的同时，spim 会检查文件的语法，并打印出相应的错误（如果有的话）。
run	执行载入的汇编语言文件。
step N	执行 N 条汇编指令。如果省略 N，则单步执行。
continue	继续执行程序。
print \$N	打印寄存器 N 的值。
print addr	打印地址中的值。
print_sym	打印符号表。
reinitialize	初始化内存和寄存器。在每次载入文件和重新执行程序前，需要执行这条命令。
breakpoint addr	在地址 addr 处插入一个断点。addr 可以地址也可以是标签（label）。
delete addr	删除地址 addr 处的断点。
list	列出所有的断点。

### 7.5.1 XSpim 和 PCSpim

XSpim和PCSpim的界面相似，以下以PCSpim为准介绍。PCSpim的界面如 图 7-14 所示。

PCSpim 界面从上至下被划分为四部分：

- 最上部的部分显示所有寄存器的值。这些值将在每次程序停止运行时更新。
- 第 2 个部分显示汇编程序代码。这些代码不仅包含载入的文件中的汇编代码，还包括 SPIM 添加进的汇编代码。每行代码均以以下形式显示

[指令地址] 指令编码 助记符形式的汇编代码；行号: 实际的汇编代码。

- 第 3 个部分显示程序运行过程中内存和栈的内容。
- 最下面的部分是 PCSpim 显示的信息。出错信息也在这里显示。

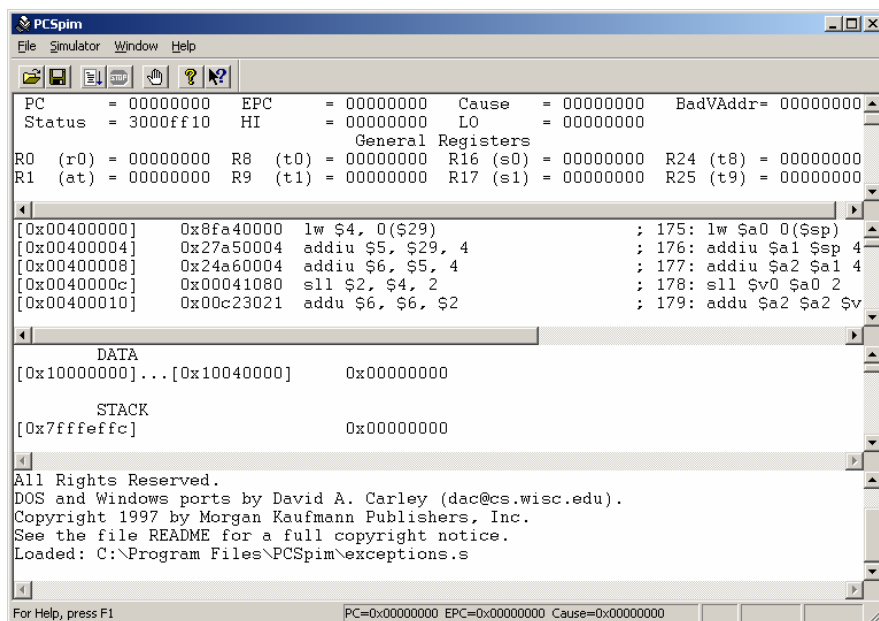


图 7-14 PCSpim 界面

汇编程序的载入、运行、单步执行、断点等功能都可以通过 PCSpim 相应的菜单项实现。

## 7.6 SPIM 提供的系统调用

SPIM 自身提供了一些系统调用，可以在汇编程序中通过 `syscall` 指令调用。要使用某个系统调用，需要先将系统调用号存入寄存器 `$v0`，将系统调用需要的参数存入寄存器 `$a0~$a3`。系统调用会将返回值存入寄存器 `$v0`。课程设计中会使用到的系统调用的调用号和参数如表 7-4 所示。

表 7-4 SPIM 提供的系统调用

系统调用	调用号	参数
打印整型数	1	\$a0 存储要打印的整型数
打印字符串	4	\$a0 存储要打印的字符串的地址

图 7-15 展示了利用系统调用打印字符串。

```
# suppose the string is stored in variable str
li $v0, 4    # load system call code
la $a0, str  # load the address of string
syscall      # print string
```

图 7-15 SPIM 系统调用

## 7.7 全局寄存器分配器

### 7.7.1 与汇编代码生成器的关系

我们在edu.ustc.cs.compile.arch.jar类库中提供了MIPS全局寄存器分配器的实现。这个全局寄存器分配器与MIPS汇编代码生成器之间的关系如图 7-16 所示。MIPS汇编代码生成器接收一个SimpleMiniJOOl或者SkipOOMiniJOOl程序的中间表示，然后对其分析得到汇编代码的内部表示；每得到一个函数的汇编代码的内部表示（即一个AssemblySequence对象，参见 7.2 节）就将其传给寄存器分配器；寄存器分配器会返回这个函数的指令序列中涉及的寄存器的分配结果；MIPS汇编代码生成器接收到分配结果后，还需要对修改改函数的汇编代码，以落实其中的寄存器分配结果。

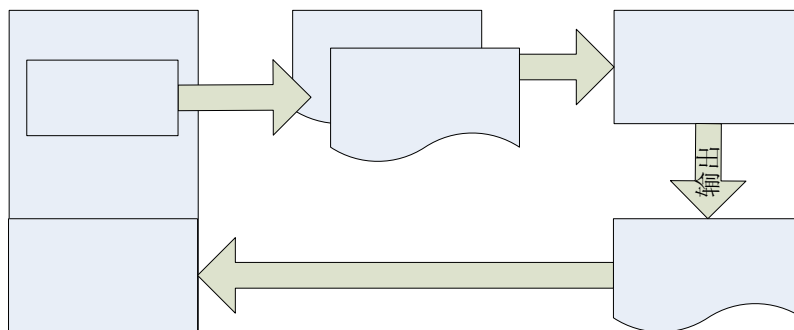


图 7-16 寄存器分配器和汇编代码生成器的关系

### 7.7.2 使用方法

我们提供的 MIPS 寄存器分配器包含在包 edu.ustc.cs.compile.arch.mips.regalloc 中。这个包中含有类 RegAllocator 和 AllocateResult 等。

你需要调用类 RegAllocator 中的公共静态方法

```
public static ArrayList<AllocateResult>
    globalRegAlloc(AssemblySequence stmts, int regVarNum)
```

为汇编代码中的每个函数分配寄存器。方法 globalRegAlloc() 的输入是汇编代码中一个函数的汇编代码序列 stmts（类型为 AssemblySequence，参见 7.2 节），这个汇编代码假设 CPU 上有无数的寄存器可以使用，这样在汇编代码生成时就无需考虑过程内部的寄存器冲突问题。这种可以任意使用的寄存器在汇编代码表示中用寄存器变量（Register Variable，参见 7.2 节）来表示。你需要将汇编代码中需要寄存器分配的位置都用寄存器变量代替。每个寄存器变量拥有一个寄存器变量号。globalRegAlloc() 会为这些寄存器变量分配 t0 到 t9 及 s0 到 s7 中的一个，或者将寄存器变量溢出到栈上的某个位置。

寄存器分配的结果会以元素类型为 AllocateResult 的列表的形式返回。列表中的第 i 个元素（i 从 0 开始）表示标号为 i 的寄存器变量的存储位置。类 AllocateResult 存储每个寄存器变量的存储位置。这个存储位置或者是一个实际的寄存器，或者是栈上的某个位置（相对

汇编代码生成器  
生成汇编码  
修改汇编码，  
落实寄存器分  
配的结果

于栈底而言，并且假设栈底地址为 0)。在类 `AllocateResult` 中提供了获取这些存储位置的公共方法。

- `boolean isRegister()`  
如果分配到一个寄存器，返回 `true`；否则返回 `false`。
- `MIPSRegister register()`  
如果分配到一个寄存器，返回这个寄存器；否则，返回 `null`。
- `boolean isSpilt()`  
如果溢出到栈上的某个位置，返回 `true`；否则，返回 `false`。
- `Integer offset()`  
如果溢出到栈上，返回相对于栈底（地址为 0）的偏移；否则，返回 `null`。

## 7.8 课程设计 5-1

本课程设计中，你需要实现一个由 `SkipOOMiniJOOOL` 语言程序的中间表示生成对应的 MIPS 汇编代码的 MIPS 汇编代码生成器。在汇编代码生成的过程中，你将使用我们提供的全局寄存器分配器进行过程内寄存器分配。但是，你仍需要自己处理过程间的寄存器分配问题。

课程设计 5-1 关联的 Java 源文件（相对于路径 `src/edu/ustc/cs/compile/generator/mips`）有：

- **Generator1.java**: 一个汇编代码生成器的框架代码，其中类 `Generator1` 是汇编代码生成器接口 `GeneratorInterface` 的一个实现；

