北京英真时代科技有限公司

Engintime

CP Lab 编译原理实验指导

版本 3.1

北京英真时代科技有限公司

地址：北京市房山区拱辰大街 98 号 7 层 0825
邮编：102488
电话：010-60357081
手机：18501161622 / 18500560938
QQ：964515564
邮箱：support@tevation.com
网址：http://www.engintime.com

目 录

CP LAB简介 4
实验 1 实验环境的使用.. 7
实验 2 NFA 到 DFA.. 22
实验 3 使用LEX自动生成扫描程序..................................... 26
实验 4 消除左递归（无替换）....................................... 32
实验 5 消除左递归（有替换）....................................... 36
实验 6 提取左因子.. 41
实验 7 FIRST 集合.. 46
实验 8 FOLLOW 集合... 51
实验 9 YACC分析程序生成器.. 56
实验 10 符号表的构建和使用... 60
实验 11 三地址码转换为P-代码....................................... 64
实验 12 GCC编译器案例综合研究...................................... 68
附录 1 TINY编译器和TM机.. 77
参考文献 80

CP Lab 简介

获得人生中的成功需要的专注与坚持不懈多过天才与机会。

——C. W. Wendte

一、概述

CP Lab 是一款专用于高等院校编译原理实验教学的集成环境，具有可视化程度高、操作简便、扩展性强等特点。CP Lab 主要由两部分组成：

- 一个功能强大的 IDE 环境。
- 一套精心设计的编译原理实验题目和配套源代码。

CP Lab 所提供的 IDE 环境可以在 Windows 操作系统上快速安装、卸载，其用户界面和操作方式与 Microsoft Visual Studio 完全类似，有经验的读者可以迅速掌握其基本用法。该 IDE 环境提供的强大功能可以帮助读者顺利的完成编译原理实验，主要功能包括对实验源代码的演示功能、编辑功能、编译功能、调试功能和验证功能。

CP Lab 提供了一套精心设计的编译原理实验题目和配套源代码。实验题目涵盖了从词法分析、语法分析、语义分析、代码生成等所有重要的编译原理和算法。完成这些实验题目后，读者可以学习到通过手工编码是如何一步步实现一个编译器的，还可以使用 Lex、Yacc 工具完成简单的词法分析程序和语法分析程序，甚至还可以深入分析一个小型的开源编译器的实现方法。所有的源代码文件都使用 C 语言编写，可以与主流编译原理教材配套使用。这些源代码以模块化的方式进行组织，并配有完善的中文注释，可读性好，完全符合商业级的编码规范。

使用 CP Lab 进行编译原理实验的过程可以参见图 0-1。

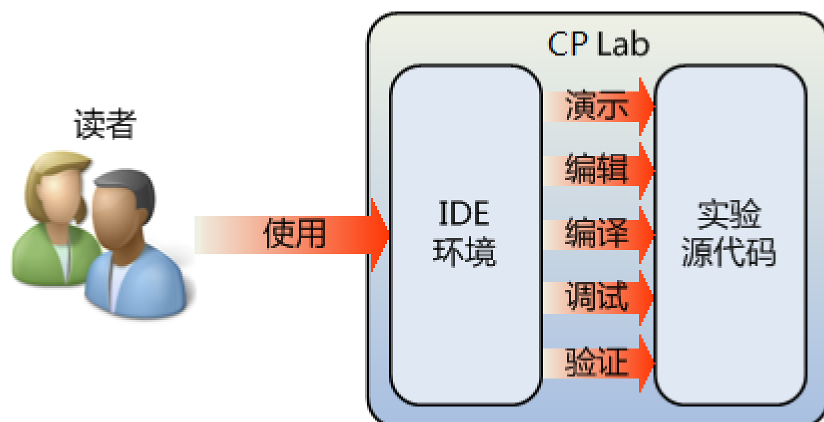

图 0-1：使用 CP Lab 进行编译原理实验

二、实验题目清单

下面的实验题目适合在学习编译原理课程的过程中逐个完成。

序号	概念	实验名称	难度	学时
1	词法分析	实验环境的使用（正则表达式到 NFA）	★★☆☆☆	2
2		NFA 到 DFA	★★★★☆	4
3		使用 Lex 自动生成扫描程序	★★☆☆☆	2
4	自顶向下的语法分析	消除左递归（无替换）	★★★☆☆	2
5		消除左递归（有替换）	★★★★☆	4
6		提取左因子	★★★☆☆	2
7		First 集合	★★☆☆☆	2
8		Follow 集合	★★☆☆☆	2
9	自底向上的语法分析	Yacc 分析程序生成器	★★☆☆☆	2
10	语义分析	符号表的构建与使用	★★☆☆☆	2
11	代码生成	三地址码转换为 P-代码	★★☆☆☆	2
12		GCC 编译器案例综合研究	★★☆☆☆	2

下面的实验题目适合在学完编译原理课程后，选择一个作为课程设计题目来完成。

序号	课程设计题目要求
1	编写一个程序，可以根据输入的正则表达式生成 NFA，然后将 NFA 转换为最小化的 DFA，最后使用得到的 DFA 完成字符串匹配。
2	参考 Lex 生成 TINY 语言扫描程序的过程，使用 Lex 为 C-Minus 语言生成一个扫描程序。
3	编写一个程序，可以为输入的 BNF 消除左递归或提取左因子，然后根据 BNF 计算出 First 集合和 Follow 集合，从而构造一个 LL(1)分析表，最终实现一个表驱动的 LL(1)分析算法。
4	参考 Yacc 生成 TINY 语言语法分析程序的过程，使用 Yacc 为 C-Minus 语言生成一个语法分析程序。
5	根据表达式的 BNF，使用 Yacc 输出表达式的 Modula-2 转换式。
6	编写一个程序，可将输入的三地址码转换为 P-代码，还可将输入的 P-代码转换为三地址码。

三、开始使用

读者迅速掌握 CP Lab 的基本使用方法，是顺利完成编译原理实验的重要前提。虽然 CP Lab 提供了非常人性化的操作界面，而且读者只需要掌握很少的几个核心功能，就可以顺利完成实验。但是，为了达到“做中学”的目的，在向读者介绍 CP Lab 的核心功能时，不会使用堆砌大量枯燥文字的方法，而是在后面的“实验 1”中，引导读者在使用 CP Lab 的过程中逐步掌握其基本使用方法，这样达到的效果最好。

实验 1 实验环境的使用

实验难度：★★☆☆☆
建议学时：2 学时

一、实验目的

- 熟悉编译原理集成实验环境 CP Lab 的基本使用方法。
- 掌握正则表达式和 NFA 的含义。
- 实现正则表达式到 NFA 的转换。

二、预备知识

- 在这个实验中 NFA 状态结构体使用了类似于二叉树的数据结构，还包括了单链表插入操作以及栈的一些基本操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。
- 实验中需要把正则表达式转换为 NFA，所以要对正则表达式和 NFA（非确定有穷自动机）有初步的理解。读者可以参考配套的《编译原理》教材，预习这一部分内容。

三、实验内容

请读者按照下面的步骤完成实验内容，同时，仔细体会 CP Lab 软件和 CodeCode.net 平台的基本使用方法。在本实验题目中，操作步骤会编写的尽量详细，并会对 CP Lab 软件和 CodeCode.net 平台的核心功能进行具体说明。但是，在后面的实验题目中会尽量省略这些内容，而将重点放在实验相关的源代码上。如有必要，读者可以回到本实验题目中，参考 CP Lab 软件和 CodeCode.net 平台的基本使用方法。

3.1 从 CodeCode.net 平台领取任务

CodeCode.net 平台是用于教师在线布置实验任务，并统一管理学生提交的作业的平台。如果没有使用到 CodeCode 平台，可以跳过此部分，直接从 3.2 节继续进行实验。

1. 读者通过浏览器访问https://www.codecode.net，可以打开 CodeCode.net平台的登录页面。
2. 在登录页面中，读者输入帐号和密码，点击“登录”按钮，可以登录 CodeCode.net 平台。
3. 登录成功后，在“课程”列表页面中，可以找到编译原理相关的实验课程。点击此课程的链接，可以进入该课程的详细信息页面。
4. 在课程的详细信息页面中，可以查看课程描述信息，该信息对于完成实验十分重要，建议读者认真阅读。点击左侧导航中的“任务”链接，可以打开任务列表页面。
5. 在任务列表中找到本次实验对应的任务，点击右侧的“领取任务”按钮，可以进入“领取任务”页面。
6. 在“领取任务”页面中，查看任务信息后，填写“新建项目名称”和“新建项目路径”，然后选择“项目所在的群组”，点击“领取任务”按钮后，可以创建个人项目用于完成本次实验，并自动跳转到该项目所在的页面。

提示：在“领取任务”页面中，“新建项目名称”和“新建项目路径”这两项的内容可以使用默认值，如果选择的群组中已经包含了名称或者路径相同的项目，就会导致领取任务失败，此时需要修改新建项目名称和新建项目路径的内容。

7. 在个人项目页面中，包括左侧的导航栏、项目信息、文件列表等，如图 1-1 所示。
8. 在个人项目页面的图 1-1 中，点击红色方框中的按钮，可以复制当前项目的 URL，在本实验后面的练习中会使用这个 URL 将此项目克隆到读者计算机的本地磁盘中，从而供读者进行修改。

提示：领取任务的本质是：教师在新建任务时填写了实验模板的 URL，然后读者在领取任务时，根据任务

中记录的 URL 来派生（Fork）出一个新的项目，供读者在其中进行修改。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：

1. 在编译原理实验课程中，新建一个实验任务。
2. 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 1 实验环境的使用”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab01 .git”。
3. 点击“新建任务”按钮，完成新建任务操作。

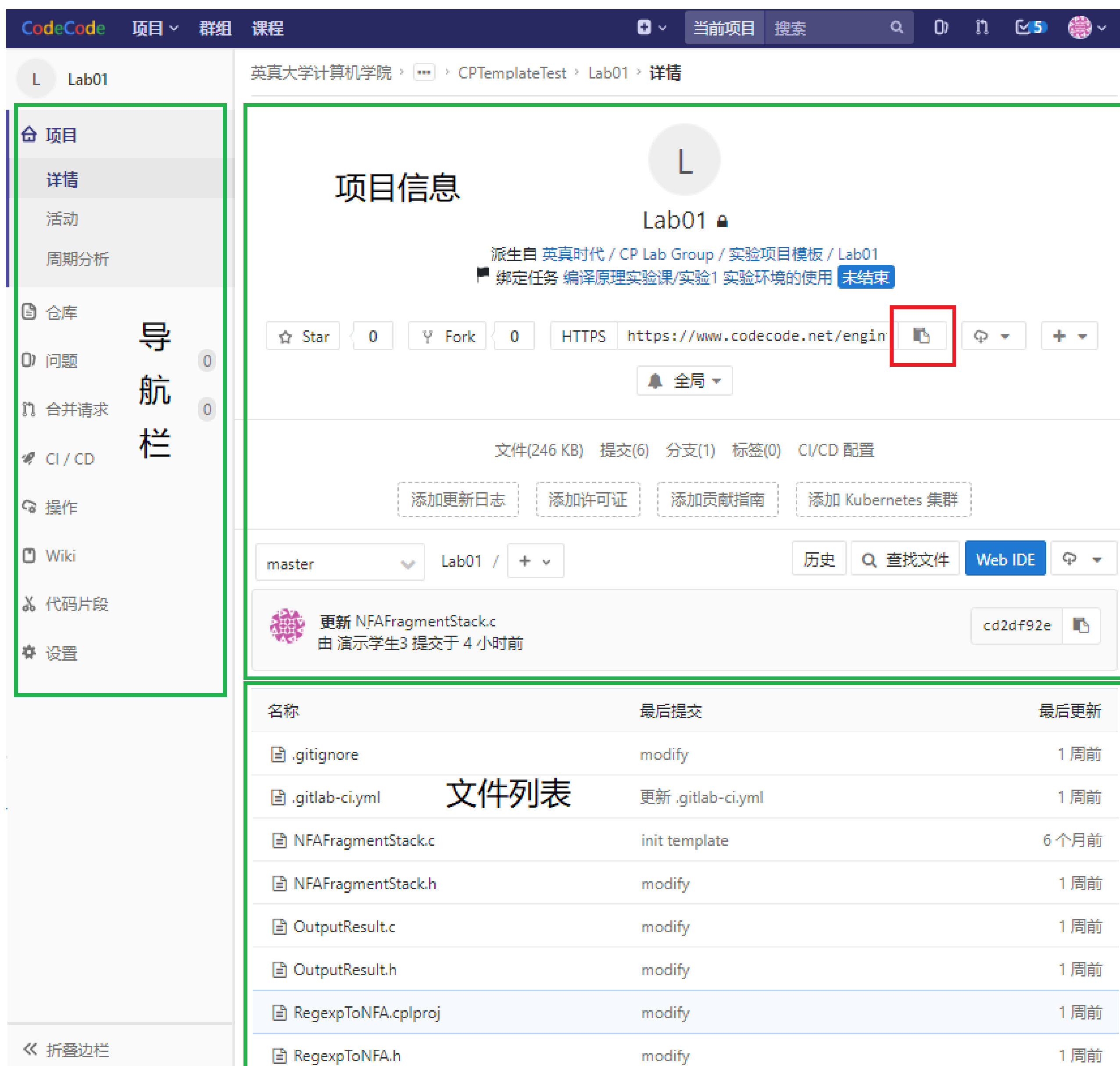

图 1-1：个人项目页面

3.2 启动 CP Lab

在安装有 CP Lab 的计算机上，可以使用两种不同的方法来启动 CP Lab：

- 在桌面上双击“Engintime CP Lab”图标。

  或者

- 点击“开始”菜单，在“程序”中的“Engintime CP Lab”中选择“Engintime CP Lab”。

启动 CP Lab 后会首先打开“登录”窗口，可以使用两种方法完成登录。

- 如果读者在CodeCode.net平台上注册了帐号，在连接互联网的情况下，可以使用CodeCode.net 平台中已注册的用户名和密码进行登录。
- 如果读者还没有 CodeCode.net 平台上的帐号，可以点击左下角“从加密锁获取授权”按钮，获取授权之后，完成登录。

3.3 主窗口布局

CP Lab 的主窗口布局由下面的若干元素组成：

- 顶部的菜单栏、工具栏。
- 停靠在左侧和底部的各种工具窗口。
- 余下的区域用来放置“起始页”和“源代码编辑器”窗口。

提示：菜单栏、工具栏和各种工具窗口的位置可以随意拖动。如果想恢复窗口的默认布局，选择“窗口”菜单中的“重置窗口布局”即可。

3.4 将 CodeCode.net 平台上的从正则表达式到 NFA 项目克隆到本地

如果读者从 CodeCode.net 平台领取了任务，可以在领取任务后将读者的个人项目克隆到本地磁盘中；如果读者没有 CodeCode.net 平台账号无法领取任务，可以直接将实验模板项目克隆到本地磁盘中。

将领取任务新建的个人项目克隆到本地

1. 选择 CP Lab 菜单“文件-》新建-》从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
2. 在“Git 远程库 URL(G)”中填入读者个人项目的 Git 远程库的 URL 地址（在图 1-1 中，点击红色方框中的按钮，可以复制当前项目的 URL）。
3. “项目文件夹名称”填入“lab1”。
4. “项目位置”选择新建项目需要保存到的磁盘目录。
5. 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地。在克隆过程中需要读者输入自己在 CodeCode.net 平台账号的密码。注意，在输入密码时是没有回显的。
6. 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

直接将实验模板项目克隆到本地

由于 CodeCode.net 平台上提供的实验模板是开放的，所有的人都可以访问。所以，CP Lab 可以直接将从正则表达式到 NFA 实验模板克隆到本地磁盘。步骤如下：

(1) 选择 CP Lab 菜单“文件-》新建-》从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。
(2) 在“Git远程库URL(G)”中填入从正则表达式到NFA实验模板Git远程库的URL地址：https://www.codecode.net/engintime/cp-lab/Project-Template/Lab01 .git
(3) “项目文件夹名称”填入“lab1”。
(4) “项目位置”选择新建项目需要保存到的磁盘目录。
(5) 点击“确定”按钮后会弹出一个 Windows 控制台窗口，并开始运行 Git 克隆命令将 Git 远程库克隆到本地，一定要等克隆成功后再关闭该窗口。
(6) 克隆项目成功后，在对话框中选择“克隆成功，打开新建项目”就可以打开新建的项目。

注意：

1. 为了能正常使用从 Git 远程库新建项目功能，用户需要在本地安装 Git 客户端程序，并且在安装 Git 的过程中会有一个步骤询问是否设置 Windows 环境变量，此时一定要设置上 Windows 环境变量。
2. 文件路径中不允许包含中文，请读者在使用之初就养成使用英文命名项目或文件的习惯。

新建完毕后， CP Lab 会自动打开这个新建的项目。在“项目管理器”窗口中（如图 1-2 所示），根节点是项目节点，各个子节点是项目包含的文件夹或者文件。读者也可以使用“Windows 资源管理器”打开磁盘上的“C:\cplab\lab1”文件夹，查看项目中包含的源代码文件。

提示：右键点击“项目管理器”窗口中的项目节点，选择快捷菜单中的“打开所在的文件夹”，即可使用“Windows 资源管理器”打开项目所在的文件夹。

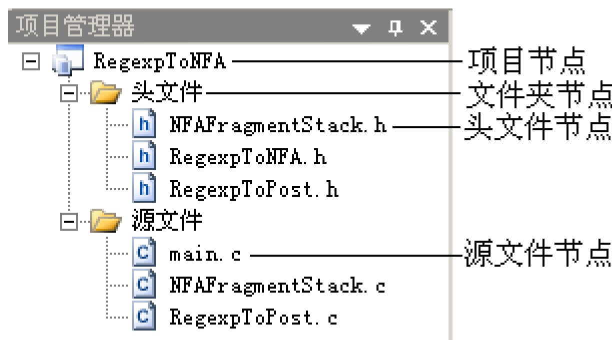

图 1-2：打开项目后的“项目管理器”窗口

3.5 阅读实验源代码

该实验包含了三个头文件“RegexpToNFA.h”、“RegexpToPost.h”、“NFAFragmentStack.h”和三个 C 源文件“main.c”、“RegexpToPost.c”、“NFAFragmentStack.c”。下面对这些文件的主要内容、结构和作用进行说明：

main.c 文件

在“项目管理器”窗口中双击“main.c”打开此文件。此文件主要包含了以下内容：

1. 在文件的开始位置，使用预处理命令包含了 RegexpToNFA.h、RegexpToPost.h 和 NFAFragmentStack.h 文件。
2. 定义了 main 函数。在其中实现了栈的初始化。然后，调用了 re2post 函数，将正则表达式转换到解析树的后序序列。最后，调用 post2nfa 函数，将解析树的后序序列转换到 NFA。
3. 在 main 函数的后面，定义了函数 CreateNFAState 和 MakeNFAFragment，这两个函数分别是用来创建一个新的 NFA 状态和构造一个新的 Fragment。接着定义了函数 post2nfa，关于此函数的功能、参数和返回值，可以参见其注释。在这个函数中用‘$’表示空转换，此函数的函数体还不完整，留给读者完成。

RegexpToPost.c 文件

在“项目管理器”窗口中双击“RegexpToPost.c”打开此文件。此文件主要包含了以下内容：

1. 在文件的开始位置，使用预处理命令包含了 RegexpToPost.h 文件。
2. 定义了 re2post 函数，此函数主要功能是将正则表达式转换成为解析树的后序序列形式。

NFAFragmentStack.c 文件

在“项目管理器”窗口中双击“NFAFragmentStack.c”打开此文件。此文件主要包含了以下内容：

1. 在文件的开始位置，使用预处理命令包含了 NFAFragmentStack.h 文件。
2. 定义了与栈相关的操作函数。在构造 NFA 的过程中，这个栈主要用来放置 NFA 片段。

RegexpToNFA.h 文件

在“项目管理器”窗口中双击“RegexpToNFA.h”打开此文件。此文件主要包含了以下内容：

1. 包含用到的 C 标准库头文件。目前只包含了标准输入输出头文件“stdio.h”。
2. 包含其他模块的头文件。目前没有其他模块的头文件需要被包含。
3. 定义了与 NFA 相关的数据结构，包括 NFA 状态 NFAState 和 NFA 片段 NFAFragment。具体内容可参见下面的两个表格。

NFAState 的域	说明
Transform	状态间转换的标识。用‘$’表示‘ε-转换’。
Next1 和 Next2	用于指向下一个状态。由于一个 NFA 状态可以存在多个转换，而在本程序中使用的是类似于二叉树的储存结构，每一个状态最多只有两个转换，所以，这里定义两个指针就足够了。当 NFA

	只有一个转换时，优先使用 Next1，Next2 赋值为 NULL。
Name	状态名称。使用整数表示(从 1 开始)，根据调用 CreateNFAState 函数的顺序依次增加。
AcceptFlag	是否为接受状态的标志。1 表示是接受状态 0 表示非接受状态。

NFAFragment 的域	说明
StartState	NFAFragment 的开始状态。
AcceptState	NFAFragment 的接受状态。在构造 NFA 的过程中总是在 NFA 的开始状态和接受状态上进行操作，所以用开始状态和接受状态表示一个 NFA 片段就足够了。

4. 声明函数和全局变量。

RegexpToPost.h 文件

在“项目管理器”窗口中双击“RegexpToPost.h”打开此文件。此文件主要包含了以下内容：

1. 包含其他模块的头文件。目前只包含了头文件“RegexpToNFA.h”。
2. 声明函数。为了使程序模块化，所以将 re2post 函数声明包含在一个头文件中再将此头文件包含到“main.c”中。

