

系统介绍编译与反编译理论、技术与工具  
作者多年科研、工程、教学经验的总结与融合

# 编译与反编译 技术实战

庞建民 主编

刘晓楠 陶红伟 岳峰 戴超 编著



COMPILING  
AND  
DECOMPILING  
IN PRACTICE



机械工业出版社  
China Machine Press

# 编译与反编译 技术实战

庞建民 主编

刘晓楠 陶红伟 岳峰 戴超 编著



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

编译与反编译技术实战 / 庞建民主编. —北京: 机械工业出版社, 2017.5  
(信息安全技术丛书)

ISBN 978-7-111-56617-5

I. 编… II. 庞… III. 计算机网络—安全技术 IV. TP393.08

中国版本图书馆 CIP 数据核字 (2017) 第 070557 号

本书从编译与反编译两个角度出发, 按照以实践为主线、以理论为辅助、两者结合的原则展开讲述。在编译部分, 从正向角度, 按照编译过程的步骤, 从词法分析器的设计与实现方面的实践入手, 逐步按自上而下语法分析器和自下而上语法分析器的设计与实现等环节展开, 而语义分析与处理实践、优化与目标代码生成实践等环节则在对 GCC 和 LLVM 等具体编译器剖析时展开论述; 同时, 结合信息安全问题对多样化编译实践进行论述; 然后, 从反向角度介绍反编译实现的相关原理与技术, 包括反汇编器的设计与实现、控制流图恢复的实现、关键行为的恢复及其在恶意代码检测中的应用等。

本书可作为软件编程、信息与网络空间安全或者软件逆向分析工作的工程技术人员的参考书, 也可作为计算机与信息安全相关专业高年级本科生或研究生的教学参考书。

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余 洁

责任校对: 殷 虹

印 刷: 北京瑞德印刷有限公司

版 次: 2017 年 5 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 23

书 号: ISBN 978-7-111-56617-5

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

“编译技术”是从事软件开发和信息安全相关工作的技术人员必须掌握的基础性技术，也是高等院校计算机科学与技术专业的一门必修专业课，这是理论与实践结合非常强的领域，对提升开发人员的技术水平和大学生科学思维的养成、解决实际问题能力具有重要作用。“反编译技术”则是近几年发展起来的新兴技术，许多计算机软件或信息安全从业者非常关心该技术的发展，但目前这方面的书籍较少，与“编译技术”结合起来讲解的书也很少，从实践角度来剖析的更是少见。本书就是在这种需求以及作者在这两方面的科研实践的驱动下诞生的，目的是为计算机软件和信息安全从业者提供编译与反编译技术方面的知识和实战技巧。

本书的编写得到了解放军信息工程大学和机械工业出版社的大力支持，在此表示诚挚的谢意。本书中的一些材料来自本书主编主持的国家自然科学基金（项目编号：61472447）、国家“863”（项目编号：2006AA01Z408）、国家重大专项某子课题等项目的研究成果，在此对这些课题的支持表示衷心的感谢！

本书是机械工业出版社2016年4月出版的《编译与反编译技术》（ISBN 978-7-111-53412-9）一书的姊妹篇，配合学习和使用效果更佳。在本书中，作者着力阐述编译与反编译技术及实战方面的相关知识和实战技巧，力图使用通用的语言讲述抽象的原理、技术和实战技能，但限于作者水平，书中难免有错误与欠妥之处，恳请读者批评指正。

作者

2017年3月



# 目 录 *Contents*

前言	2.4 本章小结	13
第1章 实践的环境与工具	第3章 词法分析器的设计与实现	14
1.1 实践环境概述	3.1 词法分析器的设计	14
1.2 词法分析生成器 LEX	3.1.1 词法分析器的功能	14
1.3 语法分析生成器 YACC	3.1.2 输入及其处理	15
1.4 编译器 GCC	3.2 词法分析器的手工实现	16
1.5 编译器 LLVM	3.3 词法分析器的 LEX 实现	31
1.6 反汇编工具 IDA	3.3.1 LEX 源文件结构	32
1.7 反汇编工具 OllyICE	3.3.2 LEX 系统中的正规式	34
1.8 仿真与分析工具 QEMU	3.3.3 LEX 的使用方式	36
1.9 动态分析工具 TEMU	3.3.4 LEX 源文件示例——C 语言	
1.10 本章小结	词法分析器	37
第2章 编译器实践概述	3.4 本章小结	41
2.1 编译器、解释器及其工作方式	第4章 语法分析器的设计与实现	42
2.2 编译器的结构	4.1 自上而下的语法分析器的	
2.3 编译器的设计与实现概述	设计与实现	42
2.3.1 利用 Flex 和 Bison 实现词法	4.2 自下而上的语法分析器的	
和语法分析	设计与实现	61
2.3.2 利用 LLVM 实现代码优化和	4.3 语法分析器的生成器	72
代码生成	4.3.1 YACC 的源文件结构	72

4.3.2	YACC 和 LEX 的接口 .....	76	6.4.5	LLVM IR 代码生成 .....	100
4.3.3	YACC 源程序示例—— 简单的台式计算器 .....	77	6.5	LLVM 的中间表示 .....	100
4.4	本章小结 .....	78	6.5.1	LLVM IR 语法 .....	102
<b>第5章</b>	<b>GCC编译器分析与实践 .....</b>	<b>79</b>	6.5.2	LLVM IR 优化实例 .....	104
5.1	GCC 编译器概述 .....	79	6.6	LLVM 后端 .....	106
5.2	GCC 编译器的系统结构 .....	80	6.6.1	后端库文件 .....	107
5.3	GCC 编译器的分析程序 .....	81	6.6.2	LLVM 目标架构描述文件 .....	108
5.4	GCC 编译器的中间语言及其 生成 .....	82	6.7	应用实例 .....	109
5.5	GCC 编译器的优化 .....	82	6.7.1	代码插桩 .....	110
5.6	GCC 编译器的目标代码生成 .....	87	6.7.2	代码保护 .....	110
5.7	本章小结 .....	88	6.8	本章小结 .....	111
<b>第6章</b>	<b>LLVM编译器分析与实践 .....</b>	<b>89</b>	<b>第7章</b>	<b>多样化编译实践 .....</b>	<b>112</b>
6.1	LLVM 编译器概述 .....	89	7.1	软件多样化的机会 .....	112
6.1.1	起源 .....	89	7.1.1	应用层的多样化机会 .....	112
6.1.2	相关项目 .....	90	7.1.2	Web 服务层的多样化机会 .....	113
6.2	经典编译器概述 .....	91	7.1.3	操作系统层的多样化机会 .....	115
6.2.1	经典编译器设计的启示 .....	91	7.1.4	组合后的多样化机会 .....	116
6.2.2	现有编译器的实现 .....	92	7.1.5	虚拟层的多样化机会 .....	116
6.3	LLVM 的设计 .....	93	7.2	多样化带来的管理复杂性 .....	117
6.3.1	LLVM 中间表示 .....	94	7.3	多样化编译技术 .....	118
6.3.2	LLVM 库文件 .....	95	7.3.1	随机化技术 .....	118
6.4	LLVM 前端 .....	96	7.3.2	代码混淆技术 .....	120
6.4.1	前端库文件 .....	97	7.3.3	与堆栈相关的多样化技术 .....	123
6.4.2	词法分析 .....	97	7.4	多样化编译的应用 .....	125
6.4.3	语法分析 .....	99	7.4.1	多样化编译在安全防御方面 的应用 .....	126
6.4.4	语义分析 .....	100	7.4.2	多样化编译工具的结构组成 及原理 .....	127
			7.5	本章小结 .....	128



**第8章 反编译的对象——可执行****文件格式分析 ..... 129****8.1 可执行文件格式 ..... 129**

## 8.1.1 PE 可执行文件格式 ..... 129

## 8.1.2 ELF 可执行文件格式 ..... 130

**8.2 main 函数的识别 ..... 133**

## 8.2.1 程序启动过程分析 ..... 136

## 8.2.2 startup 函数解析 ..... 137

## 8.2.3 main() 函数定位 ..... 140

**8.3 本章小结 ..... 142****第9章 反编译的基础——指令****系统和反汇编 ..... 143****9.1 指令系统概述 ..... 143**

## 9.1.1 机器指令及格式 ..... 145

## 9.1.2 汇编指令及描述 ..... 147

**9.2 指令解码 ..... 149**

## 9.2.1 SLED 通用编解码语言 ..... 149

## 9.2.2 x64 的 SLED 描述 ..... 154

## 9.2.3 IA64 的 SLED 描述 ..... 159

**9.3 反汇编过程 ..... 161**

## 9.3.1 线性扫描反汇编 ..... 161

## 9.3.2 行进递归反汇编 ..... 162

**9.4 反汇编工具 IDA 与 OllyICE****实践 ..... 163**

## 9.4.1 IDA 实践 ..... 163

## 9.4.2 OllyICE 实践 ..... 166

**9.5 本章小结 ..... 169****第10章 反编译的中点——从汇编****指令到中间表示 ..... 170****10.1 中间代码生成在经典反编译器****中的实际应用 ..... 170**

## 10.1.1 低级中间代码 ..... 171

## 10.1.2 高级中间代码 ..... 172

**10.2 中间表示从设计到应用的****具体实例 ..... 175**

## 10.2.1 指令基本组件描述 ..... 176

## 10.2.2 用 UMSDL 描述指令语义 ..... 179

**10.3 本章小结 ..... 184****第11章 反编译的推进1——数据****类型恢复 ..... 185****11.1 基本数据类型的分析和恢复 ..... 185**

## 11.1.1 数据类型分析的相关概念 ..... 186

11.1.2 基于指令语义的基本数据  
类型分析 ..... 18811.1.3 基于过程的数据类型分析  
技术 ..... 190**11.2 函数类型恢复 ..... 197**

## 11.2.1 问题引入 ..... 198

## 11.2.2 函数类型的恢复 ..... 198

**11.3 本章小结 ..... 203****第12章 反编译的推进2——控制流****恢复实例 ..... 205****12.1 基于关键语义子树的间接跳转****目标解析 ..... 205**

## 12.1.1 问题的提出 ..... 206

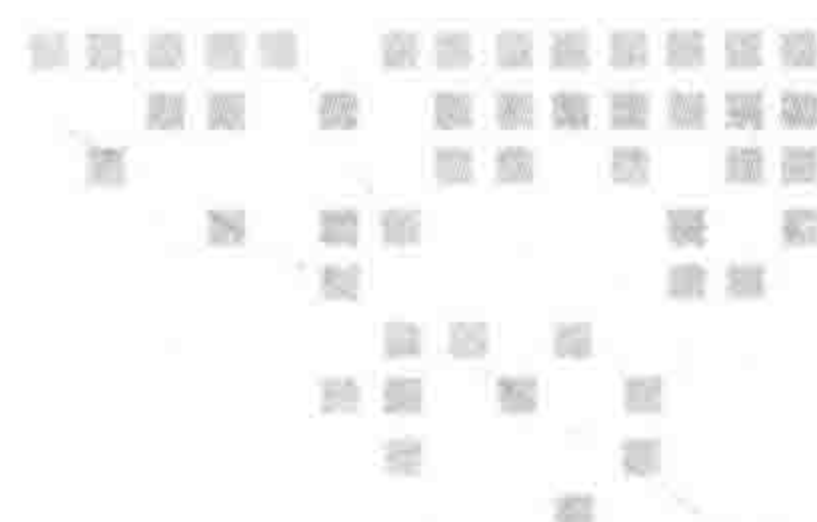
## 12.1.2 相关工作 ..... 207



12.1.3	跳转表的语义特征 .....	208	13.2.4	库函数参数的恢复 .....	262
12.1.4	基于关键语义子树的间接 跳转目标解析及翻译 .....	210	13.2.5	隐式库函数调用 .....	267
12.2	功能块概念的引入 .....	222	13.3	用户自定义过程的数据恢复 .....	279
12.2.1	分析单位 .....	222	13.3.1	基于语义映射的数据 恢复 .....	280
12.2.2	基于基本块的分析 .....	223	13.3.2	基于栈帧平衡的数据 恢复 .....	282
12.2.3	功能块 .....	228	13.4	用户函数与库函数同名的区分 .....	283
12.2.4	针对功能块的验证 .....	230	13.4.1	函数同名问题 .....	283
12.3	基于功能块的间接转移指令 目标地址的确定 .....	233	13.4.2	函数同名解决实例 .....	286
12.3.1	程序控制流图构建方法中 存在的问题 .....	233	13.5	本章小结 .....	290
12.3.2	无法处理的代码 .....	234	<b>第14章 反编译在信息安全方面的 应用实践 .....</b>		<b>291</b>
12.3.3	程序执行路径的逆向构造 .....	235	14.1	反编译在信息安全中的应用 .....	291
12.3.4	逆向构造执行路径的 控制执行 .....	245	14.1.1	反编译技术的优势 .....	292
12.3.5	针对程序执行路径逆向 构造的验证 .....	246	14.1.2	代码恶意性判定 .....	292
12.4	本章小结 .....	248	14.1.3	代码敏感行为标注 .....	293
<b>第13章 反编译的推进3——过程 定义恢复 .....</b>		<b>250</b>	14.1.4	恶意代码威胁性评估 .....	293
13.1	过程分析概述 .....	250	14.2	反编译在恶意代码分析中的 应用 .....	294
13.1.1	过程抽象 .....	250	14.2.1	基于文件结构的恶意代码 分析 .....	294
13.1.2	调用约定分析 .....	251	14.2.2	基于汇编指令的恶意代码 分析 .....	295
13.2	库函数恢复 .....	255	14.2.3	基于流图的恶意代码 分析 .....	296
13.2.1	快速库函数调用识别方法 .....	255	14.2.4	基于系统调用的恶意 代码分析 .....	298
13.2.2	基于特征数据库的模式 匹配方法 .....	256	14.3	恶意代码与反编译技术的对抗 .....	300
13.2.3	基于函数签名的库函数 识别方法 .....	257	14.3.1	混淆 .....	300



14.3.2	多态 .....	304	14.4.3	子程序异常返回 .....	319
14.3.3	变形 .....	306	14.4.4	不透明谓词混淆 .....	324
14.3.4	加壳 .....	307	14.5	实例分析 .....	330
14.3.5	虚拟执行 .....	308	14.5.1	系统设计 .....	330
14.4	反编译框架针对恶意行为的 改进 .....	309	14.5.2	系统模块划分 .....	331
14.4.1	条件跳转混淆 .....	309	14.5.3	测试结果与分析 .....	351
14.4.2	指令重叠混淆 .....	314	14.6	本章小结 .....	355
			参考文献 .....		356



## 实践的环境与工具

本书致力于通过实践及案例，从正反向两个角度介绍编译系统的一般构造原理和基本实现技术，本章首先对书中内容涉及的环境与工具进行简单介绍，这些工具都是编译与反编译过程中常用的工具。

### 1.1 实践环境概述

在编译过程中所涉及的环境主要是编译环境及工具链，常用的工具有词法分析生成器、语法分析生成器、编译器、汇编器、链接器等。在反编译过程中主要涉及反汇编器、静态或动态的调试与分析工具。下面对近年来流行的编译与反编译工具逐一进行简单介绍。

### 1.2 词法分析生成器 LEX

词法分析是编译过程的第一个阶段，其任务就是将输入的各种符号转化成相应的标识符号，转化后的标识符很容易被后续阶段处理。

LEX 是 LEXical compiler 的缩写，是 UNIX 环境下非常著名的工具，主要功能是生成一个词法分析器的 C 源码，描述规则采用正则表达式。描述词法分析器的文件 \*.l 经过 LEX 编译后生成一个 lex.yy.c 的文件，然后由 C 编译器编译生成一个词法分析器。

LEX 接收用户输入的正则表达式，识别这些表达式并且将输入流转化为匹配这些表达式的字符串。在这些字符串的分界处，用户提供的程序片段被执行。LEX 代码文件将正则表达式和程序片段关联，将每一条输入对应到由 LEX 生成的程序的表达式，且执行相应的



代码片段。

为了完成任务，除了需要提供匹配的表达式以外，用户还需要提供其他代码，甚至是由其他生成器产生的代码。用户提供一般程序设计语言的代码片段完成程序识别表达式后的工作。因此，用户自由编写动作代码时，并不影响其编写高级语言代码来匹配字符串表达式。这就避免迫使用户使用字符串语言来进行输入分析时，也必须使用同样的方法来编写字符处理程序，而这样做有时是不合适的。LEX 不是完整的语言，它是一个新语言的生成器，可以插入各种不同的被叫作“宿主语言”的程序设计语言中。就像大多数高级语言可以生成在不同计算机硬件上运行的代码一样，LEX 可以生成不同的宿主语言。宿主语言用于 LEX 生成输出代码，也用于用户插入程序片段，这使得 LEX 适用于不同的环境和不同的使用者。每一个应用程序可以是硬件、适用于该任务的宿主语言、用户背景和局部接口属性的直接结合。现在，LEX 唯一支持的宿主语言是 C，过去也支持过其他语言。Fortran LEX 自身一般运行于 UNIX 或 Linux 系统之上，但是 LEX 生成的代码可以在任何适当的编译器上使用。

LEX 将用户输入的表达式和动作代码转换为宿主语言，生成的函数一般名为 `yylex`。`yylex` 识别字符流中的表达式，并且当每一个表达式被检测出来后，输出相应的动作。

LEX 的文件结构简单，分为三个部分：

```
declarations
%%
translation rules
%%
auxiliary procedures
```

分别是声明段、规则段和辅助部分。

1) 声明段包括变量、符号常量和正则表达式的声明。希望出现在目标 C 源码中的代码，用“`%{...%}`”括起来。比如：

```
%{
#include <stdio.h>
#include "y.tab.h"
typedef char * YYSTYPE;
char * yylval;
%}
```

2) 规则段是由正则表达式和相应的动作组成的。如

```
[0-9]+          printf("Int      : %s/n",yytext);
[0-9]*/[.][0-9]+  printf("Float   : %s/n",yytext);
```

“`[0-9]+`”是描述整数的正则表达式，“`[0-9]*/[.][0-9]+`”是描述浮点的正则表达式，后面的 `printf` 即为它们对应的动作。

值得注意的是，LEX 依次尝试每一个规则，尽可能地匹配最长的输入流，即规则部分



具有优先级的概念。比如下面的规则部分

```
%%
A      {printf("run");}
AA     {printf("like ");}
AAAA   {printf("I ");}
%%
```

对于内容“AAAAAAA”，LEX 程序会输出“I like run”，首先匹配最长的 4 个“A”，之后在剩下的三个“A”中匹配两个“A”，直到最后的一个“A”。可以看出 LEX 的确按照最长的规则匹配。

3) 辅助部分放一些扫描器的其他模块，可以包含用 C 语言编写的子程序，而这些子程序可以用在前面的动作中，这样就可以达到简化编程的目的。辅助部分也可以在另一个程序文件中编写，最后再链接到一起。生成 C 代码后，需用 C 的编译器编译，链接时需要指定链接库。

本书的第 3 章将更加详细地介绍 LEX 及其用法。需要说明的是，对于 GNU/Linux 用户，与 UNIX 环境中 LEX 对应的工具是 Flex，其具体用法和 LEX 相似，这里不再赘述。

### 1.3 语法分析生成器 YACC

语法分析的任务是分析句子是否符合语法规则。YACC (Yet Another Compiler Compiler) 是一个经典的语法分析生成器。YACC 最初是由 AT&T 公司的 Steven C. Johnson 为 UNIX 操作系统开发的，后来一些兼容的程序如 Berkeley YACC、GNU Bison、MKS YACC 和 Abraxas YACC 陆续出现，它们都是在此基础上做了少许改进或者增强，但是基本概念是相同的。现在 YACC 也已普遍移植到 Windows 及其他平台上。

语法分析生成器是一个指定某个格式中的一种语言的语法作为它的输入，并为该语言产生分析过程以作为它的输出的程序。在历史上，语法分析生成器被称作编译-编译程序，这是由于按照规律可将所有的编译步骤作为包含在语法分析程序中的动作来执行。现在的观点是将语法分析程序仅考虑为编译处理的一个部分，所以这个术语也就有些过时了。YACC 迄今为止仍是常用的语法分析生成器之一。

作为 YACC 对说明文件中的“%token NUMBER”声明的对应，YACC 坚持定义所有的记号本身，而不是从别的地方引入一个定义，但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。

YACC 的输入是巴科斯范式 (BNF) 表达的语法规则以及语法归约的处理代码，YACC 输出的是基于表驱动的编译器，包含输入的语法归约的处理代码部分。

由于所产生的解析器需要词法分析器的配合，因此 YACC 经常和词法分析器的产生器 (通常就是 LEX) 联合使用，把两部分产生的 C 程序一并编译。IEEE POSIX P1003.2 标准



定义了 LEX 和 YACC 的功能和需求。

需要指出的是，本书的第 4 章还有对 YACC 及其用法更加详细的介绍。

## 1.4 编译器 GCC

GCC (GNU Compiler Collection, GNU 编译器套件) 是由 GNU 开发的编程语言编译器。它是以 GPL 许可证发行的自由软件，也是 GNU 计划的关键部分。GCC 原本作为 GNU 操作系统的官方编译器，现已被大多数类 UNIX 操作系统（如 Linux、BSD、Mac OS X 等）采纳为标准的编译器，GCC 同样适用于微软的 Windows。

GCC 原名为 GNU C 语言编译器 (GNU C Compiler)，因为它原本只能处理 C 语言。随着 GCC 的快速扩展，其可支持 C++，后来又能够支持更多编程语言，如 Fortran、Pascal、Objective-C、Java、Ada、Go 以及各类处理器架构上的汇编语言等，所以改名为 GNU 编译器套件。

### (1) 前端接口

前端将高级语言源码经过词法分析、语法分析生成与高级语言无关的低级中间层表示 GENERIC，然后经过单一化赋值转化为另一种中间表示层 GIMPLE，在中间层 GIMPLE 组建控制流程图，并在 GIMPLE 上进行一系列优化。然后将其转换为更加便于优化的 RTL 中间表示层。有了与前端无关的中间表示，GCC 的前端将不同的高级编程语言转换成这种中间表示，这就是 GCC 处理器支持多种编程语言的根本原因。

### (2) 中间接口

中间接口主要在 RTL 中间表示上进行各种优化，GCC 的优化技巧根据版本不同而有很大不同，但都包含了标准的优化算法，如循环优化、公共子表达式删除、指令重排序等。更多的优化方法也在不断地研究中。

### (3) 后端接口

GCC 后端对每条 RTL 通过模板匹配的方法调用对应的汇编模板生成汇编代码。生成的代码因处理器和结构不同而不同，GCC 后端为不同的平台提供了描述指令的汇编模板文件，这样就可以实现对不同平台的支持。这个阶段非常复杂，因为必须要考虑 GCC 可移植平台的处理器指令集的规格与技术细节，解决指令选择和寄存器分配等问题。

### (4) GCC 常用的参数

在使用 GCC 编译器的时候，必须给出一系列必要的调用参数和文件名称。GCC 编译器的调用参数大约有 100 多个，这里只介绍其中最基本、最常用的参数。具体细节内容可参考 GCC Manual。

GCC 最基本的用法是：

```
gcc [options] [filenames]
```

其中 options 就是编译器所需要的参数（也称为编译选项），filenames 给出相关的文件名称。



-c：只编译，不链接成为可执行文件，编译器只是由输入的.c等源代码文件生成以.o为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output\_filename：确定输出文件的名称为output\_filename，同时这个名称不能与源文件同名。如果不给出这个选项，GCC就给出预设的可执行文件a.out。

-g：产生符号调试工具（GNU的gdb）所必要的符号信息，要想对源代码进行调试，必须加入这个选项。

-O：对程序进行优化编译、链接。采用这个选项，整个源代码会在编译、链接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是编译、链接的速度相应地要慢一些。

-O2：比-O更好的优化编译、链接，当然整个编译、链接过程会更慢。

-v：GCC执行时执行的详细过程、GCC及其相关程序的版本号。编译程序时加上该选项可以看到GCC搜索头文件/库文件时使用的搜索路径。

## 1.5 编译器 LLVM

LLVM是构架编译器的框架系统，由C++编写而成，用于优化以任意程序语言编写的程序的编译时间、链接时间、运行时间以及空闲时间，对开发者保持开放，并兼容已有脚本。LLVM计划启动于2000年，最初由伊利诺伊大学香槟分校的Chris Lattner主持开展。2006年Chris Lattner加盟Apple公司并致力于LLVM在Apple开发体系中的应用。Apple公司也是LLVM计划的主要资助者。

LLVM的命名最早源自于Low Level Virtual Machine（底层虚拟机）的缩写，由于命名带来的混乱，目前LLVM就是该项目的全称。LLVM核心库提供了与编译器相关的支持，可以作为多种语言编译器的后台来使用，能够进行程序语言的编译期优化、链接优化、在线编译优化、代码生成。LLVM的项目是一个模块化和可重复使用的编译器和工具技术的集合。LLVM提供一个现代化的、基于SSA的编译策略，能够同时支持静态和动态的任意编程语言的编译目标。至今为止，LLVM已被应用到许多商业和开源的项目，并被广泛用于学术研究。

LLVM荣获2012年ACM软件系统奖。

对关注编译技术的开发人员，LLVM提供了很多优点：

1) 现代化的设计。LLVM的设计是高度模块化的，使得其代码更为清晰和便于排查问题所在。

2) 语言无关的中间代码。一方面，这使得通过LLVM能够将不同的语言相互联结起来，也使得LLVM能够紧密地与IDE交互和集成。另一方面，发布中间代码而非目标代码能够在目标系统上更好地发挥其潜能而又不影响可调试性（比如，在目标系统上针对本机的硬件环境产生目标代码，但又能够通过中间代码来进行行级调试）。



3) 可作为工具和函数库。使用 LLVM 提供的工具可以比较容易地实现新的编程语言的优化编译器或虚拟机, 或为现有的编程语言引入一些更好的优化 / 调试特性。

## 1.6 反汇编工具 IDA

IDA 是 IDA PRO 的简称, 是英文 Interactive Disassembler 的缩写。它是一个世界顶级的交互式反汇编工具, 其使用者覆盖了软件安全专家、军事工业和国家安全信息部门、逆向工程学者、黑客等。它是由 HEX-RAYS 公司开发的, 这是一家多年从事将二进制代码反编译到 C 的软件安全公司, 其旗舰产品就是著名的 Hex-Rays.Decompiler (IDA 的插件)。

IDA 有两种可用版本。标准版支持 20 多种处理器, 高级版支持 50 多种处理器。IDA 不存在任何注册机、注册码或破解版, 除了测试版和一个 4.9 的免费版外, 网络上能下载的都是包含用户许可证的正版, 因为所有的安装包都是 OEM 版, 所以 IDA 官网不提供软件下载, 并且软件也没有注册的选项 (完全可以正常使用, 当然这也是一种盗版或侵权的行为, 对此 IDA 公司会采取严厉打击措施)。

IDA 的界面比其他反汇编工具界面更加专业, 它提供了很多选项并允许用户自己编写插件。它的优点是更好地反汇编和进行更深层的分析, 而缺点是使用更困难。

## 1.7 反汇编工具 OllyICE

OllyICE 是一种具有可视化界面的 32 位汇编分析调试器, 是一个动态追踪调试工具, 利用 IDA 与 SoftICE 结合的思想在 Ring3 级上进行调试, 容易上手, 已代替 SoftICE 成为当今最流行的调试解密工具, 同时还支持插件扩展功能, 是目前最强大的动态调试工具。

## 1.8 仿真与分析工具 QEMU

QEMU 是一套由 Fabrice Bellard 所编写的以 GPL 许可证分发源码的模拟处理器, 在 GNU/Linux 平台上使用广泛。Bochs、PearPC 等与其类似, 但不具备其许多特性, 比如高速度及跨平台的特性, 通过 KQEMU 这个闭源的加速器, QEMU 能模拟至接近真实计算机的速度。

目前, 0.9.1 及之前版本的 QEMU 可以使用 KQEMU 加速器。在 QEMU 1.0 之后的版本都无法使用 KQEMU, 但可利用 qemu-kvm 加速模块, 并且加速效果以及稳定性明显好于 KQEMU。

QEMU 有以下两种主要运作模式:

1) 进程级模拟模式。在这种模式下, QEMU 能以二进制翻译的方式启动那些为不同处理器编译的 Linux 可执行程序。典型的程序是 Wine 及 Dosemu。



2) 系统级模拟模式。QEMU 能模拟整个计算机系统, 包括中央处理器及其他周边设备。它使得为跨平台编写的程序进行测试及除错工作变得容易。也能用来在一台主机上虚拟数台不同虚拟计算机。

该软件的优点有:

- 默认支持多种架构。可以模拟 IA 32、AMD 64、MIPS、SPARC、PowerPC、龙芯等多种架构。
- 可扩展, 可自定义新的指令集。
- 开源, 可移植, 仿真速度快。
- 在支持硬件虚拟化的 x86 架构上可以使用 KVM 加速配合内核 ksm 大页面备份内存, 速度稳定, 远超过 VMware ESX。
- 可以在其他操作系统平台上运行 Linux 程序。
- 可以存储及还原运行状态。
- 可以支持虚拟网卡等多种外设。

该软件的缺点有:

- 对微软视窗及某些主机操作系统的支持不完善(某些模拟的系统仅能运行)。
- 对不常用的架构的支持不完善。
- 除非使用 KQEMU 加速器, 否则其模拟速度仍不及其他虚拟软件, 如 VMware。
- 比其他模拟软件难安装及使用。

## 1.9 动态分析工具 TEMU

TEMU 是动态分析工具 BitBlaze 的一个组件, 是一个基于系统仿真器 QEMU 开发的动态二进制分析工具, 以 QEMU 为基础运行一个完整的系统(包括操作系统和应用程序), 并对二进制代码的执行进行跟踪和分析。

TEMU 提供以下功能:

1) 动态污点分析。TEMU 能够对整个系统进行动态污点分析, 把一些信息标记为污点(如键盘事件、网络输入、内存读写、函数调用、指令等), 并在系统内进行污点传播。这个特性可为符号执行提供插件形式的工具。许多分析都需要对二进制代码进行细粒度的分析, 而基于 QEMU 的全系统模拟器确保了细粒度的分析。

2) 获取操作系统视图。操作系统中提取的信息如进程和文件对很多分析都是很重要的。TEMU 可以使用这些信息决定当前执行的是哪个进程和模块、调用的 API 和参数, 以及文件的存取位置。全系统的视图使我们能够分析操作系统内核以及多个进程间的交互, 而许多其他的二进制分析工具(如 Valgrind、DynamoRIO、Pin)只提供了一个局部的视图(如单个进程的信息)。这对于分析恶意代码更为重要, 因为许多攻击涉及多个进程, 而且诸如 rootkits 的内核攻击变得越来越普遍。



3) 深度行为分析。TEMU 能够分析二进制文件和操作环境的交互, 如 API 调用序列、边界内存位置的访问。通过标记输入为污点, TEMU 能够进行输入和输出之间的关系分析。并且, 全系统仿真器有效地隔离了分析组件和待分析代码。因此, 待分析代码更难干扰分析结果。

TEMU 由 C 和 C++ 实现。性能要求高的代码由 C 实现, 而面向分析的代码由 C++ 编写, 以便很好地利用 C++ STL 中的抽象数据类型和类型检查。

## 1.10 本章小结

本章简要介绍了阅读本书所需要的实践环境, 主要有词法分析生成器 LEX、语法分析生成器 YACC、编译器 GCC 和 LLVM、反汇编工具 IDA 和 OllyICE、仿真与分析工具 QEMU、动态分析工具 TEMU 等。在第 1 章简单介绍这些工具, 主要是希望读者在开始时就能够在自己的机器上安装这些工具, 并能够使用这些工具进行一些简单的实验, 某些重要的工具将在后面详细介绍。

## 编译器实践概述

人与计算机之间的交流也是通过语言进行的，但人类能理解的语言与机器可以理解的语言是不同的，中间需要翻译，因此，相应的编译器诞生了。编译技术所讨论的问题就是如何把符合人类思维方式的意愿（即源程序）翻译成计算机能够理解和执行的形式（即目标程序），而实现从源程序到目标程序转换的程序被称为编译程序或编译器。最早的编译器是 20 世纪 50 年代后期的 Fortran 编译器，该编译器也为后续高级语言和编译器的涌现奠定了基础。与编译技术相反，反编译技术所讨论的问题就是如何把计算机能够理解和执行的形式（目标程序）翻译成符合人类便于理解的形式（源程序或流程图），实现从目标程序到便于人类理解的系列文档的转换程序被称为反编译程序或反编译器。反编译技术起源于 20 世纪 60 年代，虽然在时间上只比编译技术晚 10 年左右，但反编译技术的成熟度却远不如编译技术。半个世纪以来，也涌现了不少实验性的反编译器，如 Dcc、Boomerang 和 IDA 的反编译插件 Hex\_rays 等。但这些反编译器都有这样或那样的缺陷，还不能像编译器那样强健。

本章仅对编译器实践方面的知识进行简要阐述，反编译实践方面的概要介绍将在后续章节给出。

### 2.1 编译器、解释器及其工作方式

就目前计算机的硬件发展水平而言，硬件只能识别由 0、1 字符串组成的机器指令序列，即机器指令程序或目标程序。在计算机发明的早期，计算机只能按照输入的机器指令程序进行简单的计算。但是，机器指令程序不易被人类理解，用它编写程序不仅困难而且还容



易出错。于是后来就引入了代替 0、1 字符串的由助记符号表示的指令，即汇编指令，汇编指令的集合被称为汇编语言，汇编指令序列被称为汇编语言程序。汇编程序实际上与机器语言程序是一一对应的，都要求程序员按照指令工作的方式来思考 and 解决相关问题，也就是说，两者之间并无本质区别。因此，它们都被称为面向机器的语言或低级语言，以此与更高级的语言相区别，当然，早期并不知道高级别的语言是否能设计和实现。

计算机的发展和普及超乎人们的想象，应用需求的大量增长导致程序员的需求也大幅增长，但是，能够用机器语言或汇编语言编程的人员数量却不多，满足不了这种需求；同时，许多不同领域的科技工作者也想自己动手编写程序来直接解决问题。因此，抽象度更高、功能更强的语言来作为程序设计语言就成为必然，于是就产生了面向各类应用的便于人类理解与运用的程序设计语言，即高级语言。尽管人类可以借助高级语言来编写程序，但计算机硬件真正能够识别和理解的语言还是 0、1 组成的机器语言，这就需要在高级语言与机器语言之间建立转换系统，使得高级语言能够自动转换为机器语言。也就是说需要若干“翻译”，把各类高级语言翻译成机器语言。程序设计语言通常被分成三个层次：高级语言、汇编语言、机器语言。高级语言可以翻译成机器语言，也可以翻译成汇编语言，这两种翻译都被称为编译。汇编语言到机器语言的翻译称为汇编。编译和汇编属于正向工程，有时还需要将机器语言翻译成汇编语言或高级语言，这通常被称为反汇编或反编译，属于逆向工程范畴。

高级语言的工作方式有两种，一种是编译器工作方式，另一种是解释器工作方式。

在编译器工作方式下，源程序的翻译和翻译后程序的运行处于两个相互独立的阶段。用户输入源程序，编译器对该源程序进行编译，生成目标程序，这个阶段称为编译阶段。目标程序在适当的输入下执行，最终得到运行结果的过程称为运行阶段。

解释器是另一种形式的翻译器。它把翻译和运行结合在一起进行，边翻译源程序，边执行翻译结果，而这种工作方式被称为解释器工作方式。

形象地说，编译器的工作相当于翻译一本原著，原著与源程序对应，译著与目标程序对应，计算机的运行相当于阅读一本译著，这时，原著和翻译人员并不需要在场，译著是主角。解释器的工作相当于在进行现场翻译，外宾和翻译都要在场，翻译一边听外宾讲话，一边翻译给听众，翻译是听众关注的主角。解释器与编译器的最本质区别是：运行目标程序时的控制权在解释器而不是目标程序，也就是说，运行的是解释器，目标程序是解释器的输入。

## 2.2 编译器的结构

目前常用的程序设计语言都已经有很多优秀的编译器，比如 C 语言有 GCC 和 ICC、C++ 有 G++ 和 I++、Java 有 JAVAC 和 GCJ。然而，即使这些常用的程序设计语言，其本身也一直在改变，即不断地完善。因而，实现这些程序设计语言的编译器也需要做出相应的



改动。对于程序设计语言自身的改变，有的是为了弥补自身的一些缺陷，如 Java 语言从设计至今，其体积已经增大了好几倍；有的是为了适应新的软件开发需求，比如为了更容易地开发大型软件等而进行的改善。

除了那些成熟语言的改动会带来编译器软件编程的需要外，新语言的诞生也需要程序员来完成新语言的编译器实现工作。比如，现在不断涌现的各种脚本语言都需要编译器程序员来编写这些语言的编译器或解释器。对于新语言的发明，有的是为了适应特殊领域的编程需要，比如 SQL (Structured Query Language) 是为关系数据库管理系统专门设计的专用语言；有的是为了更好地利用各种系统资源（尤其是硬件资源），比如 OpenCL (Open Computing Language) 是为了更好地开发异构平台的计算能力。作为高级语言到目标代码的翻译软件（或者不同语言间的翻译软件）的编译器，对它的编程需求一直都存在。也就是说，总有实现新的编译器或者改动现有编译器的需求存在。

不同的编译器虽然实现方式各异，但编译器的结构却非常相似，通常是按照编译过程的各个阶段来实现相关的程序模块。编译过程中每个阶段工作的逻辑关系如图 2-1 所示，图中的每个阶段的工作由相关程序模块承担，但其中的符号表管理程序和错误处理程序则贯穿编译过程的各个阶段。这些程序模块构成了编译器的基本结构。

通常，编译的阶段又被分成前端和后端两部分。前端是由只依赖于源语言的那些阶段或阶段的一部分组成，往往包含词法分析、语法分析、语义分析和中间代码生成等阶段，当然还包括与这些阶段同时完成的错误处理和独立于目标机器的优化。后端是指编译器中依赖于目标机器的部分，往往只与中间语言有关而独立于源语言。后端包括与目标机器相关的代码优化、代码生成和与这些阶段相伴的错误处理和符号表操作。这种前后端的划分使得编译器的设计更加清晰、合理与高效。

基于同一个前端，重写其后端就可以产生同一种源语言在另一种机器上的编译器，这是为不同类型机器编写编译器的常用做法。反过来，把几种不同的语言编译成同一种中间语言，使得不同的前端都使用同一个后端，进而得到一类机器上的几个编译器，却只取得了有限的成功，其原因在于不同源程序语言的区别较大，使得包容它们的中间语言庞大臃肿，难以得到高效率。但是，在反编译的过程中，设计一种中间语言，将不同体系结构的目标代码先翻译成这种中间语言代码，再由这种中间代码反编译为 C 代码，则是一种较为有效的途径。

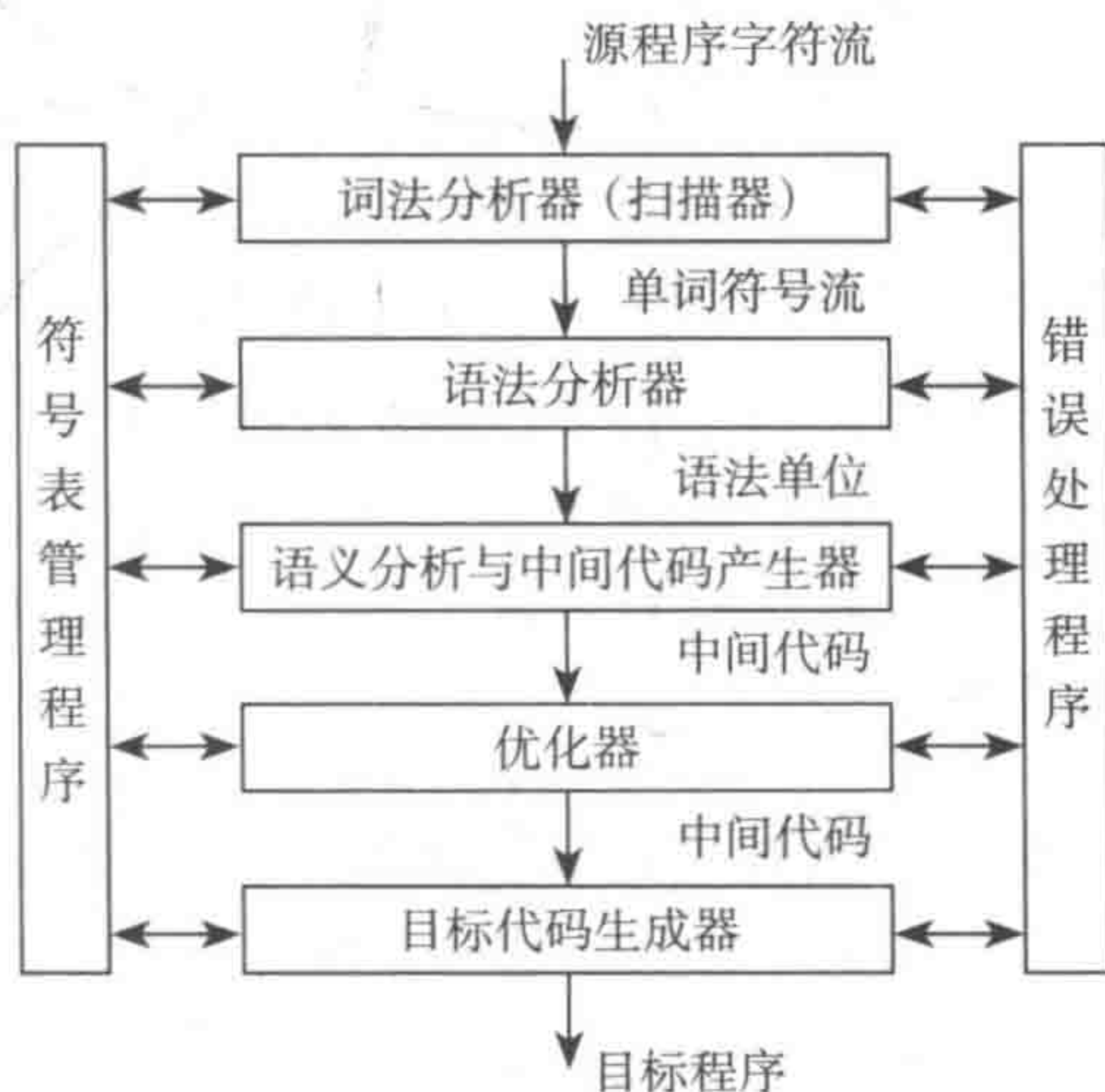


图 2-1 编译器结构图



## 2.3 编译器的设计与实现概述

根据不同的用途和侧重点，编译程序可以进一步分类，换句话说，有许多不同种类的编译器变体。譬如：用于帮助程序开发和调试的编译程序称为诊断编译程序，这类编译器可对程序进行详细检查并报告错误；另一类侧重于提高目标代码效率的编译程序称为优化编译程序，这类编译器通常使用多种混合的“变换”来改善程序的性能，但这往往是以编译器的复杂性和编译时间的增加为代价的。通常，将运行编译程序的机器成为宿主机，将运行编译程序所产生的目标代码的机器称为目标机。如果一个编译程序产生不同于其宿主机指令集的机器代码，则称它为交叉编译程序（Cross Compiler）。还有一类编译器，其目标机器可以改变，而不需要重写它的与机器无关的组件，这类编译器称为可再目标编译器（Retargetable Compiler），通常，这类编译器难以生成高效的代码，因为其难以利用特殊情况和目标机器特性。目前，很多编译程序同时提供了调试、优化、交叉编译等多种功能，用户可以通过“编译选项”进行选择。

编译器本身也是一个程序，这个程序是怎么编写的呢？早期人们使用汇编语言编写编译器。虽然用汇编语言编写的编译器代码效率很高，但由于汇编语言编程与高级语言编程相比难度较大，对编译器这种复杂的系统编写起来效率不高，因此，后来人们改用高级语言来编写编译器。随着编译技术的逐步成熟，一些专门的编译器编写工具相继涌现，比较成熟和通用的工具有词法分析器生成器（如 LEX）和语法分析器生成器（如 YACC）等。还有一些工具，如用于语义分析的语法制导翻译工具、用于目标代码生成的自动的代码生成器、用于优化的数据流工具等。下面简单介绍利用一些工具实现一个新的语言编译器的基本流程。

### 2.3.1 利用 Flex 和 Bison 实现词法和语法分析

在 UNIX 环境中编写程序，你往往会邂逅神秘的 LEX 和 YACC，而 GNU/Linux 用户则会邂逅 Flex 和 Bison。

Flex 是一个与 LEX 兼容的词法分析器生成器，可以用它来生成一个新的语言的词法分析器，Flex 就是由 Vern Paxson 实现的一个 LEX，使用它既可以节省时间，也可以提高正确性。

Bison 是一个与 YACC 兼容的语法分析器生成器，可以用它来生成一个新的语言的语法分析器，使用它也可以提高正确性并节省开发时间，实际上，Bison 是一个可以把符合 LALR(1) 文法规范的上下文无关文法转换成 C 语言程序的语法分析器生成器，是一个 GNU 版本的 YACC。

实现一种新语言，需要做的工作主要包括设计文法、进行语法制导的翻译、优化和代码生成，而后续的工作还可以由 LLVM 的相关工具提供支持。

### 2.3.2 利用 LLVM 实现代码优化和代码生成

LLVM 是一个包含一系列模块化可重用编译器和工具链技术的项目。LLVM 主要的



子项目有 LLVM Core libraries、Clang、Dragonegg、LLDB 等。其中 LLVM Core libraries (LLVM 核心库) 提供了一个不依赖于目标平台的优化器, 同时还为许多典型架构的 CPU 提供了代码生成的支持。这些库是围绕着一个有详细说明的中间代码表示形式 (LLVM IR) 建立起来的。也就是说, 只要能够把待设计的语言翻译成 LLVM IR 这种中间语言, 就可以利用 LLVM 完成代码优化和代码生成的工作。当然, 这要求目标 CPU 架构必须是 LLVM 已经支持的, 否则就得自己完成代码生成的工作。Clang 是一个 LLVM 自身的 C/C++/Objective-C 编译器, 目标是提供快速的编译。

Dragonegg 的功能是把 LLVM 的优化器、代码生成器和 GCC 4.5 的分析器结合在一起, 这样就使得 LLVM 能够编译像 Ada、Fortran 等其他 GCC 编译器前端支持的语言, 且能够拥有一些 Clang 不支持的 C 特性 (如 OpenMP 等)。LLDB 则是建立在 LLVM 库和 Clang 之上的一个非常好的本地调试器。

## 2.4 本章小结

本章首先介绍了编译器和解释器的概念及其工作方式, 然后剖析了编译器的结构, 对编译器的设计与实现进行了阐述, 并对 Flex 和 Bison 进行了简要概述, 对 LLVM 及其应用进行了简要介绍, 最后给出了基于现有工具的编译器的实现流程。



## 词法分析器的设计与实现

词法分析是编译过程的第一步，也是编译过程必不可少的步骤。编译过程中执行词法分析的程序称为词法分析器。构造词法分析器有两种方法：一种是用手工方式，即根据识别语言的状态转换图，使用某种高级语言直接编写词法分析器；另一种是利用自动生成工具（如 LEX）自动生成词法分析器。本章分别介绍如何手动和自动构造词法分析器。

### 3.1 词法分析器的设计

本节首先介绍词法分析器的功能及其输出的单词符号的表示方式，然后介绍其输入和处理。

#### 3.1.1 词法分析器的功能

词法分析器又叫作扫描器，其功能是从左往右逐个字符地对源程序进行扫描，然后按照源程序的构词规则识别出一个个单词符号，把作为字符串的源程序等价地转化成单词符号串的中间程序。单词符号是程序设计语言中基本的语法单元，通常分为 5 种：

- 1) 关键字（又称基本字或保留字）：程序设计语言中定义的具有固定意义的英文单词，通常不能用作其他用途，比如 C 语言中的 while、if、for 等都是关键字。
- 2) 标识符：用来表示名字的字符串，如变量名、数组名、函数名等。
- 3) 常数：包括各种类型的常数，如整型常数 386、实型常数 0.618、布尔型常数 TRUE 等。
- 4) 运算符：又分为算术运算符，如 +、-、\*、/ 等；关系运算符，如 =、>=、> 等；逻辑运算符，如 or、not、and 等。
- 5) 界符：如 “,” “;” “(” “)” “:” 等。

在上面所给出的 5 种单词符号中，关键字、运算符和界符是程序设计语言提前定义好的，



因此它们的数量是固定的，通常只有几十个或者上百个。而标识符和常数是程序设计人员根据编程需要按照程序设计语言的规定构造出来的，因此数量即便不是无穷，也是非常大的。

词法分析器输出的单词符号通常用二元式（单词种别，单词符号的属性值）表示。其中：

1) 单词种别。单词种别表示单词种类，常用整数编码，这种整数编码又称为种别码。至于一种程序设计语言的单词如何分类、怎样编码，主要取决于技术上的方便。一般来说，基本字可“一字一种”，也可将其全体视为一种；运算符可“一符一种”，也可按运算符的共性分为几种；界符一般采用“一符一种”分法；标识符通常统归为一种；常数可统归为一种，也可按整型、实型、布尔型等分为几种。

2) 单词符号的属性值。单词符号的属性值是反映单词特征或者特性的值，是编译中其他阶段所需要的信息。如果一个种别只含有一个单词符号，那么其种别编码就完全代表了自身的值，因此相应的属性值就不需要再单独给出。如果一个种别含有多个单词符号，那么除了给出种别编码之外还应给出单词符号自身的属性值，以便把同一种类的单词区别开来。例如，对于标识符，可以用它在符号表的入口指针作为它自身的值；而常数也可用它在常数表的入口指针或者其二进制值作为它自身的值。

### 3.1.2 输入及其处理

词法分析器的结构如图 3-1 所示。词法分析器首先将源程序文本输入到一个缓冲区中，该缓冲区称为输入缓冲区，单词符号的识别可以直接在输入缓冲区中进行。但通常情况下为了识别单词的方便，需要对输入的源程序字符串进行预处理。对于许多程序语言来说，空格、制表符、换行符等编辑性字符只有出现在符号常量中时才有意义；注释几乎可以出现在程序中的任何地方。但编辑性字符和注释的存在一般只是为了改善程序的易读性和易理解性，不影响程序本身的语法结构和实际意义，通常在词法分析阶段可以通过预处理将它们删除。因此可以设计一个预处理子程序来完成上述工作，每当词法分析器调用预处理子程序时，其便处理一串固定长度的源程序字符串，并将处理结果放在词法分析器指定的缓冲区中，称为扫描缓冲区。接下来单词符号的识别就可以直接在该扫描缓冲区中进行，而不必考虑其他杂务。

扫描器对扫描缓冲区进行扫描时通常使用两个指针，即开始指针和搜索指针，其中，开始指针指向当前正在识别的单词的起始位置，搜索指针用于向前搜索以寻找该单词的终点位置，两个指针之间的符号串就是当前已经识别出来的那部分单词。刚开始时，两个指针都指向下一个要识别的单词符号的开始位置，然后，搜索指针向前扫描，直到发现一个单词符号为止，一旦发现一个单词，搜索指针指向该单词的右部，在处理完这个单词以后，两个指针同时指向下一个要识别的单词符号的起始位置。

为了解决程序设计语言中某些单词符号可能存在公共前缀的问题，在进行词法分析时

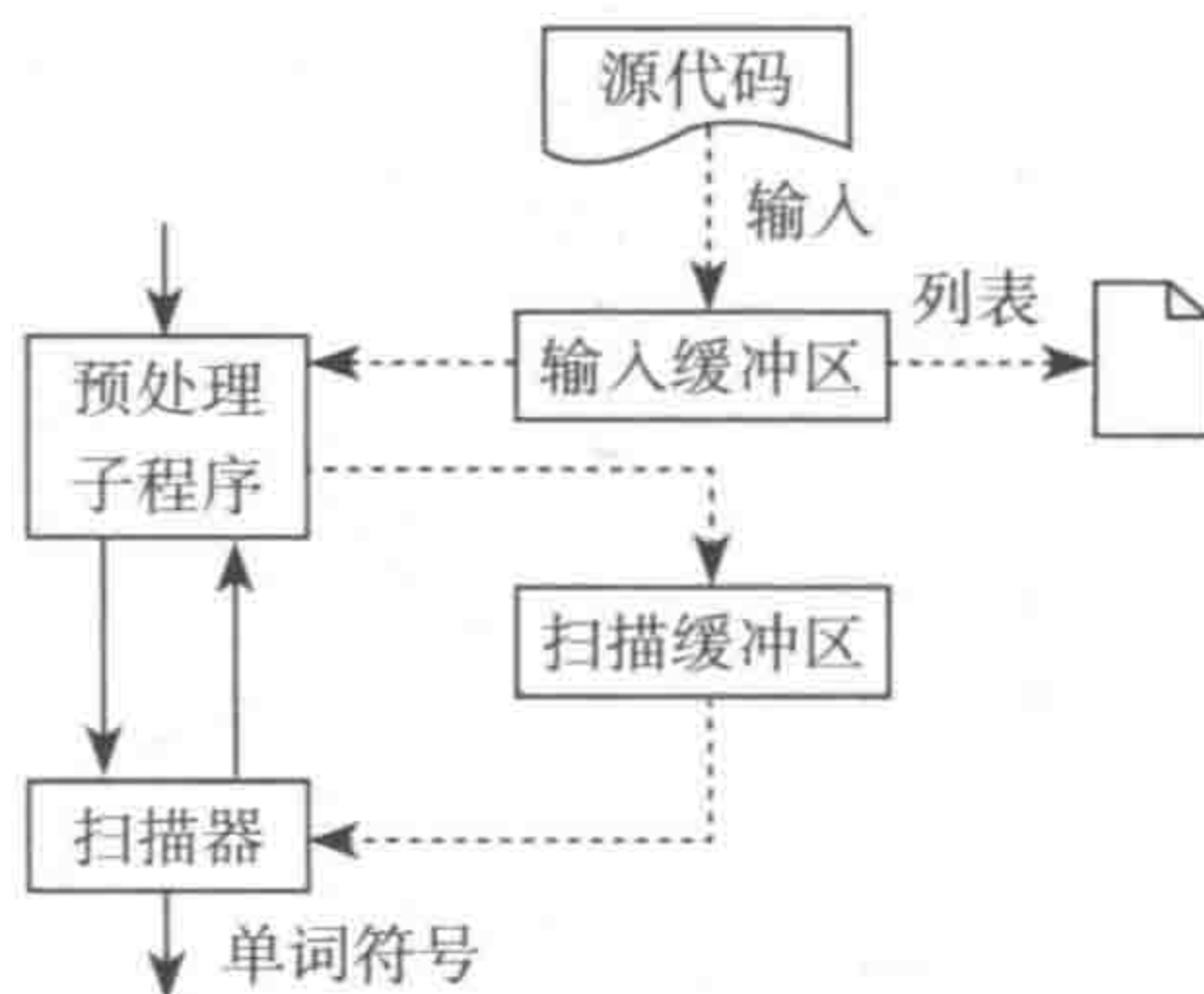


图 3-1 词法分析器结构图



需采用所谓超前搜索技术，也即词法分析器在读取单词时，为了判断是否已读入整个单词的全部字符，常采取向前多读取字符并通过读取的字符来判别的方式。

### 3.2 词法分析器的手工实现

手工构造词法分析器首先需要将描述单词符号的正规文法或者正规式转化为状态转换图，然后再依据状态转换图进行词法分析器的构造。状态转换图是一个有限方向图，结点代表状态，用圆圈表示；状态之间用箭弧连接，箭弧上的标记（字符）代表射出结点状态下可能出现的输入字符或字符类。一张转换图只包含有限个状态，其中有一个为初态，至少要有一个终态（用双圈表示）。大多数程序语言的单词符号都可以用状态转换图予以识别。具体过程如下：

- 1) 从初始状态出发。
- 2) 读入一个字符。
- 3) 按当前字符转入下一状态。
- 4) 重复步骤 2 ~ 3 直到无法继续转移为止。

在遇到读入的字符是单词的分界符时，若当前状态是终止状态，说明读入的字符组成了一个单词；否则，说明输入字符串不符合词法规则。

这里我们以一个 C 语言子集作为例子，说明如何基于状态转换图手工编写词法分析器，该语言子集几乎包含了 C 语言所有的单词符号。主要步骤是，首先给出描述该子集中各种单词符号的词法规则，其次构造其状态转换图，然后根据状态转换图编写词法分析器。表 3-1 列出了这个 C 语言子集的所有单词符号以及它们的种别编码。该语言子集所包含的单词符号有：

- 1) 标识符：以字母、下划线开头的字母、数字和下划线组成的符号串。
- 2) 关键字：标识符的子集，C 语言的关键字共有 32 个（为了测试加入了输入输出函数，共计 34 个）。
- 3) 无符号十进制整数：由 0 到 9 数字组成的字符串。
- 4) 算符和界符：“{” “}” “[” “]” “(” “)” “.” “->” “~” “++” “--” “!” “&” “\*” “/” “%” “+” “-” “<<” “>>” “>” “>=” “<” “<=” “==” “!=” “^” “|” “&&” “||” “?” “=” “/=” “\*=” “%=” “+=” “-=” “&=” “^=” “|=” “,” “#” “;” “:”，共计 44 个。

表 3-1 C 语言子集的单词符号及种别编码

单词符号	种别编码	单词符号	种别编码	单词符号	种别编码	单词符号	种别编码
void	1	long	7	typedef	13	const	19
char	2	signed	8	sizeof	14	volatile	20
int	3	unsigned	9	auto	15	return	21
float	4	struct	10	static	16	continue	22
double	5	union	11	register	17	break	23
short	6	enum	12	extern	18	goto	24



(续)

单词符号	种别编码	单词符号	种别编码	单词符号	种别编码	单词符号	种别编码
if	25	(	39	<<	53	/=	67
else	26	)	40	>>	54	*=	68
switch	27	.	41	>	55	%=	69
case	28	->	42	>=	56	+=	70
default	29	~	43	<	57	-=	71
for	30	++	44	<=	58	&=	72
do	31	--	45	= =	59	^=	73
while	32	!	46	!=	60	=	74
scanf	33	&	47	^	61	,	75
printf	34	*	48		62	#	76
{	35	/	49	&&	63	;	77
}	36	%	50		64	:	78
[	37	+	51	?	65	标识符	79
]	38	-	52	=	66	数字	80

下面为产生该 C 语言子集中所涉及的单词符号的文法的产生式。

#### 1) 关键字文法:

keyword  $\rightarrow$  void | char | int | float | double | short | long | signed | unsigned | struct | union | enum | typedef | sizeof | auto | static | register | extern | const | volatile | return | continue | break | goto | if | else | switch | case | default | for | do | while | scanf | printf

#### 2) 标识符文法:

letter  $\rightarrow$  A | ... | Z | a | ... | z  
 digit  $\rightarrow$  0 | 1 | ... | 9  
 id  $\rightarrow$  letter rid | - rid  
 rid  $\rightarrow$  letter rid | - rid | digit rid |  $\epsilon$

#### 3) 无符号整数文法:

digit  $\rightarrow$  0 | 1 | ... | 9  
 digits  $\rightarrow$  digit rdigit  
 rdigit  $\rightarrow$  digit rdigit |  $\epsilon$

#### 4) 算符和界符的文法:

op  $\rightarrow$  { | } | [ | ] | ( | ) | . | -bigger | ~ | +plus | -minus | ! | & | \* | / | %  
 | + | - | <less | >bigger | > | >equal | < | <equal | =equal | !=equal | ^ | | | &and |  
 | or | ? | = | /equal | equal | %equal | +equal | -equal | &equal | ^equal | |equal | , |  
 # | ; | :  
 bigger  $\rightarrow$  >  
 plus  $\rightarrow$  +  
 minus  $\rightarrow$  -



```

less → <
equal → =
and → &
or → |

```

依据上述文法可得到如图 3-2 所示的状态转换图。其中，终态上的星号 (\*) 表示此时还要把超前读出的字符退回，即搜索指针回调一个字符位置。在状态 2 时，所识别出的标识符应先与表的前 34 项逐一比较，若匹配，则该标识符是一个保留字，否则就是标识符。如果是标识符，则返回相应的种别编码和标识符本身。在状态 4 时，将识别的常数种别编码和常数值返回。在状态 7、12、16、19、23 时，为了识别相应的算符需进行超前搜索。

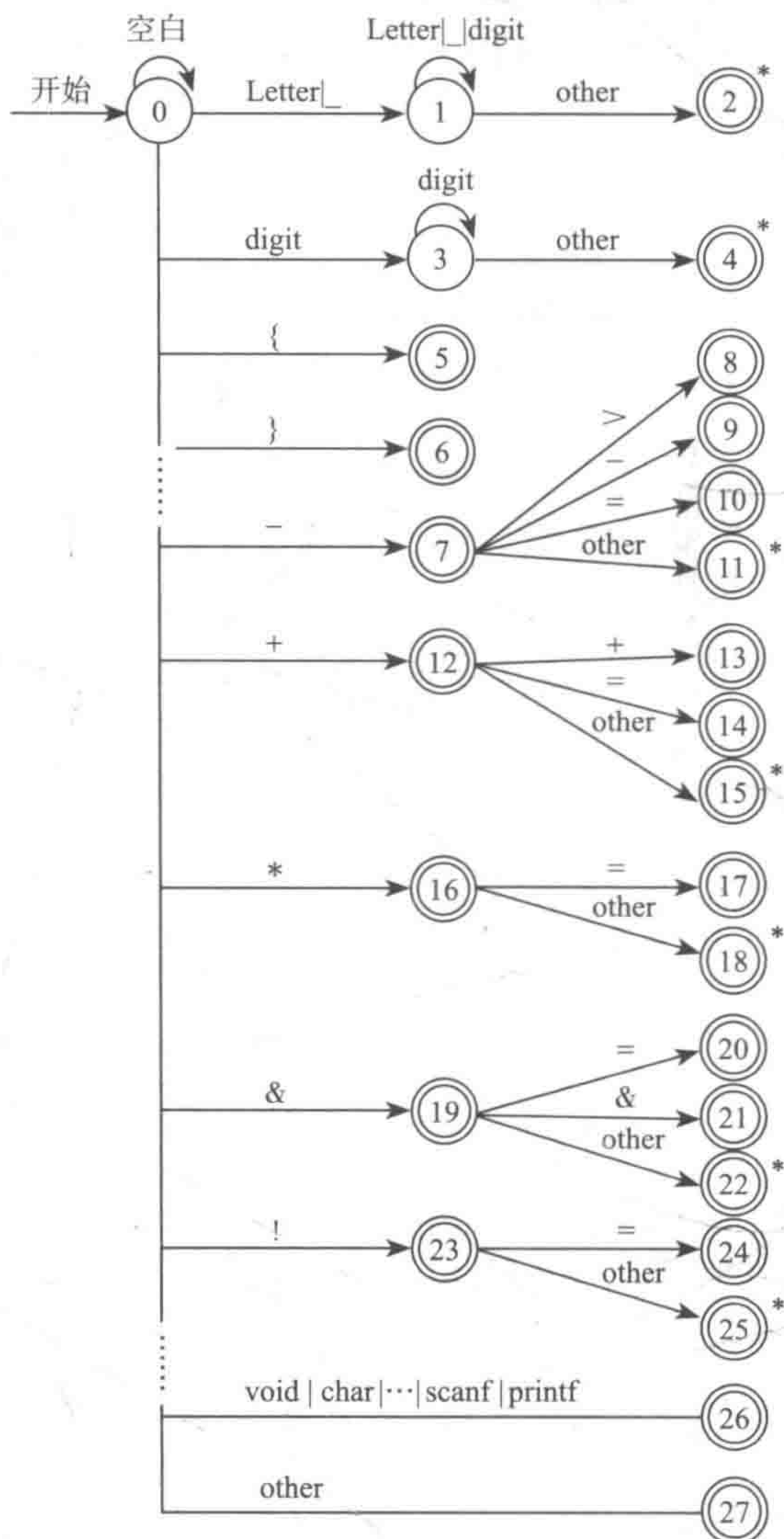


图 3-2 对 C 语言子集进行词法分析的状态转换图



状态转换图非常易于实现,最简单的方法是为每个状态编写一段程序。对于不含回路的分支状态来说,可以用一个 switch() 语句或一组 if-else 语句实现;对于含回路的状态来说,可以让它对应一个 while 语句。图 3-3 给出了整个词法分析器的程序设计流程图。

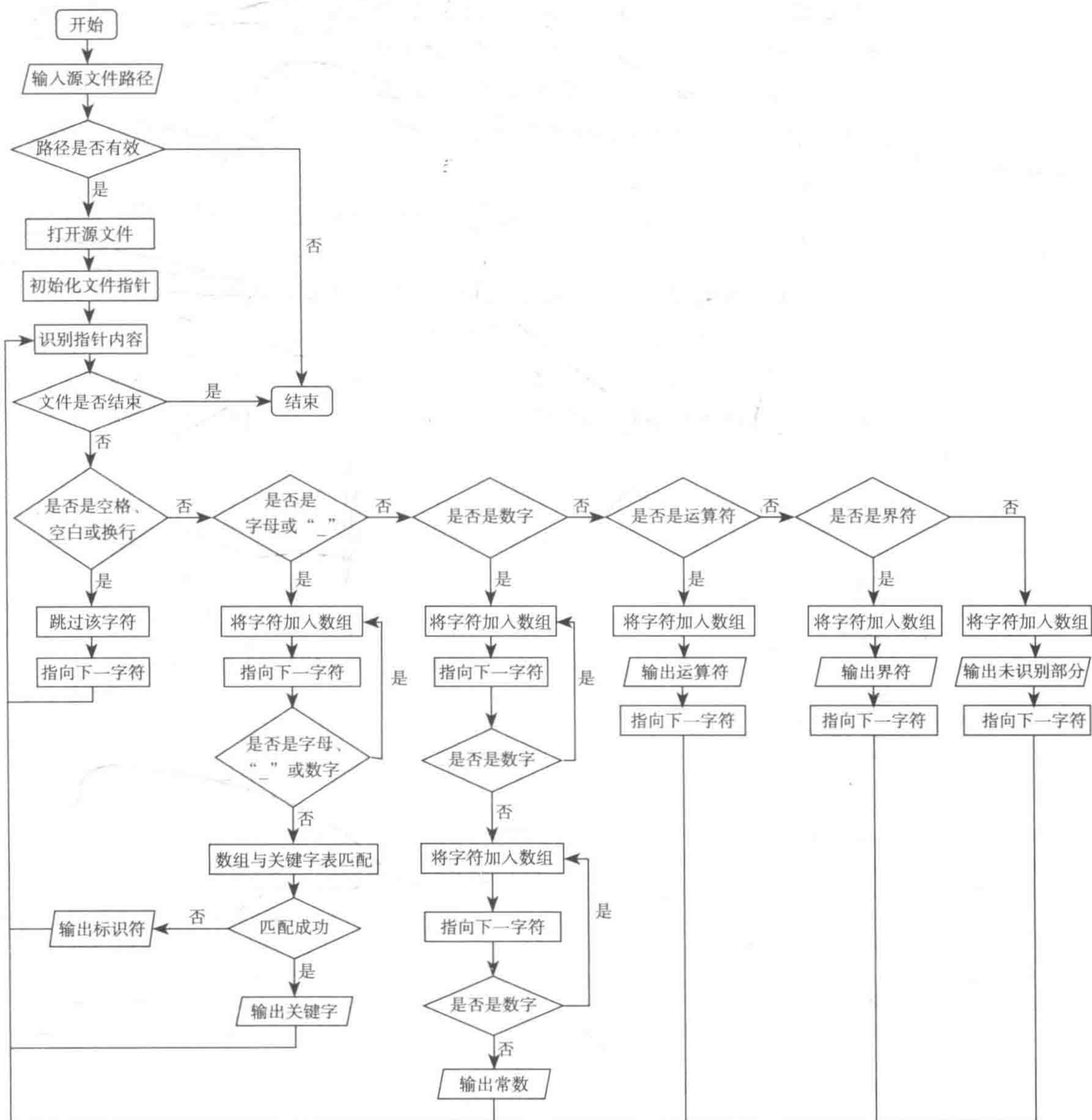


图 3-3 程序设计流程图

为便于阅读,对下面程序中所涉及的变量和过程进行以下说明:

1) ch 字符变量:存放最新读入的源程序字符。



2) strToken 字符数组：存放构成单词符号的字符串。

3) void initialization() 子程序：对单词符号设定种别编码。

4) getNextChar() 子程序过程：把下一个字符读入 ch 中。

5) getbc() 子程序过程：每次调用时，检查 ch 的字符是否为空白符、回车或者制表符，若是则反复调用 getNextChar()，直至 ch 中读入一非上述符号。

6) concat() 子程序过程：把 ch 中的字符连接到 strToken。

7) isLetter()、isDigital() 和 isUnderline 布尔函数：判断 ch 中字符是否为字母、数字或下划线。

8) reserve\_string() 整型函数：对于 strToken 中的字符串判断它是否为保留字，若它是保留字则给出它的编码，否则返回 0。

9) reserve\_operator() 整型函数：返回 strToken 中所识别出的算符和界符编码。

10) retract() 子程序：把搜索指针回调一个字符位置。

11) error()：出现非法字符：显示出错信息。

对应于图 3-2 的词法分析器构造如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#include<iostream>
#include<fstream>
using namespace std;
struct symbol
{
    char * str;
    int coding;
};
char *keyword_list[34] = { "void", "char", "int", "float", "double", "short",
"long", "signed", "unsigned", "struct", "union", "enum", "typedef", "sizeof", "auto",
"static", "register", "extern", "const", "volatile", "return", "continue", "break",
"goto", "if", "else", "switch", "case", "default", "for", "do", "while", "scanf", "printf"};
char *operator_list[44] = { "{", "}", "[", "]", "(", ")", ".", ">", "~", "++", "--",
"!", "&", "*", "/", "%", "+", "-", "<<", ">>", ">", ">=", "<", "<=", "==", "!=", "^", "|", "&&",
"||", "?", "=", "/=", "*=", "%=", "+=", "-=", "&=", "^=", "|=", ",", "#", ";", ":"};
char ch; // 读入的字符
char strToken[20] = ""; // 读入的字符串
int eof_flag = 0;
int num = 1; // 编码的数字 (为了递增)
int row = 1;
struct symbol keywords[34];
struct symbol identifiers[44];
FILE *fp = NULL;
FILE *fw = NULL;
ofstream out;
```



```

// 给单词符号设定种别编码
void initialization() {
    // 给关键字设定种别编码
    for (int i = 0; i < 34; i++)
    {
        keywords[i].str = keyword_list[i];
        keywords[i].coding = num;
        num++;
    }
    // 给算符和界符设定种别编码
    for (i = 0; i < 44; i++) {
        identifiers[i].str = operator_list[i];
        identifiers[i].coding = num;
        num++;
    }
    // 数字 79, 标识符 80
}

```