关联的其他文件有：

- **bin 目录**
  - **lab5-1.xml**: 课程设计 5-1 的 ant 编译文件。
- **config 目录**
  - **lab5-1.xml**: 用实验平台运行课程设计 5-1 的配置文件。

这个课程设计中还将使用 `ROOT_DIR/lib` 中如下类包：

- **edu.ustc.cs.compile.parser.skipoominijool.jar**: 课程设计 5-1 中默认使用我们提供的 `SkipOOMiniJOOOL` 的语法生成器获得 `SkipOOMiniJOOOL` 程序的中间表示；
- **edu.ustc.cs.compile.arch.jar**: 提供汇编代码的内部表示及寄存器分配。

### 7.8.1 课程设计指导

汇编代码生成器以 `SkipOOMiniJOOOL` 程序的高级中间表示（使用 Eclipse AST 表示）为输入。汇编代码生成器需要利用访问者类 `Generator1Visitor` 遍历 AST、收集数据，根据 AST 节点的类型生成相应的 MIPS 汇编代码指令。这里，你需要利用 7.2 节中介绍的汇编代码的表示方法存储生成的这些汇编代码。`Generator1Visitor` 中定义了两个 `AssemblySequence` 类型的成员变量 `data` 和 `text` 分别用于存放数据段和代码段的汇编代码。你可以把遍历 AST 过程中生成的汇编代码的表示添加到 `data` 和 `text` 中。当所有代码生成完毕后，在通过 `Generator1Visitor` 中的

方法 `getASM()` 将 `data` 和 `text` 综合起来。

在这个课程设计中，你无需过多地考虑过程间的寄存器分配。你可以使用我们提供的寄存器分配器进行过程内寄存器分配。使用寄存器分配器是需要注意以下几点：

1. 过程中的寄存器变量的编号必须从 0 开始，并且是连续的。
2. 传递给寄存器分配器的汇编代码是从以 “`func_name:`” 为开始，以 “`.end func_name`” 为结束的汇编代码片段，不包括之前的 “`.ent func_name`”、“`.globl func_name`” 等内容。
3. 寄存器分配器返回的是寄存器变量的存储位置，你需要自己将生成的汇编代码中的这些寄存器变量落实到它们的存储位置上。
4. 我们提供的是过程内寄存器分配器，你需要自己解决过程间的寄存器分配问题。

## 7.8.2 汇编代码生成器的编译和运行

你可以在 `bin` 目录下执行命令：

```
ant lab5-1
```

编译本课程实验，获得一个 MIPS 汇编代码生成器。

然后，你可以在 `bin` 目录下执行命令启动实验平台运行 MIPS 汇编代码生成器：

```
run -cf ../config/lab5-1.xml
```

或 `./run.sh -cf ../config/lab5-1.xml`

## 7.9 课程设计 5-2

本课程设计的内容与课程设计 5-1 基本一致。但是你要完全自己解决寄存器分配的问题。

课程设计 5-2 关联的 Java 源文件（相对于路径 `src/edu/ustc/cs/compile/generator/mips.`）有：

- **Generator2.java:** 一个汇编代码生成器的框架代码，其中类 `Generator2` 是汇编代码生成器接口 `GeneratorInterface` 的一个实现；

关联的其他文件有：

- **bin 目录**
  - **lab5-2.xml:** 课程设计 5-2 的 ant 编译文件。
- **config 目录**
  - **lab5-2.xml:** 用实验平台运行课程设计 5-2 的配置文件。

这个课程设计中还将使用 `ROOT_DIR/lib` 中如下类包：

- **edu.ustc.cs.compile.parser.skipoominijool.jar:** 课程设计 5-2 中默认使用我们提供的 `SkipOOMiniJOOL` 的语法生成器获得 `SkipOOMiniJOOL` 程序的中间表示；
- **edu.ustc.cs.compile.arch.jar:** 提供汇编代码的内部表示及寄存器分配。

### 7.9.1 课程设计指导

本课程设计的大部分内容与课程设计 5-1 类似。但是你需要自己设计寄存器分配器，以下列出一些关于寄存器分配算法和实现的资料：

1. 陈意云，张昱，编译原理，高等教育出版社，2003。该书的 8.4.3 节介绍了一种非常简单和易于实现的局部寄存器分配算法。
2. Alfred V.AHO, Ravi Sethi, Jeffrey D.Ullman, *Compilers: Principles, Techniques, and Tools*.Addison-Wesley, Reading, MA., 1986。该书的 9.7 节介绍了一种易于实现的基于使用计数的寄存器分配算法。
3. G.J.Chaitin, Register allocation via coloring, *Computer Languages* 6, 1981。这篇文章介绍了经典的图着色寄存器分配算法。图着色算法及其各种改进算法是许多现代编译器中使用的寄存器分配算法。但是这种算法实现起来比较复杂。
4. M Poletto, V Sarkar , Linear scan register allocation , *ACM Transactions on Programming Languages and Systems*, 1999.10。这篇文章中介绍了一种不基于图分配的线性时间复杂度的寄存器分配算法。虽然这种算法依赖于 live variable 分析，但是仍然要比基于图着色的寄存器分配算法易于实现，而且在大多数实际情况下拥有不比图着色算法差的性能。

### 7.9.2 汇编代码生成器的编译和运行

你可以在 bin 目录下执行命令：

```
ant lab5-2
```

编译本课程实验，获得一个 MIPS 汇编代码生成器。

然后，你可以在 bin 目录下执行命令启动实验平台运行 MIPS 汇编代码生成器：

```
run -cf ../config/lab5-2.xml
```

或 `./run.sh -cf ../config/lab5-2.xml`

## 第8章 x86 汇编代码生成

在这一章，你将为 SkipMiniJOOOL 语言完成 x86 架构下的汇编代码生成。通过本章的课程设计，你将了解到 x86 架构及汇编代码与 MIPS 架构的不同之处，以及这些不同给设计实现汇编代码生成器带来的影响，等等。

### 8.1 本章课程设计概述

在本章的课程设计中，我们将循序渐进地进行如下两个课程设计：

**课程设计 6-1 利用现有的寄存器分配器生成 x86 汇编代码。**

在这个课程设计中，你将使用我们提供的 x86 局部寄存器分配器来设计实现 MIPS 汇编代码生成器。为此，你需要理解我们提供的 x86 寄存器分配器的接口和特点，你还需要保证你的汇编代码的内部表示采用我们所提供的形式。

**课程设计 6-2 独立实现完整的 x86 汇编代码生成器。**

在这个课程设计中，你可以自行设计和实现汇编代码生成器中遇到的各种问题。

要完成本章所述的课程设计，除了需要使用 Eclipse AST、实验平台库外，还需要使用存放在 ROOT\_DIR/lib 下的 edu.ustc.cs.compile.arch.jar。

### 8.2 x86 架构、汇编语言及工具

本节将对 x86 汇编语言进行一些简要介绍，以便帮助你快速地了解 x86 汇编语言。你可以利用 gcc 产生或编译 x86 汇编代码，来帮助你了解要生成的 x86 汇编代码的格式，它还可以帮助你检测所生成的汇编代码的正确性。

需要指出的是，GCC 是按照自己的格式来产生汇编代码的，这种格式称为 GAS(GNU ASsembler)。GAS 与 Intel 文档中的格式以及 Microsoft 编译器使用的格式差异很大。一个主要的区别就是源和目的操作数是以相反的顺序给出的。本节将基于 GAS 格式<sup>[7]</sup>来介绍。

#### 8.2.1 数据类型与指令

由于 x86 是从 16 位体系结构扩展成 32 位的，Intel 用术语“字(word)”表示 16 位数据类型。故称 32 位数为“双字(double words)”，称 64 位数为“四字(quad words)”。

GAS 中的每条指令都有一个字符后缀，表明操作数的大小。例如，mov（传送数据）指令有三种形式：movb（传送字节）、movw（传送字）和 movl（传送双字）。对于浮点数来说，GAS 使用后缀“s”表示 4 字节的单精度浮点数，用“l”表示 8 字节的双精度浮点数，用“t”表示扩展精度的浮点数。



## 8.2.2 寄存器

### 8.2.2.1 整数寄存器

一个 IA32 中央处理单元(CPU)包含一组八个存储 32 位值的寄存器(register), 即%eax、%ecx、%edx、%ebx、%esi、%edi、%ebp、%esp, 这些寄存器用来存储整型数据和指针。在大多数情况下, 前六个寄存器都可以看成是通用寄存器, 对它们的使用没有限制。但是, 有些指令是以固定的寄存器作为源和/或目的寄存器。最后两个寄存器%ebp 和%esp 保存着指向程序栈中重要位置的指针, 只有根据栈管理的标准惯例才能修改这两个寄存器中的值。

字节操作指令可以独立地读或者写寄存器%eax、%ebx、%ecx 或%edx 中的两个低位字节。这些低位字节有各自的名字, 其中四个寄存器的 0~7 位可以分别通过%al、%bl、%cl、%dl 来访问, 而 8~15 位则分别通过%ah、%bh、%ch、%dh 来访问。提供这样的特性是为了后向兼容 8008 和 8080。当一条字节指令更新这些单字节“寄存器元素”中的一个时, 该寄存器余下的三个字节不会被改变。