NFAFragmentStack.h 文件

在“项目管理器”窗口中双击“NFAFragmentStack.h”打开此文件。此文件主要包含了以下内容：

1. 包含其他模块的头文件。目前只包含了头文件“RegexpToNFA.h”。
2. 定义重要的数据结构。定义了与栈相关的数据结构。
3. 声明函数。声明了与栈相关的操作函数。

提示：请读者认真理解这部分内容，其他实验题目中的源代码文件也严格遵守这些约定，如无特殊情况将不再进行如此详细的说明。

3.6 生成项目

使用“生成项目”功能可以将程序的源代码文件编译为可执行的二进制文件，操作如下：

1. 在“生成”菜单中选择“生成项目”（快捷键是 F7）。

在项目生成过程中，“输出”窗口会实时显示生成的进度和结果。如果源代码中不包含语法错误，会在生成的最后阶段提示生成成功，如图 1-3 所示。

生成项目的过程，就是将项目所包含的每个 C 源代码文件（.c 文件）编译为一个对象文件（.o 文件），然后再将多个对象文件链接为一个目标文件（.exe 文件）的过程。以本实验为例，成功生成项目后，默认会在“C:\cplab\lab1\Debug″目录下生成 “main.o”文件“RegexpToPost.o”文件“NFAFragmentStack.o”文件和 “RegexpToNFA.exe”文件。

提示：读者可以通过修改项目名称的方法来修改生成的.exe 文件的名称。方法是在“项目管理器”窗口中右键点击项目节点，选择快捷菜单中的“重命名”。待项目名称修改后，需要再次生成项目才能得到新的.exe 文件。

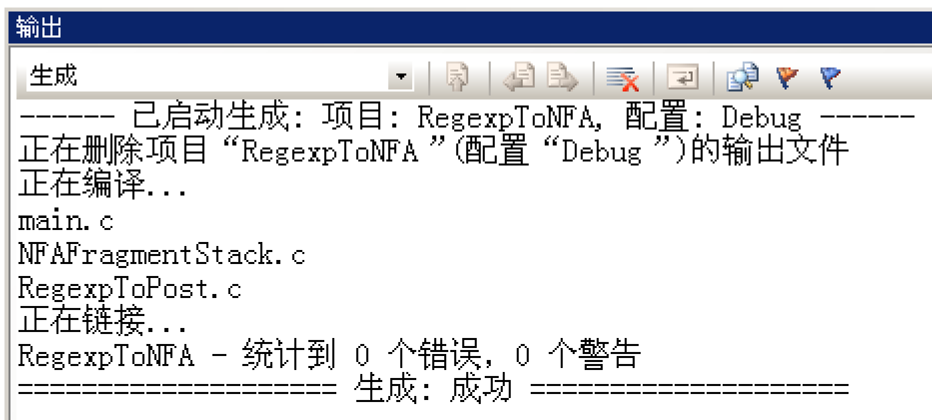

图 1-3：生成项目成功后的“输出”窗口

3.7 解决语法错误

如果在源代码中存在语法错误，在生成项目的过程中，“输出”窗口会显示相应的错误信息（包括错误所在文件的路径，错误在文件中的位置，以及错误原因），并在生成的最后阶段提示生成失败。此时，在“输出”窗口中双击错误信息所在的行，CP Lab 会使用源代码编辑器自动打开错误所在的文件，并定位到错误所在的代码行。

可以按照下面的步骤进行练习：

1. 在源代码文件中故意输入一些错误的代码（例如删除一个代码行结尾的分号）。
2. 生成项目。
3. 在“输出”窗口中双击错误信息来定位存在错误的代码行，并将代码修改正确。
4. 重复步骤 2、3，直到项目生成成功。

3.8 观察点和演示模式

这里介绍 CP Lab 提供的两个重要功能：**观察点**和**演示模式**。

观察点

一个观察点对应一个函数的起始位置和结束位置（称这个函数为观察点函数）。在调试过程中，当程序执行到观察点函数的起始位置和结束位置时就会发生中断，就好像在这两个位置上添加了断点一样。并且，只要在观察点函数内部发生中断（包括命中断点、单步调试等），就会在“转储信息”窗口中显示观察点函数正在操作的数据信息，如果在“演示模式”下，还会在“演示流程”窗口中显示观察点函数的流程信息。

以本实验为例，“观察点”窗口如图 1-4 所示（在“调试”菜单的“窗口”中选择“观察点”，可以打开“观察点”窗口），说明 post2nfa 函数是一个观察点函数。启动调试后，在 main.c 文件 post2nfa 函数的开始位置和结束位置的左侧空白处，会显示观察点图标（与“观察点”窗口中左侧的图标一致），当程序执行到 post2nfa 函数的开始位置和结束位置时会发生中断。启动调试后，观察点窗口如图 1-5 所示，可以显示出观察点所在的“文件”和“地址”。

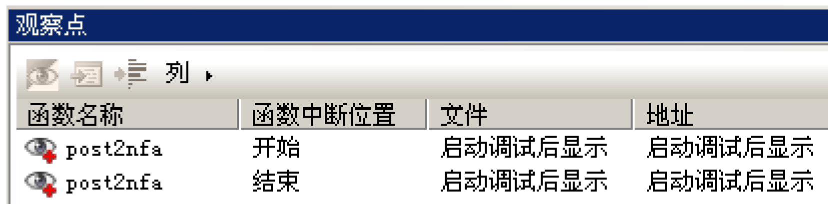

图 1-4：观察点窗口（未启动调试）

观察点

函数名称	函数中断位置	文件	地址
post2nfa	开始	main.c，行90	0x40184f
post2nfa	结束	main.c，行101	0x401863

图 1-5：观察点窗口（启动调试）

演示模式

当 CP Lab 工具栏上的“演示模式”按钮高亮显示时（如图 1-6 所示 ），CP Lab 处于演示模式。当在演示模式下调试观察点函数时，会忽略掉其函数体中的所有代码和断点，取而代之的是使用 CP Lab 提供的演示功能对观察点函数的执行过程和返回值进行演示。此特性可使观察点函数在还未完整实现的情况下，让读者了解到其应该具有的功能和执行过程，从而帮助读者正确实现此函数。

当工具栏上的“演示模式”按钮没有高亮显示时（鼠标点击工具栏上的“演示模式”按钮可以使其切换状态），CP Lab 处于非演示模式。在非演示模式下调试观察点函数时，会使用其函数体中的代码和断点。

图 1-6：工具栏上的“演示模式”按钮。

3.9 在演示模式下调试项目

读者可以按照下面的步骤，练习在演示模式下调试项目（主要是调试观察点函数）：

1. 保证工具栏上的“演示模式”按钮高亮显示。
2. 在“调试”菜单中选择“启动调试”（快捷键是 F5）。

启动调试后，程序会在观察点函数的开始位置处中断，如图 1-7 所示。源代码编辑器左侧空白处显示了相应的图标，分别标识了观察点函数的起始位置和结束位置，以及下一行要执行的代码（黄色箭头）。

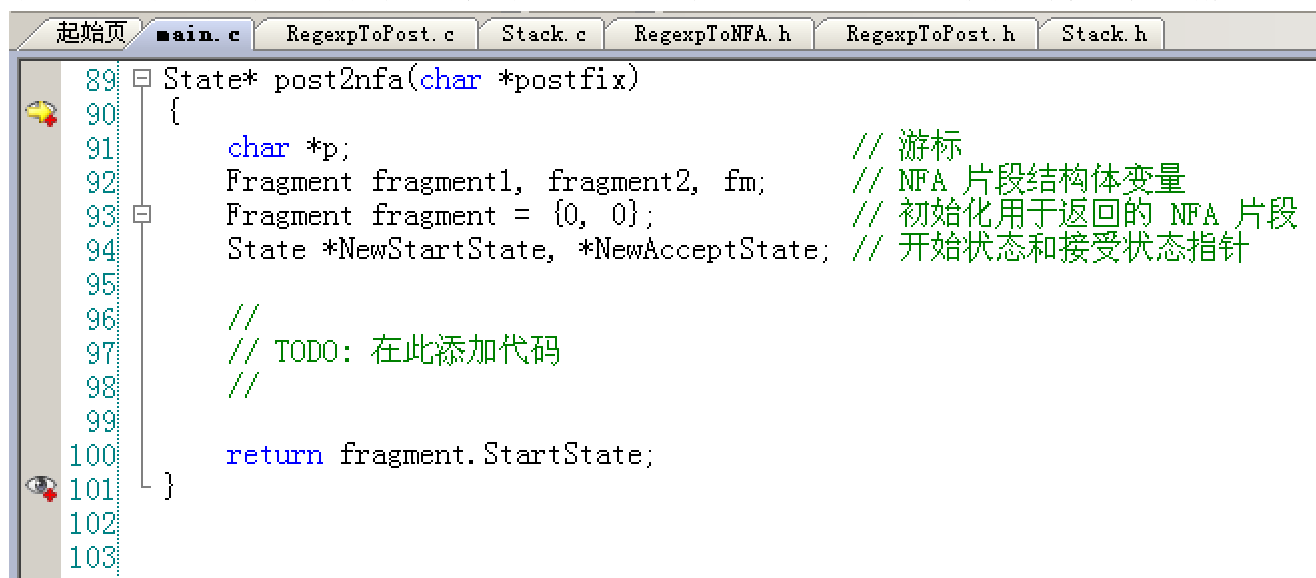

图 1-7：启动调试后，在观察点函数的开始位置中断。

同时，在“转储信息”窗口中（可以选择“调试”菜单“窗口”中的“转储信息”打开此窗口）显示了观察点函数正在操作的数据信息，如图 1-8 所示。主要包含了如下内容：

1. 函数调用信息。对本次观察点函数的调用信息进行了描述。

2. 函数返回信息。由于此时刚刚进入观察点函数，所以还无法显示其返回信息。当在观察点函数结束位置中断时，即可显示其返回信息。主要对观察点函数的返回值或者操作结果进行描述。
3. 重要的数据信息。包含了正则表达式和解析树的后序序列的描述，以及对栈中保存的 NFA 片段的信息进行了描述，包括：状态名称（用数字表示从 1 开始）转换标志以及转换到的状态。当程序运行到观察点结尾的位置时还会显示返回结果用以验证项目（验证项目会在后面详细描述）

```
转储信息

post2nfa 函数的调用信息：正则表达式到 NFA 。
post2nfa 函数的返回信息：还未返回

数据信息：

正则表达式(regexp)：
    ab

解析树的后序遍历序列(postfix)：
    ab.
    |p

NFA 片段栈(FragmentStack)：
```

图 1-8：在观察点函数开始位置中断时的“转储信息”窗口。

按照下面的步骤继续调试：

1. 在“调试”菜单中选择“继续”（快捷键是 F5）。

由于是在“演示模式”下调试观察点函数，CP Lab 会忽略掉函数体中的所有代码，取而代之的是使用 CP Lab 提供的演示功能对观察点函数的执行过程进行演示。所以 CP Lab 会自动打开“演示流程”窗口（可以选择“调试”菜单“窗口”中的“演示流程”打开此窗口），在其中显示观察点函数的演示流程，如图 1-9 所示。

观察点函数的演示流程通常采用简洁、直观的语言进行描述（一行描述可能会对应多行 C 源代码），偶尔也会在读者理解起来比较困难的地方提供 C 源代码的提示或者直接使用 C 源代码，目的就是为了方便读者将演示流程快速转换为 C 源代码。在“演示流程”窗口左侧的空白处，同样使用黄色箭头标识出了下一行要执行的代码（流程）。

按照下面的步骤继续调试：

2. 在“调试”菜单中重复选择“继续”，直到在观察点函数的结束位置中断。CP Lab 会单步执行“演示流程”窗口中的每一行（包括循环）。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，例如构造单字符 NFA 片段，构造连接 NFA 片段等，理解 NFA 片段构造的执行过程。当在观察点函数的结束位置中断时，“转储信息”窗口中将显示函数的返回信息。

按照下面的步骤结束此次调试：

1. 在“调试”菜单中重复选择“继续”，直到调试结束。或者，在“调试”菜单中选择“停止调试”。

读者可以在演示模式下重新启动调试，再次执行以上的步骤，仔细体会在“演示模式”下调试观察点函数的过程。

起始页 | main.c | 演示流程

```
State* post2nfa(char* postfix)
{
    char *p;                                    // 游标
    Fragment fragment1, fragment2, fm;          // NFA 片段结构体变量
    Fragment fragment = {0, 0};                 // 初始化用于返回的 NFA 片段
    State *NewStartState, *NewAcceptState;      // 开始状态和接受状态
    for(p=postfix; *p!=NULL; p++)
    {
        switch(*p)
        {
            default:    // 构造单字符 NFA 片段
                // 调用 CreateState 函数生成两个新的状态
                NewStartState = CreateState();
                NewAcceptState = CreateState();
                // 开始状态通过单字符转换到接受状态
                NewStartState->Transform = *p;
                NewStartState->Next1 = NewAcceptState;
                NewAcceptState->AcceptFlag = 1;
                // 调用 MakeFragment 函数生成一个新的 NFA 片段, 并入栈
                fm = MakeFragment(NewStartState, NewAcceptState);
                Push(&FragmentStack, fm);
                break
            case '.':   // 构造连接 NFA 片段
                // 栈顶的两个片段出栈，构造新的 NFA 片段
                fragment2 = Pop(&FragmentStack);
```

图 1-9：“演示流程”窗口。

3.10 验证项目（失败）

这里介绍 CP Lab 提供的另外一个重要功能：**验证功能**。

之前提到了 main.c 文件中的 post2nfa 函数还不完整，是留给读者完成的。而当读者完成此函数后，往往需要使用调试功能、或者执行功能，来判断所完成的函数是否能够达到预期的效果，即是否与演示时函数的执行过程和返回值完全一致。CP Lab 提供的验证功能可以自动化的、精确的完成这个验证过程。

验证功能分为下面三个阶段：

1. 在“演示模式”下执行观察点函数（与工具栏上的“演示模式”按钮是否高亮无关），将产生的转储信息自动保存在文本文件 ValidateSource.txt 中。
2. 在“非演示模式”下执行观察点函数，将产生的转储信息自动保存在文本文件 ValidateTarget.txt 中。
3. 自动使用 CP Lab 提供的文本文件比较工具来比较这两个文件。当这两个文件中的转储信息完全一致时，报告“验证成功”；否则，报告“验证失败”。

当读者完成的函数与演示时函数的执行过程和返回值完全一致时，就会产生完全一致的转储信息，验证功能就会报告“验证成功”；否则，验证功能就会报告“验证失败”，并且允许读者使用 CP Lab 提供的文本文件比较工具，来查看这两个转储信息文件中的不同之处，从而帮助读者迅速、准确的找到验证失败的原因，进而继续修改源代码，直到验证成功。

按照下面的步骤启动验证功能：

1. 在“调试”菜单中选择“开始验证”（快捷键是 Alt+F5）。在验证过程中，“输出”窗口会实时显示验证各个阶段的执行过程（如清单 1-1 所示），包括转储信息文件的路径、观察点函数的调用信息和返回信息、以及验证结果。由于 post2nfa 函数还不完整，所以验证失败。
2. 使用“输出”窗口工具条上的“比较”按钮（如图 1-10 所示）查看两个转储信息文件中的内容，即它们之间的不同之处。

提示：

- 在转储信息文件中，只保存了观察点函数开始位置和结束位置的转储信息，并使用单线进行分隔。观察点函数多次被调用的转储信息之间，使用双线进行分隔。
- 在转储信息文件中，为了确保验证功能的准确性，某些信息会被忽略掉（不再显示或使用“N/A”替代），例如内存中的随机值等。

```
------ 已启动验证: 项目: RegexpToNFA, 配置: Debug ------

验证第一阶段：正在使用“演示模式”生成转储信息，并写入源文件...
源文件路径：G:\Documents and Settings\Engintime\My Documents\CP Lab\Projects\lab1\Debug\ValidateSource.txt

post2nfa 函数的调用信息：正则表达式到 NFA 。
post2nfa 函数的返回信息：返回 NFA 的开始状态地址。

============================================================================

验证第二阶段：正在使用“非演示模式”生成转储信息，并写入目标文件...
目标文件路径：G:\Documents and Settings\Engintime\My Documents\CP Lab\Projects\lab1\Debug\ValidateTarget.txt

post2nfa 函数的调用信息：正则表达式到 NFA 。
post2nfa 函数的返回信息：返回 NFA 的开始状态地址。

============================================================================

验证第三阶段：正在比较转储信息源文件与目标文件的内容...
比较结果：转储信息源文件与目标文件的内容不同

==================== 验证结果：失败 ====================
```

清单 1-1：在“输出”窗口中显示的验证信息。

图 1-10：“输出”窗口工具栏上的“比较”按钮。

3.11 实现 post2nfa 函数

文件 main.c 中的 post2nfa 的函数体还不完整，需要读者补充完整。

提示：

- 在“观察点”窗口中，可以在函数名称上点击右键，选择快捷菜单中的“查看演示流程”，CP Lab 会打开“演示流程”窗口，并显示观察点函数的演示流程。这样，即使在没有启动调试的情况下，读者也可以方便的查看观察点函数的演示流程。
- 参考演示流程中构造 NFA 片段的描述，其中给出了构造单字符 NFA 片段和连接 NFA 片段的源代码，这两步操作可以完成对例 1 正则表达式到 NFA 的转换（转换后的 NFA 如图 1-13 所示），读者可以

在“演示模式”下一边调试一边理解源代码的执行过程，在此基础上完成其他形式 NFA 片段的构造。

- 对于问号的 NFA 片段的构造（如图 1-16 所示），原则上也可以将状态 1 作为开始状态，状态 2 作为接受状态，并通过 ε -转换来表达接受状态为空的情况。但是如果存在一个或多个 NFA 片段与问号 NFA 片段相连接的情况，对于上述的情况就不适用了。因为在本程序中使用的是类似于二叉树的存储结构，一个状态最多只有两个转换指针，所以，必须在原来的基础上再添加两个状态作为开始状态和接受状态。

图 1-11：NFA 状态图例。

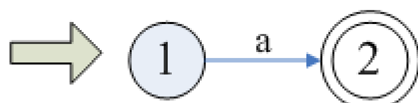

图 1-12：表示单字符的 NFA 片段。

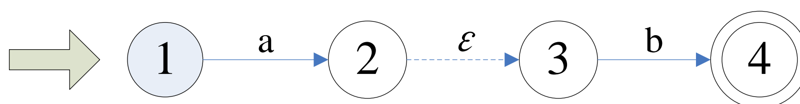

图 1-13：表示连接的 NFA 片段（对应例 1）。

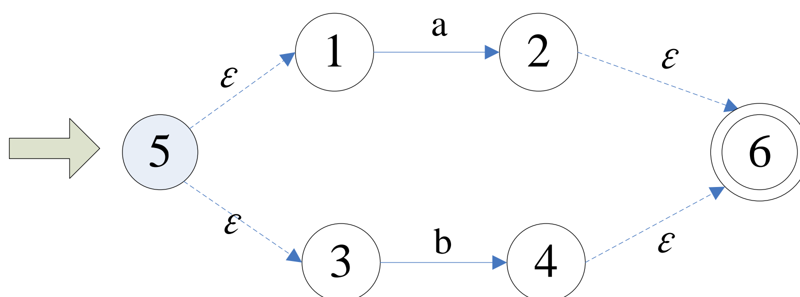

图 1-14：表示选择的 NFA 片段（对应例 2）。

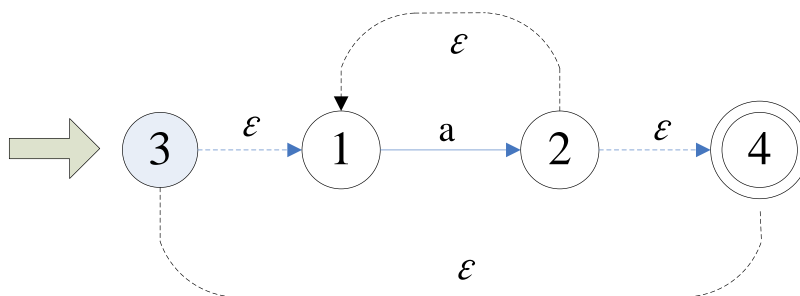

图 1-15：表示星号的 NFA 片段（对应例 3）。

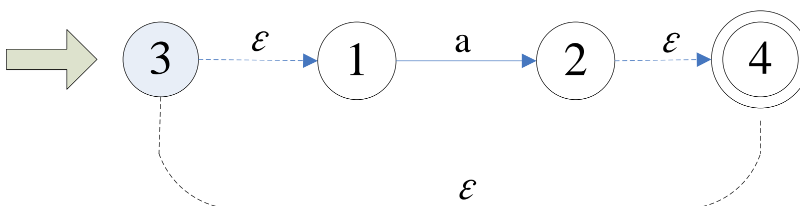

图 1-16：表示问号的 NFA 片段（对应例 4）。

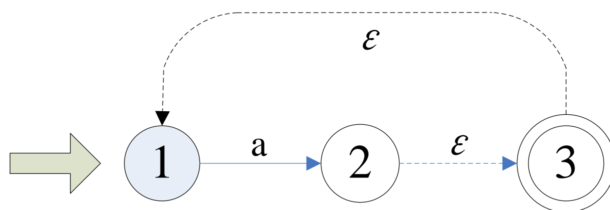

图 1-17：表示加号的 NFA 片段（对应例 5）

3.12 在非演示模式下调试项目

读者在实现了 post2nfa 函数后，可以按照下面的步骤，练习在非演示模式下调试项目（主要是调试由读者实现的观察点函数）：

1. 在“生成”菜单中选择“生成项目”。如果读者编写的源代码中存在语法错误，修改这些错误，直到可以成功生成项目。
2. 鼠标点击工具栏上的“演示模式”按钮，使其切换到非高亮显示状态。
3. 在“调试”菜单中选择“启动调试”。程序会在观察点函数的开始位置处中断。
4. 在“调试”菜单中重复选择“逐过程”（快捷键是 F10），直到在观察点函数的结束位置中断。CP Lab 会单步执行观察点函数中的每一行源代码。在调试的过程中，每执行一行源代码后，仔细观察“转储信息”窗口内容所发生的变化，例如游标的移动、构造单字符的 NFA 片段等，理解 NFA 片段的构造过程。当在观察点函数的结束位置中断时，“转储信息”窗口中将显示函数的返回信息。