```

// 把下一个字符读入 ch 中
void getNextChar(FILE *ffp)
{
    if ((ch = getc(ffp)) == EOF)
    {
        eof_flag = 1;
    }
    if (ch == '\n')
        row++;
}

```

// 检查 ch 的字符是否为空白符、回车或者制表符, 若是, 则反复调用 getNextChar (), 直至 ch 中读入一非上述符号

```

void getbc(FILE * ffp)
{
    while (ch == ' ' || ch == '\n' || ch == '\t')
    {
        getNextChar(ffp);
    }
}

```

```

// 判断 ch 是否为字母
bool isLetter(char ch)
{
    return isalpha(ch);
}

```

```

// 判断 ch 是否为数字
bool isDigit(char ch)
{
    return isdigit(ch);
}

```

```

// 判断 ch 是否为下划线

```



```

bool isUnderline(char ch)
{
    if (ch == '_')
        return 1;
    else
        return 0;
}

// 将输入的字符 ch 连接到 strToken
void concat()
{
    char * tmp = &ch;
    strcat(strToken, tmp);
}

// 把搜索指针回调一个字符位置
void retract(FILE * ffp)
{
    fseek(ffp, -1, SEEK_CUR);
    ch = ' ';
}

// 对于 strToken 中的字符串判断它是否为保留字, 若它是保留字则给出它的编码, 否则返回 0
int reserve_string(char * str) {
    for (int i = 0; i < 34; i++) {
        if ((strcmp(str, keywords[i].str)) == 0)
        {
            return keywords[i].coding;
        }
    }
    return 0;
}

// 返回 strToken 中所识别出的算符和界符编码
int reserve_operator(char* ch)
{
    for (int i = 0; i < 44; i++) {
        if ((strcmp(ch, identifiers[i].str)) == 0)
        {
            return identifiers[i].coding;
        }
    }
    return 0;
}

// 出错处理
void error()
{
    printf("\n *****Error*****\n");
    printf(" row %d  Invaild symbol ! ! ! ", row);
}

```



```
printf("\n *****Error*****\n");  
exit(0);  
}
```

// 词法分析

```
void LexiscalAnalyzer()  
{  
    int num = 0, val = 0, code = 0;  
    strcpy(strToken, "");  
    getNextChar(fp);  
    getbc(fp);  
    switch (ch)  
    {  
        case 'a':  
        case 'b':  
        case 'c':  
        case 'd':  
        case 'e':  
        case 'f':  
        case 'g':  
        case 'h':  
        case 'i':  
        case 'j':  
        case 'k':  
        case 'l':  
        case 'm':  
        case 'n':  
        case 'o':  
        case 'p':  
        case 'q':  
        case 'r':  
        case 's':  
        case 't':  
        case 'u':  
        case 'v':  
        case 'w':  
        case 'x':  
        case 'y':  
        case 'z':  
        case 'A':  
        case 'B':  
        case 'C':  
        case 'D':  
        case 'E':  
        case 'F':  
        case 'G':  
        case 'H':  
        case 'I':  
        case 'J':  
        case 'K':
```



```

case 'L':
case 'M':
case 'N':
case 'O':
case 'P':
case 'Q':
case 'R':
case 'S':
case 'T':
case 'U':
case 'V':
case 'W':
case 'X':
case 'Y':
case 'Z':
case '_':
    while (isLetter(ch) || isDigit(ch) || isUnderline(ch))
    {
        concat();
        getNextChar(fp);
    }
    retract(fp);
    code = reserve_string(strToken);
    if (code == 0)
    {
        printf("(%d , %s)\n", 79, strToken);
    }
    else
    {
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
    while (isdigit(ch))
    {
        concat();
        getNextChar(fp);
    }
    retract(fp);
    printf("(%d , %s)\n", 80, strToken);
    break;

```



```

case '{':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '}':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '[':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case ']':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '(':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case ')':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '.':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '-':
    concat();
    getNextChar(fp);
    if (ch == '>')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '-')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '=')

```

```

    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
else
{
    retract(fp);
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
}
break;
case '~':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '+':
    concat();
    getNextChar(fp);
    if (ch == '+')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '*':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {

```



```

        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '&':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '&')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '!':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '%':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);

```

```

        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '<':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '<')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '>':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '>')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }

```



```

    }
    break;
case '=':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '^':
    concat();
    getNextChar(fp);
    if (ch == '^')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case '|':
    concat();
    getNextChar(fp);
    if (ch == '|')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '|')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else
    {

```

```

        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;

case '?':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '/':
    concat();
    getNextChar(fp);
    if (ch == '=')
    {
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    else if (ch == '/') // 跳过注释
    {
        getNextChar(fp);
        while (ch != '\n') {
            getNextChar(fp);
        }

        break;
    }
    else if (ch == '*') // 跳过注释
    {
        getNextChar(fp);
        while (ch != '*') {
            getNextChar(fp);
        }
        getNextChar(fp);
        if (ch == '/');
        break;
    }
    else
    {
        retract(fp);
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
    }
    break;
case ',':
    concat();
    code = reserve_operator(strToken);
    printf("(%d , %s)\n", code, strToken);
    break;
case '#':

```



```

        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
        break;
    case ';':
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
        break;
    case ':':
        concat();
        code = reserve_operator(strToken);
        printf("(%d , %s)\n", code, strToken);
        break;
    default:
        if (ch == EOF)
        {
            eof_flag = 1;
            break;
        }
        error();
    }
}

// 主函数
int main()
{
    initialization();
    char name[1024];
    cout<<" 从文件读入: ";
    cin>>name;
    fp=fopen(name,"r");
    out.open("result.txt");
    while(!feof(fp))
    {
        if (eof_flag == 1)
        {
            exit(1);
        }
        LexiscalAnalyzer();
    }
    fclose(fp);
    out.close();
    return 0;
}

```

### 3.3 词法分析器的 LEX 实现

由于程序设计语言中的单词基本上都可用一组正规式来描述, 因此, 人们希望构造一个自动生成系统: 对于一个给定的高级语言, 只要给出用来描述其各类单词词法结构的一个

组正则表达式，以及识别各类单词时词法分析程序应采取的语义动作，该系统便可自动产生此语言的词法分析程序。1975 年美国贝尔实验室的 M. Lesk 和 Schmidt 基于正规式与有限自动机的理论研究，用 C 语言研制了一个词法分析程序的自动生成工具 LEX。对任何高级程序语言，用户只需用正规式描述该语言的各个词法类（这一描述称为 LEX 的源程序），LEX 就可以自动生成该语言的词法分析程序。

3.3.1 LEX 源文件结构

LEX 的输入是用 LEX 源语言编写的程序，它是扩展名为 .l 的文件。LEX 源程序经 LEX 系统处理后输出一个 C 程序文件，此文件含有两部分内容：一个是依据正规式所构建的状态转移表；另一个是用来驱动该状态转移表的总控程序 yylex ()。该文件再经过 C 编译器的编译就产生一个实际可以运行的词法分析程序，其使用方法如图 3-4 所示。

一般而言，一个 LEX 源程序由 “%%” 分隔的三个部分组成，其书写格式为：

定义部分  
%%  
识别规则部分  
%%  
辅助函数部分

其中，定义部分和辅助函数部分是任选的，识别规则部分则是必备的。如果辅助函数部分缺省，则第二个分隔号 “%%” 可以

省略；但由于第一个分隔号 %% 用来指示识别规则部分的开始，故即使没有定义部分，也不能将其省略。下面将对这三部分的内容及其书写格式作一概括性介绍。

1. 定义部分

定义部分对规则部分要引用的文件和变量进行说明，通常可包含头文件表、常数定义、全局变量定义、外部变量定义以及正规式定义等。正规式定义用来定义在规则部分引用的正规式，类似于 C 语言中的宏定义，所以也称为宏定义。每一个宏定义由分隔符（适当个数的空格或制表字符）连接的宏名字和宏内容组成：

D<sub>1</sub> r<sub>1</sub>  
D<sub>2</sub> r<sub>2</sub>  
... ...  
D<sub>n</sub> r<sub>n</sub>

其中，D<sub>i</sub> 是要定义的一组互不相同的宏名字，它是以字母或者下划线 “\_” 开始，由字母、数字和下划线 “\_” 组成的字符串，并且大小写敏感；每个 r<sub>i</sub> 是以后用来替换宏名字 D<sub>i</sub> 的宏内容，其都是  $\Sigma \cup \{D_1, D_2, \dots, D_{i-1}\}$  上的正规式； $\Sigma$  是相应程序设计语言的基本字符集。设置宏定义的目的在于给一些较为复杂的正规式命名，以便以后在需要出现这些正规式的地方只需写上相应的宏名字。例如：

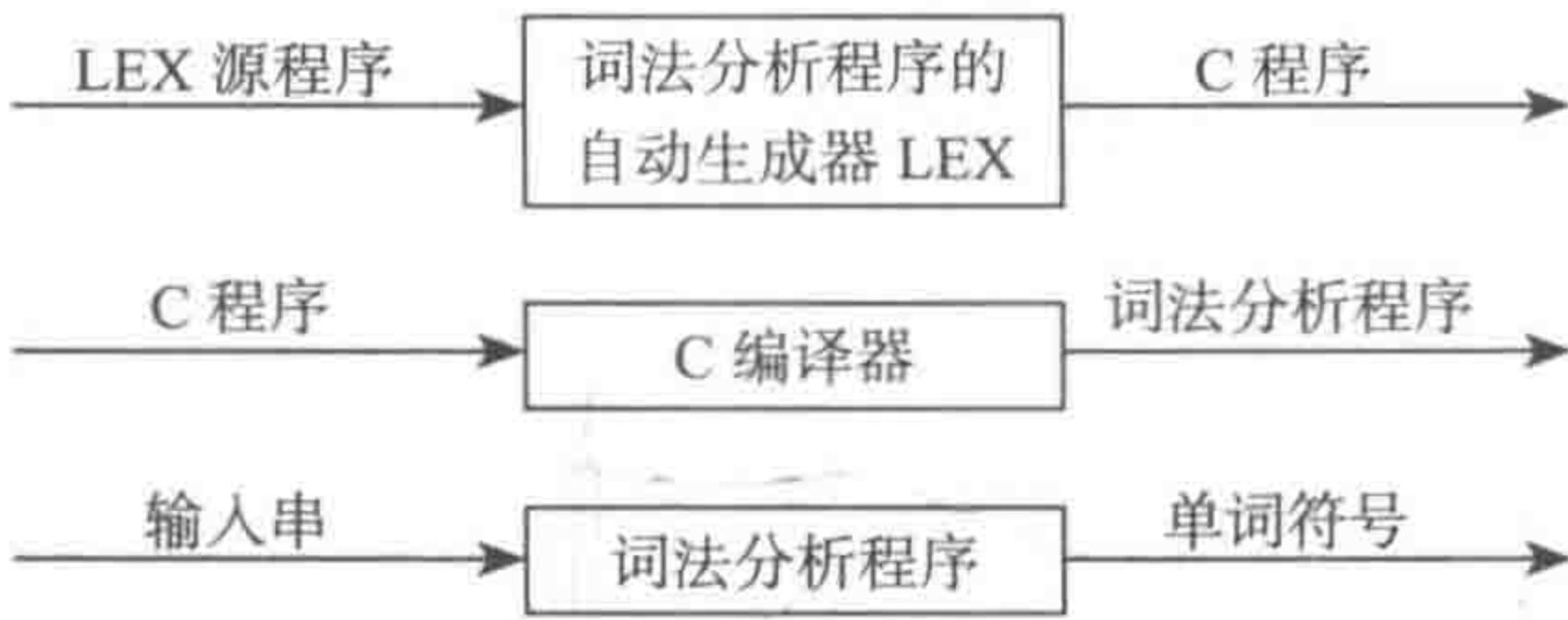


图 3-4 用 LEX 构建词法分析器



```
digit [0-9]
letter [a-zA-Z]
```

其中, `digit` 是匹配单个数字的正规式; `letter` 是匹配单个字母的正规式。需要注意的是, 在以后的定义部分和规则部分, 凡对已定义宏名字的引用都需用花括号将它们括起来。例如:

```
{letter}({letter}|{digit})*
```

LEX 扫描源文件时, 将 `{letter}` 替换为 `letter` 所定义的正规式并加上括号, 将 `{digit}` 替换为 `digit` 所定义的正规式并加上括号, 所以, 上式等价于

```
([a-zA-Z])( ([a-zA-Z]|([0-9])))*
```

它将匹配以字母开始并由字母和数字组成的字符串。

除宏定义外, 定义部分的其余代码需用符号 “`%{`” 和 “`%}`” 括起来。LEX 将 “`%{`” 和 “`%}`” 之间的内容直接复制到生成的 C 文件 `lexyy.c` 中。在 LEX 源程序中, 起标识作用的符号 “`%%`” “`%{`” 以及 “`%}`” 都必须处于所在行的最左字符位置。另外, 在其中也可随意添加 C 语言形式的注释。如

```
%{
#include <math.h>
#include <string.h>
int num_chars = 0, num_lines = 0;
%}
```

## 2. 识别规则部分

识别规则部分是具有如下形式的语句序列:

```
P1 { A1 }
P2 { A2 }
...
Pn { An }
```

其中, 每个  $P_i$  都是定义在  $\Sigma \cup \{D_1, D_2, \dots, D_n\}$  上的正规式 ( $D_i$  是定义部分所定义的宏名字), 用来描述一种单词模式;  $A_i$  是一段 C 语言源代码, 用来指出当从输入字符串中识别出词型为  $P_i$  的单词时词法分析器应执行的操作。每个  $P_i$  都必须顶行书写, 并用分隔符 (若干个空格或 `tab` 字符) 与其后的代码段  $A_i$  分开。每个代码段  $A_i$  可引用已定义的符号常量、全局变量和外部变量, 并能调用辅助函数部分所定义的函数, 必要时也可在  $A_i$  中定义自己的局部变量。 $A_i$  一般不必用花括号括起来, 但若  $A_i$  多于一行或者需要在其中定义局部变量, 则应使用花括号并且左括号 “`{`” 一定要与相应的  $P_i$  在同一行, 以便确定这些局部变量的作用域。

## 3. 辅助函数部分

在识别规则部分中所调用的函数若不是库函数, 则需要给出这些函数的定义。这些函



数在辅助函数部分给出，由用户用 C 语言编写，它们由 LEX 系统直接复制到输出的 C 程序文件之中。

表 3-2 中列出了 LEX 中常用的一些变量和函数，在与正规式匹配的动作或辅助过程中都可以使用。

表 3-2 LEX 中常用的一些变量和函数

yyin	FILE * 类型，指向 LEX 输入文件，缺省情况下指向标准输入
yyout	FILE * 类型，它指向 LEX 输出文件。缺省情况下指向标准输出
yytext	char * 类型，指向与识别规则中的一个正规式匹配的单词的首字符
yy leng	int 类型，记录与识别规则中正规式匹配的单词的长度
yy lex()	从该函数开始分析，由 LEX 自动生成
yy wrap()	文件结束处理函数，如果其返回值是 1，就停止解析。可以用来解析多个文件
echo	将 yytext 打印到 yyout

例 3.1 下面给出了一个 LEX 源文件，其功能是统计文本文件中的字符数和行数。该程序首先定义了 num\_chars 和 num\_lines 两个计数器，分别记录文本文件的字符数和行数。在 LEX 源文件中定义两个正规式 “\n” 和 “.”，分别用来匹配换行符和任意字符，并且在识别这两个正规式后其相应的计数器累加 1，从而完成对文件的字符数和行数的统计。

```
%{
/* 该 LEX 程序的功能是统计文本文件中的字符数和行数，并输出结果 */
#include <stdio.h>
int num_chars = 0, num_lines= 0; /* C 语言全局变量，定义两个计数器并置初值为 0*/
}%
%%
\n    {++num_chars; ++num_lines;} /* " \n" 匹配换行符 */
.      {++num_chars;} /* "." 匹配除换行符以外的任意字符 */
%%
main() /* 主函数 */
{
    yy lex();
    printf(" 本文件的行数为: %d, 字符数为: %d\n", num_lines, num_chars);
    return 0;
}
int yy wrap() /* 文件结束处理函数，yy lex 在读到文件结束标记 EOF 时，调用该函数，用户必须提供该函数，否则在编译链接时会出错 */
{
    return 1; /* 返回 1 表示文件扫描结束，不必再扫描别的文件 */
}
```

3.3.2 LEX 系统中的正规式

LEX 源程序中在宏定义及识别规则部分都涉及许多正规式。这些正规式除应遵循正规式基本定义的规定外，为了便于用户灵活、紧凑地构造复杂的正规式，LEX 还增添了若干个运算符和新的构造规则，下面将结合这些新增的内容对 LEX 系统中的正规式进行扼要



说明。

LEX 系统中的正规式由通常的文字字符（下面称为正文字符）和元字符组成。元字符一般用作运算符或控制符，常用的元字符有  $*$ 、 $+$ 、 $?$ 、 $|$ 、 $\{ \}$ 、 $[ ]$ 、 $()$ 、 $.$ 、 $^$ 、 $\$$ 、 $"$ 、 $\backslash$ 、 $-$ 、 $/$ 、 $<$ 、 $>$ 。利用正文字符和元字符组成正规式的规则及其匹配输入串的规则分述如下。

### (1) 单个正文字符

单个正文字符  $c$  是正规式，用于匹配与其相同的单个正文字符  $c$ 。

如果上述元字符需要以正文字符的形式出现于正规式中，则需使用双引号（ $"$ ）或反斜线（ $\backslash$ ）作为转义字符将其变为正文字符，如  $"+"$ （或  $\backslash +$ ）和  $"-"$ （或  $\backslash -$ ）在正规式中分别表示加号和减号。此外，C 语言中的一些转义字符序列也可出现在正规式中，如  $\backslash b$ 、 $\backslash f$ 、 $\backslash n$ 、 $\backslash r$ 、 $\backslash s$  和  $\backslash t$  分别表示退格、换页、换行、回车、空格和制表。

### (2) 字符类

字符类是正规式，用于匹配该字符类所确定的字符集合中的任一字符。

字符类有两种表示形式。一种是在方括号（ $[]$ ）内列出字符类中的全部字符（字符之间不需要用逗号分隔），如  $[abc123]$  将匹配小写字母  $a$ 、 $b$ 、 $c$ 、 $1$ 、 $2$ 、 $3$  中的任一字符。另一种是补集表示法，其在方括号（ $[]$ ）内列出所有不在字符类中的字符，书写格式为  $[^{\dots}]$ ，它匹配除  $^$  之后所列字符之外的所有字符，如  $[^abc123]$  将匹配除小写字母  $a$ 、 $b$ 、 $c$ 、 $1$ 、 $2$ 、 $3$  之外的任何字符。此外，对于连续的字符可以用  $-$  进行缩写，如  $[0-9a-zA-Z]$  匹配任一字母、数字字符， $[^0-9]$  匹配除数字字符之外的任何字符。

需要注意的是，除  $^$ 、 $\backslash$  和  $-$  外，其余元字符在方括号内失去其特殊含义。如果使一个字符类含有这 3 个字符中的某个字符，只需要将其放在方括号内首字符或末字符的位置上，如  $[-+.0-9]$  和  $[+ .0-9-]$  均匹配正号、负号、小数点和十进制数字中的任何字符。

### (3) “连接”与“或”

设  $r_1$  和  $r_2$  是正规式，则  $r_1 r_2$ （表示  $r_1$  和  $r_2$  的连接）和  $r_1 | r_2$ （表示  $r_1$  或  $r_2$ ）也都是正规式。

### (4) 重复

设  $r$  为正规式，则  $"r^*"$  表示  $r$  可重复零次或任意多次， $"r^+ "$  表示  $r$  可重复一次或任意多次， $"r^? "$  表示  $r$  可有可无。

### (5) 通配符

$"."$  为通配符，用来匹配除换行之外的任何字符。例如， $B \cdot C$  不但可以匹配  $BNC$ 、 $BDC$  及  $BVC$  等，而且还可匹配  $RBYCA$  中的  $BYC$ 。

### (6) 行首字符串或行末字符串

以元字符上箭头  $^$  开头的字符串用于匹配行首字符串，以元符号  $\$$  为末尾的字符串用于匹配行末字符串。例如， $^BEGIN$  表示仅当字符串  $BEGIN$  出现在某一行的开头才能获得匹配； $END\$$  表示仅当字符串  $END$  出现在一行的结尾才能获得匹配。这里所说的一行的开头，是指整个输入字符流的开始，或者紧跟在一个换行字符之后的位置，一行的结尾则指紧靠换行字符之左的位置。



### (7) 超前搜索

“/”为超前搜索符。设  $r_1$  和  $r_2$  是正规式，则  $r_1/r_2$  也都是正规式，其  $r_1$  是否与一个字符串相匹配取决于紧跟其后的超前搜索部分是否与  $r_2$  相匹配。也就是说， $r_2$  仅作为  $r_1$  获得匹配的条件，而非所要识别单词词型的一部分。例如，为了识别 Fortran 源程序中的关键字 DO，需采用超前搜索技术，相应的词型可写成

```
DO/ ({letter}|{digit})*= ({letter}|{digit})*
```

表示词法分析程序在输入缓冲区中超前扫描一串字母或数字，接着扫描等号以及后面的一串字母或者数字，最后扫描到逗号，才能确认关键字 DO 得到识别。

### 3.3.3 LEX 的使用方式

LEX 通常有两种使用方式：一种是将 LEX 作为一个单独的工具，用以生成所需的识别程序，这些识别程序多被用在诸如编辑器设计、命令行解释、模式识别、信息检索以及开关系统等一些非开发编译器的应用领域中；另一种是将 LEX 和语法分析器自动生成工具（如 YACC）结合起来使用，以生成一个编译程序的扫描器和语法分析器。

LEX 和 YACC 的最初版本都是作为 UNIX 系统下的工具软件来运行的。假定已有命名为 scanner.l 的 LEX 源文件，则可在 UNIX 系统下通过命令 `lex scanner.l` 调出 LEX 对其进行处理，处理结果是输出名字为 lex.yy.c 的 C 语言文件。再用命令 `cc lex.yy.c -ll` 调用 C 编译器对其进行编译，编译所得到的文件 a.out 便是可执行的词法分析程序，其中选择项“-ll”表示需调用 LEX 的库。如果用户需对所得目标代码文件命名，如将其命名为 Cifafenxi，则可用下面的命令进行编译：

```
cc lex.yy.c -ll -o Cifafenxi
```

LEX 和 YACC 已经成功移植到 Windows 系统下，Parser Generator 便是其中常用的工具之一，该工具是 Windows 平台下的 LEX 和 YACC 集成环境，其利用 LEX 和 YACC 能够生成 Visual C++、Borland C++ 等 C++ 代码以及相关 Java 代码。Parser Generator 非常适合于与 Visual C++（简称 VC++）集成，下面对该工具如何生成代码并使用 Visual C++ 进行编译做一简要介绍。

在安装了 Parser Generator 后，在 VC++ 中进行以下两项设置，即可使 VC++ 编译和链接由 Parser Generator 产生的文件。

#### 1. 目录设置

为了在 VC++ 中可以找到 LEX 和 YACC 的头文件 lex.h 和 yacc.h，以及 LEX 和 YACC 的库文件，需要对 VC++ 进行相关的目录设置。

- 1) 选择 Tools 菜单中的 Options 命令，打开 Options 对话框。
- 2) 选择 Directories 选项卡。



3) 在 Show Directories for 下拉列表框中选择 Include Files, 在 Directories 框中单击最后的空目录, 并填入 C:\PARGEN\INCLUDE (其中 C:\PARGEN 是 Parser Generator 的安装路径, 下同)。

4) 在 Show Directories for 下拉列表框中选择 Library Files, 在 Directories 框中单击最后的空目录, 并填入 C:\PARGEN\LIB\MSVC32。

5) 在 Show Directories for 下拉列表框中选择 Source Files, 在 Directories 框中单击最后的空目录, 并填入 C:\PARGEN\SOURCE。

6) 单击 OK 按钮, Options 对话框设置完毕。

## 2. 项目设置

对每个 VC++ 项目, 都需进行以下设置, 以使 VC++ 可以从特定的库中接收 LEX 和 YACC 所需要的函数和变量。

1) 选择 Project 菜单中的 Settings 命令, 打开 Project Settings 对话框。

2) 在 Settings For 下拉列表框中选择 Win32 Debug。

3) 选择 C/C++ 选项卡, 在 Category 下拉列表框中选择 General。在 Preprocessor Definitions 框中, 在当前文本的最后输入 “,YYDEBUG”。

4) 选择 Link 选项卡, 在 Category 下拉列表框中选择 General。在 Object/Library Modules 框中, 在当前文本的最后输入 “yl.lib”。

5) 单击 OK 按钮, Project Settings 对话框设置完毕。

### 3.3.4 LEX 源文件示例——C 语言词法分析器

在本小节, 我们给出一个针对 C 语言单词识别的 LEX 源程序文件 c.lex, 以供参考。该源程序所涉及的各种单词符号和种别编码参见表 3-1, 在该源程序中针对常数类单词只考虑了十进制整型常数。该文件已在 Parser Generator 环境下编译调试通过。为便于阅读, 特作以下说明:

1) 程序前面的行号是为了便于下面的说明而给出的, 真正的 LEX 程序不能书写行号。

2) 第 1 行到第 84 行介于 “%{” 和 “%}” 之间的内容将直接插入由 LEX 产生的 C 程序中, 第 5 行至第 83 行列出了 C 语言各类单词的名字及其相应的种别编码定义。

3) 第 85 行到第 89 行为正规式的宏定义, white 代表制表、换行及空格三个字符中的任一字符, 把它们均视为 “空白字符”。digit 代表 0 到 9 中的任一数字, letter 代表任一大写或者小写字母, number 代表十进制整数, id 代表 C 语言中的标识符。

4) 第 91 行到第 174 行之间为识别规则部分。第 91 行表示在遇到空白符时, 不需要进行任何语义处理, 用一个空语句表示, 这样就可以将输入字符串的全部空白符过滤掉。第 92 行和第 93 行对注释进行处理。第 94 行到第 173 行表示在识别出标识符、十进制整型常数、关键字和运算符时输出其相应的种别编码。第 174 行中的 “.” 表示不能与第 91 行到



第 173 行正规式匹配的其他字符。

5) 第 176 行到第 180 行为主函数，打开输入输出文件。其中，`yylex()` 是词法分析程序的入口点，每次调用 `yylex()`，就可以从被编译的源程序中得到一个单词。如果正规式的相关动作中无 `return` 语句，则 `yylex()` 并不返回值，如此例中第 94 行到第 174 行，相关的动作只是向文件 `result.txt` 中输出某些提示信息。

下面为 C 语言扫描器的 LEX 源文件 `c.lex`。

```

1. %{
2. #include <stdio.h>
3. #include <stdlib.h>
4. #define VOID 1 //void
5. #define CHAR 2 //char
6. #define INT 3 //int
7. #define FLOAT 4 //float
8. #define DOUBLE 5 //double
9. #define SHORT 6 //short
10. #define LONG 7 //long
11. #define SINGED 8 //signed
12. #define UNSINGED 9 //unsigned
13. #define STRUCT 10 //struct
14. #define UNION 11 //union
15. #define ENUM 12 //enum
16. #define TYPEDF 13 //typedef
17. #define SIZEOF 14 //sizeof
18. #define AUTO 15 //auto
19. #define STATIC 16 //static
20. #define REGISTER 17 //register
21. #define EXTERN 18 //extern
22. #define CONST 19 //const
23. #define VOLATILE 20 // volatile
24. #define RETURN 21 // return
25. #define CONTINUE 22 //continue
26. #define BREAK 23 //break
27. #define GOTO 24 //goto
28. #define IF 25 //if
29. #define ELSE 26 //else
30. #define SWITCH 27 // switch
31. #define CASE 28 // case
32. #define DEFAULT 29 // default
33. #define FOR 30 // for
34. #define DO 31 // do
35. #define WHILE 32 // while
36. #define SCANF 33 //scanf
37. #define PRINTF 34 //printf
38. #define LC 35 // {
39. #define RC 36 // }
40. #define LB 37 // [
41. #define RB 38 // ]

```



```

42. #define LP 39 // (
43. #define RP 40 // )
44. #define DOT 41 //.
45. #define STRUCTOP 42 // - >
46. #define LOGRE 43 //~
47. #define INPLUS 44 // + +
48. #define INMINUS 45 // - -
49. #define LOCRE 46 //!
50. #define AND 47 // &
51. #define STAR 48 // *
52. #define DIVOP 49 // /
53. #define COMOP 50 // %
54. #define PLUS 51 // +
55. #define MINUS 52 // -
56. #define SHIFTR 53 // >>
57. #define SHIFTL 54 // <<
58. #define RELG 55 // >
59. #define RELGEQ 56 // >=
60. #define RELL 57 // <
61. #define RELLEQ 58 // <=
62. #define EQUOP 59 // =
63. #define UEQUOP 60 // !=
64. #define XOR 61 // ^
65. #define OR 62 // |
66. #define ANDAND 63 // &&
67. #define OROR 64 // ||
68. #define QUEST 65 // ?
69. #define EQUAL 66 // =
70. #define ASSIGNDIV 67 // / =
71. #define ASSIGNSTAR 68 // * =
72. #define ASSIGNCOM 69 // % =
73. #define ASSIGNPLUS 70 // + =
74. #define ASSIGNMINUS 71 // - =
75. #define ASSIGNAND 72 // & =
76. #define ASSIGNXOR 73 // ^ =
77. #define ASSIGNOR 74 // | =
78. #define COMMA 75 // ,
79. #define SHA 76 // #
80. #define SEMI 77 // ;
81. #define COLON 78 // :
82. #define ID 79 // 标识符
83. #define NUMBER 80 // 数字
84. %}
85. white [\t\n\ ]
86. digit [0-9]
87. letter [A-Za-z]
88. id ({letter}|_)( {letter}| {digit}|_)*
89. number [1-9]{digit}*|0
90. %%
91. {white}+ ;

```

```

92. "/*"[^]\n]*"*/"      ;          /* 删除注释 */
93. "//"[^]\n]*          ;          /* 删除注释 */
94. {id} {fprintf(yyout, "(%d, %s) \n", ID, yytext); }
95. {number} {fprintf(yyout, "(%d, %s) \n", NUMBER, yytext); }
96. void {fprintf(yyout, "(%d, %s) \n", VOID, yytext); }
97. char {fprintf(yyout, "(%d, %s) \n", CHAR, yytext); }
98. int {fprintf(yyout, "(%d, %s \n", INT, yytext); }
99. float {fprintf(yyout, "(%d, %s) \n", FLOAT, yytext); }
100. double {fprintf(yyout, "(%d, %s) \n", DOUBLE, yytext); }
101. short {fprintf(yyout, "(%d, %s) \n", SHORT, yytext); }
102. long {fprintf(yyout, "(%d, %s) \n", LONG, yytext); }
103. signed {fprintf(yyout, "(%d, %s) \n", SINGED, yytext); }
104. unsigned {fprintf(yyout, "(%d, %s) \n", UNSINGED, yytext); }
105. struct {fprintf(yyout, "(%d, %s) \n", STRUCT, yytext); }
106. union {fprintf(yyout, "(%d, %s) \n", UNION, yytext); }
107. enum {fprintf(yyout, "(%d, %s) \n", ENUM, yytext); }
108. typedef {fprintf(yyout, "(%d, %s) \n", TYPEDF, yytext); }
109. sizeof {fprintf(yyout, "(%d, %s) \n", SIZEOF, yytext); }
110. auto {fprintf(yyout, "(%d, %s \n", AUTO, yytext); }
111. static {fprintf(yyout, "(%d, %s) \n", STATIC, yytext); }
112. register {fprintf(yyout, "(%d, %s) \n", REGISTER, yytext); }
113. extern {fprintf(yyout, "(%d, %s) \n", EXTERN, yytext); }
114. const {fprintf(yyout, "(%d, %s) \n", CONST, yytext); }
115. volatile {fprintf(yyout, "(%d, %s) \n" VOLATILE, yytext); }
116. return {fprintf(yyout, "(%d, %s) \n", RETURN, yytext); }
117. continue {fprintf(yyout, "(%d, %s) \n", CONTINUE, yytext); }
118. break {fprintf(yyout, "(%d, %s) \n", BREAK, yytext); }
119. goto {fprintf(yyout, "(%d, %s) \n", GOTO, yytext); }
120. if {fprintf(yyout, "(%d, %s) \n", IF, yytext); }
121. else {fprintf(yyout, "(%d, %s) \n", ELSE, yytext); }
122. switch {fprintf(yyout, "(%d, %s) \n", SWITCH, yytext); }
123. case {fprintf(yyout, "(%d, %s) \n", CASE, yytext); }
124. default {fprintf(yyout, "(%d, %s) \n", DEFAULT, yytext); }
125. for {fprintf(yyout, "(%d, %s) \n", FOR, yytext); }
126. do {fprintf(yyout, "(%d, %s) \n", DO, yytext); }
127. while {fprintf(yyout, "(%d, %s) \n", WHILE, yytext); }
128. scanf {fprintf(yyout, "(%d, %s) \n", SCANF, yytext); }
129. printf {fprintf(yyout, "(%d, %s) \n", PRINTF, yytext); }
130. "{" {fprintf(yyout, "(%d, %s) \n", LC, yytext); }
131. "}" {fprintf(yyout, "(%d, %s) \n", RC, yytext); }
132. "[" {fprintf(yyout, "(%d, %s) \n", LB, yytext); }
133. "]" {fprintf(yyout, "(%d, %s) \n", RB, yytext); }
134. "(" {fprintf(yyout, "(%d, %s) \n", LP, yytext); }
135. ")" {fprintf(yyout, "(%d, %s) \n", RP, yytext); }
136. "." {fprintf(yyout, "(%d, %s) \n", DOT, yytext); }
137. "->" {fprintf(yyout, "(%d, %s) \n", STRUCTOP, yytext); }
138. "~" {fprintf(yyout, "(%d, %s) \n", LOGRE, yytext); }
139. "++" {fprintf(yyout, "(%d, %s) \n", INPLUS, yytext); }
140. "--" {fprintf(yyout, "(%d, %s) \n", INMINUS, yytext); }
141. "!" {fprintf(yyout, "(%d, %s) \n", LOCRE, yytext); }

```



```

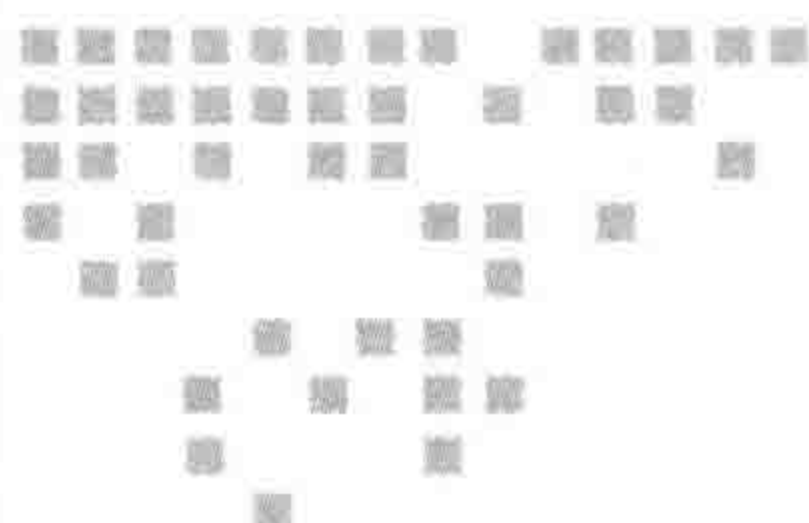
142. "&" {fprintf(yyout, "(%d, %s) \n", AND, yytext); }
143. "*" {fprintf(yyout, "(%d, %s) \n", STAR, yytext); }
144. "/" {fprintf(yyout, "(%d, %s) \n", DIVOP, yytext); }
145. "%" {fprintf(yyout, "(%d, %s) \n", COMOP, yytext); }
146. "+" {fprintf(yyout, "(%d, %s) \n", PLUS, yytext); }
147. "-" {fprintf(yyout, "(%d, %s) \n", MINUS, yytext); }
148. ">>" {fprintf(yyout, "(%d, %s) \n", SHIFTR, yytext); }
149. "<<" {fprintf(yyout, "(%d, %s) \n", SHIFTL, yytext); }
150. ">" {fprintf(yyout, "(%d, %s) \n", RELG, yytext); }
151. ">=" {fprintf(yyout, "(%d, %s) \n", RELGEQ, yytext); }
152. "<" {fprintf(yyout, "(%d, %s) \n", RELL, yytext); }
153. "<=" {fprintf(yyout, "(%d, %s) \n", RELLEQ, yytext); }
154. "==" {fprintf(yyout, "(%d, %s) \n", EQUOP, yytext); }
155. "!=" {fprintf(yyout, "(%d, %s) \n", UEQUOP, yytext); }
156. "^" {fprintf(yyout, "(%d, %s) \n", XOR, yytext); }
157. "|" {fprintf(yyout, "(%d, %s) \n", OR, yytext); }
158. "&&" {fprintf(yyout, "(%d, %s) \n", ANDAND, yytext); }
159. "||" {fprintf(yyout, "(%d, %s) \n", OROR, yytext); }
160. "?" {fprintf(yyout, "(%d, %s) \n", QUEST, yytext); }
161. "=" {fprintf(yyout, "(%d, %s) \n", EQUAL, yytext); }
162. "/=" {fprintf(yyout, "(%d, %s) \n", ASSIGNDIV, yytext); }
163. "*=" {fprintf(yyout, "(%d, %s) \n", ASSIGNSTAR, yytext); }
164. "%=" {fprintf(yyout, "(%d, %s) \n", ASSIGNCOM, yytext); }
165. "+=" {fprintf(yyout, "(%d, %s) \n", ASSIGNPLUS, yytext); }
166. "-=" {fprintf(yyout, "(%d, %s) \n", ASSIGNMINUS, yytext); }
167. "&=" {fprintf(yyout, "(%d, %s) \n", ASSIGNAND, yytext); }
168. "^=" {fprintf(yyout, "(%d, %s) \n", ASSIGNXOR, yytext); }
169. "|=" {fprintf(yyout, "(%d, %s) \n", ASSIGNOR, yytext); }
170. "," {fprintf(yyout, "(%d, %s) \n", COMMA, yytext); }
171. "#" {fprintf(yyout, "(%d, %s) \n", SHA, yytext); }
172. ";" {fprintf(yyout, "(%d, %s) \n", SEMI, yytext); }
173. ":" {fprintf(yyout, "(%d, %s) \n", COLON, yytext); }
174. "." {fprintf(yyout, "Unrecognized character: %s\n", yytext); }
175. %%
176. main()
177. { yyin = fopen("test.txt", "r");
178. yyout = fopen("result.txt", "w");
179. if (yyin != NULL) return yylex();
180. }

```

### 3.4 本章小结

词法分析是编译过程的第一个阶段，负责对源程序进行扫描，按照源程序的构词规则识别出一个个单词符号。编译过程中执行词法分析的程序称为词法分析器。本章介绍了构造词法分析器的两种方法：一种是基于状态转换图的词法分析器的手工实现，另一种是利用词法分析程序的自动生成工具 LEX 的词法分析器的自动实现。





## 语法分析器的设计与实现

语法分析是编译的核心部分，其任务是检查由词法分析器给出的单词符号序列是否是给定文法的正确句子，也即是否符合源语言的语法规则。执行语法分析任务的程序称为语法分析程序，也称为语法分析器。判断某个单词序列是否是源语言的句子，主要有两种方法：一种方法是从文法的开始符号出发，一步步推导出这个单词序列，这种分析方法称为自上而下的语法分析；另一种方法是从单词序列开始逐步归约为文法的开始符号，这种分析方法称为自下而上的语法分析。本章将分别介绍自上而下的语法分析器和自下而上的语法分析器的设计与实现，并介绍语法分析器的自动生成工具 YACC 软件。

### 4.1 自上而下的语法分析器的设计与实现

所谓自上而下的语法分析方法是指对于一个给定的输入单词符号串，尝试从文法的开始符号出发，寻求该串的一个最左推导，或者试图从根结点出发，自上而下地为该串建立一棵语法树。其本质上是一种试探过程，是一种反复使用不同产生式谋求匹配输入串的过程，对文法的限制比较多。

含有左递归的文法将会使上述的自上而下的语法分析过程陷入无限循环，因此，要进行自上而下的语法分析，文法不得含有左递归。如果非终结符  $A$  有多个候选式存在公共前缀，则自上而下的语法分析无法根据当前输入符号准确地选择用于推导的产生式，只能试探。当试探不成功时就需要退回到上一步的推导，看  $A$  是否还有其他候选式，这就是回溯。由于回溯的存在，可能在已经做了大量的语法分析工作之后，才发现走了一大段错路而必须回头，即把已经做过的一大堆语义工作（指中间代码产生工作和各种表格的登记工作）推



倒重来。回溯使得自上而下语法分析只具有理论意义而无实际使用的价值。因此,要使自上而下语法分析具有实用性就要消除回溯。

预测分析法又称 LL(1) 分析法,是一种不带回溯的非递归自上而下分析法。其基本思想是根据输入串的当前输入符号来唯一确定选用某个产生式来进行推导;当这个输入符号与推导的第一个符号相同时,再取输入串的下一个符号,继续确定下一个推导应选的产生式;如此下去,直到推导出被分析的输入串为止。预测分析程序采用预测分析法实现语法分析,又称为预测分析器或者 LL(1) 分析器。接下来我们将以预测分析器的设计与实现为例来说明如何进行自上而下的语法分析器设计与实现。

预测分析程序采用表驱动的方式来实现,包含一个输入缓冲区、一个输出缓冲区、一个符号栈、一个预测分析表(又称为 LL(1) 分析表)和一个总控程序,如图 4-1 所示。其中:

1) 输入缓冲区用来存放待分析的符号串,它以界符‘#’作为结束标志(‘#’不是文法的终结符,我们总把它当作输入串的开始符)。

2) 输出缓冲区用来存放分析过程中所使用的产生式序列。

3) 符号栈中存放分析过程中的文法符号,分析开始时栈底先放入一个‘#’,然后再压入文法的开始符号;当分析栈中仅剩‘#’,且输入串指针也指向待分析串尾的‘#’时,分析成功。

4) 预测分析表用一个矩阵(或二维数组)  $M[A, a]$  表示,其中  $A$  为非终结符,而  $a$  为终结符或‘#’。分析表元素  $M[A, a]$  中的内容为一条关于  $A$  的产生式,表明当  $A$  面临输入符号  $a$  时当前推导所应采用的候选式;当元素内容为空白(空白表示“出错标志”)时,则表明  $A$  不应该面临这个输入符号  $a$ ,即输入串含有语法错误。

5) 总控程序根据符号栈栈顶符号  $X$  和当前输入符号  $a$  来决定分析器的动作:

- ① 若  $X=a=‘#’$ ,则分析成功,分析器停止工作。
- ② 若  $X=a \neq ‘#’$ ,即栈顶符号  $X$  与当前扫描的输入符号  $a$  匹配,将  $X$  从栈顶弹出,输入指针指向下一个输入符号,继续对下一个字符进行分析。
- ③ 若  $X$  为非终结符  $A$ ,则查  $M[A,a]$ :
  - a) 若  $M[A,a]$  中为一个  $A$  的产生式,则将  $A$  自栈顶弹出,并将  $M[A,a]$  中的产生式右部符号串按逆序逐一压入栈中;如果  $M[A,a]$  中的产生式为  $A \rightarrow \epsilon$ ,则只将  $A$  自栈顶弹出。
  - b) 若  $M[A,a]$  中为空,则发现语法错误,调用出错处理程序进行处理。

给定一个上下文无关文法,在 LL(1) 分析中,必须先求出首符号集和后继终结符集;然后,构造预测分析表;最后,总控程序根据符号栈栈顶符号和当前输入符号来依据预测

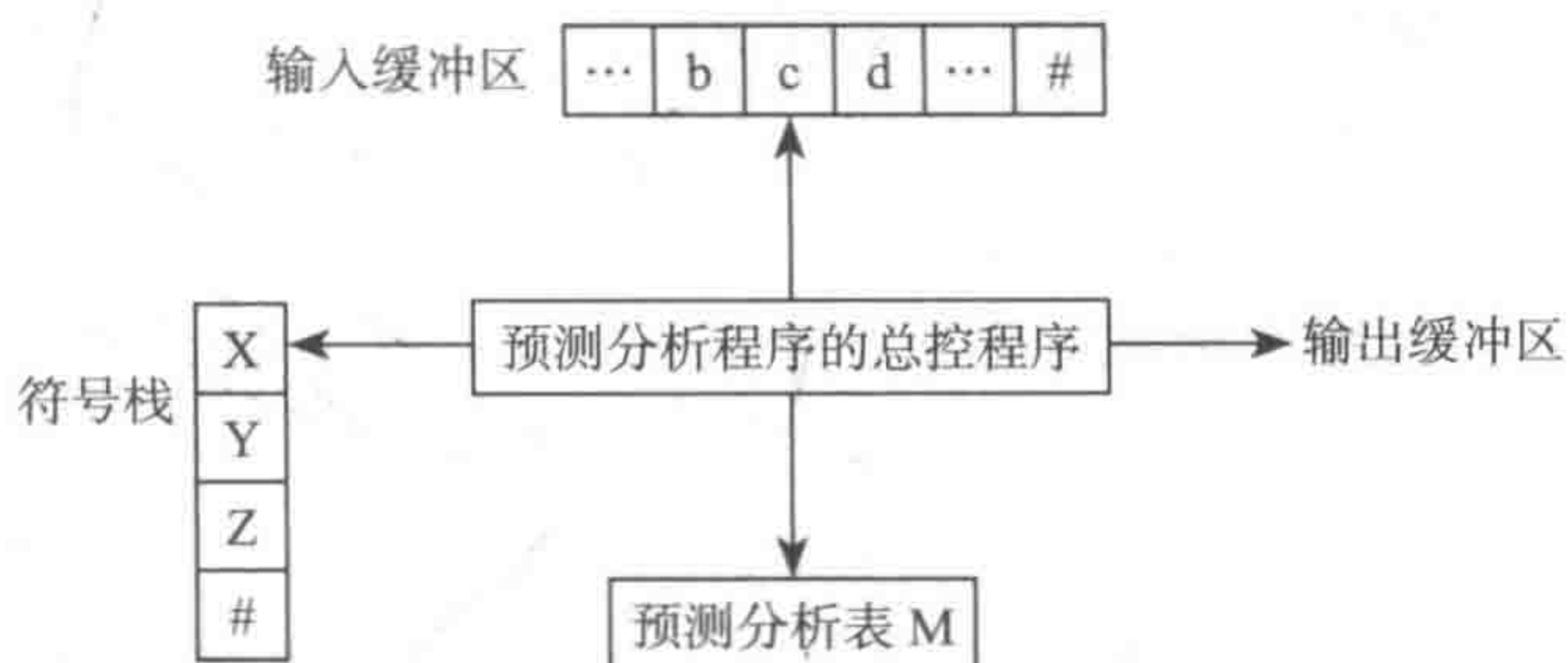


图 4-1 预测分析程序模型



分析表进行分析。

**定义 4.1 (首符号集)** 对一个给定的文法  $G(S)=(V_T, V_N, P, S)$ , 其所有非终结符的每一个候选式  $\alpha$  的首符号集  $FIRST(\alpha)$  定义如下:

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow * a \cdots, a \in V_T\}$$

特别是  $\alpha \Rightarrow * \varepsilon$  时, 规定  $\varepsilon \in FIRST(\alpha)$ 。也即,  $FIRST(\alpha)$  是  $\alpha$  的所有可能推导的开头终结符或可能的  $\varepsilon$ 。假设  $A$  是文法  $G$  的一个非终结符, 其共有  $n$  个候选式  $\alpha_1, \alpha_2, \dots, \alpha_n$ , 即  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ , 则

$$FIRST(A) = FIRST(\alpha_1) \cup \dots \cup FIRST(\alpha_n)$$

算法 4.1 和算法 4.2 分别用来计算单个文法符号  $X$  和文法符号串  $\alpha$  的  $FIRST$  集。

#### 算法 4.1 计算 $FIRST(X)$

输入: 文法  $G=(V_T, V_N, P, S)$ ,  $X \in (V_T \cup V_N)$

输出:  $FIRST(X)$

步骤: 对每一文法符号  $X \in V_T \cup V_N$ , 连续使用下面的规则, 直至每个集合  $FIRST(X)$  不再增大为止:

1. 若  $X \in V_T$ , 则  $FIRST(X) = \{X\}$ 。
2. 若  $X \in V_N$ , 且有产生式  $X \rightarrow a \cdots$ , 则把  $a$  加入  $FIRST(X)$ ;
3. 若  $X \in V_N$ , 且  $X \rightarrow \varepsilon$ , 则把  $\varepsilon$  加入  $FIRST(X)$ ;
4. 若  $X \in V_N$ , 且  $X \rightarrow Y \cdots$  是一个产生式,  $Y \in V_N$ , 则把  $FIRST(Y)$  中的所有非  $\varepsilon$ -元素都加到  $FIRST(X)$  中;
5. 若  $X \in V_N$ ,  $X \rightarrow Y_1 Y_2 \cdots Y_k$  是一个产生式,  $Y_1, \dots, Y_{i-1} \Rightarrow * \varepsilon$ , 则把  $FIRST(Y_j)$  ( $1 \leq j \leq i$ ) 中的所有非  $\varepsilon$ -元素都加到  $FIRST(X)$  中; 特别是, 若所有的  $FIRST(Y_j)$  均含有  $\varepsilon$  ( $1 \leq j \leq k$ ), 则把  $\varepsilon$  加到  $FIRST(X)$  中。

#### 算法 4.2 计算 $FIRST(\alpha)$

输入: 文法  $G=(V_T, V_N, P, S)$ ,  $\alpha \in (V_T \cup V_N)^*$ ,  $\alpha = X_1 \cdots X_n$

输出:  $FIRST(\alpha)$

步骤:

1. 计算  $FIRST(X_1)$ ;
2.  $FIRST(\alpha) := FIRST(X_1) - \{\varepsilon\}$ ;
3.  $k := 1$ ;
4. while ( $\varepsilon \in FIRST(X_k)$  and  $k < n$ ) do
5.    $\{ FIRST(\alpha) := FIRST(\alpha) \cup (FIRST(X_{k+1}) - \{\varepsilon\})$ ;
6.    $k := k + 1$  }
7. if ( $k = n$  and  $\varepsilon \in FIRST(X_k)$ ) then  $FIRST(\alpha) := FIRST(\alpha) \cup \{\varepsilon\}$ ;



例 4.1 对于文法 (4.1), 构造每个非终结符以及各个非终结符的候选式的 FIRST 集。

- (1)  $E \rightarrow TG$
  - (2)  $G \rightarrow +TG$
  - (3)  $G \rightarrow \varepsilon$
  - (4)  $T \rightarrow FH$
  - (5)  $H \rightarrow *FH$
  - (6)  $H \rightarrow \varepsilon$
  - (7)  $F \rightarrow (E)$
  - (8)  $F \rightarrow i$
- (4.1)

解: 由算法 4.1 的步骤 1 可知, 对于每一个终结符有

$\text{FIRST}(+) = \{+\}$   
 $\text{FIRST}(*) = \{*\}$   
 $\text{FIRST}(( ) ) = \{( )\}$   
 $\text{FIRST}() = \{ ) \}$   
 $\text{FIRST}(i) = \{i\}$

由算法 4.1 的步骤 2 和步骤 3 可知

$\text{FIRST}(G) = \{+, \varepsilon\}$   
 $\text{FIRST}(H) = \{*, \varepsilon\}$   
 $\text{FIRST}(F) = \{(, i\}$

因为  $T \rightarrow FH$  和  $E \rightarrow TG$ , 由算法 4.1 的步骤 4 和步骤 5 可知,  $\text{FIRST}(F) \subseteq \text{FIRST}(T) \subseteq \text{FIRST}(E)$ , 且有  $\text{FIRST}(T) = \text{FIRST}(E) = \{(, i\}$ 。

由算法 4.2 易得对各个非终结符的候选式有如下结果:

$\text{FIRST}(TG) = \{(, i\}$   
 $\text{FIRST}(+TG) = \{+\}$   
 $\text{FIRST}(FH) = \{(, i\}$   
 $\text{FIRST}(*FH) = \{*\}$   
 $\text{FIRST}((E)) = \{( )\}$   
 $\text{FIRST}(\varepsilon) = \{\varepsilon\}$

下面重点介绍计算单个文法符号 FIRST 集的实现, 至于计算文法符号串的 FIRST 集的实现与计算单个文法符号 FIRST 集的实现类似, 不再详细介绍, 留作作业。为便于阅读, 对下面程序中所涉及的过程作以下说明:

1) `bool isterminal(char x)` 用来判断一个符号是否为终结符, 这里有一点需要说明的是, 在编程实现过程中用 “~” 表示空, 并且将其作为终结符处理。

2) `bool exist(char x)` 判断符号  $x$  是否从未出现过。

3) `void read()` 读入文法。

- 4) void show() 输出用户输入的文法。
- 5) int char\_id(char x) 把符号 x 变为对应的编号。
- 6) bool in (struct set &st, char id) 判断集合 st 里面是否包含符号 id。
- 7) void add(struct set &st, char e) 把符号 e 添加到集合 st 里面。
- 8) void compute\_first() 求每个非终极符的 FIRST 集合。
- 9) void print\_first(struct set \*st) 输出每个非终极符的 FIRST 集合。

算法 4.1 的具体实现代码如下:

```
#include<iostream>
#include"string.h"
#define MAX 100
#define MaxVtNum 20/* 终结符最大的数目 */
#define MaxVnNum 20 /* 非终结符最大的数目 */
#define MaxPLength 20/* 产生式的右部最大长度 */
using namespace std;
struct product
{ int length; /* 产生式右部的文法符号串的长度 */
  char left; /* 产生式左部的非终结符 */
  char right[MaxPLength]; /* 产生式右部的文法符号串 */
}p[100]; /* 产生式结构体 */
struct set
{
  int n; /* first 和 follow 集中元素的个数 */
  char elm[100];
}first[MAX], follow[MAX]; /* first 和 follow 集结构体 */
char termin[MaxVtNum]; /* 终结符 */
char non_termin[MaxVnNum]; /* 非终结符 */
int n; /* 产生式数量 */
int VtNum; /* 终结符数量 */
int VnNum; /* 非终结符数量 */
/* 判断是否为终结符 */
inline bool isterminal(char x)
{
  if(x>='A'&&x<='Z') return false;
  return true;
}
/* 判断符号 x 是否出现过 */
bool exist(char x)
{
  int i;
  if(isterminal(x))
  {
    for(i=1;i<= VtNum;i++)
      if(termin[i]==x) return true;
    return false;
  }
  for(i=1;i<= VnNum;i++)
    if(non_termin[i]==x) return true;
```



```

        return false;
    }
    /* 读入文法 */
    void read()
    {
        int i, j, k;
        char tmp[25];
        printf(" 输入产生式的数量: ");
        scanf("%d",&n);
        printf(" 输入形如 A->abc 的产生式, 其中 ~ 表示空: \n");
        for(VnNum=VtNum=0,i=1;i<=n;i++)
        {
            scanf("%s",tmp);
            p[i].left=tmp[0];
            if(!exist(tmp[0])) non_termin[++VnNum]=tmp[0];
            for(k=0,j=3;tmp[j];j++)
            {
                p[i].right[k++]=tmp[j];
                if(isterminal(tmp[j]))
                    if(!exist(tmp[j])) termin[++VtNum]=tmp[j];
                else
                    if(!exist(tmp[j])) non_termin[++VnNum]=tmp[j];
            }
            p[i].right[k]=0;
            p[i].length=k-1;
        }
        termin[++VtNum]=non_termin[++VnNum]='#';
    }
    /* 输出用户所输入的文法 */
    void show()
    {
        int i;
        printf(" 用户所输入的产生式为: \n");
        for(i=1;i<=n;i++)
            printf("(%d)  %c->%s\n",i, p[i].left,p[i].right);
    }
    /* 把符号 x 变为对应的编号 */
    int char_id(char x)
    {
        int i;
        if(!isterminal(x))
        {
            for(i=1;i<=VnNum;i++)
                if(non_termin[i]==x) return i;
        }
        for(i=1;i<=VtNum;i++)
            if(termin[i]==x) return i+1000;
        return -1;
    }
    /* 判断符号 idt 是否在集合 st 里面 */

```

```

bool in (struct set &st, char id)
{
    int i;
    for(i=1;i<=st.n;i++)
        if(st.elm[i]==id)
            return true;
    return false;
}
/* 把符号 e 添加到集合 st 里面 */
void add(struct set &st, char e)
{
    st.n++;
    st.elm[st.n]=e;
}
/* 求 first 集 */
void compute_first()
{
    int i, j, k, idl, idr;
    bool inc;
    inc=true;
    while(inc)
    {
        inc=false;
        for(i=1;i<=n;i++)// 遍历所有产生式
        {
            idl=char_id(p[i].left);
            for(j=0;p[i].right[j];j++)
            {
                idr=char_id(p[i].right[j]);
                /* 如果当前为终结符, 并且 first[idl] 中不包含该终结符,
                则把此终结符加入 first[idl] */
                if(idr>1000)
                {
                    if(!in(first[idl], p[i].right[j]))
                    {
                        add(first[idl], p[i].right[j]);
                        inc=true;
                    }
                }
                break;
            }
            /* 否则把该非终结符 first 集里面非空元素加入 first[idl] */
            else
            {
                for(k=1;k<=first[idr].n;k++)
                {
                    if(!in(first[idl], first[idr].elm[k])&&first[idr].elm[k]!='~')
                    {
                        add(first[idl], first[idr].elm[k]);
                        inc=true;
                    }
                }
            }
        }
    }
}

```



```

        if(!in(first[id1], '~'))
            break;
    }
}
/* 若产生式右部的每一个文法符号都可以推导出空, 则 '~' 应属于 first[id1] */
if(p[i].right[j]==0&&!in(first[id1], '~'))
{
    add(first[id1], '~');
    inc=true;
}
}
}
/* 输出每个非终极符的 first 集合 */
void print_first(struct set *st)
{
    int i, j;
    puts("\n");
    for(i=1; i<=VnNum; i++)
    {
        printf("%s(%c):  ", FIRST, non_termin[i]);
        for(j=1; j<=st[i].n; j++)
            printf("%c  ", st[i].elm[j]); puts("");
    }
}

int main()
{
    read();
    show();
    compute_first();
    print_first(first);
    return 0;
}

```

输入文法 (4.1), 运行上述程序, 其运算结果如图 4-2 所示。

**定义 4.2 (后继终结符集)** 对一个给定文法  $G(S)$ ,  $A$  是其一个非终结符,  $A$  的后继终结符集  $FOLLOW(A)$  定义如下

$$FOLLOW(A) = \{a \mid S \Rightarrow * \dots Aa \dots, a \in V_T\}$$

如果  $A$  是某个句型的最右符号, 也即  $S \Rightarrow * \dots A$ , 那么 ‘#’ 属于  $FOLLOW(A)$ 。换句话说,  $FOLLOW(A)$  是  $G(S)$  的所有句型中出现在紧接  $A$  之后的终结符或 ‘#’。

算法 4.3 给出了用来计算非终结符  $A$  的  $FOLLOW$  集的方法。

```

输入产生式的数量: 8
输入形如a-abc的产生式, 其中~表示空:
E->TG
G->+TG
G->~
T->FH
H->*FH
H->~
P-><E>
F->i
用户所输入的产生式为:
(1) E->TG
(2) G->+TG
(3) G->~
(4) T->FH
(5) H->*FH
(6) H->~
(7) P-><E>
(8) F->i

FIRST(E):  ( i
FIRST(T):  ( i
FIRST(G):  + ~
FIRST(P):  ( i
FIRST(H):  * ~
Press any key to continue

```

图 4-2 针对文法 (4.1) 计算单个文法符号 FIRST 集的程序运行结果

**算法 4.3 计算 FOLLOW 集**

输入：文法  $G=(V_T, V_N, P, S)$ ,  $A \in V_N$

输出：FOLLOW(A)

步骤：

1. 对  $A \in V_N$ ,  $\text{FOLLOW}(A) := \emptyset$ ;
2.  $\text{FOLLOW}(S) := \{\#\}$ , # 为句子的结束符;
3. 对  $A \in V_N$ , 重复下面的第 4 步到第 5 步, 直到所有 FOLLOW 集不变为止。
4. 若  $A \rightarrow \alpha B \beta \in P$ , 则  $\text{FOLLOW}(B) := \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) \setminus \{\epsilon\})$ ;
5. 若  $A \rightarrow \alpha B$  或  $A \rightarrow \alpha B \beta \in P$  且  $\beta \Rightarrow^* \epsilon$  (即  $\epsilon \in \text{FIRST}(\beta)$ ), 则  $\text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$ ;

**例 4.2** 对于文法 (4.1), 构造每个非终结符的 FOLLOW 集。

**解：**依据例 4.1 中的计算结果, 构造 FOLLOW 集的步骤如下:

- 1)  $\text{FOLLOW}(E) = \{\#\}$ ;
- 2) 由  $E \rightarrow TG$  知  $\text{FIRST}(G) \setminus \{\epsilon\} \subseteq \text{FOLLOW}(T)$ , 即  $\text{FOLLOW}(T) = \{+\}$ ;
- 3) 由  $T \rightarrow FH$  知  $\text{FIRST}(H) \setminus \{\epsilon\} \subseteq \text{FOLLOW}(F)$ , 即  $\text{FOLLOW}(F) = \{*\}$ ;
- 4) 由  $F \rightarrow (E)$  知  $\text{FIRST}() \subseteq \text{FOLLOW}(E)$ , 即  $\text{FOLLOW}(E) = \{), \#\}$ ;
- 5) 由  $E \rightarrow TG$  知  $\text{FOLLOW}(E) \subseteq \text{FOLLOW}(G)$ , 即  $\text{FOLLOW}(G) = \{), \#\}$ ;
- 6) 由  $E \rightarrow TG$  且  $G \rightarrow \epsilon$  知  $\text{FOLLOW}(E) \subseteq \text{FOLLOW}(T)$ , 即  $\text{FOLLOW}(T) = \{+, ), \#\}$ ;
- 7) 由  $T \rightarrow FH$  知  $\text{FOLLOW}(T) \subseteq \text{FOLLOW}(H)$ , 即  $\text{FOLLOW}(H) = \{+, ), \#\}$ ;
- 8) 由  $T \rightarrow FG$  且  $G \rightarrow \epsilon$  知  $\text{FOLLOW}(T) \subseteq \text{FOLLOW}(F)$ , 即  $\text{FOLLOW}(F) = \{*, +, ), \#\}$ ;

最终的计算结果如下:

$\text{FOLLOW}(E) = \{), \#\}$

$\text{FOLLOW}(G) = \{), \#\}$

$\text{FOLLOW}(T) = \{+, ), \#\}$

$\text{FOLLOW}(H) = \{+, ), \#\}$

$\text{FOLLOW}(F) = \{*, +, ), \#\}$

在下面给出的关于计算单个文法符号 FOLLOW 集的实现中所涉及的全局变量或者过程, 如果未作特别说明, 同前面计算单个文法符号 FIRST 集的具体实现中所涉及的相同。下面的代码中有一个在前面代码中未出现的过程 `void print_follow(struct set *st)`, 其功能是输出每个非终极符的 FOLLOW 集合。

算法 4.3 的具体实现代码如下:

```
/* 求 FOLLOW 集 */
void compute_follow()
{
    int i, j, k, idl, idr, idf;
```



```

bool flag, inc=true;
add(follow[1], '#'); /* 把结束标志 '#' 加入开始符号的 FOLLOW 集 */
while(inc)
{
    inc=false;
    for(i=1; i<=n; i++)
    {
        idl=char_id(p[i].left);
        for(flag=true, j=p[i].length; j>=0; j--)
        {
            idr=char_id(p[i].right[j]);
            if(idr>1000)
            {
                flag=false;
                continue;
            }
            if(flag)
            {
                for(k=1; k<=follow[idl].n; k++)
                {
                    if(!in(follow[idr], follow[idl].elm[k]))
                    {
                        add(follow[idr], follow[idl].elm[k]);
                        inc=true;
                    }
                }
            }
            if(j<p[i].length) idf=cid(p[i].right[j+1]);
            else continue;
            if(idf>1000)
            {
                if(!in(follow[idr], p[i].right[j+1]))
                {
                    add(follow[idr], p[i].right[j+1]);
                    continue;
                }
            }
            for(k=1; k<=first[idf].n; k++)
            {
                if(!in(follow[idr], first[idf].elm[k]) && first[idf].
elm[k] != '~')
                {
                    add(follow[idr], first[idf].elm[k]); inc=true;
                }
            }
        }
    }
}

/* 输出每个非终极符号的 FOLLOW 集合 */
void print_follow(struct set *st)
{

```

```

int i, j;
puts("\n");
for(i=1;i<=VnNum;i++)
{
    printf("%s(%c):  ", FOLLOW, non_termin[i]);
    for(j=1;j<=st[i].n;j++)
        printf("%c  ", st[i].elm[j]); puts("");
}

int main()
{
    read();
    show();
    compute_follow();
    print_follow(follow);
    return 0;
}

```

输入文法 (4.1), 运行上述程序, 其运算结果如图 4-3 所示。

在表驱动的预测分析器中, 除了预测分析表因文法的不同而异之外, 符号栈、总控程序都是相同的。因此, 构造一个文法的预测分析器实际上就是构造该文法的预测分析表。构造预测分析表的主要思想如下:

如果  $A \rightarrow \alpha$  是产生式, 当  $A$  呈现于栈顶:

1) 当前输入符号  $a \in \text{FIRST}(\alpha)$  时,  $\alpha$  应被选作  $A$  的唯一代表, 即用  $\alpha$  展开  $A$ , 所以  $M[A, a]$  中应放入产生式  $A \rightarrow \alpha$ 。

2) 当  $\alpha \Rightarrow * \varepsilon$  时, 如果当前输入符号  $b$  (包括  $\#$ )  $\in \text{FOLLOW}(A)$ , 则认为  $A$  自动得到匹配, 因此, 应把产生式  $A \rightarrow \alpha$  放入  $M[A, b]$  中。

根据上述思想, 在对文法  $G$  的每个非终结符  $A$  及其任意候选  $\alpha$  都求出  $\text{FIRST}(\alpha)$  和  $\text{FOLLOW}(A)$  之后, 便可得到预测分析表的构造方法, 如算法 4.4 所示。

利用算法 4.4, 可以构造任何文法的分析表, 但对于某些文法, 有些  $M[A, a]$  中可能有若干条产生式, 这称为分析表的多重定义或者多重入口。一个文法  $G(S)$ , 若它的分析表  $M$  不含多重定义入口, 则称它是一个  $\text{LL}(1)$  文法, 它所定义的语言恰好就是它的分析表所能识别的全部句子。如果  $G$  是左递归或二义的, 那么,  $M$  至少含有一个多重定义入口。

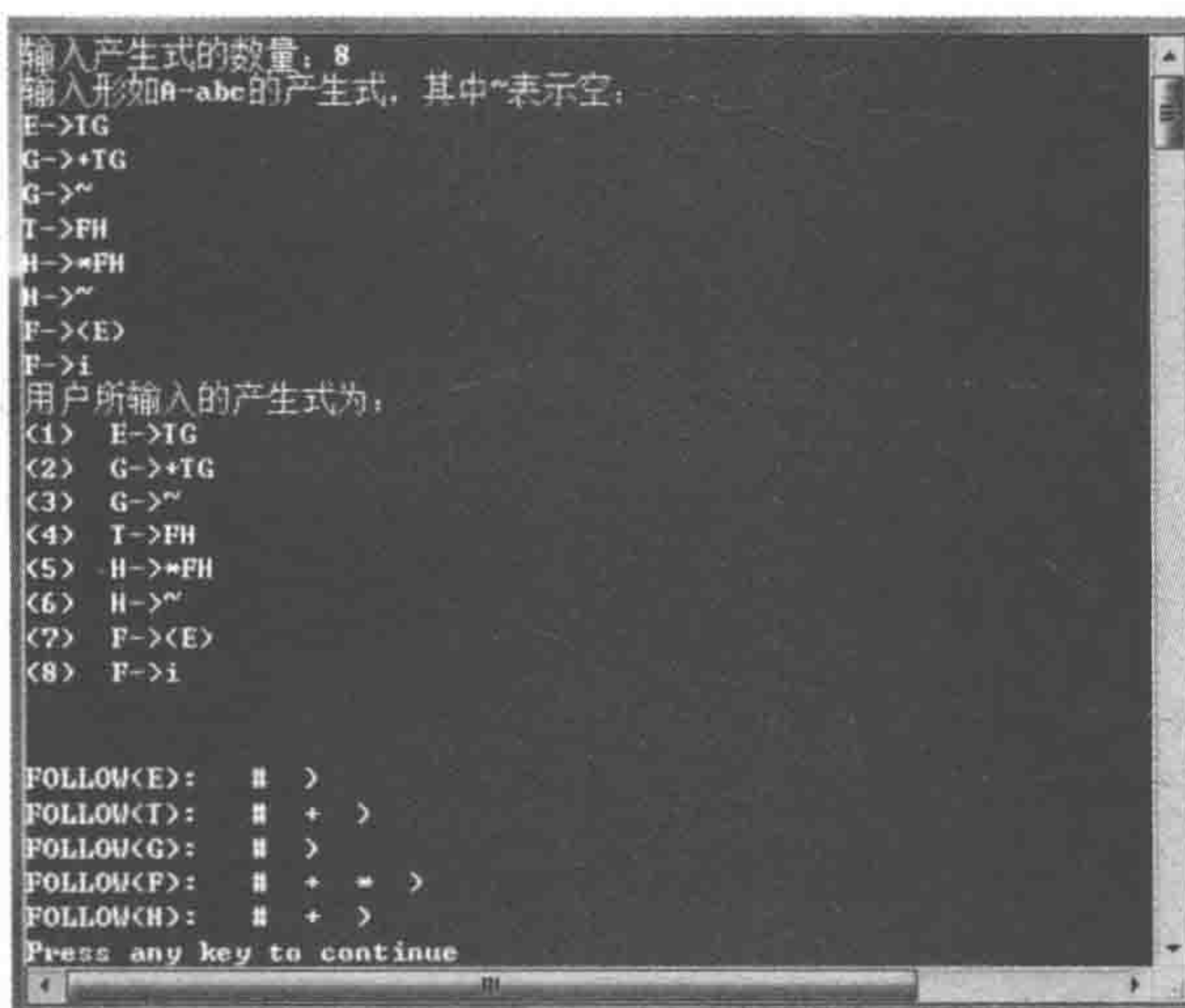


图 4-3 针对文法 (4.1) 计算单个文法符号 FOLLOW 集的程序运行结果



**算法 4.4 构造预测分析表**

输入：文法  $G=(V_T, V_N, P, S)$

输出：文法  $G$  的预测分析表  $M$

步骤：

1. for 文法  $G$  的每个产生式  $A \rightarrow \alpha$ 
  - {
  - 2.   for 每个终结符号  $a \in \text{FIRST}(\alpha)$
  - 3.     把  $A \rightarrow \alpha$  放入  $M[A, a]$  中；
  - 4.   if  $\epsilon \in \text{FIRST}(\alpha)$  then
  - 5.     { for 任何  $b \in \text{FOLLOW}(A)$
  - 6.       把  $A \rightarrow \alpha$  放入  $M[A, b]$  中； }
  - }
7. for 所有无定义的  $M[A, a]$
8.  标上错误标志。

**例 4.3** 对于文法 (4.1)，构造其相应的预测分析表。

**解：**根据算法 4.4 检查文法的每一个产生式。

1) 对产生式  $E \rightarrow TG$ ：由例 4.1 中的计算结果可知， $\text{FIRST}(TG) = \{ (, i \}$ ，故应把  $E \rightarrow TG$  放入  $M[E, (]$  和  $M[E, i]$  中。

2) 对于产生式  $G \rightarrow +TG$ ：同样由例 4.1 中的计算结果可知， $\text{FIRST}(+TG) = \{ + \}$ ，所以，应把  $G \rightarrow +TG$  放入  $M[G, +]$  中。

3) 对于产生式  $G \rightarrow \epsilon$ ：由于  $\text{FOLLOW}(G) = \{ ), \# \}$ ，故应把  $G \rightarrow \epsilon$  放入  $M[G, )]$  和  $M[G, \#]$  中。

4) 依次检查其余的产生式，就可得如表 4-1 所示的预测分析表。

表 4-1 文法 (4.1) 的预测分析表

	i	+	*	(	)	#
E	$E \rightarrow TG$			$E \rightarrow TG$		
E'		$G \rightarrow +TG$			$G \rightarrow \epsilon$	$G \rightarrow \epsilon$
T	$T \rightarrow FH$			$T \rightarrow FH$		
T'		$H \rightarrow \epsilon$	$H \rightarrow *FH$		$H \rightarrow \epsilon$	$H \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

同样在下面给出的构造预测分析表的实现中所涉及的全局变量或者过程，如果未作特别说明，同前面计算单个文法符号  $\text{FIRST}$  集的实现中所涉及的相同。下面的代码中有一个在前面代码中未出现过的过程 `void bulid_table()`，其功能是构造输入文法的预测分析表。

算法 4.4 的具体实现代码如下:

```
int table[MAX][MAX]; /* 用来存储预测分析表 */
void bulid_table()
{
    int i, j, k, idl, idr, idt;
    char ch;
    bool flag;
    memset(table, 0, sizeof(table));
    table[VnNum][VtNum] = -1;
    for(i=1; i<=n; i++)
    {
        idl = char_id(p[i].left);
        for(j=0; j<=p[i].length; j++)
        {
            ch = p[i].right[j];
            idr = char_id(ch);
            if(idr > 1000) /* 产生式右部的第一个文法符号为终结符,
                           此处仍将空串作为终结符处理 */
            {
                idr -= 1000;
                if(ch != '~') /* 产生式右部的第一个文法符号为非空终结符 */
                {
                    if(table[idl][idr])
                    {
                        printf("It's not a LL(1) language.\n");
                        return;
                    }
                    table[idl][idr] = i;
                }
            }
            else /* 产生式右部的第一个文法符号为空 */
            {
                for(k=1; k<=follow[idl].n; k++) /* 将产生式左部文法符号和其
                                                  FOLLOW集中的所有符号所对应位置都填入该产生式 */
                {
                    idt = char_id(follow[idl].elm[k]) - 1000;
                    if(table[idl][idt])
                    {
                        printf("It's not a LL(1) language.\n");
                        return;
                    }
                    table[idl][idt] = i;
                }
            }
            break;
        }
        /* 产生式右部的第一个文法符号为非终结符 */
        for(flag=false, k=1; k<=first[idr].n; k++)
        {
            idt = char_id(first[idr].elm[k]) - 1000;
```



```

        if(first[idr].elm[k]=='~')
            flag=true;
        if(table[idl][idt])
        {
            printf("It's not a LL(1) language.\n");
            return;
        }
        table[idl][idt]=i;
    }
    if(!flag)
        break;
}
if(j>p[i].length)
{
    for(k=1;k<=follow[idl].n;k++)
    {
        idt=char_id(follow[idl].elm[k])-1000;
        if(table[idl][idt])
        {
            printf("It's not a LL(1) language.\n");
            return;
        }
        table[idl][idt]=i;
    }
}
}
return;
}
int main()
{
    int i,j;
    read();
    show();
    compute_first();
    compute_follow();
    bulid_table();
    printf("\t|\t");
    for(j=1;j<=VtNum;j++)
    {
        if(termin[j]!='~')
            printf("%c\t", termin[j]);
    }
    printf("\n");
    for(i=1;i<=VtNum;i++)
        printf("- - -\t");
    printf("\n");
    for(i=1; i< VnNum; i++)
    {
        printf("%c\t|\t", non_termin[i]);
        for(j=1;j<=tnum;j++)

```

```
{
    if(termin[j]!='~')
    if(table[i][j]>0)
        printf("%d\t",table[i][j]);
    else
        printf("error\t");
}
printf("\n");
}
return 0;
}
```

输入文法（4.1），运行上述程序，其运算结果如图 4-4 所示。

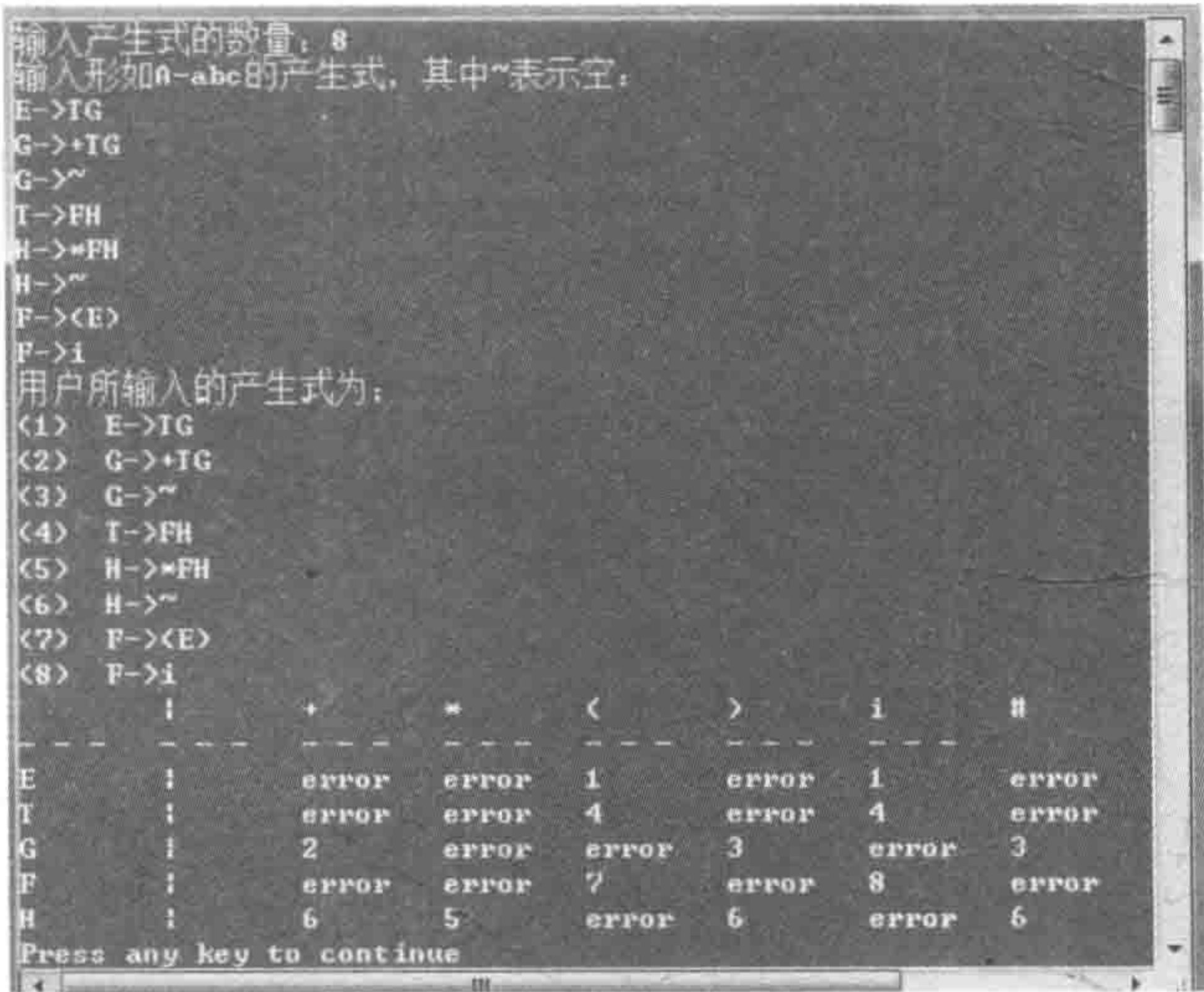


图 4-4 针对文法（4.1）构造预测分析表的程序运行结果

对于任意一个给定的文法  $G=(V_T, V_N, P, S)$ ，其预测分析程序的总控程序可以用算法 4.5 来描述。

算法 4.5 预测分析程序的总控程序。

输入：输入串  $w$  和文法  $G=(V_T, V_N, P, S)$  的预测分析表  $M$

输出：如果  $w$  属于  $L(G)$ ，则输出  $w$  的最左推导，否则报告错误

步骤：

- 1. 将栈底符号 # 和文法开始符号  $S$  压入栈中；
- 2. repeat
- 3.    $X:=$  当前栈顶符号；
- 4.    $a:=$  当前输入符号；
- 5.   if  $X \in \{V_T \cup \{\#\}\}$  then



```

6.    if X==a then
7.        { if X != # then
8.            { 将 X 弹出栈;
9.                前移输入指针 }}
10.   else error
11.   else
12.   if M[X, a]=X → Y1Y2…Yk then
13.       { 将 X 弹出栈;
14.         依次将 Yk, …, Y2, Y1 压入栈;
15.         输出产生式 X → Y1Y2…Yk }
16.   else error
17. until X=#

```

例 4.4 由例 4.3 可知文法 (4.1) 的预测分析表如表 4-1 所示, 试对输入串 “ $i_1*i_2+i_3$ ” 利用分析表 4-1 进行预测分析。

解: 利用分析表对输入串 “ $i_1*i_2+i_3$ ” 进行预测分析的步骤如表 4-2 所示。

表 4-2 对输入串 “ $i_1*i_2+i_3$ ” 进行预测分析的过程

步骤	符号栈	输入缓冲区	输出	步骤	符号栈	输入缓冲区	输出
0	#E	$i_1*i_2+i_3\#$		9	#G	$+i_3\#$	$H \rightarrow \epsilon$
1	#GT	$i_1*i_2+i_3\#$	$E \rightarrow TG$	10	#GT+	$+i_3\#$	$G \rightarrow +TG$
2	#GHF	$i_1*i_2+i_3\#$	$T \rightarrow FH$	11	#GT	$i_3\#$	+ 匹配成功
3	#GHi	$i_1*i_2+i_3\#$	$F \rightarrow i$	12	#GHF	$i_3\#$	$T \rightarrow FH$
4	#GH	$*i_2+i_3\#$	i 匹配成功	13	#GHi	$i_3\#$	$F \rightarrow i$
5	#GHF*	$*i_2+i_3\#$	$T' \rightarrow *FH$	14	#GH	#	i 匹配成功
6	#GHF	$i_2+i_3\#$	* 匹配成功	15	#G	#	$H \rightarrow \epsilon$
7	#GHi	$i_2+i_3\#$	$F \rightarrow i$	16	acc		$G \rightarrow \epsilon$
8	#GH	$+i_3\#$	i 匹配成功				

在下面给出的构造预测分析表的实现中所涉及的全局变量或者过程, 如果未作特别说明, 同前面计算单个文法符号 FIRST 集的实现中所涉及的相同。下面的代码还涉及如下过程:

- 1) void print\_stack() 用来输出分析栈中的元素。
- 2) void print\_input() 用来输出尚未分析的输入串。
- 3) int non\_termin\_row(char c) 用来将非终结符转换为行号。
- 4) int termin\_col(char c) 用来将终结符转换为列号。
- 5) bool isNT(char c) 判断 c 是否是非终结符。
- 6) bool isT(char c) 判断 c 是否是终结符。

算法 4.5 的具体实现代码如下:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MaxVtNum 20/* 终结符最大的数目 */
#define MaxVnNum 20 /* 非终结符最大的数目 */
#define MaxPLength 20/* 产生式的右部最大长度 */
#define MaxStLength 50/* 待分析输入串的最大长度 */
char stack[20]={'#','E'};/* 对分析栈 stack 赋初值 */
char input[MaxStLength];/* 剩余串 input */
char termin[MaxVtNum]={'i','+','*','(',')','#'};/* 终结符, 此处将 '#' 作为终结符处理 */
char non_termin[MaxVnNum]={'E','G','T','H','F'};/* 非终结符 */
struct product /* 产生式类型定义 */
{
    char left;/* 大写字母 */
    char right[MaxPLength];/* 产生式右边字符 */
    int length;/* 字符个数 */
};
struct product E,T,G,G1,H,H1,F,F1;/* 用来存放各个产生式的结构体变量 */
struct product M[MaxVnNum][MaxVtNum];/* 预测分析表 */
int flag=1;/* 当前正在分析字符在输入串中的下标 */
int top=1;/* 分析栈栈顶指针 */
int l;/* l 为输入串长度 */

void print_stack()/* 输出分析栈 */
{
    int i;/* 指针 */
    for(i=0;i<=top;i++)
        printf("%c",stack[i]);
}

void print_input()/* 输出剩余串 */
{
    int i;
    for(i=0;i<flag-1;i++)/* 输出对齐符 */
        printf(" ");
    for (i=flag-1;i<=l;i++)
        printf("%c",input[i]);
}

int non_termin_row(char c)/* 非终结符转换为行号 */
{
    int i;
    for(i=0;i<(int)strlen(non_termin);i++)
        if (c==non_termin[i])
            return i ;
    printf("Error in non_termin_row()>%c\n",c);
    exit(0) ;
}
```



```

int termin_col(char c)    /* 终结符转换为列号 */
{
    int i;
    for (i=0;i<(int)strlen(termin);i++)
        if (c==termin[i])
            return i;
    printf("Error in termin_col()>%c\n",c);
    exit(0);
}

bool isNT(char c)    /* 判断 c 是否是非终结符 */
{
    int i;
    for (i=0; i<(int)strlen(non_termin);i++)
        if (c==non_termin[i])
            return true;
    return false;
}

bool isT(char c)    /* 判断 c 是否是终结符 (不包括 '#') */
{
    int i;
    for (i=0;i<(int)strlen(termin)-1;i++)
        if (c==termin[i])
            return true;
    return false;
}

void main(void)
{
    int i,j=0,k=0;
    int flag2=0;
    char ch;                /* 指示输入串当前字符 */
    char X = ' ';          /* 存储栈顶字符 */
    struct product cha;
    /* 把文法产生式赋值结构体 */
    E.left='E';
    strcpy(E.right,"TG");
    E.length=2;
    T.left='T';
    strcpy(T.right,"FH");
    T.length=2;
    G.left='G';
    strcpy(G.right,"+TG");
    G.length=3;
    G1.left='G';
    G1.right[0]='~';
    G1.length=1;
    H.left='H';
    strcpy(H.right,"*FH");
    H.length=3;
}

```

```

H1.left='H';
H1.right[0]='~';
H1.length=1;
F.left='F';
strcpy(F.right,"(E)");
F.length=3;
F1.left='F';
F1.right[0]='i';
F1.length=1;
/* 填充分析表 */
M[0][0]=E;M[0][3]=E;
M[1][1]=G;M[1][4]=G1;M[1][5]=G1;
M[2][0]=T;M[2][3]=T;
M[3][1]=H1;M[3][2]=H;M[3][4]=M[3][5]=H1;
M[4][0]=F1;M[4][3]=F;
printf(" 本程序只能对由 'i','+', '*', '(', ')' 构成的以 '#' 结束的字符串进行分析, \n");
printf(" 请输入要分析的字符串:");
do/* 读入分析串 */
{
    scanf("%c",&ch);
    if ((ch!='i') &&(ch!='+') &&(ch!='*')&&(ch!='(')&&(ch!=')')&&(ch!='#'))
    {
        printf(" 输入串中有非法字符 \n");
        exit(1);
    }
    input[k]=ch;
    k++;
}while(ch!='#');
l=strlen(input);
ch=input[0];
printf(" 步数 \t 分析栈 \t 输入串 \t 所用规则 \n");
while ( 1 )
{
    printf("\n(%d)\t",j++); /* 输出当前执行步数 */
    print_stack();/* 输出当前栈的内容 (出栈前) */
    printf("\t");
    print_input();/* 输出剩余输出串 */
    printf("\t");
    if(flag2==1)
    {
        printf("%c->%s",cha.left,cha.right);/* 输出产生式 */
        flag2=0;
    }
    printf("\t");
    // 出栈
    X = stack[top--] ;
    if (X=='#')/* 如果 X 是结束符 */
    {
        if (X==ch)
        {
            printf("\tAcc\n");

```



```

    }
    else printf("\tERROR\n");
        break;
}
else if (isT(X)) /* 如果 X 是终结符 */
{
    ch=input[flag++];
}
else if (isNT(X)) /* 如果 X 是非终结符 */
{
    cha=M[non_termin_row(X)][termin_col(ch)];
    flag2=1;
    for(i=(cha.length-1);i>=0;i--) /* 产生式逆序入栈 */
        stack[++top]=cha.right[i];
    if(stack[top]=='~') /* 为空则不进栈 */
        top--;
}
else
{
    printf("Error in main()>%c\n",X);
    exit(0);
}
}
}

```

对文法 (4.1), 输入 “i\*i+i#” 运行上述程序, 其运算结果如图 4-5 所示。



图 4-5 针对文法 (4.1) 输入 “i\*i+i#” 预测分析程序运行结果

## 4.2 自下而上的语法分析器的设计与实现

所谓自下而上的语法分析就是从左向右扫描输入串, 逐步进行“归约”, 直到文法的开始符号, 或者从树末端开始, 自下而上为输入串构造语法分析树。这里的归约是指根据文



法的产生式规则，把产生式的右部替换成左部符号。

LR 分析法是一种有效的自下而上的语法分析方法，其中，L 表示从左到右扫描输入符号，R 表示为输入串构造一个最右推导的逆过程，实现 LR 分析法的程序称为 LR 分析程序，又称为 LR 分析器。LR 分析法比 LL(1) 分析法对文法的限制都要少得多。对大多数用无二义的上下文无关文法所描述的语言都可以用 LR 分析器予以识别，而且速度快，并能准确、及时地指出输入串的任何语法错误及出错位置。接下来将以 LR 分析器的设计与实现为例来说明如何进行自下而上的语法分析器设计与实现。

LR 分析方法的基本思想是在归约过程中，一方面记住已移进和归约出的整个符号串，即记住“历史”；另一方面根据所用的产生式推测未来可能遇到的输入符号，即对未来进行“展望”；当一串貌似句柄的符号串呈现于分析栈的顶端时，根据所记载的“历史”和“展望”，以及“现实”的输入符号三方面的材料，来确定栈顶的符号是否构成相对某一产生式的句柄，从而确定应该采用的分析动作。

LR 分析程序作为一种特殊的移进-归约程序，其由五部分组成，也即：一个输入缓冲区、一个输出缓冲区、一个符号栈、一张 LR 分析表、一个 LR 分析程序的总控程序，如图 4-6 所示。

1) 输入缓冲区用来存放待分析的符号串，它以界符‘#’作为结束标志。

2) 输出缓冲区用来存放 LR 分析总控程序对输入串进行分析过程中所采用的动作序列。

3) 分析栈的每一项都分为两栏，分别包括状态  $s$  和文法符号  $X$  两部分。栈里的每个状态概括了从分析开始直到某一归约阶段的全部“历史”和“展望”资料。 $(s_0, \#)$  为分析开始前预先放入栈里的初始状态  $s_0$  和句子括号 ( $\#$ )；任何时候，栈顶的状态  $s_m$  都代表了整个“历史”和已推出的“展望”，栈中自下而上所包含的符号串  $X_1X_2\cdots X_m$  是至今已移进归约出的文法符号串。

4) 一张 LR 分析表包括两部分：一部分是“动作” (ACTION) 表，另一部分是“状态转换” (GOTO) 表；它们都是二维数组。ACTION[ $s, a$ ] 规定了当状态  $s$  面临输入符号  $a$  时应采取什么动作，而 GOTO[ $s, X$ ] 规定了状态  $s$  面对文法符号  $X$  (终结符或非终结符) 时的下一状态是什么。每一项 ACTION[ $s, a$ ] 所规定的动作是以下四种情况之一：

① 移进：使  $(s, a)$  的下一状态  $s' = \text{GOTO}[s, a]$  和输入符号  $a$  进栈，下一输入符号变成现行输入符号。

② 归约：指用某一产生式  $A \rightarrow \beta$  进行归约。假若  $\beta$  的长度为  $\gamma$ ，则归约的动作是去掉栈顶的  $\gamma$  个项，即使状态  $s_{m-\gamma}$  变成栈顶状态，然后使  $(s_{m-\gamma}, A)$  的下一状态  $s' = \text{GOTO}[s_{m-\gamma}, A]$  和文法符号 (非终结符)  $A$  进栈。归约的动作不改变现行输入符号，执行归约的动

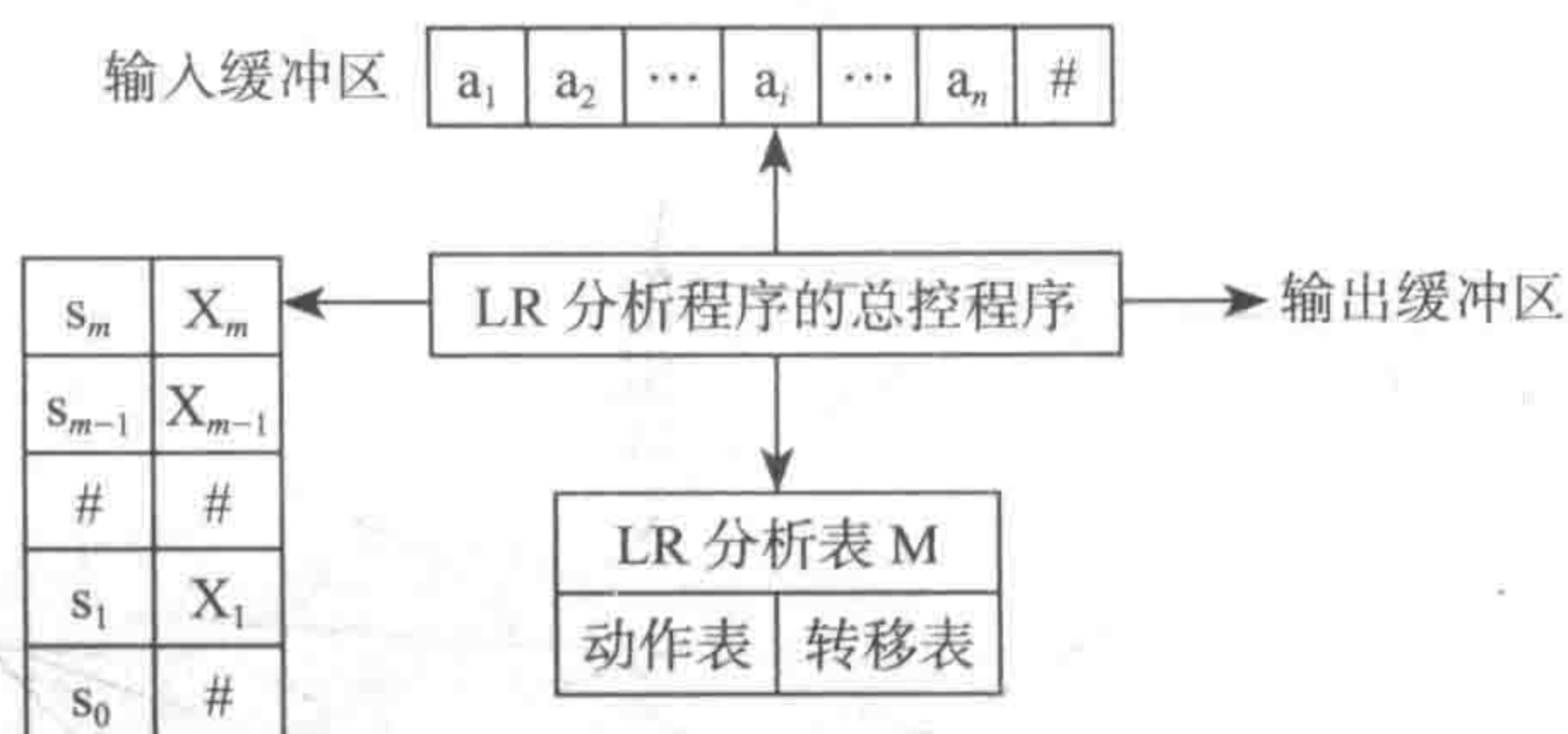


图 4-6 LR 分析器模型



作意味着呈现于栈顶的符号串  $X_{m-\gamma+1} \cdots X_m$  是一个相对于  $A$  的句柄。

③ 接受：宣布分析成功，停止分析器的工作。

④ 报错：报告发现源程序含有错误，调用出错处理程序。

5) LR 分析器的总控程序本身的工作十分简单，它的任何一步只需按分析栈的栈顶状态  $s$  和现行输入符号  $a$  执行  $\text{ACTION}[s, a]$  所规定的动作即可。

一个 LR 分析器的工作过程可看成是栈里的状态序列、已归约串和输入串所构成的三元组的变化过程。分析开始时初始三元组为

$$(s_0, \#, a_1 a_2 \cdots a_n \#)$$

其中， $s_0$  为分析器的初始状态；第一个 ‘#’ 为句子的左括号； $a_1 a_2 \cdots a_n$  为输入串，其后的 ‘#’ 为结束符（句子的右括号）。以后每步的结果可以表示为：

$$(s_0 s_1 \cdots s_m, \# X_1 \cdots X_m, a_i a_{i+1} \cdots a_n \#)$$

分析器根据  $\text{ACTION}(s_m, a_i)$  确定下一步动作：

1) 若  $\text{ACTION}(s_m, a_i)$  为移进，且  $s = \text{GOTO}(s_m, a_i)$ ，则三元组变为：

$$(s_0 s_1 \cdots s_m s, \# X_1 \cdots X_m a_i, a_{i+1} \cdots a_n \#)$$

2) 若  $\text{ACTION}(s_m, a_i)$  为按  $A \rightarrow \beta$  归约，三元组变为：

$$(s_0 s_1 \cdots s_{m-r} s, \# X_1 \cdots X_{m-r} A, a_i a_{i+1} \cdots a_n \#)$$

此处， $s = \text{GOTO}(s_{m-r}, A)$ ， $r$  为  $\beta$  的长度， $\beta = X_{m-r+1} \cdots X_m$ 。

3) 若  $\text{ACTION}(s_m, a_i)$  为“接受”，则三元组不再变化，变化过程终止，宣布分析成功。

4) 若  $\text{ACTION}(s_m, a_i)$  为“报错”，则三元组变化过程终止，报告错误。

一个 LR 分析器的工作过程就是一步一步地变化三元组的过程，直到执行接受或报错动作为止。上面讨论的分析思想可用算法 4.6 来描述。

#### 算法 4.6 LR 分析算法

输入：文法  $G$  的 LR 分析表和输入串  $w$

输出：如果  $w \in L(G)$ ，则输出  $w$  的自下而上分析，否则报错

步骤：

1. 将 # 和初始状态  $s_0$  压入栈，将  $w\#$  放入输入缓冲区；
2. 令输入指针  $ip$  指向  $w\#$  的第一个符号；
3. 令  $s$  是栈顶状态， $a$  是  $ip$  所指向的符号；
4. repeat
5. if  $\text{action}[s, a] = sj$  then /\*  $sj$  表示将下一个状态  $j$  和现行的输入符号  $a$  移进栈 \*/
6. { 把状态  $j$  和符号  $a$  分别压入栈；
7. 令  $ip$  指向下一输入符号； }
8. else if  $\text{action}[s, a] = rj$  then /\*  $rj$  表示按第  $j$  个产生式  $A \rightarrow \beta$  归约 \*/
9. { 从栈顶弹出  $|\beta|$  个符号；

- 10. 令  $s'$  是现在的栈顶状态;
- 11. 把  $\text{goto}[s', A]$  和  $A$  先后压入栈中;
- 12. 输出产生式  $A \rightarrow \beta$ ; }
- 13. else if  $\text{action}[s, a] = \text{acc}$  then
- 14. return;
- 15. else
- 16. error();

例 4.5 考虑文法 (4.2), 其 LR 分析表如表 4-3 所示。表中所引用的记号的意义是: acc 表示接受, 空白符表示出错。另外, 若  $a$  为终结符, 则  $\text{GOTO}[s, a]$  的值已列在  $\text{ACTION}[s, a]$  的  $s_j$  之中 (即状态  $j$ ), 因此 GOTO 表仅对所有非终结符 (比如  $A$ ) 列出  $\text{GOTO}[s, A]$  的值。假定输入串为 “ $i_1 * i_2 + i_3$ ”, 则 LR 分析器的工作过程如表 4-4 所示。

- (0)  $S' \rightarrow E$
- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow i$  (4.2)

表 4-3 文法 (4.2) 的 LR 分析表

ACTION										GOTO									
状态	i	+	*	(	)	#	E	T	F	状态	i	+	*	(	)	#	E	T	F
0	s5			s4			1	2	3	6	s5			s4				9	3
1		s6				acc				7	s5			s4					10
2		r2	s7		r2	r2				8		s6			s11				
3		r4	r4		r4	r4				9		r1	s7		r1	r1			
4	s5			s4			8	2	3	10		r3	r3		r3	r3			
5		r6	r6		r6	r6				11		r5	r5		r5	r5			

表 4-4  $i_1 * i_2 + i_3$  的 LR 分析过程

步骤	状态	符号	输入串	动作说明
1	0	#	$i_1 * i_2 + i_3 \#$	s5: 状态 5 和 $i_1$ 入栈
2	05	# $i_1$	$* i_2 + i_3 \#$	r6: 用 $F \rightarrow i$ 归约且 $\text{GOTO}(0, F)=3$ 入栈
3	03	# $F$	$* i_2 + i_3 \#$	r4: 用 $T \rightarrow F$ 归约且 $\text{GOTO}(0, T)=2$ 入栈
4	02	# $T$	$* i_2 + i_3 \#$	s7: 状态 7 和 $*$ 入栈
5	027	# $T*$	$i_2 + i_3 \#$	s5: 状态 5 和 $i_1$ 入栈
6	0275	# $T* i_2$	$+ i_3 \#$	r6: 用 $F \rightarrow i$ 归约且 $\text{GOTO}(7, F)=10$ 入栈
7	02710	# $T*F$	$+ i_3 \#$	r3: 用 $T \rightarrow T*F$ 归约且 $\text{GOTO}(0, T)=2$ 入栈
8	02	# $T$	$+ i_3 \#$	r2: 用 $E \rightarrow T$ 归约且 $\text{GOTO}(0, E)=1$ 入栈



(续)

步骤	状态	符号	输入串	动作说明
9	01	# E	+i <sub>3</sub> #	s6: 状态 6 和 + 入栈
10	016	# E+	i <sub>3</sub> #	s5: 状态 5 和 i <sub>3</sub> 入栈
11	0165	#E+i <sub>3</sub>	#	r6: 用 F → i 归约且 GOTO(6, F)=3 入栈
12	0163	# E+F	#	r4: 用 T → F 归约且 GOTO(6, T)=9 入栈
13	0169	# E+T	#	r1: 用 E → E+T 归约且 GOTO(0, E)=1 入栈
14	01	# E	#	acc: 分析成功

算法 4.6 的具体实现代码如下:

```
#include<stdio.h>
#include<string.h>
#define VtNum 6/* 终结符的数目 */
#define VnNum 3/* 非终结符的数目 */
#define PNum 7/* 产生式的数目 */
#define StateNum 12/* 状态的数目 */
/*ACTION TABLE*/
char*action[StateNum][VtNum]={
"s5#",NULL,NULL,"s4#",NULL,NULL,
NULL,"s6#",NULL,NULL,NULL,"acc",
NULL,"r2#","s7#",NULL,"r2#","r2#",
NULL,"r4#","r4#",NULL,"r4#","r4#",
"s5#",NULL,NULL,"s4#",NULL,NULL,
NULL,"r6#","r6#",NULL,"r6#","r6#",
"s5#",NULL,NULL,"s4#",NULL,NULL,
"s5#",NULL,NULL,"s4#",NULL,NULL,
NULL,"s6#",NULL,NULL,"s11#",NULL,
NULL,"r1#","s7#",NULL,"r1#","r1#",
NULL,"r3#","r3#",NULL,"r3#","r3#",
NULL,"r5#","r5#",NULL,"r5#","r5#"};
/*GOTO TABLE*/
int gototable[StateNum][VnNum]={
1,2,3,
0,0,0,
0,0,0,
0,0,0,
8,2,3,
0,0,0,
0,9,3,
0,0,10,
0,0,0,
0,0,0,
0,0,0,
0,0,0};
char termin[VtNum]={'i','+','*','(',')','#'};/* 存放非终结符 */
char non_termin[VnNum]={'E','T','F'};/* 存放终结符 */
char *LR[PNum]={"S->E#","E->E+T#","E->T#","T->T*F#","T->F#","F->(E)#","F->i#"};/*
存放产生式 */
```

```

int stack[StateNum]; /* 状态栈 */
char symbol[StateNum]; /* 符号栈 */
char input[StateNum]; /* 输入栈 */
void main()
{
    int m,n,g;
    int i,p,y;
    int l; /* 待归约产生式右部的长度 */
    int count; /* 步骤的编号 */
    int h; /* 产生式的编号 */
    int z; /* 状态栈栈顶状态 */
    int j; /* 非终结符数组下标 */
    int k; /* 终结符数组下标 */
    char x; /* 当前正在处理的输入符号 */
    char c;
    int top_stack=0; /* 状态栈栈顶 */
    int top_symbol=0; /* 符号栈栈顶 */
    int top_input=0; /* 输入栈栈顶 */
    int top=0; /* 目前输入串的位置 */
    stack[0]=0; /* 分析开始前把初始状态 0 加入状态栈 */
    y=stack[0];
    symbol[0]='#'; /* 分析开始前把 # 加入符号栈 */
    count=0;
    z=0;
    printf(" 本程序只能对由 'i', '+', '*', '(', ')' 构成的以 '#' 结束的字符串进行分析。\\n");
    printf(" 请输入要分析的字符串 :");
    do{
        scanf("%c",&c);
        input[top_input]=c;
        top_input=top_input+1;
    } while(c!='#');
    printf(" 步骤 \\t 状态栈 \\t \\t 符号栈 \\t \\t 输入串 \\t \\t ACTION \\t GOTO \\n");
    do{
        y=z; /* y,z 指向状态栈栈顶 */
        x=input[top];
        count++;
        printf("%d\\t",count); /* 输出步骤 */
        m=0;
        while(m<=top_stack) /* 输出状态栈 */
        {
            printf("%d",stack[m]);
            m=m+1;
        }
        printf("\\t\\t");
        n=0;
        while(n<=top_symbol) /* 输出符号栈 */
        {
            printf("%c",symbol[n]);
            n=n+1;
        }
        printf("\\t\\t");
        g=top;
        while(g<=top_input) /* 输出输入串 */
        {

```



```

        printf("%c",input[g]);
        g=g+1;
    }
    printf("\t\t");
    j=0;
    while(x!=termin[j]&&j<=VtNum)j++;
    if(j==VtNum&&x!=termin[j]) /* 输入字符串不是由终结符组成, 报错 */
    {
        printf(" 输入字符串不是由终结符组成 \n");
        return;
    }
    if(action[y][j]==NULL) /* 出现语法错误, 报错 */
    {
        printf("error\n");
        return;
    }
    else
    if(action[y][j][0]=='s') /* 处理移进 */
    {
        if(action[y][j][2]!='#')
            z=(action[y][j][1]-'0')*10+action[y][j][2]-'0';
        else
            z=action[y][j][1]-'0';
        top_stack=top_stack+1;
        top_symbol=top_symbol+1;
        stack[top_stack]=z;
        symbol[top_symbol]=x;
        top=top+1;
        i=0;
        while(action[y][j][i]!='#')
        {
            printf("%c",action[y][j][i]);
            i++;
        }
        printf("\n");
    }
    if(action[y][j][0]=='r') /* 处理归约 */
    {
        i=0;
        while(action[y][j][i]!='#')
        {
            printf("%c",action[y][j][i]);
            i++;
        }
        h=action[y][j][1]-'0';
        k=0;
        while(LR[h][0]!=non_termin[k])k++;
        l=strlen(LR[h])-4;
        top_stack=top_stack-l+1;
        top_symbol=top_symbol-l+1;
        y=stack[top_stack-1];
        p=gototable[y][k];
        stack[top_stack]=p;
    }

```

```

        symbol[top_symbol]=LR[h][0];
        z=p;
        printf("\t");
        printf("%d\n",p);
    }
} while(action[y][j]!="acc");
    printf("acc\n");
}

```

针对文法 (4.2), 输入 “i\*i+i#” 运行上述程序, 其运算结果如图 4-7 所示。

不同的 LR 分析器, 其总控程序都一样, 不同的是其 LR 分析表, 构造 LR 分析表的方法不同就形成各种不同的 LR 分析法。常见分析表的构造方法有 4 种, 它们是:

1) LR(0) 分析表构造法: 这种方法局限性很大, 但它是建立一般 LR 分析表的基础。

2) SLR(1) 分析表 (即简单 LR 表) 构造法: 这种方法较易实现又极有使用价值。

3) LR(1) 分析表 (即规范 LR 分析表) 构造法: 这种方法适用于大多数上下文无关文法, 但分析表体积庞大。

4) LALR(1) 分析表 (即向前 LR 分析表) 构造法: 该方法能力介于 SLR(1) 分析表和 LR(1) 分析表之间。

表 4-3 所给出的分析表即 SLR(1) 分析表, 接下来我们以 SLR(1) 分析表构造为例来说明如何构造 LR 分析表。为了构造 SLR(1) 分析表, 要用到下述几个重要概念和函数。

**定义 4.3 (活前缀)** 活前缀是指规范句型的一个前缀, 这种前缀不含句柄之后的任何符号。即对于规范句型  $\alpha\beta\delta$ ,  $\beta$  为句柄, 如果  $\alpha\beta=u_1u_2\cdots u_r$ , 则符号串  $\varepsilon$ 、 $u_1u_2\cdots u_i$  ( $1 \leq i \leq r$ ) 是  $\alpha\beta\delta$  的活前缀 ( $\delta$  必为终结字符串)。

之所以称为活前缀, 是因为在其右边增添一些终结符号之后, 就可以使它成为一个规范句型。在使用 LR 分析法进行分析的过程中, 只要已分析的符号串是正确的, 符号栈里的文法符号由底到顶就构成规范句型的一个活前缀, 当这个活前缀包含句柄时就归约, 否则移进。因此, 只要能识别出所有活前缀, 并识别活前缀是否包含句柄, 就能决定什么时候移进, 什么时候归约。下面的问题就是, 给定一个文法, 如何识别该文法的所有活前缀以及活前缀与句柄之间的关系。为此, 引入 LR(0) 项目的概念。

**定义 4.4 (LR(0) 项目)** 给定一个文法  $G(S)$ , 右部某个位置上标有圆点的产生式称为该文法的一个 LR(0) 项目。其中, 圆点在产生式最右端的 LR(0) 项目称为归约项目; 对文法

本程序只能对由 'i', '+', '\*', '<', '>', '(', ')', '#' 构成的以 '#' 结束的字符串进行分析。

请输入要分析的字符串: i\*i+i#

步骤	状态栈	符号栈	输入串	ACTION	GOTO
1	0	#	i*i+i#	s5	
2	05	#i	*i+i#	r6	3
3	03	#P	*i+i#	r4	2
4	02	#T	*i+i#	s7	
5	027	#T*	i+i#	s5	
6	0275	#T*i	+i#	r6	10
7	02710	#T*iF	+i#	r3	2
8	02	#T	+i#	r2	1
9	01	#E	+i#	s6	
10	016	#E+	i#	s5	
11	0165	#E+i	#	r6	3
12	0163	#E+iF	#	r4	9
13	0169	#E+iT	#	r1	1
14	01	#E	#	acc	

Press any key to continue.

图 4-7 针对文法 (4.2) 输入 “i\*i+i#” LR 分析程序运行结果



开始符号的归约项目又称接受项目；圆点后第一个符号为非终结符号的 LR(0) 项目称为待约项目；圆点后第一个符号为终结符号的 LR(0) 项目称为移进项目。



**注意** 产生式  $A \rightarrow \alpha$  只有一个 LR(0) 归约项目  $A \rightarrow \cdot$ 。LR(0) 项目中的圆点符分割已获取的内容和待获取的内容，点的左边代表历史信息，右边代表展望信息。直观地讲，LR(0) 项目表示在分析过程的某一阶段，已经看到了产生式的多大部分以及希望看到的部分。

为了保证文法开始符号只出现在一个产生式的左边，亦即保证分析器只有一个接受状态，对该文法进行拓广，构造其拓广文法，便会有一个仅含项目  $S' \rightarrow S \cdot$  的状态，这就是唯一的“接受”态。

**定义 4.5 (拓广文法)** 假定文法  $G$  是一个以  $S$  为开始符号的文法，构造一个  $G'$ ，它包含了整个  $G$ ，但它引进了一个不出现在  $G$  中的非终结符  $S'$ ，并加入一个新产生式  $S' \rightarrow S$ ，而这个  $S'$  是  $G'$  的开始符号。那么，称  $G'$  是  $G$  的拓广文法。

为了识别活前缀，还需要定义 LR(0) 项目集的闭包 (CLOSURE) 和状态转换函数  $GO$ 。

**定义 4.6 (LR(0) 项目集的闭包 (CLOSURE))** 假定  $I$  是文法  $G$  的任一 LR(0) 项目集合，定义  $I$  的闭包  $CLOSURE(I)$  如下：

- 1)  $I$  的任何项目都属于  $CLOSURE(I)$ 。
- 2) 若  $A \rightarrow \alpha \cdot B \beta$  属于  $CLOSURE(I)$ ，那么，对任何关于  $B$  的产生式  $B \rightarrow \gamma$ ，项目  $B \rightarrow \cdot \gamma$  也属于  $CLOSURE(I)$ 。
- 3) 重复执行上述两步骤直至  $CLOSURE(I)$  不再增大为止。

直观地说， $CLOSURE(I)$  中的项目  $A \rightarrow \alpha \cdot B \beta$  是指：在分析过程的某一时刻，希望看到可从  $B \beta$  推出的符号串；若  $B \rightarrow \gamma$  是一个产生式，那么在同一时刻，我们当然也希望看到可从  $\gamma$  推出的符号串。因此，也把  $B \rightarrow \cdot \gamma$  加到  $CLOSURE(I)$ 。

**定义 4.7 (LR(0) 项目集的状态转换函数  $GO$ )** 假定  $I$  是文法  $G$  的一个 LR(0) 项目集， $X$  是文法  $G$  的一个文法符号，函数值  $GO(I, X)$  定义为：

$$GO(I, X) = CLOSURE(J)$$

其中  $J = \{ \text{任何形如 } A \rightarrow \alpha X \cdot \beta \text{ 的项目} \mid A \rightarrow \alpha \cdot X \beta \text{ 属于 } I \}$ 。 $GO(I, X)$  称为转换函数，项目  $A \rightarrow \alpha X \cdot \beta$  称为  $A \rightarrow \alpha \cdot X \beta$  的后继。

通过  $CLOSURE$  和  $GO$  函数很容易构造文法  $G$  的拓广文法  $G'$  的 LR(0) 项目集规范族，即在  $CLOSURE$  和  $GO$  函数作用下所得到的文法  $G'$  的 LR(0) 项目集的全体，具体见算法 4.7。对于算法 4.7 的实现，在这里不再详细介绍其具体实现过程，只通过一个例子来说明如何利用该算法构造文法的 LR(0) 项目集规范族。

#### 算法 4.7 构造文法的 LR(0) 项目集规范族

输入：文法  $G = (V_T, V_N, P, S)$  的拓广文法  $G'$

输出： $G'$  的 LR(0) 项目集规范族  $C$



步骤:

1.  $C := \{ \text{CLOSURE}(\{S' \rightarrow \cdot S\}) \};$
2. repeat
3. for  $C$  中每个项目集  $I$  和  $G'$  的每个符号  $X$
4. if  $\text{GO}(I, X)$  非空且不属于  $C$  then
5. 把  $\text{GO}(I, X)$  放入  $C$  族中;
6. until  $C$  不再增大

例 4.6 对文法 (4.2), 利用算法 4.7 计算其 LR (0) 项目集规范族。

解:  $I_0 = \text{CLOSURE}(\{E' \rightarrow \cdot E\})$

$$= \{E' \rightarrow \cdot E, E \rightarrow \cdot E + T, E \rightarrow \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot i\}$$

$$I_1 = \text{GO}(I_0, E) = \text{CLOSURE}(\{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}) = \{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}$$

$$I_2 = \text{GO}(I_0, T) = \text{CLOSURE}(\{E \rightarrow T \cdot, T \rightarrow T \cdot * F\}) = \{E \rightarrow T \cdot, T \rightarrow T \cdot * F\}$$

$$I_3 = \text{GO}(I_0, F) = \text{CLOSURE}(\{T \rightarrow F \cdot\}) = \{T \rightarrow F \cdot\}$$

$$I_4 = \text{GO}(I_0, () = \text{CLOSURE}(\{F \rightarrow (\cdot E)\})$$

$$= \{F \rightarrow (\cdot E), E \rightarrow \cdot E + T, E \rightarrow \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot i\}$$

$$I_5 = \text{GO}(I_0, i) = \text{CLOSURE}(\{F \rightarrow i \cdot\}) = \{F \rightarrow i \cdot\}$$

$$I_6 = \text{GO}(I_1, +) = \text{CLOSURE}(\{E \rightarrow E + \cdot T\})$$

$$= \{E \rightarrow E + \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot i\}$$

$$I_7 = \text{GO}(I_2, *) = \text{CLOSURE}(\{T \rightarrow T * \cdot F\}) = \{T \rightarrow T * \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot i\}$$

$$I_8 = \text{GO}(I_4, E) = \text{CLOSURE}(\{F \rightarrow (E \cdot), E \rightarrow E \cdot + T\}) = \{F \rightarrow (E \cdot), E \rightarrow E \cdot + T\}$$

$$\text{GO}(I_4, T) = \text{CLOSURE}(\{E \rightarrow T \cdot, T \rightarrow T \cdot * F\}) = I_2$$

$$\text{GO}(I_4, F) = \text{CLOSURE}(\{T \rightarrow F \cdot\}) = I_3$$

$$\text{GO}(I_4, () = \text{CLOSURE}(\{F \rightarrow (\cdot E)\}) = I_4$$

$$\text{GO}(I_4, i) = \text{CLOSURE}(\{F \rightarrow i \cdot\}) = I_5$$

$$I_9 = \text{GO}(I_6, T) = \text{CLOSURE}(\{E \rightarrow E + T \cdot, T \rightarrow T \cdot * F\}) = \{E \rightarrow E + T \cdot, T \rightarrow T \cdot * F\}$$

$$\text{GO}(I_6, F) = \text{CLOSURE}(\{T \rightarrow F \cdot\}) = I_3$$

$$\text{GO}(I_6, () = \text{CLOSURE}(\{F \rightarrow (\cdot E)\}) = I_4$$

$$\text{GO}(I_6, i) = \text{CLOSURE}(\{F \rightarrow i \cdot\}) = I_5$$

$$I_{10} = \text{GO}(I_7, F) = \text{CLOSURE}(\{T \rightarrow T * F \cdot\}) = \{T \rightarrow T * F \cdot\}$$

$$\text{GO}(I_7, () = \text{CLOSURE}(\{F \rightarrow (\cdot E)\}) = I_4$$

$$\text{GO}(I_7, i) = \text{CLOSURE}(\{F \rightarrow i \cdot\}) = I_5$$

$$I_{11} = \text{GO}(I_8, )) = \text{CLOSURE}(\{F \rightarrow (E) \cdot\}) = \{F \rightarrow (E) \cdot\}$$

$$\text{GO}(I_8, +) = \text{CLOSURE}(\{E \rightarrow E + \cdot T\}) = I_6$$

$$\text{GO}(I_9, *) = \text{CLOSURE}(\{T \rightarrow T * \cdot F\}) = I_7$$



至此,已经构造出文法(4.2)的项目集规范族为  $\{I_0, I_1, \dots, I_{11}\}$ 。

假定 LR(0) 项目集规范族的一个项目集  $I$  中含有  $m$  个移进项目:

$$A_1 \rightarrow \alpha \cdot a_1 \beta_1, A_2 \rightarrow \alpha \cdot a_2 \beta_2, \dots, A_m \rightarrow \alpha \cdot a_m \beta_m$$

同时含有  $n$  个归约项目:

$$B_1 \rightarrow \alpha \cdot, B_2 \rightarrow \alpha \cdot, \dots, B_n \rightarrow \alpha \cdot$$

如果集合  $\{a_1, \dots, a_m\}$ 、 $\text{FOLLOW}(B_1)$ 、 $\dots$ 、 $\text{FOLLOW}(B_n)$  两两不相交(包括不得有两个 FOLLOW 集含有 '#'),则隐含在  $I$  中的动作冲突可通过检查现行输入符号  $a$  属于上述  $n+1$  个集合中的哪个集合而获得解决,也即

- 1) 若  $a$  是某个  $a_i$ ,  $i=1, 2, \dots, m$ , 则移进。
- 2) 若  $a \in \text{FOLLOW}(B_i)$ ,  $i=1, 2, \dots, n$ , 则用产生式  $B_i \rightarrow \alpha$  进行归约。
- 3) 此外,报错。

冲突性动作的这种解决办法叫作 SLR(1) 分析法。

基于 SLR(1) 解决办法的思想,对任给的一个文法  $G(S)$ ,可用算法 4.8 构造其 SLR(1) 分析表。算法 4.8 中所涉及的计算非终结符 FOLLOW 集的实现在 4.1 节中已经详细介绍过,我们将算法中剩余部分的实现留作作业,在此仅通过一个例子来说明如何利用该算法构造文法的 SLR(1) 分析表。

#### 算法 4.8 构造 SLR(1) 分析表

输入: 文法  $G=(V_T, V_N, P, S)$  的拓广文法  $G'$

输出: 文法  $G'$  的 SLR(1) 分析表

步骤:

1. 构造  $G'$  的 LR(0) 项目集规范族  $C=\{I_0, I_1, \dots, I_n\}$  和识别活前缀自动机的状态转换函数  $GO$ , 令每个项目集  $I_k$  的下标  $k$  作为分析器的状态, 包含项目  $S' \rightarrow \cdot S$  的集合  $I_k$  的下标  $k$  为分析器的初态。

2. ACTION 子表的构造:

(1) 若项目  $A \rightarrow \alpha \cdot a \beta \in I_k$  且  $GO(I_k, a)=I_j$ ,  $a$  为终结符, 则置  $\text{ACTION}[k, a]$  为  $sj$ , 表示将  $(j, a)$  移进栈。

(2) 若项目  $A \rightarrow \alpha \cdot \in I_k$ , 则对任何终结符  $a$ ,  $a \in \text{FOLLOW}(A)$ , 置  $\text{ACTION}[k, a]$  为  $rj$ , 其中, 假定  $A \rightarrow \alpha$  为文法  $G'$  的第  $j$  个产生式。

(3) 若项目  $S' \rightarrow S \cdot \in I_k$ , 则置  $\text{ACTION}[k, \#]$  为  $\text{acc}$ , 表示分析成功。

3. GOTO 子表的构造:

若  $GO(I_k, A)=I_j$ ,  $A$  为非终结符, 则置  $\text{GOTO}[k, A]=j$ 。

4. 分析表中凡不能用规则 2~3 填入的空白格均置为“出错标志”。

例 4.7 构造文法(4.2)的 SLR(1) 分析表。

解: 依次考查例 4.6 中计算出来的 12 个项目集。



考查  $I_0$ :

因为  $F \rightarrow \cdot (E)$   $F \rightarrow \cdot i$  属于  $I_0$ , 并且  $GO(I_0, () = I_4$ ,  $GO(I_0, i) = I_5$ 。所以有  $ACTION[0, id] = s5$ ,  $ACTION[0, (] = s4$ 。  $I_0$  中其余的项目不生成动作,  $GO[0, E] = 1$ ,  $GO[0, T] = 2$ ,  $GO[0, F] = 3$ 。

考查  $I_1$ :

因为  $E' \rightarrow E \cdot$  为接受项目, 而  $E \rightarrow E \cdot + T$  是移进项目, 故这两个项目存在移进-接受冲突, 又因  $FOLLOW(E') = \{ \# \}$ ,  $+ \notin FOLLOW(E')$ , 故此冲突可以解决, 根据算法 4.8 有  $ACTION[1, \#] = acc$ ,  $ACTION[1, +] = s6$ 。

考查  $I_2$ :

项目  $E \rightarrow T \cdot$  与  $T \rightarrow T \cdot * F$  之间同样存在移进-归约冲突, 因为  $FOLLOW(E) = \{ \#, ), + \}$ , 而  $* \notin FOLLOW(E)$ , 所以根据算法 4.8 第一个项目使得  $ACTION[2, \#] = r2$ ,  $ACTION[2, )] = r2$ ,  $ACTION[2, +] = r2$ ; 第二个项目使得  $ACTION[2, *] = s7$ 。从而  $I_2$  的冲突可以解决。

考查  $I_3$ :

项目  $T \rightarrow F \cdot$  是归约项目, 因为  $FOLLOW(T) = \{ *, +, ), \# \}$ , 所以有  $ACTION[3, *] = ACTION[3, +] = ACTION[3, )] = ACTION[3, \#] = r4$ 。

依次考查该文法 LR(0) 项目集规范族中的每一个状态集, 根据算法 4.8 可以得到文法 (4.2) 的 SLR 分析表, 如表 4-3 所示。

## 4.3 语法分析器的生成器

与 LEX 一样, YACC 也是贝尔实验室用 C 语言在 UNIX 操作系统上首先实现的语法分析程序自动生成工具, 目前借助于 YACC 已经实现了数百个编译器。

### 4.3.1 YACC 的源文件结构

YACC 输入用户提供的语言的语法描述规格说明, 也即 YACC 源文件, 其以 .y 为扩展名。然后, 基于 LALR(1) 语法分析的原理, 输出两个文件, 一个是包含有语法分析函数 `yyparse()` 的 C 程序, 扩展名为 .c; 另一个是包含源文件中所有终结符 (词法分析意义下的单词) 编码的宏定义文件, 扩展名为 .h。接下来, 这两个文件经过 C 编译器的编译就生成一个语法分析器, 该语法分析器的输入是源程序经过词法分析的结果, 输出可以是一棵语法树, 或者是所生成的目标代码, 也可以是关于输入串是否符合语法规则的信息, 具体的输出形式是可以在 YACC 源程序中自己定义的。其使用方法如图 4-8 所示。

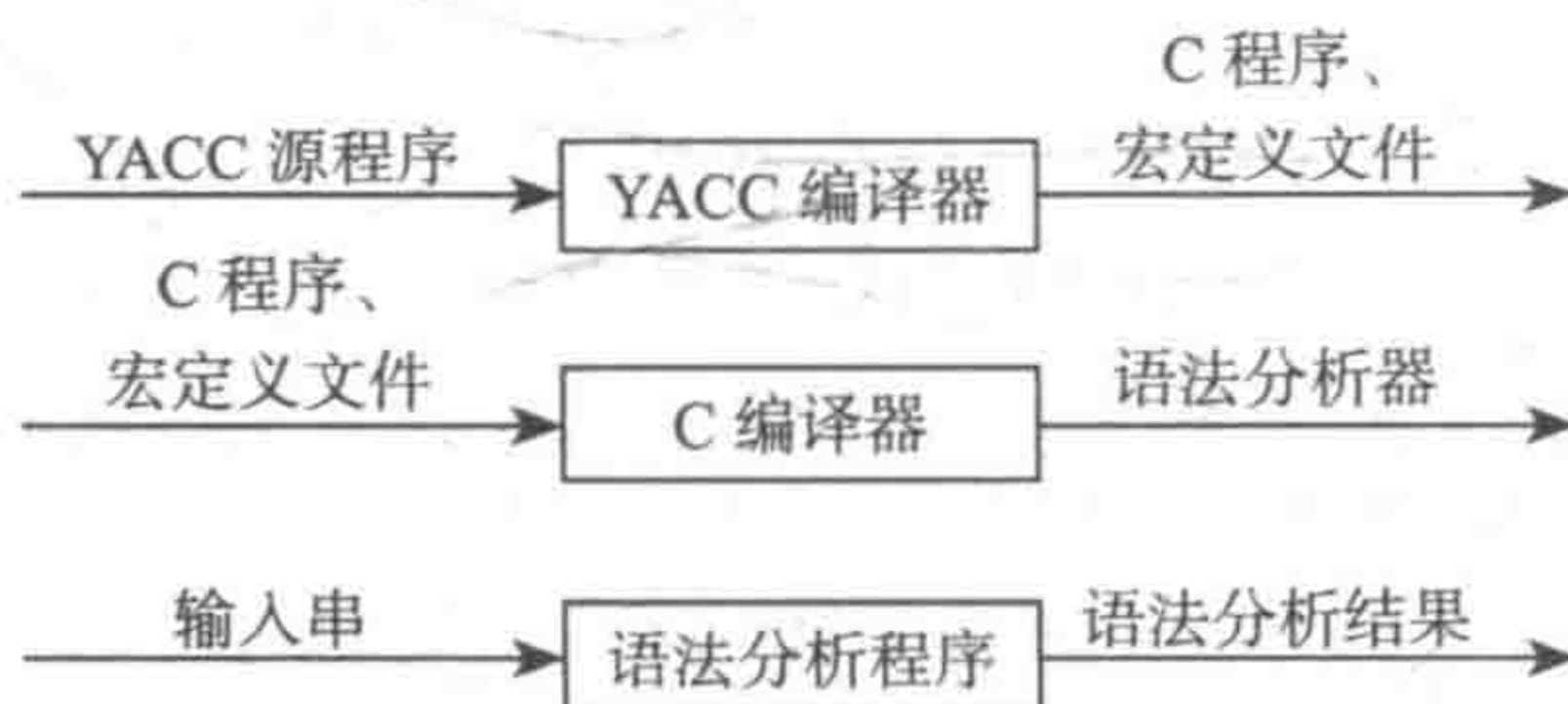


图 4-8 用 YACC 构建语法分析器

YACC 源程序又称 YACC 规格说明, 同



LEX 源程序类似，也由说明部分、翻译规则和辅助过程三部分组成，形式如下：

```
[ 说明部分 ]
%%
翻译规则
[%%
辅助过程 ]
```

其中，用方括号括起来的部分可以省略，但是翻译规则部分不能省略。下面我们通过一个例子来说明 YACC 源程序。

### 1. 说明部分

YACC 源程序说明部分有任选的两部分。

第一部分是处于“%{”和“%}”之间的 C 语言代码部分，其用来定义翻译规则和辅助过程中用到的全局变量和相关文件等，YACC 在分析源文件时，将此部分直接复制到输出的分析器中。

第二部分是文法记号的声明，该部分的每一项均以“%”开头。此部分用来定义文法符号、语义动作中使用的数据类型，以及翻译规则中运算符的优先级等。

#### (1) 语义值类型定义

利用 YACC 软件所产生的语法分析器除了包含用来保存文法符号和状态的分析栈之外，还有一个用来保存与分析栈中文法符号相对应的语义值的语义栈。语义值的类型定义就是确定语义栈中元素的数据结构。在 YACC 所产生的分析器源程序中，该数据类型为 YYSTYPE，是一个抽象的数据类型，缺省时为整型。但用户可自定义 YYSTYPE 的类型，方式有两种。

1) 如果语义值的数据类型为简单类型，则用户可直接在 C 语言代码部分用宏定义 YYSTYPE 为所需的数据类型。例如：

```
%{
#define YYSTYPE float
%}
```

将分析器中出现的每一个 YYSTYPE 都替换为 float，即语义值的数据类型为 float 类型。

2) 如果语义值的数据类型包含多种类型，也即，不同的语法符号有不同的语义值，YACC 利用 C 语言的共用体结构来实现对不同语法符号的语义值选择不同的数据类型。例如，如果作为终结符的标识符的语义值是指向符号表中该元素的指针，表达式的语义值是指向表达式二叉树结点的指针，则可以按如下方式定义一个共用体类型：

```
%union
{
    SYMBOL *sym; /* 符号表元素指针，终结符 id 使用该类型 */
    ENODE *node; /* 二叉树结点指针，非终结符 expr 使用该类型 */
}
```



YACC 编译源文件时把上述定义翻译为:

```
typedef union
{
    SYMBOL *sym; /* 符号表元素指针, 终结符 id 使用该类型 */
    ENODE *node; /* 二叉树结点指针, 非终结符 expr 使用该类型 */
}
YYSTYPE
```

即语法分析器还是以 YYSTYPE 作为语义值的数据类型, 但此时语义值是一个有多种类型可选的共用体结构。

引用时采用以下方式:

```
%token <sym> id
%type <node> expr
```

其中, 表示标识符的终结符 id 是指向符号表元素的指针类型, 表示表达式的非终结符 expr 是二叉树结点的指针类型, 尖括号中的名字为相应的终结符和非终结符的语义值类型, 终结符类型的定义要以 %token 开始, 非终结符类型的定义要以 %type 开始。

## (2) 文法开始符定义

一般以 %start S 的形式说明文法的开始符号为 S, 默认为规则部分中出现的第一个产生式的左部非终结符号是文法的开始符号。

## (3) 终结符定义

在 YACC 源程序语法规则部分出现的所有终结符 (单个字符本身作为终结符除外, 其在语法规则中直接加单引号使用) 必须用 %token 定义, 否则将被 YACC 当成非终结符处理。定义形式有两种:

1) 如果语义值的数据类型为简单类型, 或者是多类型但所定义的终结符的语义值不被任何语义动作所引用, 可用如下方式定义:

```
%token 终结符 1 终结符 2.....
```

如:

```
%token if else while /* 关键字的语义值不参与任何计算 */
```

2) 如果语义值的数据类型是多类型的, 并且所定义的终结符的语义值被某一个语法规则的语义动作引用, 则用如下方式定义:

```
%token <类型> 终结符 1 终结符 2.....
```

如前面提到的 %token <sym> id。

YACC 将对每个终结符进行唯一性编码, 编码的原则是: 单个字符作为终结符, 其编码值为对应的 ASCII 码 (其值不会大于 256); 自定义的终结符, 其编码值从 257 开始, 并用 C 语言宏定义的方式实现编码, 依次递增, 每次加 1。当词法分析程序从输入符号串中



识别出一个终结符时,将返回该终结符的编码。

#### (4) 非终结符定义

一般来说,YACC 中非终结符不必特别声明,只有当语义值是多类型的,并且该非终结符的语义值参与了动作部分的语义计算,才需要声明。其定义方式如下:

```
%type <类型> 非终结符 1 非终结符 2...非终结符 n
```

如前面所提到的 %type <node> expr。

#### (5) 优先级及结合性定义

YACC 允许用户规定运算符的优先级和结合性。其中,以 %left 开头的行表示所定义的运算符的结合性是左结合,以 %right 开头的行表示所定义的运算符的结合性是右结合,以 %nonassoc 开头的行表示所定义的运算符的结合性是无结合性。出现在同一结合性说明中各个运算符具有相同的优先级,而结合性说明排列的次序决定了运算符优先级的高低,排列在前面的结合性说明所定义的运算符的优先级较低,而排列在后面的结合性说明所定义的运算符的优先级较高。

### 2. 翻译规则

YACC 源程序翻译规则部分中的每条规则由一个产生式和有关的语义动作组成。形如  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  的产生式,在 YACC 说明文件中写成

```
A :  $\alpha_1$  { 语义动作 1 }
    |  $\alpha_2$  { 语义动作 2 }
    ...
    |  $\alpha_n$  { 语义动作 n }
    ;
```

在 YACC 产生式里用单引号括起来的单个字符是终结符号,没有用引号括起来、也没有被说明成 token 类型的字母数字串是非终结符号。产生式左部非终结符之后是一个冒号,右部候选式之间用竖线分隔。在规则的末尾用“;”表示规则的结束。YACC 语义动作是用户用 C 语言为产生式编写一段相关的语义处理程序,它们需用“{ }”括起来。在生成语法分析程序时,YACC 把语义动作嵌入到 LALR 分析表相应的位置。在 YACC 语义动作中用“\$\$”表示与产生式左部非终结符号相关的属性值,用“\$i”表示与产生式右部第  $i$  个文法符号相关的属性值。由于语义动作都是放在产生式右部的尾部,所以,每当用某一个产生式进行归约时,执行与之相关的语义动作。这样,可以在每个 \$i 值都计算出来之后,再求 \$\$。

### 3. 辅助过程

辅助过程部分是由 C 语言程序组成的,主要包括主程序 main()、词法分析程序 yylex()、出错处理程序 yyerror() 和语义动作中所调用的用户自定义函数等。其中必须包含词法分析程序 yylex(),其他函数则视需要而定。



### (1) 主程序

YACC 在处理完 YACC 源程序后, 所输出的 C 程序文件中包含语法分析函数 `yyparse`, 主程序 `main()` 的主要作用是调用函数 `yyparse()` 对源程序进行语法分析。当语法分析成功结束时, `yyparse()` 返回值为 0; 而在发现源程序有语法错误时返回值为 1, 同时调用 `yyerror()` 函数输出出错信息。

若用户要在调用函数 `yyparse` 之前或之后做一些其他处理, 也在 `main` 函数中完成。若用户只需要在 `main` 中调用 `yyparse`, 则可以不必要自己编写 `main`, 而直接使用 YACC 库提供的 `main`。库中的 `main` 定义格式如下:

```
main()
{
    return(yyparse());
}
```

### (2) 出错处理程序

语法分析函数 `yyparse()` 在对输入进行分析时, 如果出现某一状态和输入单词在分析表中找不到对应操作的情形, 就认为输入有误, 此时分析器将调用函数 `yyerror()` 报错; 如果没有任何恢复措施, 函数 `yyparse()` 将返回 1 结束执行。

YACC 库提供了出错处理 `yyerror` 函数, 定义格式如下:

```
int yyerror( char *s)
{
    fprintf(stderr, "%s\n", s);
}
```

即打印字符串 `s` 到标准错误输出。用户可以根据需要, 自己编写一个 `yyerror()` 函数, 比如在发生错误时, 报告出错的行号。

### (3) 词法分析程序

每次调用函数 `yylex()` 时, 得到一个二元式:  $\langle \text{记号}, \text{属性值} \rangle$ 。记号通过 `return` 语句回送给 `yyparse()`, 其必须事先在 YACC 说明文件的第一部分中用 `%token` 说明; 属性值通过全局变量 `yylval()` 回送给 `yyparse()`。因此, `yyparse()` 的输入来源于 `yylex()` 的返回信息。`yylex()` 可由用户手工编写或通过 LEX 工具自动生成。第 3 章介绍 LEX 时曾经指出, 由 LEX 自动生成的词法分析程序包含在一个 C 程序中, 其名字就是 `yylex`。

## 4.3.2 YACC 和 LEX 的接口

前面提到 LEX 编译器将提供词法分析程序 `yylex()`, 其可用于 YACC, `yylex` 就是 YACC 所需要的词法分析器的名字。为了使 `yyparse()` 能使用 LEX 所生成的 `yylex()` 函数, 最简便的方法是在 YACC 源文件的程序部分写上一条形如

```
# include "lex.yy.c"
```

的语句即可。使用这条语句, 程序 `yylex()` 可以访问 YACC 中记号的名字, 因为 LEX 的输



出是 YACC 输出文件的一部分, 所以每个 LEX 动作都返回 YACC 知道的终结符号。

联合使用 LEX 和 YACC 来自动生成语法分析器的处理流程如图 4-9 所示。假设已有用户编写好的 LEX 源文件 `scanner.l` 和 YACC 源文件 `parser.y`, 在 UNIX 环境下, 可以使用下列命令得到所需要的语法分析器。

```
lex scanner.l /* 生成含有 yylex() 的 lex.yy.c 文件 */
yacc parser.y /* 生成含有 yyparse() 的 y.tab.c 文件 */
cc -o yaccdemo y.tab.c lex.yy.c /* 生成 yaccdemo 文件 */
```

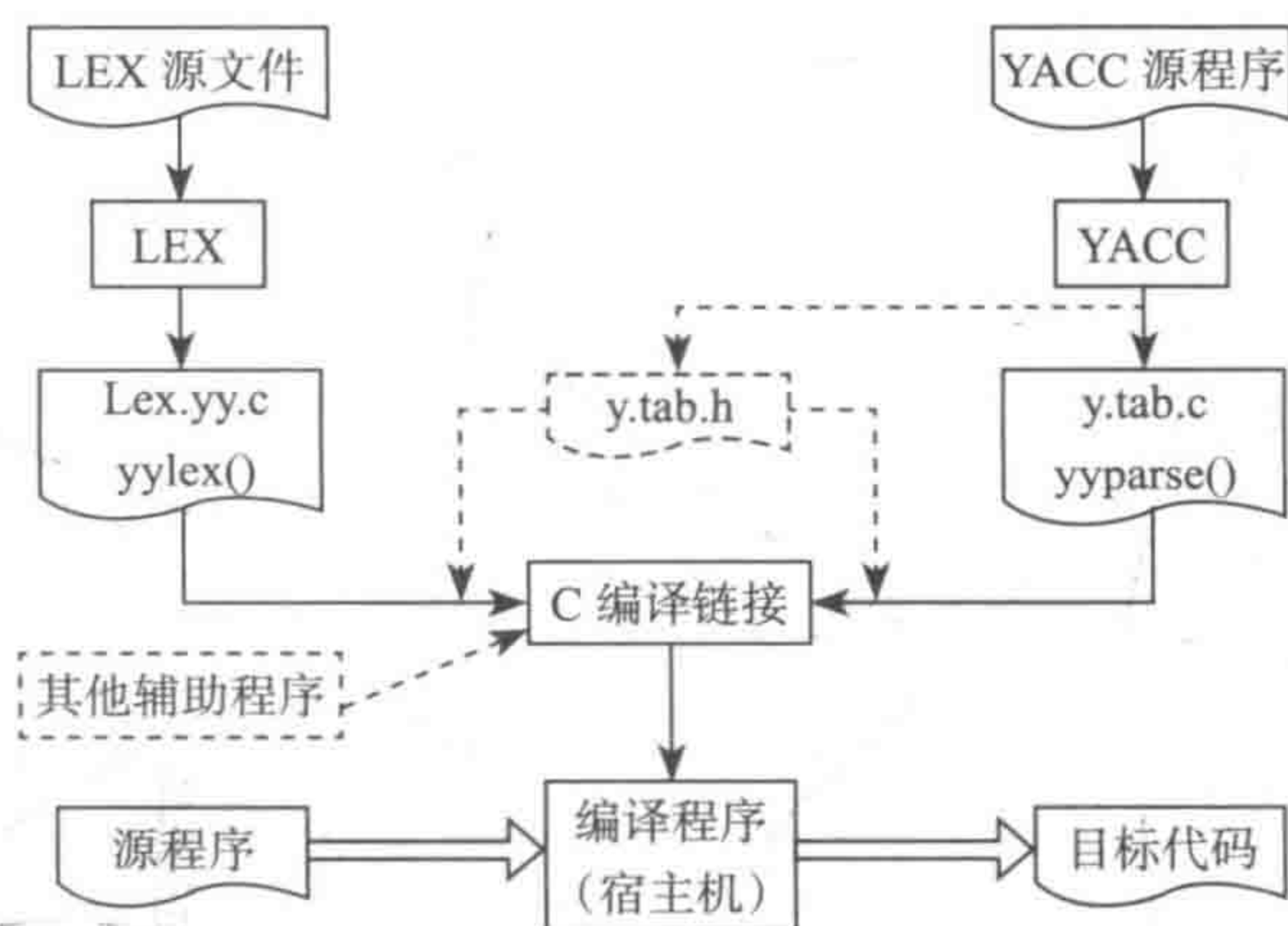


图 4-9 用 LEX 和 YACC 合建编译程序

### 4.3.3 YACC 源程序示例——简单的台式计算器

在本小节, 我们构造一个简单的台式计算器以供参考。该计算器读入一个算术表达式, 然后计算并打印它的值。该算术表达式文法的产生式为:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{digit}$$