此外, 字操作指令可以读或者写上述八个寄存器中每一个的低 16 位。相应的名字是在对应的寄存器名中去掉字母'e', 即分别是%ax、%bx、%cx、%dx、%si、%di、%bp 和%sp。提供这个特性是因为 IA32 是从 16 位微处理器演化而来的。

### 8.2.2.2 条件码寄存器

除了整数寄存器, CPU还包含一组单个位的条件码(condition code)寄存器, 它们描述最近的算术或逻辑操作(见第 173 页的 8.2.5.2 节)的属性。对于这些寄存器的检测, 将有助于执行条件分支指令。最有用的条件码是:

- CF: 进位标志。最近的操作使最高位产生了进位, 它可以用来检查无符号操作数的溢出。
- ZF: 零标志。最近的操作得到的结果为 0。
- SF: 符号标志。最近的操作得到的结果是负数。
- OF: 溢出标志。最近的操作导致一个二进制补码溢出(正溢出或负溢出)。

## 8.2.3 操作数的格式

大多数指令有一个或多个操作数, 指示执行一个操作所要引用的源数据值, 以及存放结果的目的位置。IA32 支持多种操作数格式(如表 8-1)。源数据值可以以常数形式给出(即立即数, immediate), 或者从寄存器或内存中读出, 结果可以存放在寄存器或内存中。因此, 各种操作数的可能性被分为三种类型: 立即数、寄存器、内存引用。

### 1、立即数

在 GAS 中, 立即数的表示方式是“\$”后面跟一个整数, 如\$-123 或\$0x2B, 这里的整数采用标准 C 语言中的表示方法。

## 2、寄存器

对双字操作来说，存放操作数的寄存器可以是八个 32 位寄存器中的一个，如 `%eax`；对字节操作来说，则可以是八个单字节寄存器元素中的一个，如 `%al`。在表 8-1 中，我们用符号  $E_a$  表示任意寄存器  $a$ ，用引用  $R[E_a]$  表示寄存器  $E_a$  的值。

## 3、内存引用

当操作数是内存引用时，需要先计算确定内存地址，然后再根据这个地址来访问内存单元。由于一般将内存看成是一个很大的字节数组，故用符号  $M_b[Addr]$  表示对存储在内存中从地址  $Addr$  开始的  $b$  字节值的引用，其中字节数  $b$  取决于内存引用所处的指令类型，即是字节指令、字指令，还是双字指令。为简便起见，这里省去符号中的下标  $b$ 。

表 8-1 列出了各种可能的内存寻址模式(addressing mode)，以允许不同形式的内存引用。 $Imm(E_b, E_i, s)$  是最一般的形式，它包含四个部分：一个立即数偏移  $Imm$ ，一个基址寄存器  $E_b$ ，一个变址或索引寄存器  $E_i$  和一个伸缩因子(scalar factor)  $s$ ，这里的  $s$  必须是 1、2、4 或者 8。

表 8-1 操作数的格式

类型	格式	操作数的值	名称
立即数	$Imm$	$Imm$	立即数寻址
寄存器	$E_a$	$R[E_a]$	寄存器寻址
内存引用	$Imm$	$M[Imm]$	绝对寻址
内存引用	$(E_a)$	$M[R[E_a]]$	间接寻址
内存引用	$Imm(E_b)$	$M[R[E_b] + Imm]$	(基址+偏移量)寻址
内存引用	$(E_b, E_i)$	$M[R[E_b] + R[E_i]]$	变址寻址
内存引用	$Imm(E_b, E_i)$	$M[R[E_b] + R[E_i] + Imm]$	变址寻址
内存引用	$(, E_i, s)$	$M[R[E_i] \cdot s]$	伸缩化的变址寻址
内存引用	$Imm(, E_i, s)$	$M[R[E_i] \cdot s + Imm]$	伸缩化的变址寻址
内存引用	$(E_b, E_i, s)$	$M[R[E_b] + R[E_i] \cdot s]$	伸缩化的变址寻址
内存引用	$Imm(E_b, E_i, s)$	$M[R[E_b] + R[E_i] \cdot s + Imm]$	伸缩化的变址寻址

### 8.2.4 程序栈

### 8.2.5 一些常用的指令

#### 8.2.5.1 数据传送指令

最频繁使用的指令是执行数据传送的指令。表 8-2 列出了一些重要的数据传送指令。其中  $S$  代表源操作数，它指定一个值，可以是立即数，也可以存放在寄存器或内存中； $D$  代表目的操作数，它指定一个位置，可以是寄存器，也可以是内存地址。IA32 对传送指令加了一条限制，即传送指令的两个操作数不能都指向内存位置。若想将一个值从一个内存位置复制到另一个内存位置，就需要两条指令：第一条指令将源值加载到寄存器中，第二条将该寄

寄存器的值写入到目的位置。

表 8-2 数据传送指令

指令	执行效果	描述
<code>movl S, D</code>	$S \rightarrow D$	传送双字
<code>movw S, D</code>	$S \rightarrow D$	传送字
<code>movb S, D</code>	$S \rightarrow D$	传送字节
<code>movsbl S, D</code>	符号扩展( $S \rightarrow D$ )	传送符号扩展的字节
<code>movzbl S, D</code>	零扩展( $S \rightarrow D$ )	传送零扩展的字节
<code>pushl S</code>	$R[\%esp]-4 \rightarrow R[\%esp];$ $S \rightarrow M[R[\%esp]]$	压栈
<code>popl D</code>	$M[R[\%esp]] \rightarrow D;$ $R[\%esp] + 4 \rightarrow R[\%esp];$	出栈

`movsbl` 和 `movzbl` 指令的源操作数都是单字节的，两条指令都是将源操作数复制到目的操作数的低 8 位，并设置目的操作数的高 24 位。只是 `movsbl` 是执行符号扩展，即将高 24 位设置为源操作数的最高位；而 `movzbl` 是执行零扩展，即将高 24 位均设置为 0。

`pushl` 和 `popl` 都是用来将数据压入栈中和从栈中弹出数据的。程序栈存放在内存中的某个区域，栈是向下增长的，这样栈顶元素的地址就是所有栈中元素地址中最低的。`%esp` 保存着栈顶元素的地址。从执行行为上看，指令 `pushl %ebp` 等价于指令 `subl $4,%esp` 和 `movl %ebp,(%esp)` 的顺次执行；而指令 `popl %eax` 等价于指令 `movl (%esp), %eax` 和 `addl $4,%esp` 的顺次执行。但是，它们的区别在于直接使用 `pushl` 或 `popl` 能减少目标代码的长度。

### 8.2.5.2 算术和逻辑指令

表 8-3 列出了一些双字整数的算术和逻辑指令。除了加载有效地址(Load Effective Address)指令 `leal` 以外，每条指令都有对应的对字和对字节操作的指令。例如，`addl` 对应有 `addw` 和 `addb`。

`leal` 指令实际上是 `movl` 指令的变形。它与 `movl` 的区别在于，`leal` 指令并不从指定的源操作数位置读入数据，而是将源操作数的有效地址写入到目的操作数中。例如，如果寄存器 `%eax` 的值为  $x$ ，则指令 `leal 6(%eax), %edx` 将设置寄存器 `%edx` 的值为  $x+6$ 。此外，`leal` 指令不改变任何条件码。

### 8.2.5.3 条件码的设置与访问

在 8.2.2.2 节中，我们介绍了几个最有用的条件码寄存器。除 `leal` 指令以外，所有在字、字节、双字上操作的算术和逻辑指令(参见表 8-3)都会设置条件码。

`cmpb`、`cmpw` 和 `cmpl` 这些比较指令以及 `testb`、`testw` 和 `testl` 这些测试指令在执行时也会对条件码进行设置，但是不改变任何其他寄存器。比较指令的格式形如 “`cmpb S2, S1`”，它根据  $S1-S2$  的值来设置条件码，如果两个操作数相等，就会将零标志 ZF 设置为 1，而其他的标志可以用来确定两个操作数之间的大小关系。测试指令的格式形如 “`testb S2, S1`”，它根据  $S1 \& S2$  的值来设置零标志 ZF 和负数标志 SF。通常可以利用测试指令来测试一个操