以上的练习说明，CP Lab 可以让读者在非演示模式下调试项目，并观察“转储信息”窗口内容所发生的变化，从而理解每一行源代码对内存数据的操作结果。如果读者发现所编写的源代码存在异常行为（例如死循环、数组越界访问或者验证失败），可以在非演示模式下单步调试项目，来查找异常产生的原因。

3.13 验证项目（成功）

按照下面的步骤启动验证功能：

1. 在“调试”菜单中选择“开始验证”。在“输出”窗口中，如果提示验证成功，说明代码编写的没有问题；如果提示验证失败，读者可以参考之前的内容来查找原因并修改源代码中的错误，直到验证成功。

提示：

1. 由于本实验默认情况下只验证了算例 1（即例 1 中的表达式），提交作业后，算例 1 可得到 50 分，为了得到更高的分数，可以验证剩余的算例。
2. 验证剩余算例的方法是，将 main.c 文件的开头部分例 1 中的正则表达式注释，将例 2 中的正则表达式的注释打开，这样就可以验证算例 2。当验证的算例通过越多，得分就会越高。如果本实验包含的 8 个算例都通过验证，就可以得到满分。

3.14 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的个人项目，并将个人项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

提交作业的步骤：

1. 在项目管理器窗口中，右键点击项目节点，在弹出的菜单中，选择“Git”中的“推送当前分支到 Git 远程库”菜单项，弹出“推送当前分支到 Git 远程库”的对话框。
2. 在“推送当前分支到 Git 远程库”对话框中，输入本次项目更改的提示信息。
3. 点击“推送”按钮，可以将当前项目推送到 CodeCode.net 平台的个人项目中。

可以按照下面的步骤查看推送后的项目：

1. 使用浏览器访问https://www.codecode.net，使用已注册的帐号登录CodeCode.net平台。
2. 在“课程”列表中选择编译原理实验课程，打开编译原理实验课程的详细信息页面，点击左侧导航栏的“任务”链接，打开任务列表页面。
3. 在任务列表页面找到本次实验对应的任务，点击右侧的项目图标，可以跳转到读者的个人项目页面。
4. 在个人项目页面中，可以查看当前项目的全部内容。
5. 在左侧的导航栏中点击“仓库”中的“提交”链接，可以打开提交列表页面。
6. 在提交列表页面中，点击最后一次提交，可以查看此次提交发生变更的文件。

技巧：在 CP Lab 的项目管理器窗口中，选中项目节点，点击鼠标右键，在弹出的右键菜单中，选择“Git”中的“使用浏览器访问 Git Origin”菜单项，可以自动打开本地项目在 CodeCode.net 平台上对应的个人项目。

3.15 线上查看流水线的运行结果

读者提交项目之后，使用浏览器访问 CodeCode.net 平台，打开已提交项目所在的页面，可以查看项目的完成结果。

查看流水线运行结果的步骤：

1. 在项目的左侧导航栏中，单击“CI/CD”中的“流水线”，可以查看流水线的运行结果，并显示出流水线的得分情况，如图 1-18 所示。
2. 手动 CI 是由教师执行的，学生没有执行手动 CI 的权限，教师执行完之后，可以查看流水线的最终运行结果。
3. 在流水线列表中，点击最近一次流水线的状态的链接，进入流水线的详细信息页面。
4. 在流水线的详细信息页面中，可以查看流水线、作业、代码质量的详细信息。

提示：本实验中包含 8 个测试算例，流水线中 case1 对应测试算例 1， case2 对应测试算例 2，以此类推。

图 1-18：流水线

3.16 线上查看代码完成的质量

读者提交项目之后，CI 已经执行完，就可以查看已提交项目的代码质量。

查看已提交代码质量的步骤：

1. 打开已提交的项目，在左侧的导航栏中选择“仓库”->“提交”，可以打开“提交”列表。
2. 在“提交”列表中，点击最近一次提交的链接，在新打开的页面中，显示了代码质量的分析结果，如图 1-19 所示。
3. 例如：点击 info 中的“keyword ’if’ not followed by a single space”链接，可以在该项目的源代码中定位存在该代码质量问题的位置。
4. 点击链接后面的图标，可以查看修改该代码质量问题的方法。

小助同学 @xiaozhu commented · 1 天前 Developer

代码质量分析发现了 69 个问题。

- 0 blocker
- 0 critical
- 0 major
- 2 minor
- 67 info

注意: 存在下列问题的代码行在本次提交中没有发生变更，无法使用代码行评论的方式进行报告。所以将下列问题汇总显示在这里（点击问题链接可以转到对应的源代码行）：

1. Cppcheck cannot find all the include files. Cppcheck can check the code without the include files found. But the results will probably be more accurate if all the include files are found. Please check your project's include directories and add all of them as include directories for Cppcheck. To see what files Cppcheck cannot find use --check-config. (ProjectKey-895)
2. scanf without field width limits can crash with huge input data. Add a field width specifier to fix this problem: %s => %20s Sample program that can crash: #include int main() { char c[5]; scanf("%s", c); return 0; } Typing in 5 or more characters may make the program crash. The correct usage here is 'scanf("%4s", c);', as the maximum field width does not include the terminating null byte. Source: http://linux.die.net/man/3/scanf Source: http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/libkern/stdio/scanf.c
3. keyword 'if' not followed by a single space
4. full block {} expected in the control structure
5. keyword 'if' not followed by a single space
6. full block {} expected in the control structure
7. too many consecutive empty lines

图 1-19：代码质量分析

3.17 总结

读者使用 CP Lab 进行编译原理实验的步骤可以总结如下：

1. 启动 CP Lab。
2. 使用在 CodeCode 上已注册的用户进行登录。
3. 从 Git 远程库新建实验项目。
4. 在演示模式下调试项目，理解观察点函数的执行过程（通常观察点函数还未完整实现）。
5. 结合观察点函数的演示流程，修改观察点函数的源代码，实现其功能。
6. 生成项目（排除所有的语法错误）。
7. 验证观察点函数。如果验证失败，可以使用“比较”功能，或者在非演示模式下调试项目，从而定位错误的位置，然后回到步骤 6。
8. 提交作业。
9. 退出 CP Lab。

3.18 获得帮助

如果读者在使用 CP Lab 的过程中遇到问题需要专业的解答，或者有一些心得体会想和其他 CP Lab 用户分享，欢迎加入 CodeCode：

- 选择 CP Lab“帮助”菜单中的“提交问题或建议”。

  或者
- 直接访问https://www.codecode.net/engintime/cp-lab/cp-lab/issues

这里列出了读者在使用 CP Lab 的过程中可能遇到的一些问题和使用技巧，用于帮助读者更好的使用 CP Lab，获得最佳的实验效果。

1. 读者时常会遇到在自己编写的源代码中存在死循环的情况，这就会造成 CP Lab 的调试功能，特别是验证功能无法自行结束。此时，读者可以选择“调试”菜单中的“停止调试”（快捷键是 Shift+F5）来强制结束这些功能。随后，读者可以检查自己编写的源代码，或者在“非演示模式”下单步调试项目，从而找到造成死循环的原因。

2. 读者时常会遇到的另外一个情况是“数组越界访问”。此时，CP Lab 会弹出一个调试异常对话框，读者只要选择对话框中的“是”按钮，就可以立即定位到异常所在的代码行。
3. CP Lab 作为一个 IDE 环境，提供了强大的调试功能，包括单步调试、添加断点、查看变量的值、查看调用堆栈等。读者在调试过程中可以灵活使用这些功能，提高调试效率。注意，在“演示模式”下，观察点函数中的断点会被忽略。
4. 在“演示模式”下，对观察点函数只能进行单步调试（无论是按快捷键 F5，还是 F10），如果观察点函数中存在多次循环，会造成调试过程比较缓慢。此时，读者可以选择“调试”菜单中的“结束观察”（快捷键是 Shift+Alt+F5），直接跳转到观察点函数的结束位置中断。
5. “输出”窗口、“演示流程”窗口、以及“转储信息”窗口中的文本信息可以被选中并复制（但是不能修改），读者可以很方便的将这些信息保存下来，用于完成实验报告等工作。
6. CP Lab 提供的实验项目通常不会在 Windows 控制台窗口中打印输出任何信息，因为在“转储信息”窗口、“输出”窗口中已经为读者提供了足够多的信息。当然，读者也可以根据自己的喜好，使用 printf 函数，在 Windows 控制台窗口中打印输出一些信息（这些信息不会对“验证”结果产生任何影响）。如果读者想快速查看程序在 Windows 控制台窗口中打印输出的信息，可以使用“调试”菜单中的“开始执行”功能（快捷键是 Ctrl+F5）。

四、思考与练习

1. 编写一个 FreeNFA 函数，当在 main 函数的最后调用此函数时，可以将整个 NFA 的内存释放掉，从而避免内存泄露。
2. 读者编写完代码之后可以对 main.c 中 main 函数之前的例 2 到例 5 进行一一验证，确保程序可以将所有形式的正则表达式转换为正确的 NFA，并验证通过。
3. 对例 6、例 7、例 8 的正则表达式进行验证，并画出例 7 和例 8 的 NFA 状态图。
4. 详细阅读 re2post 函数中的源代码，并尝试在源代码中添加注释。然后尝试为本实验中所有的例子绘制解析树（类似二叉树）。

实验 2 NFA 到 DFA

实验难度：★★★★☆
建议学时：4 学时

一、 实验目的

- 掌握 NFA 和 DFA 的概念。
- 掌握ε-闭包的求法和子集的构造方法。
- 实现 NFA 到 DFA 的转换。

二、 预备知识

- 完成从正则表达式到 NFA 的转换过程是完成本实验的先决条件。虽然 DFA 和 NFA 都是典型的有向图，但是基于 NFA 自身的特点，在之前使用了类似二叉树的数据结构来存储 NFA，达到了简化的目的。但是，DFA 的结构相对复杂，所以在这个实验中使用了图的邻接链表来表示 DFA。如果读者对有向图的概念和邻接链表表示法有一些遗忘，可以复习一下数据结构中相关的章节。
- 对 DFA 的含义有初步的理解，了解ε-闭包的求法和子集的构造方法。读者可以参考配套的《编译原理》教材，预习这一部分内容。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为
https://www.codecode.net/engintime/cp-lab/Project-Template/Lab02.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 2 NFA 到 DFA”；还需要填写“模板项目URL”，应该使用
“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab02.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

NFAToDFA.h 文件

主要定义了与 NFA 和 DFA 相关的数据结构，其中有关 NFA 的数据结构在前一个实验中有详细说明，所以这里主要说明一下有关 DFA 的三个数据结构，这些数据结构定义了 DFA 的邻接链表，其中 DFAState 结构体用于定义有向图中的顶点（即 DFA 状态），Transform 结构体用于定义有向图中的弧（即转换）。具体内容可参见下面的表格。

DFA 的域	说明
DFAlist	DFA 状态集合。
length	集合中的 DFA 状态数量。与前一项构成一个线性表。

DFAState 的域	说明
NFAlist	NFA 状态集合。用于保存 DFA 状态中的 NFA 状态集合。
NFAStateCount	集合中的 NFA 状态数量。与前一项构成一个线性表。
firstTran	指向第一个转换。

Transform 的域	说明
TransformChar	状态之间的转换符号。
DFAStateIndex	DFA 状态在线性表中的下标。用于指示转换的目标 DFA 状态。
NFAlist	NFA 状态集合。用于保存构造的子集以及生成新的 DFA 状态。
NFAStateCount	集合中的 NFA 状态数量。与前一项构成一个线性表。
NextTrans	指向下一个转换。

main.c 文件

定义了 main 函数。在 main 函数中首先初始化了栈，然后调用了 re2post 函数，将正则表达式转换到解析树的后序序列，最后调用了 post2dfa 函数将解析树的后序序列转换到 DFA。

在 main 函数的后面，定义了一系列函数，有关函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
CreateDFATransform	创建一个新的 DFA 转换。每当求得一个ε-闭包后可以调用此函数来创建一个 DFA 转换。
CreateDFAState	利用转换作为参数构造一个新的 DFA 状态，同时将 NFA 状态子集复制到新创建的 DFA 状态中。
NFAStateIsSubset	当一个子集构造完成后，需要调用此函数来判断是否需要为该子集创建一个新的 DFA 状态。如果构造的子集是某一个 DFA 状态中 NFA 状态集合的子集，就不需要新建 DFA 状态了。此函数的函数体还不完整，留给读者完成。
IsTransformExist	判断 DFA 状态的转换链表中是否已经存在一个字符的转换。每当求得一个ε-闭包后可以调用此函数来决定是新建一个转换，还是将ε-闭包合并到已有的转换中。此函数的函数体还不完整，留给读者完成。
AddNFAStateArrayToTransform	将一个 NFA 集合合并到一个 DFA 转换的 NFA 集合中，并确保重复的 NFA 状态只出现一次。当需要将ε-闭包合并到已有的转换中的 NFA 集合中时，可以调用此函数。此函数的函数体还不完整，留给读者完成。
Closure	使用二叉树的先序遍历算法求一个 NFA 状态的ε-闭包。注意，优先使用二叉树的先序遍历算法,否则会造成ε-闭包中 NFA 状态集合顺序不同，进而导致无法通过自动化验证。此函数的函数体还不完整，留给读者完成。
post2dfa	将解析树的后序序列转换为 DFA。在这个函数中会调用函数 post2nfa，所以在此函数中只需要专注于 NFA 转换到 DFA 的源代码的编写即可。此函数的函数体还不完整，留给读者完成。

RegexpToPost.c 文件

定义了 re2post 函数，此函数主要功能是将正则表达式转换成为解析树的后序序列形式。

PostToNFA.c 文件

定义了 post2nfa 函数，此函数主要功能是将解析树的后序序列形式转换成为 NFA。关于此函数的功能、参数和返回值，可以参见其注释。注意，此函数的函数体还不完整，读者可以直接使用之前实验中编写的代码。

NFAFragmentStack.c 文件

定义了与栈相关的操作函数。注意，这个栈是用来保存 NFA 片段的。

NFAStateStack.c 文件

定义了与栈相关的操作函数。注意，这个栈是用来保存 NFA 状态的。

RegexpToPost.h 文件

声明了相关的操作函数。为了使程序模块化，所以将 re2post 函数声明包含在一个头文件中再将此头文件包含到“main.c”中。

PostToNFA.h 文件

声明了相关的操作函数。为了使程序模块化，所以将 post2nfa 函数声明包含在一个头文件中再将此头文件包含到“main.c”中

NFAFragmentStack.h 文件

定义了与栈相关的数据结构并声明了相关的操作函数。

NFAStateStack.h 文件

定义了与栈相关的数据结构并声明了相关的操作函数。

3.3 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解 NFA 到 DFA 的转换过程。正则表达式 a(a|1)* 对应的 NFA 状态图可以参见图 2-1，DFA 状态图可以参见图 2-2。“转储信息”窗口显示的数据信息包括：

- 正则表达式。
- 解析树的后序序列。
- NFA 向 DFA 转换过程中构造的 ε-闭包。包括元素个数和闭包中的元素。
- DFA 邻接表。包括 DFA 状态在线性表中的下标，DFA 状态中的 NFA 状态集合以及 DFA 状态的转换链表。在调试的最后，DFA 的接受状态会包含在中括号中。
- 调用 post2nfa 函数返回的 NFA。详细的内容可以参看之前实验中的相关说明。

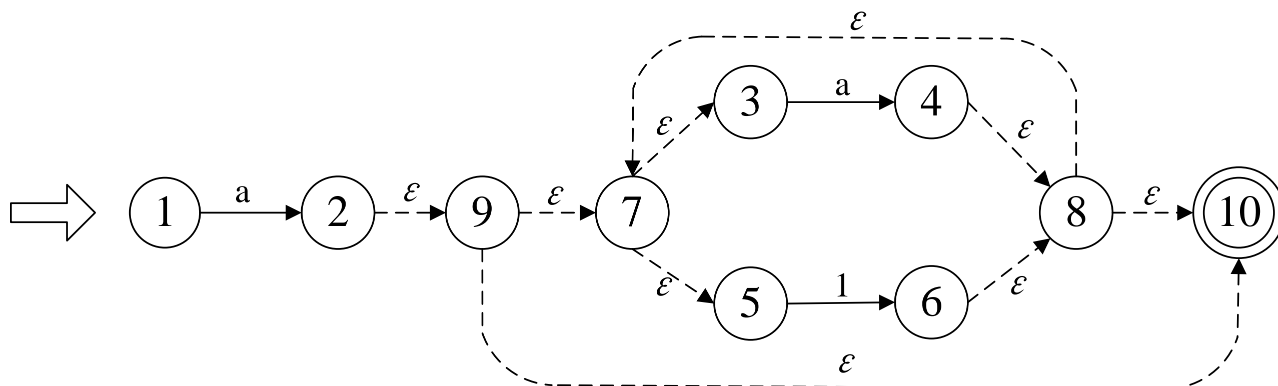

图 2-1：调用 post2nfa 函数后返回的 NFA

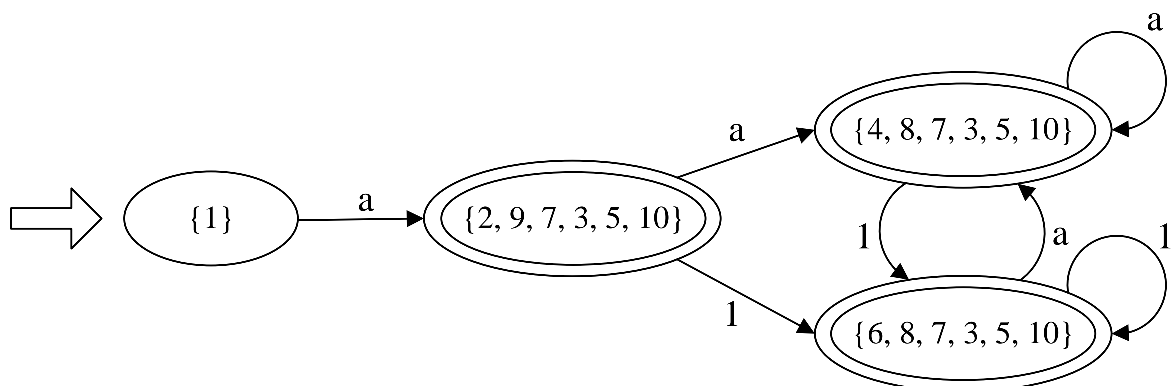

图 2-2：NFA 对应的 DFA

3.4 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 post2dfa 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

注意：在实现 Closure 函数时，尽量使用二叉树的先序遍历算法，否则会造成 ε-闭包中 NFA 状态集合顺序不同，进而导致无法通过自动化验证。

四、思考与练习

1. 编写一个 FreeNFA 函数和一个 FreeDFA 函数，当在 main 函数的最后调用这两个函数时，可以将整个 NFA 和 DFA 的内存分别释放掉，从而避免内存泄露。
2. 读者可以尝试使用自己编写的代码将 main 函数之前的例 2 和例 3 转换成 DFA，并确保能够通过自动化验证。在验证通过后，根据 DFA 的邻接链表数据绘制出 DFA 的状态转换图，并尝试编写一个 Minimize 函数，此函数可以将 DFA 中的状态数最小化。
3. 编写一个 Match 函数，此函数可以将一个字符串与正则表达式转换的 DFA 进行匹配，如果匹配成功返回 1，否则返回 0。

实验 3 使用 Lex 自动生成扫描程序

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 掌握 Lex 输入文件的格式。
- 掌握使用 Lex 自动生成扫描程序的方法。

二、 预备知识

- 要求已经学习了正则表达式的编写方法，能够正确使用“*”、“?”、“+”等基本的元字符，并且学习了 Lex 程序中定义的特有的元字符，例如“[ ]”、“\n”等。
- 了解了标识符和关键字的识别方法。
- 本实验使用 Lex 的一个实现版本——GNU Flex 作为扫描程序。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab03.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 3 使用Lex自动生成扫描程序”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab03.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

sample.txt 文件

在 sample.txt 文件中是一个使用 TINY 语言编写的小程序，这个小程序在执行时从标准输入（键盘）读取一个整数，计算其阶乘后显示到标准输出（显示器）。

在本实验中并不需要执行这个小程序，只需要使用 Lex 生成的扫描程序对这个 TINY 源代码文件进行扫描，最后统计出各种符号的数量。TINY 语言中的符号说明可以参考下面的表格。