其中, digit 表示 0 ~ 9 的单个数字。根据这一文法写出的 YACC 源程序如下:

```
%{
#include <ctype.h>
#include <stdio.h>
%}
% token DIGIT
%%
lines : expr '\n' {printf ( "%d \n", $1 ) ;}
;
expr : expr '+' term {$$ = $1 + $3; }
| term
;
term : term '*' factor {$$ = $1 * $3; }
```

```

        | factor
        ;
factor   :   '(' expr ')'           { $$ = $2; }
        | DIGIT
        ;
%%
main () /* 主函数 */
{
    yyparse ();
}
yyerror (char * s) /* 语法分析函数 yyparse() 出错时调用该函数 */
{
    printf ("%s\n", s);
}
yylex () {
    int c;
    /* 跳过白字符 */
    while ((c = getchar ()) == ' ' || c == '\t')
        ;
    /* 处理数字 */
    if (isdigit (c) )
    {
        yylval = c - '0';
        return DIGIT;
    }
    if (c == '\n')
        return c;
}

```

在上面的 YACC 源程序中，产生式  $E \rightarrow E+T|T$  及其相关的语义动作表示为

```

expr: expr '+' term { $$ = $1 + $3; }
    | term
    ;

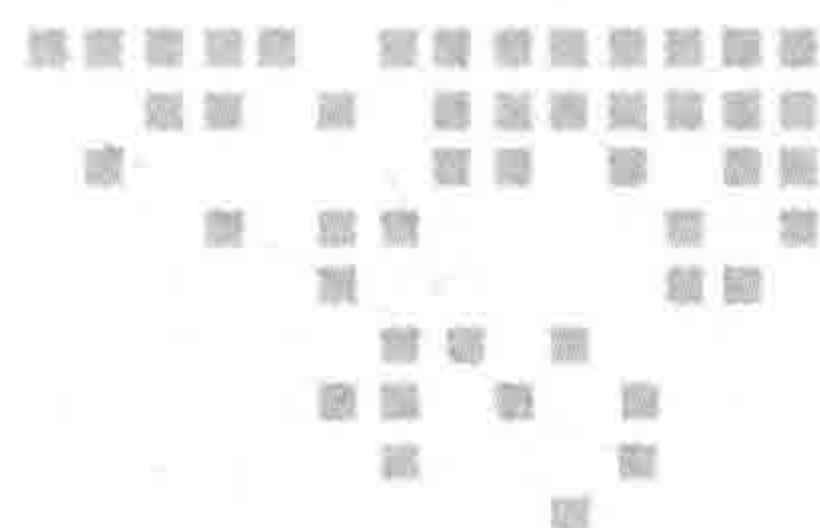
```

在第一个产生式中，非终符 `term` 是右部的第 3 个文法符号，“+”是第 2 个文法符号。第一个产生式的语义动作是把右部 `expr` 的值和 `term` 的值相加，把结果赋给左部非终结符 `expr` 作为它的值。第二个产生式的语义动作描述省略，因为当右部只有一个文法符号时，语义动作缺省就是表示值的复写，即它的语义动作是  $\{ \$\$ = \$1; \}$ 。对于产生式  $T \rightarrow T * F | F$  及其相关的语义动作的处理类似。

## 4.4 本章小结

本章分别以预测分析器的设计与实现为例介绍了如何进行自上而下的语法分析器设计与实现，以 LR 分析器的设计与实现为例介绍了如何进行自下而上的语法分析器设计与实现；给出了这些分析器设计中所涉及的关键算法和具体实现。与此同时还介绍语法分析器的自动生成工具 YACC 软件，给出了其源程序结构、YACC 和 LEX 的接口以及程序示例。





## GCC 编译器分析与实践

GCC 编译器是目前 Linux 操作系统最流行的编译器之一，是 GNU 项目的一个产品。该项目始于 1984 年，目标是以自由软件的形式开发一个完整的类 UNIX 的操作系统。像所有这种规模的软件一样，GNU 项目也经历了一些波折，但目标最终还是实现了。实际上现在在一个功能完备的类 UNIX 操作系统——Linux，已经在世界上广为流传了，并被不计其数的公司、政府和个人成功应用，而该系统及其所有工具和应用都是基于 GCC 编译的。

### 5.1 GCC 编译器概述

作为自由软件的旗舰项目，Richard Stallman 在刚开始编写 GCC 的时候，仅把它当作 C 程序语言的编译器，GCC 是 GNU C Compiler 的缩写。然而，经过多年的发展后，GCC 已经不仅能支持 C 语言，还支持 Ada 语言、C++ 语言、Java 语言、Objective C 语言、Pascal 语言、COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言等。GCC 也不再只是 GNU C 编译器的意思了，而是变成了 GNU Compiler Collection 即 GNU 编译器家族的缩写。

另一方面，GCC 对于各种硬件平台的支持，也可以说是无所不在。对于几乎所有有实际用途的硬件平台，甚至包括有些使用者较少的硬件平台，比如 Don Knuth 设计的 MMIX 计算机，GCC 都提供了完善的支持。

这使得用 GCC 开发的程序具有相当高的可移植性，可以在不修改源代码的基础上轻松地移植到不同的处理器平台上。

GCC 自 2005 年以来的发布历史如图 5-1 所示。

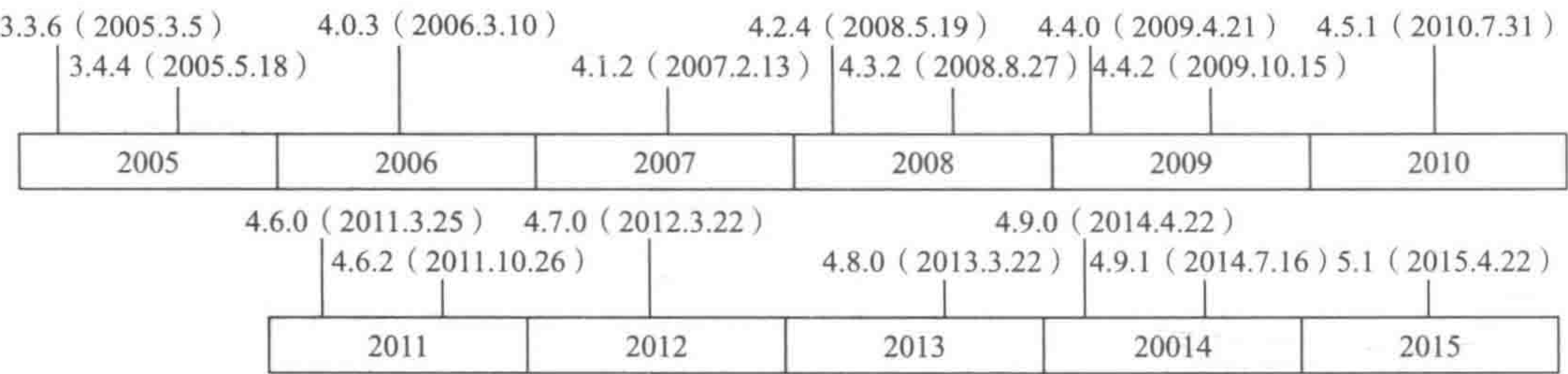


图 5-1 GCC 发展历史

GCC 源代码移植到了超过 60 种平台上，是目前现存的最复杂的开源系统，GCC 现在最新的版本是 5.1，拥有超过 150 万行源代码。

GCC 的最简单用法（以 `hello.c` 为源程序）是在命令行输入下面的语句（从而得到目标程序 `hello`）。

```
gcc hello.c -o hello
```

## 5.2 GCC 编译器的系统结构

GCC 编译器是一种管道（pipeline）架构，通过不同的层次为不同类型的数据进行通信。前端编译器针对特定语言进行词法和语法分析，解析为树状结构和中间表示代码（使用寄存器传递语言（Register Transfer Language, RTL））。后端编译器提供与具体语言无关的分析优化和针对特定目标架构的代码生成，这样的架构利于使用 RTL 创建更快速或者更简洁的代码，最优化的 RTL 被代码生成器获取，然后生成目标代码。

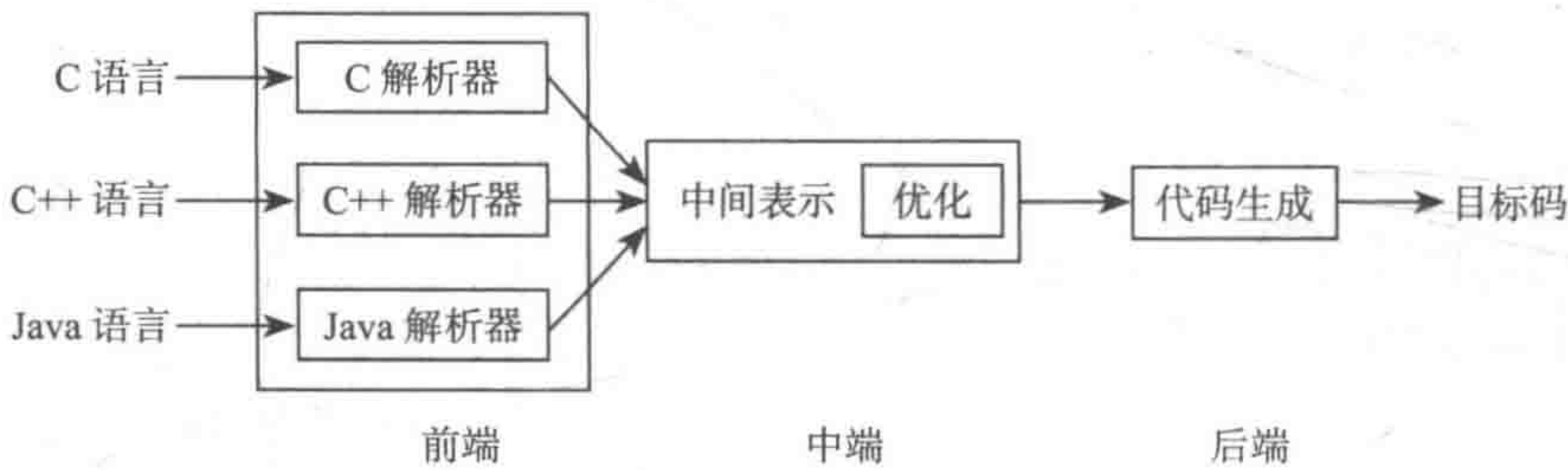


图 5-2 GCC 编译器架构

GCC 的外部接口类似于一个标准的命令行编译器。使用者在命令行下输入 `gcc` 命令，以及一些命令参数，以便决定每个输入文件使用的个别编译选项，输出目标平台的目标代码，并且选择性地执行链接器以产生可执行程序。

除了 Ada 前端大部分以 Ada 写成外，几乎全部的 GCC 都由 C 写成。

### (1) 前端接口

GCC 前端的功能在于产生一棵可让中后端处理的语法树。语法解析器是一个递归调用的函数。



直到 2004 年,程序的语法树结构尚无法与目标处理器架构完全分离,语法树的规则有时针对不同的源语言也不一样,前端会提供一些特定源语言的语法树规则。

2005 年,两种与语言无关的新语法树纳入 GCC,它们是 GENERIC 与 GIMPLE。语法解析器先产生与语言相关的临时语法树,再将它们转成 GENERIC,之后使用 `gimplifier` 技术降低 GENERIC 的复杂结构,成为以较简单的静态单赋值(Static Single Assignment form, SSA)形式为基础的 GIMPLE 形式。这种形式是与语言和处理器架构无关的最优化通用中间表示,适用于大多数的现代编程语言。

### (2) 中端接口

现代编译器通常把与源语言和处理器平台无关的部分放在编译器中端,目的是在中间表示上进行更多的优化。常用的优化技术包括死代码清除、删除无用赋值、常量传播等。

### (3) 后端接口

GCC 后端因不同的处理器宏和特定架构的功能而不同,如不同的数据格式大小、调用方式与大小端等。后端接口的前半部分利用这些信息决定其 RTL 的生成形式,因此虽然 GCC 的 RTL 在理论上不受处理器影响,但在此阶段其抽象指令已被转换成目标架构的格式。

后端经由一次重读取步骤后,利用描述目标处理器的指令集时所取得的信息,将抽象寄存器替换成处理器的真实寄存器,即寄存器分配。寄存器分配非常复杂,因为它必须关注所有 GCC 可移植平台的处理器指令集的规格与技术细节。

后端的最后步骤相当公式化,仅仅将前一阶段得到的汇编语言代码转换其寄存器与内存位置成相对应的机器码。

## 5.3 GCC 编译器的分析程序

在前端,主要包含词法分析和语法分析两方面的工作,GCC 中包含了几千行的 YACC 代码。语法分析负责构造出程序的语法分析树。

为了支持已经存在及以后将会加入的前端,GCC 定义了数十种树节点(不是所有节点都能作为叶子节点)。

以 GCC 4.3.1 为例,对前端和高级的分析及优化而言,树是其核心的数据结构。经过语法分析的源程序都表示为树的形式。这里要说明一下,GCC 中实际上有三种树:GENERIC、GIMPLE 和 SSA。

- **GENERIC**: 语言无关的一种通用表示,源程序经过前端的语法分析后,会被转化成这种形式。其实,在这种形式中可能有与语言相关的节点,对于这样的节点,前端要负责提供一个将其转化成 GIMPLE 的函数。几乎没有什么分析和优化是在这个中间表示上进行的。
- **GIMPLE**: GENERIC 加了某些限制的简化版本,GENERIC 表示的程序会被转化成 GIMPLE 表示,这是真正的与语言无关的三地址表示。有一些分析和优化是在这个



表示上进行的。

- SSA：另外一种语言无关的中间表示，由 GIMPLE 转化而来，大量的分析和优化在其上进行，如循环优化、无用语句清除、死代码清除、phi 结点优化、别名优化、冗余清除等。

## 5.4 GCC 编译器的中间语言及其生成

在树结构优化完成后，开始 RTL 的生成与优化。RTL 是一种低级的中间表示，包含 5 种对象类型：表达式、整型、宽整型、串和向量。

整型相当于 C 中的 int，用十进制表示。宽整型是长度扩展的整型对象，其书写形式也用十进制表示。

字符串即一串字符，在存储器中以 C 的 char\* 形式表示且按 C 语法规则书写。RTL 中的字符串不会为空值。若机器描述中有一空字符串，它在存储器中则表示成一个空指针，而不是通常意义上的指向空字符的指针。在某些上下文中，允许用这种空指针表示空字符串。在 RTL 代码中，字符串经常出现在表达式中。

向量由任意多个指向表达式的指针组成。一个向量中的元素数目是确定的。向量的书写形式为  $[exp_1 \cdots exp_n]$ ，exp 是指向表达式的指针，exp 之间以空格隔开。不生成长度为 0 的向量，而以空指针代替。

表达式根据“表达式代码”（又称 RTX 代码）分类。RTX 代码是 RTL 规范中定义的一个名字，类似于 C 语言中的一个枚举常量。表达式的代码和意义与机器无关。表达式是最重要的一种类型，通常使用指针来引用。

RTL 更加面向后端，它的设计灵感源于 lisp 语言。RTL 是线性的，与机器语言很靠近。也就是说，RTL 可以认为是一个高度抽象的计算机机器语言，由 RTL 语言表示的程序可以很容易被转换为各种体系结构的机器语言程序。在低版本的 GCC 中可以使用 dr 选项（如 `gcc -dr test.c`）来输出程序所对应的 RTL 中间形式。

## 5.5 GCC 编译器的优化

GCC 包含了众多的优化手段，可以从命令行中涉及的优化选项看出来。GCC 提供了 O0、O1、O2、O3 以及 Os 这几种不同的优化级别供用户选择，在这些选项中包含了大部分有效的编译优化选项，并且可以在这个基础上对某些选项进行屏蔽或添加，从而大大降低使用的难度。O0 选项代表不执行优化。O1 添加了 `-fthread-jumps -floop-optimize`。O2 添加了 `-freturn-loop-opt -fgcse -fgcse-sm -fgcse-las -fsched-spec-load -fsched-spec-load-dangerous`。

### （1）O1 包含的选项

优化选项 O1 主要用于进行线程跳转（Thread Jump）和延迟退栈（Deferred Stack Pop）



两种优化。使用本项优化，编译器会尝试减小生成代码的尺寸，缩短执行时间，但并不进行需要占用大量编译时间的优化。

打开的优化选项（需要优化选项时在名字前加上 `-f`，不需要此选项可以用 `-fno-` 前缀）如下：

<code>-fauto-inc-dec</code>	<code>-fcprop-registers</code>	<code>-fdce</code>	<code>-fdefer-pop</code>
<code>-fdelayed-branch</code>	<code>-fdse</code>		<code>-fguess-branch-probability</code>
<code>-fif-conversion2</code>	<code>-fif-conversion</code>		<code>-finline-small-functions</code>
<code>-fipa-pure-const</code>	<code>-fipa-reference</code>		<code>-fmerge-constants</code>
<code>-fsplit-wide-types</code>	<code>-ftree-ccp</code>		<code>-ftree-ch</code>
<code>-ftree-copyrename</code>	<code>-ftree-dce</code>		<code>-ftree-dominator-opts</code>
<code>-ftree-dse</code>	<code>-ftree-fre</code>	<code>-ftree-sra</code>	<code>-ftree-ter</code>
<code>-funit-at-a-time</code>	<code>-fthread-jumps</code>		<code>-floop-optimize</code>

- `-fdefer-pop`：延迟栈的弹出时间。当完成一个函数调用时，参数并不马上从栈中弹出，而是在多个函数被调用后一次性弹出。
- `-fmerge-constants`：尝试横跨编译单元合并同样的常量。
- `-fthread-jumps`：如果某个跳转分支的目的地存在另一个条件比较，而且该条件比较包含在前一个比较语句之内，那么执行本项优化，根据条件是 `true` 或者 `false`，前面那条分支重定向到第二条分支的目的地或者紧跟在第二条分支后面。
- `-floop-optimize`：执行循环优化，将常量表达式从循环中移除，简化判断循环的条件，并且选择性地做强度削弱，或者将循环打开等。在大型复杂的循环中，这种优化比较显著。
- `-fif-conversion`：尝试将条件跳转转换为等价的无分支形式。优化实现方式包括使用条件移动、`min`、`max`、设置标志、`abs` 指令，以及一些算术技巧等。
- `-fdelayed-branch`：这种技术试图根据指令周期重新安排指令。它还试图把尽可能多的指令移动到条件分支前，以便最充分地利用处理器的缓存。
- `-fguess-branch-probability`：当没有可用的反馈信息或 `__builtin_expect` 内联标识时，编译器采用随机模式猜测分支被执行的可能性，并移动对应汇编代码的位置，这有可能导致不同的编译器编译出迥然不同的目标代码。
- `-fcprop-registers`：因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性（两个段要求使用相同的寄存器），并且删除不必要的寄存器复制操作。

## （2）O2 包含的选项

O2 是比 O1 更高级的选项，可进行更多的优化。GCC 将执行几乎所有的不包含时间和空间折中的优化。当设置 O2 选项时，编译器并不进行循环展开（`loop unrolling`）以及函数内联。与 O1 比较，O2 优化增加了编译的时间，但提高了生成代码的执行效率。

O2 打开所有的 O1 选项，并打开以下选项：



-falign-functions	-falign-jumps	-falign-loops
-falign-labels	-fcaller-saves	fcrossjumping
-fcse-follow-jumps	-fcse-skip-blocks	-fdelete-null-pointer-checks
-fexpensive-optimizations	-fforce-mem	-fgcse-lm
-foptimize-sibling-calls	-fregmove	-freorder-blocks
-freorder-functions	-frerun-cse-after-loop	-fsched-interblock
-fsched-spec	-fschedule-insns	-fschedule-insns2
-fstrength-reduce	-fstrict-aliasing	-fstrict-overflow
-ftree-pre	-ftree-vrp	-frerun-loop-opt
-fgcse	-fgcse-sm	-fgcse-las
-fsched-spec-load	-fsched-spec-load-dangerous	

- **-fforce-mem**：在做算术操作前，强制将内存数据传送到寄存器中以后再执行。这会使所有的内存引用潜在的公共表达式，进而产出更高效的代码，当没有公共子表达式时，指令合并将排除个别的寄存器载入。这种优化只涉及单一指令的变量，也许不会有很大的优化效果。但是对于在很多指令（必须是数学操作）中都涉及的变量来说，这会是很显著的优化，因为与访问内存中的值相比，处理器访问寄存器中的值要快得多。
- **-foptimize-sibling-calls**：优化相关的以及末尾递归的调用。通常，递归的函数调用可以被展开为一系列一般的指令，而不是使用分支。这样处理器的指令缓存能够加载展开的指令并且处理它们，与指定需要分支操作的单独函数调用相比，这样更快。
- **-fstrength-reduce**：这种优化技术对循环执行优化并且删除迭代变量。迭代变量是捆绑到循环计数器的变量。
- **-fcse-follow-jumps**：在通常的公共子表达式删除（cse）时，若目标跳转不会被其他路径可达，则扫描整个跳转表达式。例如，当公共子表达式删除遇到 if...else... 语句时，若条件为 false，那么公共子表达式删除会跟随着跳转。
- **-fcse-skip-blocks**：与 -fcse-follow-jumps 类似，不同的是根据特定条件，跟随着 cse 跳转的会是整个块。
- **-frerun-cse-after-loop**：在循环优化完成后，重新进行公共子表达式删除操作。
- **-frerun-loop-opt**：两次运行循环优化。
- **-fgcse**：执行全局公共子表达式删除的遍（pass）。这个遍还执行全局常量和复制传播。这些优化操作试图分析生成的汇编语言代码并且结合通用片段，删除冗余的代码段。如果代码使用计算性的 goto，GCC 指令推荐使用 -fno-gcse 选项。
- **-fgcse-lm**：全局公共子表达式删除将试图移动那些仅仅被自身 store 操作致使无效的 load 操作的位置。这将允许循环内的 load/store 操作序列中的 load 转移到循环的外面（只需要装载一次），而在循环内改变成 copy/store 序列。在选中 -fgcse 后，默认打开。
- **-fgcse-sm**：在一个全局公共子表达式删除后面运行一个 store 移动阶段，这个过程试图将 store 操作转移到循环外面。如果与 -fgcse-lm 配合使用，那么 load/store 操作将



会转变为在循环前 load，在循环后 store，从而提高运行效率，减少不必要的操作。

- `-fgcse-las`：全局公共子表达式删除阶段将删除在 store 后面的不必要的 load 操作，这些 load 与 store 通常是同一块存储单元（全部或局部）。
- `-fdelete-null-pointer-checks`：通过对全局数据流的分析，识别并删除无用的对空指针的检查。编译器假设间接引用空指针将停止程序。如果在间接引用之后检查指针，它就不可能为空。
- `-fexpensive-optimizations`：进行一些从编译的角度来说代价高昂的优化（这种优化据说对于程序执行未必有很大的好处，甚至有可能降低执行效率）。
- `-fregmove`：编译器试图重新分配 move 指令或者其他类似操作数等简单指令的寄存器数目，以便最大化地捆绑寄存器的数目。这种优化尤其对双操作数指令的机器帮助较大。
- `-fschedule-insns`：编译器尝试重新排列指令，用以删除由于等待未准备好的数据而产生的延迟。这种优化将对慢浮点运算的机器以及需要访存的指令的执行有所帮助，因为此时允许其他指令执行，直到访存的指令完成，或浮点运算的指令再次需要 CPU。
- `-fschedule-insns2`：与 `-fschedule-insns` 相似。但是当寄存器分配完成后，会请求一个附加的指令计划阶段。这种优化对寄存器较少，并且访存操作时间大于一个时钟周期的机器有非常好的效果。
- `-fsched-interblock`：这种技术使编译器能够跨越指令块调度指令。这可以非常灵活地移动指令以便使等待期间完成的其他工作最大化。
- `-fsched-spec-load`：允许一些 load 指令进行一些投机性的动作。相同功能的还有 `-fsched-spec-load-dangerous`，其允许更多的 load 指令进行投机性操作。这两个选项在选中 `-fschedule-insns` 时默认打开。
- `-fcaller-saves`：在过程调用时保存和恢复相关寄存器，这种优化只会在看上去能产生更好的代码的时候才被使用（如果调用多个函数，这样能够节省时间，因为只进行一次寄存器的保存和恢复操作，而不是在每个函数调用中都进行）。
- `-freorder-blocks`：在编译函数的时候重新安排基本块，目的在于减少分支的个数，提高代码的局部性。
- `-freorder-functions`：在目标文件范围内重排函数。这种优化的实施依赖特定的已存在的信息，`.text.hot` 用于告知访问频率较高的函数，`.text.unlikely` 用于告知基本不被执行的函数。
- `-fstrict-aliasing`：这种技术强制实行高级语言的严格别名规则。对于 C 和 C++ 程序来说，它确保不在数据类型之间共享变量。例如，整数变量不与单精度浮点变量使用相同的内存位置。
- `-funit-at-a-time`：在代码生成前，先分析整个汇编语言代码。这将使一些额外的优化



得以执行，但是在编译期间需要消耗大量的内存。

- **-falign-functions**：这个选项用于使函数对准内存中特定边界的开始位置。大多数处理器按照页面读取内存，并且确保全部函数代码位于单一内存页面内，就不需要交换代码所需的页面。**-falign-functions=N** 将所有函数的起始地址在  $N$  ( $N=1,2,4,8,16, \dots$ ) 的边界上对齐，默认为机器自身的默认值，指定为 1 表示禁止对齐。**-falign-functions=N, -falign-jumps=N, -falign-loops=N, -falign-labels=N** 这四个对齐选项在“-O2”中打开，其中根据不同的平台  $N$  使用不同的默认值。如果需要指定不同于默认值的  $N$ ，也可以单独指定。比如，对于  $L2\text{-Cache} \geq 1\text{MB}$  的 CPU 而言，指定 **-falign-functions=64** 可能会获得更好的性能。建议在指定了 **-march** 的时候不明确指定这里的值。
- **-falign-jumps**：对齐分支代码到 2 的  $n$  次幂边界。
- **-falign-loops**：对齐循环到 2 的  $n$  次幂边界。
- **-falign-labels**：对齐分支到 2 的  $n$  次幂边界。
- **-fcrossjumping**：这是对跨越跳转的转换代码处理，以便组合分散在程序各处的相同代码。这样可以减少代码的长度，但是也许不会对程序性能造成直接影响。

### (3) O3 包含的选项

打开所有 O2 的优化选项并且增加：

```
-finline-functions      -funswitch-loops      -fpredictive-commoning
-fgcse-after-reload    -ftree-vectorize      -fweb
-frename-registers
```

- **-finline-functions**：内联简单的函数到被调用函数中。由编译器启发式地决定哪些函数足够简单可以进行这种内联优化。在默认情况下，编译器限制内联的尺寸，3.4.6 中限制为 600 条伪指令，可以通过 **-finline-limit=n** 改变这个长度。这种优化技术不为函数创建单独的汇编语言代码，而是把函数代码包含在调度程序的代码中。对于多次被调用的函数来说，为每次函数调用复制函数代码。虽然这样对于减少代码长度不利，但是通过最充分的利用指令缓存代码，而不是在每次函数调用时进行分支操作，可以提高性能。
- **-funswitch-loops**：将无变化的条件分支移出循环，取而代之的将结果副本放入循环中。
- **-fgcse-after-reload**：为了清除多余的溢出，在重载之后执行一个额外的载入删除步骤。
- **-ftree-vectorize**：在 trees 上执行循环向量化。O3 会自动打开 **-ftree-vectorize** 选项，关闭向量化的选项是 **-fno-tree-vectorize**。
- **-fweb**：构建用于保存变量的伪寄存器网络。伪寄存器包含数据，就像它们是寄存器一样，但是可以使用各种其他优化技术进行优化，比如 cse 和 loop 优化技术。这种优化会使得调试变得更加的不可能，因为变量不再存放于原本的寄存器中。
- **-frename-registers**：在寄存器分配后，试图驱除代码中的假依赖关系。这会使调试变得非常困难，因为变量不再存放于原本的寄存器中了。





**注意** 选项的顺序很重要，如果有两个选项互相冲突，则以后一个为准。如“-O3”将打开-finline-functions选项，但是“-O3 -fno-inline-functions”既使用-O3的功能又关闭函数内嵌功能。

#### (4) Os 包含的选项

Os 选项主要是对程序的尺寸进行优化。其打开了大部分 O2 优化中不会增加程序大小的优化选项，并对程序代码的大小进行更深层的优化。

Os 会关闭如下选项：

-falign-functions	-falign-jumps	-falign-loops
-falign-labels	-freorder-blocks	-fprefetch-loop-arrays

- -fprefetch-loop-arrays：生成数组预取指令，对于使用巨大数组的程序可以加快代码执行速度，适合数据库相关的大型软件等。

#### (5) 其他优化选项

- -funroll-loops：执行循环展开的优化，仅对循环次数能够在编译时或运行时确定的循环进行。如果在编译时可以确定迭代的次数非常少而且循环中的指令也非常少，可以使用该选项进行循环展开，以去除循环和复制指令。该选项将循环按照适当的循环因子展开，使得循环内部的指令数目增多，然后发掘展开后的循环中的各指令间的并行（展开后通常会存在），从而加快循环的执行速度。GCC 编译器提供的自动向量化优化机制也是基于循环展开策略的。
- -funroll-all-loops：执行循环展开的优化，对所有循环进行，通常使程序运行得更慢。
- -ftime-report：编译完成后显示编译耗时的统计信息。
- -fmove-all-movables：将所有不变的表达式移动到循环体之外，这种做法的好坏取决于源代码中的循环结构。
- -fprofile-arcs：在使用该调试选项编译程序并运行它以创建包含每个代码块的执行次数的文件后，程序可以再次使用-fbranch-probabilities编译，文件中的信息可以用来优化那些经常选取的分支。如果没有这些信息，GCC 将猜测哪个分支将被经常运行以进行优化。这类优化信息将会存放在一个以源文件为名字的并以“.da”为后缀的文件中。

## 5.6 GCC 编译器的目标代码生成

在目标代码生成阶段，GCC 专门有一个阶段负责将 RTL 转换为 ASM，代码生成的依据是机器描述。

针对每个 CPU 平台，GCC 有对应的 Machine Description 用于指导指令生成。这些代码存放在“gcc/config/<平台名称>”的目录下，比如 Intel 平台在 gcc/config/i386/ 目录下。

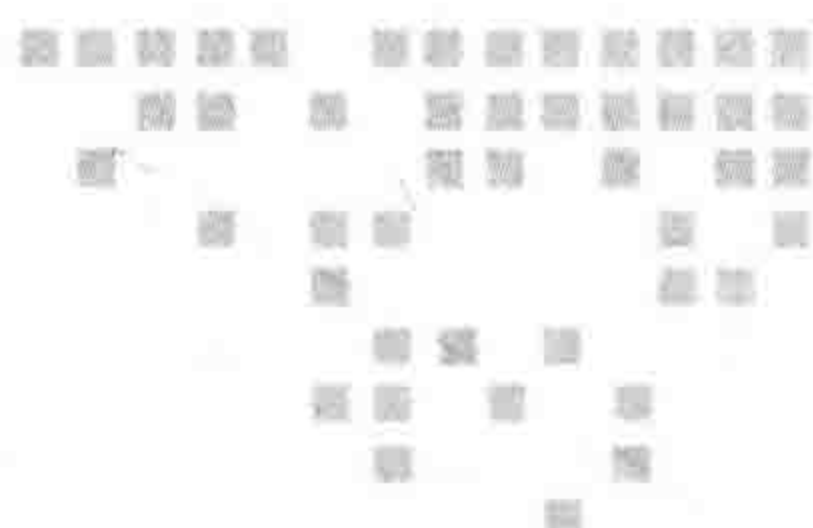
一个 Machine Description 文件是对应平台的核心，比如 gcc/config/i386/i386.md (i386.md 是 md 文件，IA32 和 x86-64 的机器描述) 文件。

GCC 的未来是光明的。GCC 将支持更多的处理器和架构，几乎涵盖整个开发语言。在开发方面还支持一系列不同的语言，比如 Mercury、GHDL (一个用于 VHDL 的 GCC 前台语言) 和统一的并行 C 语言 (Unified Parallel C Language, UPC 语言)。由于 GCC 的进步，几乎所有的软件将从 GCC 的进步中获得好处 (从 Linux 和 Berkeley 软件到 Apache 等)，通过 GCC 进行软件编译将更加简洁和快速。

## 5.7 本章小结

本章对编译器 GCC 进行了详细介绍，主要对其系统结构、分析程序、中间语言及其生成、优化、目标代码生成等方面及相关使用方法进行了阐述。





## LLVM 编译器分析与实践

### 6.1 LLVM 编译器概述

#### 6.1.1 起源

LLVM 是底层虚拟机的英文缩写。起初 LLVM 项目作为伊利诺伊大学香槟分校的研究原型以及该校编译课程的教学框架。由于 LLVM 项目新颖的设计理念、模块化的设计思路以及同现代编译器课程的紧密联系，越来越多的研究人员采用 LLVM 作为原型实现其在编译领域的新思路。在学术界，LLVM 成功架起了编译理论同实践的桥梁。除此之外，同 GCC 相比，LLVM 项目具有更加开放的授权，因此 LLVM 项目也引起了业界的高度关注。众多公司出于不同需求参与到 LLVM 项目当中，并做出了各自的贡献，大大拓展了基于 LLVM 的编译器所针对的编程语言，以及生成代码所针对的目标架构，从而使得 LLVM 项目的成熟度显著提高，并从学术实验软件过渡到了商业产品中的健壮框架。该项目名称也从底层虚拟机 (Low Level Virtual Machine) 更改为 LLVM。

目前，根据不同的应用场景，LLVM 可能指示以下内容：

- 1) LLVM 项目 / 基础架构。此时，LLVM 指代多个构建一个完整编译器的项目，包括前端、后端、优化器、汇编器、链接器、libc++、compiler-rt 以及 JIT 引擎。
- 2) 基于 LLVM 的编译器。此时，LLVM 指代部分或者完全采用 LLVM 基础架构构建的编译器。例如，某个编译器可以采用 LLVM 作为前端或者后端，但是使用 GCC 以及 GNU 系统库函数进行最后的链接。
- 3) LLVM 库。此时，LLVM 指代 LLVM 基础架构中可复用的代码部分。
- 4) LLVM 内核。此时，LLVM 指代在中间表示级别上所进行的优化以及后端算法。
- 5) LLVM 中间表示 (LLVM IR)。此时，LLVM 指代 LLVM 编译器的中间表示。



## 6.1.2 相关项目

随着 LLVM 项目的发展，该项目已经衍生出一系列子项目，其中较为重要的如下所示：

### 1. LLVM 内核 (LLVM Core)

LLVM 内核库文件提供了一个与源语言以及目标架构无关的现代优化器，同时为很多主流 CPU 提供代码生成。这些库基于 LLVM IR 进行构建，这些库文件大大简化了为新型编程语言设计编译器或者将其移植到现有编译器的难度。

### 2. Clang

Clang 是一个支持 C、Objective-C、C++ 和 Objective-C++ 语言的开源编译器。在 LLVM 项目的开展过程中，其设计上的最重大的决策是将后端与前端分离开来作为两个单独的项目，也就是 LLVM 内核和 Clang。LLVM 成为围绕 LLVM 中间表示 (LLVM IR) 的一组工具集，并依赖改进的 GCC 将高级语言代码转化为存储在 Bitcode 文件中的特定中间表示。Bitcode 文件同 Java 中的 Bytecode 文件相类似。Clang 作为 LLVM 项目组第一个专门设计的前端的出现，对于 LLVM 项目而言是一个里程碑事件。Clang 将 C 或者 C++ 代码转换为 LLVM IR，此外，利用 Clang 的库文件还能够开发一系列功能强大的工具。

### 3. Clang extra tools

Clang extra tools 是一组基于 Clang 构建的工具，该工具体能够读取 C 或者 C++ 代码进行重构以及代码分析。工具集中的主要工具包括：Clang Modernizer、Clang Tidy、Modularize 以及 PPTrace 等。

### 4. Compiler-RT

Compiler-RT 项目对目标架构提供硬件不支持的部分底层功能。例如，32 位的目标架构通常缺少支持 64 位目标架构的指令。Compiler-RT 通过提供与目标架构相关的优化函数在使用 32 位指令的情况下能够实现对 64 位架构的支持。

### 5. DragonEgg

DragonEgg 将 LLVM 优化器以及代码生成器同 GCC 解析器进行集成，使得 LLVM 能够编译 Ada、Fortran 以及其他 GCC 编译器前端所支持的语言，并能够获得 Clang 所不支持的 C 语言特性。

### 6. LLDB

LLDB 项目基于 LLVM 和 Clang 的库文件构建了一个功能强大的原生调试器。LLDB 使用 Clang 的 AST 以及表达式解析器、LLVM JIT、LLVM 反汇编器等工具提供了天衣无缝的使用体验。其执行速度快、同 GDB 相比具有更高的内存使用效率。

### 7. libc++ 与 libc++ ABI

libc++ 与 libc++ ABI 项目为 C++ 标准库提供了标准规划以及高效实现，包括对 C++ 编



程语言标准 C++ 11 的全部支持。

### 8. Vmkit

Vmkit 项目旨在基于 LLVM 技术实现 Java 以及 .NET 虚拟机。

### 9. Klee

Klee 项目实现了一个符号虚拟机，该虚拟机适用定理证明器尝试遍历代码中的所有动态执行路径，进而找到代码中的缺陷或者证明代码的安全特性。Klee 的主要特点是其能够在检测到代码缺陷时自动生成测试用例。

### 10. Poly

Poly 项目实现了一组本地缓存的优化，以及使用 polyhedral 模型的自动并行化以及向量化。

### 11. lld

lld 项目旨在实现 Clang/LLVM 的内建链接器。目前，Clang 必须调用系统链接器才能够生成可执行代码。

## 6.2 经典编译器概述

对于传统静态编译器（例如大多数的 C 编译器）而言最流行的设计方式是三阶段设计方式，其中各自阶段的主要组成部分分别是前端、优化器和后端。如图 6-1 所示。



图 6-1 三阶段编译器的主要组件

前端主要负责对源代码进行解析，检查是否存在错误，并建立一个同语言相关的抽象语法树（AST）表示输入代码。根据实际情况，可以将 AST 转化为一种新的表示形式进行优化，优化器和后端则对前端得到的代码进行操作。

优化器主要负责通过各种变换来尝试改善代码的运行时间，如消除冗余的计算等工作。优化器通常或多或少地独立于源语言和目标架构。

后端（也称为代码生成器）将代码映射到目标架构指令集。除了要映射到正确代码，还需要保证所生成的代码能够利用所支持架构的特性。编译器后端的公共部分主要包括指令选择、寄存器分配和指令调度等。

该模型也同样适用于解释器和 JIT 编译器。

### 6.2.1 经典编译器设计的启示

当编译器需要对多种源语言或目标架构提供支持时，经典编译器设计上的优越性就全部展现了出来。如果编译器在优化器中使用通用代码表示，那么前端可以针对任何其所能



编译的语言，而后端可以适用于任何目标架构，如图 6-2 所示。

采用三阶段设计方式的编译器主要优势包括：

优势 1：基于三阶段设计，对编译器进行移植使之对新的源语言（例如 C# 或者 F#）提供支持仅仅需

要实现一个新的前端，现有优化器和后端则可实现复用。如果编译器中的上述内容没有分离，那么实现对一种新的源语言的支持则需要从头开始，假设需要支持  $N$  种目标架构以及  $M$  种源语言，那么就将需要  $N \times M$  种编译器。

优势 2：同仅仅支持某一种源语言或者某一种目标架构的编译器相比，采用三阶段设计的编译器可以服务于更多的开发团队。对于开源项目而言，这意味着能够有更多开发人员参与到开发社区中，从而对编译器做出更多改进。这就是为什么服务于更多社区的开源编译器（例如 GCC），同应用范围有限的编译器（例如 FreePascal）相比往往会产生更好的优化机器码。

优势 3：实现前端所需要的技术要求同实现优化器和后端所需要的技术要求是不同的。分离设计使得对于前端开发人员而言，改进与维护编译器都变得更加容易。这是三阶段设计的另一个优势。虽然这不是一个技术问题，但是在实践过程中却很重要，特别是对于那些希望尽可能减少障碍的开源项目团队而言，任一项工作难度的降低都可能带来开发效率的提高。

## 6.2.2 现有编译器的实现

虽然三阶段设计具有明显的优势，并且在编译器教材中进行了详细介绍，但是在实际工作中却几乎从未完全实现。纵观开源语言的发展历程不难发现，Perl、Python、Ruby 以及 Java 语言都不共享代码。此外，虽然 Haskell 编译器（GHC）和 FreeBASIC 可以重定向到多种不同的处理器，但是它们仅提供对特定源语言的支持。JIT 编译器也主要采用各种专用编译器技术，适用于包括图像处理、正则表达式、显卡驱动程序，以及其他需要 CPU 密集工作的领域。

关于三阶段设计的还有三个案例。第一个案例是 Java 虚拟机。Java 虚拟机包括 JIT 编译器、运行时支持以及定义明确的字节码格式。这意味着，任何编译到字节码的语言都只需将关注点放在优化器以及 JIT 编译器上。这样做的考虑是，上述实现对运行时选择能够提供一定程度的灵活性：都有效地开展 JIT 编译、垃圾收集，并且能够使用特定的对象模型。当语言同该模型之间的匹配并不紧密时，编译过程难以达到最优的性能。

另一个案例是编译器技术复用最常见的方式，即将输入源代码翻译为 C 代码（或其他语言），然后将其发送到现有的 C 编译器，从而实现了优化器和代码生成器的复用。这种方

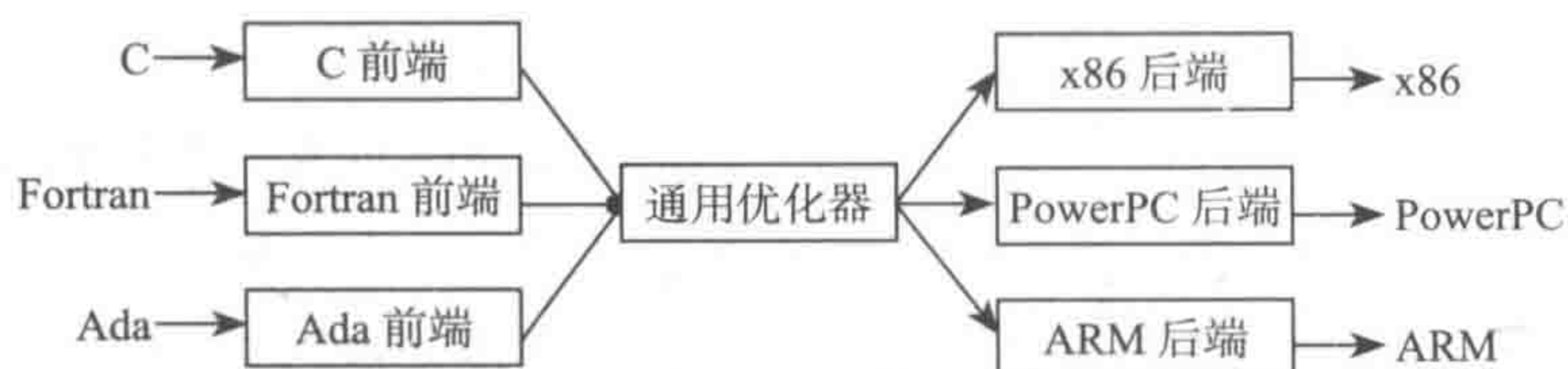


图 6-2 三阶段编译器设计的重定向能力



式具有较好的灵活性并能够提供运行时的控制，对于前端开发人员而言，前端的理解、实现与维护也变得更加容易。但是，这种方式可能影响异常处理的效率，造成较差的调试体验，减慢编译的速度，并且对于需要保证尾部调用（或者 C 语言不支持的其他特性）的语言而言可能存在多种问题。

最后一个案例是 GCC。GCC 支持多种前端和后端，并且具有积极、广泛的社区开发人员。GCC 很早就被作为 C 编译器，支持多种目标架构。随着岁月的流逝，GCC 的设计更加清晰。GCC 在版本 4.4 中，在优化器中采用了一种全新的表示形式（称为“GIMPLE 元组”），同之前相比这种表示方式更接近同前端表示分离的表示方式。另外，Fortran 和 Ada 前端都采用简洁的抽象语法树（AST）进行表示。

虽然上述三个案例都取得了成功，但同时也存在较大的局限性，这是因为它们在设计之初都被当作一个整体程序而设计。举例来说，将 GCC 嵌入到其他应用程序中、使用 GCC 作为运行时 / JIT 编译器，或者提取和复用 GCC 中的代码片段都是不现实的。因此，想用 GCC 的 C++ 前端进行文档生成、代码索引、重构以及静态分析工具的人们不得不将 GCC 作为一个整体程序使用，将其生成的信息作为 XML，或者编写插件插入到 GCC 过程以注入外部代码。

GCC 中的代码片段之所以不能复用为库文件有多种原因，包括全局变量的滥用、弱执行不变量、设计不佳的数据结构、庞大的代码库、宏的使用，这些都影响了编译器对多种源代码 - 目标架构的组合提供支持。但是，最困难的还是源自于 GCC 的早期架构。其中，对 GCC 造成较大影响的是其分层和抽象泄漏（Leaky Abstraction）问题：后端需要遍历前端生成的 AST 信息才能生成调试信息，前端直接生成后端的数据结构，以及整个编译器依赖于由命令行接口设置的全局数据结构。

### 6.3 LLVM 的设计

基于 LLVM 的编译器也采用了三阶段的设计方式，包括前端、优化器以及后端。图 6-3 展示了基于 LLVM 编译器的三阶段设计。

前端：前端负责对输入代码进行解析，对输入代码中的错误进行验证与诊断，然后将完成解析的代码转换为 LLVM IR（转换方式通常是构建 AST，然后将 AST 转换

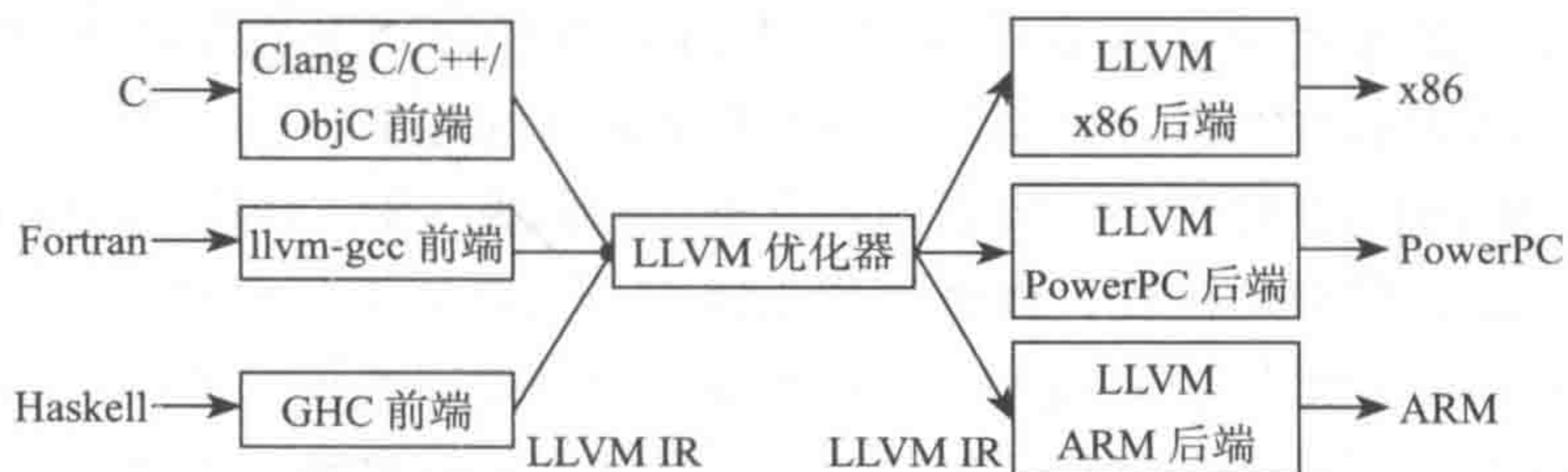


图 6-3 LLVM 三阶段设计的实现

为 LLVM IR)。前端包括词法分析器、语法分析器、语义分析器以及 LLVM IR 代码生成器。Clang 项目实现了所有与前端相关的步骤，并且提供了插件结构以及独立的静态分析工具进



行更加深入的分析。

优化器：优化器主要对 LLVM IR 有选择性地进行分析，从而对代码进行改进，然后发送到代码生成器生成本地机器码。LLVM IR 具有两种文件表示形式，即便于人工理解的表示形式以及二进制编码的表示形式。

后端：后端主要负责代码生成，将 LLVM IR 转换成与目标架构相关的汇编代码或者对象文件。寄存器分配、循环优化、窥孔优化、目标架构相关的优化等工作主要在后端完成。

### 6.3.1 LLVM 中间表示

在编译过程的不同阶段，LLVM 采用不同的形式进行代码表示：

- LLVM IR：LLVM 将高级语言转换为 LLVM IR，LLVM IR 采用静态单赋值（SSA）进行表示，LLVM IR 的代码组织为三地址指令，并拥有数量无限的寄存器。
- 抽象语法树（AST）：当将 C 或者 C++ 转换到 LLVM IR 时，Clang 在内存中采用抽象语法树（类 TranslationUnitDecl）表示代码。
- 有向无环图：当将 LLVM IR 转换到机器相关的汇编语言时，LLVM 首先将代码转换为有向无环图（DAG）进行简单的指令选择（类 SelectionDAG），然后将其再转换为三地址表示形式进行指令调度（类 MachineFunction）。
- 对象文件：在汇编器以及链接器的实现过程中，LLVM 使用对象文件（类 MCModule）来存储代码表示。

其中，LLVM IR 是 LLVM 代码表示中最重要的代码表示形式。LLVM IR 不仅可以保存在内存中，还可以存储在磁盘上，LLVM IR 的编码形式反映了 LLVM 项目早期对代码进行全生命周期优化的考虑。

LLVM IR 定义规范，同时也是 LLVM 前端同优化器交互的接口。这意味着在 LLVM 前端的开发过程中仅需要了解 LLVM IR 是什么，它是如何工作的，以及它所需要的不变量。由于 LLVM IR 可以采用文本表示形式，所以，构建的前端可以以文本形式输出 LLVM IR，然后将其发送到用户选择的优化器以及代码生成器。

LLVM IR 是 LLVM 在不同应用中取得成功的关键因素之一。虽然 GCC 编译器同样也是一款架构出众的编译器，但是 GCC 编译器却不具备这样的特性。因为 GCC 编译器的中间表示 GIMPLE 并不是一种自包含的表示形式。举个例子，当 GCC 的代码生成器抛出 DWARF 调试信息时，GCC 返回并遍历源码级“树”结构。GIMPLE 自身使用“元组”（tuple）表达代码中的操作，但是至少 GCC 4.5 仍然以源码级树结构索引的形式表示操作数。因此，为了实现 GCC 前端，前端开发人员需要知道并生成 GCC 的树数据结构以及 GIMPLE。GCC 的后端也存在相似的问题，开发者需要知道 RTL 后端的工作细节。由于 GCC 中间表示的表达能力有限，并且不能以文本形式读写 GIMPLE（以及形成代码表示的相关数据结构），造成的结果就是难以利用 GCC 对编译技术进行实验，也因此导致 GCC 的前端相对较少。



### 6.3.2 LLVM 库文件

同 GCC 或者虚拟机（例如 JVM 和 .NET 虚拟机）不同，LLVM 的目标并不是成为一个整体的命令行编译器，而是被设计为一组库文件的集合。LLVM 作为基础架构，拥有诸多实用的编译技术，可以用来解决各种特定问题（例如构建一个 C 编译器，或者构建某一特效流水线的优化器）。虽然这是 LLVM 功能最强大的特性之一，但也是其最不容易被理解的设计初衷之一。

以优化器的设计为例：优化器读取 LLVM IR，对其进行详细分析，然后生成执行速度可能更快的 LLVM IR。在 LLVM 中（如同在其他编译器一样），优化器被组织为一条执行不同优化遍（Pass）的流水线，每一遍都对输入进行某些优化操作。优化操作的常见情况包括内联（在调用位置替换调用实体）、表达式关联、循环不变代码外提等。依据优化级别的不同，执行不同的优化遍。例如，在 O0 级别（无优化），Clang 编译器不执行任何的优化遍，在 O3 级别，编译器（如 LLVM 2.8）在优化过程中执行 67 遍处理。

LLVM 的每一遍处理都编程实现为 Pass 类的派生类。大多数的遍编程实现为单个 .cpp 文件，Pass 类的子类在匿名空间进行定义。为了使每一遍扫描发挥实际作用，文件之外的代码也能够使用该类，文件中需要导出一个函数（创建 pass）。图 6-4 是一个创建遍的实例。

```
namespace {
    class Hello : public FunctionPass {
    public:
        // 在被优化的 LLVM 中间表示中打印出函数名
        bool runOnFunction(Function &F) {
            cerr << "Hello: " << F.getName() << "\n";
            return false;
        }
    };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

图 6-4 扫描创建实例

如上所述，LLVM 优化器提供了很多不同的遍处理方式，每个遍的编程风格均类似。这些遍被编译成一个或多个 .o 文件，这些 .o 文件构成一系列的库文件（类似于 UNIX 系统中的 .a 文件）。这些库文件提供各种类型的分析和转换能力，并且各遍尽可能采用松耦合的方式进行编程：每个遍都能够独立运行，或者通过显式声明对其他遍的依赖调用相应的功能。当需要进行一系列的优化处理时，LLVM 的遍管理器（LLVM Pass Manager）使用显式依赖信息描述遍间的依赖关系，并且对遍的执行进行优化。

虽然库文件以及库文件中封装的抽象能力作用巨大，但是实际上它们并不解决问题。只有当构建需要运用编译器技术的工具时，上述内容才能够发挥作用。举个例子，当试图构建一个图像处理语言的 JIT 编译器时，该 JIT 编译器的开发人员需要考虑很多限制，如图



像处理语言对编译时间延迟是高度敏感的，还有出于性能方面的考虑，优化过程可能移除部分不适用的语言特性。这时，LLVM 库文件的设计能够大大提高构建效率。

LLVM 优化器采用的基于库文件的设计使得开发人员能够选择所需的遍，并能够设定遍的执行顺序：如果所有内容都在一个函数内进行定义，则没有必要浪费时间进行内联。如果没有指针，那么重名分析和内存优化也无需考虑。但是，LLVM 也并不能解决所有优化问题。由于遍实现为模块，而遍管理器自身不知道任何遍的内部信息，所以开发人员需要开发与语言相关的遍来克服 LLVM 优化器中的缺陷，或者将语言相关的优化时机展示出来。图 6-5 是一个图像处理编译程序的例子。

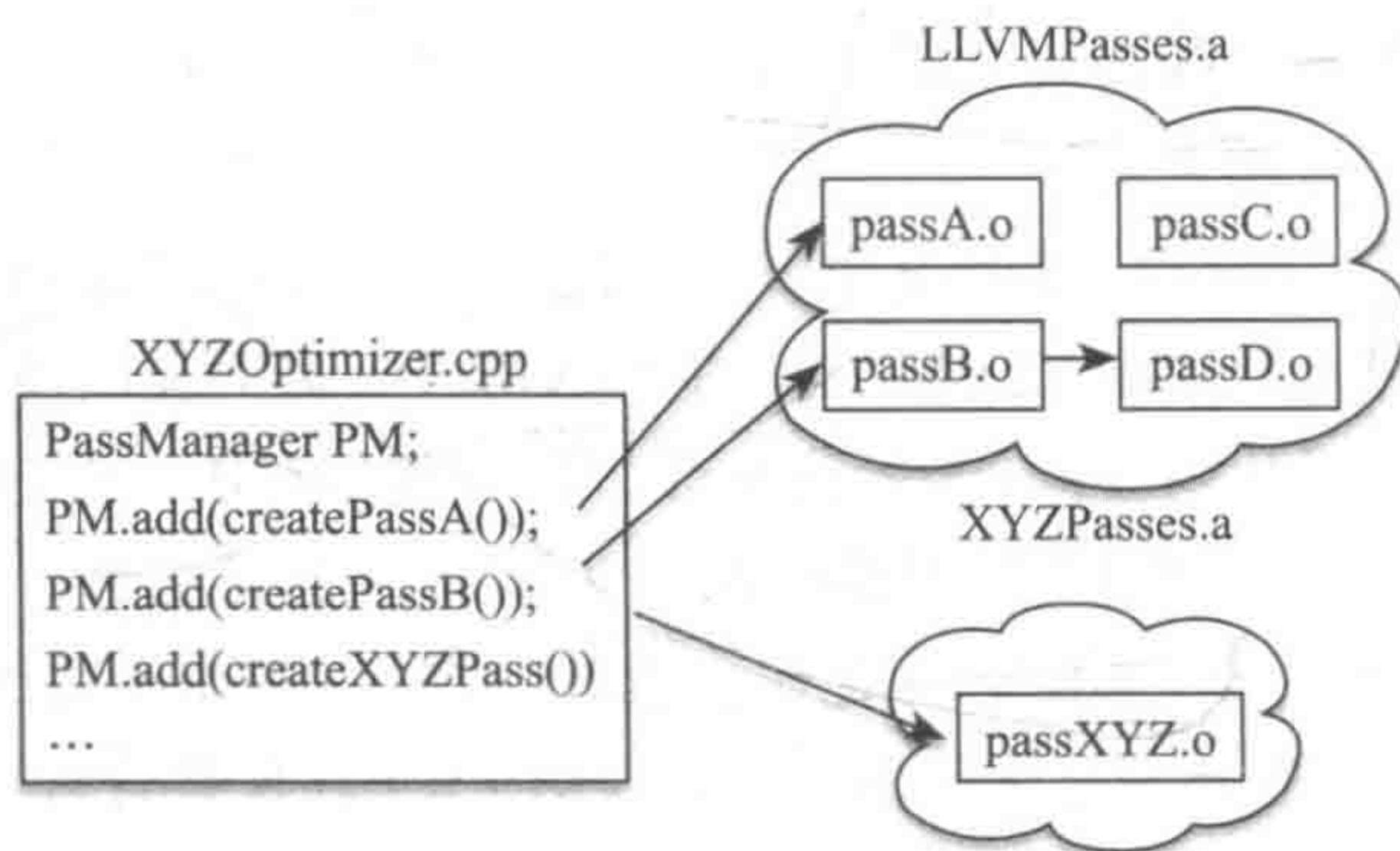


图 6-5 使用 LLVM 的 XYZ 系统

一旦选定所需进行的优化（或者代码生成器中做出类似的决策），那么可以

将图像处理编译器构建为一个可执行程序或者动态库。由于 LLVM 编译器遍集合的唯一索引是在每个 .o 文件中定义的 create 函数，而且由于优化器位于 .a 文件库中，所以只有实际使用的优化遍才参与到同末端代码的链接，而无需链接整个 LLVM 优化器。在上面的例子中，由于存在一个到 PassA 和 PassB 的索引，因此 PassA 和 PassB 将被链接。由于 PassB 使用了 PassD 进行分析，所以需要链接 PassD。但是，由于 PassC 并未使用，因此 PassC 的代码不会被链接到图像处理编译器中。

这正是 LLVM 基于库的设计的优势所在。这种设计方式使得 LLVM 具备大量特性，其中部分特性适用于部分特定用户，并且不会给只需要完成简单工作的用户带来任何不便。相反，传统编译器的优化器通常构建为耦合紧密的代码，因此很难被分割、梳理并进行加速。而使用 LLVM 则能够在无需了解整个系统是如何搭建在一起的情况下，清晰地理解单个优化器的作用。

但是，基于库文件的设计方式也是人们对 LLVM 产生误解的一个原因：LLVM 库文件具备众多特性，但是实际上它们自己不做任何事情，而是由基于库文件所构建系统的设计者来决定库文件的使用。这种分层、托管和集中于子集的特性也是 LLVM 优化器能够被用于不同应用的原因。

## 6.4 LLVM 前端

编译器前端主要负责在生成目标架构的代码之前，将源代码转化为编译器的中间表示。由于每种编程语言都具有不同的语法特点和语义特点，所以前端通常仅仅针对一种编程语言或者一族类似的编程语言。



Clang 是 LLVM 项目的原生编译器，支持 C、Objective-C、C++ 和 Objective-C++ 语言。同 LLVM 的名称类似，在不同的情况下，Clang 具有不同含义：

- 前端（以 Clang 库文件形式实现）。
- 编译器驱动（以 clang 命令行以及 Clang 驱动库文件形式实现）。
- 编译器（以 clang -ccl 命令实现）。但是在 clang -ccl 命令执行过程中所涉及的编译器并不仅仅只涉及 Clang 库文件，该命令的执行广泛应用了 LLVM 中的库文件来实现编译器的优化、后端以及集成汇编器。

在通常情况下，Clang 多指代 Clang 库文件，即 LLVM 项目中类 C 语言的前端。为了将源代码转换为 LLVM IR 的 bitcode，源代码需要经过一系列中间步骤，如图 6-6 所示。



图 6-6 前端所涉及的主要步骤

#### 6.4.1 前端库文件

Clang 采用了模块化设计，通过若干个库文件予以实现，而不仅仅是作为驱动程序或者编译程序。其中 libclang 是同 Clang 用户交互的最重要接口，通过 C 语言实现前端功能。libclang 中包含若干个 Clang 库文件，每个库文件既可以单独链接也可以组合链接。

- libclangLex: 用于词法分析、宏定义、标记的处理以及编译指示信息的构造。
- libclangAST: 抽象语法树构建、操作以及遍历等功能。
- libclangParse: 对词法分析阶段得到的结果进行语法分析。
- libclangSema: 用于语义分析，对 AST 进行验证。
- libclangCodeGen: 使用目标架构相关的信息对 LLVM IR 代码生成进行处理。
- libclangAnalysis: 包含用于静态分析的资源。
- libclangRewrite: 对代码重写提供支持，并为代码重构工具的开发提供基础架构。
- libclangBasic: 提供基础功能实现，包括内存分配抽象、源代码定位以及代码诊断等。

#### 6.4.2 词法分析

词法分析是前端处理过程的第一个步骤，词法分析将以文本形式输入的源代码分解为一组单词与标记。其中每个单词与标记都是编程语言的一个子集，编程语言中的保留字被转换为编译器中的内部表示。在 Clang 中，保留字存放在文件 include/clang/Basic/TokenKinds.def 中，TokenKinds.def 文件版段如图 6-7 所示。

文件中的定义位于 tok 命名空间，在词法分析过程中，编译器通过访问该命名空间可以判断保留字的存在。



```

//===-----//
// Preprocessor keywords.
//===-----//
.....

// GNU Extensions.
PPKEYWORD(import)
PPKEYWORD(include_next)
PPKEYWORD(warning)
PPKEYWORD(ident)
PPKEYWORD(sccs)
PPKEYWORD(assert)
PPKEYWORD(unassert)
.....

//===-----//
// Language keywords.
//===-----//
.....

// C99 6.4.6: Punctuators.
PUNCTUATOR(plus, "+")
PUNCTUATOR(plusplus, "++")
PUNCTUATOR(plusequal, "+=")
PUNCTUATOR(minus, "-")
.....

PUNCTUATOR(percentequal, "%=")
PUNCTUATOR(less, "<")
PUNCTUATOR(lessless, "<<")
PUNCTUATOR(lessequal, "<=")
.....

KEYWORD(float ,KEYALL)
KEYWORD(for ,KEYALL)
KEYWORD(goto ,KEYALL)
KEYWORD(if ,KEYALL)
.....

KEYWORD(struct ,KEYALL)
KEYWORD(switch ,KEYALL)
KEYWORD(typedef ,KEYALL)
KEYWORD(union ,KEYALL)
KEYWORD(unsigned ,KEYALL)
KEYWORD(void ,KEYALL)
KEYWORD(volatile ,KEYALL)
KEYWORD(while ,KEYALL)

//===-----//
// Objective-C @-preceded keywords.
//===-----//
.....