作数是否是负数、零或正数，如利用指令 `testl %eax, %eax` 测试 `%eax`；也可以利用测试指令来测试操作数中的某些位。

表 8-3 一些双字整数操作指令

指令	执行效果	描述
<code>leal S, D</code>	$\&S \rightarrow D$	加载有效地址
<code>incl D</code>	$D + 1 \rightarrow D$	自增 1
<code>decl D</code>	$D - 1 \rightarrow D$	自减 1
<code>negl D</code>	$-D \rightarrow D$	取负
<code>notl D</code>	$\sim D \rightarrow D$	取补码
<code>addl S, D</code>	$D + S \rightarrow D$	加法
<code>subl S, D</code>	$D - S \rightarrow D$	减法
<code>imul S, D</code>	$D * S \rightarrow D$	乘法
<code>xorl S, D</code>	$D \wedge S \rightarrow D$	按位异或
<code>orl S, D</code>	$D   S \rightarrow D$	按位或
<code>andl S, D</code>	$D \& S \rightarrow D$	按位与
<code>sall k, D</code>	$D \ll k \rightarrow D$	将 $D$ 中的值左移 $k$ 位
<code>shll k, D</code>	$D \ll k \rightarrow D$	将 $D$ 中的值左移 $k$ 位，与 <code>sall</code> 等同
<code>sarl k, D</code>	$D \gg k \rightarrow D$	将 $D$ 中的值算术右移 $k$ 位，即填上符号位
<code>shrl k, D</code>	$D \gg k \rightarrow D$	将 $D$ 中的值逻辑右移 $k$ 位，即填上 0

表 8-4 set 指令（根据条件码，对字节进行设置）

指令	执行效果	描述
<code>cmpl S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较双字。这条指令会执行 <code>subl S, D</code> ，然后值设置 <code>SF</code> 和 <code>ZF</code> ，最后抛弃减法结果。
<code>cmpw S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较字。这条指令会执行 <code>subl S, D</code> ，然后值设置 <code>SF</code> 和 <code>ZF</code> ，最后抛弃减法结果。
<code>cmpb S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较字节。这条指令会执行 <code>subl S, D</code> ，然后值设置 <code>SF</code> 和 <code>ZF</code> ，最后抛弃减法结果。
<code>testl S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较双字，只改变条件码的值。
<code>testw S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较字，只改变条件码的值。
<code>testb S, D</code>	$S - D > 0 ? 1 : 0 \rightarrow SF$ $S - D > 0 ? 1 : 0 \rightarrow ZF$	比较字节，只改变条件码的值。

通常，并不直接读取条件码，而是根据条件码的某个组合来设置一个整数寄存器或是执

行一条条件分支指令。

表 8-4 中列出的各种 set 指令是根据条件码的某个组合,将字节型的目的操作数(如字节寄存器%al 等,或者是存储一个字节的内存位置)设置为 0 或者 1。

#### 8.2.5.4 流程控制转移指令

在程序中,可以改变顺序执行的流程而转向所需执行的指令,这样的控制流由控制转移指令来操作。其中,控制转移指令又分条件转移指令和非条件转移指令。条件转移指令是指在满足特定条件时才改变指令的执行顺序的指令;而非条件转移指令会无条件地改变指令执行顺序到所指定的位置。通常,执行条件转移指令之前,需要执行条件判断指令,如(cmp<sub>l</sub> %eax, %ebx),来更改某些条件码,然后条件转移指令会访问条件码的值来作出是否跳转的决定。

另外针对循环语句,AT&T汇编也加入了循环跳转指令,它属于条件转移指令的一种。但是在GCC中,这些指令没有被使用,故这里不考虑这类指令。在下面的指令介绍中,我们把控制转移指令分成无符号跳转指令(表 8-5)、有符号跳转指令(表 8-6)和其它跳转指令(表 8-7)来介绍。

表 8-5 无符号跳转指令

指令	跳转条件	描述
jmp Label	永真	直接跳转
jmp *Operand	永真	间接跳转
ja/jnbe D	$\sim\text{CF} \& \sim\text{ZF}$	当比较结果是大于时跳转
jae/jnb D	$\sim\text{CF}$	当比较结果是大于等于时跳转
jb/jnae D	CF	当比较结果是小于时跳转
jbe/jna D	$\text{CF} \mid \text{ZF}$	当比较结果是小于等于时跳转
jc D		当进位标志 CF 被设置时跳转
je/jz D	ZF	当比较结果是等于时跳转
jne/jnz D	$\sim\text{ZF}$	当比较结果是不等于时跳转

表 8-6 有符号跳转指令

指令	跳转条件	描述
jg/jnle D	$\sim(\text{SF} \wedge \text{OF}) \& \sim\text{ZF}$	当比较结果是大于时跳转
jge/jnl D	$\sim(\text{SF} \wedge \text{OF})$	当比较结果是大于等于时跳转
jl/jnge D	$(\text{SF} \wedge \text{OF})$	当比较结果是小于时跳转
jle/jng D	$(\text{SF} \wedge \text{OF}) \mid \text{ZF}$	当比较结果是小于等于时跳转
js D	SF	当符号标志 SF 不为零时跳转
jns D	$\sim\text{SF}$	当符号标志 SF 为零时跳转
jo D	OF	当溢出标志 OF 不为零时跳转
jno D	$\sim\text{OF}$	当溢出标志 OF 为零时跳转

表 8-7 其它跳转指令

指令	跳转条件	描述
call Subroutine		调用方法名为 Subroutine 的子进程
ret Imm		子进程返回。如果 ret 不带参数，则系统仅仅是弹去栈顶值，不做其它任何事情。如果带一个参数 Imm 立即数，则在弹栈后还要舍去栈中 Imm 个比特的值。
int Imm		调用中断过程号为 Imm

### 8.2.5.5 其它指令

GAS 中还保留了一个 nop 指令，执行它会使 CPU 流水空转一个周期。

### 8.2.6 gcc 与 gdb

课程设计所用的汇编代码是 x86 汇编语言，但是你在实际实验时只会涉及到很少的汇编知识，你只要能掌握源语言中的一个语句能用什么样的汇编操作码来实现即可。

为帮助你理解这个 x86 汇编代码的生成过程，我们提供一个简单的汇编程序样例 simple.s，它是从 simple.mj 转化而来。

课程设计用到的工具主要是 gcc 和 gdb，在 Linux 下，一般已经装好了 gcc 和 gdb。如果没有安装，请分别到 <http://gcc.gnu.org> 和 <http://www.gnu.org/software/gdb/gdb.html> 下载安装。在 Windows 下建议使用 cygwin，它提供了 Windows 下的 Linux 环境，包含了 gcc 和 gdb，请在 <http://www.cygwin.com/> 下载安装。你也可以使用 MinGWStudio，它是 gcc 的 Windows 实现。

你可以使用 gcc 编译所生成的汇编语言文件，得到可执行文件（unix 下叫 a.out），如下：

```
gcc simple.s
```

你也可以先把 mj 文件转化为对应的 C 程序文件；然后，使用“-S”选项，执行 gcc 对该 C 程序文件进行编译，产生对应的汇编文件。我们提供的样例 simple.s 就是这么得来的。

```
gcc -S simple.c
```

你可以先从 simple.s 中感受一下汇编代码与 MiniJOOL 语言的对应，然后再尝试生成代码。

你可以使用工具 gdb 调试汇编程序，若要使用 gdb 进行调试，则要求在用 gcc 编译时加入 -g 参数。

```
$gcc -o output -s input.s -g
```

然后调用 gdb，

```
$gdb output
```

进入 gdb 后读入文件：

```
(gdb) file output
```

就可以开始调试了。你可以查看程序每步执行情况和寄存器状态，可以单步执行汇编程序。

常用的命令有以下几个：

表 8-8 gdb 常用命令

名称 (简写)	说明
step (s)	执行一步
run (r)	连续运行
print (p)	打印变量值，如：print a
continue (c)	恢复执行
breakpoint (b)	设断点
information (i)	查看变量
register (r)	查看寄存器

更多的用法请参考：<http://www.unknownroad.com/rtfm/gdbtut/gdbadvanced.html>

## 8.3 一些语句到 x86 汇编代码的映射

语句到 x86 汇编代码的映射一般并不唯一，本节给出的只是我们针对此的理解，供你在实现时作为参考。

### 8.3.1 总体生成的汇编代码结构

```
{
.text
.def __main; .scl 2; .type32; .endef
LC0:
.ascii "%s\12\0"
LC1:
.ascii "%d\12\0"
.
其余字符串
.
.align 2
.global _main
.def _main; .scl 2; .type32; .endef
_main:
pushl    %ebp
movl     %esp, %ebp
subl     $24, %esp
```

```

    andl $-16, %esp
    movl  $0, %eax
    movl  %eax, -16(%ebp)
    movl  -16(%ebp), %eax
    call  __alloca
    call  __main
    代码部分
    movl  %ebp, %esp
    popl %ebp
    ret
    其余函数代码部分
}

```

### 8.3.1.1 语句到 x86 汇编代码的映射

#### 1、赋值语句

**赋值语句：【标识符 = 表达式;】**

对应的汇编代码

```

{
    表达式计算代码部分 //结果在%eax 中
    movl  %eax, (标识符的存储位置)
}

```

#### 2、前缀表达式

**前缀表达式：【前缀操作符 表达式】**

```

{
    表达式计算代码部分 //结果在%eax 中
    根据前缀操作符对%eax 作相应的操作的代码 //其后的值仍然在%eax 中
}

```

#### 3、中缀表达式

**中缀表达式：【左边表达式 中缀操作符 右边表达式】**

```

{
    左边表达式代码部分 //值在%eax 中
    movl  %eax, %ecx //左边值到了%ecx 中
    右边表达式代码部分 //值在%eax 中
    根据中缀操作符对两个寄存器作相应操作的代码 // 结果在%eax 中
}