序号	符号或语句	说明
1	{…}	在两个大括号之间的是注释。
2	read	从标准输入（键盘）读取数据。是一个关键字。

3	write	将数据写入标准输出（屏幕）。是一个关键字。
4	if … then… else…end	if 语句。if 的后面是一个布尔表达式，then 和 else 的后面是一个语句块，其中 else 是可选的。end 表示结束。包括了四个关键字。
5	repeat…until…	repeat 语句。repeat 后面是一个语句块，until 后面是一个布尔表达式。包括了两个关键字。
6	<、=	比较运算符。<是小于符号。=是等于符号
7	+、-、*、/	算数运算符。
8	:=	赋值运算符。
9	正整数	由数字 0-9 组成。
10	标识符	由大写字母和小写字母组成。
11	;	在语句的结束位置有一个分号。

define.h 文件

在此文件中定义了一个枚举类型，对应于 TINY 语言中的各种符号。注意，还包括了“文件结束”和“错误”，其中 yylex 函数在遇到文件结束时，默认会返回 0，所以将“文件结束”定义在开始位置，这样它的值也为 0。

scan.txt 文件

此文件是 Lex 的输入文件。根据 Lex 输入文件的格式，此文件分为三个部分（由%%分隔），各个部分的说明可以参见下面的表格。

名称	说明
第一部分	● 在%{和%}之间的直接插入 C 源代码文件的内容。包括了要包含的头文件，以及 TINY 语言中各种符号的计数器，用于保存扫描的结果。 ● 定义了换行（newline）和空白（whitespace）的正则表达式。注意，空白包括了空格和制表符。
第二部分	包含了一组规则，当规则中的正则表达式匹配时，yylex 函数会执行规则提供的源代码。主要包含了下面的规则： ● 比较运算符、算数运算符等的规则。这些规则直接使用字符串进行匹配。若匹配成功，yylex 函数返回对应的枚举值。 ● 匹配换行的规则。匹配成功，就统计行数。 ● 匹配空白的规则。匹配成功，就忽略。 ● 由于编写注释的正则表达式比较复杂，所以在匹配注释的开始符号后，直接编写 C 代码来匹配注释的结束符号。这里用到的 input 函数是一个 Lex 的内部函数，它会返回一个输入字符。 ● 使用 Lex 正则表达式中的“.”元字符匹配其他的任何字符。当之前的规则均匹配失败时，就会在此规则匹配成功，从而返回错误类型。
第三部分	这部分中的内容会直接插入 C 源代码文件。此部分内容的说明可以参见下面的表格。

内容	说明
main 函数	主函数。这里用到了 C 语言定义的 main 函数的两个参数，这两个参数的用法可以参考函数的注释。 在 main 函数中首先使用 fopen 函数打开了待处理的文件，然后将此文件作为 Lex 扫描程序的输入（yyin），之后调用 yylex 函数开始扫描，并调用 stat 函数统计各种符号的数量，最后调用 output 函数输出统计的结果。
id2keyword 函数	此函数将标识符转换为对应的关键字类型，如果通过参数传入的字符串不能与任

	何关键字匹配，就仍然返回标识符类型。此函数的函数体还不完整，留给读者完成。
stat 函数	此函数根据 tt 参数传入的符号类型，增加符号类型对应的计数器。如果 tt 是标识符类型（ID），会首先调用 id2keyword 函数,尝试将标识符类型转换为对应的关键字类型。
output 函数	输出统计的结果。

main.c 文件

此文件默认是一个空文件。Lex 根据输入文件生成的 C 源代码会输出到此文件中。此文件会包含自动生成的 yylex 函数的定义，此函数实现了与输入文件相对应的 DFA 表驱动。

3.3 生成项目

按照下面的步骤生成项目：

1. 在“生成”菜单中选择“重新生成项目”（快捷键 Ctrl+Alt+F7）。

在生成的过程中，CP Lab 会首先使用 Flex 程序根据输入文件 scan.txt 来生成 main.c 文件，然后，将 main.c 文件重新编译、链接为可以运行的可执行文件。

在生成的 main.c 文件中，尝试找到 scan.txt 文件中第一部分和第三部分 C 源代码插入的位置，并尝试查找 input 函数和 yylex 函数的定义，以及 yyin 和 yytext 等变量的定义。

注意：在下面的实验步骤中，如果需要生成项目，应尽量使用“重新生成项目”功能。如果习惯使用“生成项目”（快捷键 F7）功能，可能需要连续使用两次此功能才能生成最新的项目。

3.4 运行项目

在没有对项目的源代码进行任何修改的情况下，按照下面的步骤运行项目：

1、选择“调试”菜单中的“开始执行（不调试）”（快捷键 Ctrl+F5）。

在启动运行时，CP lab 会自动将 sample.txt 文件的名称作为参数传给可执行文件（即 main 函数中 argv[1]指向的字符串），所以，程序运行完毕后，会在 Windows 控制台窗口中显示对 sample.txt 文件的扫描结果，如图 3-1 所示。

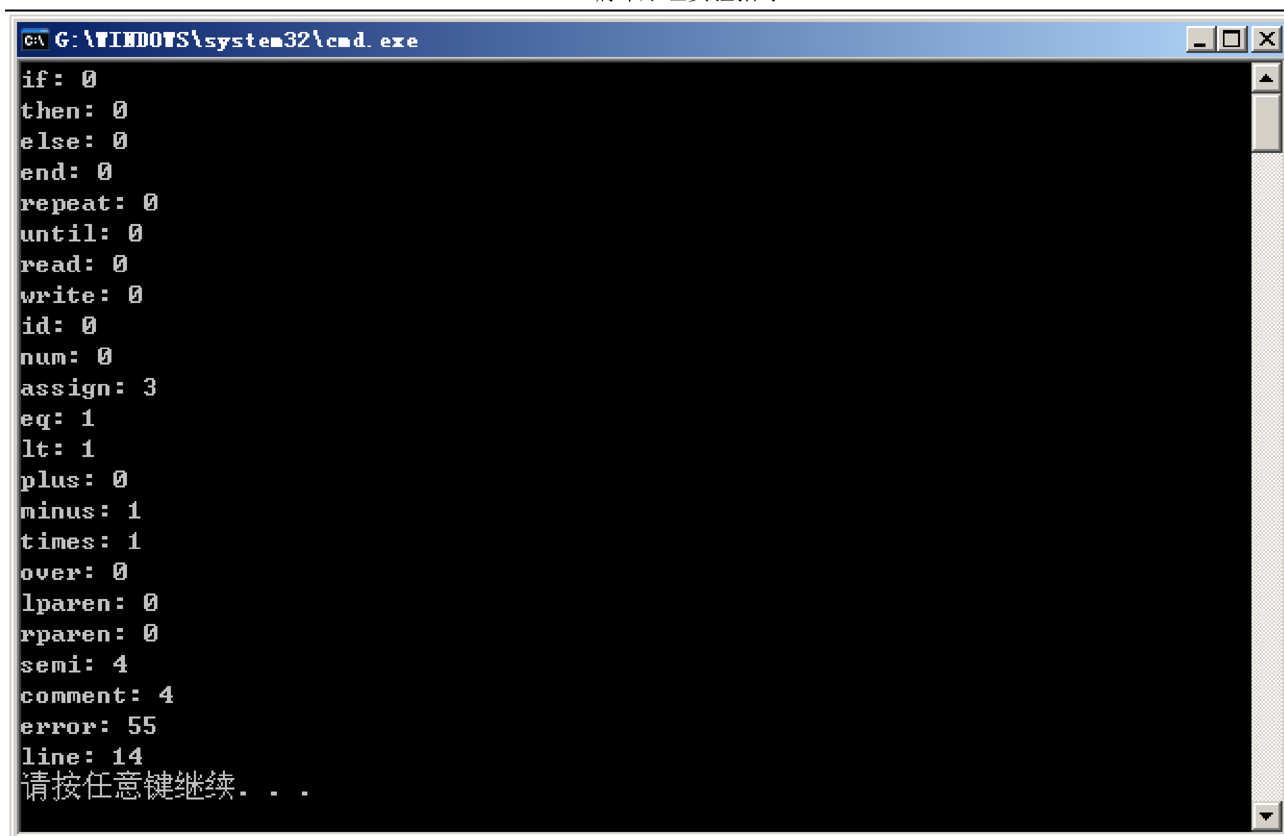

图 3-1：对 sample.txt 文件的扫描结果。

由于此时还没有在 scan.txt 中添加能够与标识符和正整数匹配的正则表达式，所以 id 和 num 的值均为 0，而标识符和正整数会被匹配为错误，所以，error 的数量大于 0。

3.5 添加标识符和正整数的统计功能

按照下面的步骤完成此练习：

1. 在 scan.txt 文件的第一部分添加标识符和正整数的正则表达式。
2. 在 scan.txt 文件的第二部分添加标识符和正整数的正则表达式匹配时的 C 源代码。注意，标识符的正则表达式也用来匹配所有的关键字，不要直接使用字符串来逐个的匹配关键字。
3. 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
4. 按 Ctrl+F5 启动执行项目。

执行的结果应该如图 3-2 所示，已经可以正确统计出标识符和正整数的数量。注意，由于此时 id2keyword 函数还无法将标识符转换为关键字，所以，所有的关键字都匹配成了标识符，关键字的数量都为 0。

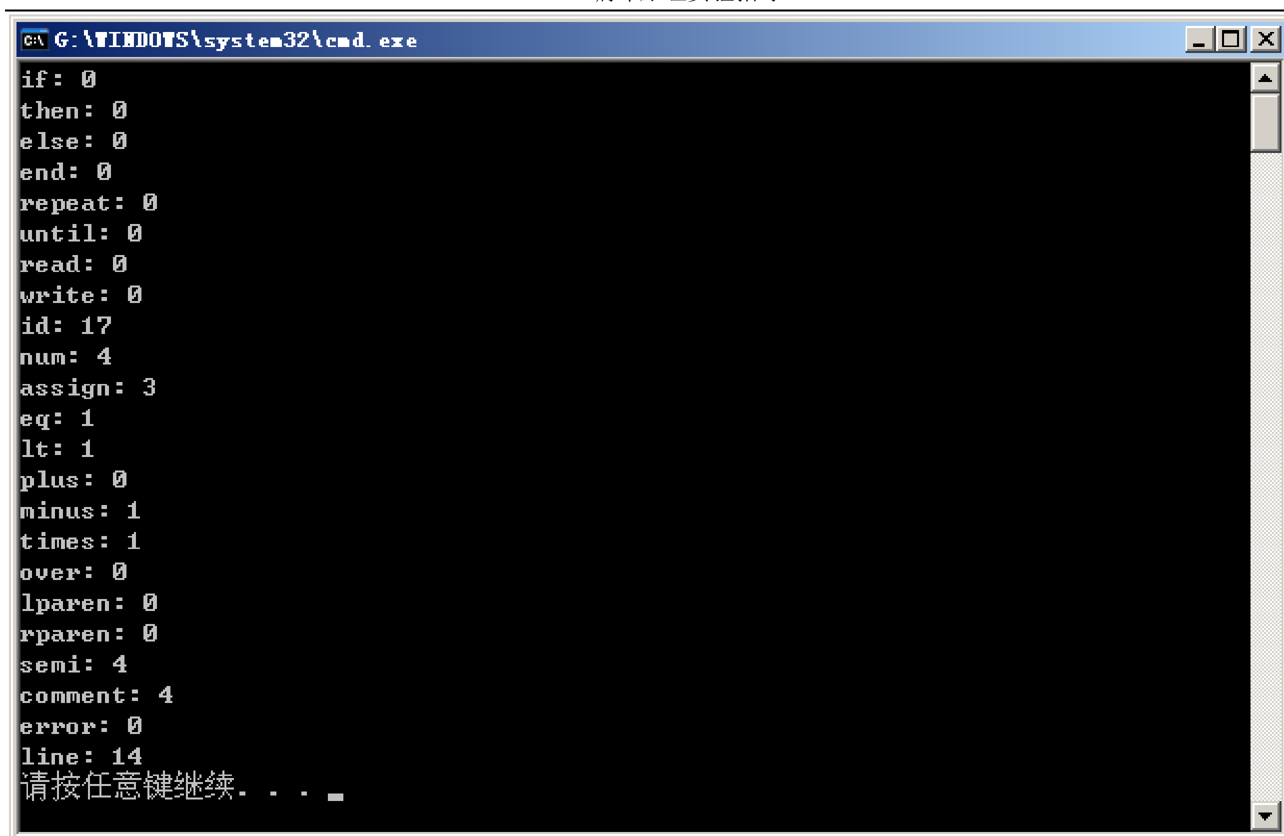

图 3-2：正确统计出标识符和正整数的数量。

3.6 添加关键字的统计功能

按照下面的步骤完成此练习：

1. 为 id2keyword 函数编写源代码。要求使用此函数前面定义的 key_table 表格中的数据，通过线性搜索的方式，根据标识符的字符串确定其对应的关键字类型。
2. 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Ctrl+F5 启动执行项目。

执行的结果应如图 3-3 所示，已经可以正确统计出各个关键字的数量。

注意：

1、 本实验的模板不提供演示功能，所以，如果执行的结果不正确，可以通过添加断点和单步调试的方法来查找错误的原因。

2、 断点应该添加在 scan.txt 文件中需要中断的 C 源代码行，不要添加在 main.c 文件中，否则无法命中断点。

```
G:\WINDOWS\system32\cmd.exe
if: 1
then: 1
else: 0
end: 1
repeat: 1
until: 1
read: 1
write: 1
id: 10
num: 4
assign: 3
eq: 1
lt: 1
plus: 0
minus: 1
times: 1
over: 0
lparen: 0
rparen: 0
semi: 4
comment: 4
error: 0
line: 14
请按任意键继续. . .
```

图 3-3：正确统计出各个关键字的数量。

3.7 添加 C 语言风格的注释

按照下面的步骤完成此练习：

1. 将 sample.txt 文件中的多行注释包括在“/*”和“*/”之间，将语句末尾的注释放在“//”的后面。
2. 修改 scan.txt 文件，使之能够正确匹配和统计这两种 C 语言风格的注释。
3. 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
4. 按 Ctrl+F5 启动执行项目，确保统计的注释数量是正确的。

四、思考与练习

1. 修改 key_table 表格中数据的顺序，使关键字按照字母顺序排列，然后修改 id2keyword 函数中的代码，使用二分法从 key_table 表格中查找关键字。
2. 使用 gperf 工具为 TINY 语言的关键字生成杂凑表（哈希表），然后修改 id2keyword 函数中的代码从杂凑表中查找关键字。（**提示：**CP Lab 提供了 gperf 工具，可以选择 CP Lab“工具”菜单中的“CP Lab 命令提示”，在弹出的 Windows 控制台窗口中输入命令“gperf C:\a.txt --output-file=C:\a.c”，即可为 a.txt 文件中列出的关键字生成杂凑表到 a.c 源代码文件中。选择 CP Lab“帮助”菜单中“其他帮助文档”中的“gperf 手册”可以获得更多帮助）。
3. 选择 CP Lab“帮助”菜单中“其他帮助文档”中的“Flex 手册”，学习 Flex 工具的更多用法。如果需要修改 Flex 程序在处理输入文件时的选项，可以在“项目管理器”窗口中，使用右键点击文件“scan.txt”，在弹出的快捷菜单中选择“属性”，然后选择“属性页”左侧“自定义生成步骤”的“常规”，就可以编辑右侧的 “命令行”选项了。

实验 4 消除左递归（无替换）

实验难度：★★★☆☆
建议学时：2 学时

一、 实验目的

- 了解在上下文无关文法中的左递归的概念。
- 掌握直接左递归的消除算法。

二、 预备知识

- 在这个实验中用到了单链表插入和删除操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。
- 理解指针的指针的概念和用法。在这个实验中，指针的指针被用来确定单链表插入的位置，以及插入的具体操作。
- 理解左递归和右递归的含义以及左递归的各种形式，如：简单直接左递归、普遍的直接左递归、一般的左递归。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为
https://www.codecode.net/engintime/cp-lab/Project-Template/Lab04.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：

(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 4 消除左递归（无替换）”;还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab04.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

RemoveLeftRecursion.h 文件

主要定义了与文法相关的数据结构，这些数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表，RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符。具体内容可参见下面两个表格。

Rule 的域	说明
RuleName	文法的名称

pFirstSymbol	指向文法的第一个 Select 的第一个 Symbol
pNextRule	指向下一条文法

RuleSymbol 的域	说明
pNextSymbol	指向下一个 Symbol
pOther	指向下一个 Select
isToken	是否为终结符。1 表示终结符，0 表示非终结符
TokenName	终结符的名称。isToken 为 1 时这个域有效
pRule	指向 Symbol 对应的 Rule。isToken 为 0 时这个域有效

下面是一个简单文法，并使用图表说明了该文法的存储结构：

```
A -> Aa | aB
B -> bB
```

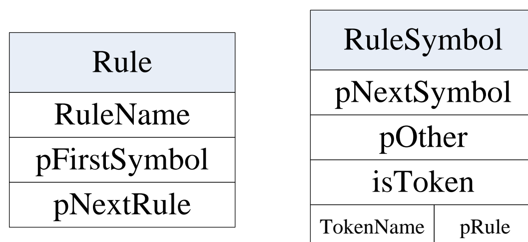

图 4-1：Rule 和 RuleSymbol 结构体图例

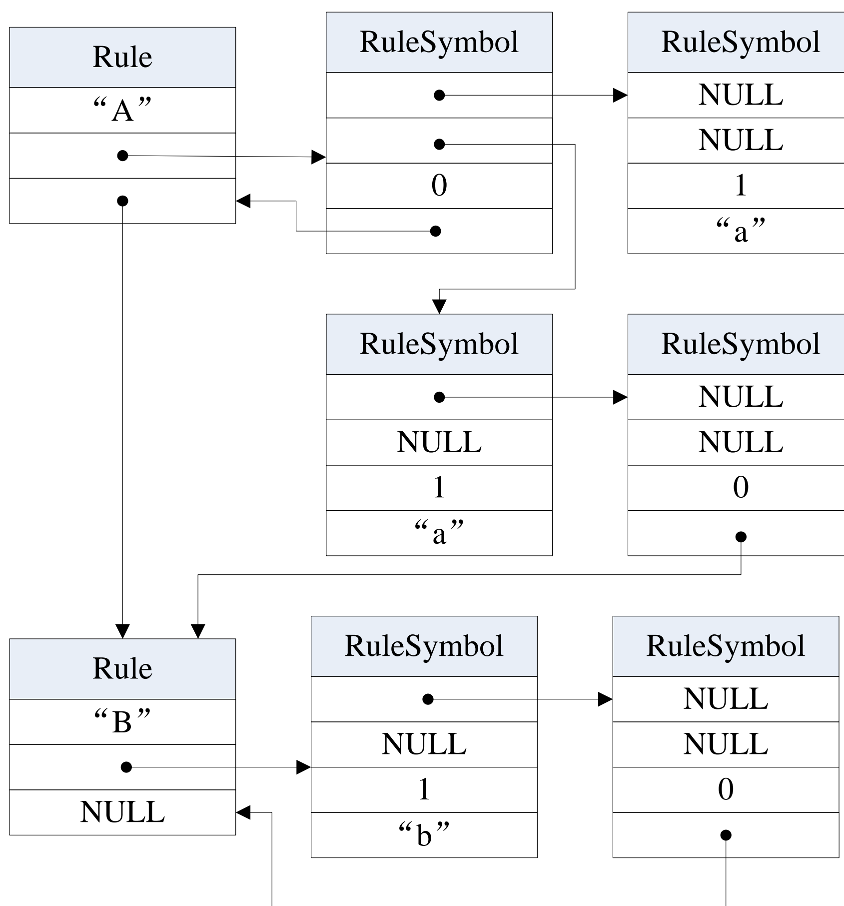

图 4-2：简单文法的存储结构

main.c 文件

定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法，然后调用了 PrintRule 函

数，打印消除左递归之前的文法，接着调用 RemoveLeftRecursion 函数对文法消除左递归，最后再次调用了 PrintRule 函数打印消除左递归之后的文法。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
AddSymbolToSelect	将一个 Symbol 添加到 Select 的末尾。此函数的函数体还不完整，留给读者完成。
AddSelectToRule	将 Select 加入到文法末尾，如果 Select 为 NULL 则将 ε 终结符加入到文法末尾。在本程序中 ε 可以用 $ 来代替。此函数的函数体还不完整，留给读者完成。
RemoveLeftRecursion	对文法消除左递归。在本函数中使用指针的指针 pSelectPtr 来确定符号在单链表中插入和删除的位置，在进入下一次循环之前应为 pSelectPtr 设置正确的值。此函数的函数体还不完整，留给读者完成。
InitRules	使用给定的数据初始化文法链表。
CreateRule	创建一个新的 Rule。
CreateSymbol	创建一个新的 Symbol。
FindRule	根据 RuleName 在文法链表中查找名字相同的文法。
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成。

3.3 为函数 InitRules 添加注释

在 InitRules 函数之前定义了两个结构体，这两个结构体用来定义初始化文法数据的存储形式。具体内容可参见下面两个表格。

SYMBOL 的域	说明
isToken	是否为终结符。1 表示终结符，0 表示非终结符。
Name	终结符和非终结符的名称。

RULE_ENTRY 的域	说明
RuleName	文法的名称。
Selects	SYMBOL 结构体的二维数组，其中每一行表示一个 Select，一行中的每个元素分别表示一个终结符或非终结符。

为 InitRules 函数添加注释。(注意，在本程序中使用了指针的指针，体会其在单链表的插入和删除操作中的作用)。

3.4 为 PrintRule 函数编写源代码

为 PrintRule 函数编写源代码，同时理解在本程序中文法的链式存储结构。编写完源代码后，选择“调试”菜单中的“调试/开始执行（不调试）”，会在控制台窗口输出文法的产生式。如图 4-3 所示。由于还没有为 RemoveLeftRecursion 函数和其他未完成的函数编写源代码，所以前后两次输出的文法是一样的。

```
G:\WINDOWS\system32\cmd.exe
Before Remove Left Recursion:
A->Aa|bA|c|Ad

After Remove Left Recursion:
A->Aa|bA|c|Ad
请按任意键继续. . .
```

图 4-3：打印文法

3.5 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解对文法消除左递归的过程。“转储信息”窗口显示的数据信息包括：

- 文法。由于在本实验中没有进行替换操作，所以只能处理一条文法的情况。
- 新文法。对原文法消除左递归后，新生成的包含右递归的文法。
- 执行到函数末尾时，会将新文法加入到文法链表中。
- 在消除左递归的过程中会使用加粗的中括号表示出游标指向的 Select。

3.6 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 RemoveLeftRecursion 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