OBJC1_AT_KEYWORD(not_keyword)
OBJC1_AT_KEYWORD(class)
OBJC1_AT_KEYWORD(compatibility_alias)
OBJC1_AT_KEYWORD(defs)
OBJC1_AT_KEYWORD(encode)
OBJC1_AT_KEYWORD(end)
OBJC1_AT_KEYWORD(implementation)
OBJC1_AT_KEYWORD(interface)
OBJC1_AT_KEYWORD(private)
OBJC1_AT_KEYWORD(protected)
.....

```

图 6-7 TokenKinds.def 文件片段



例如对于图 6-8 中所示代码，可以将词法分析后得到的标记以及结果通过命令 `clang -cc1` 进行转储，得到如图 6-9 所示内容。

```
int min(int a,int b){
    if(a<b)
        return a;
    return b;
}
```

图 6-8 代码示例

图 6-9 中每种构造都以其类型所对应的前缀开头，例如，“(”的类型为 `l_paren`，“<”的类型为 `less`，不是保留字的字符串类型为 `identifier`。

```
if 'if' [StartOfLine] [LeadingSpace] Loc=<min.c:2:3>
l_paren '(' [LeadingSpace] Loc=<min.c:2:6>
identifier 'a' Loc=<min.c:2:7>
less '<' [LeadingSpace] Loc=<min.c:2:9>
identifier 'b' [LeadingSpace] Loc=<min.c:2:11>
r_paren ')' Loc=<min.c:2:12>
return 'return' [StartOfLine] [LeadingSpace] Loc=<min.c:3:5>
identifier 'a' [LeadingSpace] Loc=<min.c:3:12>
semi ';' Loc=<min.c:3:13>
```

图 6-9 代码示例语法分析结果

### 6.4.3 语法分析

词法分析对源代码进行了标记，语法分析对得到的标记进行组合，形成表达式、语句以及函数体，并检查一组标记是否同其所放置的位置相匹配。这一分析过程又可以称为解析，以标记流作为输入，以抽象语法树（AST）作为输出。

AST 中的结点表示声明、语句以及类型。因此，表示 AST 中结点主要涉及 3 种核心类：Decl、Stmt 以及 Type。C 或者 C++ 编程语言中的每种结构都在 Clang 中以继承自上述 3 种核心类的 C++ 类的形式表示。图 6-10 展示了类的继承架构。

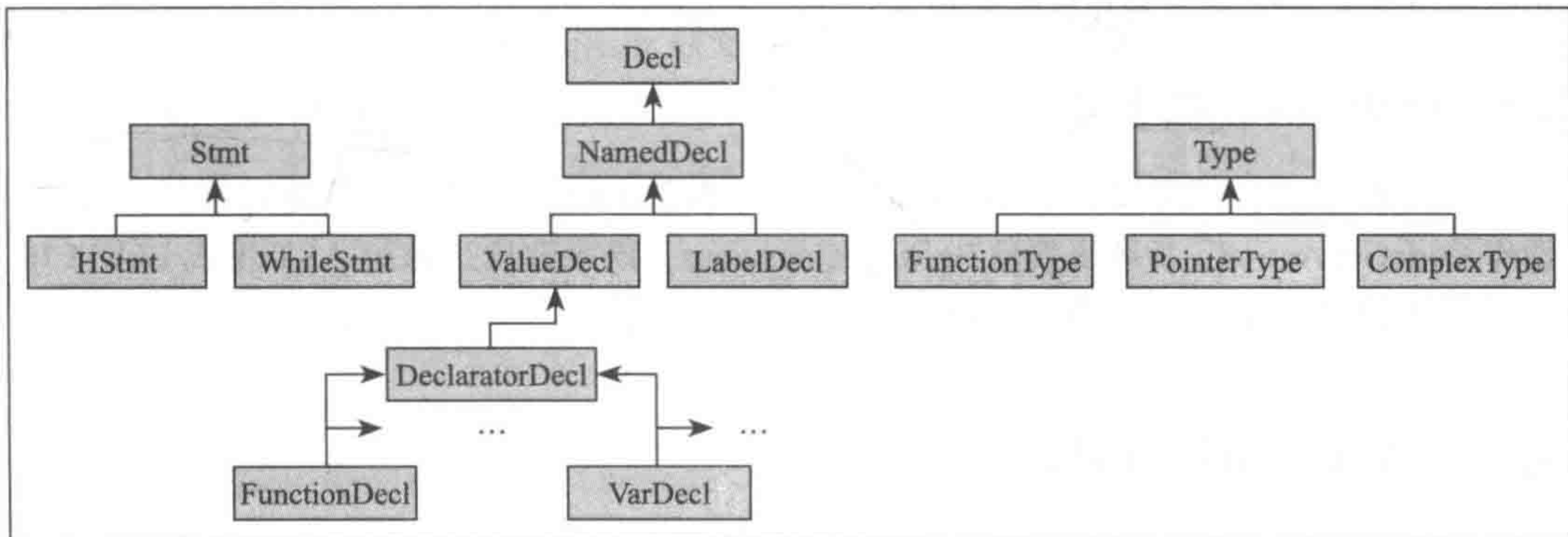


图 6-10 Clang 核心类继承关系



例如，类 `IfStmt`（表示 `if` 语句）直接继承自类 `Stmt`。类 `FunctionDecl` 与 `VarDecl` 分别用于存储函数以及变量的声明或者定义，均以类 `Decl` 为父类。

顶层 AST 结点是 `TranslationUnitDecl`，它是所有其他 AST 结点的根结点，并且表示整个转换单元。

词法分析阶段生成的标记在解析阶段进行处理，如果发现满足条件的一组标记则生成 AST 结点。例如，如果发现标记 `tok::kw_if`，那么调用函数 `ParseIfStatement`，对 `if` 语句中的所有标记进行处理，生成以 `IfStmt` 为根结点的所有子 AST 结点。

仍以 `min.c` 文件为例，通过 `-ast-dump` 可以获得 AST 节点，如图 6-11 所示。

```
TranslationUnitDecl ...
|-TypedefDecl ... __int128_t '__int128'
|-TypedefDecl ... __uint128_t 'unsigned __int128'
|-TypedefDecl ... __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl ... <min.c:1:1,line:5:1> min'int(int, int)'
|-ParmVarDecl ... <line:1:7,col:11> a 'int'
|-ParmVarDecl ... <col:14,col:18> b 'int'
`-CompoundStmt ... <col:21, line:5:1>
...
```

图 6-11 代码示例类继承关系

#### 6.4.4 语义分析

语义分析可保证代码不违反编程语言的类型系统。在符号表中存储着标识符同其对应类型的映射。类型检查的直觉方法是在解析工作完成后，对 AST 进行遍历，同时从符号表中收集到关于类型的信息进行类型检查。

但是，Clang 并不在解析后对 AST 进行遍历。而是在 AST 结点生成的过程中进行类型检查。仍以 `min.c` 文件为例。`ParseIfStatement` 函数调用语义动作 `ActOnIfStmt` 对 `if` 语句进行语义检查，并作出相应诊断。

为了辅助语义分析，基类 `DeclContext` 包含从第一个 `Decl` 节点到最后一个节点的引用，这大大方便了语义分析进行名字引用的符号查找并检查符号类型以及符号是否实际存在。语义分析引擎通过分析 AST 节点查找符号声明。

#### 6.4.5 LLVM IR 代码生成

语义分析之后，函数 `ParseAST` 调用方法 `HandleTranslationUnit`。如果编译器驱动使用前端动作 `CodeGenAction`，后续函数是 `BackendConsumer`，该函数通过遍历 AST 生成 LLVM IR。其中 LLVM IR 实现在 AST 中表示的相同行为。

### 6.5 LLVM 的中间表示

LLVM IR 是连接 LLVM 前端与 LLVM 后端的主干，保证 LLVM 实现对多种不同源语



言的解析，以及生成针对不同目标架构的代码。前端生成 LLVM IR，后端主要对 LLVM IR 进行操作。IR 也是大多数 LLVM 中与目标架构无关的优化所实施的地方。

编译器中间表示的选择是一个非常重要的问题。一方面，高级中间表示能够帮助优化器提取源代码信息；另一方面，低级中间表示能够帮助编译器生成特定目标架构的机器码。关于目标架构的所能获取的信息越多，可能利用的目标架构特性也越多。而且，低级中间表示必须小心处理，因为编译器将代码变换为接近于机器指令的中间表示，那么找到代码版段同其原始代码之间的对应关系将变得更加困难。而且，如果编译器设计与特定目标架构的表示联系非常紧密，那么生成的代码将难以应用于其他目标架构。

设计上的不同考虑导致了编译器的不同实现。一些编译器不支持多种目标架构代码的生成，仅仅关注于一种目标机器架构，这种编译器可以使用专用的中间表示，从而使得编译器更加高效，如 Intel 公司的 C++ 编译器 (icc)。但是，仅为单一架构生成代码的编译器如果试图支持其他目标架构，那么代价巨大。而为每种目标架构单独开发一种编译器又不现实，因此最好是实现某种编译器适用于各种不同的目标架构，这也是 GCC 以及 LLVM 的设计目标。

能够支持不同目标架构的编译器都可以称为可重定向编译器，这类编译器的开发需要面对基于不同目标架构的代码生成所提出的巨大挑战。降低可重定向编译器开发难度的关键在于使用通用中间表示，即便源代码翻译为不同架构下的指令，但是不同后端对代码的理解是相同的。通过通用中间表示，可以在不同的后端共享与目标架构无关的优化，但是如何设计通用的中间表示是一个问题。

LLVM 在设计之初的中间表示处于比 Java 的字节码 (bytecode) 更靠近底层的层面，这也是“底层虚拟机”缩写的由来。其思路是寻找低级优化的时机，进行链接时的优化。链接时优化通过将中间表示写入磁盘来实现，类似于 Java 中的字节码。字节码使得用户可以将多个模块融合在一个文件中，然后再进行过程间优化。通过这种方式，优化作用于多个编译单元，就好像在同一个模块中。

当前，LLVM 既不是 Java 的竞争者也不是虚拟机，所以 LLVM 采用其他中间表示形式来提高效率。例如，除了与目标架构无关优化操作的公共中间表示 LLVM IR，当代码以类 MachineFunction 和类 MachineInstr 表示时，后端还可以应用目标架构相关的优化。类 MachineFunction 和类 MachineInstr 表示使用了目标架构的指令的代码。

另一方面，类 Function 和类 Instruction 表示多个目标架构共享通用的中间表示。这种中间表示是目标架构无关的，而且也是官方的 LLVM 中间表示。为了避免混淆，LLVM 也从其他层面上表示代码，这使得它们也是中间表示，虽然并不将其称为 LLVM IR。

LLVM IR 具有 3 种等价形式：

- 内存表示 (类 Instruction)。
- 注重空间效率的磁盘表示 (bitcode 文件)。
- 便于人工阅读的磁盘表示 (LLVM 汇编文件)。



LLVM 提供了工具以及库文件对所有形式的中间表示进行操作。因此，这些工具可以将 LLVM IR 的磁盘表示转换为内存表示，也可以将其内存表示转换为磁盘表示：llvm-as 命令将文本 .ll 文件汇编为包含 bitcode 的 .bc 文件，llvm-dis 命令将 .bc 文件转化为 .ll 文件。如图 6-12 所示。

6.5.1 LLVM IR 语法

LLVM 设计中的最重要的方面是 LLVM 的中间表示 (IR)，中间表示是代码在编译器中的一种表示形式。LLVM IR 被设计用来在编译过程的优化阶段进行中级分析和转换。LLVM IR 设计时需要考虑许多特定目标架构，包括对轻量级运行时优化的支持、跨函数 / 过程间优化、对程序整体的分析以及积极的重构变换等。图 6-13 是一个 .ll 文件的示例。

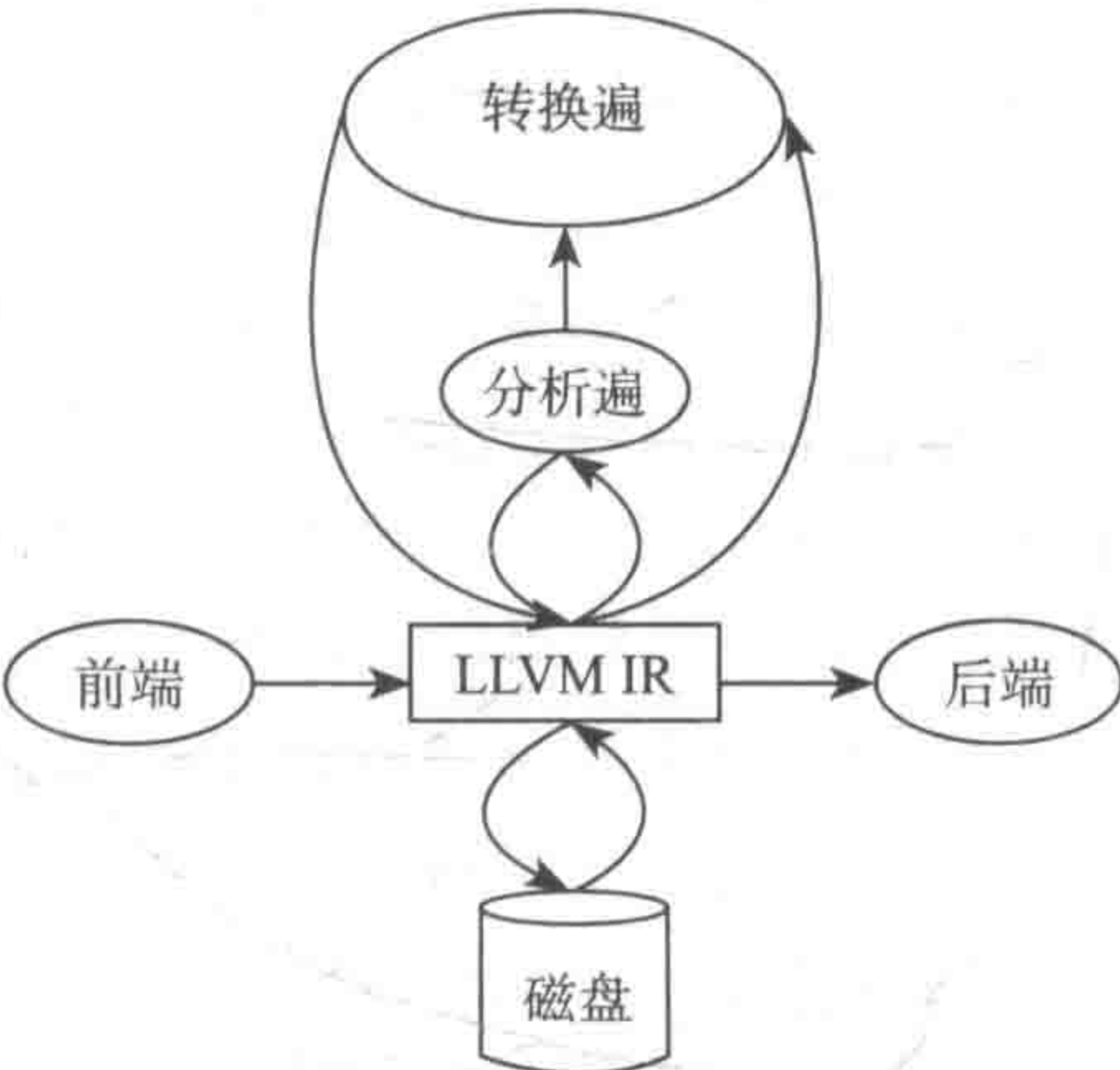


图 6-12 LLVM IR 转化关系

```
define i32 @add1(i32 %a,i32 %b) {
entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}

define i32 @add2(i32 %a,i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done,label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2,i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

图 6-13 LLVM IR 示例

从图 6-13 可以看出，LLVM IR 是一种低级的类 RISC 的虚拟指令集。但是，同大多数 RISC 指令集不同，LLVM 采用了强类型的类型系统（例如，i32 是 32 位整数，i32\*\* 是指向 32 位整数指针的指针），目标架构的部分细节被抽象忽略。例如，函数调用通过 call 与 ret 指令以及显式参数进行了抽象。



LLVM 中局部变量采用类似于汇编语言中寄存器的命名方式，以 “%” 符号开始。因此，“%tmp1 = add i32 %a,%b” 的含义是局部变量 %a 与 %b 相加，然后将结果返回到新的局部变量 % tmp1 中。用户可以对局部变量随意命名，或者使用数字编号命名局部变量。从示例中可以看出 LLVM IR 具有以下特点。

LLVM 使用静态单赋值，因此每个变量仅赋值一次，不会出现对变量的重新赋值。所以，每个变量的使用可以立即回溯到对其进行赋值的唯一指令，这对于简化优化过程具有重大作用，因为 SSA 所创建的 use-def 链，就是能够达到变量使用位置的赋值传导链表。如果 LLVM 未使用 SSA 形式，则需要使用单独的数据流分析获取 use-def 链，这对于经典优化过程，如常量传播、公共子表达式消除是必需的。

代码组织为三地址指令形式。数据处理指令有两个源操作数，并将结果放在另一个目的操作数中。

拥有无限的寄存器。需要注意的是 LLVM 局部变量可以是以 “%” 开头的任何名字，包括从 0 开始的数值，如 %0、%1 等，对变量的最大数量没有限制。

函数声明采用类似于 C 语言的语法形式，例如

```
define i32 @add1(i32 %a, i32 %b)
```

该函数表示返回类型为 i32 的变量，有两个 i32 参数，即 %a、%b。局部标识符通常需要 “%” 前缀，而全局标识符则使用 “@”。LLVM 支持多种数据类型，主要包括：

任意尺寸的整数，形式为 iN，如 i32、i64 以及 i128。

浮点类型，如 float 表示 32 位单精度类型，double 表示 64 位双精度类型。

向量类型，形式为 <<# elements> × <elementtype>>。例如，4 个 i32 类型元素的向量为 <4 × i32>。

函数主体进一步可以分解为基本块，并可以使用标记创建新的基本块。与基本块有关的标记同指令中变量标识符的标记方式相同。如果忽略标记声明，那么 LLVM 汇编器在命名空间中自动生成标记声明。基本块是一组指令序列，以第一条指令为单独入口点，最后一条指令为单独出口点。当代码跳转到某标记对应的基本块之后，将会执行该基本块中的所有指令。

在 LLVM 文件中，函数 add1 只有一个基本块，因为没有跳转、循环、调用等指令。entry 标记函数的开始，以返回指令 ret 标记函数的结束。而函数 add2 则有两个基本块。

图 6-13 中所示 LLVM IR 对应于图 6-14 中的 C 代码，该段代码的功能是提供了两种不同的整数相加方式。

编译器的中间表示作用巨大，是优化器工作的理想场所。一方面，同编译器的前端和后端不同，优化器不受特定源语言或目标架构的限制。另一方面，中间表示需要同前端和后端良好衔接，前端生成中间表示的过程复杂度不能太高，而对于目标架构而言又需要具备足够的表达能力，使得能够开展重要的优化过程。



```

unsigned add1(unsigned a, unsigned b) {
    return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}

```

图 6-14 图 6-13 示例对应的 C 代码

### 6.5.2 LLVM IR 优化实例

转换到 LLVM IR 后，可以对代码进行各种与目标架构无关的代码优化。例如，优化可以每次仅仅作用于某一函数或者某一模块。对模块进行优化，当优化是过程间优化时，为了增强过程间优化的影响，用户可以使用 `llvm -link` 将多个 LLVM 模块链接为一个模块。这使得优化的作用范围更大。有时可以称之为链接时优化，因为优化可能已经超越了转换单元的边界。LLVM 可以使用 `opt` 工具访问和获取所有优化，并可以单独调用某种优化。

`opt` 工具使用 Clang 编译器驱动中的优化标记，即 O0、O1、O2、O3、Os 和 Oz。Clang 支持 O4 级优化，但是 `opt` 不支持。O4 可以看作 O3 的增强，O4 同时还进行链接时优化，但是 LLVM 中的链接时优化依赖于如何组织输入文件，每个标记都会调用不同的优化流水线，涉及特定顺序的优化类型。

O0 级优化即“无优化”，该级优化的编译速度最快并能够生成适合调试的代码。O2 是中级优化，该级优化实现了大多数优化。Os 类似于 O2 级优化，区别在于 Os 在优化过程中对代码规模进行削减。Oz 级优化类似于 Os，区别在于其对代码规模的削减幅度更大。O3 级优化类似于 O2 级优化，区别在于 O3 级能够进行编译时间消耗更长或者生成代码规模更大的优化。在所支持的平台上，O4 级优化进行链接时优化，对象文件以 LLVM 的 `bitcode` 文件格式存储，所有代码的优化在链接时完成。O1 的优化程度介于 O0 与 O2 之间。

为了直观地介绍优化工作的开展，下面结合实例进行介绍。编译器的优化有很多不同类型，因此对于某一问题很难找到一个统一的答案。也就是说，大多数优化过程主要遵循以下三个步骤：

- 寻找需要进行变换的模式。
- 验证变换对于与之匹配的实例是安全的 / 正确的。
- 进行变换，更新代码。

最简单的优化是对算术运算标识进行模式匹配。例如，对于任意整数  $x$ ， $x-x$  的结果是 0， $x-0$  的结果是  $x$ ， $(x \times 2) - x$  的结果是  $x$ 。那么需要面对的第一个问题是这些计算结果在 LLVM IR 中的表示形式。以图 6-15 中所示代码为例。



对于这种类型的窥孔优化变换, LLVM 提供了一种简化接口, 高级变换都可以利用这些接口予以完成。这些变换在函数 `SimplifySubInst` 中完成, 代码形式如图 6-16 所示。

在如图 6-16 所示代码中, `Op0` 和 `Op1` 是整数减法指令的左右操作数 (重要的是这些参数表示无需遵循 IEEE 浮点存储格式)。LLVM 采用 C++ 编程实现, 而 C++ 的模式匹配能力并不强大 (相对于函数式编程语言, 如 Objective Caml), 但是 C++ 提供一个较为通用的模板系统以实现类似功能。`match` 函数和 `m_` 函数能够对 LLVM IR 代码进行声明式的模式匹配操作。例如, `m_Specific` 断言只有在乘法运算的左值同 `Op1` 相同时才成立。

```

: : :
%example1 = sub i32 %a, %a
: : :
%example2 = sub i32 %b, 0
: : :
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
: : :

```

图 6-15 LLVM IR 示例

```

// x - 0 -> x
if (match(Op1, m_Zero()))
    return Op0;

// x - x -> 0
if (Op0 == Op1)
    return Constant::getNullValue(Op0-&gtgetType());

// (x*2) - x -> x
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
    return Op1;

...

return 0; // Nothing matched, return null to indicate no transformation.

```

图 6-16 LLVM IR 示例优化

这三种例子都是模型匹配, 如果匹配成功那么函数返回匹配结果, 如果匹配不成功则返回 `null` 指针。该函数 (`SimplifyInstruction`) 调用者实质上是一个分发器, 根据指令 `opcode` 进行转换、分发至每个 `opcode` 的相关函数。该函数在不同类型的优化中进行调用。简单的驱动代码如图 6-17 所示。

```

for(BasicBlock::iterator I = BB->begin(), E = BB->end(); I!=E;++I)
    if(Value*V = SimplifyInstruction(I))
        I->replaceAllUsesWith(V);

```

图 6-17 驱动代码

图 6-17 中的代码仅仅遍历基本块中每个指令, 检查每条指令是否能够进行简化。如果存在满足条件的指令的话 (`SimplifyInstruction` 返回非空指针), 使用 `replaceAllUsesWith` 函数对代码进行简化操作。



## 6.6 LLVM 后端

LLVM 后端即代码生成器负责将 LLVM IR 变换为特定目标架构的机器代码。一方面，后端任务就是为某一目标架构生成最优的机器代码。在理想情况下，每个后端都应该针对目标架构生成与之完全对应的代码。但另一方面，每个目标架构的代码生成器又都需要解决类似的问题。例如，每个目标架构都需要对寄存器赋值，虽然各个目标架构可能使用不同的寄存器文件，但其所使用的算法应该是可以通用的。

与优化器所使用的方法相似，LLVM 的后端将代码生成问题划分为各自独立的遍过程，包括指令选择、寄存器分配、调度、代码布局优化和组装执行发射等，后端中还有内建默认执行的遍。目标架构的开发人员能够选择缺省遍，或者对缺省遍进行重写定制同目标架构相关的遍。举个例子，由于 x86 只有很少的寄存器，所以 x86 后端采用降低寄存器压力的调度方式。但是，PowerPC 架构中寄存器数量较多，所以 PowerPC 后端采用延迟优化调度。x86 后端使用定制遍处理 x87 浮点指针堆栈，ARM 后端也使用定制遍在函数中需要的位置部署常量池。这种灵活性使得目标架构的开发人员无需完全重写代码生成器即可生成满意的代码。

图 6-18 展示了从 LLVM IR 到目标代码或者汇编指令的所需步骤，其中白色方框可以进行优化扫描进一步改进翻译质量。

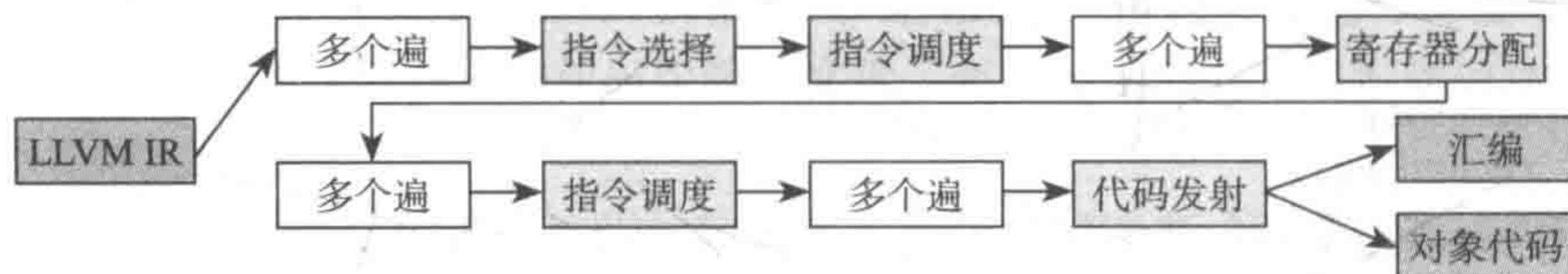


图 6-18 后端处理流程

后端所涉及的不同阶段以浅灰色方框表示，也被称为 *superpass*，因为它们通过若干个规模更小的遍进行实现。浅灰色方框和白色方框之间的区别在于，通常浅灰色方框表示一组对于后端成败非常关键的遍，而白色方框则对于提高生成代码的效率非常重要。

**指令选择：**指令选择阶段将内存中的 IR 转换为目标架构相关的 SelectionDAG 节点。起初，该阶段将 LLVM IR 的三地址结构转换为有向无环图（DAG）。每个有向无环图都能够表示单个基本块中的运算，这意味着每个基本块对应于不同的有向无环图。使用 DAG 的转换对于 LLVM 代码生成器库文件使用基于树的模式匹配指令选择算法非常重要。在该阶段的末尾，DAG 将其所有的 LLVM IR 节点转换为目标架构节点，即该节点表示机器指令而不是 LLVM 指令。

**指令调度（寄存器分配前调度）：**指令选择之后，可以知道哪条目标架构指令可以用于执行基本块中的运算。这一过程在类 SelectionDAG 中实现。但是，需要返回到三地址表示以确定基本块中指令的执行顺序，因为 DAG 并不表示指令间相互依赖的执行顺序。此阶段的指令调度又称为寄存器分配前调度，其目的主要是挖掘指令级的并行性，进而确定指令



的执行顺序。然后指令被转换为类 `MachineInstr` 表示的三地址表示。

寄存器分配：LLVM IR 中具有数量无限的寄存器，寄存器分配阶段将无限的虚拟寄存器引用集合转换到有限的目标架构相关寄存器集合。

指令调度（寄存器分配后调度）：因为此时能够获取真实的寄存器信息，因此结合部分类型寄存器的安全性考虑与执行效率，可以对指令执行顺序进行进一步调整。

代码发射：代码发射阶段将指令从类 `MachineInstr` 表示的形式转换为类 `MCInst` 的实例。新的表示形式更加适合汇编器以及链接器，同时可以对格式进行选择，包括发射汇编代码或者发射二进制代码块到特定的对象代码形式。

通过上面的描述，在后端处理流程中主要涉及 4 种不同的指令表示：内存中的 LLVM IR、SelectionDAG 节点、类 `MachineInstr` 以及类 `MCInst`。

### 6.6.1 后端库文件

`llc` 非共享代码的代码量较少，而且其中的大部分功能已作为复用库文件实现，同 LLVM 中的其他工具类似。在 `llc` 中，其功能由代码生成器库文件提供。该组库文件由目标架构相关的部分以及目标架构无关的部分组成，用户可以同所需目标架构后端相链接。例如，在 LLVM 配置文件中使用 `--enable-targets=x86,arm` 命令，那么将只有 x86 和 ARM 后端库文件链接到 `llc`。

目标架构无关的代码生成器库文件主要包括：

`AsmParser.a`：该库文件包括汇编文本解析以及汇编器实现的代码。

`AsmPrinter.a`：该库文件包括汇编语言打印，以及生成汇编文件的后端实现代码。

`CodeGen.a`：该库文件包括代码生成算法。

`MC.a`：该库文件包括类 `MCInst` 以及与其相关的类，用来在 LLVM 的最底层表示代码。

`MCDisassembler.a`：该库文件包括读取对象文件，并对 `MCInst` 对象解码的反汇编器实现。

`MCJIT.a`：该库文件包括 JIT 代码生成器的实现。

`MCParser.a`：该库文件包含类 `MCAsmParser` 的接口，可用于实现汇编文本解析组件，并承担汇编器的部分工作。

`SelectionDAG.a`：该库文件包括类 `SelectionDAG` 以及相关的类。

`Target.a`：该库文件包含目标架构无关算法使用目标架构相关功能的接口。

目标架构相关的库文件主要包括：

`<Target>AsmParser.a`：该库文件包含 `AsmParser` 库文件中同目标架构相关的部分，主要负责为目标架构生成汇编器。

`<Target>AsmPrinter.a`：该库文件包含打印目标架构指令的功能，保证后端生成汇编文件。

`<Target>CodeGen.a`：该库文件包含后端同目标架构相关的大部分功能，包括特定寄存



器处理规则、指令选择与调度等。

<Target>Desc.a：该库文件包含关于低级 MC 架构的目标架构相关的信息，并且能够注册目标相关的 MC 对象，如 MCCodeEmitter。

<Target>Disassembler.a：该库文件同 MCDisassembler 库文件相互补充，提供构建后段过程中同目标架构相关的功能，包括字节读取以及将其解码为 MCInst 目标指令等功能。

<Target>Info.a：该库文件对 LLVM 代码生成器中的目标指令进行注册，并提供类 façade，利用目标相关的代码生成库文件，从而实现与目标架构相关的功能。

## 6.6.2 LLVM 目标架构描述文件

混合匹配（mix and match）方法使得目标架构开发人员能够选择其体系结构所适用的选项，以及在不同目标架构中复用大量代码。这种方法的难点在于，每个共享组件都需要能够通过一种通用方式获取得到目标架构的特性。例如，寄存器分配器需要知道每种目标架构的寄存器文件，以及指令和寄存器操作之间的约束关系。LLVM 的解决方案是为每个目标架构提供一个目标架构描述文件，该文件采用声明性的域相关语言（.td 文件）编写。其思路是在唯一位置声明机器的特性，如在 <Target>InstrInfo.td 文件对机器指令进行描述，然后使用基于该文件的 TableGen 后端完成特定任务。当前，TableGen 可用于描述所有类型的目标架构相关的信息，如指令格式、指令、寄存器、模式匹配 DAG、指令选择匹配顺序、调用规范以及目标架构 CPU 特性等。.td 文件可以采用 tblgen 工具生成。对 x86 目标架构的构建流程如图 6-19 所示。

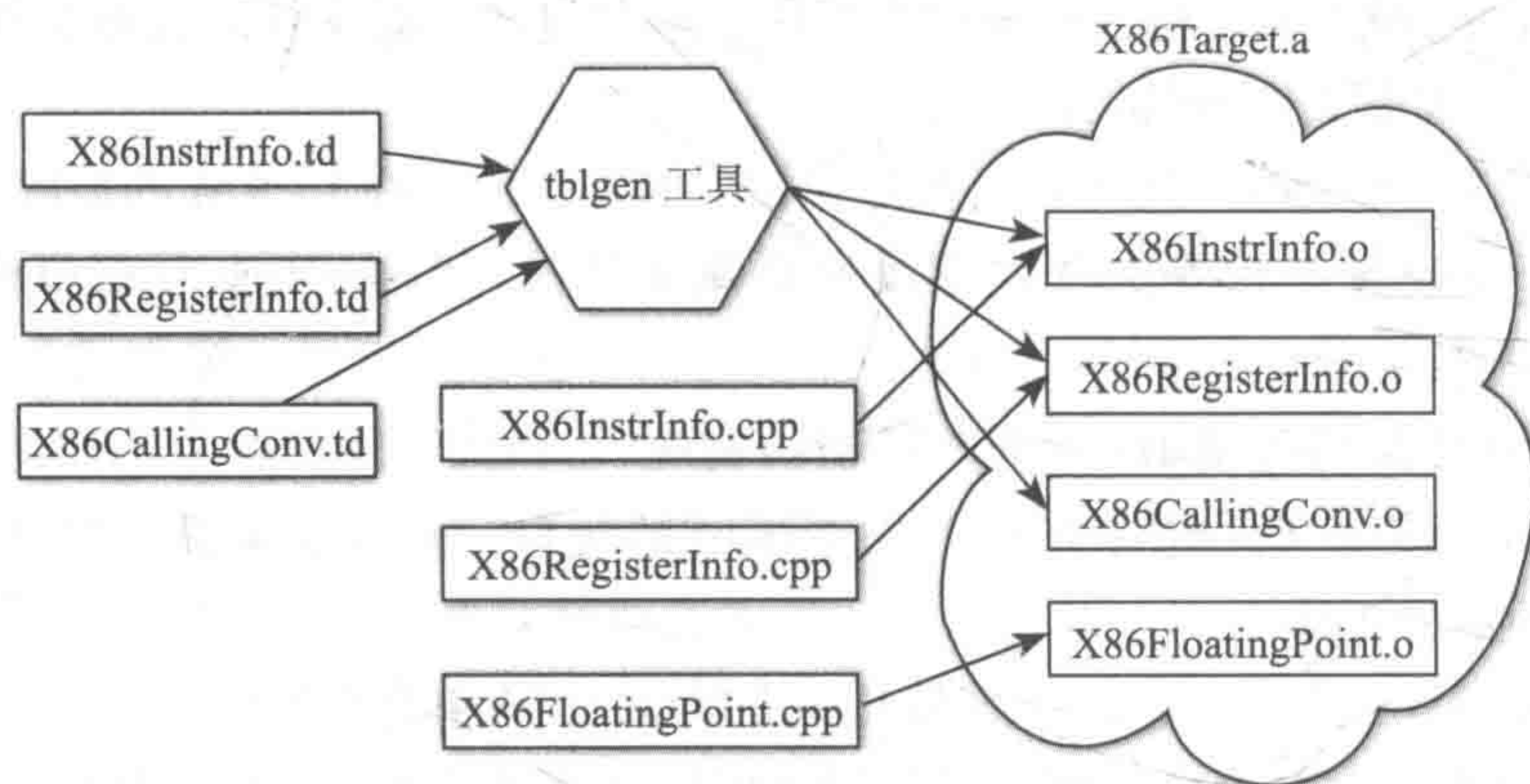


图 6-19 x86 目标架构定义

TableGen 语言由定义与类构成。定义 def 用于对关键字 class 以及 multiclass 中的记录初始化。这些记录被 TableGen 后端进一步处理，生成代码生成器的域相关信息。

.td 文件所支持的不同子系统使得目标架构开发人员能够对目标架构的不同部分进行描述。例如，x86 后端定义了一个名为“GR32”的寄存器类保存所有的 32 位寄存器，定义如图 6-20 所示。



```
def GR32 : RegisterClass<[i32], 32,
    [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
     R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
```

图 6-20 .td 文件寄存器定义示例

根据上述定义，该类寄存器可以存储 32 位的整数值（“i32”），并指定了 16 个寄存器（定义在 .td 文件中），同时利用更多信息指定分配顺序以及其他内容。根据上述定义，特定指令引用该寄存器，将其作为操作数。例如，32 位寄存器取反操作指令的定义如图 6-21 所示。

```
let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
    (outs GR32:$dst), (ins GR32:$src),
    "not{l}\t$dst",
    [(set GR32:$dst, (not GR32:$src))]>;
```

图 6-21 td 文件指令定义示例

上述定义表明 NOT32r 是一条指令，指示了编码信息（0xF7, MRM2r），定义输出为 32 位寄存器 \$dst，输入为 32 位寄存器 \$src

（GR32 寄存器类定义了哪些寄存器对该操作数有效），说明了指令的汇编语法（使用“{}”处理 AT&T 和 Intel 语法），还说明了指令的作用，并在最后一行说明了需要匹配的模式。第一行中的“let”约束告诉寄存器分配器，输入和输出寄存器应该被分配至相同的物理寄存器。

该定义对指令进行了详尽的描述，因此，通常利用定义中提供的信息（借助 tblgen 工具）LLVM 代码能够开展很多工作。这个定义能够帮助从编译器的输入 IR 代码中通过模式匹配进行指令选择。它也告诉寄存器分配器如何处理，如何通过对该指令进行编码和解码来得到机器码字节，如何以文本方式解析和打印指令。这些特性使得 x86 目标架构能够根据目标架构的描述生成一个独立的 x86 汇编器以及反汇编器。

除了提供有用的功能，从同一段信息中获取多种信息具有诸多优势。这种方法使得汇编器和反汇编器能够避免在汇编语法或者二进制编码方面出现不一致的情况。也使得目标架构描述便于测试，因为可以在无需牵涉整个代码生成器的情况下对指令编码进行单元测试。

虽然以声明性的方式可以从 .td 文件中获取很多目标架构信息，目标架构开发人员仍然可以通过 C++ 代码开发支持例程，实现所需要的目标架构特定遍（例如 X86FloatingPoint.cpp 文件，该文件主要对 x87 浮点数堆栈进行操作）。但是，随着 LLVM 所支持的目标架构越来越多，以 .td 文件的形式增加目标架构的数量变得越来越重要，而且需要不断提高增加 .td 文件的表达能力来解决该问题。

## 6.7 应用实例

LLVM 在实际工作中得到了广泛应用，基于 LLVM，很多研究人员进一步开发出了诸多应用，下面以两个例子来说明 LLVM 在现实中的应用。



### 6.7.1 代码插桩

动态代码插桩是构建程序分析调试工具的核心技术之一。当前大部分动态插桩工具关注于相同指令集架构的插桩，即宿主代码与插桩代码采用相同的指令集编写。但是不同指令集架构下的插桩却不常见，如将 ARM 可执行代码插入 x86 架构下运行的代码。

DBILL 是一种跨指令集架构的动态插桩框架，在框架的构建过程中用到了可重定向的动态二进制翻译工具 QEMU。图 6-22 是 DBILL 的架构。DBILL 框架的输入是一段二进制代码（包括可执行程序、共享库文件或者动态链接库等）。如图中左边路径所示，QEMU 的目标架构前端将该输入的二进制代码转换为 TCG（Tiny Code Generator）表示，然后 TCG 前端将 TCG 表示转换为 LLVM IR 表示形式。LLVM 遍管理器在 LLVM IR 中插入分析代码，然后由即时翻译器在运行时生成与插桩完成的 LLVM IR 相对应的对象代码。

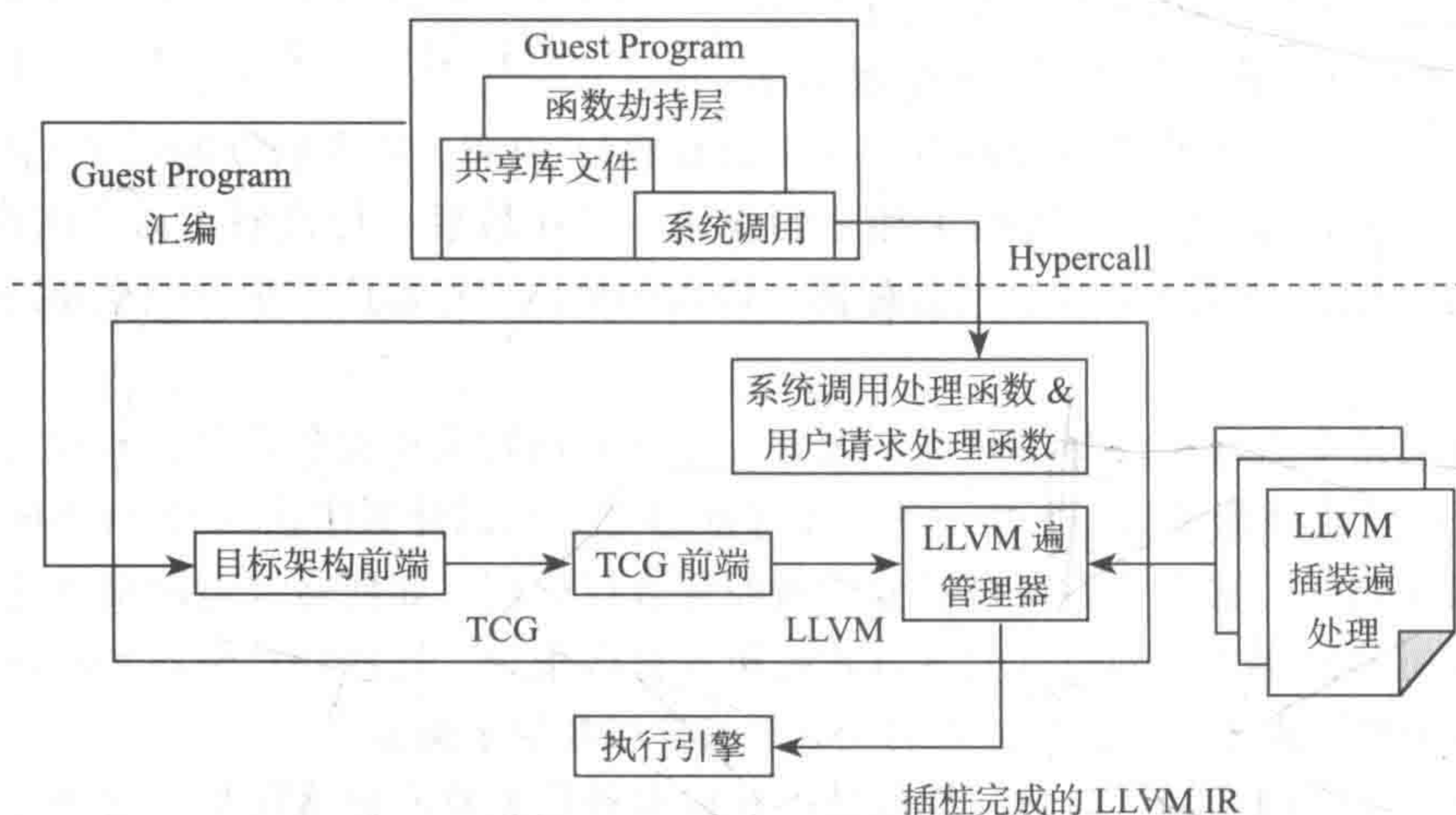


图 6-22 DBILL 框架架构示意图

### 6.7.2 代码保护

针对代码篡改、恶意嵌入以及逆向分析等问题，代码保护面临着诸多难题。其采用的主要保护方式包括密码机制与代码混淆。其中，代码混淆通过大幅提高逆向分析的代价阻止对代码的逆向分析。

Obfuscator-LLVM (ollvm) 是在 LLVM 优化器中实现的一组代码混淆变换，旨在提高代码抵抗逆向分析以及篡改的能力。ollvm 具有语言无关、平台无关的特性，所以可以同 LLVM 所支持的所有编程语言协同工作，包括 C、C++、Objective-C 等编程语言，以及 x86、x86-64、PowerPC、PowerPC-64、ARM、ARM-64、Sparc、Alpha 以及 MIPS 等架构。当前 ollvm 已经实现了指令替换、虚假控制流插入、基本块分割、控制流扁平化、过程合并以及抗篡改机制插入等内容，在其开发计划中还包括常量加密、垃圾代码插入以及抗调



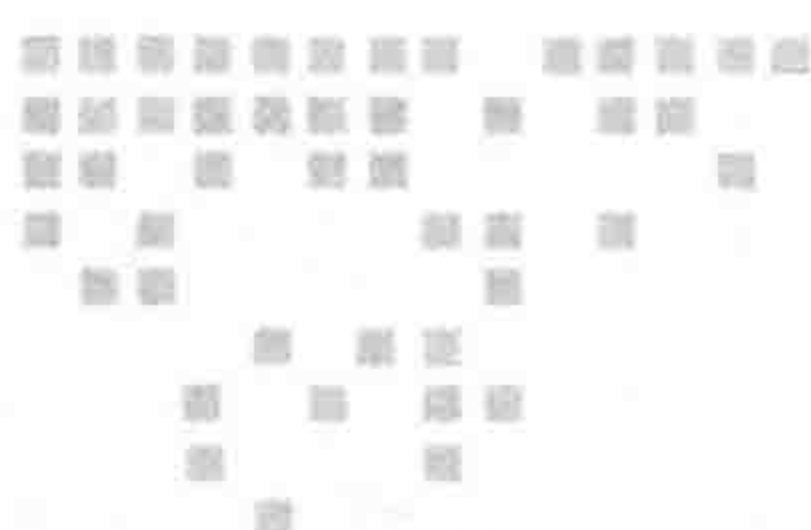
试代码插入。

在 LLVM 编译套件中，优化器主要负责无效代码或者冗余代码移除、函数内联、循环展开、死循环删除、控制流图简化等工作。由于 `ollvm` 实现为优化器中的遍（`pass`）处理，因此混淆以及抗篡改机制主要在 LLVM 的优化器中实现。这种方式无需知道代码的编程语言以及目标架构，大幅削减了实现过程中需要考虑的因素。但是该方式无法实现所有保护技术，如抗调试代码的插入应该在代码生成阶段，即 LLVM 后端完成，而目前 `ollvm` 所完成的操作主要在优化器中进行。

## 6.8 本章小结

每位程序员在不同阶段都会接触到编译器。简单来说，编译器的作用就是将高级语言转换为可执行的机器代码，但是这一过程涉及大量复杂的算法，LLVM 项目的出现使得对编译器的了解过程变得更加简单。采用面向对象的 C++ 语言编写、模块化的设计以及同编译理论直接的映射关系都使得 LLVM 成为了解编译过程的极佳工具。同时作为开源项目，众多开发人员参与到 LLVM 项目中，随着 LLVM 的不断成熟，其已从学术实验软件过渡到了商业产品中的健壮框架，并在诸多领域得到了广泛应用。





## Chapter 7 第 7 章

# 多样化编译实践

软件安全保障方面的不足，不只表现在方法和技术方面的欠缺，更表现在地位上的被动。具体而言表现为，当漏洞被发现后，开发者才开始修复这个漏洞和由此带来的其他后果，但这时可能已经造成了不可挽救的损失和后果。这种局面对攻击者而言显然是非常有利的，因为他们仅仅需要找到一个漏洞，就可以充分利用所有合法漏洞的软件，而对防护者而言，却不得不阻止所有漏洞的利用。编译器的改进被认为是这个问题相关解决方案的核心：当编译器转换高级源代码到低级的机器代码的时候，它能自动地多样化机器代码，生成同一程序的多个功能等价的不同的变体。使用多变体执行，一个监控器同时执行多个变体，同时检查它们行为的差异来预示攻击。使用软件的多样性，每个用户可得到自己的变体，而攻击者并不知道这个变体的内部结构细节，从而不易实施有效攻击。这两种技术使得攻击者进行一次成功的攻击变得更加困难。在本章，我们讨论多样化编译器的方法和实现技术，揭示在多样化编译方面的实践技巧。

## 7.1 软件多样化的机会

对 Web 服务而言，通过引入多样化能够提高 Web 服务对入侵的抵抗能力。在软件栈的不同层次都存在着大量多样化的机会，本节将重点阐述在软件栈的应用层、Web 服务器层、操作系统层、虚拟层应用多样化的机会。

### 7.1.1 应用层的多样化机会

目前有许多基于 Java、PHP、Perl、Python 以及 Ruby 等编程语言的编程平台。作为 .NET



架构的一部分，微软也提供了一套平台，并称为 WCF 数据服务（Web 服务是该框架的一部分）。

虽然一种编程语言往往有多个版本，但是在 Web 应用程序开发中同时采用多种版本的情况却较少。在应用层实现多样化的一种可行的方法包括采用不同版本的函数库、解释器等，然而编程语言的不同版本通常都是同特定操作系统相绑定的。例如，红帽子公司的 Enterprise Linux 5 所支持的 Ruby 解释器版本为 1.8.5，而 Ubuntu 8.04 所支持的 Ruby 解释器版本为 1.8.6。因此，可将编程语言的版本问题看作操作系统多样化的一部分，稍后再讨论。

以 Rails 为例，Ruby 解释器有两种实现（不仅仅是版本不同）：由 Ruby 开发者 Yukihiro Matsumoto（网络上昵称“Matz”）实现的官方版本和基于 Java 实现的版本 JRuby。Ruby 的两种不同实现构成了 Ruby on Rails 平台的多样化的一种方式（其实还有其他实现方式，但没有上述两种实现方式高级，因此在此予以忽略）。

需要指出的是，一个语言具有多种实现方式并不罕见。例如，Java 代码可以在多个不同的 Java 虚拟机（JVM）上执行，谷歌的 Dalvik 虚拟机就是其中一种。由于 Dalvik 使用的字节码不同于标准 Java 字节码，所以 Dalvik 的使用在应用层上也引入了指令集随机化。

需要强调的是，虽然采用不同的解释器不会消除应用程序的漏洞，但确实能够改变解释器的攻击面（通俗地讲，系统的攻击面是其被用于攻击的资源（以某种权限运行的方法、通道和数据等）的子集，攻击者可以利用这些资源来攻击系统）。反过来，多样化应用程序也并不一定会修复逻辑上的设计缺陷或者保护其他服务器组件免遭漏洞攻击。

### 7.1.2 Web 服务层的多样化机会

表 7-1 显示了 5 个主要的 HTTP 服务器和其所需的平台。在该表中，“开源”表示与多种主流操作系统之间的兼容性，比如 Windows、Linux、BSD 和 Solaris。除了微软的 IIS，其他的服务器软件要么是兼容的，要么可以移植到许多操作系统之上。

表 7-1 Web 服务器软件和操作系统的兼容性

Web 服务器软件	操作系统兼容性	Lighttpd	开源
Apache	开源	IIS	Windows 平台
Nginx	开源	Tomcat	开源

事实上，多样化可以更多地来自于对运行 Web 应用程序的特定 Web 服务器软件的不同配置。根据 Web 编程语言和平台的不同，Web 服务配置也会随之变化。以 Ruby on Rails 平台为例，在此强调不同的服务配置并不需要更改应用程序代码。运行 Rails 应用程序的已知部署配置列表如下：

- 1) Apache + mod fastcgi
- 2) Apache + mod rails
- 3) Apache + Mongrel cluster



- 4) Apache + Thin cluster
- 5) Nginx + mod rails
- 6) Nginx + Mongrel cluster
- 7) Nginx + Thin cluster
- 8) Lighthttpd + mod fastcgi
- 9) Lighthttpd + Mongrel cluster
- 10) Lighthppd + Thin cluster
- 11) IIS + Fast-CGI
- 12) Tomcat + JRuby

首先解释在 Apache 下运行 Rails 应用程序的各种配置, 以及不同配置对增加攻击面多样性的帮助。

第 1 种配置 Apache + mod fastcgi 是传统的配置方法。应用程序的 Ruby 代码可以用作 CGI 代码, mod fastcgi 插件用于需要时调用特定的 Ruby 代码。在这种情况下, Apache 需要处理 URL 输入, 因此它的漏洞很可能被利用。此外, mod fastcgi 是一种通用 CGI 接口, 能够支持许多其他的 CGI 语言, 如 Python、PHP、Perl 等, 该插件的漏洞也可能被利用。

第 2 种配置 Apache + mod rails, 使用自定义的 CGI 接口, 该接口只支持 Rails。这种配置同上一个配置都可能遭受利用 Apache 漏洞的攻击。然而, CGI 接口——mod rails 是专门针对单一平台的, 同通用的 mod fastcgi 相比漏洞较少。两种配置拥有不同的漏洞集合, 可以被用作不同攻击面的组件。

第 3 种配置 Apache + Mongrel cluster, 该配置只允许 Apache 转发 URL 请求到 Mongrel 服务器集群的多路复用器。Mongrel 是只能运行 Rails 应用程序的单线程服务器。它的单线程方法和专用的 Web 编程平台大大简化了设计和实现过程。然而, 由此导致可能需要多个 Mongrel 进程来处理并发请求, 因此将其称作“Mongrel 集群”。利用这种配置, 除了将请求转发给 Mongrel 进程, Apache 不涉及请求处理的任何方面。因此, 触发 Apache 漏洞的可能性被降到了最低。此外, Mongrel 的简洁性使得可能的漏洞也相对较少。这种配置的关键在于, 同上述两种基于插件的配置相比, 这种配置具有完全不同的漏洞。

第 4 种配置 Apache + Thin cluster, 这种配置同使用 Mongrel cluster 的配置较为类似, 但该配置使用了另一种尺寸较小的 RailWeb 服务器。第 4 种配置的攻击面也具有常见特征, 但同 Mongrel 集群相比具有不同的实现细节。

总的来讲, 上述 4 种运行 Rails 应用程序都是基于 Apache 服务器的, 但是配置不同也会形成不同的攻击面。

第 5 ~ 11 种 Web 服务配置的解释同上述描述类似, 不再赘述。而第 12 种配置 (Tomcat + JRuby) 与其他配置相比有较大区别。它用到了前面提到的另一种基于 Java 实现的 Ruby 解释器 JRuby。Tomcat Web 服务器是基于 Java Servlet 和 JavaServer Pages 技术实现的开源服务器。因此相较于其他配置, 该服务配置是基于 Java 的, 从而可用来创建



完全不同的攻击面。需要指出的是，由 Apache 软件基金会研发的 Tomcat 被称为 Apache Tomcat。但是，Apache 和 Apache Tomcat 是不同的 Web 服务器解决方案。

除了“IIS + Fast-CGI”配置，其他 11 种配置都是基于开源软件的，均被主流操作系统所支持，或可移植到其他主流操作系统之上。当前已知的例外情况都会在后面列出。因为不同的 Web 服务配置会生成不同的攻击面，即便部分配置中可能使用相同的 Web 服务器软件，但在 Web 服务器层面上会有 12 种多样化方式，而不是仅有 5 种 Web 服务器程序。

### 7.1.3 操作系统层的多样化机会

Web 服务器所基于的主要操作系统平台包括 Windows、BSD、Solaris 系统的衍生版本和不同的 Linux 发行版本。由于可用的操作系统数量很多，本节将使用下列规则。首先，操作系统必须得到维护，具体而言，就是必须有一个公司或社区负责及时修补该操作系统的漏洞。其次，在版本号或次版本号不同的操作系统版本没有显著增加多样性的假设下，不对其版本号或次版本号加以区分。基于该假设，对于 Windows 7 的家庭版和专业版，仅以 Windows 7 表示。与之类似，Ubuntu 的 8.04 和 8.10 都标记为 Ubuntu 8。

另外，也需要区分操作系统的 32 位版本和 64 位版本。这是因为诸如面向返回的编程 (Return-Oriented Programming, ROP) 攻击等高级且危险程度较高的攻击能否成功取决于精确的地址长度。而且，32 位和 64 位指令格式之间的不同也可以看作一种指令集随机化。但因为表 7-2 中的所有操作系统都支持 32 位和 64 位两种版本的操作系统，所以没有分别列出。

表 7-2 主流 Web 服务器操作系统 (乘以 2 是因为操作系统既有 32 位版本也有 64 位版本，假定多样化编译提供了  $n$  种机会)

平台	版本	初始变体数量	多样化编译后变体数量
Windows	Vistanbul/7/8; Windows Server 2003/2008	$5 \times 2 = 10$	$10n$
BSD	FreeBSD (7/8), NetBSD (4/5), OpenBSD 4	$5 \times 2 = 10$	$10n$
Solaris	Oracle Solaris 10/11	$2 \times 2 = 4$	$4n$
RedHat	RedHat Enterprise Linux 4/5/6	$3 \times 2 = 6$	$6n$
Ubuntu	Ubuntu Server Editions 6/8/9/10	$4 \times 2 = 8$	$8n$
SUSE	SUSE Linux Enterprise, OpenSUSE	$2 \times 2 = 4$	$4n$

Windows XP 未再列出，因为微软已经宣布停止了对它的维护。从 Windows 平台开始，我们既列出客户端的操作系统也列出服务器端的，因为客户端和服务端都支持 IIS 并且与开源 Web 服务器软件兼容。在 Solaris 平台上，列出了 Oracle 官方支持的两个版本。表 7-2 省略了 OpenSolaris 操作系统，因为 Oracle 公司收购 Sun 公司之后，该系统前景尚不明朗。对于 RedHat Enterprise Linux 系列操作系统，4 版本之前的版本未再列出，因为已经对之前的版本停止了维护。其 Fedora 版本虽然在一些 Linux 爱好者中广为流行，但是由于该系列生命周期通常较短，较不稳定，所以将其从列表中略去。表 7-2 还列出了 4 种 Ubuntu 版



本, Ubuntu 7 不在列表之中的原因是, 只有偶数版本号的 Ubuntu 服务器才能享有长期 (5 年) 的维护, 而奇数版本号的 Ubuntu 服务器只能获得 1 年半左右的支持。需要强调的是, 表 7-2 中的操作系统是在本节撰写时所挑选出的, 满足选择要求的候选操作系统必然会随着时间而改变。综上所述, 表 7-2 中已经确定了 42 个不同的操作系统, 这些操作系统稳定并且能够得到维护, 在提高多样性方面能够发挥较大作用。

#### 7.1.4 组合后的多样化机会

表 7-3 显示了 7.1.2 节中介绍的 Web 服务器配置和表 7-2 中操作系统组合后的多样化机会, 得到了在不同的 Web 服务器配置下的主流操作系统上运行 Rails Web 应用程序的 452 种组合方式。而且, 还尚未考虑采用不同虚拟化解决方案的影响。

表 7-3 Web 服务配置与操作系统组合后的多样化机会 (假定多样化编译提供了  $n$  种机会)

平台	Web 服务配置 (共 12 种)	初始变体数量	多样化编译后变体数量
Windows	除了 7.1.2 节介绍的第 2 和第 5 (mod rails 在 Windows 上不工作) 种配置之外的 10 种	$10 \times 10 = 100$	$100n$
BSD	除第 11 种配置 (即 IIS) 之外的 11 种	$10 \times 11 = 110$	$110n$
Solaris	除第 11 种配置 (即 IIS) 之外的 11 种	$4 \times 11 = 44$	$44n$
RedHat	除第 11 种配置 (即 IIS) 之外的 11 种	$6 \times 11 = 66$	$66n$
Ubuntu	除第 11 种配置 (即 IIS) 之外的 11 种	$8 \times 11 = 88$	$88n$
SUSE	除第 11 种配置 (即 IIS) 之外的 11 种	$4 \times 11 = 44$	$44n$

#### 7.1.5 虚拟层的多样化机会

对虚拟层的攻击在现实中也已经出现了。但是, 这常常需要在操作系统上运行的代码, 而且虚拟层需要存在未知的或者未打补丁的漏洞。相比其他层, 虚拟层漏洞出现的频率相对较少。虽然针对虚拟层漏洞的可能性较低, 但是如果虚拟层存在漏洞, 则所造成的影响或导致的后果会非常严重: 因为对虚拟层的攻击可以蔓延到主机上的所有虚拟服务器 (简称 VS)。这就对虚拟层自身软件栈的多样化提出了较高的要求。

如前所述, 我们这里只考虑那些用于生产环境的虚拟化技术。这就排除了包括 QEMU 在内的虚拟机。一般而言, 虚拟化技术分为两种类型: 基于管理程序的虚拟化技术和 OS 层虚拟化技术。前者如 VMware 系列产品和 Xen, 这种虚拟化技术由于其代码体积小以及实现了同 VS 的高层次隔离, 通常认为具有更高的安全性。后者如 Linux 的 OpenVZ 和 Solaris Zones, 由于同主机内核的紧密结合, 因此具有较低的运行开销。然而, 同主机内核的紧密结合提高了已经攻陷某台 VS 的攻击者通过内核漏洞实现虚拟服务器“逃逸”的可能性。但是, 该问题可以通过应用 SELinux 模块有所缓解, 如 sVirt 模块, 该模块采用强制访问控制 (MAC) 策略实现了虚拟机和容器之间的隔离。最后, 相比于基于管理程序的操作系统, 实现轻量级虚拟化的客户操作系统数量更加有限。例如, OpenVZ 以 Linux 为操作系统的 VS 作为客户操作系统。Solaris Zone 则只支持运行 Solaris 或 Linux 系统的客户操作系统, 对



其他平台不予支持。

本节给出了符合上述要求的虚拟化技术。

1) 支持快照和恢复功能的虚拟化解决方案, 如 VMware Workstation、VMware ESX 以及 Oracle VirtualBox 的方案。在上述系统的支持下, 新建 VS 启动时, 在其上线服务之前创建一个快照。快照包含内存和文件系统状态。当 VS 转换到脱机状态时, 需要恢复到原始状态的快照。通过这种方式实现了非持久化要求的强制执行。

2) OpenVZ 的方案, OpenVZ 是一种轻量级 (操作系统级) 的虚拟化技术。它不支持快照和恢复机制。然而, 目前已经对其设计出了一种变通的解决方案, 该解决方案利用了 Linux 上的逻辑卷管理。

这样, 在已经拥有了 VMware Workstation、ESX 以及 VirtualBox 这 3 种能够满足要求的基于管理程序的虚拟化技术的前提下, 结合上面所讨论的  $452n$  种 Web 服务配置, 就可以得到  $1356n$  种不同的软件栈 (或称为攻击面)。另外, OpenVZ 还支持表 7-3 中基于 Linux (RedHat、Ubuntu 和 SUSE) 的  $198n$  种配置。总共可以获得  $(1356 + 198)n = 1554n$  个不同的攻击面。

## 7.2 多样化带来的管理复杂性

引入多样化的软件增加了管理的复杂性, 实际上存在两个不同的问题: ①部署这样一套复杂系统所需要的成本, ②后续的持续管理和维护所带来的挑战。创建  $N$  台不同的 VS 比创建单一的系统需要更多的资源 and 努力。该问题可以通过增量部署得以缓解。即不是在系统进入运行状态前设置好所有  $N$  种 VS 类型, 而是从层次较低的多样化开始启动, 当新的 VS 类型准备好时再引入到系统中来。这将缩短产品上市时间, 但是整体投入有限。然而, 由于大多数的组件栈的组合都是现成的, 所以要做的工作包括整合具有不同特点的现有软件栈 (例如, 一个 LAMP 软件栈或者 Microsoft 软件栈) 以及移植应用程序代码到新软件栈等。上述方案的实现代价与不同团队采用多版本编程模式 (NVP) 构建的完全自定义的解决方案实现代价相比要小。

对于日常管理工作而言, 降低复杂性的首要关键在于使用主模板 VS 来管理不同类型的 VS 软件栈。当对某种 VS 类型中的部分组件存在可用的补丁 / 更新时, 与这些组件有关的所有 VS 主模板也进行更新。然后由更新的主模板 VS 再克隆得到  $K$  个新的 VS。主模板 VS 不进行在线部署, 以防止遭受攻击。所增加的工作内容包括: ①识别所有使用更新组件的 VS 类型, ②将更新应用到识别出的主模板上, ③在下次轮换前对每种 VS 类型部署新克隆的主模板。

第二个关键是自动化问题。上述过程可以在许多地方实现自动化。在实际系统中, 主要对每个 VS 软件栈实例的主模板映像进行更新, 而非处于在线服务状态的映像。然后克隆已完成更新的主模板, 并在轮换过程中进行部署——如上述的步骤③所述。最后的步骤同



标准服务器管理升级、部署步骤相同，当前市场中已经有很多工具可以使用。

换句话说，借助于标准服务器管理工具，管理复杂性并未远远超出当前的能力范围。然而，从实际情况来看，大多数 IT 公司往往倾向于选择 Microsoft 或 UNIX。支持多种服务器要求工作人员具有多平台环境工作的经验。总之，复杂性的主要成本在于在多种平台上开发同一 Web 服务的多种实现。但是，在不增加市场投放时间的情况下，成本可以随着时间的推移逐渐摊薄。

## 7.3 多样化编译技术

本节主要讨论一些用来修改程序的变形方式，从而体现功能等价意义下的多样化变体。几乎所有的多样化变体都可以在编译时生成，其中一些（如系统随机化技术、代码混淆技术等）可以通过对代码的修改来实现。此外还有一些易于理解的变形技术，单独使用时没有什么效果，但是在多变体执行环境中却能发挥强大的作用。本节只介绍那些不改变程序规格说明中内在行为的方法，这些方法让运行时环境中对变体的比较工作难以实现。

### 7.3.1 随机化技术

#### 1. 寄存器随机化技术

寄存器分配是编译器中一个历久弥新的问题，因为它是编译器在输出汇编代码前必须经历的阶段，寄存器分配算法的好坏关系着生成代码的性能和大小。一方面，为了追求极致性能，很多编译器都在寄存器分配上做了很多改善，不惜引入非常复杂的算法。另一方面，寄存器分配算法本身的性能也很关键，在诸多的 JIT 编译器（Just-In-Time compiler）中，编译器的性能同时也是程序本身的性能，因此在 JIT 编译器中还需要关注寄存器分配算法本身的效率问题。而在逆向过程中寄存器分配技术也可以通过不同的分配策略对其进行干扰，常见的分配算法包括图着色和线性扫描算法。

寄存器随机化技术的关键就是交换两个寄存器的含义。例如 Intel x86 架构的堆栈指针寄存器 esp，可以随机地用其他寄存器（如 eax）来交换。大多数攻击依赖于寄存器中的具体内容，如存放系统调用号到 eax 的攻击，执行该系统调用的攻击可能失败，因为系统将系统调用值存到 esp 中了。因为没有硬件架构支持随机化后的寄存器，所以需要在执行隐式依赖 esp 和 eax 的指令（如栈操作和系统调用指令）时交换寄存器的值。扩展现有的架构或所有寄存器完全可交换的指令集，将大幅度简化这一变体技术。一种更便于移植和轻量级的做法是只交换分配给临时变量的寄存器。例如，`addl%eax,%ebx` 指令可以很容易地与 `addl%esi,%ecx` 替换。

#### 2. 代码序列随机化技术

该技术使用指令调度、调用内联、代码吊装（code hoisting）、循环分布、部分冗余消



除, 以及许多其他编译转换来改变生成的机器代码。这些转换可以进一步改变以创建随机化的输出。多样化的应用不再易受面向返回的编程攻击和依赖于一条确定指令出现在一个确定位置的类似攻击, 从而提高了安全性。

### 3. 指令集随机化技术

机器指令的一般形式通常是固定的操作码, 操作码后跟零个或多个参数。随机化操作码的编码将生成一个全新的指令集。用这种方式修改后的程序在一个普通的 CPU 上表现出不同运行状态。事实上, 一种简单的随机化技术是在指令流中应用一个随机密钥的 XOR 函数。CPU 执行之前, 操作码必须使用随机密钥进行解码。这既可以通过软件做到, 也可以通过在硬件上扩展 CPU 来减少开销。如果攻击者注入的代码不是适当编码过的代码, 虽然在执行之前仍然可以通过解码处理, 但这会导致代码非法, 很可能在几条指令之后引发 CPU 异常, 或者至少达不到预期目的。但研究表明这种技术并不能抵御只修改栈或堆变量和改变程序的控制流的攻击。相关研究还表明, 在特定情况下可以查看到受攻击的程序, 并可通过猜测随机密钥来抵御指令集随机化, 但这些都是需要时间和多次尝试才能突破的, 由于密钥变化的随机性, 这种突破的难度是很大的, 因此, 在一定程度上提高了安全性。

### 4. 堆布局随机化技术

可以通过堆布局随机化使堆溢出攻击失效。由于在堆上动态分配的内存是随机放置的, 所以很难预测下次分配的内存块的位置。DieHard 等工具能显示如何用堆布局随机化防止堆溢出, 为采用该技术抵御堆溢出攻击提供了帮助。

### 5. 栈基址随机化技术

作为一种保护机制, 栈基址随机化技术已被多个操作系统在其新版本中使用。在应用程序的每次启动时, 栈开始于一个不同的基址。由于栈的基址不固定, 攻击者攻击一个系统的难度将大大增加。事实上, Linux 内核的 PAX 补丁已经应用了这种技术。

### 6. 系统调用号随机化技术

这种技术与指令集随机化有关。使用直接硬编码系统调用的所有攻击需要先知道正确的系统调用号。通过改变系统调用号, 注入的代码执行一个随机的系统调用, 导致一个完全不同的行为甚至错误。但是, 因为数量有限, 暴力攻击可以获得新的系统调用号。该技术有一个缺点: 内核必须了解新的系统调用号, 或者有一个重写工具在执行前恢复系统调用号。这种方法最早被用来保护 Linux 和 Windows 系统。

### 7. 库入口点随机化技术

获得对系统控制的另一种方法是直接调用库函数, 而不是使用硬编码的系统调用。对于这种方法, 攻击者必须事先知道库函数的确切地址。实际上, 猜测库函数的地址是相当容易的, 因为相同的操作系统往往会将共享库映射到同一个虚拟地址。随机化库的入口点是防止这种攻击的有效方法, 可以在程序中重写函数名或者在加载时完成。重写的优点是



只需要进行一次。该技术并不能防止传统的缓冲区溢出，但是可以使注入的代码无效以保护系统。Linux 补丁 PAX 通过改变加载动态库的 `mmap` 函数的基址来实现库入口点的随机化，从而达到保护系统安全的目的。

8. 程序基址随机化技术

地址空间布局随机化 (ASLR) 技术作为一种安全技术目前已经应用到大多数主要的操作系统中，依靠运行时随机化来提高安全性，它已被证明是一种防止攻击的有效方法。然而，由于当前的二进制格式的设计早于这项技术，所以许多现有的程序是根据在固定内存地址中加载的假设来生成的。对于这些程序，ASLR 技术只能用于程序使用的动态库，而不是程序本身。然而，一种模拟该随机化的方法是在链接时随机化程序的加载地址，使每个程序处在攻击者无法预测的不同地址。例如，Linux 操作系统上的程序有一个默认的基址 `0x08048000`，包含 128MB 的地址空间。该地址之前的空间从未被程序使用过，这是一种浪费。可以通过缩小或扩大该间隙随意调整程序在内存中的布局来达到随机化目的，当然，这样做有时会牺牲一些程序的可用内存。图 7-1 是该技术的示例。一种基址随机化的实现是使用重写技术对编译之后的程序代码进行变形，从而实现多样化。



图 7-1 程序基址随机化实例

7.3.2 代码混淆技术

越来越多的软件 (如 Java、.NET) 以平台无关的中间代码发布。以这种方式发布的软件代码与源码类似，比起传统的二进制可执行代码更易遭到静态分析、逆向工程和篡改等恶意攻击。随着网络技术的发展，更多软件是在不确定 (甚至是恶意) 的环境下运行的，主机可以任意地对软件进行分析和跟踪，而且随着各种逆向工程技术的发展，使得对软件的攻击变得更加容易。如何保护软件中的核心算法和机密数据成为人们关注的一个焦点。

代码混淆技术是一种重要的软件保护方法。代码混淆指对拟发布的应用程序进行保持语义的变换，使得变换后的程序和原来的程序在功能上相同或相近，但是更难以被静态分析和逆向工程所攻击。Barak 等在理论上证明了在终端运行且没有硬件辅助保护机制的代码混淆技术不可能提供彻底的保护，但在实际应用中，代码混淆并不需要对软件提供绝对的保护，只要能使得攻击者的攻击变得不经济，则可以认为达到目的。

1. 控制流混淆

该类混淆的目的是使得攻击者对程序的控制流难以理解，如加入模糊谓词、用伪装的



条件判断语句来隐藏真实的执行路径。

模糊谓词就是具有对施加混淆者易于判断而对攻击者来说难于推导的特性谓词。如果一个谓词  $P$  在  $p$  点的输出在加混淆时是已知的，对一直是输出 FALSE 和 TRUE 的谓词分别记为 PF 和 PT，对可能是 TRUE 也可能是 FALSE 的记为 “ $P?$ ”

混淆过程之间的固有逻辑关系，如内嵌 (in-line) 和外联 (out-line)，也可以达到隐藏真实的逻辑关系的目的。内嵌是将一小段程序嵌入被调用的每一个程序点，外联是将没有任何逻辑联系的一段代码抽象成一段可被多次调用的程序。有的编译器能够静态地分析谓词的真假条件，如果某些路径从不执行，就会移除这些路径。Collberg 等人的方法基于对假名分析、并发分析、数据依赖分析的复杂性来构造对自动分析和解混淆工具具有抵抗力的模糊谓词。

## 2. 花指令的插入

花指令即为设计者特别构思的一段代码，希望使反汇编的时候出错，让破解者无法清楚地反汇编程序的内容。经典的花指令如目标位置是另一条指令的中间，这样在反汇编的时候便会出现混乱。花指令有可能利用各种指令，如 jmp、call、ret 的一些堆栈技巧、位置运算等。

## 3. 无功能代码的插入

无功能代码的插入可分为无功能源代码的插入和无功能汇编（或目标）代码的插入。在无功能源代码的插入方面，一种是通过在源文件中添加一些无功能的垃圾代码，但其又与源文件中某些变量绑定，使其不会在编译器优化过程中被优化，从而阻碍攻击者分析，使得漏洞的发现更为困难。另一种是把经过精心挑选的能影响控制流 / 数据流的源代码级别的干扰代码插入待编译程序中的适当位置，能够在很大程度上干扰逆向分析，从而提高代码的安全性。但这些干扰代码需要精心设计，首先是不能影响程序的正常功能，同时还要防止在编译优化时被优化掉，或者被逆向分析时发现是干扰代码而被去掉。这些干扰代码段的设计与实现以及插入位置的选择，是源代码级干扰码插入技术所要面对的关键问题。

还有一种类似于栈帧填充和栈布局随机化的方法，它使得攻击者难于利用已有的对二进制目标码布局方面的知识来实施攻击，该方法在防止面向返回的编程 (return oriented programming) 方面非常有用。有些序列是短的代码序列，并且执行后没有实际影响。这些序列能够在代码中作为填充，使后面的代码向后推移几字节。通过这些无实际功能的指令（或称为 NOP 指令）来增加地址的偏移量，累加代码的长度，可以移动后续的代码序列，这可以防止那些对处于固定位置的某些已知字节识别后而进行的攻击。例如下面三条指令都属于无实际操作功能的指令：`xchgl%esi,%esi`、`leal(%edi),%edi` 和 `movl%eax,%eax`。在 PittSField 系统中软件故障隔离采用 NOP 插入技术确保跳转目标的对齐，所以攻击者根本无法利用现有的跳转指令跳转到一个正确指令的中部，从而提高了系统的抗攻击性。



#### 4. 等效替换技术

此处要介绍的等效替换技术分为中间代码级等价替换技术、汇编（或目标）代码等效替换技术，以及程序段和函数重排序技术。

中间代码级等价替换技术主要针对中间代码，实施中间代码级别的等效代码序列替换，从而进一步增强代码的多样性，产生更多的不易被逆向分析和攻击的变体。该技术依赖于对中间代码的深刻把握，且应该能确保等价替换的结果不会被后端优化破坏。由于中间代码的种类较多，不同的中间代码有不同的等价替换方法，因此，除了一些共性的中间代码级等价替换技术之外，还有一些特有的等价替换技术，这些都是需要认真关注和研究的问题。