```



#### 4、if 语句

**if 语句有两种：**

**【if(bool 表达式) then 语句 else 部分】**

```
{  
Bool 表达式计算代码 //值在%eax 中  
cmpl    $0, %eax  
je      Lx  
then 语句代码  
jmp     Ly  
Lx:  
else 部分代码  
Ly:  
}
```

**【if(bool 表达式) then 语句】**

```
{  
Bool 表达式计算代码 //值在%eax 中  
cmpl    $0, %eax  
je      Lx  
then 语句代码  
Lx:  
}
```

#### 5、while 语句

**while 语句: 【while (bool 表达式) 循环体】**

```
{  
Lx:  
bool 表达式代码 //值在%eax 中  
cmpl    $0, %eax  
jne     Ly  
jmp     Lz  
Ly:  
循环体代码  
jmp     Lx  
Lz:  
  
}
```

## 6、函数定义

**函数定义:**【类型 函数名 (参数列表) 函数体】

```
{
.align 2
.global _函数名
.def _函数名;.scl 2; .type32; .endef
_函数名:
pushl %ebp
movl    %esp, %ebp
函数体部分代码
}
```

## 7、函数调用

**函数调用:**【函数名 (实参表达式列表)】

```
{
esp 寄存器处理代码 //防止栈溢出
实参表达式计算代码
实参结果入栈的固定位置代码
重复 n 次, n 等于实参个数
call _函数名
}
```

## 8、整数和 bool 常量

**整数和 bool 常量:**【常量】

```
{
movl $常量值, %eax //bool 值转化为整数
}
```

## 9、return 语句

**return 语句:**【return 表达式;】

```
{
表达式计算代码 //值在%eax 中
popl %ebp
ret
}
```

## 8.4 x86 寄存器分配器

x86 寄存器分配采用局部寄存器分配算法, 由于许多 x86 指令隐含有对寄存器的操作,

这使得其寄存器分配变得复杂。我们在 `edu.ustc.cs.compile.arch.jar` 类库中提供了 x86 局部寄存器分配器的实现。

与 MIPS 汇编代码生成器不同的是，x86 汇编代码生成器首先采用程序中的变量名称来表示指令中的操作数（如寄存器或内存地址），而不是像 MIPS 汇编代码生成器那样用寄存器变量来预先分配寄存器位置。之后，x86 汇编代码生成器再调用寄存器分配器获得所分配的寄存器，如果寄存器不够，寄存器分配器会自动生成溢出代码。

我们提供的局部寄存器分配器包含在包 `edu.ustc.cs.compile.arch.x86.regalloc` 中。这个包中含有类 `RegAllocator` 等。下面简要介绍 `RegAllocator` 的使用方法：

1、在生成汇编代码的内部表示时请保留程序中声明的各个变量的名称（要保证这些名称和变量能够一一对应），请注意同名变量的覆盖问题。这里提出几个方法：

- 1) 保留变量的名称，但要注意不同变量在作用域重叠时同名的情况。
- 2) 将变量的名称统一换成另一种格式，如 `RegisterVariable` 实例。

2、针对生成的汇编代码的内部表示，需要再对它做一次扫描，将变量名称替换为寄存器名称。你可以用类似于如下的代码来完成：

```
RegAllocator ra = new RegAllocator();
...
//下面对生成的汇编码序列作一次扫描，对每个要替换的变量名称，需要做如下处理：
//start of process
//在寄存器分配器内会维护一个变量名称表，用 RegAllocator 中的 getReg 方法可以
//试着得到一个空的 reg
X86Register reg = ra.getReg(variableName);
if (reg == null){
    //如果当前没有空的寄存器，就溢出一个寄存器，即产生一段溢出代码
    //首先选择一个要溢出的寄存器，用 LRU 算法。
    reg = ra.selectReg();//一定要先 selectReg 再 spill，目的是在分配器中更新记录
    //自动生成一段溢出代码，导入生成的指令序列
    AssemblySequence spill = (ra.spill(reg));
    //这里需要用将溢出的代码序列导入到生成的汇编码指令中，使之能够
    //正确地溢出寄存器
    addSpill2Stmts();
}
//现在 reg 是可以用的了，接下来请生成一条新的指令或者更改原来的指令中变量
//的名称为 reg，并把它再加入 assemblySequence 中。
```

你最好由寄存器分配器来管理所有的寄存器分配。有些情况下，某些指令需要特定的寄存器（比如 `mutl` 指令隐含需要 `eax` 寄存器），这时候我们需要得到指定寄存器，其作法如下：

```
//假定我们需要使用 eax 寄存器
```

```

if (ra.isFree(X86Register.eax)){//如果 eax 是空的，则直接使用它。
    //TODO
}else{//否则，就应该溢出 eax
    //首先告诉分配器要使用 eax:
    ra.useReg(X86Register.eax);
    //然后产生溢出代码,加入到生成的序列中的正确位置后，eax 就是可以使用的了。
    AssemblySequence spill = ra.spill(X86Register.eax);
    //TODO
}

```

## 8.5 课程设计 6-1

在本课程设计中，你需要为 SkipOOMiniJOOOL 语言的中间表示生成相应的 x86 汇编代码。在汇编代码生成的过程中，你可以使用由我们提供的寄存器分配器来进行寄存器分配工作，你需要处理其他代码生成的相关工作以生成正确的 x86 汇编代码。

课程设计 6-1 关联的 Java 源文件（相对于路径 `src/edu/ustc/cs/compile/generator/x86`）有：

- **Generator1.java**: 一个汇编代码生成器的框架代码，其中类 `Generator1` 是汇编代码生成器接口 `GeneratorInterface` 的一个实现；

关联的其他文件有：

- **bin 目录**
  - **lab6-1.xml**: 课程设计 6-1 的 ant 编译文件。
- **config 目录**
  - **lab6-1.xml**: 用实验平台运行课程设计 6-1 的配置文件。

这个课程设计中还将使用 `ROOT_DIR/lib` 中如下类包：

- **edu.ustc.cs.compile.parser.skipoominijool.jar**: 课程设计 5-1 中默认使用我们提供的 SkipOOMiniJOOOL 的语法生成器获得 SkipOOMiniJOOOL 程序的中间表示；
- **edu.ustc.cs.compile.arch.jar**: 汇编代码的中间表示。

### 8.5.1 课程设计指导

汇编代码生成器以 SkipOOMiniJOOOL 程序的 AST 表示为输入，它需要利用访问者类来遍历 AST 树，在附加相关信息的帮助下（如符号表），生成有效的汇编代码序列（此时不考虑寄存器分配问题）。最后，针对 x86 的特性，用我们提供的 x86 寄存器分配器来将其中的变量置换成相应的寄存器，然后输出到文件中。

本章的课程设计与上一章的课程设计密切相关，上一章的课程设计的指导原则同样可以用来指导本章的课程设计的实现。但是，要注意如下几点区别：

1. x86 寄存器分配器采用局部寄存器分配算法。在为不同的函数生成寄存器分配策略

之前,请根据需要保存仍在使用中的寄存器。我们建议根据惯例,在调用子过程之前保存某些寄存器,(寄存器`%eax`、`%edx`、`%ecx`被划分为调用者保存),同样在子过程中按需要保存其它的寄存器。

2. 受汇编码表示的影响,x86 寄存器分配是按变量名称(表示实际的内存地址或寄存器操作数)来分配寄存器的,其中没有使用寄存器变量。因此需要注意同名变量在作用域重叠时的情况。x86 寄存器分配器的使用比 MIPS 的要麻烦一些,请在阅读过 x86 寄存器分配器的框架说明后再使用它。
3. 同样受结构的影响,X86 寄存器分配器提供了生成溢出代码操作的指令序列。因此不需要你手动编写溢出寄存器的相关代码,但是需要注意如何将生成的溢出代码指令序列插入到正确的位置。

## 8.5.2 汇编代码生成器的编译和运行

你可以在 bin 目录下执行命令:

```
ant lab6-1
```

编译本课程实验,获得一个 x86 汇编代码生成器。

然后,你可以在 bin 目录下执行命令启动实验平台运行 x86 汇编代码生成器:

```
run -cf ../config/lab6-1.xml
```

或 `./run.sh -cf ../config/lab6-1.xml`

## 8.6 课程设计 6-2

我们已经在课程设计 6-1 中实现了一个代码生成器,在这个课程设计中我们要自己实现一个新的寄存器分配算法,使之能够正确并有效地生成汇编代码。

课程设计 6-2 关联的 Java 源文件(相对于路径 `src/edu/ustc/cs/compile/generator/x86`)有:

- **Generator2.java:** 一个汇编代码生成器的框架代码,其中类 `Generator2` 是汇编代码生成器接口 `GeneratorInterface` 的一个实现;

关联的其他文件有:

- **bin 目录**
  - **lab6-2.xml:** 课程设计 6-2 的 ant 编译文件。
- **config 目录**
  - **lab6-2.xml:** 用实验平台运行课程设计 6-2 的配置文件。

这个课程设计中还将使用 `ROOT_DIR/lib` 中如下类包:

- **edu.ustc.cs.compile.parser.skipoominijool.jar:** 课程设计 6-2 中默认使用我们提供的 `SkipOOMiniJool` 的语法生成器获得 `SkipOOMiniJool` 程序的中间表示;
- **edu.ustc.cs.compile.arch.jar:** 提供汇编代码的内部表示及寄存器分配。

在 7.9 节中已经提到过不少参考资料,此处再增添部分与 x86 相关的书籍。

(美) 费雷泽 (Fraser, C.W.), 可变目标 C 编译器—设计与实现。该书讲述了 LCC 编译器如何生成 MIPS、X86、SPARCS 三种不同的代码, 其中 x86 部分具有不错的参考价值。

关于汇编代码生成器的编译与运行与 7.8.2 类似, 这里不再赘述。

## 第9章 综合课程设计

### 9.1 课程设计要求

根据老师的要求，每个学生以 SkipOOMiniJOOB 语言作为要实现的源语言，独立完成这个语言的编译器的前端（或后端），并自行选择完成后端（或前端）的合作伙伴。

前端要求完成：词法分析、语法分析、语义检查并生成抽象语法树（AST）。后端则要求由 AST 生成 x86 汇编代码或 MIPS 汇编代码，不要求代码优化，但需要考虑寄存器分配等问题；生成的 x86 汇编代码应能直接用 gcc 汇编连接得到可执行文件，而生成的 MIPS 汇编代码则应能在 SPIM 上执行。

前后端的学生需要定义好接口，不开放源代码给对方，而只提供 jar 文件和接口说明，在运行时应能输出前端和后端的作者名。

### 9.2 课程设计开发的目录结构

- **ROOT\_DIR** 目录：即本书实验软件包存放的根目录，如 E:\CompilerProj\student
  - **lab** 目录：前面各章课程设计的软件包，参见 3.1 节。
  - **lib** 目录：存放分析器、汇编码表示及寄存器分配等的类库文件，参见 3.1 节。
  - **lib/AST** 目录：按版本分子目录存放 Eclipse AST 的相关类库文件。
  - **platform/config** 目录：实验平台的配置文件样本、描述配置文件的 XML Schema 文件。
  - **platform/lib** 目录：实验平台的类库文件。
  - **doc** 目录：一些类库文件的接口说明文档。
  - **tools** 目录：分子目录存放 JFlex、CUP、JavaCC 等的类库文件。
  - **test** 目录：存放我们提供的测试文件。
  - **PB05011** 目录：按“PB05011001\_姓名”分子目录存放学生的实验内容
  - **PB05011/demo** 目录：我们提供的一个开发框架，学生可以参照其中的文件和目录结构来构造你的开发目录。

### 9.3 课程设计提交的目录结构

以下是学生提交课程设计的目录结构，我们将把这个目录复制到 **ROOT\_DIR/PB05011** 下，来测试你的编译器，请务必遵守这些约定。

- **PB05011001\_姓名** 目录
  - ◆ **bin** 目录
 

存放 ant 编译文件、可执行文件等。
  - ◆ **config** 目录
 

存放实验平台配置文件以及语言的词法、文法规范描述文件等。

◆ **lib** 目录

存放你的合作伙伴以及你所做的编译器前端和后端的类库文件，类库文件的命名方式是：“学号\_frontend.jar”或者“学号\_backend.jar”。

◆ **doc** 目录

存放你的接口说明文档以及设计实现文档。

◆ **src** 目录

存放你的 Java 源程序。

◆ **test** 目录

存放你所使用的测试程序。

**注意：**

- 1) 不得提交生成的 class 文件。
- 2) Java 源程序中应该加有注释，能用 Javadoc 生成 HTML 文档。
- 3) 实验所需的 AST 类库文件在 **ROOT\_DIR/lib/AST** 下，按版本分子目录存放（参见 3.1 节）；所需的实验平台类库文件在 **ROOT\_DIR/platform/lib** 下；所需的分析器、汇编码表示及寄存器分配等的类库文件在 **ROOT\_DIR/lib** 下。请修改你的实验平台配置文件和 ant 编译文件，使用相对路径设置这些资源的位置。
- 4) 是否遵循软件工程规范的要求，例如目录结构，也是考评的内容之一。

## 9.4 课程设计的时间节点

- 1) 5 月份开始，可在电三楼 517 机房上机。
- 2) 安排两次课堂辅导（张昱老师）：第一次介绍可以利用的源代码（4 月下旬），第二次编译器设计答疑（5 月 10 日左右）。
- 3) 6 月 13 日发布测试程序。
- 4) 6 月 20 日 24 点以前通过网络提交课程设计，提交内容：编写的源代码、类库文件、测试程序、设计文档等。提交方式另行通知。
- 5) 考评安排在 6 月下旬，具体时间另行通知。

## 9.5 课程设计的考评方法

在考评方面，将学生分成若干组，每组的 10 人。各组用一个上午或下午的时间进行现场测试、答辩和公开评分；评委由教师、研究生助教和同组所有同学担任，教师主导测试过程、学生自己动手按老师要求操作，并用投影机当众显示测试过程；老师和同学均可以提问，学生需当众回答，所提问题主要围绕完成的设计和编程，以及测试中暴露出的设计或编程错误等展开。

评委的评分依据工程的规范性、编译器的正确性、错误定位与恢复能力、所生成的目标代码质量、回答问题时所表现出的对本课程设计所涉及知识的掌握程度，对自己设计和编码的前端（后端）的熟悉程度，操作的熟练程度，提交物的完整性、条理性及其中反映的分析和设计的思想，等等。每个评委当场给该组的全部同学排名次；由助教根据所有有效排名表，



给出最终的排名；由老师根据本组的情况确定本组的最高分和最低分，并依据排名，主要按等间隔确定每个同学的分。

其他一些评分细则包括：

- 如被老师、助教和过半数同学认为所提交的不是自己的课程设计成果时，为 0 分；
- 未按时提交也是 0 分；
- 独自完成整个编译器，分组评定成绩后降 10 分；
- 当前后端人数比例严重失调，则抬高少数人一端分数；
- 若所开发的前端（或后端）被多个同学（开发的合作伙伴除外）采用，则在分组评分的基础上加分，加分原则是：
  - 每增加两个采用者加 1 分；
  - 课程设计和平时作业合计不超过 50 分。

## 参考文献

- [1] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java<sup>TM</sup> Language Specification* ( Second Edition). Addison Wesley Longman, Inc., ISBN 0-201-31008-2, June 2000.
- [2] Scott E. Hudson, *CUP User's Manual (v0.11a)*, March 2006.  
Available at <http://www.cs.princeton.edu/~appel/modern/java/CUP/>;  
<http://www2.cs.tum.edu/projects/cup/>.
- [3] S.C.Johnson, *YACC—yet another compiler-compiler*, Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.
- [4] *Berkeley Yacc(v1.9)*, 1993. Available at <ftp://ftp.cs.berkeley.edu/ucb/4bsd/>.
- [5] <http://www.gnu.org/software/bison/bison.html>.
- [6] <http://compilers.iecc.com/comparch/article/98-06-039>.
- [7] The GNU Assembler, <http://wzt.wzt.googlepages.com/TheGNUAssembler.htm>

## 附录

## 附录 1 MiniJOOL 语言的词法记号 ID

词法记号 ID	含义	词法记号 ID	含义
CLASS	“class”	MINUS	“-”
STATIC	“static”	MULT	“*”
FINAL	“final”	DIV	“/”
EXTENDS	“extends”	NOT	“!”
VOID	“void”	MOD	“%”
INT	“int”	LT	“<”
BOOLEAN	“boolean”	GT	“>”
STRING	“String”	LTEQ	“<=”
IF	“if”	GTEQ	“>=”
ELSE	“else”	EQEQ	“==”
WHILE	“while”	NOTEQ	“!=”
BREAK	“break”	ANDAND	“&&”
CONTINUE	“continue”	OROR	“  ”
RETURN	“return”	EQ	“=”
PRINT	“print”	MULTEQ	“*=”
READ	“read”	DIVEQ	“/=”
THIS	“this”	MODEQ	“%=”
SUPER	“super”	PLUSEQ	“+=”
NEW	“new”	MINUSEQ	“-=”
INSTANCEOF	“instanceof”	IDENTIFIER	标识符
LBRACE	“{”	INTEGER_LITERAL	整数，包括十进制、八进制(以 0 开头)、十六进制(以 0x 或 0X 开头)
RBRACE	“}”	BOOLEAN_LITERAL	“true”或“false”
LPAREN	“(”	STRING_LITERAL	双引号包含的字符序列
RPAREN	)”	NULL_LITERAL	“null”
LBRACK	“[”		
RBRACK	“]”		
SEMICOLON	“;”		
COMMA	“,”		
DOT	“.”		
PLUS	“+”		