四、思考与练习

1. 请读者仔细检查自己编写的源代码，查看在消除左递归的过程中，是否调用了 free 函数释放了从文法链表中移除的 Symbol，否则会造成内存泄露。
2. 编写一个 FreeRule 函数，当在 main 函数的最后调用此函数时，可以将整个文法的内存释放掉，从而避免内存泄露。
3. 本实验目前要求读者编写的代码只能够处理一条包含直接左递归的文法，请读者思考一下，如果有多条文法都包含左递归，甚至还包含了间接左递归，应该如何改进现有的源代码。

实验 5 消除左递归（有替换）

实验难度：★★★★☆
建议学时：4 学时

一、 实验目的

- 了解在上下文无关文法中的左递归的概念。
- 掌握直接左递归、一般左递归的消除算法。

二、 预备知识

- 在这个实验中用到了单链表插入和删除操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。
- 理解指针的指针的概念和用法。在这个实验中，指针的指针被用来确定单链表插入和删除的位置，以及用来完成插入和删除的具体操作。
- 理解左递归和右递归的含义以及左递归的各种形式，如：简单直接左递归、普遍的直接左递归、一般的左递归。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为
https://www.codecode.net/engintime/cp-lab/Project-Template/Lab05.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 5 消除左递归（有替换）”;还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab05.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

RemoveLeftRecursion.h 文件

主要定义了与文法相关的数据结构，这些数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表，RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符。具体内容可参见下面两个表格。

Rule 的域	说明
RuleName	文法的名称

pFirstSymbol	指向文法的第一个 Select 的第一个 Symbol
pNextRule	指向下一条文法

RuleSymbol 的域	说明
pNextSymbol	指向下一个 Symbol
pOther	指向下一个 Select
isToken	是否为终结符。1 表示终结符，0 表示非终结符
TokenName	终结符的名称。isToken 为 1 时这个域有效
pRule	指向 Symbol 对应的 Rule。isToken 为 0 时这个域有效

下面是一个简单文法，并使用图表说明了该文法的存储结构：

```
A -> Aa | aB
B -> bB
```

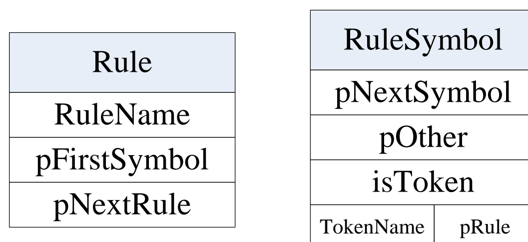

图 5-1：Rule 和 RuleSymbol 结构体图例

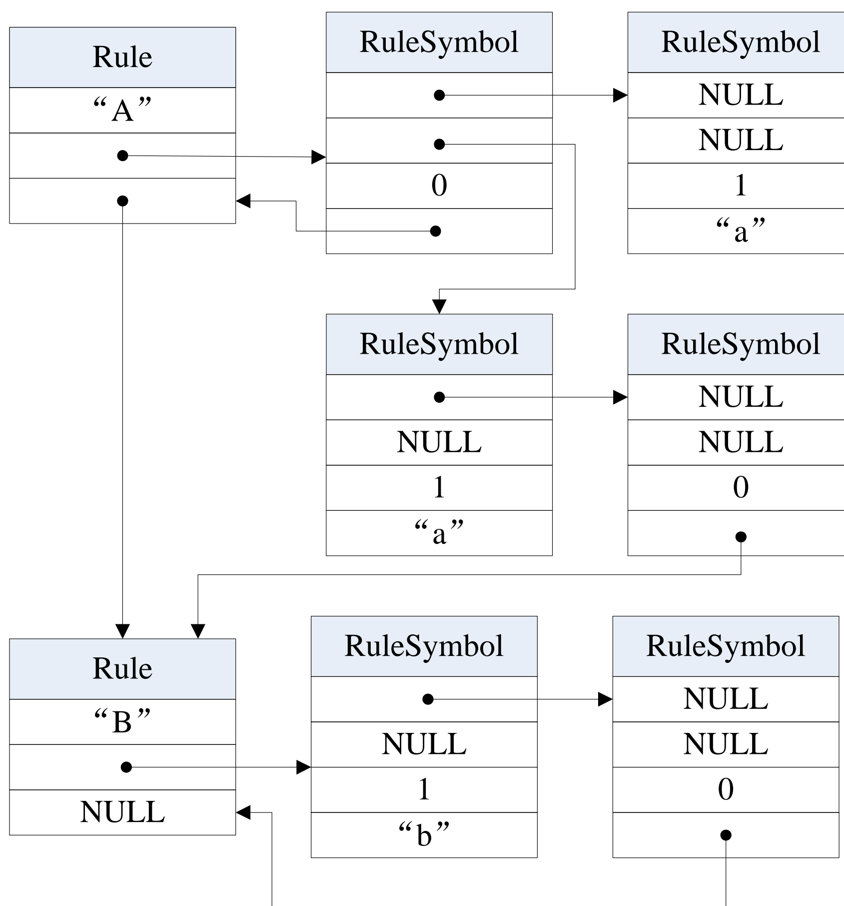

图 5-2：简单文法的存储结构

main.c 文件

定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法，然后调用了 PrintRule 函

数，打印消除左递归之前的文法，接着调用 RemoveLeftRecursion 函数对文法消除左递归，最后再次调用了 PrintRule 函数打印消除左递归之后的文法。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
SymbolNeedReplace	判断当前 Rule 中的一个 Symbol 是否需要被替换。如果 Symbol 是一个非终结符，且 Symbol 对应的 Rule 在当前 Rule 之前，就需要被替换。此函数的函数体还不完整，留给读者完成。
CopySymbol	拷贝一个 Symbol。此函数的函数体还不完整，留给读者完成。
CopySelect	拷贝一个 Select。在此函数中可以调用 CopySymbol 函数，将 Select 中的 Symbol 逐个进行拷贝。此函数的函数体还不完整，留给读者完成。
ReplaceSelect	替换一个 Select 的第一个 Symbol。 **提示：** ● 在调用此函数之前，已经调用了 SymbolNeedReplace 函数，确保 Select 的第一个 Symbol 需要被替换。 ● Select 的第一个 Symbol 是一个非终结符，该 Symbol 对应的 Rule 就是用于替换该 Symbol 模板。 ● 当需要拷贝 Select 时，可以调用 CopySelect 函数。 此函数的函数体还不完整，留给读者完成。
FreeSelect	释放一个 Select 的内存。此函数的函数体还不完整，留给读者完成。
RuleHasLeftRecursion	判断一条 Rule 是否存在左递归。 **提示：** ● 只有一条 Rule 中的一个 Select 存在左递归，那么这条 Rule 就存在左递归。 ● 如果一个 Select 的第一个符号（非终结符）对应的就是此条文法，这个 Select 就存在左递归。 此函数的函数体还不完整，留给读者完成。
AddSymbolToSelect	将一个 Symbol 添加到 Select 的末尾。此函数的函数体还不完整，留给读者完成。
AddSelectToRule	将一个 Select 加入到文法末尾，当 Select 为 NULL 时就将一个 ε 终结符加入到文法末尾。在本程序中 ε 可以用 $ 来代替。此函数的函数体还不完整，留给读者完成。
RemoveLeftRecursion	对文法消除左递归。 **提示：** ● 在本函数中包含了替换算法，从而可以处理间接左递归的情况。而且由于替换后还可能需要替换，所以设置了一个标识 isChange，并初始化为 0，每当发生替换之后就将该标识赋值为 1，并将此标识作为循环条件，直到没有替换发生时，才能结束替换。 ● 在本函数中使用指针的指针 pSelectPtr 来确定符号在单链表中插入和删除的位置，在进入下一次循环之前应为 pSelectPtr 设置正确的值。 ● 在每处理完一条文法后，会将文法链表的游标指向新文法，这样在继续执行 for 循环的移动游标操作后，就会跳过这条新文法，从而继续对后面的文法进行消除左递归操作（新文法不包含左递归，所以并不需要对这条文法进行处理）。 此函数的函数体还不完整，留给读者完成。
InitRules	使用给定的数据初始化文法链表。

CreateRule	创建一个新的 Rule。
CreateSymbol	创建一个新的 Symbol。
FindRule	根据 RuleName 在文法链表中查找名字相同的文法。
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成。

3.3 为函数 InitRules 添加注释

在 InitRules 函数之前定义了两个结构体，这两个结构体用来定义初始化文法数据的存储形式。具体内容可参见下面两个表格。

SYMBOL 的域	说明
isToken	是否为终结符。1 表示终结符，0 表示非终结符。
Name	终结符和非终结符的名称。

RULE_ENTRY 的域	说明
RuleName	文法的名称。
Selects	SYMBOL 结构体的二维数组，其中每一行表示一个 Select，一行中的每个元素分别表示一个终结符或非终结符。

为 InitRules 函数添加注释。(注意，在本程序中使用了指针的指针，体会其在单链表的插入和删除操作中的作用)。

3.4 为 PrintRule 函数编写源代码

为 PrintRule 函数编写源代码，同时理解在本程序中文法的链式存储结构，编写完源代码后，选择“调试”菜单中的“调试/开始执行（不调试)”，会在控制台窗口输出文法的产生式。如图 5-3 所示。由于还没有为 RemoveLeftRecursion 函数和其他未完成的函数编写源代码，所以前后两次输出的文法是一样的。

```
G:\WINDOWS\system32\cmd.exe
Before Remove Left Recursion:
A->Ba|Aa|c
B->Bb|Ab|d

After Remove Left Recursion:
A->Ba|Aa|c
B->Bb|Ab|d
请按任意键继续. . .
```

图 5-3：打印文法

3.5 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解对文法消除左递归的过程。“转储信息”窗口显示的数据信息包括：

- 文法。在消除左递归的过程中会使用游标来指向正在操作的文法。
- 新文法。对原文法消除左递归后，新生成的包含右递归的文法。
- 执行到最外层的循环的末尾时，会将新文法加入到文法链表中。
- 在消除左递归的过程中会使用加粗的中括号表示出游标指向的 Select。

3.6 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 RemoveLeftRecursion 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

四、思考与练习

1. 请读者仔细检查自己编写的源代码，查看在消除左递归的过程中，是否调用了 free 函数释放了从文法链表中移除的 Symbol，是否调用了 FreeSelect 函数释放了从文法链表中移除的 Select，否则会造成内存泄露。
2. 编写一个 FreeRule 函数，当在 main 函数的最后调用此函数时，可以将整个文法的内存释放掉，从而避免内存泄露。
3. 使用自己编写的代码对下面两个例子进行验证，确保程序可以为所有形式的文法消除左递归，并验证通过（文法中的终结符用粗体表示）。

   例 1： A -> B**a** | A**a** | **c**

   B -> B**b** | A**b** | D

   D -> A**d**

   例 2： exp -> exp addop term| term

   addop -> **+** | **-**

   term -> term mulop factor | factor

   mulop -> *****

   factor -> **(**exp**)** | **number**

实验 6 提取左因子

实验难度：★★★☆☆
建议学时：2 学时

一、 实验目的

- 了解在上下文无关文法中的左因子的概念。
- 掌握提取左因子的算法。

二、 预备知识

- 在这个实验中用到了单链表插入和删除操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。
- 理解指针的指针的概念和用法。在这个实验中，指针的指针被用来确定单链表插入和删除的位置，以及用来完成插入和删除的具体操作。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab06.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：

(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 6 提取左因子”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab06.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

PickupLeftFactor.h 文件

主要定义了与文法相关的数据结构，这些数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表，RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符。具体内容可参见下面两个表格。

Rule 的域	说明
RuleName	文法的名称
pFirstSymbol	指向文法的第一个 Select 的第一个 Symbol
pNextRule	指向下一条文法

RuleSymbol 的域	说明
pNextSymbol	指向下一个 Symbol
pOther	指向下一个 Select
isToken	是否为终结符。1 表示终结符，0 表示非终结符
TokenName	终结符的名称。isToken 为 1 时这个域有效
pRule	指向 Symbol 对应的 Rule。isToken 为 0 时这个域有效

下面是一个简单文法，并使用图表说明了该文法的存储结构：

```
A -> Aa | aB
B -> bB
```

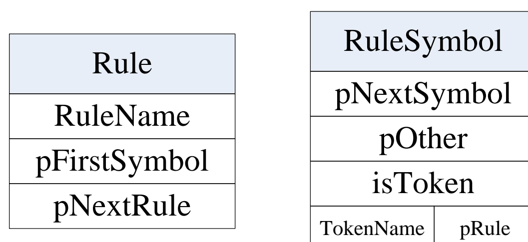

图 6-1：Rule 和 RuleSymbol 结构体图例

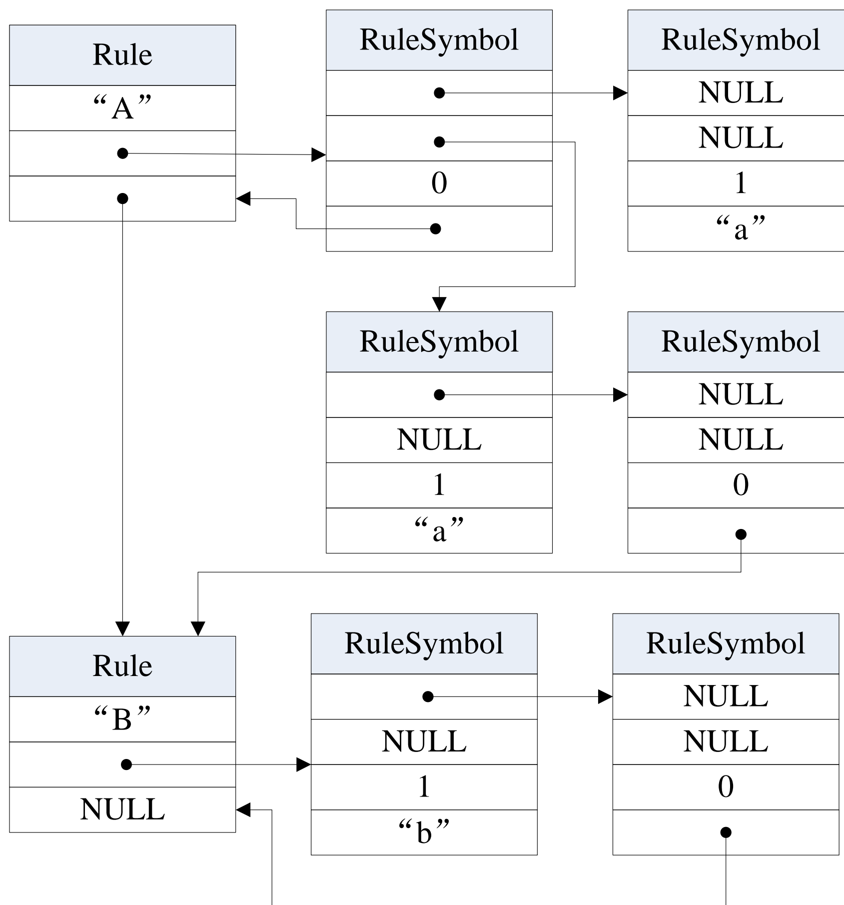

图 6-2：简单文法的存储结构

main.c 文件

定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法，然后调用了 PrintRule 函数，打印提取左因子之前的文法，接着调用 PickupLeftFactor 函数对文法提取左因子，最后再次调用了 PrintRule 函数打印提取左因子之后的文法。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
GetSymbol	根据下标找到 Select 中的一个 Symbol。
LeftFactorMaxLength	以一个 Select 为模板，确定左因子的最大长度。 **提示：** ● 虽然 Select 中的 Symbol 是个单链表，但是可以调用 GetSymbol 函数通过下标访问 Symbol，这样可以简化逐个访问 Symbol 的过程。 ● 当需要比较两个 Symbol 是否相同时，可以调用 SymbolCmp 函数。 此函数的函数体还不完整，留给读者完成。
SymbolCmp	比较两个相同类型(同为终结符或同为非终结符)的 Symbol 是否具有相同的名字。此函数的函数体还不完整，留给读者完成。
NeedPickup	取文法中的一个 Select 与 SelectTemplate 进行比较，判断该 Select 是否需要提取左因子。 **提示：** ● 可以调用 GetSymbol 函数通过下标访问 Symbol，这样可以简化逐个访问 Symbol 的过程。 ● 当需要比较两个 Symbol 是否相同时，可以调用 SymbolCmp 函数。 此函数的函数体还不完整，留给读者完成。
AddSelectToRule	将一个 Select 加入到文法末尾，当 Select 为 NULL 时就将一个ε终结符加入到文法末尾。在本程序中ε可以用 $ 来代替。此函数的函数体还不完整，留给读者完成。
GetUniqueRuleName	此函数会在新文法的名称后面添加后缀，直到新文法的名称唯一。
FreeSelect	释放一个 Select 的内存。此函数的函数体还不完整，留给读者完成。
PickupLeftFactor	对文法提取左因子。 **提示：** ● 由于对一条文法提取左因子之后还可能存在左因子，所以设置了一个标识 isChange，并初始化为 0，每当提取了左因子之后就将该标识赋值为 1，并将此标识作为循环条件，直到没有提取左因子操作发生时，才能结束循环。 ● 在本函数中使用指针的指针 pSelectPtr 来确定符号在单链表中插入和删除的位置，在进入下一次循环之前应为 pSelectPtr 设置正确的值。 此函数的函数体还不完整，留给读者完成。
InitRules	使用给定的数据初始化文法链表。
CreateRule	创建一个新的 Rule。
CreateSymbol	创建一个新的 Symbol。
FindRule	根据 RuleName 在文法链表中查找名字相同的文法。
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成。

3.3 为函数 InitRules 添加注释

在 InitRules 函数之前定义了两个结构体，这两个结构体用来定义初始化文法数据的存储形式。具体内容可参见下面两个表格。

SYMBOL 的域	说明
isToken	是否为终结符。1 表示终结符，0 表示非终结符。
Name	终结符和非终结符的名称。

RULE_ENTRY 的域	说明
RuleName	文法的名称。
Selects	SYMBOL 结构体的二维数组，其中每一行表示一个 Select，一行中的每个元素分别表示一个终结符或非终结符。

为 InitRules 函数添加注释。(注意，在本程序中使用了指针的指针，体会其在单链表的插入和删除操作中的作用)。

3.4 为 PrintRule 函数编写源代码

为 PrintRule 函数编写源代码，同时理解在本程序中文法的链式存储结构，编写完源代码后，选择“调试”菜单中的“调试/开始执行（不调试）”，会在控制台窗口输出文法的产生式。如图 6-3 所示。由于还没有为 PickupLeftFactor 函数和其他未完成的函数编写源代码，所以前后两次输出的文法是一样的。

```
G:\WINDOWS\system32\cmd.exe
Before Pickup Left Factor:
A->abC|abcD|abcE

After Pickup Left Factor:
A->abC|abcD|abcE
请按任意键继续. . .
```

图 6-3：打印文法

3.5 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解对文法提取左因子的过程。“转储信息”窗口显示的数据信息包括：

- 文法。在提取左因子的过程中会使用游标来指向正在操作的文法。
- 新文法。新生成的包含原文法左因子之后部分的文法。
- 执行到循环的末尾时，会将新文法加入到文法链表中。
- 在查找包含左因子的 Select 模板过程中会使用空心中括号表示出游标指向的 Select
- 在提取左因子的过程中会使用加粗的中括号表示出游标指向的 Select。

3.6 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 PickupLeftFactor 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

四、思考与练习

1. 请读者仔细检查自己编写的源代码，查看在消除左递归的过程中，是否调用了 free 函数释放了从文法链表中移除的 Symbol，是否调用了 FreeSelect 函数释放了从文法链表中移除的 Select，否则会造成内存泄露。

2. 编写一个 FreeRule 函数，当在 main 函数的最后调用此函数时，可以将整个文法的内存释放掉，从而避免内存泄露。
3. 使用自己编写的代码对下面的例子进行验证，确保程序可以为所有形式的文法提取左因子，并验证通过。

   A -> B | abcD | abB | abcE | abcM | D | aB

   N -> ab | abD
4. 本实验使用的例子第一次提取了左因子“ab”，请读者尝试在第一次提取左因子“abc”，并与之前的结果进行比较。然后尝试修改提取左因子的算法，总是从文法中优先提取最长的左因子。

实验 7 First 集合

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 了解在上下文无关文法中的 First 集合的定义。
- 掌握计算 First 集合的方法。

二、 预备知识

- 对 First 集合的定义有初步的理解。读者可以参考配套的《编译原理》教材，预习这一部分内容。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab07.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 7 First 集合”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab07.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

First.h 文件

主要定义了与文法和集合相关的数据结构。

有关文法的数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表，RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符（**注意：**此处的 RuleSymbol 结构体与之前在消除左递归和提取左因子中的 RuleSymbol 结构体有一些差异，原因是在计算 First 集合时，将文法中的每个选择都单独写成了一个产生式，所以 RuleSymbol 结构体就进行了相应的简化）。

有关集合的数据结构定义了集合的线性表存储形式。其中 Set 结构体用于定义集合的名称和终结符数组，SetList 结构体用于定义一个以 Set 为元素的线性表。具体内容可参见下面的表格。

Rule 的域	说明
RuleName	文法的名称
pFirstSymbol	指向文法的第一个 Symbol

pNextRule	指向下一条文法

RuleSymbol 的域	说明
pNextSymbol	指向下一个 Symbol
isToken	是否为终结符。1 表示终结符，0 表示非终结符
SymbolName	终结符和非终结符的名称。