汇编（或目标）代码等效替换技术则是针对指令集的特征而采取的等效替换技术，许多指令集体系结构提供了不同的指令或指令序列，但它们在某些特定的情况下具有相同的效果，因此可以相互替换。可以在没有性能损失的情况下用等价的指令替换原有指令，从而改变二进制序列，抵抗相应的恶意行为攻击。

例如，指令（以字节编码）：

```
movl    %edx, %eax      89 D0
xchgl   %edx, %eax      92
```

可以被替换为：

```
leal    (%edx), %eax    8D 02
xchgl   %eax, %edx      87 D0
```

不难看出，尽管替换的指令序列与之前的指令序列功能等价，但是它们的二进制代码是不同的。

不同形式但功能等价的指令替换一般不会对性能有影响，但却以不同的方式改变了静态的属性和内容。类似的变化也可用在算术指令，如改变 `mul` 为 `shl` 或反之，但这种变化已经由编译器的优化技术完成，对性能可能有显著影响。一个众所周知的优化是由程序员手动或自动由编译器所完成的乘法指令优化，可以将 2 的幂变成移位。乘法比移位慢很多，然而这些变换却可以提高应用程序的安全性，因此，从安全性而言是合适的。

在程序段和函数重排序技术方面，考虑到大型程序往往包括很多模块，每个模块通常对应一个独立的源文件。每个模块被分为不同类型的段，如数据段和代码段等，模块通常包含一些互相调用的函数。有些攻击依赖于特定的全局函数的特定位置，因此，一种使程序更安全的方法就是在模块一级重新排序函数，在链接的时候重新排序代码段。另一种方法是链接时优化，在全局层面上对程序中的函数进行简单的重新排序。该技术可以被应用到数据段和变量，即之前提出的堆布局随机化；也可应用到二进制中的其他任何段，因此是一项简单有效的安全防护技术。

#### 5. 代码混淆技术性能指标

对程序进行混淆变换后的效果如何，通常从强度（potency）、耐受性（resilience）、开销



(cost)、隐蔽性 (stealth) 4 个方面来评估。

1) 强度。指混淆变换究竟给原始程序加上了多大的复杂度。对于软件复杂度的评估, 以 McCabe、Hearrion 等人的角度, 一个应用程序所包含的谓词越多、条件和循环结构嵌套的次数越多, 该程序的复杂度就越高。另外, 数据结构的复杂度越高、形式参数的数量越多、继承树的层次越多, 该程序也就越复杂。所以, 可以从以上这些方面大致地判断变换后的程序复杂度增强了多少。

一般来说, 增加程序的长度并引入新的类和方法; 引入新的谓词, 并增加条件语句和循环结构的嵌套层数; 增加方法的参数的个数; 增加继承树的高度等方法都可以提高混淆变换的强度。

2) 耐受性。指变换后的程序对使用自动去混淆工具进行攻击的抵抗度。某些强度很高的混淆变换对攻击者解混淆的抵抗能力很差。耐受性的评估应该从编制自动去混淆程序的难度和去混淆程序执行的时间、空间开销两个方面来考虑。

与强度相比, 耐受性主要是评估一个加混淆变换对自动去混淆工具的耐受性, 而强度则是评估混淆变换在多大程度上干扰了人对程序的阅读和理解。

如果某个变换是单向 (one-way) 的, 即变换是不可恢复的, 如从程序中移除有助于人阅读和理解的格式、变量名、注释等不影响程序执行的有关信息, 这种变换的耐受性就是最强的。对于其他如在程序中加入不改变程序行为的假的谓词等变换, 能够加大攻击者人工阅读的工作量, 但却是可恢复的变换。

3) 开销。指经过混淆变换后的程序在执行时由变换所带来的额外的执行时间和所需存储空间的开销。某些变换 (如改名、格式移除) 是零开销的, 但多数变换都会带来或多或少的时空开销。

变换的开销主要来自于代码增加、数据膨胀和循环增多。而大多数混淆变换所引入的动态数据的膨胀是开销的主要来源。程序执行时动态数据的增加会导致内存与硬盘的频繁数据交换, 对 Java 来说还会加重 Garbage Collector 的负担。更为严重的情况是, 原本能正常运行的程序在经过混淆变换后, 可能因为耗尽内存空间而无法运行。

开销的大小与程序执行时的上下文有关。如在多层循环内的一条简单赋值语句所带来的开销要远大于在循环体之外的同样语句所带来的开销。

4) 隐蔽性。耐受性好的混淆变换不容易被自动去混淆工具所去除, 但却可能很容易被攻击者的人工分析识破。特别是, 如果一种变换所引入的代码和原始程序有较大的差异性, 就会轻易地被攻击者识破。例如在程序中引入一个 if 语句, 判断一个极大的整数是否为素数, 这样一个语句对自动去混淆工具来说具有极好的耐受性, 但有经验的攻击者会很轻易地察觉。显然, 隐蔽性也是与程序的上下文高度相关的。

### 7.3.3 与堆栈相关的多样化技术

#### 1. 反向堆栈技术

大部分处理器架构将栈增长的方向设计成单方向的。例如, 在 Intel x86 指令集中, 所



有预定义的堆栈操作如 `push` 和 `pop` 仅适用于向下生长的堆栈。通过对栈指针的加减法增加栈的操作指令，可以产生一个向上生长堆栈的变体。因为栈的布局，包括在栈中分配缓冲区和变量，与之前的方法完全不一样，因此，可以防护臭名昭著的缓冲区溢出和经典的依赖于向下增长栈的栈溢出攻击。换句话说，通过改变栈增长方向，缓冲区溢出覆盖受影响的栈区域包含完全不同的数据和控制值，从而摆脱缓冲区溢出攻击。

## 2. canaries 技术

一种栈溢出的保护机制是在一个缓冲区和一个活动记录（返回地址和帧指针）之间插入一个 `canary` 值。若活动记录被缓冲区溢出利用修改，则 `canary` 值也将被覆盖。在函数返回之前，对 `canary` 值进行检查，如果 `canary` 值被改变了，就中止程序的执行。常见的 `canary` 值有以下几种。

### （1）终结字符 canaries (Terminator canaries)

由于绝大多数的溢出漏洞都是由那些不做数组越界检查的 C 字符串处理函数引起的，而这些字符串都是以 `NULL` 作为终结字符的。选择 `NULL`、`CR`、`LF` 这样的字符作为 `canary` 值就成为很自然的事情。例如，若 `canary` 值为 `0x000aff0d`，为了使溢出不被检测到，攻击者需要在溢出字符串中包含 `0x000aff0d` 并精确计算 `canaries` 的位置，使 `canaries` 看上去没有被改变。然而，`0x000aff0d` 中的 `0x00` 会使 `strcpy()` 结束复制，从而防止返回地址被覆盖。而 `0x0a` 会使 `gets()` 结束读取。插入的终结字符 `canaries` 给攻击者制造了很大的麻烦。

### （2）随机 canaries (Random canaries)

这种 `canaries` 是随机产生的，并且这样的随机数通常不能被攻击者读取。这种随机数在程序初始化时产生，然后保存在一个未被映射到虚拟地址空间的内存页中。这样当攻击者试图通过指针访问保存随机数的内存时就会引发段错（`segment fault`）。但是由于这个随机数的副本最终会作为 `canary` 值被保存在函数栈中，攻击者仍有可能通过函数栈获得 `canary` 值。

### （3）随机 XOR canaries

这种 `canaries` 由一个随机数和函数栈中的所有控制信息、返回地址通过异或运算得到。这样，函数栈中的 `canaries` 或者任何控制信息、返回地址被修改就都能被检测到。

目前主要的编译器堆栈保护实现，如 `Stack Guard`、`Stack-smashing Protection (SSP)` 均把 `canaries` 探测作为主要的保护技术，但是 `canaries` 的产生方式各有不同。

`Stack Guard` 是第一个使用 `canaries` 探测的堆栈保护实现，它于 1997 年作为 GCC 的一个扩展发布。最初版本的 `Stack Guard` 使用 `0x00000000` 作为 `canary` 值。尽管很多人建议把 `Stack Guard` 纳入 GCC，作为 GCC 的一部分来提供堆栈保护。但 GCC 3.x 没有实现任何堆栈保护。直到 GCC 4.1 堆栈保护才被加入，并且 GCC 4.1 所采用的堆栈保护实现的并非是 `Stack Guard`，而是 `Stack-smashing Protection (SSP)`，又称 `ProPolice`。

`SSP` 在 `Stack Guard` 的基础上进行了改进和提高。它是由 IBM 公司的工程师 Hiroaki Rtoh 开发并维护的。与 `Stack Guard` 相比，`SSP` 保护函数返回地址的同时还保护了栈中的 `EBP` 等



信息。此外，SSP 还有意将局部变量中的数组放在函数栈的高地址，而将其他变量放在低地址，这样就使得通过溢出一个数组来修改其他变量（比如一个函数指针）变得更为困难。

GCC 4.1 中有 3 个与堆栈保护有关的编译选项，它们的具体说明如下：

- 1) `-fstack-protector`：启用堆栈保护，不过只为局部变量中含有 `char` 数组的函数插入保护代码。
- 2) `-fstack-protector-all`：启用堆栈保护，为所有函数插入保护代码。
- 3) `-fno-stack-protector`：禁用堆栈保护。

上述这种 Canaries 技术可以防止标准的堆栈溢出攻击，但不能防止堆和函数指针覆盖的缓冲区溢出。除此之外，也存在一些特殊情况，攻击者可以在不修改 canary 值的情况下覆盖活动记录，使得原有检测方法失效。

### 3. 变量重排序技术

该技术增强了前面提出的 canary 值保护的有效性。即使有 canary 值，攻击者也可以覆盖被放置在栈上的缓冲区和 canary 值之间的局部变量，来达到其目的。为了防止这种情况发生，缓冲区被立即置于 canary 变量后，其他变量和函数参数的副本放在所有缓冲区之后。这种技术与 canary 组合对在检查 canary 值之前接管程序的攻击更有效。

### 4. 栈帧填充技术

扩展栈帧的长度是一种用来防止基于栈的缓冲区溢出的方法。通过扩展栈帧，基于栈的缓冲区溢出无法成功地实施，因为有效载荷没有大到足以覆盖栈帧中的返回地址。添加虚设的堆栈对象，即填充值，是实现这种随机化的一个直接方法。这可以通过两种方式来实施：大的空间放置到栈帧的顶部，或空闲空间放置在两个栈对象之间。尽管目前对栈帧之间填充区域的大小没有理论上的限制，但在递归程序中不可能使用大的填充空间。

## 7.4 多样化编译的应用

多样化编译技术目前主要在提高安全性方面得到了应用，本章前几节中介绍的方法和技术的适用性和适应性是不同的，表 7-4 列举了在一个多变体执行环境和海量软件多样性所采用的多样化编译技术及其适用性和适应性。该表可以为那些希望利用多种方式来实现更高安全性的程序员提供指南。

表 7-4 多样化编译对多变体执行环境 (MVEE) 和海量软件多样性 (MSSD) 提供的方法

适用性	目标	技术	实现
MVEE 和 MSSD 均适用	代码	源代码级干扰码插入	把经过精心挑选的能影响控制流 / 数据流的源代码级别的干扰代码插入到待编译程序中的适当位置，提高代码的安全性
		寄存器随机化	寄存器的含义被交换，作用被改变
		库入口点随机化	库的加载地址被随机化
		Nop 插入	无实际操作功能的指令（也称为 NOP 类指令）被插入到指令流中



(续)

适用性	目标	技术	实现
MVEE 和 MSSD 均适用	数据	堆布局随机化	动态分配的内存被随机放置在堆中
		栈帧填充	在局部变量和返回地址之间添加随机填充
		变量重排序	非缓冲局部变量被放置在缓冲区局部变量之前
仅 MVEE 适用	代码	指令集随机化	指令流被一个随机选择的密钥随机化（如通过 XOR 函数实施随机化等）
		系统调用号随机化	分配给系统调用的号是随机改变的
	数据	反向堆栈	栈与原生架构生长方向相反
		canaries	一个随机值（称为 canary）放在栈上，位于函数帧的返回地址之前，并在结尾阶段进行检查
仅 MSSD 适用	代码	代码序列随机化	编译转换有选择地使用随机化的指令流
		等价指令替换	指令被功能相当的等价指令替换
	代码 / 数据	中间代码级等价替换	实施中间代码级别的等效代码段替换，从而增强代码的多样性，产生更多的不易被逆向分析和攻击的变体
		程序基址随机化	该程序的加载地址是随机改变的
		程序段、函数重排序	功能和区段被放置在地址空间的任意位置
	数据	栈基址随机化	堆栈被放置在地址空间的一个位置

7.4.1 多样化编译在安全防御方面的应用

多样化编译技术的策略是通过不同层次的一系列多样化方法，产生许多不同的多变体，当多变体数目足够多时，锁定目标的攻击将会需要很大的代价，因为如果攻击者想要破解这些多变体，他需要开发大量的不同攻击代码，并且攻击者无法事先知道某个多变体的漏洞所在。不同多变体之间的区别同样使得攻击者用于某个多变体的攻击方法对另一个多变体可能并不适用，这些编译器的多样化方法同样让攻击者通过反编译破解软件极其困难。攻击者破解软件通常需要两个重要的信息，即软件的版本和相应的补丁信息，而在多样化的软件环境中，软件的每个实例都是特殊的，它们的二进制文件之间都是不一样的，从而大大提高了破解软件的难度。

随着信息技术的发展，计算机已成为人们工作、学习和生活中不可缺少的部分，计算机软件正是推动这一发展的主要动力。然而，盗版现象日益严重，引起了许多企业和学者的关注。要解决盗版问题，一方面应该依靠法律手段，另一方面应该研究和讨论如何借助各种技术手段来有效保护软件产品（这对于防止盗版有着重要的现实意义）。根据产品实现手段，软件保护技术分为两类，一类是纯软件的，另一类是纯硬件或软硬结合的。由于部署便捷和几乎为零的单份拷贝附加成本，软件保护技术主要集中在第一类，而这类研究又主要集中在混淆变换、加密、软件水印三个领域。现实世界的攻击是多角度与多层次的，为了给软件提供更强的保护，使软件更能抵御静态解析、逆向工程和篡改等恶意攻击，将多种软件保护技术结合起来是一种有效的选择。

基于多样化编译技术可以编写一系列多样化编译工具，目前，这些工具产生的软件多



样化的地址空间分布及随机化技术已经得到了一定的应用,应用结果表明,其对栈地址溢出等攻击非常有效,相对应的软件多样化给软件的反编译以及破解造成了较大困难。伴随着近年来各种信息技术的飞速发展,云计算、云存储、CPU 性能的提高都为多样化编译技术的应用提供了极大的可能性。

其次,随着近年来网络安全环境不断恶化,蠕虫借助软件漏洞迅速传播,它们严重威胁着计算机网络基础设施安全。多样化编译技术带来的软件多样化,从根本上改变了传统的防御观念,其着眼于增大攻击者攻击难度,使得群体性的破坏得到遏制,而传统的防御技术存在滞后性较高、维护成本高等缺点。在这种形势下,多样化编译工具带来的软件多样化自然成为安全防御的创新性技术之一。

### 7.4.2 多样化编译工具的结构组成及原理

多样化编译工具可以通过在源代码到中间代码阶段和中间代码到目标代码阶段增加编译器配置选项进行多样的变化。图 7-2 给出了作者设计的多样化编译工具的组成原理图。

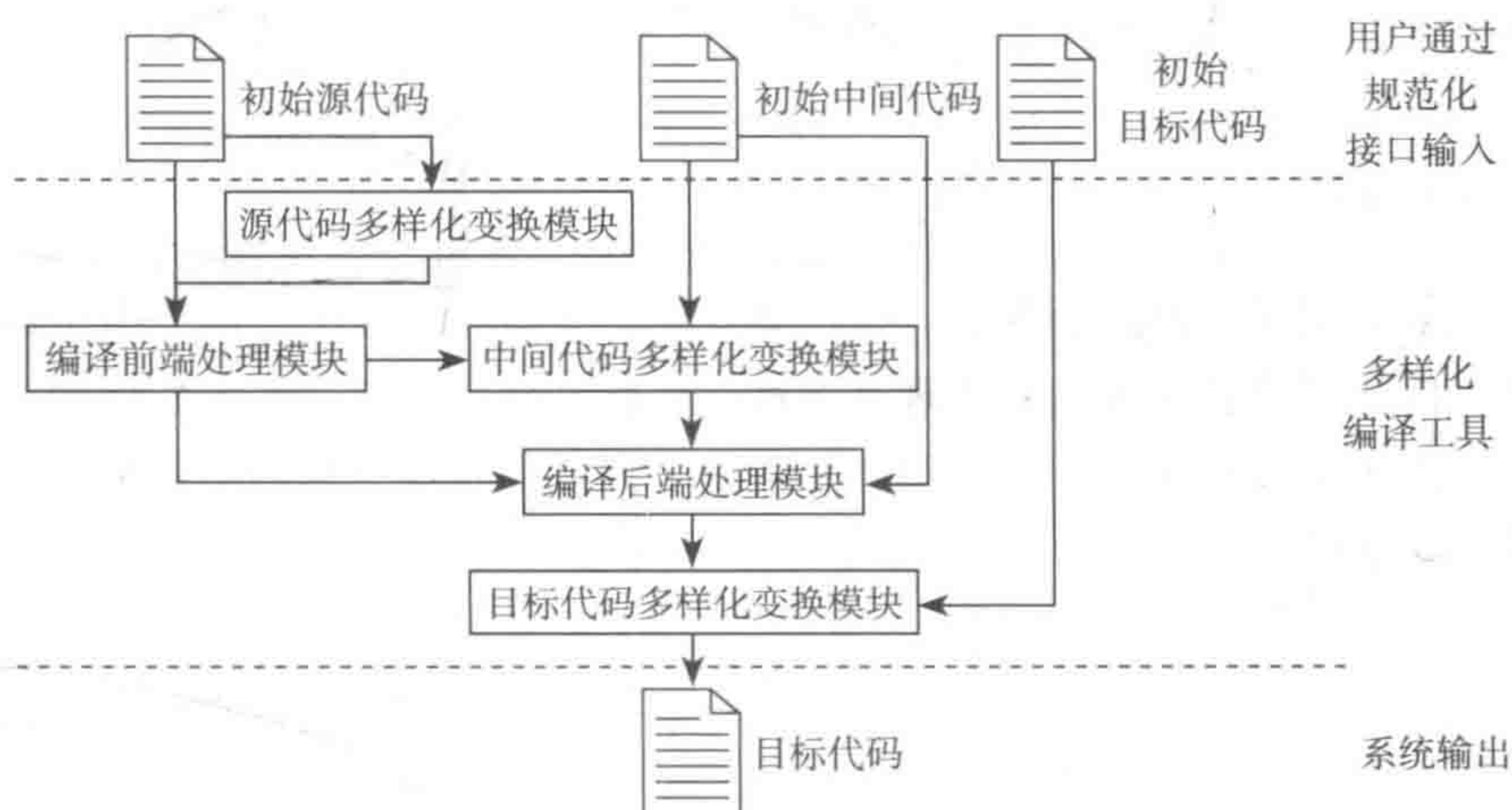


图 7-2 多样化编译工具的组成原理图

图 7-2 所示的多样化编译工具由源代码多样化变换模块、编译前端处理模块、中间代码多样化变换模块、编译后端处理模块和目标代码多样化变换模块,以及规范化的用户接口组成。其中,编译前端处理模块和编译后端处理模块与普通的编译处理过程中的相关模块功能一致。下面只对其他几个模块的原理加以阐述。

源代码多样化变换模块的基本原理:首先设计并实现能够影响控制流和数据流的源代码级别的干扰代码,源代码多样化变换模块能够为待编译程序选择合适的干扰代码,并能够插入到合适位置,从而干扰逆向分析,提高代码的安全性。当然,这些干扰代码需要精心设计并合理插入,防止在编译优化时被优化,还应具有抗逆向分析能力,防止在被逆向分析时发现而被去除。干扰代码的管理也非常重要,可以采用建立和维护干扰代码库的方



式来加强管理，提高效率。

中间代码多样化变换模块的基本原理：中间代码多样化变换模块提供的等价替换技术，主要针对中间代码，可以有效实施中间代码级别的等效代码段替换，从而增强中间代码的多样性，产生更多的不易被逆向分析和攻击的变体。上一节讲述的适合于中间代码层的等价替换技术都可以被采用和不断优化。

目标代码多样化变换模块的基本原理：目标代码多样化变换模块采用多种目标代码级的变换技术，从指令流、控制流、数据流等多个流程和堆栈布局、文件布局、入口点随机化、基址随机化等多个角度实施变换，从而增加目标代码的多样性，抵御逆向分析和攻击。

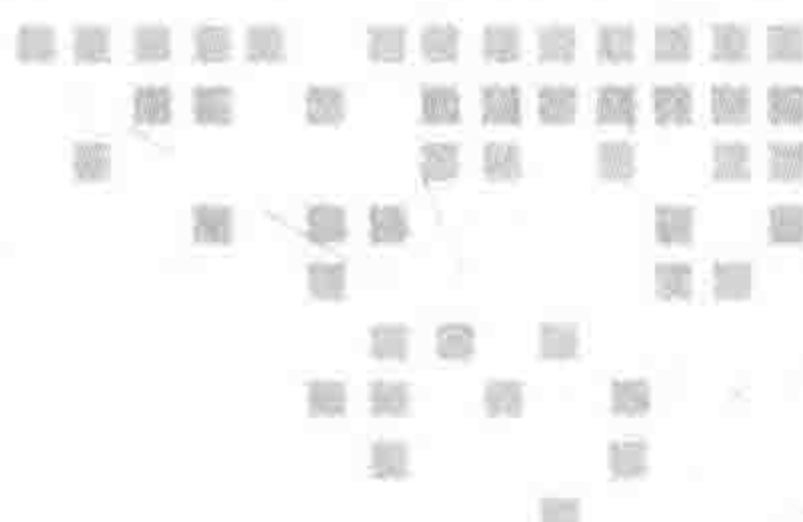
规范化的用户接口基本原理：提供统一的编译命令模式和一体化的图形界面，将多层次的等价变换技术和调试优化技术集成在一起，方便用户使用。

目前，多样化编译技术的应用已经深入到操作系统的实现、网络协议的实现，以及具体应用的实现等多个方面。作者课题组将多样化编译技术应用到 Web 服务器的实现及应用中，通过多个层面的多样化技术，大大提高了系统的抗攻击性，展示了多样化编译技术的美好应用前景。

## 7.5 本章小结

本章首先从应用层、Web 服务层、操作系统层、虚拟层等多个层面介绍了软件多样化的机会，然后阐述了多样化带来的管理复杂性，在此基础上，探讨了多样化编译的几项关键技术，最后简述了多样化编译的应用，并给出了多样化编译工具的结构组成原理图，概述了相关模块的原理。





# 反编译的对象——可执行文件格式分析

## 8.1 可执行文件格式

可执行文件是按照一定的格式组织的，往往由操作系统来决定。两种常见的可执行文件格式是 PE 和 ELF 格式，分别对应 Windows 和 Linux 操作系统。由于反编译的对象是二进制代码的可执行文件，因此本章重点分析可执行文件的格式。

### 8.1.1 PE 可执行文件格式

Windows 系统中的 PE 文件格式（Portable Executable File Format），即可移植执行文件格式，这种文件格式的框架如表 8-1 所示。

表 8-1 PE 文件格式框架

#### (1) DOS MZ header 和 DOS stub

所有的 PE 文件（包含 32 位的 DLL）必须以一个简单的 DOS MZ header 开始。该结构的作用是判断操作系统的类型：若程序在 DOS 下执行，DOS 系统就能识别出这是有效的执行体，然后运行紧随 MZ header 之后的 DOS stub。DOS stub 实际上是一个有效的可执行代码，在不支持 PE 文件格式的操作系统中它将简单显示一个错误提示，类似于字符串 “This program cannot run in DOS mode” 或者程序员可根据自己的意图实现完整的 DOS 代码。通常 DOS stub 结构没有特殊作用。

DOS MZ header
DOS stub
PE header
Section table
Section 1
Section 2
Section ...
Section n

#### (2) PE header

PE header 是 PE 文件头结构 IMAGE\_NT\_HEADERS 的简称，包含 IMAGE\_FILE\_HEADER 和 IMAGE\_OPTIONAL\_HEADER32 两个结构，其中定义了许多 PE 装载器用到的关键字段，如 Machine 字段指明硬件平台类型、Characteristics 字段指明文件类型是 DLL 还是普通的可



执行文件、AddressOfEntryPoint 字段指明文件执行的入口地址、DataDirectory 字段指明不同用途的数据块等。执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器将从 DOS MZ header 中的 e\_lfanew 字段找到 PE header 的起始偏移量，因而跳过了 DOS stub 直接定位到真正的文件头 PE header。

(3) Section

PE 文件的真正内容划分成块，称为 Sections (节)。每节是一块拥有共同属性的数据，比如代码 / 数据、读 / 写等。Windows 在将可执行文件装载到内存时，采用与文件映射类似但又不完全相同的做法，根据不同的节属性和节大小等因素把节加载到不同的内存区域中，内存页的属性反映对应的节的属性。节的划分是基于各组数据的共同属性，而不是逻辑概念。节划分的依据不是数据或代码是如何使用的，若 PE 文件中的数据或代码拥有相同属性，它们就能被归入同一节中。节中类似于“data”、“code”或其他逻辑概念（节名称仅仅是区别不同节的符号，类似“data”、“code”的命名只是为了便于识别，只有节的属性设置决定了节的特性和功能）并无重要作用，如果某块数据想赋予只读属性，就可以将该块数据放入置为“只读”的节中，当 PE 装载器映射节内容时，它会检查相关节属性并置对应内存块为指定属性。

(4) Section table

PE 文件中所有节的属性都被定义在 Section table (节表) 中，PE header 接下来的数组结构 Section table 就是 PE 格式中等价于目录的内容。节表由一系列的 IMAGE\_SECTION\_HEADER 结构排列而成，每个结构用来描述一个节，包含对应节的属性、文件偏移量、虚拟偏移量等。节表的顺序和数量与节在文件中的顺序和数量是一致的，因此，可以把节表视为逻辑磁盘中的根目录，每个数组成员等价于根目录中的目录项，节就相当于各种文件。

8.1.2 ELF 可执行文件格式

1. ELF 文件格式概述

ELF (Executable and Linked Format) 是 UNIX 系统实验室 (USL) 作为应用程序二进制接口 (Application Binary Interface) 开发和发布的可执行链接格式。ELF 文件是 Linux 目标文件格式 (Object File Format)，目标文件既参与程序执行也参与程序链接。目标文件格式提供两种视图，一种是从文件链接的角度，文件由 text、data、rodata、bss 等不同 section 组成，另一种是从程序执行的角度，文件由 segment 组成，如图 8-1 所示 (本书将 section 称为节，将 segment 称为段)。

文件链接角度	程序执行角度
ELF header	ELF header
Program header table (可选)	Program header table
Section 1	Segment 1
...	...
Section n	Segment n
...	...
...	...
Section header table	Section header table (可选)

ELF header 在文件开始处描述了整个

图 8-1 ELF 文件格式



文件的组织。ELF 文件有两个可选择的头表：程序头表（Program header table）或节头部表（Section header table）。通常，链接文件具有节头部表，可执行文件有程序头表，共享目标文件既有节头部表又有程序头表。Section 提供了目标文件的各项信息，如数据、指令、重定位信息、符号表等）。

## 2. ELF 文件的 header

文件头用于定位文件的其他部分，位于文件开始部分。如图 8-2 所示的 64 位 ELF 文件 header，前两字节为 ELF 文件的标识符，翻译器通过该标识符来确定此文件是否为 ELF 格式文件，获得操作系统的相关信息。在文件头还有其他表的偏移，如程序头表、节头部表等，翻译器应用这些偏移获得访问信息所在表的位置。翻译器通过表项的个数判断访问表内信息时是否越界。文件头能得到程序执行的第一条指令的位置，即入口点信息。

```
typedef struct
{
    unsigned char    e_ident[16];    /*ELF 标识 */
    Elf64_Half       e_type;          /* 目标文件类型 */
    Elf64_Half       e_machine;       /* 处理器类型 */
    Elf64_Word       e_version;       /* 目标文件版本 */
    Elf64_Addr       e_entry;         /* 目标文件入口 */
    Elf64_Off        e_phoff;         /* 程序头偏移 */
    Elf64_Off        e_shoff;         /* 节头部偏移 */
    Elf64_Word       e_flags;         /* 处理器规范标识 */
    Elf64_Half       e_ehsize;        /* ELF 头大小 */
    Elf64_Half       e_phentsize;     /* 程序头表项大小 */
    Elf64_Half       e_phnum;         /* 程序头表项数量 */
    Elf64_Half       e_shentsize;     /* 节头部表项大小 */
    Elf64_Half       e_shnum;         /* 节头部表项数量 */
    Elf64_Half       e_shstrndx;      /* 节名字符串表索引 */
}Elf64_Ehdr;
```

图 8-2 ELF64 程序头

## 3. ELF 文件中的表

节头表是反编译器经常使用的表之一。当节被装入内存执行时，反编译器由 ELF 程序头找到节头表，从节头表找到初始化数据节、未初始化数据节、只读数据节的起始虚地址和大小。节头表结构如图 8-3 所示。

字符串表中包含节名和符号名字符串，字符串表是一个包含 null 终止字符串的字节数组。节头部表项和符号表项通过相对于串表开始的索引来引用字符串表中的字符串，字符串表中的第一字节定义为 null，索引 0 指向 null 或者不存在的名字。

符号表存放动态链接例程的名字，用于动态链接，符号表入口格式如图 8-4 所示，包括三类信息：



- 符号名：表中符号的名字，类型为 SRTING。
- 符号地址：符号的本地地址，类型为 ADDRESS。
- 符号大小：与符号相关的字节的大小，类型为 INT。

```
typedef struct
{
    Elf64_Word      sh_name;          /* 节名 */
    Elf64_Word      sh_type;          /* 节类型 */
    Elf64_XWord     sh_flags;         /* 节属性 */
    Elf64_Addr      sh_addr;          /* 内存中的虚拟地址 */
    Elf64_Off       sh_offset;        /* 文件偏移 */
    Elf64_XWord     sh_size;          /* 节大小 */
    Elf64_Word      sh_link;          /* 到其他节的链接 */
    Elf64_Word      sh_info;          /* 其他信息 */
    Elf64_Xword     sh_addralign;     /* 地址对齐边界 */
    Elf64_Xword     sh_entsize;      /* 节表项的大小 */
}Elf64_Shdr;
```

图 8-3 ELF64 节头部表结构

ELF 文件在装入时，动态链接器从 DYNAMIC 节的内容获取动态链接需要的信息。当程序中存在间接调用动态链接库时，DYNAMIC 节将指出重定位表 Elf64\_Rela 的入口地址及其大小。重定位表提供了在程序执行前需要重定位的地址信息，其结构如图 8-5 所示。

结构中域 r\_offset 指示需要进行重定向的位置，在可执行文件中是被重定位存储单元的虚地址。域 r\_info 分为两部分，一部分表示需要被重定位的符号在符号表中的索引，另一部分表示的是重定位类型。域 r\_addend 是一个常量，用来计算存储在重定位域中的值。

```
Typedef struct
{
    Elf64_Word      st_name;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf64_Half      st_shndx;
    Elf64_Off       st_value;
    Elf64_Xword     st_size;
}Elf64_Sym;
```

图 8-4 符号表表项结构

```
typedef struct
{
    Elf64_Addr      r_offset;         /* 引用地址 */
    Elf64_Xword     r_info;           /* 符号索引和重定位类型 */
}Elf64_Rel;
Typedef struct
{
    Elf64_Addr      r_offset;         /* 引用地址 */
    Elf64_Xword     r_info;           /* 符号索引和重定位类型 */
    Elf64_Sxword    r_addend;        /* 表达式的常量部分 */
}Elf64_Rela;
```

图 8-5 ELF64 重定位表结构



#### 4. ELF 文件中的代码组织

ELF 文件通常包含有数据、文本和 bss 节，即 .text、.data 和 .bss 节，除此以外还有 .init 和 .fini 节。输入文件都拥有一系列不同类型的节，链接器从输入文件中将不同节收集起来，并记下需要在运行时从库文件中解析的符号，生成 .interp、.got、.plt 节和符号表节以支持动态链接。然后链接器会按照约定的顺序安排存储空间，在 Linux 系统中按如下顺序安排：文本节 .text、数据节 .data、.bss 节。

如上所述，在编译生成 ELF 文件的过程中链接器要处理两种不同类型的输入文件：用户文件和库文件。编译器大都同时采用静态链接和动态链接两种链接策略，以获得最佳的运行效果。对用户文件的处理，一般情况下是直接将用户代码放置在 .text 节中合适的位置上。

#### 5. ELF 可执行文件的加载

加载文件最重要的是完成两件事：加载数据段和程序段到内存；重定位外部定义符号。具体如下。

内核首先读取驱动文件的头部，根据头部的数据指示分别读入各种数据结构，找到标记为可加载（loadable）的段，并调用函数 `mmap()` 把段内容加载到内存中；内核分析出驱动文件标记为 `PT_INTERP` 的段中所对应的动态链接器名称，加载动态链接器。内核把控制传递给动态链接器。

动态链接器检查程序对外部文件（共享库）的依赖性，在需要时对其进行加载；动态链接器对程序的外部引用进行重定位，即说明其引用的外部变量 / 函数的地址，此地址位于共享库被加载在内存的区间内。动态链接还有一个延迟（Lazy）定位的特性，即只在“真正”需要引用符号时才重定位，这有利于提高程序运行效率。动态链接器执行在内核驱动文件中标记为 .init 节的代码，进行程序运行的初始化，然后把控制传递给程序，从内核驱动文件头部中定义的程序进入点开始执行。

## 8.2 main 函数的识别

对可执行程序进行反汇编操作，获取程序的控制流图，是实现软件维护、代码理解、程序安全性分析等目标的重要步骤。在由 C 或 C++ 语言编写的恶意程序中，程序的主入口函数为用户自定义的函数 `main()`。对这类可执行程序进行反汇编操作的一种常用思路，是以 `main()` 函数为主入口点对二进制程序进行反汇编，再在反汇编代码上展开一系列的分析工作。目前，获取 `main()` 函数地址的常用方法是查找程序的符号表。但是，当待分析的程序中不包含符号信息时，这种方法将无法正确定位程序中 `main()` 函数的地址。可见，基于符号表定位二进制程序中 `main()` 函数地址的方法不具有普适性。

为了解决该问题，本章在对可执行程序的执行过程进行分析的基础上，提出了一种基



于模板匹配的可执行程序的主入口点定位方法。由于该方法不依赖于程序中的符号表信息，因而对二进制代码分析具有更强的适用性。

### (1) 定位 main() 函数的必要性

二进制代码分析技术是一种用于分析程序内容和结构的技术。该技术被广泛应用于二进制修改、二进制翻译、二进制代码匹配、计算机安全等众多研究领域。在理想情况下，二进制代码分析需要提供关于程序代码的信息（包括指令、基本块、函数等）、程序的结构信息（包括控制流和数据流等），以及数据结构信息等（全局变量、内存变量等）。这些信息的质量和可用性将直接影响依赖于二进制分析技术的相关工具的应用效果。

二进制代码分析的第一步是反汇编，即通过指令解码将二进制流转化成机器指令流。反汇编操作的第一步则是确定二进制可执行程序的主入口点。解码主入口点的选择有两种可能，一种是以二进制文件执行的第一条指令位置为解码的主入口点；第二种则是以程序中主函数 main() 的地址为解码的主入口点。对程序的加载与执行过程进行动态跟踪可以发现，二进制程序执行的第一条指令的地址并不是 main() 函数的地址，而是一些由编译器添加的初始化函数的地址。这些初始化函数中往往存在大量的间接过程调用，并且其内容往往与体系结构密切相关。如果以这些函数为主入口点进行解码，则无疑会增加反汇编算法实现的难度，从而降低二进制代码分析工具的整体效率。因此，许多二进制代码分析技术都选择以程序的主函数 main() 作为程序解码的主入口地址。

除此以外，定位程序中主函数 main() 也是许多二进制代码分析技术应用的需要。首先，在二进制翻译等二进制代码分析技术的应用中必须准确地定位 main() 函数，正确恢复程序的主函数 main() 是翻译后生成的可执行程序能够正确运行的重要条件之一。其次，许多二进制代码分析都是在控制流图上完成的，而作为控制流图的根节点的 main() 函数，其正确定位将有助于构建完整的控制流图。最后，在关注用户行为分析的二进制代码分析技术应用中，找到用户定义的主函数 main() 也有助于更加高效地理解用户行为。

概括而言，从 main() 函数开始对可执行程序展开分析工作的必要性体现在以下几个方面：首先，由于分析的是真正用户定义的行为，因而分析的针对性更强；其次，由于不用分析系统函数或库函数，因而降低了分析的难度，也从一定程度上提高了分析的准确性；最后，由于不用分析系统函数，因而提高了对程序进行分析的效率。

由此可见，定位二进制可执行程序中函数 main() 的地址，对于可执行程序分析而言是迫切而必要的。

### (2) 传统定位方法存在的问题

目前，获取 main() 函数地址的常用方法是查找程序的符号表。以 IA32/Linux 平台下的 ELF 程序为例。使用 Objdump 工具获得一段示例程序的符号表（Symbol Table）信息，可知符号表中与主函数 main() 对应的地址为 0x80483f0；通过与反汇编结果的比较可确定，该地址即为 main() 函数的入口地址。示例程序的符号表及代码段反汇编结果的部分信息如图 8-6 所示。



```

SYMBOL TABLE:
080480f4 l d .interp      00000000
...
08048300 g F .text        00000000  _start
08049590 g *ABS*          00000000  __bss_start
080483f0 g F .text00000024  main
...

Disassembly of section .text:
...

080483f0 <main>:
80483f0: 55                push %ebp
80483f1: 89 e5             mov %esp,%ebp
80483f3: 83 ec 08          sub $0x8,%esp
...

```

图 8-6 基于符号表的 main() 函数地址定位方法

然而，通过读取符号表中与符号“main”相关的地址以获取程序入口点的方法却存在以下缺陷。

首先，该方法不适用于分析缺少符号表相关信息的 stripped 可执行程序。

去除符号以及调试等信息的可执行程序（简称 stripped 可执行程序），可将其定义为一类缺少关于位置、大小、函数及对象展开等相关信息的可执行程序。由于与符号有关的符号表等信息并不存在，因而查找符号表定位 main() 函数地址的方法并不可行。

目前，二进制代码分析人员面对的二进制可执行代码大多是 stripped 可执行程序。一些系统库的发布版常常是 strip 后的版本，这是因为 strip 后的程序与 strip 前相比，体积更小，因而能够减少库文件对磁盘空间的需求量；一些商业软件的发布版也是 stripped 可执行程序，因为对可执行程序进行 strip 操作减少了程序中可用于分析的信息，从而增加了对其进行逆向分析的难度，可用于防止对软件的未授权使用；一些恶意软件编写者也会使用相应的工具去除其编写的程序中与符号有关的相应内容，其目的是阻碍安全防护软件对其进行的分析检测，从而阻止其被安全防护软件查杀。

产生 stripped 程序一般存在两种情况。一种情况是当符号表或调试符号等符号信息内嵌在可执行程序的二进制代码内时，可对该程序使用相关工具，以实现去除符号表等相关信息的目的。比如 GNU 提供的二进制工具 STRIP，可以被用来去除 ELF 格式的目标文件或指定文件的调试符号表信息，根据其选项开关设置的不同，该工具可用于去除不同的信息。

另一种情况则是发生在符号等信息存在于可执行程序体外的其他文件中时。由于符号表、调试信息等内容存在于二进制代码内会造成程序体积的急剧膨胀，因此，有些编译器，包括一些早期的主流调试系统，会将程序中的符号信息从可执行程序体中剥离出来，放在一个独立的文件中。此时，可以通过在编译链接的过程中设置相应的选项开关以去除相关文件中的符号信息。以微软的编程环境 Microsoft Visual Studio 为例。在默认配置下，编译生成可执行程序的同时会产生一个后缀名为 .pdb 的程序数据库文件，与程序有关的符号等



信息实际上存放在该文件中。通常，程序数据库文件中包含以下信息：公共符号、对象文件列表 FPO 调试记录、类型信息以及 CodeView 符号等，其中 CodeView 符号中包括函数名、变量名以及静态数据。在大多数情况下，程序开发人员不希望用户看到自己定义的符号等信息，即所谓的私有符号（private symbol）。此时，在编译链接程序的时候选用开关“/PDBSTRIPPED”即可去除程序数据库文件中的类型信息、CodeView 符号等私有符号信息。因此，当 PDB 文件缺失或是链接时使用了选项“/PDBSTRIPPED”，此时的可执行程序即可看作 stripped 可执行程序。

其次，该方法不适用于分析恶意程序，包括病毒、蠕虫，以及移动代码等。

造成这一现象的原因有两点。第一，许多潜在的恶意程序中并不存在符号表和调试信息，以避免被安全防护软件查杀；第二，即使程序中存在符号表等相关信息，在有些情况下这些信息也并不可信，因为这些信息存在被篡改的可能。

由以上分析可见，传统的基于符号表的 main() 函数定位方法不适用于分析大多数的可执行程序，需要针对 stripped 可执行程序的特点，设计适用性更强的 main() 函数定位方法。

另外，需要特殊说明的是，考虑到编译器使用的普遍性，本章中研究的 PE 可执行程序主要针对由 Microsoft Visual Studio 编译生成的 EXE 应用程序，暂不考虑 DLL 可执行程序；ELF 可执行程序则主要针对由 GCC 编译生成的可执行程序，默认优化选项为“-O2”。

### 8.2.1 程序启动过程分析

一个 PE 应用程序从双击开始（当然也可以有其他启动方法）的执行过程可以简单地划分为五步：进程创建、主线程创建、PE 文件加载、程序的初始化函数启动以及用户定义的 main() 执行。具体而言，从双击一个应用程序到该应用程序执行完毕返回的执行流程如下：

1) 由 shell 调用系统函数 CreateProcess() 以激活应用程序。shell 即命令解释器，是 Windows32 操作系统基于浏览器的一个 32 位用户接口，可将其看作所有应用程序进程的父进程。shell 完成的工作包括启动应用程序、管理文件系统、关联应用程序和相应文件等。

2) 系统产生一个进程核心对象（process kernel object），计数值为 1。核心对象可以理解为系统的一种资源，包括用于内存映射文件的核心对象以及用于执行线程同步化的对象等。系统赋予核心对象计数值的目的是为了便于管理，该值表示的是核心对象被使用的次数，使用 1 次计数值加 1。

3) 系统为此进程建立大小为 4GB 的地址空间。

4) 加载器完成程序加载功能，加载的内容包括应用程序的代码和数据、应用程序需要用到的运行环境、程序运行所需的动态链接函数库等。

5) 系统为该进程建立主执行线程（primary thread）。

6) 系统调用 C/C++ 运行时函数库（C/C++ runtime library）的启动代码（startup code），该代码执行相关的安全性检查，完成中断处理、全局变量初始化，以及内存分配函数和构造函数的初始化等功能。



7) C/C++ 运行时库启动代码调用应用程序的主入口函数 `main()`。

8) 应用程序开始执行。

9) 应用程序执行完成后, 回到 C/C++ 运行时库启动代码。

10) 回到系统, 调用系统函数 `ExitProcess()` 结束进程。

分析以上流程可以发现, 对主入口函数 `main()` 的调用是由运行时库启动代码完成的。实际上, PE 文件头中给出的程序入口点正是指向这段启动代码的起始位置。由于这段启动代码是由链接器在编译程序的过程中自动添加到程序的开始位置的, 因此, 该代码相对于用户编写的程序而言应该较为固定。因此, 对这段代码的深入分析, 将有助于从这段代码中找到定位函数 `main()` 地址信息的方法。

需要特殊说明的是, 为了便于分析, 将这段由链接器添加的、用于完成启动和初始化功能的函数称为 `startup` 函数。

ELF 格式可执行程序的执行过程与 PE 格式可执行程序的执行过程相近, 其主要流程也包括程序加载、动态链接器加载、主程序运行等。由于本章关注的是可执行程序中 `main()` 函数的定位问题, 因此分析的重点在于动态链接器的加载及执行过程。

在程序加载完成后, 系统会将控制权交给动态链接器, 接下来调用 `_dl_start()` 函数以获得程序的入口地址。ELF 格式可执行程序的文件头中给出了程序执行的入口地址, 即程序中第一条执行的指令的地址。使用 `objdump` 等二进制工具进行分析可以发现, 该地址并不指向函数 `main()`, 而是指向 ELF 程序中 `.text` 代码段的起始位置, 也是共享对象初始化函数 `_start()` 的起始位置。`_start()` 函数会将某些参数压栈, 然后调用函数 `_libc_start_main()`, 该函数完成运行程序的初始化功能, 包括为动态链接器提供析构函数等, 并最终将控制权交给用户程序入口 `main()` 函数。

由以上分析过程可以发现, 无论是何种格式的可执行程序, 在用户定义的入口函数 `main()` 执行之前, 已经执行了一系列的函数调用。这些函数由链接器添加到程序中, 因而其代码较为固定。因此, 只要能够分析出这些启动函数的特征, 就能够从这些初始化代码中找到定位 `main()` 地址的信息。同时也将这些启动函数称为 `startup` 函数。

### 8.2.2 startup 函数解析

Microsoft Visual C++ 在其 C/C++ 运行时库中提供了四个版本的 `startup` 函数, 分别对应四种类型的应用程序, 具体说明见表 8-2 所示。

表 8-2 初始化函数及其对应的用户程序的入口函数

startup 函数名称	用户程序的主入口函数	用户程序特征描述
<code>mainCRTStartup()</code>	<code>main()</code>	需要 ASCII 字符和字符串的 CUI 应用程序
<code>wmainCRTStartup()</code>	<code>wmain()</code>	需要 Unicode 字符和字符串的 CUI 应用程序
<code>WinMainCRTStartup()</code>	<code>winMain()</code>	需要 ASCII 字符和字符串的 GUI 应用程序
<code>wWinMainCRTStartup()</code>	<code>wWinMain()</code>	需要 Unicode 字符和字符串的 GUI 应用程序



Microsoft Visual C++ 关于启动函数的宏定义见图 8-7 所示。由该定义可知，无论采用四种 startup 函数中的哪一种，程序入口点处实际执行的函数为 `_tmainCRTStartup`，只是在不同类型的应用程序中该函数的命名稍有不同。与此类似，编译器也常常将以上四种用户程序的主入口函数统一命名为 `_tmain`。

启动函数 `_tmainCRTStartup` 的定义见图 8-8 所示。`__security_init_cookie()` 函数主要完成与缓冲区溢出有关的一些安全检查功能，而运行时库的初始化功能大部分都通过 `__tmainCRTStartup()` 函数完成。

函数 `__tmainCRTStartup()` 的部分源代码见图 8-9 所示。运行时库的初始化功能实际上是由函数 `__tmainCRTStartup()` 完成的，该函数会在初始化工作完成后调用主入口函数 `_tmain()`，并在 `_tmain()` 执行结束后将 `_tmain()` 的返回值返回。

```
#ifdef _WINMAIN_
#ifdef WPRFLAG

#define _tmainCRTStartup wWinMainCRTStartup
#else /* WPRFLAG */
#define _tmainCRTStartup WinMainCRTStartup
#endif /* WPRFLAG */

#else /* _WINMAIN_ */

#ifdef WPRFLAG
#define _tmainCRTStartup wmainCRTStartup
#else /* WPRFLAG */
#define _tmainCRTStartup mainCRTStartup
#endif /* WPRFLAG */
#endif /* _WINMAIN_ */
```

图 8-7 Microsoft Visual C++ 中关于启动函数的部分源代码

```
int __tmainCRTStartup( void )
{
    __security_init_cookie();
    return __tmainCRTStartup();
}
```

图 8-8 启动函数 `_tmainCRTStartup` 的源代码

```
__declspec(noinline)
int __tmainCRTStartup( void )
{
    unsigned int osplatform = 0;
    ...
#ifdef _WINMAIN_
    lpszCommandLine = _twincmdln();
    mainret = _tWinMain( (HINSTANCE)&__ImageBase,
        NULL,
        lpszCommandLine,
        StartupInfo.dwFlags & STARTF_USESHOWWINDOW
            ? StartupInfo.wShowWindow
            : SW_SHOWDEFAULT
        );
#else /* _WINMAIN_ */
    _tinitenv = _tenviron;
    mainret = _tmain(__argc, __argv, _tenviron);
#endif /* _WINMAIN_ */
    ...
    return mainret;
}
```

图 8-9 函数 `__tmainCRTStartup()` 的部分源代码



从以上代码分析的结果可知程序从入口处的 startup 函数开始直到主函数 `_tmain()` 被调用的调用轨迹如图 8-10 所示。

对于 ELF 格式的可执行程序，其 startup 函数调用主函数 `main()` 的调用轨迹与 PE 格式可执行程序不同。ELF 文件头中给出的程序的入口点地址指向函数 `_start()`，该函数由链接器在编译时添加到可执行程序的头部。函数 `_start()` 的代码如图 8-11 所示。

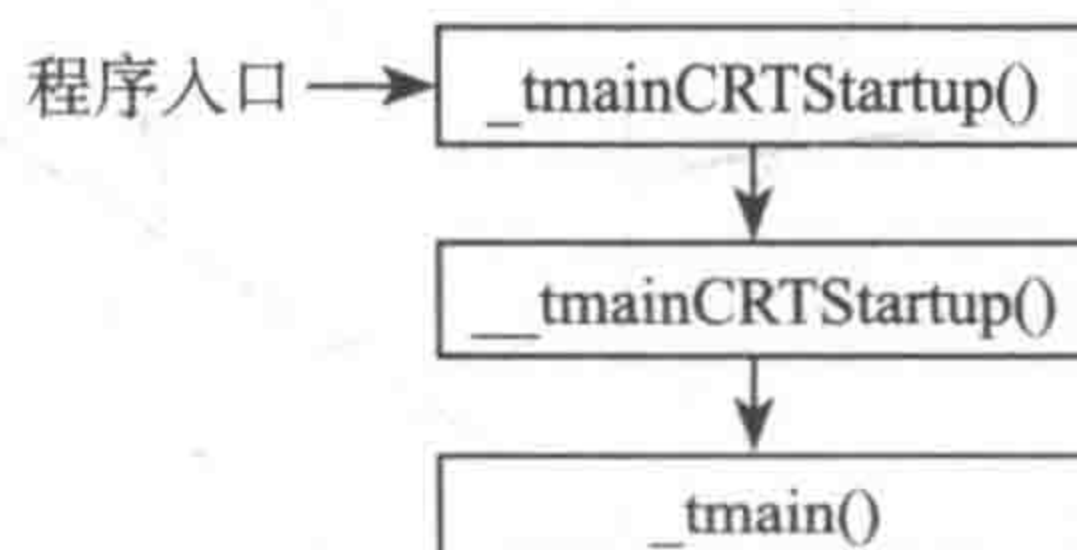


图 8-10 PE 程序 startup 函数调用主函数的过程示意图

`_start()` 函数在调用函数 `__libc_start_main()` 之前，会执行一系列的压栈操作。其中，最后三次压栈的数据，依次为主函数 `main()` 的第二个入参、第一个入参以及 `main()` 函数的地址。接下来，执行函数 `__libc_start_main()` 的代码，完成对 `libc` 库的初始化、动态链接器的析构器的准备、`main()` 函数入参的准备等工作，然后通过语句：

```

.text
.globl _start
_start:
    xorl %ebp, %ebp
    popl %esi /* 参数个数弹出 */
    movl %esp, %ecx
    andl $0xffffffff8, %esp
    pushl %eax
    pushl %esp
    pushl %edx
    pushl $_fini /* 将函数 _fini 和 _init 的入口地址入栈 */
    pushl $_init
    pushl %ecx /* 将 main() 的第二个入参 argv 入栈 */
    pushl %esi /* 将 main() 的第一个入参 argc 入栈 */
    pushl $main /* 将用户自定义入口函数 main 的地址入栈 */
    call __libc_start_main /* 调用函数 __libc_start_main, 该函数
                           调用 main 函数, 并将其返回值返回 */
    hlt
  
```

图 8-11 `_start()` 函数的源代码

```
exit (stinfo->main (argc, argv, __environ, auxvec));
```

调用用户自定义程序入口函数 `main()`。函数 `__libc_start_main()` 的部分源代码如图 8-12 所示。

```

int __libc_start_main (int argc, char **argv, char
**envp,
    void *auxvec, void (*rtld_fini) (void),
    struct startup_info *stinfo,
    char **stack_on_entry)
{
  
```

图 8-12 `__libc_start_main()` 函数的部分源代码



```

if (*stack_on_entry != NULL)
{
    argc = *(int *) stack_on_entry;
    argv = stack_on_entry + 1;
    envp = argv + argc + 1;
    ...
}
__libc_stack_end = stack_on_entry + 4;
__environ = envp;
...; /* 完成一系列的初始化工作，包括替动态链接器和程序自身安排
      destructor，完成程序初始化等功能，最后调用主函数 main() */
exit (stinfo->main (argc, argv, __environ, auxvec));
}

```

图 8-12 (续)

从以上代码分析的结果可知程序从入口处的 startup 函数开始直到主函数 main() 被调用的调用轨迹如图 8-13 所示。

### 8.2.3 main() 函数定位

从以上调用过程可以看出，定位 PE 可执行程序中的 main() 函数可由以下三步骤实现：

- 1) 通过分析 PE 程序文件头信息获取入口点信息，得到函数 \_tmainCRTStartup() 的地址。
- 2) 通过分析函数 \_tmainCRTStartup() 的汇编代码，获取函数 \_\_tmainCRTStartup() 的地址。
- 3) 通过分析函数 \_\_tmainCRTStartup() 的汇编代码，从中得出主函数 main() 的地址信息。

步骤一可以通过读取文件头 Optional Header 的 AddressOfEntryPoint 字段实现，具体细节可参见 PE 格式文档。

步骤二可以通过定义指令模板采用模板匹配技术实现。由函数 \_tmainCRTStartup() 的源代码可知，该函数体中仅有两次函数调用行为，两次调用都没有参数传递，且第二次调用后直接返回。而在汇编代码中，实现函数调用一般有两种方式，一种是通过调用指令 CALL 实现，另一种则是在进行尾调优化后由跳转指令 JMP 实现。因此，可定义如下两个指令模板：

**模板一：** CALL; CALL

**模板二：** CALL; JMP

对函数 \_tmainCRTStartup() 进行反汇编，并对反汇编结果执行模板匹配操作。一旦两个模板中有一个匹配成功，则两个模板中第二条指令的目标地址即为函数 \_\_tmainCRTStartup() 的入口地址。

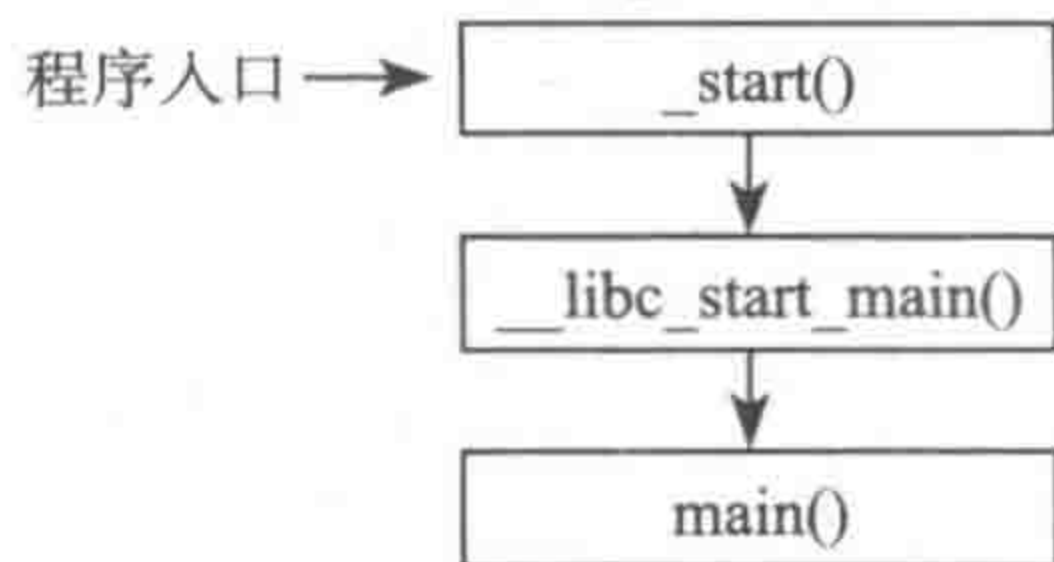


图 8-13 ELF 程序 startup 函数调用主函数的过程示意图



步骤三的实现仍然需要借助于模板匹配技术。经过对大量 PE 程序中 `__tmainCRTStartup()` 函数的反汇编代码进行总结可以发现,在大多数情况下,函数 `__tmainCRTStartup()` 会通过 3 条压栈指令和 1 条调用指令实现对 `main()` 函数的调用,参照 VC 8.0 的源代码可知,3 条压栈指令分别对应于 `main()` 函数的三个入参: `__argc`、`_targv` 和 `_tenviron`,最后由 `CALL` 指令调用函数 `main()`。因此,以下指令模板可用于识别 `__tmainCRTStartup()` 函数对函数 `main()` 的调用。

**模板三:** `PUSH; PUSH; PUSH; CALL`

一旦模板三匹配成功,则模板三中最后一条指令 `CALL` 的目标地址即为主入口函数 `main()` 的地址。

如果参照分析 PE 程序中 `main()` 函数地址的方法,则分析 ELF 程序中 `main()` 函数的地址可以按照以下步骤进行:

- 1) 通过分析 ELF 程序的文件头信息获取入口地址,得到函数 `_start()` 的地址。
- 2) 通过分析函数 `_start()` 的汇编代码,获取函数 `__libc_start_main()` 的地址。
- 3) 通过分析函数 `__libc_start_main()` 的汇编代码,从中得出主函数 `main()` 的地址信息。

以上步骤中,步骤一的实现比较容易,读取文件头 ELF Header 的 `e_entry` 字段即可,具体细节可参见 ELF 格式文献。

步骤二的实现则相对复杂,其主要原因在于编译器采用的惰性绑定 (`lazy binding`) 机制。当动态链接器采用惰性绑定方式时,确定某个库函数的具体地址是在该程序真正被调用的时候才进行的。因此,在可执行程序的二进制代码中并不包含函数 `__libc_start_main()` 的代码,该函数的代码只有当程序真正运行时才会由动态链接器加载到内存中执行。实际上,反汇编结果显示,函数 `_start()` 调用函数 `__libc_start_main()` 时的调用指令处给出的调用地址指向的是 ELF 文件中 PLT 段中该函数的函数描述符所在位置,并不是 `__libc_start_main()` 函数的入口地址,因此,要想通过静态分析得出库函数 `__libc_start_main()` 的地址,并从中分析出函数 `main()` 的地址,并不是一种可行的方法。

然而,对 `_start()` 函数的特征进一步分析可以发现, `_start()` 函数存在两个重要的特征:

**特征 1:** 函数 `_start()` 的源代码中只有唯一的一次函数调用行为。

**特征 2:** 函数 `_start()` 传递给函数 `__libc_start_main()` 若干个参数,其中最后一个参数为函数 `main()` 的地址。

其中,特征 1 表明 `_start()` 函数中只会有一条 `CALL` 指令,特征 2 表明 `_start()` 函数中 `CALL` 指令的前一条指令的操作数是主入口函数 `main()` 的入口地址。因此,我们同样可以用基于模板匹配的方法实现对 ELF 可执行程序中主入口函数 `main()` 的定位。定义模板如下:

**模板四:** `CALL`

在对函数 `_start()` 的反汇编过程中,一旦出现匹配该指令模板的指令,则提取该指令的前一条指令的操作数,并将该操作数记为主入口函数 `main()` 的入口地址。

基于模板匹配实现 ELF 可执行程序中主入口函数 `main()` 的定位是一种简单可行的方

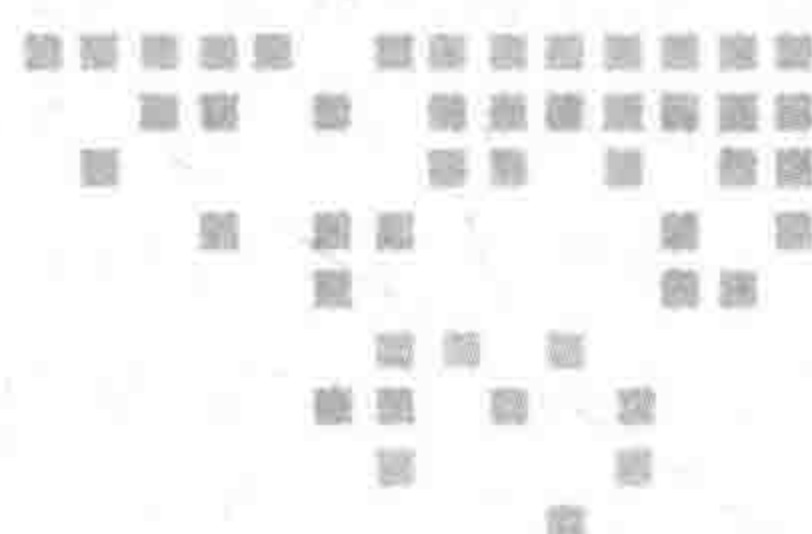


法。从理论上来说,该方法也具备一定的有效性。首先,由于 `_start()` 函数中只会出现一条 `CALL` 调用指令,因此在该函数的反汇编代码执行模板匹配操作时不会出现多条指令匹配成功的情况。其次,虽然在函数 `_start()` 调用函数 `__libc_start_main()` 之前会有多次压栈操作,但是函数 `_start()` 对函数 `__libc_start_main()` 的调用行为是紧接在对 `main()` 函数地址的压栈操作之后进行的,因此不会出现因为有多压栈操作而无法决定取哪个操作数的情况。

### 8.3 本章小结

本章介绍了可执行文件的两种格式,一种是 Windows 平台上的 PE 格式,另一种是 Linux 平台上的 ELF 格式。本章还对 `main` 函数的识别问题进行了阐述。





## 反编译的基础——指令系统和反汇编

### 9.1 指令系统概述

指令系统是计算机硬件的语言系统，也叫机器语言，指机器所具有的全部指令的集合，它是软件和硬件的主要界面，反映了计算机所拥有的基本功能。从系统结构的角度看，它是系统程序员看到的计算机的主要属性。因此指令系统表征了计算机的基本功能并决定了机器所具有的能力，也决定了指令的格式和机器的结构。设计指令系统就是要选择计算机系统（包括操作系统和高级语言中的）一些基本操作应由硬件实现还是由软件实现，选择某些复杂操作是由一条专用的指令实现还是由一串基本指令实现，然后具体确定指令系统的指令格式、类型、操作以及对操作数的访问方式。

指令系统是指计算机所能执行的全部指令的集合，它描述了计算机内全部的控制信息和“逻辑判断”能力。不同计算机的指令系统包含的指令种类和数目也不同。一般均包含算术运算型、逻辑运算型、数据传送型、判定和控制型、移位操作型、位（位串）操作型、输入和输出型等指令。指令系统是表征一台计算机性能的重要因素，它的格式与功能不仅直接影响机器的硬件结构，而且也直接影响系统软件、机器的适用范围。

指令系统的发展经历了从简单到复杂的演变过程。早在 20 世纪五六十年代，计算机大多数采用分立元器件的晶体管或电子管组成，其体积庞大，价格也很昂贵，因此计算机的硬件结构比较简单，所支持的指令系统也只有十几至几十条最基本的指令，而且寻址方式简单。

到 20 世纪 60 年代中期，随着集成电路的出现，计算机的功耗、体积、价格等不断下降，硬件功能不断增强，指令系统也越来越丰富。

在 20 世纪 70 年代，高级语言已成为大、中、小型机的主要程序设计语言，计算机应用日益普及。由于软件的发展超过了软件设计理论的发展，复杂的软件系统设计一直没有



很好的理论指导，导致软件质量无法保证，从而出现了所谓的“软件危机”。人们认为，缩小机器指令系统与高级语言语义差距，为高级语言提供很多的支持，是缓解软件危机有效和可行的办法。计算机设计者们利用当时已经成熟的微程序技术和飞速发展的 VLSI 技术，增设各种各样的复杂的、面向高级语言的指令，使指令系统越来越庞大。这是几十年来人们在设计计算机时，保证和提高指令系统有效性方面传统的想法和做法。

### （1）复杂指令系统

早期的计算机中存储器是一个很昂贵的资源，因此人们希望指令系统能支持生成最短的程序，还希望程序执行时所需访问的程序和数据位的总数越少越好。在微代码出现后，以前由一串指令所完成的功能转移到了微代码中，从而改进了代码密度；它也避免了从主存取指令的较慢动作，提高了执行效率。在微代码中实现功能的另一论点是：这些功能可以较好地支持编译程序。如果一条高级语言的语句能被转换成一条机器语言指令，这可使编译软件的编写变得非常容易。此外，在机器语言中含有类似高级语言的语句指令，便可使机器语言与高级语言的间隙减少。这种发展趋向导致了复杂指令系统（CISC）设计风格的形成，即认为计算机性能的提高主要依靠增加指令复杂性及其功能来获取。

CISC 的主要特点是：

1) 指令系统复杂。具体表现在以下几个方面：

- ① 指令数多，一般大于 100 条。
- ② 寻址方式多，一般大于 4 种。
- ③ 指令格式多，一般大于 4 种。

2) 绝大多数指令需要多个机器时钟周期方可执行完毕。

3) 各种指令都可以访问存储器。

CISC 主要存在如下 3 方面问题：

1) CISC 中各种指令的使用频度相差很悬殊，大量的统计数字表明，大约有 20% 的指令使用频度比较高，占据了 80% 的处理器时间。换句话说，有 80% 的指令只在 20% 的处理器运行时间内才被用到。

2) VLSI 的集成度迅速提高，使得生产单芯片处理器成为可能。在单芯片处理器内，希望采用规整的硬布线控制逻辑，不希望用微代码。而在 CISC 处理器中，大量使用微代码技术以实现复杂的指令系统，给 VLSI 工艺造成很大困难。

3) 虽然复杂指令简化了目标程序，缩小了高级语言与机器指令之间的语义差距，然而增加了硬件的复杂程度，会使指令的执行周期大大加长，从而有可能使整个程序的执行时间增加。

### （2）精简指令系统

由于 CISC 技术在发展中出现了问题，计算机系统结构设计的先驱者们尝试从另一条途径来支持高级语言及适应 VLSI 技术特点。1975 年 IBM 公司 John Cocke 提出了精简指令系统（RISC）的设想。到了 1979 年，美国加州大学伯克莱分校由 Patterson 教授领导的研究组



首先提出了 RISC 这一术语,并先后研制了 RISC-I 和 RISC-II 计算机。1981 年美国的斯坦福大学在 Hennessy 教授领导下的研究小组研制了 MIPS RISC 计算机,强调高效的流水和采用编译方法进行流水调度,使得 RISC 技术设计风格得到很大补充和发展。

20 世纪 90 年代初,IEEE 的 Michael Slater 对于 RISC 的定义做了如下描述:RISC 处理器所设计的指令系统应使流水线处理能高效率执行,并使优化编译器生成优化代码。

RISC 为使流水线高效率执行,应具有下述特征:

- 1) 简单而统一格式的指令译码。
- 2) 大部分指令可以单周期执行完成。
- 3) 只有 LOAD 和 STORE 指令可以访问存储器。
- 4) 简单的寻址方式。
- 5) 采用延迟转移技术。
- 6) 采用 LOAD 延迟技术。

RISC 为使优化编译器便于生成优化代码,应具有下述特征:

- 1) 三地址指令格式。
- 2) 较多的寄存器。
- 3) 对称的指令格式。

RISC 的主要问题是编译后生成的目标代码较长,占用了较多的存储器空间。但由于半导体集成技术的发展,使得 RAM 芯片集成度不断提高和成本不断下降,目标代码较长已不成为主要问题。RISC 技术存在另一个潜在缺点,即对编译器要求较高,除了常规优化方法外,还要进行指令顺序调度,甚至能替代通常流水线中所需的硬件联锁功能。

指令系统的性能决定了计算机的基本功能,它的设计直接关系到计算机的硬件结构和用户的需要。一个完善的指令系统应满足如下四方面的要求:

- 完备性。指用汇编语言编写各种程序时,指令系统直接提供的指令足够使用,而不必用软件来实现。完备性要求指令系统丰富、功能齐全、使用方便。
- 有效性。指利用该指令系统所编写的程序能够高效率地运行。高效率主要表现在程序占据存储空间小、执行速度快。
- 规整性。包括指令系统的对称性、匀齐性、指令格式和数据格式的一致性。对称性是指在指令系统中所有的寄存器和存储器单元都可同等对待,所有的指令都可使用各种寻址方式;匀齐性是指一种操作性质的指令可以支持各种数据类型;指令格式和数据格式的一致性是指指令长度和数据长度有一定的关系,以方便处理和存取。
- 兼容性。至少要能做到“向上兼容”,即低档机上运行的软件可以在高档机上运行。

### 9.1.1 机器指令及格式

#### (1) 机器指令

机器指令是 CPU 能直接识别并执行的指令,它的表现形式是二进制编码。机器指令通



常由操作码和操作数两部分组成，操作码指出该指令所要完成的操作，即指令的功能；操作数指出参与运算的对象，以及运算结果所存放的位置等。

由于机器指令与 CPU 紧密相关，所以，不同种类的 CPU 所对应的机器指令也就不同，而且它们的指令系统往往相差很大。但对于同一系列的 CPU 来说，为了使各型号之间具有良好的兼容性，要做到新一代 CPU 的指令系统必须包括先前同系列 CPU 的指令系统。只有这样，先前开发出来的各类程序在新一代 CPU 上才能正常运行。

用机器语言编写程序是早期经过严格训练的专业技术人员的工作，普通的程序员一般难以胜任，而且用机器语言编写的程序不易读、出错率高、难以维护，也不能直观地反映用计算机解决问题的基本思路。

由于用机器语言编写程序有以上诸多的不便，几乎没有程序员这样编写程序了。以下是一些示例。

#### 指令部分的示例

```
0000 代表 加载 (LOAD)
0001 代表 存储 (STORE)
...
```

#### 寄存器部分的示例

```
0000 代表寄存器 R1
0001 代表寄存器 R2
...
```

#### 存储器部分的示例

```
000000000000 代表地址为 0 的存储器
000000000001 代表地址为 1 的存储器
000000010000 代表地址为 16 的存储器
100000000000 代表地址为  $2^{11}$  的存储器
```

#### 集成示例

```
0000,0000,000000010000 代表 LOAD R1, 16
0000,0001,000000000001 代表 LOAD R2, 1
0001,0001,000000010000 代表 STORE R2, 16
0001,0001,000000000001 代表 STORE R2, 1
```