## 附录 2 算符的优先级与结合性

优先级	类型	运算符	名称	结合性
-----	----	-----	----	-----

1		( ) [ ] ·	圆括号 下标 成员	自左至右
2	单目	+ - !	正 负 逻辑非	自右至左
3	算术(双目)	* / %	乘 除 取余	自左至右
4	算术(双目)	+ -	加 减	自左至右
5	数值比较 (双目)	< <= > >=	小于 小于或等于 大于 大于或等于	自左至右
6	数值相等 (双目)	== !=	等于 不等于	自左至右
7	逻辑 (双目)	&&	逻辑与	自左至右
8	逻辑 (双目)		逻辑或	自左至右
9	赋值 (双目)	= *= /= += -= %=	赋值或 算术赋值	自右至左

## 附录3 MiniJOOL 语言语法的 EBNF 表示

program	=	class_declaration { class_declaration }
class_declaration	=	“class” IDENTIFIER [ “extends” IDENTIFIER ] class_body
class_body	=	“{” class_body_declaration { class_body_declaration } “}”
class_body_declaration	=	field_declaration   method_declaration   constructor_declaration
field_declaration	=	[ “static” ] [ “final” ] type variable_declarators “;”
variable_declarators	=	variable_declarator { “,” variable_declarator }
variable_declarator	=	variable_declarator_id [ “=” variable_initializer ]
variable_declarator_id	=	IDENTIFIER
variable_initializer	=	expression   array_initializer
array_initializer	=	“{” [ expression { “,” expression } ] “}”
method_declaration	=	method_header method_body
method_header	=	[ “static” ] ( type   “void” ) method_declarator
method_declarator	=	IDENTIFIER “(” [ formal_parameter_list ] “)”
formal_parameter_list	=	formal_parameter { “,” formal_parameter }
formal_parameter	=	type variable_declarator_id
method_body	=	block
constructor_declaration	=	constructor_declarator constructor_body
constructor_declarator	=	IDENTIFIER “(” [ formal_parameter_list ] “)”
constructor_body	=	“{” [ explicit_constructor_invocation ] [ block_statements ] “}”
explicit_constructor_invocation	=	(“this”   “super”) “(” [ argument_list ] “)” “;”
argument_list	=	expression { “,” expression }
block	=	“{” [ block_statements ] “}”
block_statements	=	block_statement { block_statement }
block_statement	=	local_variable_declaration_statement   statement
local_variable_declaration_statement	=	type variable_declarators “;”
statement	=	“;”   statement_expression “;”   “break” “;”   “continue” “;”   “return” [ expression ] “;”   block   “if” “(” expression “)” statement   “if” “(” expression “)” statement “else” statement

	“while” “(” expression “)” statement
	“print” “(” expression “)” “;”
	“read” “(” lvalue “)” “;”
statement_expression	= assignment_expression
	method_invocation
	class_instance_creation_expression
primary	= literal
	“this”
	“(” expression “)”
	class_instance_creation_expression
	field_access
	method_invocation
	array_access
class_instance_creation_expression	= “new” class_type “(” [ argument_list ] “)”
field_access	= ( primary   “super” ) “.” IDENTIFIER
method_invocation	= ( name   (“super”   primary) “.” IDENTIFIER ) “(” [argument_list] “)”
array_access	= ( name   field_access ) “[” expression “]”
expression	= name   primary
	unary_operator expression
	expression binary_operator expression
	expression “instanceof” class_type
	assignment_expression
lvalue	= name   field_access   array_access   “(” lvalue “)”
assignment_expression	= lvalue assignment_operator assignment_expression
binary_operator	= “*”   “/”   “%”   “+”   “-”   “==”   “!=”   “<”   “<=”   “>”   “>=”   “  ”   “&&”
unary_operator	= “+”   “-”   “!”
primitive_type	= “int”   “boolean”
array_type	= (primitive_type   “String” ) “[” [ constant_expression ] “]”
class_type	= IDENTIFIER
constant_expression	= expression
literal	= INTEGER_LITERAL   BOOLEAN_LITERAL   STRING_LITERAL   NULL_LITERAL
name	= IDENTIFIER   name “.” IDENTIFIER

## 附录 4 SimpleMiniJOOL 语言语法的 EBNF 表示

```

program    = class_declaration
class_declaration = "class" "Program" "{" "static" "void" "main" "(" ")" block "}"
block      = "{" [block_statements] "}"
block_statements = statement { statement }
statement  = ";;"
            | assignment_expression ";"
            | "break" ";"
            | "continue" ";"
            | "return" ";"
            | block
            | "if" "(" expression ")" statement
            | "if" "(" expression ")" statement "else" statement
            | "while" "(" expression ")" statement
            | "print" "(" expression ")" ";"
            | "read" "(" lvalue ")" ";"

primary    = literal
            | "(" expression ")"
expression = name
            | primary
            | unary_operator expression
            | expression binary_operator expression
            | assignment_expression

binary_operator = "*" | "/" | "%" | "+" | "-"
               | "==" | "!=" | "<" | "<=" | ">" | ">=" | "||" | "&&"

unary_operator = "+" | "-" | "!"

assignment_expression = name assignment_operator assignment_expression
assignment_operator   = "=" | "*=" | "/=" | "%=" | "+=" | "-="

literal = INTEGER_LITERAL
name    = IDENTIFIER

```

## 附录 5 SkipOOMiniJOOL 语言语法的 EBNF 表示

program	=	class_declaration
class_declaration	=	“class” “Program” class_body
class_body	=	“{” class_body_declaration { class_body_declaration } “}”
class_body_declaration	=	field_declaration   method_declaration
field_declaration	=	[ “static” ] [ “final” ] type variable_declarators “;”
variable_declarators	=	variable_declarator { “,” variable_declarator }
variable_declarator	=	variable_declarator_id [ “=” variable_initializer ]
variable_declarator_id	=	IDENTIFIER
variable_initializer	=	expression   array_initializer
array_initializer	=	“{” [expression { “,” expression } ] [ “,” ] “}”
method_declaration	=	method_header method_body
method_header	=	[ “static” ] ( type   “void” ) method_declarator
method_declarator	=	IDENTIFIER “(” [formal_parameter_list] “)”
formal_parameter_list	=	formal_parameter { “,” formal_parameter }
formal_parameter	=	type variable_declarator_id
method_body	=	block
block	=	“{” [block_statements] “}”
block_statements	=	block_statement { block_statement }
block_statement	=	local_variable_declaration_statement   statement
local_variable_declaration_statement	=	type variable_declarators “;”
statement	=	“;”   statement_expression “;”   “break” “;”   “continue” “;”   “return” [expression] “;”   block   “if” “(” expression “)” statement   “if” “(” expression “)” statement “else” statement   “while” “(” expression “)” statement   “print” “(” expression “)” “;”   “read” “(” lvalue “)” “;”
statement_expression	=	assignment_expression



	method_invocation
primary	= literal
	“(” expression “)”
	method_invocation
	array_access
method_invocation	= name “(” [ argument_list ] “)”
argument_list	= expression { “,” expression }
array_access	= name “[” expression “]”
expression	= name
	primary
	unary_operator expression
	expression binary_operator expression
	assignment_expression
binary_operator	= “*”   “/”   “%”   “+”   “-”
	“==”   “!=”   “<”   “<=”   “>”   “>=”   “  ”   “&&”
unary_operator	= “+”   “-”   “!”
lvalue	= name   array_access   “(” lvalue “)”
assignment_expression	= lvalue assignment_operator
	assignment_expression
assignment_operator	= “=”   “*=”   “/=”   “%=”   “+=”   “-=”
type	= primitive_type   “String”   array_type
primitive_type	= “int”   “boolean”
array_type	= ( primitive_type   “String” ) “[” [ constant_expression ] “]”
class_type	= IDENTIFIER
constant_expression	= expression
literal	= INTEGER_LITERAL
	BOOLEAN_LITERAL
	STRING_LITERAL
name	= IDENTIFIER