Set 的域	说明
Nane	集合的名称
Terminal	终结符数组
nTerminalCount	终结符数组元素个数。与前一项构成一个线性表

SetList 的域	说明
Sets	集合数组
nSetCount	集合数组元素个数。与前一项构成一个线性表

下面是一个简单文法，并使用图表说明了该文法的存储结构：

```
A -> Aa
A -> aB
B -> bB
```

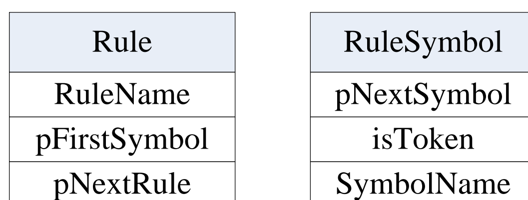

图 7-1：Rule 和 RuleSymbol 结构体图例

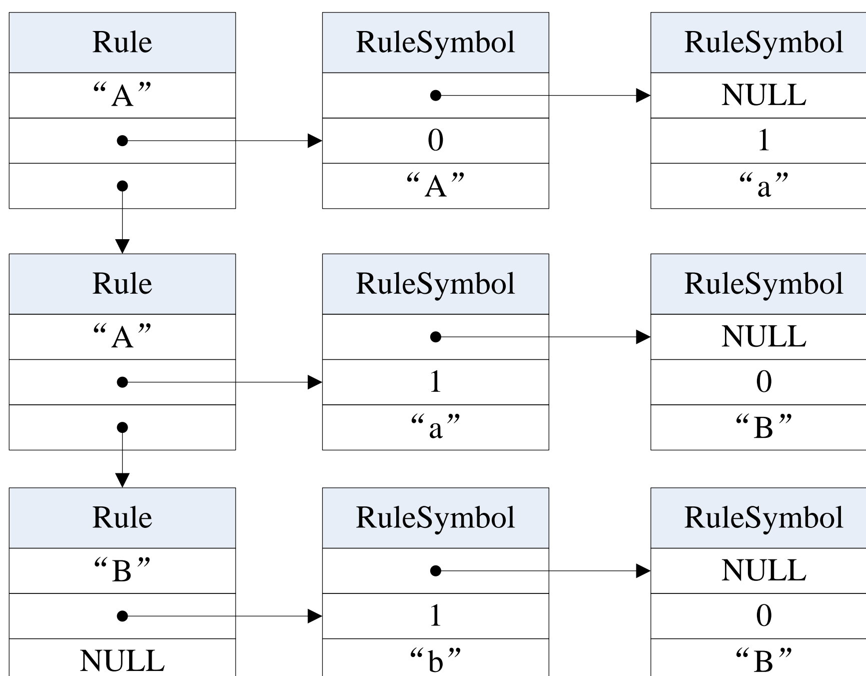

图 7-2：简单文法的存储结构

main.c 文件

定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法，然后调用了 PrintRule 函数打印文法，接着初始化了保存 First 集合的线性表，最后调用 First 函数求文法的 First 集合。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
AddOneSet	添加一个 Set 到 SetList 中。 **提示：** ● 在这个函数中可以首先调用一次 GetSet 函数，用来判断该 SetList 是否已经存在同名的 Set，从而忽略重复的名称。 此函数的函数体还不完整，留给读者完成。
GetSet	根据名称在 SetList 中查找 Set。此函数的函数体还不完整，留给读者完成。
AddTerminalToSet	添加一个终结符到 Set。 **提示：** ● 需要忽略重复的终结符。 此函数的函数体还不完整，留给读者完成。
AddSetToSet	将源 Set 中的所有终结符添加到目标 Set 中。 **提示：** ● 需要忽略源 Set 中的ε。 ● 可以通过调用 AddTerminalToSet 函数将终结符加入到目标 Set 中。 此函数的函数体还不完整，留给读者完成。
SetHasVoid	判断 Set 的终结符中是否含有ε。此函数的函数体还不完整，留给读者完成。
First	求文法的 First 集合。 **提示：** ● 由于对一个文法符号（一个终结符或非终结符）求 First 集合可能会依赖于其他的文法符号的 First 集合，所以 设置了一个标识 isChange，并初始化为 0，每当集合列表中的某个集合发生变化后就将该标识赋值为 1，并将此标识作为循环条件，直到没有变化发生时，才能结束循环。 ● 在循环末尾，如果当前文法产生式的每一个符号的 First 集合都包含ε，就将ε终结符添加到当前文法名称的 First 集合中。 此函数的函数体还不完整，留给读者完成。
InitRules	使用给定的数据初始化文法链表。
CreateRule	创建一个新的 Rule。
CreateSymbol	创建一个新的 Symbol。
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成。

3.3 为函数 InitRules 添加注释

在 InitRules 函数之前定义了两个结构体，这两个结构体用来定义初始化文法数据的存储形式。具体内容可参见下面两个表格。

SYMBOL 的域	说明
isToken	是否为终结符。1 表示终结符，0 表示非终结符。
SymbolName	终结符或非终结符的名称。

RULE_ENTRY 的域	说明

RuleName	文法的名称。
Symbols	SYMBOL 结构体数组，其中每个元素表示一个终结符或非终结符。

为 InitRules 函数添加注释。（注意，在本程序中使用了指针的指针，体会其在单链表的插入和删除操作中的作用）。

3.4 为 PrintRule 函数编写源代码

为 PrintRule 函数编写源代码，同时理解在本程序中文法的链式存储结构，编写完源代码后，选择“调试”菜单中的“调试/开始执行（不调试）”，会在控制台窗口输出文法的产生式。

```
G:\WINDOWS\system32\cmd.exe
exp -> exp addop term
exp -> term
addop -> +
addop -> -
term -> term mulop factor
term -> factor
mulop -> *
factor -> ( exp )
factor -> number
请按任意键继续. . .
```

图 7-3：打印文法

3.5 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解求文法的 First 集合的过程。“转储信息”窗口显示的数据信息包括：

- 在求文法的 First 集合的过程中会使用游标来指向正在操作的文法。
- 在求文法的 First 集合的过程中会使用加粗的中括号表示出游标指向的 Symbol。
- First 集合列表。显示每一个文法符号的 First 集合的构造过程。

3.6 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 First 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

四、思考与练习

1. 编写一个 FreeRule 函数，当在 main 函数的最后调用此函数时，可以将整个文法的内存释放掉，从而避免内存泄露。
2. 使用自己编写的代码对下面的例子进行验证，确保程序可以为所有形式的文法求 First 集合，并验证通过。（文法中的终结符用粗体表示）。

   ```
   A -> B
   A -> ε
   ```

```
B -> C
B -> ε
C -> AB
C -> *
```

实验 8 Follow 集合

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 了解在上下文无关文法中的 First 集合和 Follow 集合的定义。
- 掌握计算 First 集合和 Follow 集合的方法。

二、 预备知识

- 对 First 集合和 Follow 集合的定义有初步的理解。读者可以参考配套的《编译原理》教材，预习这一部分内容。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab08.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 8 Follow 集合”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab08.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

Follow.h 文件

主要定义了与文法和集合相关的数据结构。

有关文法的数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表，RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符（**注意：**此处的 RuleSymbol 结构体与之前在消除左递归和提取左因子中的 RuleSymbol 结构体有一些差异，原因是在计算 First 集合和 Follow 集合时，将文法中的每个选择都单独写成了一个产生式，所以 RuleSymbol 结构体就进行了相应的简化）。

有关集合的数据结构定义了集合的线性表存储形式。其中 Set 结构体用于定义集合的名称和终结符数组，SetList 结构体用于定义一个以 Set 为元素的线性表。具体内容可参见下面的表格。

Rule 的域	说明
RuleName	文法的名称

pFirstSymbol	指向文法的第一个 Symbol
pNextRule	指向下一条文法

RuleSymbol 的域	说明
pNextSymbol	指向下一个 Symbol
isToken	是否为终结符。1 表示终结符，0 表示非终结符
SymbolName	终结符和非终结符的名称

Set 的域	说明
Nane	集合的名称
Terminal	终结符数组
nTerminalCount	终结符数组元素个数。与前一项构成一个线性表

SetList 的域	说明
Sets	集合数组
nSetCount	集合数组元素个数。与前一项构成一个线性表

下面是一个简单文法，并使用图表说明了该文法的存储结构：

```
A -> Aa
A -> aB
B -> bB
```

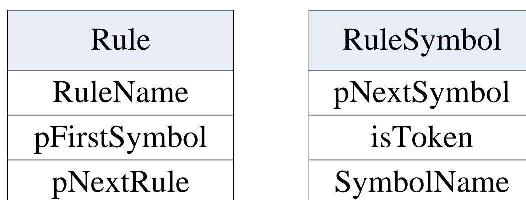

图 8-1：Rule 和 RuleSymbol 结构体图例

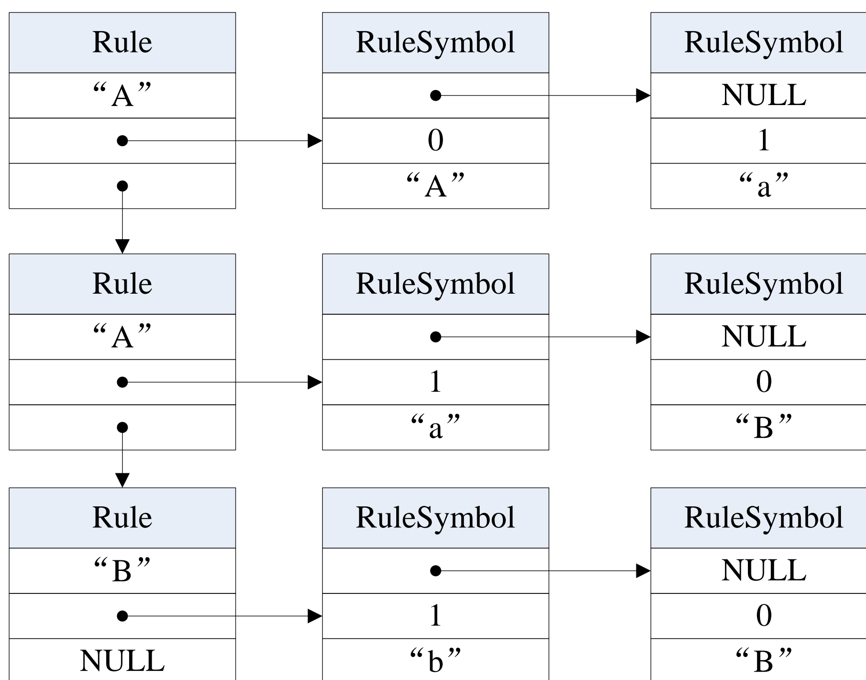

图 8-2：简单文法的存储结构

main.c 文件

定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法，然后调用了 PrintRule 函数打印文法，接着初始化了保存 First 集合和 Follow 集合的线性表，最后调用 Follow 函数求文法的 First 集合和 Follow 集合。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
AddOneSet	添加一个 Set 到 SetList 中。 **提示：** ● 在这个函数中可以首先调用一次 GetSet 函数，用来判断该 SetList 是否已经存在同名的 Set，从而忽略重复的名称。 此函数的函数体还不完整，留给读者完成。
GetSet	根据名称在 SetList 中查找 Set。此函数的函数体还不完整，留给读者完成。
AddTerminalToSet	添加一个终结符到 Set。 **提示：** ● 需要忽略重复的终结符。 此函数的函数体还不完整，留给读者完成。
AddSetToSet	将源 Set 中的所有终结符添加到目标 Set 中。 **提示：** ● 需要忽略源 Set 中的 ε 。 ● 可以通过调用 AddTerminalToSet 函数将终结符加入到目标 Set 中。 此函数的函数体还不完整，留给读者完成。
SetHasVoid	判断 Set 的终结符中是否含有 ε 。此函数的函数体还不完整，留给读者完成。
First	求文法的 First 集合。 **提示：** ● 由于对一个文法符号（一个终结符或非终结符）求 First 集合可能会依赖于其他的文法符号的 First 集合，所以 设置了一个标识 isChange，并初始化为 0，每当集合列表中的某个集合发生变化后就将该标识赋值为 1，并将此标识作为循环条件，直到没有变化发生时，才能结束循环。 ● 在循环末尾，如果当前文法产生式的每一个符号的 First 集合都包含 ε ，就将 ε 终结符添加到当前文法名称的 First 集合中。 此函数的函数体还不完整，留给读者完成。
Follow	求文法的 Follow 集合。 **提示：** ● 如果 A 是开始符号（第一个文法产生式的左边）那么就将终结符“$”加入 Follow（A)。 ● 给出一个非终结符 A，那么集合 Follow（A）则是由终结符组成，此外可能还有“$ ” ● 在函数的开始位置首先要调用 First 函数，求文法的 First 集合。 ● 由于对一个文法符号（非终结符）求 Follow 集合可能会依赖于其他的文法符号的 First 集合或者 Follow 集合，所以设置了一个标识 isChange，并初始化为 0，每当 Follow 集合列表中的某个集合发生变化后就将该标识赋值为 1，并将此标识作为循环条件，直到没有变化发生时，才能结束循环。 此函数的函数体还不完整，留给读者完成。
InitRules	使用给定的数据初始化文法链表。

CreateRule	创建一个新的 Rule。
CreateSymbol	创建一个新的 Symbol。
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成。

3.3 为函数 InitRules 添加注释

在 InitRules 函数之前定义了两个结构体，这两个结构体用来定义初始化文法数据的存储形式。具体内容可参见下面两个表格。

SYMBOL 的域	说明
isToken	是否为终结符。1 表示终结符，0 表示非终结符。
SymbolName	终结符和非终结符的名称。

RULE_ENTRY 的域	说明
RuleName	文法的名称。
Symbols	SYMBOL 结构体数组，其中每个元素表示一个终结符或非终结符。

为 InitRules 函数添加注释。(注意，在本程序中使用了指针的指针，体会其在单链表的插入和删除操作中的作用)。

3.4 为 PrintRule 函数编写源代码

为 PrintRule 函数编写源代码，同时理解在本程序中文法的链式存储结构，编写完源代码后，选择“调试”菜单中的“调试/开始执行（不调试)”，会在控制台窗口输出文法的产生式。

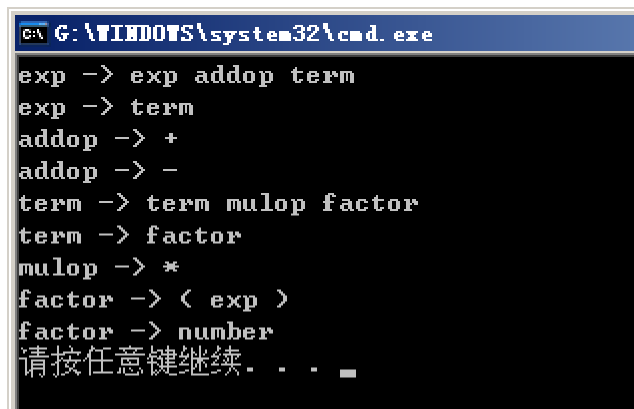

图 8-3：打印文法

3.5 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解求文法的 Follow 集合的过程。“转储信息”窗口显示的数据信息包括：

- 在求文法的 Follow 集合的过程中会使用游标来指向正在操作的文法。
- 在求文法的 Follow 集合的过程中会使用加粗的中括号表示出游标指向的 Symbol。
- First 集合列表。
- Follow 集合列表。显示每一个文法符号的 Follow 集合的构造过程。

3.6 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 Follow 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

四、思考与练习

1. 编写一个 FreeRule 函数，当在 main 函数的最后调用此函数时，可以将整个文法的内存释放掉，从而避免内存泄露。
2. 使用自己编写的代码对下面的例子进行验证，确保程序可以为所有形式的文法求 Follow 集合，并验证通过（文法中的终结符用粗体表示）。

   A -> B
   A -> **ε**
   B -> C
   B -> **ε**
   C -> AB
   C -> *****

3. 编写一个程序，此程序可以根据 First 集合和 Follow 集合判断文法是否为 LL(1)文法。
4. 编写一个程序，此程序可以根据 LL(1)文法的 First 集合和 Follow 集合构造 LL(1)分析表。
5. 编写一个程序，此程序可以使用 LL(1)分析表自动分析输入的字符串，并输出 LL(1)的分析动作和错误信息。

实验 9 Yacc 分析程序生成器

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 掌握 Yacc 输入文件的格式。
- 掌握使用 Yacc 自动生成分析程序的方法。

二、 预备知识

- 要求已经学习了 BNF（上下文无关文法），能够正确编写简单的 BNF。
- 熟练掌握了各种自底向上的分析方法，特别是 Yacc 所使用的 LALR(1)算法。
- 本实验使用 Yacc 的一个实现版本——GNU Bison 作为分析程序生成器。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab09.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 9 Yacc分析程序生成器”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab09.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

sample.txt 文件

此文件是 Yacc 的输入文件。根据 Yacc 输入文件的格式，此文件分为三个部分（由%%分隔），各个部分的说明可以参见下面的表格。

名称	说明
第一部分	在%{和%}之间的直接插入 C 源代码文件的内容。包括了要包含的头文件，以及用于指示分析程序输出调试信息的 YYDEBUG 预定义类型。
第二部分	定义了一个简单的文法规则“A -> (A) | a”，但是，对规则进行匹配时，没有执行任何操作。
第三部分	这部分中的内容会直接插入 C 源代码文件。此部分内容的说明可以参见下面的表格。

内容	说明
main 函数	在 main 函数中首先声明了 Yacc 的全局变量 yydebug，然后将其赋值为 0（0 表示不启动调试），最后调用 yyparse 函数对从标准输入读取到的内容进行分析。
yylex 函数	此函数从标准输入读取字符并进行相关的处理。首先会忽略包括空格在内的空白，然后，如果输入了回车，就返回 0，让分析程序停止，否则就返回从标准输入读取到的字符，用于和文法进行匹配。
yyerror 函数	当从标准输入读取到的字符与文法匹配失败时，会调用此函数输出错误信息。

ytab.c 文件

此文件是 Yacc 输出的 C 源代码文件。当使用 Yacc 处理 sample.txt 文件时，就会生成此文件。新建项目中，此文件的内容是空的。

ytab.h 文件

此文件是 Yacc 输出的头文件。为 Yacc 使用选项“--defines=ytab.h”时，就会生成此文件。此文件可以被包括在需要使用 Yacc 所生成的定义的任何文件中。新建项目中，此文件的内容是空的。

y.output.txt 文件

此文件是 Yacc 输出的文件。为 Yacc 使用选项“--report-file=y.output.txt”时，就会生成此文件。此文件包含了被分析程序使用的 LALR(1)分析表的文本描述。新建项目中，此文件的内容是空的。

y.output.html 文件

此文件是 y.output.txt 文件内容的 HTML 语言表示，可以使用更加直观的方式显示分析表的信息。新建项目中，此文件的内容是空的。

y.dot.txt 文件

此文件是 y.output.txt 文件内容的 DOT 语言表示，可以使用图形化的方式显示 DFA 自动机。新建项目中，此文件的内容是空的。

3.3 生成项目

按照下面的步骤生成项目：

1. 在“生成”菜单中选择“重新生成项目”（快捷键 Ctrl+Alt+F7）。

在生成的过程中，CP Lab 会首先使用 Bison 程序根据输入文件 sample.txt 来生成各个输出文件，然后，将生成的 ytab.c 文件重新编译、链接为可以运行的可执行文件。

如果成功生成了 ytab.c 文件，CP Lab 还会自动使用 DOTTY 程序来打开 y.dot.txt 文件，读者可以使用图形化的方式查看 DFA 自动机。

读者可以在“项目管理器”窗口中双击 y.output.html 文件，使用浏览器打开此文件,其内容与 y.output.txt 文件类似，但是查看更加方便直观。（注意，如果浏览器中显示乱码，需要将浏览器的编码改为“UTF-8”，简单的修改方法是在乱码页面中点击右键，选择“编码”中的“UTF-8”）。

在生成的 ytab.c 文件中，尝试找到 sample.txt 文件中第一部分和第三部分 C 源代码插入的位置，并尝试查找 yylex 函数和 yyerror 函数是在哪里被调用的。

提示：如果需要使用 DOTTY 程序手动打开 y.dot.txt 文件，需要首先在 CP Lab 的“工具”菜单中选择“Dotty”，然后在 DOTTY 程序中点击右键，选择菜单中的“load graph”，打开项目目录中的 y.dot.txt 文件。

注意：在下面的实验步骤中，如果需要生成项目，应尽量使用“重新生成项目”功能。如果习惯使用“生成项目”（快捷键 F7）功能，可能需要连续使用两次此功能才能生成最新的项目。

3.4 运行项目

在没有对项目的源代码进行任何修改的情况下，按照下面的步骤运行项目：

1、选择“调试”菜单中的“开始执行（不调试）”（快捷键 Ctrl+F5）。

在 Windows 控制台窗口中输入“(a)”字符串后按回车，扫描程序不会输出任何错误信息，如图 9-1

所示，说明文法匹配成功。而如果在 Windows 控制台窗口中输入类似“()”或“b”字符串后按回车，就会输出默认的错误信息，如图 9-2 所示。

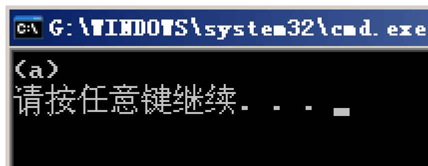

图 9-1：字符串匹配成功。

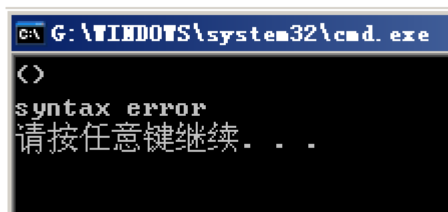

图 9-2：字符串匹配失败。

3.5 编写一个简单的计算器程序

按照下面的步骤完成此练习：

1. 修改 sample.txt 文件中的内容，实现一个简单的计算器程序，其文法如下（粗体表示终结符）：

   exp → exp addop term | term

   addop → + | -

   term → term mulop factor | factor

   mulop → *

   factor → (exp) | **number**

2. 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Ctrl+F5 启动执行项目。

执行的结果应该如图 9-3 所示，当在 Windows 控制台窗口中输入表达式“2+3”后按回车，可以正确计算出表达式的结果。

注意：

1、 本实验的模板不提供演示功能，所以，如果执行的结果不正确，可以通过添加断点和单步调试的方法来查找错误的原因。

2、 断点应该添加在 sample.txt 文件中需要中断的 C 源代码行，不要添加在 ytab.c 文件中，否则无法命中断点。

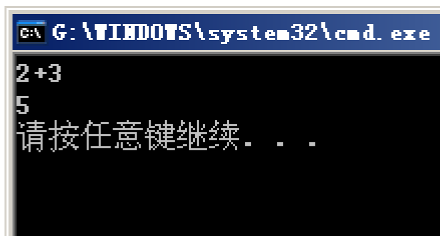

图 9-3：正确计算表达式的结果。

四、思考与练习

1. 尝试为计算器文法绘制 LALR(1)的分析表，并绘制表达式“2+3”的分析动作表。提示：将 main 函数中的 yydebug 赋值为 1 后，就可以在 Windows 控制台窗口中获得分析程序的分析动作。
2. 尝试为计算器程序添加整除运算符“/”，并可以为包含整除运算符的表达式计算出正确的值。
3. 修改计算器的 YACC 输入文件，使之能够输出以下有用的错误信息：
   - 为表达式“(2 +3”生成错误信息“丢失右括号”
   - 为表达式“2+3)”生成错误信息“丢失左括号”