#### (2) 指令格式

计算机的指令格式与机器的字长、存储器的容量及指令的功能都有很大的关系。从便于程序设计、增加基本操作并行性、提高指令功能的角度来看，指令中应包含多种信息。但在有些指令中，由于部分信息可能无用，这将浪费指令所占用的存储空间，并增加访存次数，也许反而会影响速度。因此，如何合理、科学地设计指令格式，使指令既能给出足够的信息，又使其长度尽可能地与机器的字长相匹配，以节省存储空间，缩短取值时间，提高机器的性能，这是指令格式设计中的一个重要问题。



计算机是通过执行指令来处理各种数据的。为了指出数据的来源、操作结果的去向及所执行的操作，一条指令必须包含下列信息：

- 1) 操作码。它具体说明了操作的性质及功能。一台计算机可能有几十条至几百条指令，每一条指令都有一个相应的操作码，计算机通过识别该操作码来完成不同的操作。
- 2) 操作数的地址。CPU 通过该地址就可以取得所需的操作数。
- 3) 操作结果的存储地址。把对操作数的处理所产生的结果保存在该地址中，以便再次使用。
- 4) 下一条指令的地址。执行程序时，大多数指令按顺序依次从主存中取出执行，只有在遇到转移指令时，程序的执行顺序才会改变。为了压缩指令的长度，可以用一个程序计数器 PC 存放指令地址。每执行一条指令，PC 的指令地址就自动加 1（设该指令只占一个主存单元），指出将要执行的下一条指令的地址。当遇到执行转移指令时，则用转移地址修改 PC 的内容。由于使用了 PC，指令中就不必明显地给出下一条将要执行指令的地址。

一条指令实际上包括两种信息，即操作码和地址码。操作码用来表示该指令所要完成的操作（如加、减、乘、除、数据传送等），其长度取决于指令系统中的指令条数。

地址码用来描述该指令的操作对象，它或者直接给出操作数，或者指出操作数的存储器地址或寄存器地址（即寄存器名）。

指令包括操作码域和地址域两部分。根据地址域所涉及的地址数量，常见的指令格式有以下几种。

- 1) 三地址指令：一般地址域中 A1、A2 分别确定第一、第二操作数地址，A3 确定结果地址。下一条指令的地址通常由程序计数器按顺序给出。
- 2) 二地址指令：地址域中 A1 确定第一操作数地址，A2 同时确定第二操作数地址和结果地址。
- 3) 单地址指令：地址域中 A 确定第一操作数地址。固定使用某个寄存器存放第二操作数和操作结果。因而在指令中隐含了它们的地址。
- 4) 零地址指令：在堆栈型计算机中，操作数一般存放在下推堆栈顶的两个单元中，结果又放入栈顶，地址均被隐含，因而大多数指令只有操作码而没有地址域。
- 5) 可变地址数指令：地址域所涉及的地址的数量随操作定义而改变。如有的计算机指令中的地址数可少至 0 个，多至 6 个。

### 9.1.2 汇编指令及描述

汇编指令是汇编语言中使用的一些操作符和助记符，还包括一些伪指令（如 `assume`、`end`），即用于告诉汇编程序如何进行汇编的指令，它既不控制机器的操作也不被汇编成机器代码，只能为汇编程序所识别并指导汇编如何进行。

#### (1) 寻址方式

根据指令内容确定操作数地址的过程称为寻址。完善的寻址方式可为用户组织和使用



数据提供方便。

1) 直接寻址: 指令地址域中表示的是操作数地址。

2) 间接寻址: 指令地址域中表示的是操作数地址的地址, 即指令地址码对应的存储单元所给出的是地址 A, 操作数据存放在地址 A 指示的主存单元内。有的计算机的指令可以多次间接寻址, 如 A 指示的主存单元内存放的是另一地址 B, 而操作数据存放在 B 指示的主存单元内, 这称为多重间接寻址。

3) 立即寻址: 指令地址域中表示的是操作数本身。

4) 变址寻址: 指令地址域中表示的是变址寄存器号  $i$  和位移值  $D$ 。将指定的变址寄存器内容  $E$  与位移值  $D$  相加, 其和  $E+D$  为操作数地址。许多计算机具有双变址功能, 即将两个变址寄存器内容与位移值相加, 得操作数地址。变址寻址有利于数组操作和程序共用。同时, 位移值长度可短于地址长度, 因而指令长度可以缩短。

5) 相对寻址: 指令地址域中表示的是位移值  $D$ 。程序计数器内容 (即本条指令的地址)  $K$  与位移值  $D$  相加, 得操作数地址  $K+D$ 。当程序在主存储器浮动时, 相对寻址能保持原有程序功能。

此外, 还有自增寻址、自减寻址、组合寻址等寻址方式。寻址方式可由操作码确定, 也可在地址域中设标志, 指明寻址方式。

## (2) 指令分类

随着计算机系统结构的发展, 有些计算机还不断引入新指令。如“测并置”指令是为在多机系统和多道程序中防止重入公用子程序而设置的。指令先测试标志位以判断该子程序是否正在使用。如未被使用, 则转入子程序并置该标志位, 以防其他进程重入。后来又出现功能更强的信号 (PV 操作) 指令。有的计算机还设置“执行”指令。“执行”指令执行由地址域所确定的存储单元中的指令, 其目的是避免用程序直接修改程序中的指令, 这对程序的检查和流水线等技术的应用均有好处。有的计算机采用堆栈实现程序的调用指令和返回指令, 调用时将返回地址和各种状态、参数压入堆栈顶部, 这样就能较好地实现子程序的嵌套和递归调用, 并可使子程序具有可重入性。另外, 一些计算机使不少复杂的操作固定化, 形成诸如多项式求值、队列插项、队列撤项和各种翻译、编辑等指令。

按功能划分:

1) 数据处理指令: 包括算术运算指令、逻辑运算指令、移位指令、比较指令等。

2) 数据传送指令: 包括寄存器之间、寄存器与主存储器之间的传送指令等。

3) 程序控制指令: 包括条件转移指令、无条件转移指令、转子程序指令等。

4) 输入/输出指令: 包括各种外围设备的读、写指令等。有的计算机将输入/输出指令包含在数据传送指令类中。

5) 状态管理指令: 包括诸如实现设置存储保护、中断处理等功能的管理指令。

其他划分:

1) 向量指令和标量指令: 有些大型机和巨型机设置功能齐全的向量运算指令系统。向



量指令的基本操作对象是向量，即有序排列的一组数。若指令为向量操作，则由指令确定向量操作数的地址（主存储器起始地址或向量寄存器号），并直接或隐含地指定如增量、向量长度等其他向量参数。向量指令规定处理器按同一操作处理向量中的所有分量，可有效地提高计算机的运算速度。不具备向量处理功能，只对单个量（即标量）进行操作的指令称为标量指令。

2) 特权指令和用户指令：在多用户环境中，某些指令的不恰当使用会引起机器的系统性混乱。如设置存储保护、中断处理、输入输出等指令，均称为特权指令，不允许用户直接使用。为此，处理器一般设置特权和用户两种状态，或称管（理）态和目（的）态。在特权状态下，程序可使用包括特权指令在内的全部指令。在用户状态下，只允许使用非特权指令，或称用户指令。用户若使用特权指令则会发生违章中断。如用户需要申请操作系统进行某些服务，如输入、输出等，可使用“广义指令”“访管”等指令。

## 9.2 指令解码

本节引入一种编解码描述语言（Specification Language for Encoding and Decoding, SLED），分析了其对 Pentium 指令的描述工作，针对 x86\_64 指令集新特性给出了详细的解决办法，最后设计了程序的解码算法。

### 9.2.1 SLED 通用编解码语言

NJMCT（New Jersey Machine Code Toolkit）提出了 SLED，用户可以通过使用 SLED 描述机器的指令集来编写处理器二进制码程序。SLED 提供了一种更简洁、更不容易出错的方式来描述指令的二进制表示。NJMCT 已经完成对 MIPS R3000、SPARC、Alpha 和 Intel Pentium 指令集的描述。NJMCT 分为 Icon 版本和 ML 版本，ML 版本可用来自动生成解码器，同时比 Icon 版本提供更多的优化，该版本称为 MLTK。以下内容将对 SLED 展开深入分析。

SLED 规范定义了机器指令的抽象表示、二进制码和汇编语言之间的对应关系。它使用下面 4 个元素来描述二进制机器指令：

- 1) token：表示指令的二进制表示中一串连续的位的名称。
- 2) field：token 内部一块连续位域的名称。
- 3) pattern：描述指令的二进制表示。
- 4) constructor：构造抽象层与二进制表示之间的映射。

下面更详细地阐述 SLED，并使用对 Pentium 指令集的描述举例说明。

#### 1. token 和 field

机器的指令并不总是占据一个机器字，一条指令通常由不同类型的一个或多个 token 组成。如图 9-1 所示，一条 Pentium 指令可能包括几个 8 位的前缀、一个 8 位的操作码、8 位



ModR/M 寻址模式字节、8 位的 SIB 字节、不多于 4 字节的偏移量和立即操作数。通常，前缀和操作码应属于同一类型的 token，寻址模式字节和操作数的类型却不同。

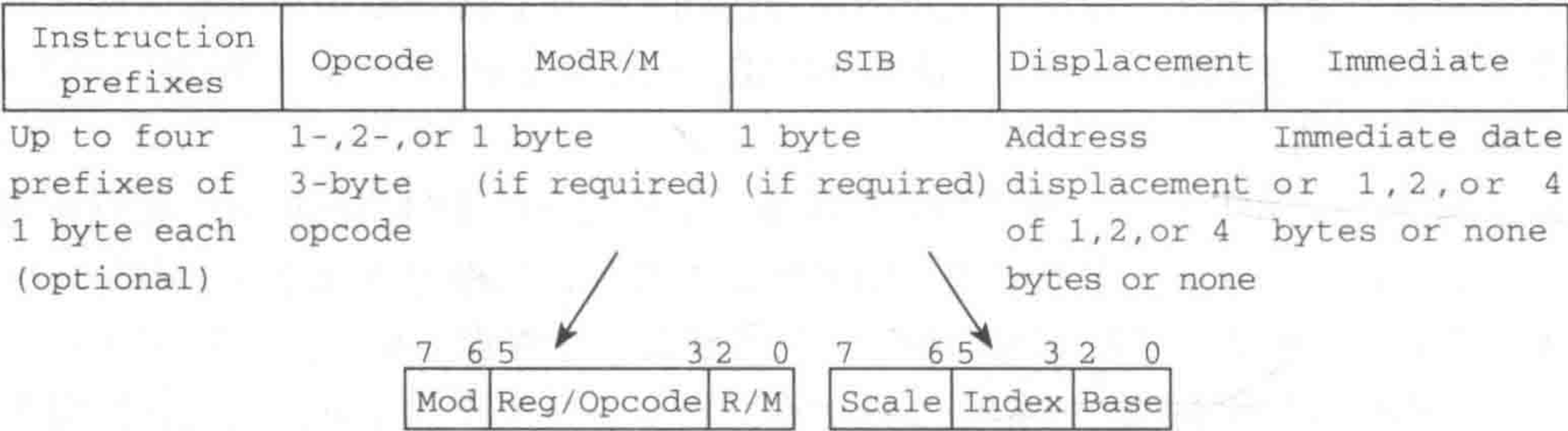


图 9-1 Pentium 指令结构

每一个 token 划分为多个 field，每个 field 是 token 内部一块连续的位域。field 通常包含了操作码、操作数、寻址模式和其他的一些信息。一个 token 可能有多种划分为 field 的方法。我们可以用关键字 fields 声明 field 的名称和其绑定的位域，并且指明它所在 token 的位数。图 9-2 是 Pentium 指令的 token 和 field 描述。

```
fields of opcode (8) row 4:7 col 0:2 page 3:3 r32 0:2 sr16 0:2 r16 0:2 r8 0:2
fields of modrm (8) mod 6:7 reg_opcode 3:5 r_m 0:2
fields of sib (8) ss 6:7 index 3:5 base 0:2
fields of I8 (8) i8 0:7
fields of I16 (16) i16 0:15
fields of I32 (32) i32 0:31
```

图 9-2 Pentium 指令结构成分描述

上列描述声明了一个名称为“opcode”的 8 位 token，这个 token 被划分为多个 field，其中“row”、“col”、“page”表示该指令操作码在 Pentium 指令手册 opcode 表中的位置；“r32”、“sr16”、“r16”、“r8”等 field 表示指令操作码字节中蕴含的寄存器寻址信息。另外还声明了名称为“modrm”、“sib”、“I8”、“I16”和“I32”的其他几类 token，分别表示 Pentium 指令中的其他组成元素。

2. pattern

pattern 描述了指令、一组指令或指令组成的一部分的二进制表示，它起着约束 field 值的作用，一条 pattern 描述可能只约束一个单独 token 中的 field，也可能同时约束多个 token。pattern 通常使用的约束形式有两种：取值范围约束和域值绑定。pattern 由与 (&)、连接 (;)、或 (|) 组合而成。一个简单的 pattern 能够用来描述操作码，而较复杂的 pattern 能够用来描述寻址模式或者一组三操作数算术指令。下列 pattern 描述了单个 MOV 指令、一组算术运算指令操作码和指令操作码中的寻址模式域的二进制表示。



```

patterns
  MOVib is row = 11 & page = 0
  arith is any of [ ADD OR
                    ADC SBB
                    AND SUB
                    XOR CMP ], which is row = {0 to 3} & page = [0 1]
  [ Eb.Gb Ev.Gv Gb.Eb Gv.Ev AL.Ib rAX.Iv ] is col = {0 to 5}

```

### 3. constructor

constructor 将指令的抽象表示、二进制码和汇编语言连接在一起。在抽象层，指令是应用到一组操作数上的一个功能函数（constructor）。使用 constructor 能产生一个给出指令二进制表示的形式，这个形式是一个典型的 token 序列。每一个 constructor 又与一个能产生指令的汇编语言表示的函数关联起来，在编写描述文件时能够使用 constructor 定义与汇编语言对应的抽象表示。应用程序编写者通过在匹配语句中使用 constructor 匹配指令和提取指令的操作数来解码指令。

SLED 在设计时为二进制表示增加了类型信息，正如每一种 token 都有自己的类型一样，也需要为每一个 constructor 定义其类型，SLED 为 constructor 提供了一个预定义的匿名类型来产生整个指令。我们也可以引进更多的 constructor 类型来表示有效地址和结构化的操作数，这样 constructor 类型就与操作数的分类相对应，并且每一个 constructor 类型对应一种访问模式。

Pentium 的有效地址通常以一个单字节的类型为 ModR/M 的 token 开头，ModR/M 中包含了一个寻址模式域和一个寄存器域。在变址模式下，ModR/M 字节后通常紧接着一个单字节的类型为 SIB 的 token，SIB 包含了变址、基址寄存器和一个索引因子“ss”。有效地址中用到的 token 和 field 在图 9-2 中已经预先定义。Pentium 的大部分指令对所有的寻址模式都支持，但某些指令只能访问操作数在内存中的有效地址，而不支持寄存器立即寻址。所以，我们在定义有效地址的 constructor 时引入类型 Mem，与操作数在内存中的寻址相对应，而引入类型 Eaddr 包含所有的寻址类型。这种区别就需要我们定义一个 constructor E 将 Mem 类型的有效地址映射到 Eaddr 类型上，如图 9-3 所示。

上述 constructor 中冒号的左边表示的是寻址模式的名称和有效地址的各组成部分，描述中用到的方括号和星号是汇编语言语法中建议使用的符号；冒号右边紧接着的“Eaddr”和“Mem”是我们为每个寻址模式定义的类型信息；大括号里的内容是对某些 field 取值范围的限定；“is”后面的描述是该 constructor 对应的 pattern。

下面给出了一些指令的 constructor：

```

constructors
MOVib r8, i8!          is MOVib & r8; i8
MOV^"mrb" Eaddr, reg  is MOV & Eb.Gb; Eaddr & reg_opcode = reg ...
arith^"iAL" i8!       is arith & AL.Ib ; i8

```



```

constructors
Indir      [reg] : Mem { reg != 4, reg != 5 } is mod = 0 & r_m = reg
Disp8 i8! [reg] : Mem { reg != 4 } is mod = 1 & r_m = reg; i8
Disp32 d [reg] : Mem { reg != 4 } is mod = 2 & r_m = reg; i32 = d
Abs32      [a] : Mem          is mod = 0 & r_m = 5; i32 = a
Reg        reg : Eaddr        is mod = 3 & r_m = reg
Index [base][index * ss] : Mem { index != 4, base != 5 } is
                        mod = 0 & r_m = 4; index & base & ss
Base  [base] : Mem { base != 5 } is
                        mod = 0 & r_m = 4; index = 4 & base
Index8 i8! [base][index * ss] : Mem { index != 4 } is
                        mod = 1 & r_m = 4; index & base & ss; i8
Base8 d! [base] : Mem is
                        mod = 1 & r_m = 4; index = 4 & base; i8 = d
Index32 d [base][index * ss] : Mem { index != 4 } is
                        mod = 2 & r_m = 4; index & base & ss; i32 = d
Base32 d [base] : Mem is
                        mod = 2 & r_m = 4; index = 4 & base; i32 = d
ShortIndex d [index * ss] : Mem { index != 4 } is
                        mod = 0 & r_m = 4; index & base = 5 & ss; i32 = d
IndirMem [d] : Mem is
                        mod = 0 & r_m = 4; index = 4 & base = 5; i32 = d
E Mem : Eaddr is Mem

```

图 9-3 Pentium 寻址模式的 constructor 描述

上述第一个 constructor “MOVib” 表示的指令是将立即数的值 i8 传送到寄存器 r8 中；“MOVmrb” 表示的指令是将 ModR/M 字节的 reg\_opcode 域表示的 8 位寄存器里的值传送到 Eaddr 所表示的 8 位寄存器或内存中；第三个 constructor 表示了一组算术指令，这些算术指令的操作数都是 AL 和 i8。

使用以上阐述的 SLED 的几个元素对机器指令进行完整的描述后就形成一个后缀名为 .spec 的文件，用于后期自动生成解码器的输入。

#### 4. 匹配语句

NJMCT 提供的解码程序通常使用 C 和 Modula-3 语言编写并嵌入匹配语句，匹配语句用来驱动二进制指令流解码，一个匹配语句类似于 C 语言中的 case 语句，但是它的分支却用一个模式（pattern）标记，而不是 case 语句中的值。哪个分支标记的 pattern 第一个被匹配成功，哪个分支就会被执行。匹配语句使用关键字 match 来标识，每一个匹配分支用符号 “|” 来标识，并且分支中用于匹配的 pattern 与执行代码通过符号 “=>” 分隔开。

下面我们通过某些指令的匹配语句举例说明。

图 9-4 描述了部分指令和寻址模式的匹配语句。假如我们当前要解码的指令是 “MOV m8, r8”，那么 decodeInstruction() 函数中含有 “MOVmrb (Eaddr, reg)” 匹配模式的分支将会被执行，在执行分支中调用 print\_Eaddr() 函数提取操作数的有效地址，在 print\_Eaddr()



函数中又根据寻址模式的匹配语句找到含有标识“E (mem)”的分支，进入 print\_Mem() 函数中进行真正的寻址模式分析。其中 print\_Abs32()、print\_Dis32()、print\_Index() 和 dis\_reg() 均为相应的处理函数。

为每一条指令设计匹配语句后，形成匹配文件 decoder.m。MLTK 的 translator 模块能够通过输入机器的 SLED 描述文件 \*.spec 和匹配文件 decoder.m 自动地将 decoder.m 中的匹配语句转换成相应的 C 或 Modula-3 语句，生成能从二进制指令流中某个地址识别出单个二进制指令的指令识别函数。用户可以设计自己的解码算法，驱动指令识别函数不断地识别二进制指令，完成指令解码工作。

```
print_Eaddr (unsigned pc)
{
    match pc to
    | Reg (reg) =>
        printf("%s", dis_reg(reg));
    | E (mem) =>
        print_Mem (mem);
    endmatch
}

print_Mem (unsigned pc)
{
    match pc to
    | Abs32 (a) => /* [a] */
        print_Abs32(a);
    | Disp32 (d, base) => /* m[ r[ base] + d] */
        print_Dis32(d,base);
    | Index (base, index, ss) => /* m[ r[base] + r[index] * ss] */
        print_Index(base, index, ss);
    ...
    endmatch
}

decodeInstruction(unsigned pc, unsigned uNativeAddr)
{ Unsigned nextPC; /* the address of next instruction*/
    match [nextPC] pc to
    | MOVib (r8, i8) =>
        printf("%s,%s,%d", "MOVib" , dis_reg(r8), i8);
    | MOVmrb (Eaddr, reg) =>
        printf("%s", "MOVmrb");
        print_Eaddr(Eaddr);
        printf("%s", dis_reg(reg));
    | ADDiAL (i8) =>
        printf("%s ,%d", "ADDiAL" , i8);
    ...
    endmatch
    return nextPC;
}
```

图 9-4 Pentium 指令和寻址模式匹配语句



9.2.2 x64 的 SLED 描述

与 x86 相比，x86\_64 将软件的线性地址空间扩展到 64 位，并且引入了一种新的操作模式，即 IA32e 模式。

IA32e 模式由两个子模式组成：①兼容模式，使得 64 位的操作系统能够不用修改地运行传统 32 位的软件。② 64 位模式，支持 64 位虚拟寻址的全地址空间和扩展寄存器的访问。与传统的模式相比，64 位模式拥有更多的新特性。

- 1) 64 位线性地址空间。
- 2) 8 个新增的通用寄存器 (GPR)。
- 3) 扩展 GPR 和指令指针的位数到 64 位。
- 4) 8 个新增的 SIMD 扩展寄存器。
- 5) 统一的字节寄存器寻址方式。
- 6) 快速中断优先机制。
- 7) 新的 RIP 相对数据寻址模式。

64 位模式的默认地址长度是 64 位，默认的操作数是 32 位，引入了一种新的操作码前缀 (REX) 来访问扩展的寄存器和指定 64 位的操作数。64 位模式扩展的指令格式如图 9-5 所示。

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp1, Grp2, Grp3, Grp4 (Optional)	(Optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	Address byte (if required)	Immediate data or 1, 2, or 4 bytes or none of 1, 2, or 4 bytes	

图 9-5 x86\_64 指令格式

下面详细地介绍了 x86\_64 和 x86 在指令构成和寻址模式上的主要区别。

1. x86\_64 和 x86 区别

(1) REX 前缀

表 9-1 简要给出了 REX 前缀的格式。从中可以看出 REX 前缀的某些组合是无效的，将会被处理器忽略。下面给出了 REX 前缀中各位更详细的使用方法：

- REX.W 位决定了操作数的长度，但不是决定操作数长度的唯一因素，同 66H 前缀一样，64 位操作数大小的重载不会对字节操作产生影响。
- 对于非字节操作，如果 66H 前缀和 REX 前缀同时使用 (REX.W = 1)，66H 前缀将会被忽略。
- 若 66H 前缀与 REX 前缀同时使用，并且 REX.W = 0，操作数长度将是 16 位。
- 当一个 GPR、SSE、控制或调试寄存器使用 ModR/M 字节的 reg 域编码时，REX.R



用于扩展该域。而当 ModR/M 指定其他寄存器或用来定义扩展操作码时, REX.R 将会被忽略。

- REX.X 用来扩展 SIB 字节的 index 域。
- REX.B 用于扩展 ModR/M 字节中的 r/m 域或者 SIB 字节的 base 域; 或者扩展 opcode 字节的 reg 域来访问 GPR。

表 9-1 REX 前缀格式

域名	比特位置	定义
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

### (2) 64 位立即操作数

在 64 位模式下, 立即操作数的一般长度仍然是 32 位, 当指令的操作数要求为 64 位时, 处理器在使用之前对其进行 64 位符号扩展。

x86\_64 通过扩展现有的 MOV 指令 (MOV reg, imm16/32) 支持 64 位立即操作数。这些指令通常将 16 位或者 32 位的立即数传送到 GPR, 而当使用 REX 前缀将指令的操作数修改为 64 位时, 可以实现传送一个 64 位的立即操作数到 GPR 中。例如:

```
48 B8 8877665544332211
MOV RAX,1122334455667788H
```

其中, 首字节“48”代表一个 REX 前缀, 并且 REX.W = 1, 第二字节“B8”表明该指令是一个将立即数存入 GPR 的 MOV 指令, 后面的连续 8 字节则是一个 64 位的立即操作数。

### (3) RIP 相对寻址

RIP 相对寻址是 x86\_64 在 64 位模式时实现的一种新的寻址方式。在含有 RIP 寻址的指令中, 操作数的有效地址由指令中给出的偏移量加上下一条指令的 64 位 RIP 得到。

在 IA32 体系结构和兼容模式下, 相对指令指针寻址仅在控制转移指令中是可用的。在 64 位模式下, 使用 ModR/M 寻址的指令都可能使用 RIP 相对寻址。如果没有使用 RIP 相对寻址, 那么所有 ModR/M 指令模式的寻址都是相对地址为 0 的寻址。RIP 相对寻址通过指定 ModR/M 模式使用一个相对 64 位 RIP 的有符号 32 位偏移的方式来实现。表 9-2 显示了使用相对 RIP 寻址时 ModR/M 和 SIB 字节的编码。

如表 9-2 所示, 在 64 位模式下, ModR/M Disp32 的编码不再仅是一个偏移量, 而是被重新定义为 RIP+Disp32。



表 9-2 RIP 相对寻址

ModR/M 和 SIB 中各域的编码		兼容模式操作	64 位模式操作
ModR/M 字节	mod ==0	Disp32	RIP+Disp32
	r/m ==101		
SIB 字节	base ==101	if mod = 0, Disp32	与传统模式相同
	index ==100		
	scall = 0,1,2,4		

RIP 相对寻址在 64 位模式下使用，而不是指在 64 位地址大小时。使用地址大小前缀并不能禁止 RIP 相对寻址的使用，地址大小前缀仅起着将计算出的有效地址截断或扩展到 32 位的作用。

2. x86\_64 指令 SLED 描述的构建

针对上述 x86\_64 指令集的新特性，下面用 SLED 对其进行描述，从而为指令解码奠定基础。

(1) REX 前缀的 SLED 描述

REX 前缀是包含了操作码表中从 40H 到 4FH 共 16 个操作码的集合，整整占据了操作码表的一行。在 IA32 和兼容模式下，这些操作码都表示一些其他的合法指令（INC 和 DEC）。而在 64 位模式下，同样的操作码表示为指令的 REX 前缀，而不再表示单个指令。因此，首先用 patterns 描述 REX 前缀的二进制表示：

```
patterns
  REXPrefix is row = 4
```

我们可以得出 REX 前缀和操作码都属于名称为 opcode 的一类 token。从表 9-1 中可以看出，REX 前缀中不同的位域具有不同的功能，所以针对 REX 中每一个有用的位，我们给出其 field 在整个 token 中的位域描述：

```
fields of opcode (8) row 4:7 col 0:2 page 3:3
  r64 0:2 r32 0:2 sr16 0:2 r16 0:2 r8 0:2
  Rex 0:3 Rex.B 0:0 Rex.X 1:1 Rex.R 2:2 Rex.W 3:3
```

“Rex”域与 REX 前缀的 0 ~ 3 位绑定，包含了 REX 前缀中的所有有用信息。“Rex.B”“Rex.X”“Rex.R”“Rex.W”域分别与 REX 前缀的 0、1、2、3 位绑定，以用于实现各自的功能。

然后，我们针对每条指令给出其对应的 constructor。通常 REX 前缀中的某些位域是指令操作数的一部分，比如使用 REX.R 域与 ModR/M 字节的 reg 域组合来访问扩展寄存器。在 NJMCT 所提供的对指令集的描述中，通常所有的操作数都在一个连续的位域。而在 x86\_64 指令集中，如果一条指令包含 REX 前缀字节，并且使用 REX 字节的某一位作为寄存器的扩展位，那么扩展后的寄存器表示将由两部分构成，一部分是原来的 reg 位域，另一部分则是相应的 REX 中的某一位，NJMCT 并没有对这种不连续的组合位域提供直接的解



决办法。

因此，在遇到使用 REX 前缀访问扩展寄存器的情况时，我们做一个延迟处理，把 Rex 域作为一个 constructor 额外的操作数传递给匹配语句中指令的匹配分支。例如，使用 REX 前缀的 MOV 指令的 SLED 描述如下：

```
constructors
    MOV^"Rex"^^"mrb" Rex, Eaddr, reg is
                                RexPrefix&Rex; MOV & Eb.Gb;
                                Eaddr & reg_opcode = reg ...
```

最后，我们在匹配分支后的执行语句中通过判断 Rex 的值修改 reg 的值：

```
| MOVRexmrb(Rex, Eaddr, reg) =>
    if(Rex&0x0004)
        reg = reg + 8;
    printf("%s", "MOVmrb");
    print_Eaddr(Eaddr, Rex);
    printf("%s", dis_reg(reg));
```

函数调用 print\_Eaddr (Eaddr, Rex) 的目标函数 print\_Eaddr (unsigned pc, unsigned Rex) 是对图 9-4 中 print\_Eaddr (unsigned pc) 函数的重载。因为 Rex 的值会对寻址模式中 ModR/M 和 SIB 字节的某些域进行扩展，所以需要将 Rex 的值传递给该函数，再在函数内部根据 Rex 的值在匹配分支中对相应的寄存器编号进行修正。

通过以上 SLED 描述的添加和匹配语句的修正，解决了由 REX 前缀带来的访问扩展寄存器和 64 位操作数等问题。

### (2) 立即操作数的 SLED 描述

在 64 位模式下，有时立即操作数的大小是 64 位的，而 NJMCT 提供的能处理的最长二进制数据是 32 位。所以，我们在描述指令时，将 64 位的立即数拆分为高 32 位和低 32 位，因此我们定义了相应的 token：

```
fields of  IH32 (32)  ih32  0:31
fields of  IL32 (32)  il32  0:31
```

通过 ih32 域和 il32 域的组合，可以表示一个 64 位长度的立即数。

它们在 constructors 中的使用方法用指令 MOV r64, imm64 举例说明如下：

```
constructors
    MOVRexiqid Rex, r64, il32, ih32 is
                                RexPrefix&Rex&Rex.W = 1; MOViv & r64; il32; ih32
```

最后，在匹配文件的匹配分支里将参数 il32 和 ih32 组合成一个 64 位的立即数。

### (3) RIP 相对寻址情况的处理

由于在使用 RIP 相对寻址的指令中，操作数的有效地址由指令中的偏移量加上下一条指令的地址构成，所以需要在相应指令的匹配分支里先计算出下一条指令的虚拟地址



nextNativePC, nextNativePC 可以使用下一条指令的主机地址 nextPC 减去当前指令的主机地址 pc, 再加上当前指令的虚拟地址 uNativeAddr 得到。

然后, 将该虚拟地址传递给有效地址提取函数 print\_Eaddr(), print\_Eaddr() 中调用函数 print\_Mem() 进行具体内存寻址模式匹配。从之前的分析可以得出, RIP 相对寻址采用了传统内存寻址模式中的 Abs32 模式, 所以在 print\_Mem 中的 Abs32 匹配分支中, 将传入的下一条指令的参数 nextNativePC 加上原来的偏移量 a 得到真正的操作数内存地址。如下所示:

```
print_Mem (unsigned pc, unsigned nextNativePC)
{
    match pc to
    | Abs32 (a) => /* RIP addressing*/
        print_Abs32(a+ nextNativePC);
    .....
    endmatch
}
```

### 3. 指令解码器的实现

NJMCT 只能为 x86\_64 生成一个从某二进制指令流中识别出一条汇编指令的指令识别函数 decodeInstruction()。我们仍需要设计一个指令解码算法来驱动该函数从一个二进制文件的程序入口点开始解码, 并不断获取下一条要解码的指令地址, 直至整个二进制文件的可执行代码全部被解码。

本书设计的解码程序基于改进的递归遍历算法。如图 9-6 所示, 算法的流程如下:

1) 对程序 (Prog) 进行预处理, 创建一个过程 Proc 队列, 该队列初始情况下只含有一个以程序的入口点为首地址的 proc。

2) 判断 Proc 队列是否为空, 若为空, 则转入步骤 6, 若不为空, 则从 Proc 队列中取出队首 Proc, 并对该 Proc 进行相应的预处理, 如建立一个空的控制流图 CFG、创建一个初始情况下只包含一个地址的待解码目标地址队列 targetQueue。

3) 判断 targetQueue 队列是否为空, 若为空, 则该 proc 已解码完毕, 返回步骤 2 寻找下一个待解码的 Proc; 若不为空, 则取出队首地址从该地址进行解码。

4) 调用指令识别函数, 从给出的地址识别出指令, 并映射到中间表示 RTL。

5) 判断识别出的指令类型, 并进入相应的分支处理。如果是普通指令, 则直接计算下一条指令地址, 转入步骤 4; 如果是转移指令, 则生成新的基本块, 更新 Proc 的 CFG, 然后根据转移指令类别区别处理:

- 如果是条件跳转指令, 则判断跳转目标地址是否已解码, 若未解码, 需要将目标地址加入 targetQueue 队列, 若已解码则直接计算下一条指令地址, 转入步骤 4。
- 如果是过程调用指令, 则分析目标 Proc 是否已经解码, 若未解码, 则需将目标 Proc 加入 Proc 队列, 若已解码, 则直接计算下一条指令地址, 转入步骤 4。
- 如果是无条件跳转指令和间接跳转指令, 则判断目标地址是否已解码, 如果未解码,



则需要把目标地址加入 targetQueue 队列，然后转入步骤 3。

- 如果是过程返回指令，则进行单个 Proc 解码结束的相关处理，然后转入步骤 2。

6) 程序解码结束。

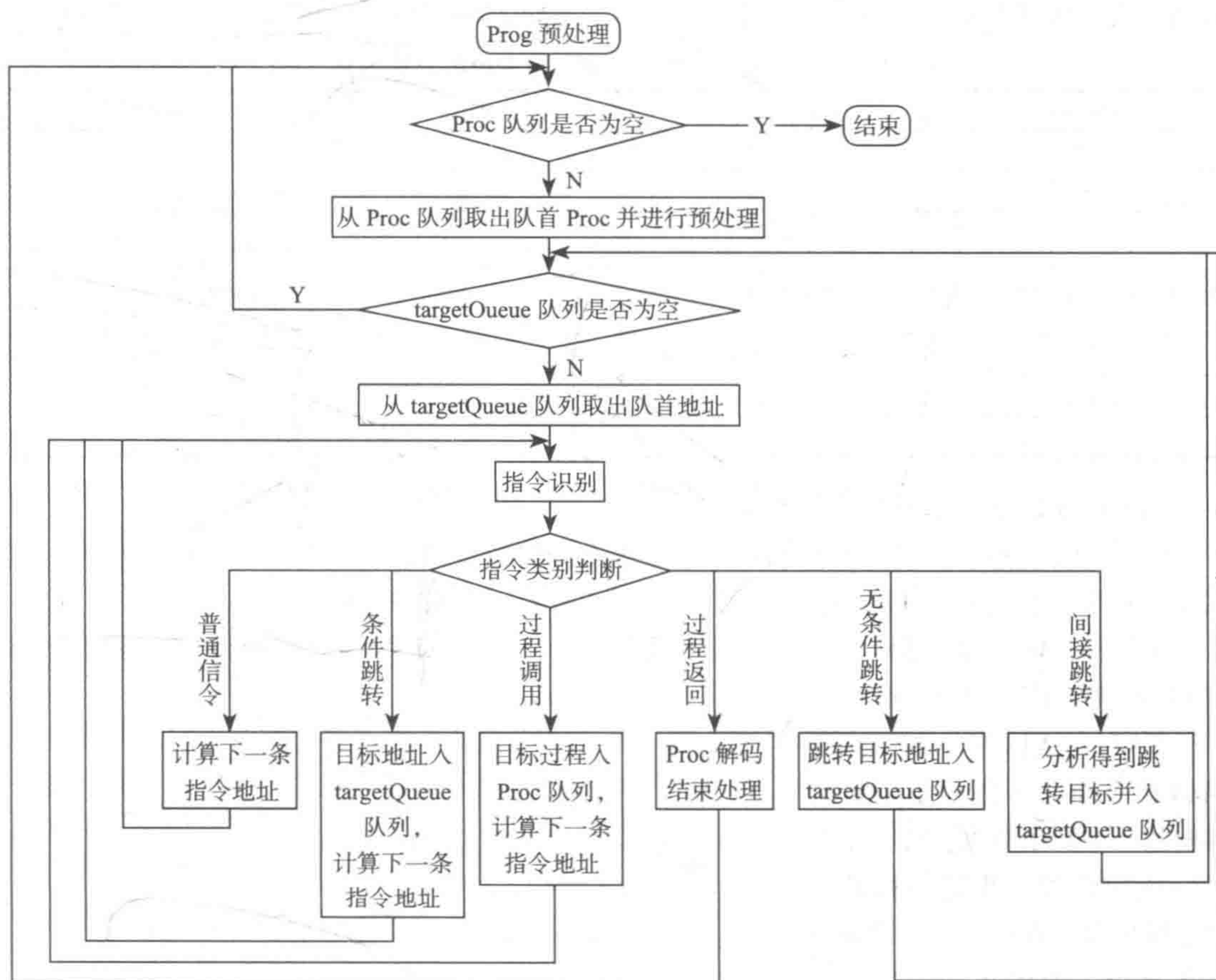


图 9-6 x86\_64 指令解码算法流程图

### 9.2.3 IA64 的 SLED 描述

#### (1) IA64 指令特点

Intel 的 IA64 指令系统具有 EPIC (显式并行指令计算) 特性，兼容了 RISC 和超长指令字各自的优点。其中指令 bundle (如图 9-7 所示) 128 位，包含 3 个 41 位的指令槽 (slot) 和一个 5 位的 template，所有的 128 位信息被处理器一次装载并解码，依靠指令的 template 信息，3 条无冲突的指令可以同时不同执行单元处理。

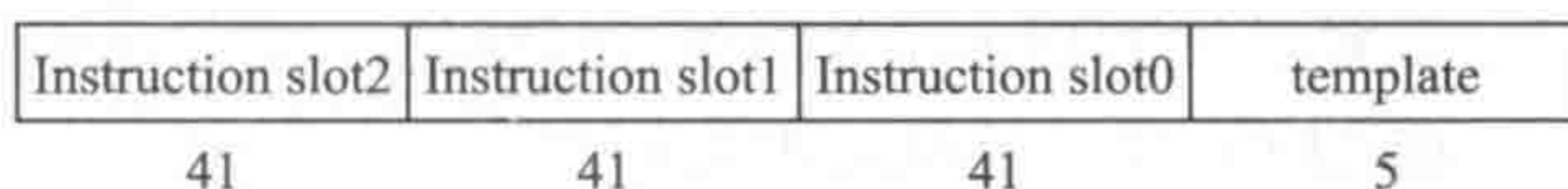


图 9-7 bundle 的格式



IA64 指令按照指令类型分为 6 类：A (Integer Alu)、I (NonAlu Integer)、M (Memory)、B (Branch)、F (Float Point)、X (Extended)，分别对应 I/M、I、M、F、B、I/B 执行单元。

执行机制按每个 bundle 的起始 5 位所规定的 template 把 bundle 中的三条指令有效地加载到正确的执行单元。IA64 定义了 24 个不同的 template (其他 8 个保留)，如表 9-3 所示。

表 9-3 template 域的编码与指令槽的对应关系

template	slot0	slot1	slot2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
...			
IC	M-unit	I-unit	I-unit
ID	M-unit	I-unit	I-unit
...			

(2) 描述思想

使用 SLED 对 IA64 指令进行描述，通过使用 MLTK 工具生成相当于 Decoder 的 .m.d 文件，并希望生成的文件能够覆盖全部指令且能够唯一识别指令。

采用分类树结构表示对 IA64 所有指令集的划分，如图 9-8 所示，并通过以下步骤唯一识别 IA64 每条指令的反汇编器。

步骤一：建立树。根节点是 IA64 指令全集，每条指令是这棵树的一个叶子节点，按一定的指令规则将指令集沿着树的深度逐级划分，直至能唯一识别每一条指令，即到达相应的叶子节点。

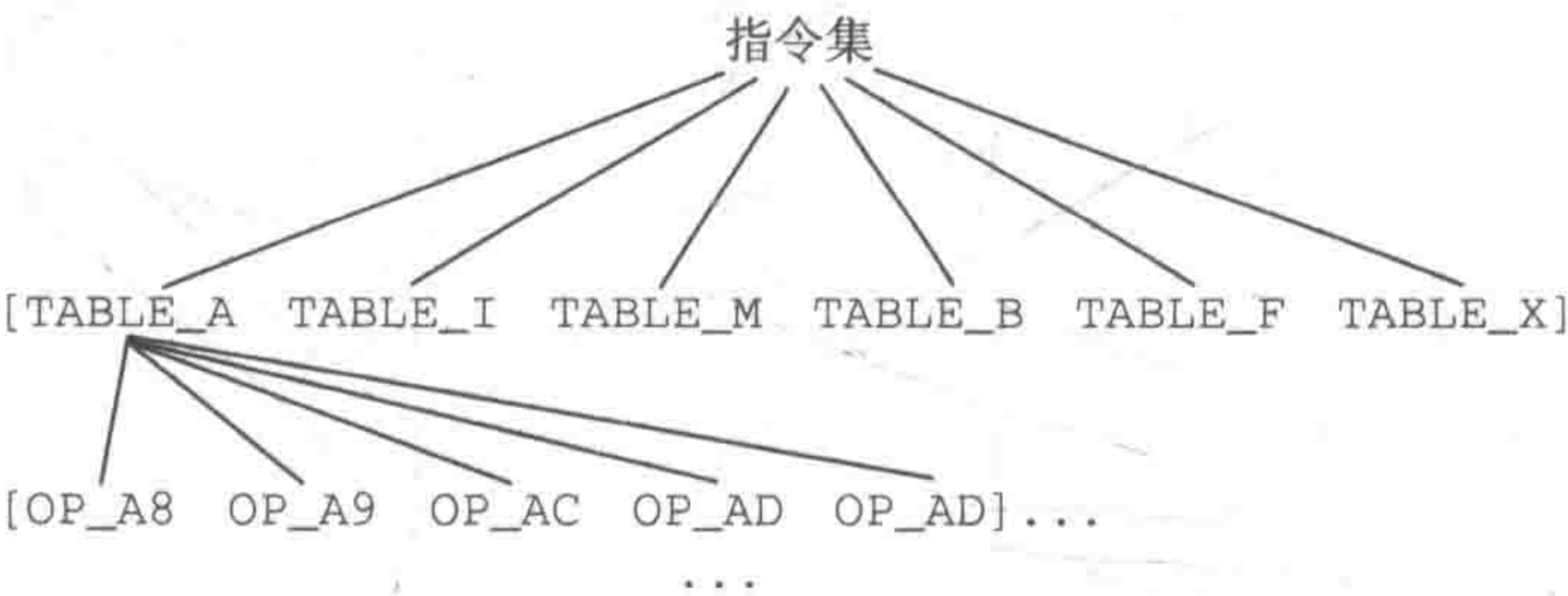


图 9-8 指令集的划分树

步骤二：扩展指令位。将指令在原有 41 位的基础上增加 3 位，通过增加的 3 位确定指令的执行单元扩展类型。

首先，根据指令的执行单元类型把指令集划分为 6 个子集 A、I、M、B、F、X。由于 IA64 指令的 41 位结构中没有与执行单元类型相关的位信息，因此在实现过程中必须在指令的 41 位基础上再增加 3 位表示执行单元扩展类型。这样分类不仅符合 IA64 指令结构中指令的设计思想，而且是最短的指令分类树。

步骤三：用 SLED 对指令进行描述，编写 .spec 文件。

用 SLED 描述这棵树即为：

patterns # 以表的形式描述部分第一层结点



```
[ TABLE_A TABLE_I TABLE_M TABLE_F TABLE_B TABLE_X _ _ ] is ty = { 0 to 7 }
patterns # 部分第二层结点
[ OP_A8 OP_A9 _ _ OP_AC OP_AD OP_AE ] is op = { 8 to 14 }
patterns # 部分第三层结点, 其中 adds_imm14, _, app4_imm14 为叶子, 即 IA64 指令
[ ALUI adds_imm14 _ app4_imm14 ] is OP_A8 & x2a = { 0 to 3 } & ve = 0
ALUM is OP_A8 & x2a = 2
patterns
[ add add_ _ _
  sub_ sub _ _
  addp4 _ _ _
  add addcm or xor ] is ALUI & x4 = { 0 to 3 } & x2b = { 0 to 3 }
```

步骤四：编写 .m 文件。.m 文件是解码器的一部分，它是含有一个或多个匹配状态的文件，该文件主要描述通过匹配地址，对指令集进行处理。编写 .m 的重点是按照 .spec 文件的指令结构，书写各条指令对应的匹配语句，提交上层所需的指令参数。

步骤五：通过 MLTK 工具生成 IA64 的 .m.d 文件。.m.d 文件是对 .spec 文件中定义的指令通过 .m 文件生成的指令解码程序。它提供了二进制文件解码器，从输入的二进制文件中读入指令流，然后使用机器指令的语法描述将该指令流解析成源机器指令序列。

步骤六：测试生成的 .m.d 文件的正确性。

## 9.3 反汇编过程

对可执行文件进行分析最常见的方法是反汇编，使用反汇编器进行静态分析获得程序信息。其中，静态反汇编是指在反汇编的过程中不执行代码，通过对可执行文件的静态分析得到汇编代码，从而获得程序中机器代码的方法。简单地说，就是将用于执行的二进制序列转化为人们容易理解的汇编指令。为了实现这项功能，目前有两种主要的反汇编算法，分别是线性扫描算法和行进递归算法。

### 9.3.1 线性扫描反汇编

#### 线性扫描 (Linear Sweep) 算法

线性扫描算法从可执行程序入口点 (entry point) 开始反汇编，对整个代码段进行扫描，按照地址顺序将二进制序列根据具体指令集映射为相应的机器指令。遇到非法指令时，算法终止或者从下一字节处继续反汇编。算法描述如图 9-9 所示。

线性扫描算法流程简单，根据反汇编得到当前指令长度 (length)，同时确定下一条指令的开始地址 (addr)，不对指令的类型进行分析。其优点是反汇编覆盖率高，能够覆盖整个代码段；缺点是无法区分数据与代码。由于在冯·诺依曼体系结构下，数据和指令混合存储，并不进行区分，因此很容易将代码段中嵌入的数据识别为指令，影响反汇编的正确性。目前，常用的静态反汇编器 GNU 的 objdump 和动态分析工具 Ollydbg 都采用线性扫描算法进行反汇编。



```

global startAddr, endAddr;
proc DisasmLinear(addr)
{
    while (startAddr ≤ addr ≤ endAddr) // 保证 addr 在代码段的开始位置和结束位置之间
    {
        instr = DecodeInstr(addr); // 得到在 addr 处的机器指令
        if (instr is an invalid instruction) // 如果得到的指令错误
        {
            1) break; // 跳出循环, 反汇编过程结束
            or 2) addr += 1; continue; // 或者从下一字节处继续反汇编
        }
        addr += Length(instr); // 顺序扫描下一条指令, 直到代码段结束
    }
}

```

图 9-9 线性扫描反汇编算法

### 9.3.2 行进递归反汇编

#### 行进递归 (Recursive Traversal) 算法

行进递归算法是利用解码器得到指令类型, 同时构建程序的控制流图, 按照控制流确定下一步反汇编的位置, 对每条可能的路径都进行扫描。如果当前指令不是控制转移指令, 则按照地址顺序进行反汇编; 若遇到控制转移指令 (如跳转、调用、返回指令等), 则先利用静态分析技术确定可能跳转地址的集合, 再从转移地址处继续进行反汇编。其算法描述如图 9-10 所示。

```

global startAddr, endAddr;
addr=entrypoint; // 将程序入口点作为开始解码的初始位置
proc DisasmRec(addr)
{
    while (startAddr ≤ addr ≤ endAddr)
    {
        if (addr has been visited already) // 如果地址 addr 已经被访问过, 则跳过该地址
            return;
        instr = DecodeInstr(addr); // 得到在 addr 处的机器指令
        addr.visited = true; // 标记 addr 已访问
        if (instr is a control transfer instruction) // 如果是控制转移指令
        {
            for (each target ∈ T) // 对控制流转移指令 instr 的每个目标地址 target 进行处理, 其中 T 是指令 instr 所有可能控制流后继的集合
                DisasmRec(target); // 递归调用, 对跳转目标地址 target 反汇编
            return;
        }
        else
            addr += Length(instr); // 如果不是控制转移指令, 则顺序遍历下一条指令
    }
}

```

图 9-10 行进递归反汇编算法



行进递归算法的缺点在于准确定位间接转移目标地址的难度较大, 容易造成代码分析的空隙, 从而降低了程序分析的覆盖率。尤其是恶意代码中经常使用混淆手段, 阻止反汇编器识别指令控制转移的目标地址。目前, 采用行进递归算法的反汇编工具主要有 W32dsm、IDA 等。

## 9.4 反汇编工具 IDA 与 OllyICE 实践

本节通过一个简单的例子来介绍反汇编工具的简单使用方法, 例子 test.cpp 的功能是用一个循环结构求 1 ~ 100 的和, 再打印到屏幕。代码如下:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    int i,a=0;

    for(i=1;i<=100;i++) {
        a=a+i;
    }

    printf("a=%d\n",a);
    return 0;
}
```

首先用 Visual Studio 2010 编译该例子, 生成 test.exe 文件, 之后用 IDA 和 OllyICE (由于 Ollydbg1.1 官方不再更新, 所以采用一些爱好者维护的基于 Ollydbg 的版本 OllyICE) 反汇编该文件。

### 9.4.1 IDA 实践

启动 IDA 会出现一个基本界面, 提供 3 种选择: 打开新的待反汇编文件、一个空白的界面或继续之前的工作。

选择新建后将会打开文件选择框, 选定分析的文件后 IDA 将启动加载器进行加载。加载 test.exe 后的软件界面如图 9-11 所示。

在空白区域右击, 选择 “text view”, 软件界面如图 9-12 所示。

在软件最左边的 “Functions window” 窗口中找到 \_wmain 函数, 该函数就是该程序的 main 函数, 双击后, 右边的 IDA viewA 视图显示了 main 函数包含的汇编指令, 如图 9-13 所示。

定位到 “text:00411A1E” 的位置, 经过分析不难发现, [ebp+i] 表示源代码中的变量 i, [ebp+a] 表示源代码中的变量 a。



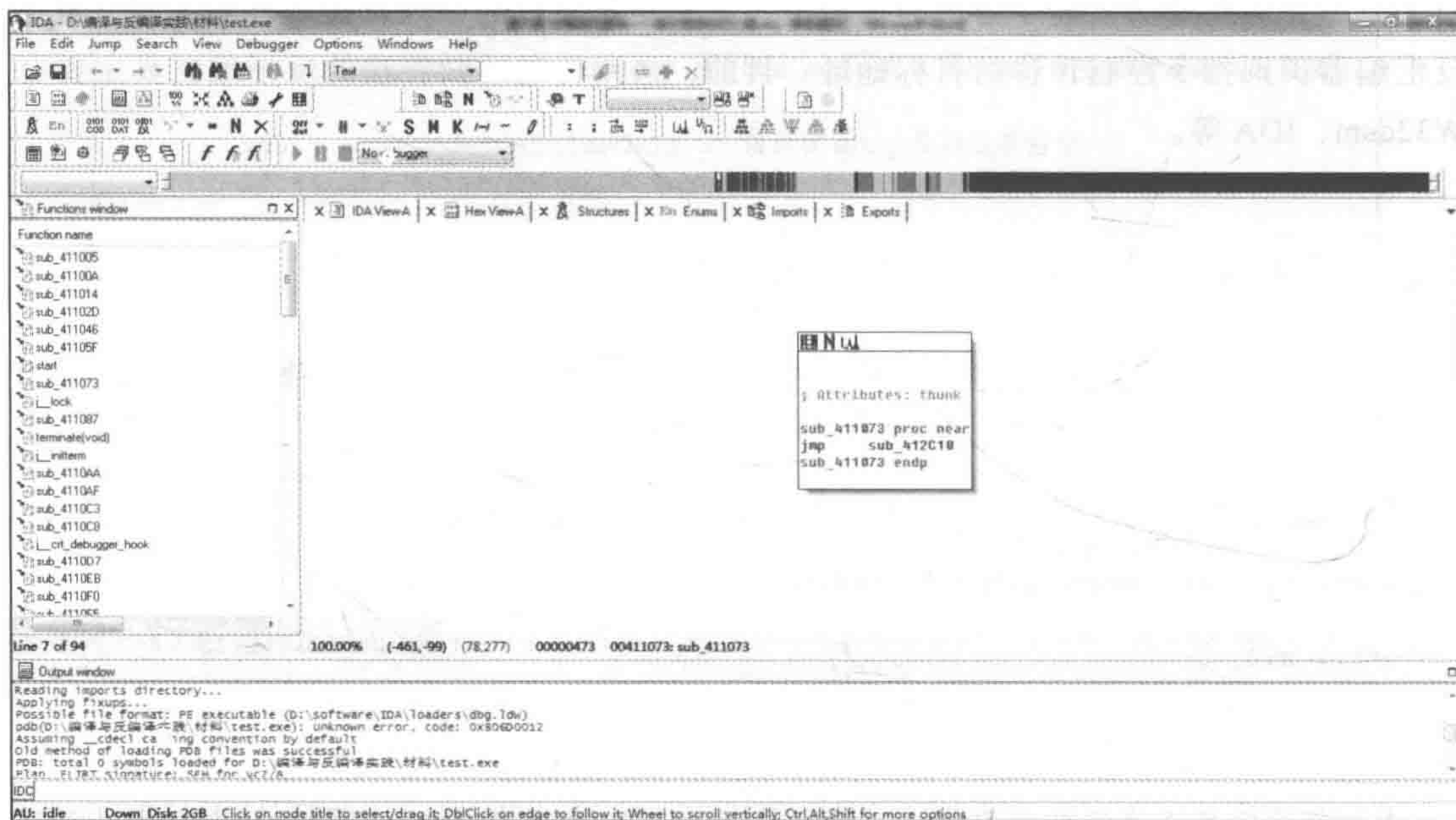


图 9-11 软件加载后的界面

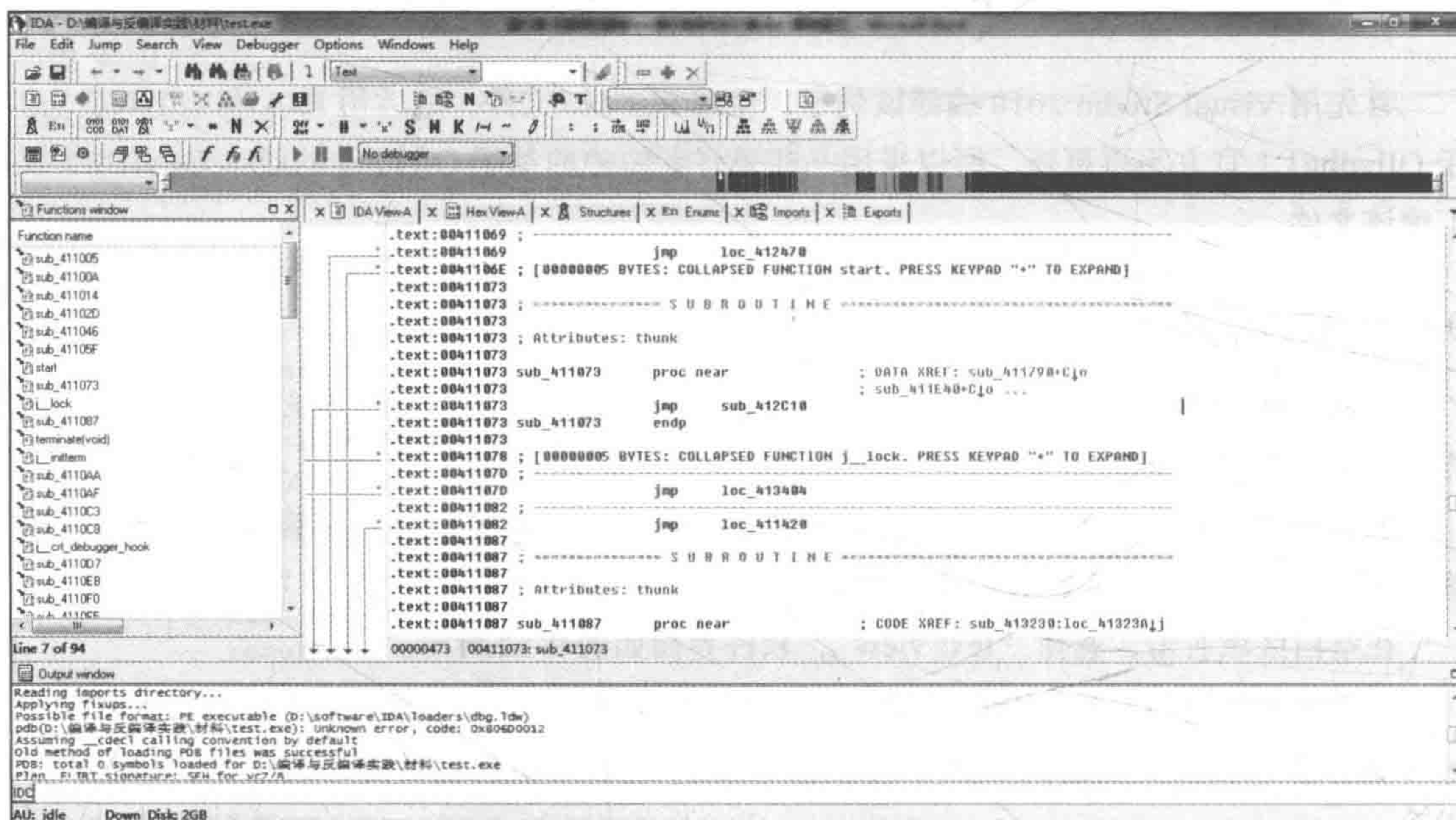


图 9-12 “text view” 视图



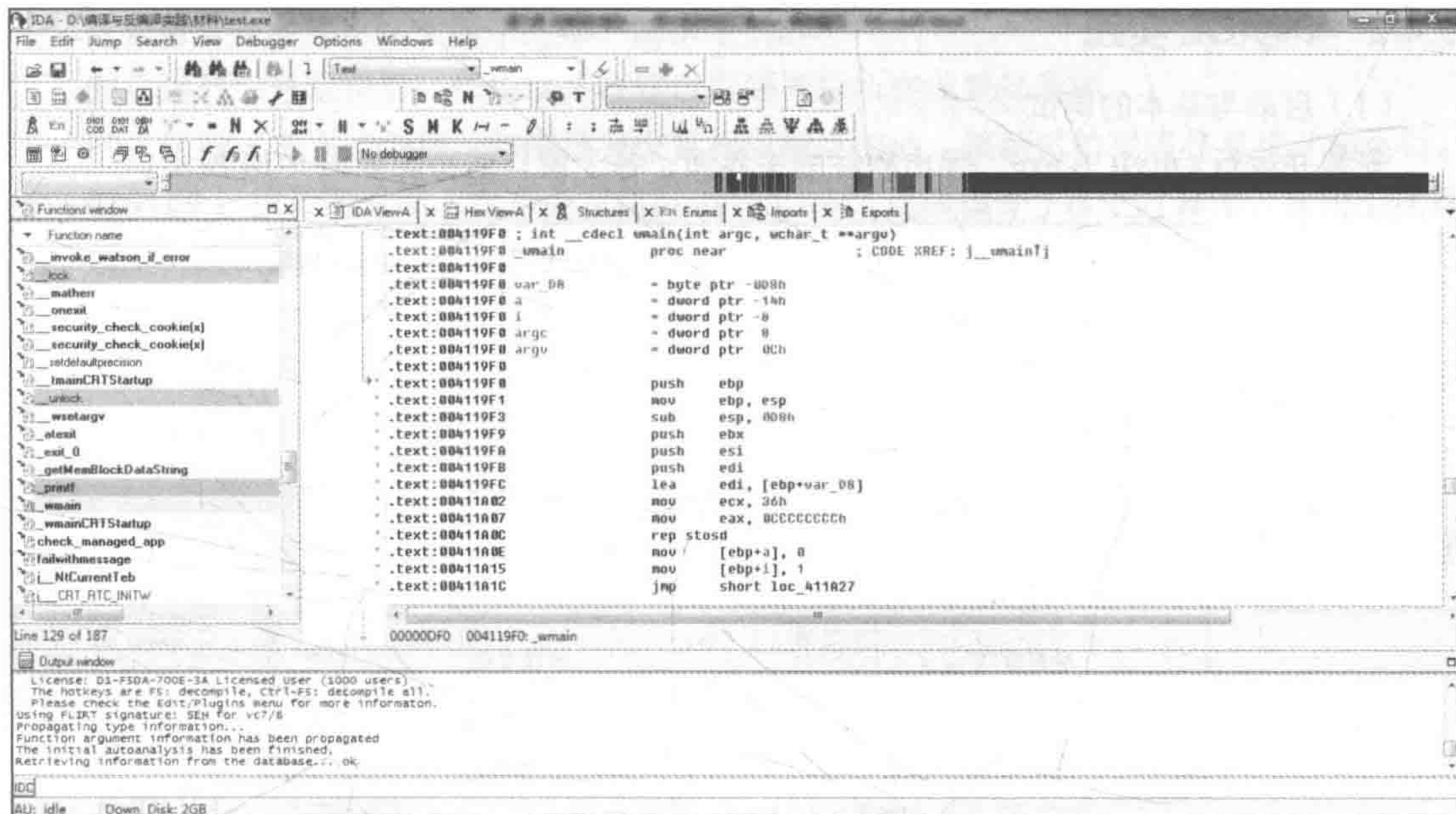


图 9-13 main 函数位置

```
.text:00411A1E loc_411A1E:                                ; CODE XREF: _wmain+46j
.text:00411A1E      mov     eax, [ebp+i]
.text:00411A21      add     eax, 1
.text:00411A24      mov     [ebp+i], eax
.text:00411A27
.text:00411A27 loc_411A27:                                ; CODE XREF: _wmain+2C165j
.text:00411A27      cmp     [ebp+i], 64h
.text:00411A2B      jg      short loc_411A38
.text:00411A2D      mov     eax, [ebp+a]
.text:00411A30      add     eax, [ebp+i]
.text:00411A33      mov     [ebp+a], eax
.text:00411A36      jmp     short loc_411A1E
.text:00411A38 ; -----
.text:00411A38
.text:00411A38 loc_411A38:                                ; CODE XREF: _wmain+3B165j
```

后面的地址“.text:00411A43”是调用 printf 函数的位置, 参数是由