## 附录 6 语法结构与 AST 节点的对应关系

AST 节点类	用处	说明
ArrayAccess	数组元素访问表达式	例如: <code>a[10]</code>
ArrayInitializer	数组初始化表达式	例如: <code>int[3] i = {1,2,3}</code> 中的 <code>{1,2,3}</code>
ArrayType	数组类型	java 中数组类型不需要指定数组长度, SkipOOMiniJOOl 和 MiniJOOl 中可能需要指定数组长度
Assignment	赋值表达式	例如: <code>a = 2</code>
Block	语句块	由花括号括起来的语句序列
BooleanLiteral	布尔型常量	
BreakStatement	break 语句	
ClassInstanceCreation	类创建表达式	例如: <code>new A(p1, p2)</code>
CompilationUnit	表示整个程序	是 AST 的根节点类型
ConstructorInvocation	构造器调用表达式	指形如 <code>this(p1,p2)</code> 的构造器调用
ContinueStatement	continue 语句	
EmptyStatement	空语句	
ExpressionStatement	表达式语句	将一个表达式封装成语句
FieldAccess	成员变量访问表达式	用于访问类中的非 static 变量。访问 static 成员需要使用 QualifiedName。
FieldDeclaration	变量声明语句	在 SkipOOMiniJOOl 中, 变量指全局变量; 在 MiniJOOl 中, 变量指类变量和实例变量
IfStatement	if 语句	
InfixExpression	中缀表达式	
InstanceOfExpression	instanceof 表达式	
MethodDeclaration	方法声明	同时包括了方法的定义
MethodInvocation	方法调用表达式	
Modifier	modifier	在 SkipOOMiniJOOl 和 MiniJOOl 中 modifier 包括: static 和 final
NullLiteral	null 常量	
NumberLiteral	数字常量	在 SkipOOMiniJOOl 和 MiniJOOl 中仅仅指

		整型常量
ParenthesizedExpression	括号表达式	
PrefixExpression	前缀表达式	
PrimitiveType	基本类型	在 SkipOOMiniJOOl 和 MiniJOOl 中包括 int, boolean 和 void
QualifiedName	限定名	在 MiniJOOl 中指形如 A.id 的表达式。其中 A 是类型名, id 是 A 中的成员。
ReturnStatement	return 语句	
SimpleName	简单名称	通常表示变量名, 方法名等
SimpleType	简单类型	在 SimpleMiniJOOl 和 MiniJOOl 中用来表示不在 PrimitiveType 中的类型
SingleVariableDeclaration	单变量声明表达式	
StringLiteral	字符串常量	
SuperConstructorInvocation	超类构造器调用表达式	指形如 super(p1,p2)的构造器调用
SuperFieldAccess	超类变量访问表达式	指形如 super.id 的超类变量访问
SuperMethodInvocation	超类方法调用表达式	指形如 super.f()的超类方法调用
ThisExpression	this 表达式	指形如 this.expression 的表达式
TypeDeclaration	类型声明	在 SkipOOMiniJOOl 和 MiniJOOl 中用于声明一个类
VariableDeclarationExpression	变量声明表达式	
VariableDeclarationStatement	变量声明语句	

## 附录7 MIPS 汇编语言的 EBNF 定义

(待修订)

Program	=	{Line}, EOF
Line	=	{Statement, “,”}, Statement, EOL BLANK
Statement	=	Directive   Label, “:”   Instruction
Directive	=	Director, [“ “, Argument], {“ ”, Argument}
Director	=	“align”   “ascii”   “asciiz”   “byte”   “data”   “extern”   “globl”   “half”   “kdata”   “ktext”   “space”   “text”   “word”
Argument	=	String   Expression
Expression	=	(Integer   Label) {“(” (“+”   “-”   “*”   “/”) (Integer   Label))”   (“+”   “-”   “*”   “/”) (Integer   Label)}
Label	=	Identifier
Instruction	=	Opcode, [“ “, Operand, [“ ”, Operand, [“ ”, Operand]]] “add”   “addi”   “addu”   “addiu”   “and”   “andi”   “div”   “divu”   “mul”   “mulo”   “mulou”   “mult”   “multu”   “neg”   “negu”   “nor”   “not”   “or”   “ori”   “rem”   “remu”   “rol”   “ror”   “sll”   “sllv”   “sra”   “sra”   “srl”   “srlv”   “sub”   “subu”   “xor”   “xori”   “li”   “lui”   “seq”   “sge”   “sgeu”   “sgt”   “sgtu”   “sle”   “sleu”   “slt”   “slti”   “sltu”   “sltiu”   “sne”   “b”   “bczt”   “bczf”   “beq”   “beqz”   “bge”   “bgeu”   “bgez”   “bgezal”   “bgt”   “bgtu”   “bgtz”   “ble”   “bleu”   “blez”   “bgezal”   “bltzal”   “blt”   “bltu”   “bltz”   “bne”   “bnez”   “j”   “jal”   “jalr”   “jr”   “la”   “lb”   “lbu”   “ld”   “lh”   “lhu”   “lw”   “lwc”   “lwl”   “ulh”   “ulhu”   “ulw”   “sb”   “sd”   “sh”   “sw”   “swc”   “swl”   “ush”   “usw”   “move”   “mfhi”   “mflo”   “mthi”   “mtlo”   “mfcz”   “mfc”   “mtcz”   “syscall”   “nop” Register,   Integer,
Operand	=	[Integer], “(”, Register, “)”   Label   Label, (“+”   “-”), Integer   Label, (“+”   “-”), Integer, “(”, Register, “)” “\$zero”   “\$at”   “\$v0”   “\$v1”   “\$v2”   “\$a0”   “\$a1”   “\$a2”   “\$a3”   “\$t0”   “\$t1”   “\$t2”   “\$t3”   “\$t4”   “\$t5”   “\$t6”   “\$t7”   “\$t8”   “\$t9”   “\$s0”   “\$s1”   “\$s2”   “\$s3”   “\$s4”   “\$s5”   “\$s6”   “\$s7”   “\$k0”   “\$k1”   “\$gp”   “\$sp”   “\$fp”   “\$ra”   “\$”, ((Digit, [Digit]) - (“0”, Digit))
Integer	=	[“-”, Digit, {Digit}]   “0”, (“x”   “X”), (Digit   Hex), {(Digit   Hex)}
String	=	newline \n tab \t

由双引号“ ”包括的任意字符串，其中特殊字符使用转义字表示：

	quote	\"
	slash	\\
	end	\0
Identifier	=	((Letter   "_"   "."), {(Letter   Digit   "_"   ".")}) - Opcode
Digit	=	"0"   "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"
Hex	=	"A"   "B"   "C"   "D"   "E"   "F"   "a"   "b"   "c"   "d"   "e"   "f" "A"   "B"   "C"   "D"   "E"   "F"   "G"   "H"   "I"   "J"   "K"   "L"   "M"   "N"   "O"   "P"   "Q"   "R"   "S"   "T"   "U"   "V"   "W"   "X"   "Y"   "Z"   "a"   "b"   "c"   "d"   "e"   "f"   "g"   "h"   "i"   "j"   "k"   "l"   "m"   "n"   "o"   "p"   "q"   "r" "s"   "t"   "u"   "v"   "w"   "x"   "y"   "z"
BLANK	=	空
EOL	=	表示行结束
EOF	=	表示文件结尾

## 附录 8 x86 的 AT&amp;T 汇编语言的 EBNF 定义

(待修订)

Director	=	“.data”   “.text”   “.align”   “.asciz”   “.ascii”   “.globl”   “.global”   “.end”   “.word”   “.file”   “.version”   “.type”   “.size”   “.ident”   “.comm”   “.include”   “.lcomm”   “.p2align”   “.if”   “.else”   “.elseif”   “.def”   “.endef”   “.endif”
Instruction	=	Opcode [“,”, Operand [“,”, Operand]]
Register	=	“%eax”   “%ebx”   “%ecx”   “%edx”   “%esi”   “%edi”   “%esp”   “%ebp”   “%cs”   “%ds”   “%es”   “%ss”   “%fs”   “%gs”
Operand	=	Register   Label   Integer   Register, “:”, Integer   [Register, “:”, [Integer]], “(”, Register, [“,”, Register, [“,” Integer]]“(”   [Register, “:”, Integer, “(”, “,”, Register, [“,” Integer]“(”
Integer	=	[“\$”], [“-”, Digit, {Digit}]
Opcode	=	“pushal”   “popal”   “pushf”   “popf”   “ret”   “hlt”   “jmp”   “call”   “ja”   “jnbe”   “jae”   “jnb”   “jb”   “jnae”   “jbe”   “jna”   “jg”   “jnle”   “jge”   “jnl”   “jl”   “jnge”   “jle”   “jng”   “je”   “jz”   “jne”   “jnz”   “jc”   “jnc”   “jno”   “jnp”   “jpo”   “loop”   “loope”   “loopz”   “loopne”   “loopnz”   “jcxz”   “jecxz”   “shl”   “sal”   “shr”   “sar”   “rol”   “ror”   “rcl”   “rcr”   “imul”   “mul”   “div”   “idiv”   “pushl”   “popl”   “inc”   “dec”   “negl”   “movl”   “addl”   “subl”   “cmpl”   “xchg”   “andl”   “orl”   “xorl”   “notl”   “test”
Argument	=	String   Expression
Expression	=	(Integer   Label) {“(”, (“+”   “-”   “*”   “/”) (Integer   Label))”   (“+”   “-”   “*”   “/”) (Integer   Label)}