- 为表达式“2 3”生成错误信息“丢失运算符”
- 为表达式“(2+)”生成错误信息“丢失操作数”

4. 选择 CP Lab“帮助”菜单中“其他帮助文档”中的“Bison 手册”，学习 Bison 工具的更多用法。如果需要修改 Bison 程序在处理输入文件时的选项，可以在“项目管理器”窗口中，使用右键点击文件“sample.txt”，在弹出的快捷菜单中选择“属性”，然后选择“属性页”左侧“自定义生成步骤”的“常规”，就可以编辑右侧的 “命令行”选项了。

实验 10 符号表的构建和使用

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 了解符号表的结构。
- 掌握符号表的插入、查找和删除等基本操作。

二、 预备知识

- 学习了符号表的概念以及作用域的基本规则。
- 在这个实验中主要用到了杂凑表（哈希表）的插入和查找等操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab10.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：

(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 10 符号表的构建和使用”;还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab10.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

SymbolTable.h 文件

主要定义了与符号表相关的数据结构，这些数据结构定义了一个符号表的单链表存储形式，允许每个作用域使用独立的符号表。其中 Symbol 结构体用于定义符号的相关信息和 Symbol 单链表，SymbolTable 结构体用于定义作用域对应的杂凑表和 SymbolTable 单链表。具体内容可参见下面两个表格。

Symbol 的域	说明
SymbolName	符号名称
SymbolType	符号类型
ClashCount	冲突次数。当一个符号名称在其作用域中被重复定义时，此计数器就要增加 1。
RefCount	引用次数。当一个符号名称在其作用域中被引用时，此计数器就要增加 1。

pNext	指向下一个 Symbol

SymbolTable 的域	说明
Bucket	杂凑表（桶）
Invalid	作用域是否无效的标志。1 表示无效，0 表示有效
pNext	指向下一个 SymbolTable

main.c 文件

首先，定义了符号表链表的头指针和符号引用失败计数器两个全局变量，然后定义了 main 函数。在 main 函数中调用了 CreateSymbolTable 函数构造符号表。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
NewSymbol	创建一个新的符号。注意，调用了 memset 函数将符号的内容清空，这与将符号结构体中的每个字段分别清空是等效的,但是使用memset函数可以编写更少的代码，执行效率更高。
NewSymbolTable	创建一个新的符号表。注意，也调用了 memset 函数将符号表的内容清空。
PushScope	在符号表链表的表头添加一个作用域。 **提示：** ● 可以调用 NewSymbolTable 函数创建符号表。 ● 添加一个作用域和入栈操作类似。 此函数的函数体还不完整，留给读者完成。
PopScope	将符号表链表中最内层的作用域设置成无效。 **提示：** ● 并不需要将符号表从符号表链表中移除，只需要将该符号表中的 Invalid 域置成 1 即可。 ● 设置一个作用域无效和出栈操作类似。 此函数的函数体还不完整，留给读者完成。
AddSymbol	向符号表中添加一个 Symbol。 **提示：** ● 每当定义一个新的变量时，会调用此函数向符号表中添加一个 Symbol。 ● 如果新定义变量的名称在其作用域中与已定义变量的名称重复，就不能添加 Symbol，而应将已定义变量的 ClashCount 域增加 1。 此函数的函数体还不完整，留给读者完成。
RefSymbol	对 Symbol 进行一次引用。 **提示：** ● 每当使用一个已定义的变量时，会调用此函数将所用变量的 RefCount 域增加 1。 ● 如果使用的变量未定义，此函数需将符号引用失败计数器（RefErrorCount）增加 1。 此函数的函数体还不完整，留给读者完成。
Hush	求 Symbol 的哈希值。
CreateSymbolTable	**提示：** ● 通过调用 PushScope 函数模拟进入作用域。

<table>
<tr><td></td><td>
● 通过调用 PopScope 函数模拟退出作用域。
● 通过调用 AddSymbol 函数模拟定义一个变量。
● 通过调用 RefSymbol 函数模拟使用一个变量。
此函数中的代码模拟了下列源代码在进入作用域、退出作用域、定义变量和使用变量时对符号表的操作。
<pre>
{
    int i, j;
    int f(); // 函数声明
    {
        char i;
        int size;
        char temp;
        {
            char* j;
            long j; // 重复定义
            j = NULL;
            i = ‘p’;
            size = f();
            new = 0; // 引用失败
        }
        j = 3;
    }
}
</pre>
</td></tr>
</table>

3.3 在演示模式下调试项目

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解符号表的构造和使用过程。“转储信息”窗口显示的数据信息包括：

- 符号引用失败次数。
- 作用域是否无效。
- 符号表链表。包括索引、项链表以及每一项的引用次数和冲突次数。
- 显示每一个符号表的构造过程。处在相同作用域的不重名变量，如果哈希值相同，则将新定义的变量插入项列表的表头。

3.4 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为函数体不完整的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

提示:如果需要通过验证，请不要修改 CreateSymbolTable 函数中的代码，只需将其他函数体补充完整即可。

四、思考与练习

1. 编写一个 FreeSymbolTable 函数，当在 main 函数的最后调用此函数时，可以将整个符号表链表的内存释放掉，从而避免内存泄露。
2. 编写一个 ScanSymbolTable 函数，当在 main 函数的最后调用此函数时，可以对整个符号表进行扫描，对于未被引用的变量发出警告。
3. 修改 PopScope 函数，不再使用符号表的无效标志，而是将作用域对应的符号表从链表头移除，并考虑此时如何对未被引用的变量发出警告。

实验 11 三地址码转换为 P-代码

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 了解三地址码和 P-代码的定义。
- 实现三地址码到 P-代码的转换。

二、 预备知识

- 对三地址码和 P-代码的定义有初步的理解，了解三地址码的四元式实现的数据结构。读者可以参考配套的《编译原理》教材，预习这一部分内容。
- 了解三地址码指令和 P-代码指令的用法和对应关系。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab11.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 11 三地址码转换为P-代码”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab11.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

T2P.h 文件

主要定义了与三地址码和 P-代码相关的数据结构。

三地址码使用了四元式的数据结构。其中 AddrKind 是枚举类型，表示地址的类型可以是空、整数常量或字符串。Address 结构体用于定义三地址码中的地址，包括地址类型以及地址中的具体值。TOpKind 是枚举类型，表示三地址码中的指令类型。TCode 结构体用于定义 1 个指令类型和 3 个地址。

P-代码的数据结构。其中也包括地址的数据结构，与三地址码不同的是 P-代码结构体中只包含一个地址。另外 P-代码拥有自己的指令集。具体内容可参见下面的表格。

Address 的域	说明
Kind	地址类型。这个域是一个枚举类型

Value	地址的整数值。地址类型为整数常量时，这个域有效
Name	地址的字符串。地址类型为字符串时，这个域有效

TCode 的域	说明
Kind	三地址码指令类型。这个域是一个枚举类型
Addr1, Addr2, Addr3	三个地址。

PCode 的域	说明
Kind	P-代码指令类型。这个域是一个枚举类型
Addr	地址

main.c 文件

定义了 main 函数。在 main 函数中首先定义了 TCodeList 和 PCodeList 两个数组，然后调用了 memset 函数将两个数组的内容清空（使用 memset 函数可以编写更少的代码，执行效率更高）。接着调用 InitTCodeList 函数初始化三地址码数组，最后调用 T2P 函数将三地址码转换为 P-代码。

在 main 函数的后面，定义了一系列函数，有关这些函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
T2P	将三地址码转换为 P-代码。 **注意：** ● 在函数中使用了 switch 语句，根据三地址码的指令类型来转换为 P-代码，所以需要了解每条三地址码指令和 P-代码指令的用途和对应关系。 此函数的函数体还不完整，留给读者完成。
InitAddress	使用给定的数据初始化三地址码。 **注意：** ● 在初始化的过程中，由于初始化结构体中统一用字符串来存储地址的值，所以当地址类型为 intconst 时需要调用 atoi 函数转换为整型。
InitCodeList	初始化三地址码列表。在此函数中会调用 InitAddress 函数。

在 InitAddress 函数之前定义了两个结构体，这两个结构体用来定义初始化三地址码的存储形式。具体内容可参见下面两个表格。

AddressEntry 的域	说明
Kind	地址的类型。这个域是一个枚举类型。
Content	地址的值。

TCodeEntry 的域	说明
Kind	三地址码指令类型。这个域是一个枚举类型。
Addr1, Addr2, Addr3	三个地址。

3.3 在演示模式下调试项目

在本实验中默认例子的源代码是：

```
read x;
x := x * 2;
if x then
    write x
end
```

这段源代码对应的三地址码是：

```
read x
t1 = x * 2
x = t1
if_false x goto L1
write x
label L1
halt
```

以上三地址码对应的四元式已经写到本实验模板的源代码中。

按照下面的步骤调试项目：

1. 按 F7 生成项目。
2. 在演示模式下，按 F5 启动调试项目。程序会在观察点函数的开始位置中断。
3. 重复按 F5，直到调试过程结束。

在调试的过程中，每执行“演示流程”窗口中的一行后，仔细观察“转储信息”窗口内容所发生的变化，理解三地址码到 P-代码的转换过程。“转储信息”窗口显示的数据信息包括：

- 三地址码（四元式）数组。使用游标来指向正在进行转换的三地址码。
- P-代码数组。显示每一条 P-代码的产生过程。

3.4 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 T2P 函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 F7 生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。
3. 按 Alt+F5 启动验证。如果验证失败，可以使用“输出”窗口中的“比较”功能，或者在“非演示模式”下按 F5 启动调试后重复按 F10 单步调试读者编写的源代码，从而定位错误的位置，然后回到步骤 1。

3.5 一个更为复杂的例子

源代码：

```
read x;
if 0 < x then
    fact := 1;
    repeat
        fact := fact * x;
        x := x - 1
    until x = 0;
    write fact
end
```

对应的三地址码：

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
```

```
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

三地址码对应的四元式:

```
(rd, x,_,_)
(gt, x, 0, t1)
(if_f, t1, L1,_)
(asn, 1, fact,_)
(lab, L2,_,_)
(mul, fact, x, t2)
(asn, t2, fact,_)
(sub, x, 1, t3)
(asn, t3, x,_)
(eq, x, 0, t4)
(if_f, t4, L2,_)
(wri, fact,_,_)
(lab, L1,_,_)
(halt,_,_,_)
```

读者可以将以上的四元式作为三地址码的初始化数据写到本实验的源代码中，并验证通过。

四、思考与练习

1. 将以下源代码的三地址码转换为P-代码，并验证通过。

```
i := 5 * 6;
j := 40 - i;
if(20 > j && 10 == j) then
    write j
end
```

2. 改进现有三地址码转换为P-代码的程序，在得到的P-代码中，尽量减少临时变量的使用。
3. 编写一个P-代码到三地址码的转换程序。

实验 12 GCC 编译器案例综合研究

实验难度：★★☆☆☆
建议学时：2 学时

一、 实验目的

- 了解 GCC 提供的 C 编译器。
- 掌握 GCC 提供的 C 编译器在 32 位 Windows 操作系统上产生的汇编代码，及 C 语言运行时环境。

二、 预备知识

- 要求已经学习了编译器生成目标机器的可执行代码的相关知识。
- 学习了基于栈和堆的运行时环境。
- 对本实验使用的 C 编译器案例——GCC 有一定的了解。

三、 实验内容

3.1 准备实验

请读者按照下面方法之一在本地创建一个项目，用于完成本次任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建个人项目，然后使用 CP Lab 提供的“从 Git 远程库新建项目”功能将个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，就需要使用 CP Lab 提供的“从 Git 远程库新建项目”功能直接将实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/cp-lab/Project-Template/Lab12.git。

建议教师按照下面的步骤在 CodeCode.net 平台上布置此次实验任务：
(1) 在编译原理实验课程中，新建一个实验任务。
(2) 在“新建任务”页面中，教师需要添写“任务名称”，推荐使用“实验 12 GCC编译器案例综合研究”；还需要填写“模板项目URL”，应该使用“https://www.codecode.net/engintime/cp-lab/Project-Template/Lab12.git”。
(3) 点击“新建任务”按钮，完成新建任务操作。

3.2 阅读实验源代码

本实验模板创建的项目中包括了两个文件：case.h 和 case.c。

由于本实验主要关注于编译器产生的汇编代码，使用的 C 代码都非常简单，所以在后面的实验步骤中再对相关的源代码进行介绍。

3.3 案例 1——基于栈的函数调用

按照下面的步骤完成此练习：

1. 在 case1 函数唯一的代码行添加一个断点。
2. 按 F7 生成项目。
3. 按 F5 启动调试项目。此时程序会在刚刚添加的断点处中断。
4. 选择“调试”菜单“窗口”中的“反汇编”，打开“反汇编”窗口。

在打开的“反汇编”窗口中，显示了 case1 函数的反汇编代码，如下面的代码清单所示（为了方便对代码进行说明，添加了行号）：

```
1) push   ebp
2) mov    ebp,esp
3) sub    esp,0x18
4) mov    DWORD PTR [esp+0x4],0x2
5) mov    DWORD PTR [esp],0x1
6) call   0x401359 <add>
7) mov    DWORD PTR [ebp-0x4],eax
8) leave
9) ret
```

下面对汇编代码进行详细说明：

1. 32 位 Windows 操作系统一般运行在 32 位的 Intel x86 处理器上，此时使用 32 位的寄存器。为了与 Intel 8086 处理器上 16 位寄存器的名称相区分，在 16 位寄存器名称的前面增加了字母“e”。例如，ax 是 16 位寄存器的名称，相应的 eax 是 32 位寄存器的名称。
2. 第 1 行和第 2 行代码初始化了 case1 函数的栈，如图 12-1 所示，此时 ebp 和 esp 指向相同的地址。由于 ebp 是 32 位寄存器，所以当其压入栈后会占用 4 个字节（深色表示已经压入栈的字节）。注意，这里用到的栈是从高地址向低地址方向生长，esp 总是指向栈顶元素，ebp 总是指向当前函数的“栈底”，esp 和 ebp 之间的栈空间用来存放当前函数的临时变量。
3. 如前所述，第 3 行代码在栈上开辟了一个 24（0x18）字节的空间，用来存放当前函数的临时变量，此时的栈如图 12-2 所示。
4. 第 4 行和第 5 行代码将调用 add 函数所需的两个参数按照从右到左的顺序压入到了栈中，如图 12-3 所示。其中，在 esp+0x4 两边加中括号表示的是这个地址处的内存，而 DWORD PTR 表示这个内存的大小是 4 字节。由于第 3 行代码已经开辟了栈空间，所以这里直接使用了 mov 指令，请读者练习使用 push 指令改写这两行代码，并适当修改第 3 行代码，使栈增长的空间仍然为 24 个字节。

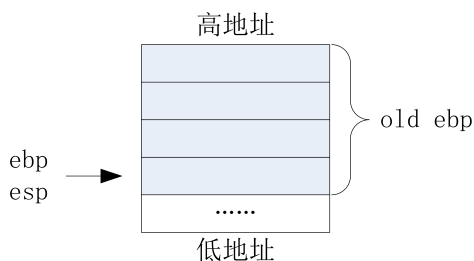

图 12-1：初始化栈。

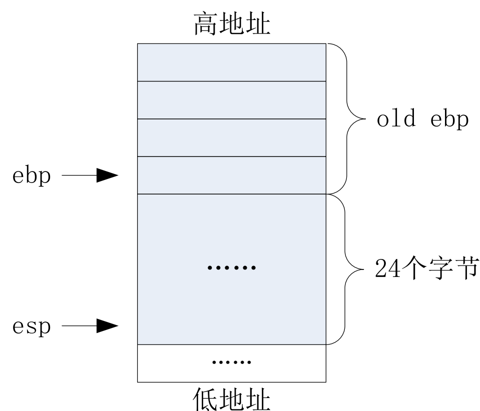

图 12-2：开辟栈空间。

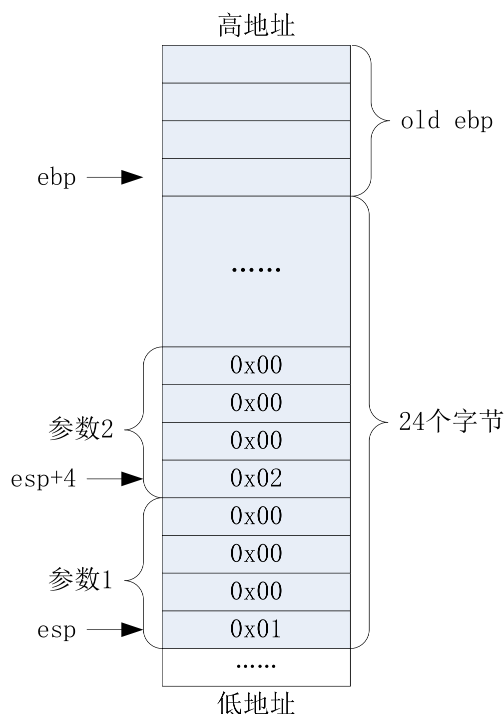

图 12-3：将参数压入栈。

5. 第 6 行代码使用 call 指令调用 add 函数。指令 call 会首先将 add 函数返回后继续执行的地址（也就是第 7 行代码的地址）压入栈，然后跳转到 add 函数第一条指令的地址开始执行。请读者记录下 call 指令跳转的地址以及第 7 行代码所在的地址，留待后面进行验证。

按照下面的步骤继续此练习：

1. 按 F11 单步进入 add 函数，会在 add 函数内中断，并在“反汇编”窗口中显示 add 函数的反汇编代码（已经执行了第 1 行和第 2 行代码），参见下面的代码清单。此时的栈如图 12-4 所示。
2. 在“反汇编”窗口中查看 add 函数第一条指令的地址，验证是否与之前记录的地址相同。
3. 选择“调试”菜单中的“快速监视”，打开“快速监视”窗口。在“表达式”控件中输入“$ebp”后按回车（必须有开始的$符号），记录下此时 ebp 寄存器指向的地址。在“表达式”控件中重新输入“* ((long*) (ebp 指向的地址 + 4))”即可查看由 call 指令压入栈中的返回地址，验证是否与之前记录的地址相同。

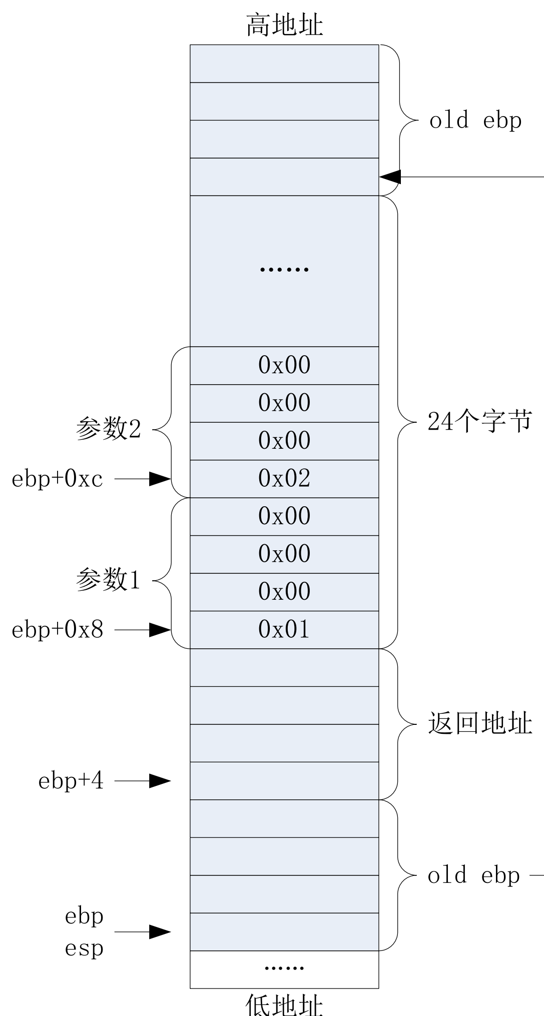

图 12-4：进入 add 函数后的栈。

```
1) push    ebp
2) mov     ebp, esp
3) mov     eax, DWORD PTR [ebp+0xc]
4) add     eax, DWORD PTR [ebp+0x8]
5) pop     ebp
6) ret
```

下面继续对汇编代码进行详细说明：

1. add 函数的第 1 行和第 2 行代码初始化了栈。在第 1 行压入栈中的 ebp 就是之前 case1 函数的“栈底”，在图 12-4 中使用了一个箭头表示了这种链接关系；第 2 行将 epb 指向了 add 函数的“栈底”。
2. 第 3 行将参数 2 的值复制到了累加寄存器 eax 中。
3. 第 4 行将参数 1 的值累加到了 eax 中，并且使用 eax 作为函数的返回值。
4. 第 5 行从栈中弹出了 ebp，这会让 ebp 重新回到图 12-3 中所示的位置。
5. 第 6 行的 ret 指令从栈中弹出了函数的返回地址到指令计数器（eip 寄存器）中，这样，整个栈就恢复到图 12-3 所示的样子了，而且会从 case1 函数反汇编代码的第 7 行继续执行。
6. case1 函数的第 7 行将 add 函数的返回值从 eax 寄存器复制到临时变量 i 中。由于 int 类型占用 4 个字节，所以，ebp-4 的位置就是临时变量 i 在栈中的位置。 请读者在图 12-3 中添加临时变量 i 在栈中的位置。
7. case1 函数第 8 行的 leave 指令与下面的两条指令

```
mov esp,ebp
pop ebp
```

完全等价，其中第一条指令负责展开栈，包括释放了传递给 add 函数的参数。第 9 行的 ret 指令会让程序返回到 main 函数中继续执行。

至此，可以对 C 语言函数调用做如下总结：

1. 使用栈传递参数，并按照从右到左的顺序将参数压入栈。
2. 使用 eax 寄存器保存函数的返回值。
3. 由调用者负责展开栈，并释放参数占用的栈空间。

3.4 案例 2——数组元素的引用

按照下面的步骤完成此练习：

1. 按 Shift+F5 结束之前的调试。
2. 按 Ctrl+Shift+F9 删除之前添加的所有断点。
3. 在 case2 函数第一行代码处添加一个断点。
4. 按 F5 启动调试项目。此时程序会在刚刚添加的断点处中断。
5. 选择“调试”菜单“窗口”中的“反汇编”，打开“反汇编”窗口。

在打开的“反汇编”窗口中，显示了 case2 函数的反汇编代码，如下面的代码清单所示（为了方便对代码进行说明，添加了行号）：

```
1)  push    ebp
2)  mov     ebp,esp
3)  sub     esp,0x28
4)  mov     DWORD PTR [ebp-0xc],0x1
5)  mov     DWORD PTR [ebp-0x28],0x6
6)  mov     DWORD PTR [ebp-0x24],0x7
7)  mov     eax,DWORD PTR [ebp-0xc]
8)  mov     DWORD PTR [ebp+eax*4-0x24],0x8
9)  leave
10) ret
```

下面对汇编代码进行详细说明：

1. 第 3 行代码在栈上开辟了一个 40（0x28）字节的空间，用来存放当前函数的临时变量，包括变量 i 和数组 a。
2. 第 4 行代码将变量 i 的值赋值为 1。
3. 第 5 行和第 6 行代码分别为 a[0]和 a[1]赋值。
4. 第 7 行代码将变量 i 的值复制到 eax 寄存器中。
5. 第 8 行代码使用公式 ebp-0x28+(eax+1)*4 计算数组元素的地址并完成赋值操作。

请读者参考案图 12-3，绘制出案例 2 中栈的图示，并在栈中标示出临时变量 i 的位置和数组 a 中各个元素的位置。

3.5 案例 3——结构体中域的引用

按照下面的步骤完成此练习：

1. 按 Shift+F5 结束之前的调试。
2. 按 Ctrl+Shift+F9 删除之前添加的所有断点。

3. 在 case3 函数第一行代码处添加一个断点。
4. 按 F5 启动调试项目。此时程序会在刚刚添加的断点处中断。
5. 选择“调试”菜单“窗口”中的“反汇编”，打开“反汇编”窗口。

在打开的“反汇编”窗口中，显示了 case3 函数的反汇编代码，如下面的代码清单所示（为了方便对代码进行说明，添加了行号）：

```
1)  push    ebp
2)  mov     ebp,esp
3)  sub     esp,0x18
4)  mov     DWORD PTR [ebp-0x10],0x1
5)  mov     eax,DWORD PTR [ebp-0x10]
6)  mov     BYTE PTR [ebp-0x14],al
7)  mov     eax,DWORD PTR [ebp-0x10]
8)  mov     DWORD PTR [ebp-0x18],eax
9)  leave
10) ret
```

下面对汇编代码进行详细说明：

1. 第 3 行代码在栈上开辟了一个 24（0x18）字节的空间，用来存放当前函数的临时变量 x。域 x.i 的大小是 4 字节，对应的地址是 ebp-0x18；域 x.c 的大小是 1 字节，按 4 字节对齐后大小也是 4 字节，但是只有最低的一个字节有效，对应的地址是 ebp-0x14；域 x.j 的大小是 4 字节，对应的地址是 ebp-0x10。
2. 第 4 行代码将 x.j 的值赋值成 1。
3. 第 5 行和第 6 行代码将 x.j 的值赋值给 x.c。注意，第 6 行代码只是将 eax 最低的一个字节 al 的值赋值给了 x.c 占用的一个字节（由 BYTE PTR 指定为一个字节）。
4. 第 7 行和第 8 行代码将 x.j 的值赋值给 x.i。

请读者参考案图 12-3，绘制出案例 3 中栈的图示，并在栈中标示出临时变量 x 的各个域的位置。

3.6 案例 4——指针的引用

按照下面的步骤完成此练习：

1. 按 Shift+F5 结束之前的调试。
2. 按 Ctrl+Shift+F9 删除之前添加的所有断点。
3. 在 case4 函数第一行代码处添加一个断点。
4. 按 F5 启动调试项目。此时程序会在刚刚添加的断点处中断。
5. 选择“调试”菜单“窗口”中的“反汇编”，打开“反汇编”窗口。

在打开的“反汇编”窗口中，显示了 case4 函数的反汇编代码，如下面的代码清单所示（为了方便对代码进行说明，添加了行号）：

```
1)  push    ebp
2)  mov     ebp,esp
3)  sub     esp,0x8
4)  mov     DWORD PTR [esp],0xc
5)  call    0x40195c <malloc>
6)  mov     DWORD PTR [ebp-0x4],eax
7)  mov     edx,DWORD PTR [ebp-0x4]
```

```
8)  mov     eax,DWORD PTR [ebp-0x4]
9)  mov     DWORD PTR [edx+0x4],eax
10) mov     eax,DWORD PTR [ebp-0x4]
11) mov     eax,DWORD PTR [eax+0x8]
12) mov     DWORD PTR [ebp-0x4],eax
13) leave
14) ret
```

下面对汇编代码进行详细说明：

1. 第 3 行代码在栈上开辟了一个 8 字节的空间，用来存放当前函数的临时变量 p 和传给 malloc 函数的参数。
2. 第 4 行代码将传给 malloc 函数的参数 0xc（TreeNode 结构体的大小为 12 个字节）放入栈顶。
3. 第 5 行代码调用 malloc 函数。
4. 第 6 行代码将 malloc 函数的返回值从 eax 寄存器中复制到临时变量 p 中，这样变量 p 就指向了由 malloc 函数在堆中刚刚分配的 12 个字节的基址。堆中的 12 个字节是这样分配的：域 val 占用了最低的 4 个字节，域 lchild 占用了中间的 4 个字节，域 rchild 占用了最高的 4 个字节。此时的堆和栈如图 12-5 所示。

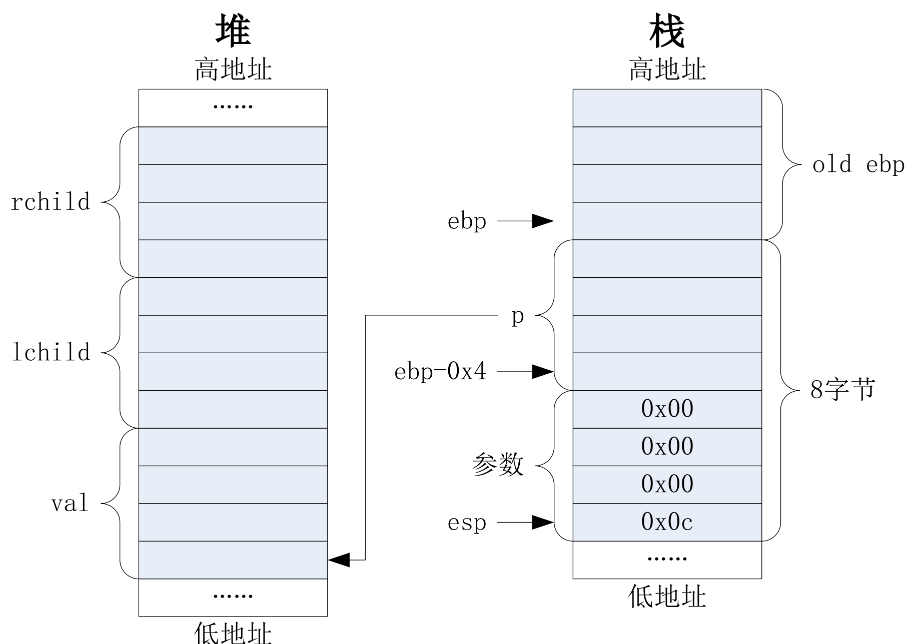

图 12-5：调用 malloc 函数后的堆和栈。

5. 第 7 行将变量 p 的值赋值给了 edx，这样 edx 也就指向了堆中 12 字节的基址。第 8 行又将变量 p 的值赋值给了 eax。第 9 行使用 edx 为基址，加 4 后就是 lchild 在堆中的位置，从而完成了将变量 p 的值赋值给 p->lchild，此时 p->lchild 也就指向了堆中 12 字节的基址。
6. 与之前的代码类似，第 10、11、12 行将 p->rchild 的值赋值给变量 p，此时 p 就不再指向之前的位置了，而是指向了一个随机的位置（因为 p->rchild 没有初始化）。

3.7 案例 5——if 语句

按照下面的步骤完成此练习：

1. 按 Shift+F5 结束之前的调试。
2. 按 Ctrl+Shift+F9 删除之前添加的所有断点。
3. 在 case5 函数第一行代码处添加一个断点。
4. 按 F5 启动调试项目。此时程序会在刚刚添加的断点处中断。

5. 选择“调试”菜单“窗口”中的“反汇编”，打开“反汇编”窗口。

在打开的“反汇编”窗口中，显示了 case5 函数的反汇编代码，如下面的代码清单所示（为了方便对代码进行说明，保留了每行指令相对于函数起始地址的偏移）：

```
<case5+0>:  push   ebp
<case5+1>:  mov    ebp,esp
<case5+3>:  sub    esp,0x8
<case5+6>:  mov    eax,DWORD PTR [ebp-0x4]
<case5+9>:  cmp    eax,DWORD PTR [ebp-0x8]
<case5+12>: jle    0x4013e5 <case5+21>
<case5+14>: lea    eax,[ebp-0x8]
<case5+17>: inc    DWORD PTR [eax]
<case5+19>: jmp    0x4013ea <case5+26>
<case5+21>: lea    eax,[ebp-0x4]
<case5+24>: dec    DWORD PTR [eax]
<case5+26>: leave
<case5+27>: ret
```

下面对汇编代码进行详细说明：

1. 偏移为 3 的代码在栈上开辟了一个 8 字节的空间，用来存放当前函数的临时变量 x 和 y，其中变量 x 的位置是 ebp-0x4，变量 y 的位置是 ebp-0x8。
2. 偏移为 6 和 9 的代码比较 x 和 y。
3. 偏移为 12 的代码行判断当 x 小于等于 y 时，就跳转执行偏移为 21 和 24 的代码行，完成 x--操作后从函数中返回。相当于源代码中的 else 分支。
4. 如果 x 大于 y，就会继续执行偏移为 14 和 17 的代码行，完成 y++操作。然后在执行偏移为 19 的代码行，跳转到偏移为 26 的代码行后从函数中返回。

四、思考与练习

1. 在案例 1 中研究的是 C 语言默认的函数调用约定（__cdecl），C 语言函数还可以使用其他的调用约定，例如常用的 stdcall、fastcall 等。尝试将 add 函数分别修改为 int __stdcall add(int i, int j)和 int __fastcall add(int i, int j)，从而显式指定函数的调用约定（注意关键字的前缀是双下划线），然后通过研究不同调用约定下 case1 函数和 add 函数的反汇编代码，总结出这三种函数调用约定的异同,并尝试说明哪种调用约定支持可变参数。
2. 编写一个递归函数，在递归调用的过程中，结合其反汇编代码说明在发生递归时栈的构造与展开的过程。
3. 使用下面的函数，结合其反汇编代码说明二维数组中的元素是如何被引用的。

```
void case6()
{
    int i = 1;
    int a[3][3];
    a[0][0] = 6;
    a[1][1] = 7;
    a[1][i + 1] = 8;
}
```

4. 使用下面的函数，结合其反汇编代码说明联合中的域是如何被引用的。

```
typedef struct _Cub
```

```
{
    int i;
    union {
        char c;
        int j;
    }u;
}Cub;
void case7()
{
    Cub x;
    x.i = 1;
    x.u.c = x.i;
    x.u.j = x.i;
}
```

5. 使用下面的函数，结合其反汇编代码说明指针的指针是如何被引用的。

```
void case8()
{
    TreeNode* p = (TreeNode*)malloc(sizeof(TreeNode));
    TreeNode** pp = &p;
    (*pp)->val = 1;
}
```

6. 使用下面的函数，结合其反汇编代码说明 while 语句是如何跳转的。

```
void case9()
{
    int x, y;
    while(x<y)
        y-=x;
}
```

7. 验证 GCC 提供的 C 编译器是否实现了短回路布尔操作。

附录 1 TINY 编译器和 TM 机

一、TINY 语言

TINY 语言的程序结构很简单，它在语法上与 Ada 或 Pascal 的语法相似：仅是一个有分号分隔开的语句序列。另外，它既无过程也无声明。所有的变量都是整型变量，通过对其赋值可较轻易地声明变量。它只有两个控制语句：if 语句和 repeat 语句，这两个控制语句本身也可包含语句序列。If 语句有一个可选的 else 部分且必须由关键字 end 结束。除此之外，read 语句和 write 语句完成输入和输出。在花括号中可以有注释，但注释不能嵌套。

TINY 的表达式也局限于布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式的比较组成，该比较使用 < 与 = 比较算符。算术表达式可以包含整数常量、变量、参数以及 4 个整型算符+、-、*、/，此外还有一般的数学属性。布尔表达式可能只作为测试出现在控制语句中—而没有布尔变量、赋值或 I/O。

下面是该语言中的一个阶乘函数的简单程序示例。

```
{ Sample program
  int TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
    fact := 1;
    repeat
        fact := fact * x;
        x := x - 1
    until
    x = 0;
    write fact { output factorial of x }
end
```

二、TINY 编译器

TINY 编译器包含以下的 C 文件：

```
globals.h   main.c
util.h      util.c
scan.h      scan.c
parse.h     parse.c
symtab.h    symtab.c
analyze.h   analyze.c
code.h      code.c
cgen.h      cgen.c
```

任何代码文件都包含 globals.h 头文件，它包含了数据类型的定义和整个编译器均使用了全局变量。main.c 文件包括运行编译器的主程序，它还分配和初始化全局变量。其他的文件包含了头/代码文件对、在头文件中给出了外部可用的函数原型以及相关代码文件中的实现（包括静态局部变量）。scan、parse、analyze 和 cgen 文件分别对应扫描程序、分析程序、语义分析程序和代码生成器各个阶段。util 文件包括了实用程序函数，生成源代码（语法树）的内部表示和显示列表与出错信息均需要这些函数。symtab 文

件包括执行与 TINY 应用相符的符号表的杂凑表。code 文件包括用于依赖目标机器（依赖目标机器在这里指 TM 机，会在稍后做具体介绍）的代码生成的实用程序。

虽然这些文件的交互少了，但是编译器仍有 4 遍：第 1 遍由构造语法树的扫描程序和分析程序组成；第 2 遍和第 3 遍执行语义分析，其中第 2 遍构造符号表而第 3 遍完成类型检查；最后一遍是代码生成器。

CP Lab 可编译 TINY 编译器，生成的可执行文件是 tiny，通过使用以下命令：

tiny sample.txt

就可用 TINY 编译器编译文本文件 sample.txt 中的 TINY 源程序。并将目标代码输出到 sample.tm.txt 文件中(在下面将谈到的 TM 机中使用)。

具体的操作可以遵循以下的步骤：

1. 打开CP Lab在“文件”菜单中选择“新建”，然后单击“从 Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。使用 https://www.codecode.net/engintime/cp-lab/Project-Template/TINY.git 新建一个项目。
2. 在“项目管理器”窗口中除 TINY 编译器包含的 C 文件以外，在 TINY 语言样例筛选器下还包含了 sample.txt 和 sample.tm.txt 两个文件，其中 sample.txt 文件中是一个从标准输入中读取数据，计算其阶乘后标准输出的TINY 语言程序。sample.tm.txt 文件用于保存编译后生成的目标代码(TM 机中使用)。
3. 按 F7 生成项目。
4. 按 F5 启动调试，这时可以看到在 sample.tm .txt 文件中生成了目标代码。

三、TM 机

TM 机的汇编语言作为 TINY 编译器的目标语言。TM 机的指令仅够作为诸如 TINY 这样的小型语言的目标。实际上 TM 具有精简指令集计算机（RISC）的一些特性。在 RISC 中，所有的算法和测试均须在寄存器中进行，而且地址，地址模式极为有限。为了使读者了解到该机制的简便之处，我们将下面的 C 表达式的代码

a[index] = 6

翻译成 TM 汇编语言

```
LDC 1, 0 (0)    load 0 into reg 1
下面的指令假设 index 在存储器地址 10 中
LD 0, 10 (1)    load val at (10+R1 into R0
LDC 1, 2, (0)   load 2 into reg 1
MUL 0, 1, 0     put R1*R0 into R0
LDC 1, 0, (0)   load 0 into reg 1
下面的指令假设 a 在存储器地址 20 中
LDA 1, 20, (1)  load 20+R1 into R0
ADD 0, 1, 0     put R1+R0 into R0
LDC 1, 6, (0)   load 6 into reg 1
ST 1, 0, (0)    store R1 at 0+R0
```

装入操作中有 3 个地址模式并且是由不同的指令给出的：LDC 是“装入常量”LD 是“由存储器装入”，而 LDA 是“装入地址”。另外，该地址通常必须给成“寄存器+偏差”值。例如“10（1)”，它代表将偏差 10 加到寄存器 1 的内容中计算该地址。(因为在前面的指令中，0 已经被装入到寄存器 1 中，这实际是指绝对位置 10)。我们还看到算数指令 MUL 和 ADD 可以是“三元”指令且只有寄存器操作数，其中可以单独确定结果的目标寄存器。

TM 机的模拟程序直接从一个文件中读取汇编代码并执行它，因此应避免将由汇编语言翻译为机器代码的过程复杂化。但是，这个模拟程序并非一个真正的汇编程序，他并没有符号地址或标号。因此，TINY 编译器必须仍然计算跳转的绝对地址。此外为了避免与外部的输入/输出例程连接的复杂性，TM 机有内部整型的 I/O 设备；在模拟时，它们都对标准设备读写。

CP Lab 可以将 tm.c 源代码编译成 TM 模拟程序，并运行 TM 指令。具体的操作可以遵循以下的步骤：

1. 打开CP Lab在“文件”菜单中选择“新建”，然后单击“从Git 远程库新建项目”，打开“从 Git 远程库新建项目”对话框。使用 https://www.codecode.net/engintime/cp-lab/Project-Template/TM.git 新建一个项目。
2. 在“项目管理器”窗口中， 双击 sample.tm.txt 文件可以看到这个文件就是 TINY 编译器由 sample.txt 源文件生成的目标代码文件。
3. 按 F7 生成项目。
4. 按 F5 启动调试，CP Lab 会自动启动 TM 机执行 sample.tm.txt 文件中的程序。按以下计算可以得到 7 的阶乘：

```
TM simulation(enter h for help)…
Enter command: go
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0, 0, 0
Halted
Enter command: quit
Simulation done.
```

参考文献

[1] Kenneth C. Louden．Compliler Construction :Principles and Practice． 冯博琴，冯岚等译. 编译原理及实践（第 1 版）. 北京：机械工业出版社，2009
[2] 陈火旺，刘春林等编著. 程序设计语言编译原理（第 3 版）. 北京：国防工业出版社，2011
[3] 张菁 .编译原理及实践（中英双语版）. 北京：清华大学出版社，2007
[4] Andrew W Appel . 现代编译原理 C 语言描述. 北京：人民邮电出版社，2006
[5] 王雷，刘志成，周晶. 编译原理课程设计.北京：机械工业出版社，2005.
[6] 王晓斌. 程序设计语言与编译. 北京：电子工业出版社，2015
[7] 吴春香. 编译原理——习题与解析. 北京：清华大学出版社，2006
[8] 赵炯著。Linux 内核完全剖析。北京：机械工业出版社，2006
[9] [英] Peter Abel 著。IBM PC 汇编语言程序设计（第五版）。沈美明，温冬婵译。北京：人民邮电出版社，2002
[10]刘星等著。计算机接口技术。北京：机械工业出版社，2003
[11]唐朔飞著。计算机组成原理。北京：高等教育出版社，2000
[12][希腊] Diomidis Spinellis 著。代码阅读方法与实践。赵学良译。北京：清华大学出版社，2004
[13][美] 科学、工程和公共政策委员会著。怎样当一名科学家。何传启译。北京：科学出版社，1996
[14]Engintime CP Lab 帮助文档。北京英真时代科技有限公司。http://www.engintime.com/node/27,2008
[15]Intel Co. INTEL 80386 Programmes's Refercence Manual 1986,INTEL CORPORATION,1987
[16]Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3:System Programming Guide. http://www.intel.com/,2005
[17]Microsoft Co. FAT: General Overview of On-Disk Format. MICROSOFT CORPORATION,1999
[18]IEEE-CS，ACM. Computing Curricula 2001 Computer Science. 2001
[19]The NASM Development Team. NASM — The Netwide Assembler. Version 2.04 2008
[20]Bochs simulation system. http://bochs.sourceforge.net/