```
.text:00411A3D      push    eax
.text:00411A3E      push    offset Format      ; "a=%d\n"
```

进行传递的。

上面的步骤简单地将可执行文件进行了加载以及分析, IDA 作为一款强大的静态反汇编工具, 还有其他很多高级的功能来辅助分析。



## 9.4.2 OllyICE 实践

### (1) 启动与基本的调试

安装并运行 OllyICE.exe，弹出软件的主界面，各个窗口的功能如图 9-14 所示。

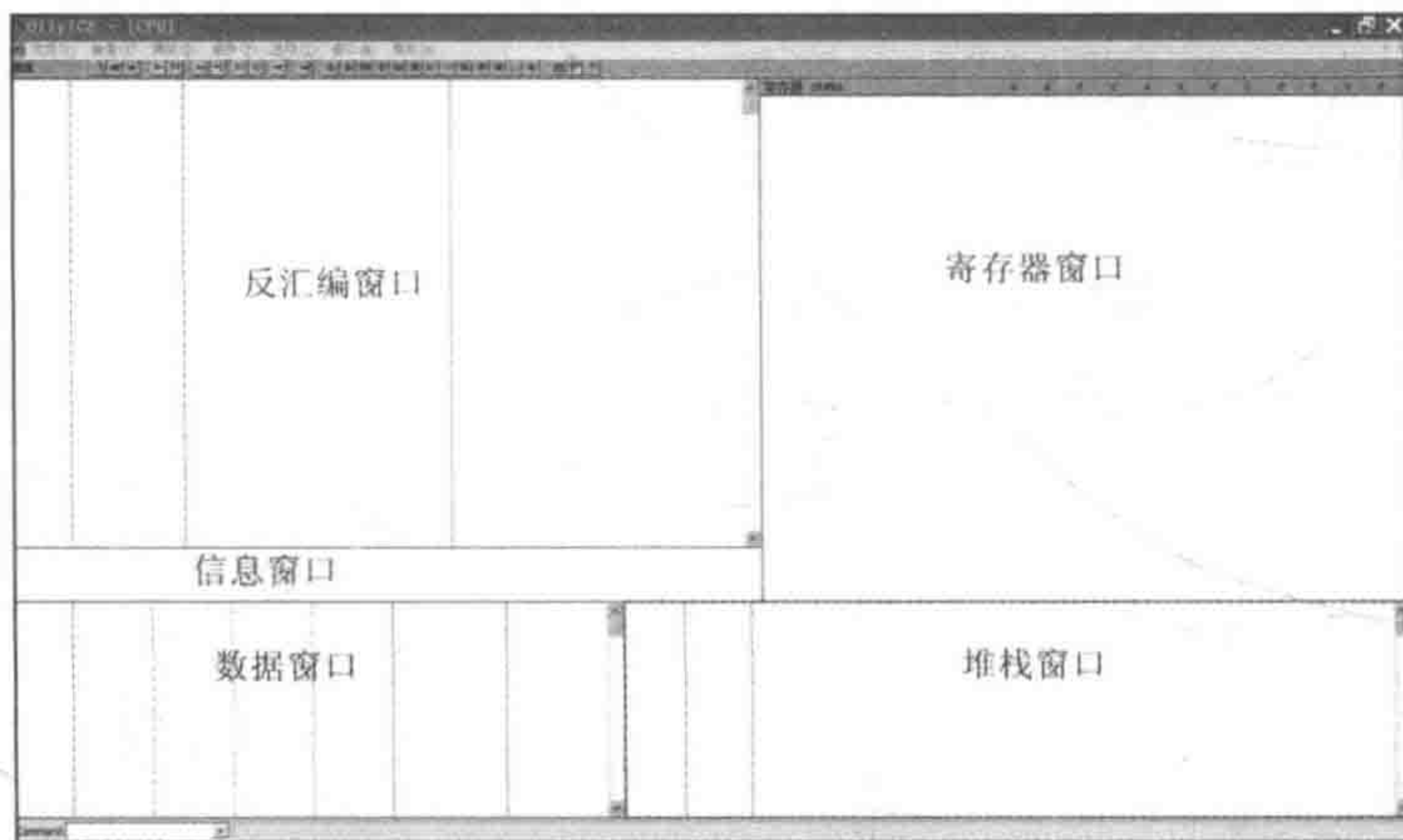


图 9-14 OllyICE 主界面

**反汇编窗口：**显示被调试程序的反汇编代码，标题栏上的“地址”“HEX 数据”“反汇编”“注释”选项可以通过在窗口中右击所出现的菜单界面选项里的“隐藏标题”或“显示标题”来切换是否显示。用鼠标左键单击注释标签可以切换注释显示的方式。

**寄存器窗口：**显示当前所选线程的 CPU 寄存器内容。同样单击标签“寄存器 (FPU)”可以切换显示寄存器的方式。

**信息窗口：**显示反汇编窗口中选中的第一个命令的参数及一些跳转目标地址、字符串等。

**数据窗口：**显示内存或文件的内容。右键菜单可用于切换显示方式。

**堆栈窗口：**显示当前线程的堆栈。

OllyICE 支持插件功能，插件的安装也很简单，只要把下载的插件（一般是 DLL 文件）复制到 OllyICE 安装目录下的 PLUGIN 目录中就可以了，OllyICE 启动时会自动识别。

OllyICE 有两种方式来载入程序进行调试，一种是单击菜单“文件”→“打开”（快捷键是 F3）来打开一个可执行文件进行调试，另一种是单击菜单“文件”→“附加”来附加到一个已运行的进程上进行调试。注意这里要附加的程序必须已运行。

调试中我们经常要用到的快捷键如下所示：

**F2：**设置断点，只要在光标定位的位置（如图 9-15 中灰色条）按 F2 键即可，再按一次 F2 键则会删除断点。

**F8：**单步步过。每按一次这个键，执行反汇编窗口中的一条指令，遇到 CALL 等子程序不进入其代码。

**F7：**单步步入。功能与单步步过（F8）类似，区别是遇到 CALL 等子程序时会进入其



中, 进入后首先会停留在子程序的第一条指令上。

F4: 运行到选定位置。作用就是直接运行到光标所在位置处暂停。

F9: 运行。按下这个键, 如果没有设置相应断点的话, 被调试的程序将直接开始运行。

Ctrl+F9: 执行到返回。此命令在执行到一个 ret (返回指令) 指令时暂停, 常用于从系统“领空”返回到被调试的程序“领空”。

Alt+F9: 执行到用户代码。可用于从系统“领空”快速返回到被调试的程序“领空”。

上面提到的几个快捷键对于一般的调试基本上已够用了。要开始调试只需设置好断点, 找到感兴趣的代码段再按 F8 或 F7 键来一条条分析指令功能就可以了。

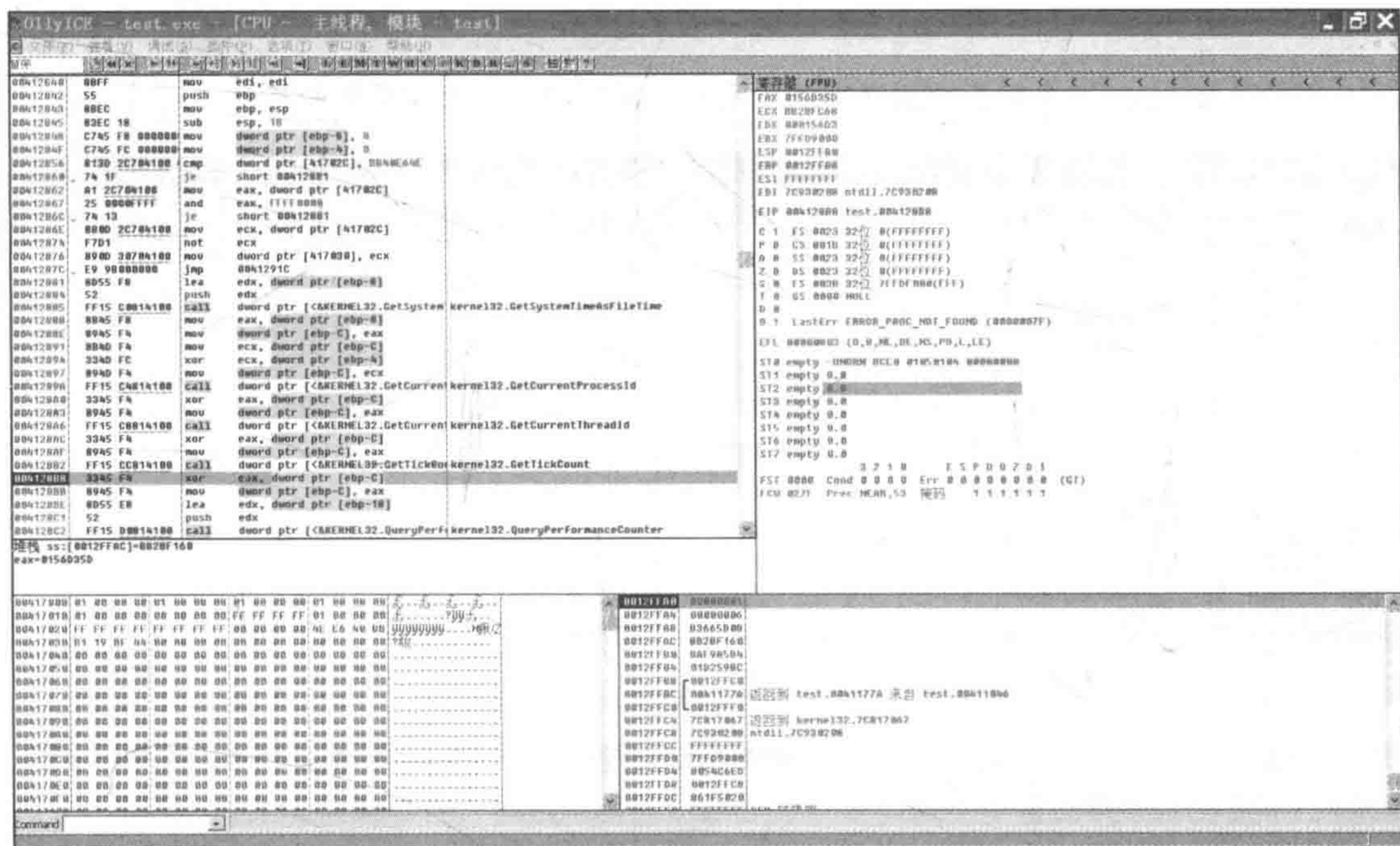


图 9-15 OllyICE 加载程序后界面

## (2) 字符串查找

在反汇编窗口中右击, 会弹出一个菜单, 在“查找”→“所有参考文本字符串”上单击左键, 如图 9-16 所示。

当然如果使用“超级字符串参考+”插件会更方便, 但我们的目标是熟悉 OllyICE 的一些操作, 就尽量使用 OllyICE 自带的功能, 少用插件。现在弹出来另一个对话框, 在这个对话框里右击, 选择“查找文本”菜单项, 输入查找的字符串, 找到一处。在已经找到的字符串上右击, 再在弹出来的菜单上单击“反汇编窗口中跟随”, 来到反汇编窗口中引用该字符串的指令位置。

除字符串参考外, 还有函数参考、内存断点、消息断点、汇编等常用的高级功能, 本节



不再详细介绍。

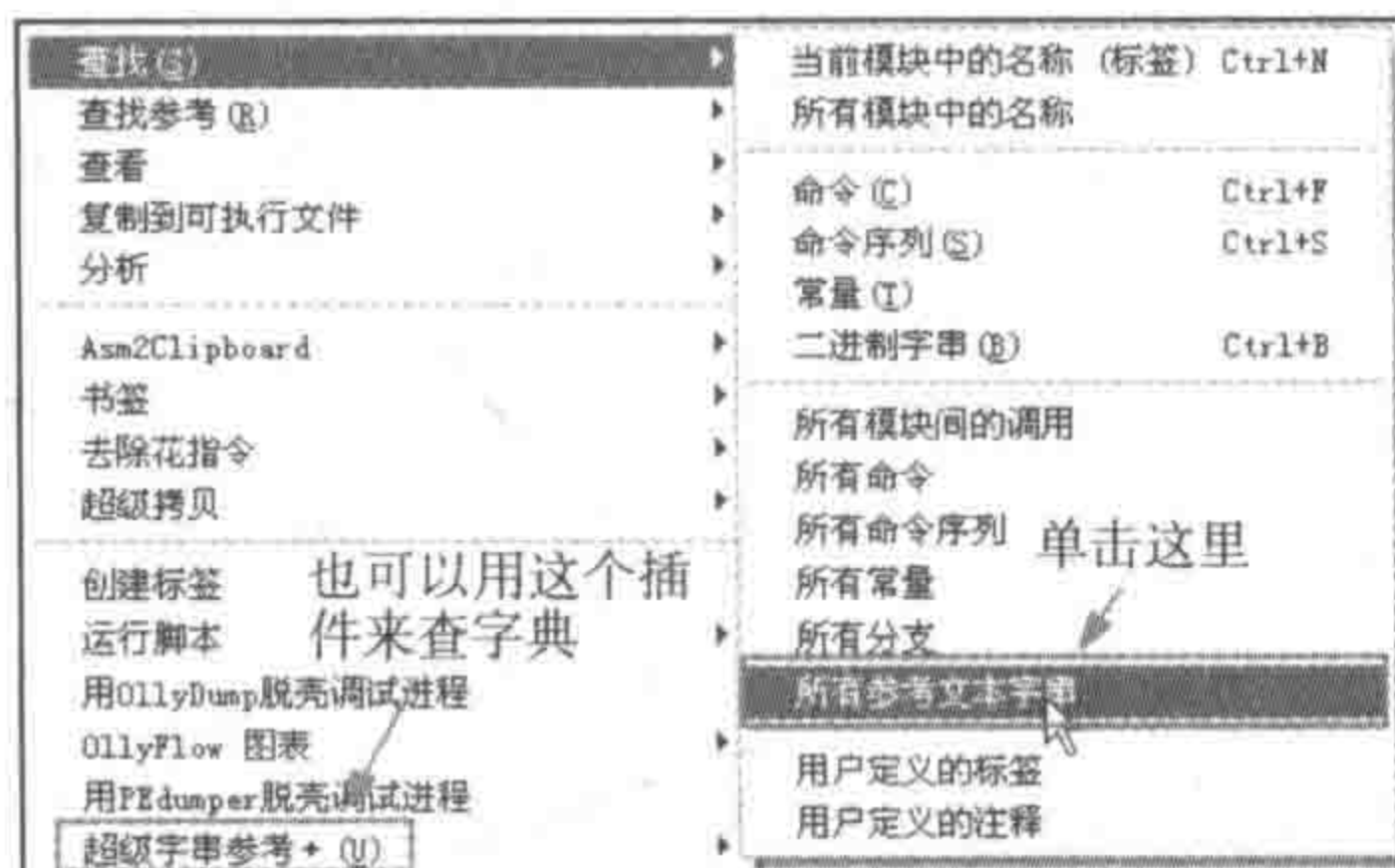


图 9-16 字符查找操作

还是以 test.exe 这个例子为测试用例，用 OllyICE 加载后的界面如图 9-17 所示。

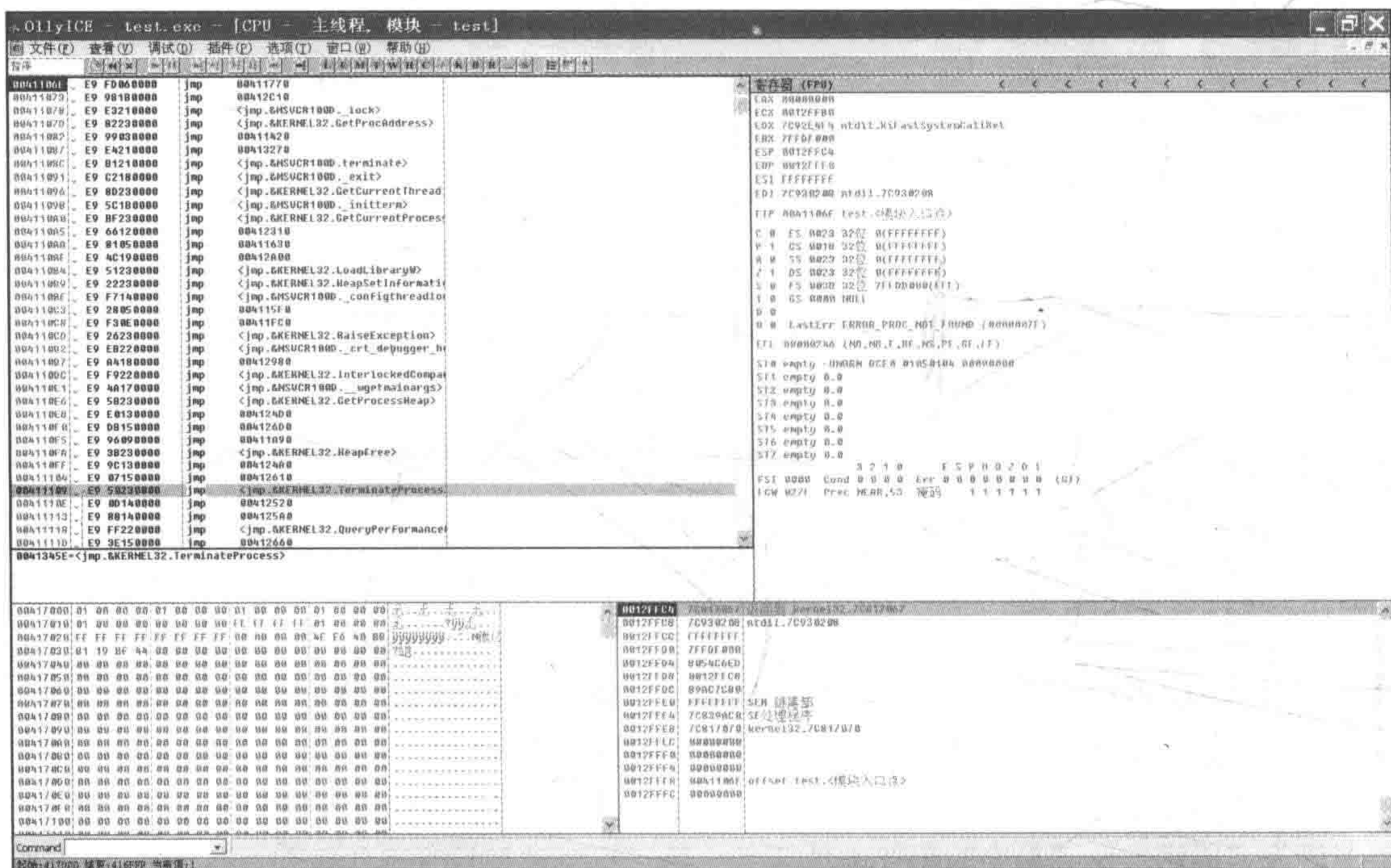


图 9-17 OllyICE 加载 test.exe 后的界面

在位置 0x00411A1E 处右击，在弹出的菜单中选择“断点”→“切换”来设置断点，然后在“调试”菜单中选择“运行”，调试器会执行到位置 0x00411A1E 处停止，如图 9-18 所示。

下面可以通过单步步过的方式逐条执行汇编指令，在右侧的寄存器窗口和下面的堆栈窗口观察指令执行时的情况。

继续在地地址 0x00411A43 处设置断点，观察 printf 函数的执行情况，从而完成调试与分析。



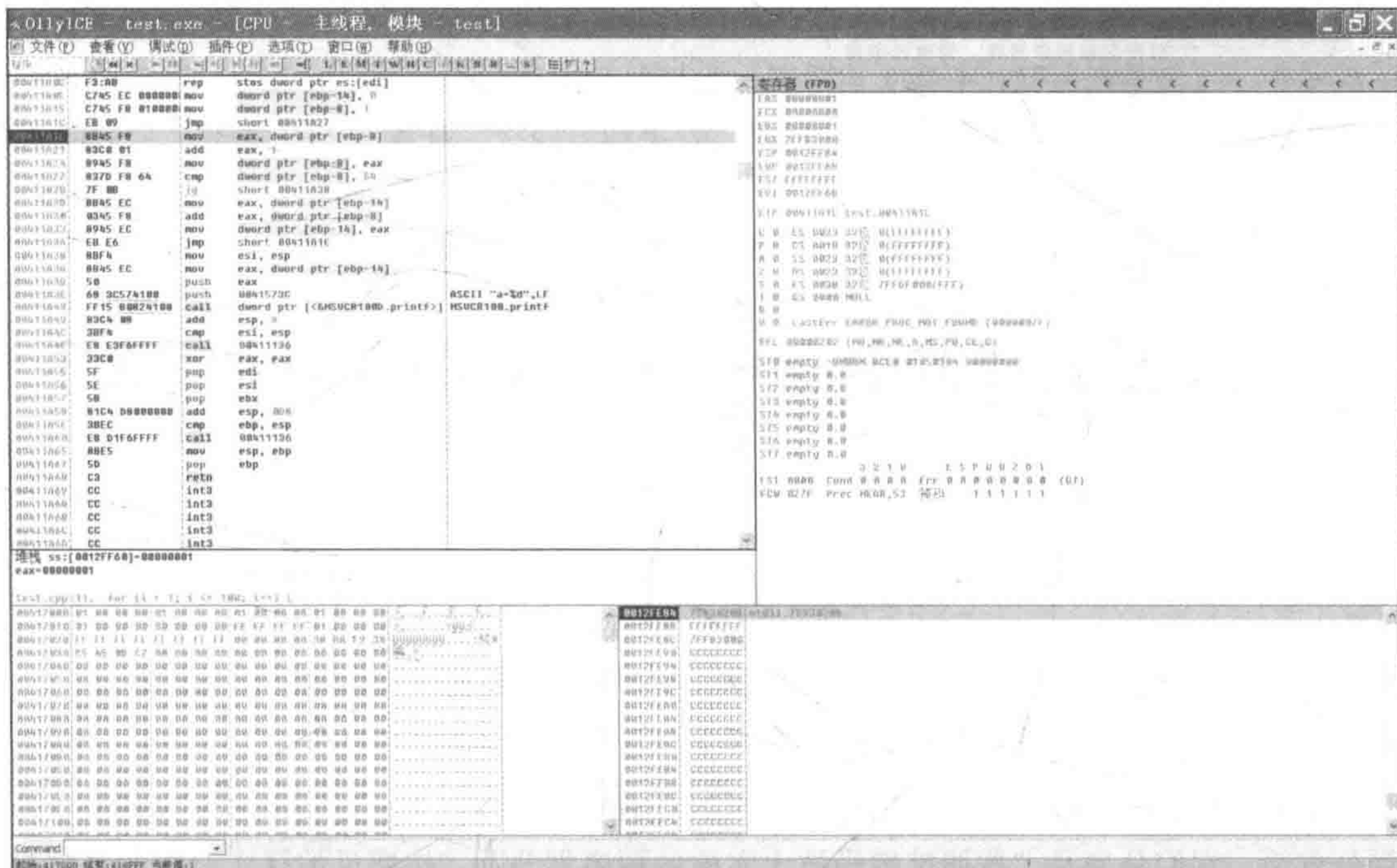


图 9-18 运行到断点的界面

## 9.5 本章小结

本章首先简要介绍了复杂指令系统和精简指令系统，然后剖析了机器指令及其格式、汇编指令及其描述；接下来介绍了 SLED 通用编解码语言，并给出了 x64 和 IA64 的 SLED 描述；然后，对反汇编的过程，特别是线性扫描和行进递归的反汇编过程进行了介绍；最后，结合实例详细介绍了反汇编工具 IDA 与 OllyICE 的用法。



## 反编译的中点——从汇编指令到中间表示

在笔者所著的《编译与反编译技术》<sup>①</sup>一书的第 10 章中给出过反编译基本过程的三种划分方法，它们分别是“按照反编译技术实施的顺序划分”、“按照反编译实践中的具体操作划分”，以及“按照反编译器的功能块进行划分”。在这三种划分方法中，都分别提到了有关中间代码或者中间表示的相关内容。

从反编译技术实施的顺序角度来看，反编译器的“前端”由那些机器依赖的和机器语言依赖的阶段组成。这些阶段包括词汇、句法和语义的分析，以及中间代码生成和控制流图生成。总而言之，这些阶段产生一个中间的、与机器无关的程序表示法。所产生的中间代码，主要用来实施反编译后续过程中向高级代码的转换。从这个角度分析，中间代码的生成是所谓反编译前端的最终任务和目标，也是反编译中端的素材和输入。

从反编译实践中的具体操作角度来看，前端包含文件装载、指令解码、语义映射等操作步骤。也就是说通过前端的处理，机器语言被处理成中间表示形式。如果这个中间表示形式设计得好，在这一部分就可以屏蔽所有源机器的特征信息，甚至可以将多种不同的源机器语言映射为统一的中间表示语言，加强反编译器的通用性。从这个角度分析，中间表示不仅可以用作反编译中后续高级代码的生成，也可以在二进制翻译等更为实际的应用中直接映射成目标机器的汇编指令，顺利完成不同体系架构之间的语义映射，进而实现二进制翻译的功能。

### 10.1 中间代码生成在经典反编译器中的实际应用

在一个经典反编译器中，反编译器的“前端”将机器语言源代码翻译成适合给通用反

<sup>①</sup> 该书已由机械工业出版社出版，书号为 978-7-111-53412-9。——编辑注



编译机器 (Universal Decompiling Machine, UDM) 作分析的一个中间表示法。关于经典反编译中前端、中端和后端的划分及其具体组成, 请参考《编译与反编译技术》一书的第 10 章的内容。如图 10-1 所示是“中间代码生成”阶段、“语义分析程序”和前端的最后阶段“控制流图生成”的关系图。图中所示的“中间代码生成”工作, 目的是采用一个与目标语言无关的表示法 (即通常所说的“中间表示”), 使得我们只需为不同的目标语言编写一个后端附加到反编译器上就可以产生各种不同的反编译结果。

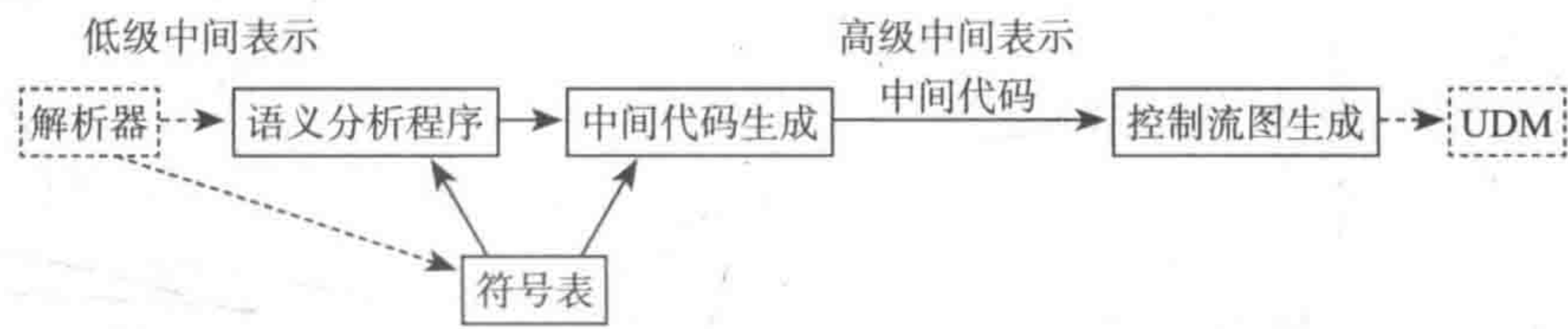


图 10-1 中间代码生成器的交互

在本节作为示例的经典反编译器的实现过程中, 除了用到本章开头提到的“中间表示”以外, 还用到了一种与之类似的“低级代码中间表示”。虽然这种相对较为“低级的中间表示”并非反编译的中点, 但是由于两种中间表示十分相似, 且相辅相成, 因此在本章的介绍中也一并加以说明。

在图 10-1 所示的反编译阶段中, 会分别用到低级和高级两种中间表示, 也就是所谓的“两步法”: 首先用低级的中间表示法来表现机器语言程序。在这个表示法里面可以进行成语分析和类型传播, 并从中产生汇编代码 (这是适合反汇编器使用的一个中间代码, 因为它不对代码作高级分析)。然后把这个表示法转换成一个高级的中间表示法, 它适合用来生成高级语言。该表示法需要具有足够的一般性, 以便将来可以生成高级语言的代码。

10.1.1 低级中间代码

低级的中间表示法与目标机器的汇编语言很相近, 很适合用来表现低级中间代码, 这样我们可以对代码进行语义分析并且从中生成汇编语言程序。对于每一条完整的机器语言指令, 中间代码必须有对应的指令。复合的机器指令也必须用一条中间指令表现。

例如, 在图 10-2 中, 机器指令 B720 对应的中间指令是 `mov bh, 20`。机器指令 2E 及其后的 FFEFC006 (即一条用 CS 段前缀覆盖的 `jmp` 指令) 用一条

`jmp` 指令代替, 显式地使用寄存器。而最后, 复合的机器指令 F3A4 等同于汇编指令 `REP` 和 `movs di, si`。这两条指令

机器指令	B720	2E FFEFC006	F3 A4
低级中间代码	↓ <code>mov bh, 20</code>	↓ <code>jmp cs:06C0[bx]</code>	↓ <code>rep_movs di, si</code>

图 10-2 低级中间指令 - 例子

只用一条中间指令 `rep_movs` 表示, 就把数据移动的目的寄存器和源寄存器都清楚地表示出来了。



### 低级中间代码的实现

低级中间表示法用一个四元式实现一条指令，显式地表现它所使用的操作数，如图 10-3 所示。opcode 字段保存低级的中间操作码，dest 字段保存目的操作数（即一个标识符），src1 字段和 src2 字段保存指令的源操作数。有些指令不使用两个源操作数，也就是只使用 src1 字段。

例 1：一条 `add bx, 3` 机器指令用四元式表示如图 10-4 所示。

其中寄存器 `bx` 是目的操作数和第一个源操作数，常数 `3` 是第二个源操作数。

例 2：一条 `push cx` 机器指令表示如图 10-5 所示。

其中寄存器 `cx` 是源操作数，寄存器 `sp` 是目的操作数。在例 2 中，读者可以看到，由于仅需要一个源操作数，因此 `push` 指令的四元组表示并没有使用第二个源操作数字段。

opcode	dest	src1	src2
--------	------	------	------

图 10-3 四元式的一般表示法

add	bx	bx	3
-----	----	----	---

图 10-4 加法指令的四元组表示法示例

push	sp	cx	\
------	----	----	---

图 10-5 `push` 指令的四元组表示法示例

### 10.1.2 高级中间代码

三地址代码是三地址机器汇编代码的一般形式。由于三地址代码可以构建程序的抽象语法树，（Abstract Syntax Tree, AST），所以这个中间代码最适合在反编译器中使用。在反编译过程中，我们能够在数据流分析期间重建程序完整的 AST。

三地址指令的一般形式如“`x := y op z`”形式。

其中 `x`、`y` 和 `z` 是标识符，`op` 是一个算术运算符或逻辑运算符。结果地址用 `x` 表示，而两个操作数地址分别用 `y` 和 `z` 表示。

#### 1. 三地址语句的类型

三地址语句与高级语言语句很相似。假如数据流分析要重建程序的 AST，三地址指令不仅提供单个标识符，而且提供表达式。一个标识符可以看作一个表达式的最小形式。

各种指令类型如下：

类型 1: `asgn <exp>, <arithExp>`

赋值指令，把一个算术表达式赋给一个标识符或表达式（即标识符用一个表达式表现，比如对一个数组的索引）。这个语句可表示三种不同类型的高级赋值指令：

- `x := y op z`，其中 `x`、`y` 和 `z` 是标识符，`op` 是一个二元算术运算符。
- `x := op y`，其中 `x` 和 `y` 是标识符，`op` 是一个一元算术运算符。
- `x := y`，其中 `x` 和 `y` 是标识符。

在数据流分析之后，算术表达式不仅表现一个二进制运算，而且是一个由算术运算符和标识符组成的、完整的分析树。

在这个意义上讲，一个有返回值的子程序（即函数）也被看成一个标识符，因为对它的调用将返回一个结果，并赋给另一个标识符（例如 `a := sum(b, c)`）。



**类型 2: jmp**

无条件跳转指令，除了跳转的目标地址之外不关联任何表达式。这条指令使控制转到目标地址。因为该地址会通过含有这条指令的基本块的出边予以说明，所以没有显式地作为该指令的一部分。这条指令等价于如下高级指令：

```
goto L
```

其中 L 是跳转的目标地址。

**类型 3: jcond <boolExp>**

条件跳转指令，带有一个与之关联的布尔表达式，由它判定分支是否转移。该布尔表达式的形式为  $x \text{ relop } y$ ，其中  $x$  和  $y$  是标识符，relop 是一个关系运算符，比如  $<$ 、 $\geq$ 、 $=$ 。这个语句等价于如下高级语句：

```
if x relop y goto L
```

在这个中间指令中，目标分支地址 (L) 和直通地址（即下一条指令的地址）不是该指令的一部分，因为在控制流图中它们会通过含有这条指令的基本块的出边予以说明。这就意味着，分析中还要结合控制流图才能得到该语句的完整语义。

**类型 4: call <procId> <actual parameters>**

调用指令，表现一个子程序调用。子过程标识符 (<procId>) 是一个指针，它指向被调用子过程的流向图。其中，实际参数的列表要等到数据流分析期间才被建立。如果所调用的子程序是一个函数，它还定义了持有返回值的寄存器。在这种情况下，该指令等价于  $\text{asgn } \langle \text{regs} \rangle, \langle \text{procId} \rangle \langle \text{actual parameters} \rangle$ 。

**类型 5: ret [<arithExp>]**

返回指令，确定一个子过程在一条路径上的终点。如果子程序是一个子过程，就没有返回值；如果是函数就有返回值。

除了上述 5 种常规类型以外，还有两种被称为伪高级中间指令的类型。之所以被称作“伪高级中间指令”，原因是这两类指令只在数据流分析中作为中间指令出现，而在数据流分析结束时被清除。

这两种特殊类型如下：

**类型 6: push <arithExp>**

进栈指令，把有关算术表达式放在一个临时栈上。

**类型 7: pop <ident>**

退栈指令，取出临时栈顶端的表达式或者标识符，而且将它赋给标识符 ident。

**2. 高级中间代码的实现**

高级的中间表示法用三元组实现。在一个三元组中，有两个显式的表达式和指令操作码，如图 10-6 所示。result 和 arg 字段是指向一个表达式的指针，它的最小形式是一个标识符，

opcode	result	arg
--------	--------	-----

图 10-6 三元组的一般表示法



指向符号表。

(1) 赋值语句的三元组表示

赋值语句  $x := y \text{ op } z$  用三元组表示，那么各部分的含义为：

- opcode 字段是 asgn 操作码。
- result 字段中存放的是一个指向标识符  $x$  的指针，这个指针进而指向存放在符号表中的标识符  $x$ 。
- arg 字段中存放的是一个指向某二进制表达式的指针，该表达式用一个抽象语法树表示，语法树中的两个指针分别指向符号表项目  $y$  和  $z$ 。

所有字段含义的描述如图 10-7 所示。

(2) 条件跳转语句的三元组表示

类似地，条件跳转语句  $\text{if } a \text{ relop } b$  用一个三元组表示，那么各部分的含义为：

- opcode 字段是 jcond 操作码。
- result 字段用一个指针指向一个测试关系的抽象语法树。
- arg 字段未使用。

所有字段含义的描述如图 10-8 所示。

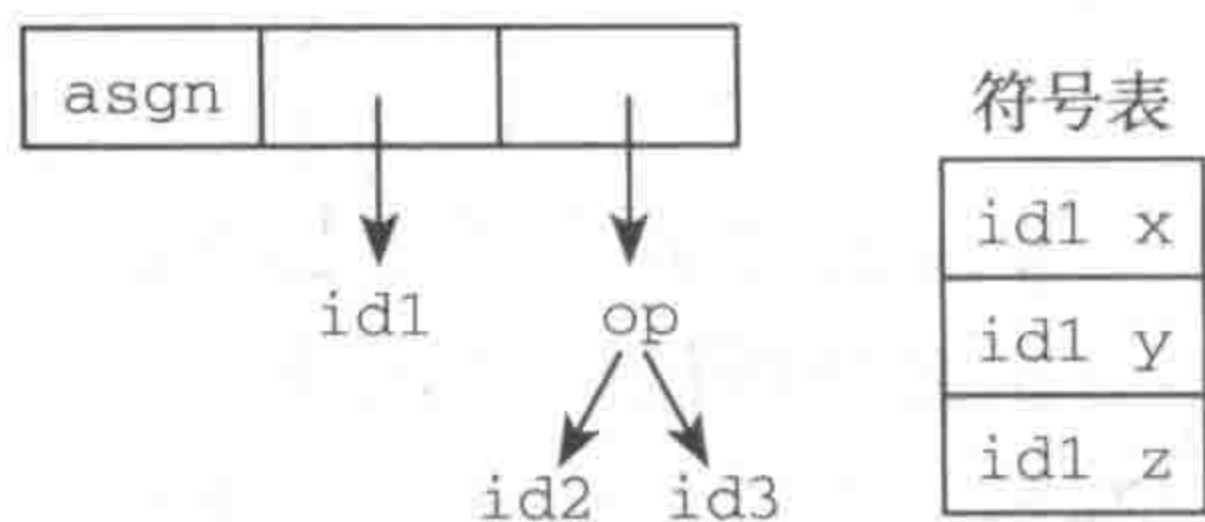


图 10-7 赋值语句的三元组表示

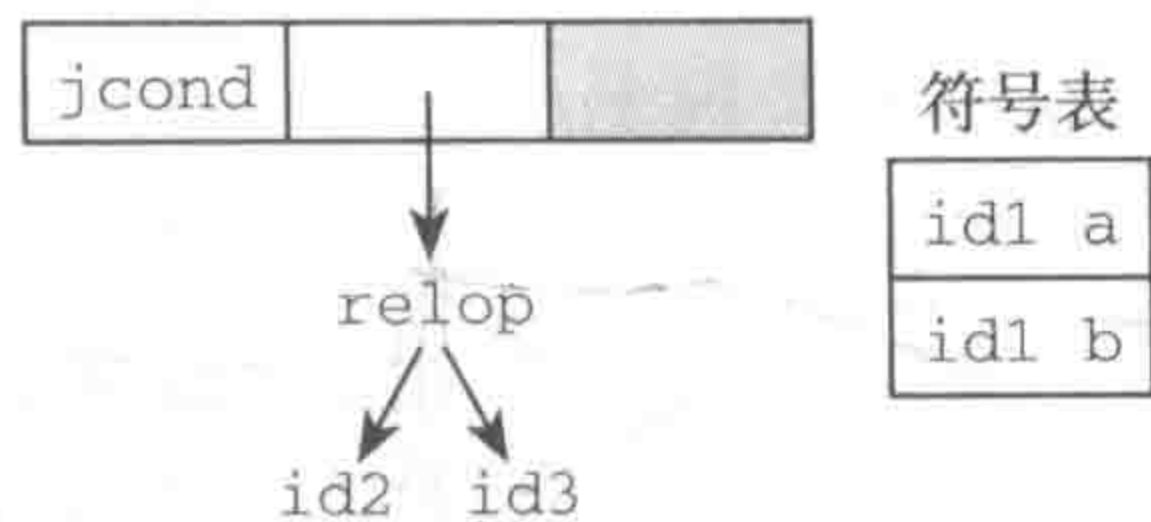


图 10-8 条件跳转语句的三元组表示

(3) 无条件跳转语句的三元组表示

无条件跳转语句  $\text{goto } L$  用一个三元组表示，那么各部分的含义为：

- 字段 result 不使用，设为空。
- 字段 arg 不使用，设为空。
- opcode 字段被设置为 jmp。

所有字段含义的描述如图 10-9 所示。

(4) 子过程调用语句的三元组表示

子过程调用语句  $\text{procX}(a, b)$  用一个三元组表示，那么各部分的含义为：

- opcode 字段存放 call 操作码。
- result 字段指向保存在符号表中的子过程名字。
- arg 字段用于子过程参数，它是一连串指向符号表的参数。

所有字段含义的描述如图 10-10 所示。



图 10-9 无条件跳转语句的三元组表示

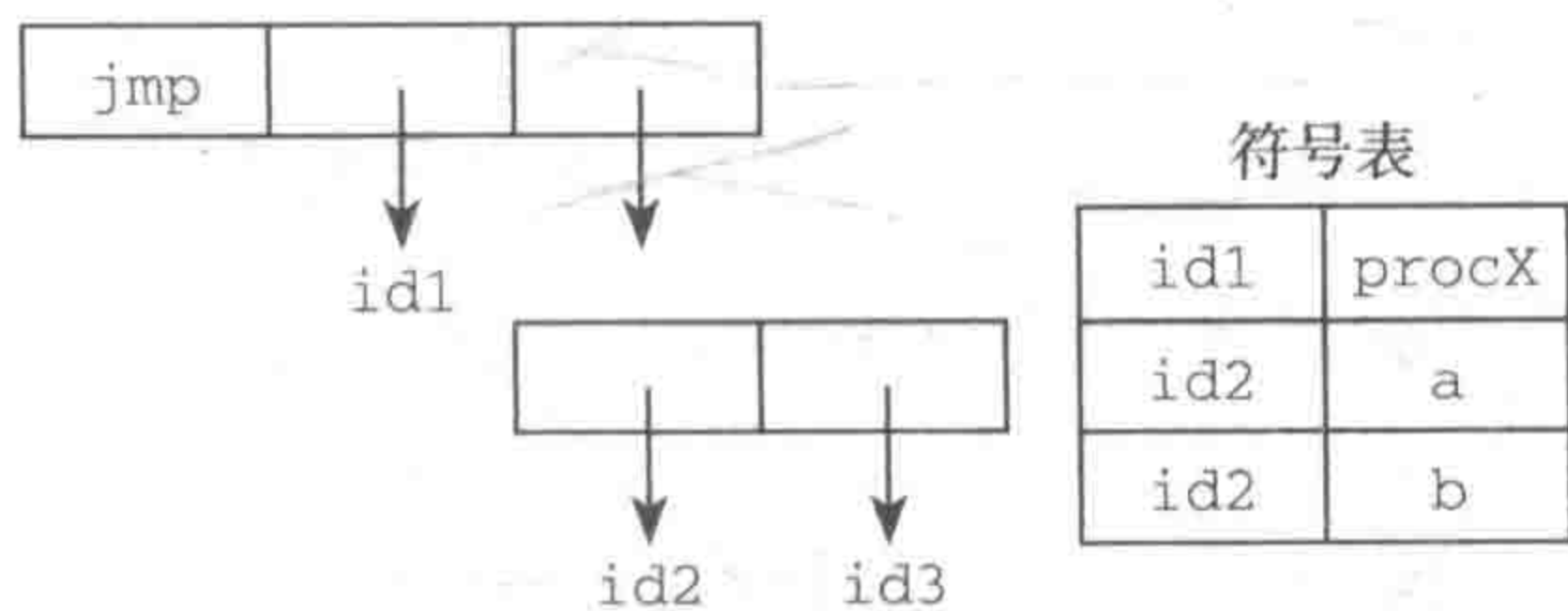


图 10-10 子过程调用语句的三元组表示



### (5) 返回语句的三元组表示

子过程返回语句 `ret a` 用一个三元组表示, 那么各部分的含义为:

- opcode 字段存放 `ret` 操作码。
- result 字段用于标识符 / 表达式。
- arg 字段未使用。

所有字段含义的描述如图 10-11 所示。

### (6) push 语句的三元组表示

伪高级指令 `push a` 用一个三元组表示, 那么各部分的含义为:

- opcode 字段存放 `push` 操作码。
- arg 字段用于要入栈的标识符。
- result 字段未使用。

所有字段含义的描述如图 10-12 所示。

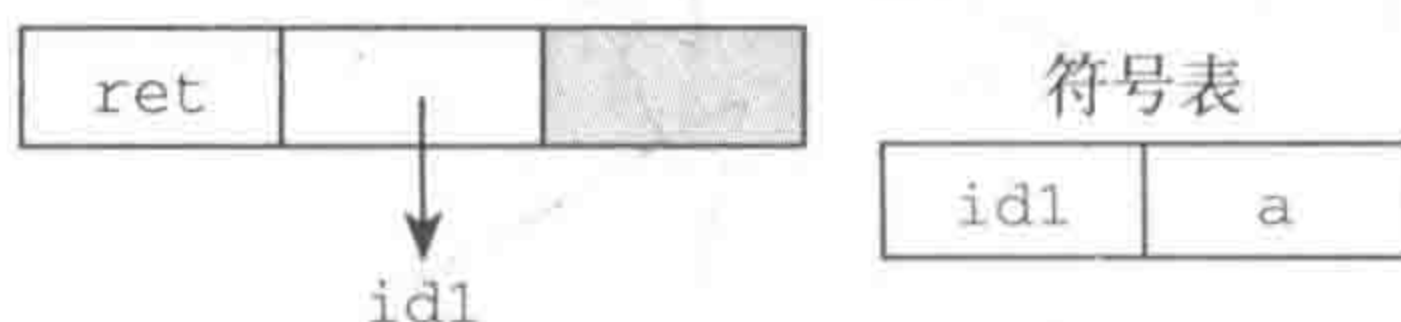


图 10-11 返回语句的三元组表示

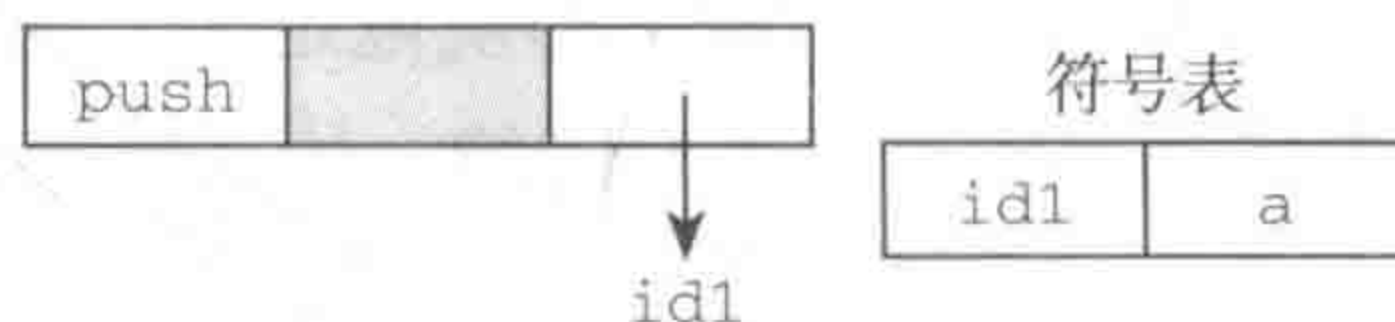


图 10-12 push 语句的三元组表示

### (7) pop 语句的三元组表示

类似地, 出栈指令 `pop a` 语句用一个三元组表示, 那么各部分的含义为:

- opcode 字段存放 `pop` 操作码。
- result 字段用于存放标识符。
- arg 字段多数时候都是留空, 并不使用 (只有在数据分析阶段, 用 `pop` 要弹出的表达式填充该字段)。

所有字段含义的描述如图 10-13 所示。

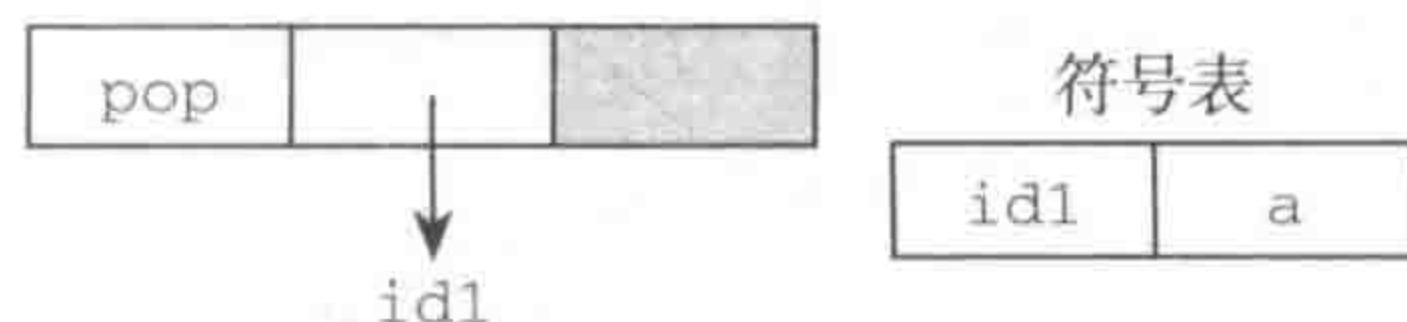


图 10-13 pop 语句的三元组表示

## 10.2 中间表示从设计到应用的具体实例

从目前计算机软件技术发展的现状和趋势来看, 单一功能的反编译器适用面十分有限。而与反编译密切相关的软件分析、虚拟机和二进制翻译等技术则展现出蓬勃的生命力。本节所讲述的“中间表示应用实例”中所包含的方法和技术既可以应用到单纯的反编译器构造中, 为后续高级代码生成奠定基础; 也可以应用到多目标的二进制翻译器中, 实现源平台向目标平台机器指令在语义层面等价转换的功能。通俗地讲, 我们以在一个“基于反编译技术实现的二进制翻译器”的构造过程中, 中间表示生成的具体方法为实例, 以实践的方式说明汇编指令是如何转换成中间表示的。



等价语义转换的思想是二进制翻译的核心,即实现不同指令集之间指令序列的完全等价变换。这部分转换的主要工作就是设法将源平台的汇编指令用一种中间表示方法等价描述出来。如果后期所做工作以反编译为目的,那么该中间表示就成为反编译过程的中点;如果后续所做工作以二进制翻译为目标,则该中间表示也能够被用来完成向目标平台的语义等价转换。但是无论出于哪种目标考虑,都须首先对多种源体系架构的汇编语言采用统一的规范进行语义描述。

通过研究发现,不同指令集体系结构(ISA)中都包含功能相近的指令类别,比如存取指令、分支跳转指令等。这类指令的功能相近,除了格式和名称不同外,其余细节差距微小。因此,这类指令可以通过规范定义而形成统一的表示方式。而那些差异较为明显的指令一般比较复杂,但是几乎所有复杂指令都可以通过简单指令的组合而实现。出于上述考虑,本节提出了一种统一最小语义描述语言(Unified Minimal Semantic Description Language, UMSDL),并使用该语言对反编译过程或者二进制翻译中的各种源平台的汇编指令进行了统一最小语义的规范定义,进而设计和实现了可兼容多源平台的中间表示方法,并能够实现其向目标平台的统一语义转换。

本节所述的统一语义描述的基本流程是:通过解构“指令操作”的基本组成,定义不可再分的“最小操作”(Minimum Operation, MO);通过分析多源平台的指令寻址模式,抽取最为基础的寻址操作,定义多源平台都具备的“最小可用寻址模式”(Minimum Available Addressing Mode, MAAM);基于MO和MAAM以及数据基本单元的统一描述,实现对统一最小语义的规范定义;最后,在上述描述基础上,构建基于“有效最小操作”(Effective MO, EMO)划分的中间表示形式,完成对多源平台汇编语言的统一语义描述。

### 10.2.1 指令基本组件描述

除了只有操作码的特殊指令以外,大部分指令都是由三种基本组件构成,即指令的操作、操作所引用或改变的数据,以及存取数据的方式。这三个基本组件分别对应着三类基本描述:指令操作描述、寻址模式描述和操作数描述(下文提及的MO是针对指令操作的描述;MAAM是针对指令寻址模式的描述)。三种基本描述构成了UMSDL的一个表达式,而一条指令由一个或一组UMSDL表达式联合描述。

#### 1. MO分类和指令操作的描述

MO大致可以划分为“3+1”类,前3类为流程控制类、数据运算类、数据传递类;有一些MO不属于上述分类,也不值得单独划分为一类,后续描述中将它们归为“附加类”。而对于少数无法由MO组合实现的复杂操作(或者是某架构独有的操作),则称为非最小操作(Non-MO),在语义描述时需要进行特殊处理。

##### (1) 流程控制类

流程控制类操作(Flow Control Class Operation, FCCO)是指可以改变程序的控制流程的操作。此类操作通常发生在过程调用与返回、条件(无条件)跳转时。FCCO发生时,程



序计数器 PC 的值将发生改变。

该类操作可描述为：

$FCCO \in \{opJump, opCall, opRet\}; opJump \in \{(\uparrow)\}; opCall \in \{(\rightarrow)\}; opRet \in \{(\leftarrow)\}$

## (2) 数据运算类

不同源平台的数据运算类操作 (Data Computing Class Operation, DCCO) 都可以分成算术操作、逻辑操作和比较操作。这一类指令数量众多, 在指令集中占据较高比例, 在程序运行时也占用较长的运行时间。因此在设计语义描述语言时, 需要对数据运算类操作进行筛选, 进而定义出数据运算类操作的 MO。在选择作为 MO 的数据运算类操作时, 需要考虑以下两点: 一是要包含基本运算操作, 如加、减、乘、除运算或基本逻辑运算; 二是要针对高频使用的组合运算, 比如“不大于”, 即小于且等于操作, 虽然可以由小于和等于两种 MO 组合表示, 但为了描述简洁也应将此组合运算定义成特殊 MO (Special MO, SMO)。为此, 本节定义三条筛选数据运算类 MO 的规则如下:

R1: { 种类应足以实现指令集中的操作 };

R2: { 为了便于在目标机器上实现, 种类也不能过多 };

R3: { 考虑翻译过程效率, 使用几率大的组合操作应该定义为单独的 SMO };

综上, 该类操作可描述为:

$DCCO \in \{opArith, opLogic, opCmp\};$

$opArith \in \{(+), (-), (*), (/), (\%), (f+), (f-), (f*), (f/)\};$  // 包括整数和浮点数的加减乘除

$opLogic \in \{(\&), (|), (\sim), (\wedge), (<<), (>>), (r<<), (r>>)\};$  // 包括与、或、非、左右(循环)移位

$opCmp \in \{ (=), (\neq), (>), (>=), (<), (<=) \}$  // 包括等、不等、大于、不小于、小于、不大于

## (3) 数据传递类

数据传递类操作 (Data Transmission Class Operation, DTCO) 即实现数据在存储单元间 (寄存器 R、内存 M) 传递。按照数据传递的源和目的不同, 可将数据传递类指令分为  $R \rightarrow R$ 、 $R \rightarrow M$ 、 $M \rightarrow R$ , 以及  $M \rightarrow M$  四类。本节给出了适合这四类数据传递指令的 MO 的统一定义, 称为“赋值 MO”。

赋值 MO 可描述为:  $DTCO \in \{opAssign\}; opAssign \in \{(:=)\}$

## (4) 附加类

为降低描述复杂度, 仅选择四种使用率高的操作作为附加类的 MO: 取址操作、有符号运算标志、连续比特位的提取操作和条件选择操作。

附加类操作 (Attach Class Operation, ACO) 可描述为:

$ACO \in \{opAddr, opSign, opBitExtra, opIf\};$

$opAddr \in \{(a[ ])\}; opSign \in \{(!)\}; opBitExtra \in \{(@<j, k>)\}; opIf \in \{(?:)\}$

## (5) 非最小操作类 (Non-MO, NMO)

该类操作有的可以用 MO 的组合表示, 但太过复杂, 在二进制翻译中不宜使用; 有的



则无法用上述定义的 MO 来表示，如 x86 架构下的标志位变化操作。针对此类操作，描述语言提供外部解决接口，也就是支持用 C 语言编制函数来模拟此类复杂动作。比如 x86 架构下执行一个 opArith (+) 操作，将导致标志位寄存器的变化，则可以定义一个函数接口来模拟：flags := ADDFLAGS32( tmp1, op1, op2 )。

2. MAAM 和寻址模式的描述

不同源体系架构所支持的寻址方式不尽相同，经统计大致包含 10 种常用类型，分别为：立即寻址、直接寻址、寄存器寻址、按位寻址、存储器间接寻址、寄存器间接寻址、基址加变址寻址、比例变址寻址、寄存器相对寻址，以及相对基址加变址寻址。其中，前 3 种寻址方式使用频率高，并且不能由其他寻址方式组合构成，可将其称为“最小可用寻址模式”，即 MAAM。其余 7 种寻址方式可由最小可用寻址模式组合构成。为与前面关于基本操作数描述中表达方式进行区别，最小可用寻址模式用大写字母 I、M、R 引导，具体描述方式如表 10-1 所示。

表 10-1 MAAM 规范描述

寻址方式	符号化	说明
A1：立即寻址	I[i[x]]	以立即数 i[x] 直接构成操作数
A2：直接寻址	M[m[x]]	以 m[x] 为内存地址取操作数
A3：寄存器寻址	R[r[x]]	以寄存器值 r[x] 为操作数

以 MAAM 寻址方式作为组件可以描述其他复杂的寻址方式，如寄存器间接寻址就可以用 A2+A3 的方式描述为复合寻址方式，如 A4:M[A3]=>A4:M[R[r[x]]]。采用类似方法可得其他寻址方式的复合表示，如表 10-2 所示。

表 10-2 复合寻址方式的描述

复杂寻址方式	符号化	复合方式
A4：寄存器间接寻址	M[R[r[x]]]	A2+A3
A5：存储器间接寻址	M[M[r[x]]]	A2+A2
A6：寄存器相对寻址	M[R[r[x]]+ I[y]]	A2+A3+A1
A7：基址 + 变址寻址	M[R[r[x]]+R[r[y]]]	A2+A3+A3
A8：相对 + 基址 + 变址寻址	M[R[r[x]]+R[r[y]]+ I[z]]	A2+A3+A3+A1
A9：比例变址寻址	M[R[r[x]]+ I[x]*R[r[z]]]	A2+A3+A1*A3
A10-1：存储器位寻址	M[m[x]] @<I[y], I[z]>	A2@<A1,A1>
A10-2：寄存器位寻址	R[r[x]]@<I[y], I[z]>	A3@<A1,A1>

3. 操作数描述

指令操作数由三种基本组件构成，分别是立即数、寄存器、内存值。立即数在指令中使用数字常量表示；寄存器则按照整数序号递增的顺序依次命名，这样做的好处是，既命名简单，又可以提供不限个数的寄存器表示范围；内存值按照惯例由存储空间线性字节地址标示。本节通过对相关基本组件的描述，实现对操作数的描述，具体描述见表 10-3。



表 10-3 指令操作数符号化规范描述

操作数类型	符号化	说明
Type1: 立即数	$i[x]$ (或简写成 $x$ )	$i[]$ 标示操作数, $x$ 为立即数具体数值
Type2: 内存地址	$m[x]$	表示数值 $x$ 为内存地址
Type3-1: 独占型通用寄存器	$r[x]$ (或简写成 $rx$ )	$r[]$ 表示取寄存器, $x$ 为寄存器编号
Type3-2: 共享型通用寄存器	$r[x]@<j,k>$	$@<j,k>$ 表示取寄存器中第 $j$ 到第 $k$ 位作为寄存器值
Type3-3: 专用寄存器	单独符号化表示: $\%PC, \%flag$	“ $\%$ ” + 指定的名称来表示专用寄存器
Type4: 操作数位限制	$\{x\}$	“ $\{\}$ ” 强制操作数以低 $x$ 位参加计算, $x \in \{1,4,8,16,32,64\}$

根据表 10-3 的定义, 可以实现对具体架构下的操作数定义。例如, “Type3-2 型描述” 适用 x86 中 8 个可共享的通用寄存器, 如 EAX、EBX 等; 以 32 位 ESI 寄存器为例, 该寄存器的低 16 位为 SI 寄存器, 若对 ESI 指定序号 30, 则 ESI 符号化为  $r[30]$ , 相应的 SI 符号化为  $r[30]@<0,15>$ 。

### 10.2.2 用 UMSDL 描述指令语义

UMSDL 是用来统一描述多源一体化二进制翻译中异构源平台指令语义的描述语言。该语言设计的基本思想是: 定义描述指令语义的最基本操作; 基于对最基本操作的描述, 给出 UMSDL 的 EBNF 范式表示; 在规范表示的基础上, 实现对多源指令语义的统一中间表示方式的定义。

#### 1. 描述指令语义的基本操作

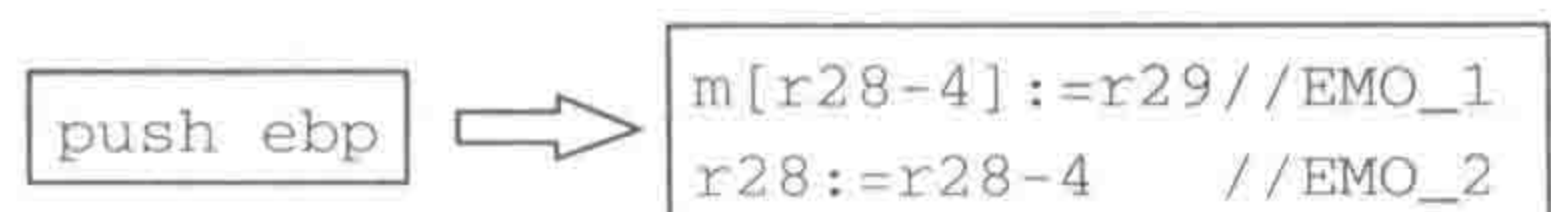
“指令效能”是指令执行后对机器状态改变的总和, 而指令语义可以体现单条指令的指令效能。机器状态的总和这一说法比较抽象, 在描述指令语义的时候, 可以用“存储单元内容”的改变来刻画机器状态的改变。这里提到的存储单元的内容既包含存储数据的内存, 也包含存储数据的寄存器(包含特殊寄存器, 比如 x86 的 PC 寄存器)。相关研究成果的文献在提到有关程序行为的理解时, 也佐证了“机器状态由存储单元内容来决定”这一思想基础。从微观看程序是由一条条指令组成, 其行为也是由一条条指令执行来体现。因此, 通过描述指令执行对存储单元内容的改变, 是可以刻画“指令效能”的。

由于一条指令的执行可能伴随着多个存储单元内容的改变, 而这样的改变是由若干 MO 组合而成。在这些 MO 中, 有些确实是改变了某一个存储单元, 而另外一些则为存储单元内容的改变做附加工作。如果将改变存储单元的 MO 定义为有效最小动作, 那么指令对存储单元的改变就是由这样一组具有先后顺序的 EMO 组合而成。与之对应的, 没有改变存储单元值的最小动作被定义为普通最小动作 (General MO, GMO)。

以一条 x86 中简单的 “push ebp” 指令为例, 说明上述有关指令语义、指令效能和有效最小动作的定义, 如图 10-14 所示。



图 10-14 中单条指令“push ebp”的效能是将通用寄存器 ebp 的值保存在栈内, 如果 ebp 用 r29 表示, 则与压栈动作相关的通用寄存器 esp 用 r28 表示。该指令效能从指令语义上分析, 可以分解为两条完成赋值动作的 opAssign 型 MO。其中第一条 MO 改变了 m[r28-4] 的值, 可以看成是一条 EMO; 第二条 MO 改变了寄存器 r28 的值, 也是一条 EMO。与两条 EMO 对应的是两次 r28-4 的 opArith 型 MO, 可以看到这两次 MO 明显属于 GMO, 它们为存储单元内容的改变做附加工作。



说明: 1. 两次赋值操作 (:=) 为 EMO  
2. 两次赋值操作 (-) 为 GMO

图 10-14 EMO 含义示意图

可以看出, 在描述指令语义时, 可以按照新的分类准则对 MO 进行分类, 即以 MO 是否改变存储器值为标准将 MO 划分为两大类: 有效最小动作类和普通最小动作类。接下来对 EMO 和 GMO 进一步说明。

#### 说明 1: 有效最小动作 (EMO)

改变存储单元值的 MO 为有效最小动作, 包含赋值动作、跳转动作、过程调用和返回动作, 可描述为  $EMO \in \{opAssign, opJump, opCall, opRet\}$ 。opAssign 操作直接改变存储单元值; 而 opJump、opCall、opRet 同属改变程序控制流的操作, 它们都会默认修改寄存器“%PC”的值, 为了更为简洁地描述指令语义, 将此 MO 也划分到 EMO 类。

#### 说明 2: 普通最小动作 (GMO)

除 EMO 之外的所有 MO 都划分为 GMO, 如算术运算类操作, 此类操作并不会直接体现出指令效能, 其动作结果需要通过 EMO 类动作才能传递到存储单元中。

综上所述, 一条指令由一个 (或一组) EMO 来描述, 隶属于同一条指令的 EMO 具有原子性, 也就是说它们要么全部执行, 要么全部不执行, 且该组 MO 的执行顺序不能改变。

## 2. UMSDL 的规范化表示

UMSDL 是基于 EMO 来描述多源一体化二进制翻译中计算机指令语义的描述语言, 其 EBNF 表示如图 10-15 所示, 其中图 10-15a 为基本元素定义, 图 10-15b 为复合元素定义。

<pre> Dez ::= '1'   '2'   '3'   '4'         '5'   '6'   '7'   '8'   '9' D ::= '0'   Dez F ::= [-][D]*.[D]* Dh = 'A'   'B'   'C'   'D'         'E'   'F' N ::= [-][D]*   '0x'[Dh D]*   [-]         '2*' [D]* O ::= 'm['N']'   'r['N']'         'i['N']'   'i['F']' </pre>	<pre> E ::= E opBin E         opUnary E         C '?' E ':' E         'M['E']'         'R['E']'         'I['E']'         O         O '@&lt; i['j'], i['k'] &gt;' C ::= E opCmp E  Instr ::= EMO* EMO ::= C '=&gt;' EMO           E opAssign E           opJump E           opCall E           opRet </pre>
--	--

图 10-15 UMSDL 的 EBNF 表示

图 10-15 中出现的, 且在前文中未作说明的符号将在表 10-4 中予以说明。



表 10-4 UMSDL 符号说明表

符号表示	说明	符号表示	说明
Dez	数字 1 ~ 9	E	表达式
D	数字 0	opBin	二元运算
F	浮点型常量	opUnary	一元运算
Dh	十六进制数字 A ~ F	C	比较条件
N	整型常量	=>	表示若条件成立, 则如何
O	基本数据单元		

### 3. 基于 EMO 划分的中间表示定义

依据前文所述, 利用 UMSDL 可以实现多源代码语义的统一描述, 无论源体系架构的指令如何表述, 它都可以按照规范拆分成以 EMO 为单位的若干表达式 (或者通过定义函数接口表示)。对于可以划分为 EMO 序列的指令, 其每一个被单独划分出来的 EMO 都可以按照表达式成分继续分解, 形成一棵 EMO 划分树 (emoTree), 该树以 EMO 为根节点, 以 GMO 为内部节点, 以 Type-1 型操作数为叶子节点, 图 10-14 中示例指令对应的 emoTree 如图 10-16 所示。

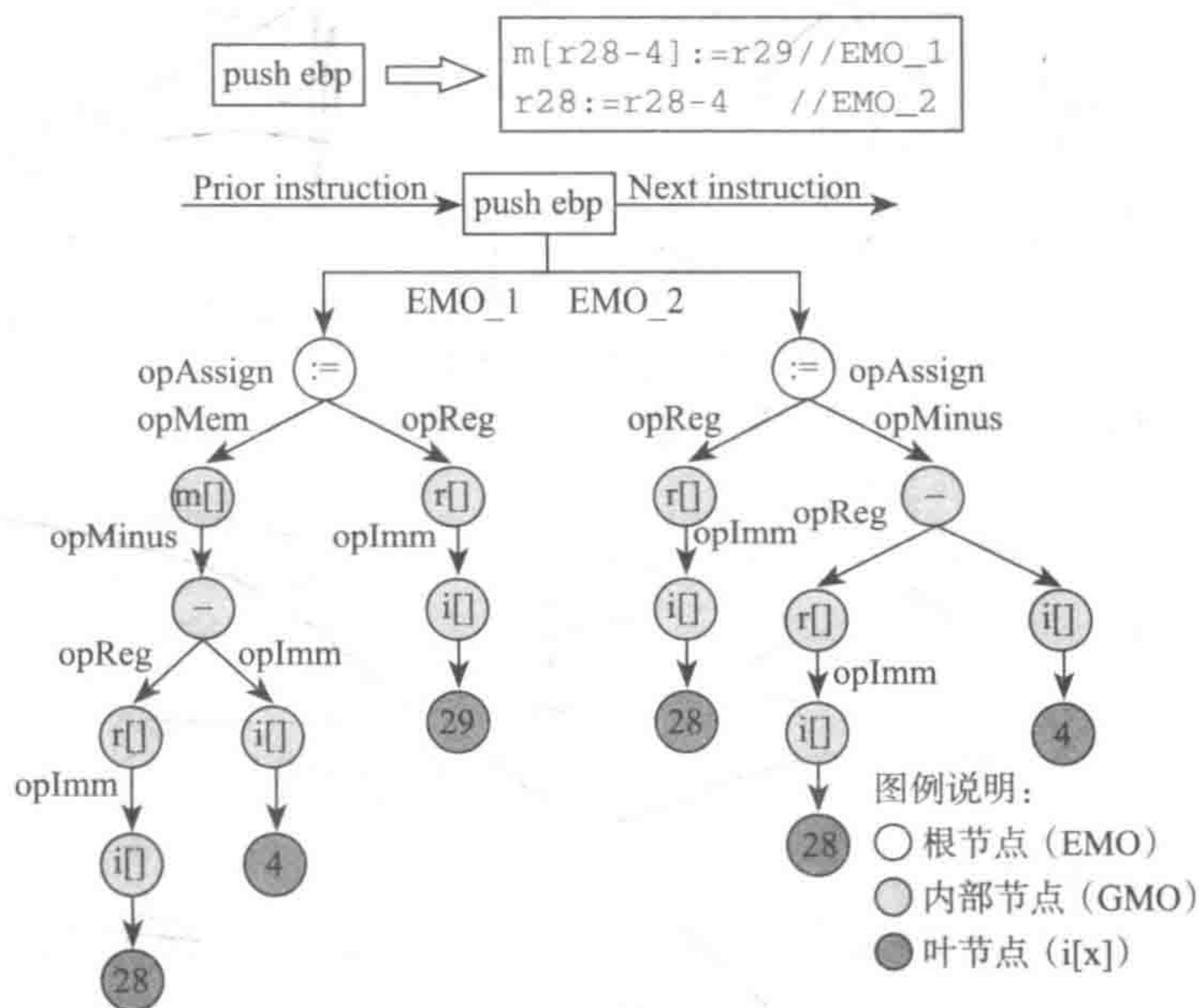


图 10-16 EMO 划分树 (emoTree)

#### (1) 常规 emoTree

图 10-16 中示例的指令 “push ebp” 是一条不含 “前导条件判断” 的普通语句, 它的指令语义可划分为两个 EMO, 而每一个 EMO 都可以单独展开成一棵 EMO 划分树。在 EMO 划分树中内部节点都是 GMO 类型的子树。



## (2) 带条件判断的 emoTree

而对于含有“前导条件判断”的特殊语句，在构建 EMO 划分树时，还有一个“前导条件”子树（Pre\_if\_subTree）的拼接动作，其具体情况和构建过程如图 10-17 所示。假设有如下两条指令顺序执行：指令 1 “Cmp eax, 20h”，指令 2 “JNZ imm-Address”，其组合含义为根据 eax 寄存器与立即数 20H 比较的结果来决定指令 2 的跳转是否执行。依据前文定义，指令 1 并不能被分解成一个 EMO，那么这样的指令其指令语义如何体现呢？

针对这样的情况，可以将指令 1 和指令 2 看作一个 EMO，也就是说该 EMO 是否发生需要依据前导条件的判断结果。因此对于类似这样的情况，应该将条件比较指令 opCmp 的 GMO 作为一棵 Pre\_if\_subTree 拼接到你相关后续指令 emoTree 的最左边分支。在绝大多数情况下，Pre\_if\_subTree 的拼接会使得原本为单支的二叉树形成一棵左右子树都完全的二叉树。在有些情况下，若原 emoTree 已经是具有左右子树的二叉树，那么 Pre\_if\_subTree 则作为最左子树插入到原 emoTree 根节点下，形成三叉树结构。

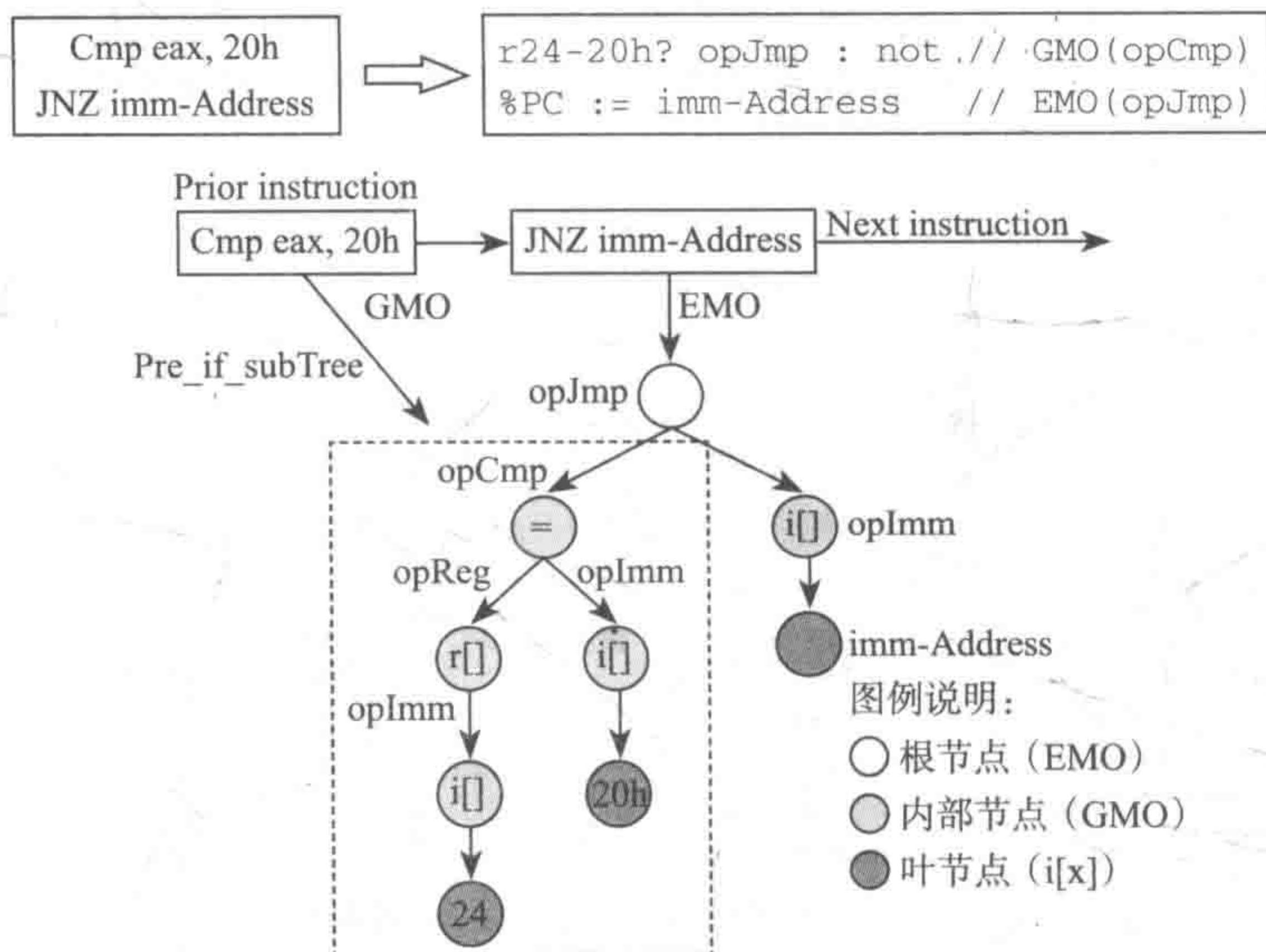


图 10-17 带前导条件子树的 EMO 划分树

## (3) 双形式对应的中间表示

本章设计的一体化翻译架构的中间表示是一种由 UMSDL 描述的，由 EMO 表达式构成的表达式序列。从图 10-16 和图 10-17 中所表示的 emoTree 关系可以推出，该表达式序列还可对应于由多棵 emoTree 构成的语义树森林。因此一种中间表示可以有两种表示形式，其中表达式序列方便阅读、理解和分析，而 emoTree 树形结构能够更加自然地表现指令的层次结构和语义特征，更方便后续语义等价转化的算法的实现。

## (4) 指令效能的链式结构

从图 10-16 和图 10-17 中的示例可见，指令按照执行逻辑顺序连接成链表，该链表节点



为表示指令信息的数据结构，包含标记指令地址及指令所属基本块等相关信息。同时指令节点将当前指令所关联的 emoTree 链接在一起，共同完成指令的中间表示。

该类节点的处理正是由前文所述分支替换的具体算法实现的，处理时需要根据 GMO 节点的操作类型，以及其所涉及的寻址方式区分对待。将 emoTree 中的每个节点映射成具体的目标平台指令操作符和操作数，最终实现输入 emoTree 所表达的语义可以等价转换。由于目标平台只支持数据在寄存器之间传递，所以目标平台的“写内存”操作是通过先将内存地址存入寄存器，然后再用该寄存器作为指令目的操作数的方式实现的。因此在“内存”类操作 GMO 节点的迭代处理中，必须将上一层递归迭代所返回的“内存值”结果统一存入寄存器中，即将所有由 UMSDL 描述的、待处理的 GmoExp 表达式，在进行指令映射后都存储在寄存器中（该寄存器的内容为 GmoExp 所表示的运算结果），并返回给调用者。

例如，若待处理的 GmoExp 表达式为“m[r28-4]”，那么它需要三层迭代处理。即首先接收由叶节点处理（具体实现在下面给出）返回的“28”和“4”两个叶表达式，其中的“28”需要交由第三层迭代处理（Handle\_G3）生成表达式“r28”。然后“r28”和“4”返回给第二层迭代（Handle\_G2），在此层将 r28-4 的值传递给临时寄存器，生成目标平台指令“ldw \$5, -4(\$15);”，其含义为用临时寄存“\$5”作为中间值返回给第一层迭代（Handle\_G1），在这一层模拟 opMem 操作，并根据表达式的左、右部关系，以“\$5”作为目的或源操作数，以便构造后续的目标平台指令。图 10-18 给出了 GMO 类节点的语义转换算法伪代码描述。

```
/*GmoExp 为输入表达式，str 为指令构造字符串流，result 为对应的 SW 表达式 */
std::string SWCode::HandleExpGmo( Exp *GmoExp, std::ostringstream &str)
{
    /* 交由叶节点类处理 */
    if(GmoExp 是叶节点类)
        return HandleLeaf(GmoExp);
    /* 当前表达式是内存寻址，对标志变量赋值，后续需要特殊处理 */
    if(GmoExp 子树根 GMO 为 opMem)
        Mem_flag=true;
    /* 根据 GmoExp 根节点的 mo 类型和几元表达式，开始迭代构造对应操作数 */
    if(GmoExp 为一元表达式)
    {
        Operand1=HandleExpGmo(GmoExp->getSubExp1(), str);
        /* 分配临时寄存器给 Operand2 */
        Operand2=SettmpReg(gettmpReg());
        /* 根据 GMO 的操作符和构造的操作数，构造 SW 指令，并将其输入 str 流中 */
        str<<PutInstruction(GmoExp->Op, Operand1, Operand2);
    }
    else if(GmoExp 为二元表达式)
    {
        Operand1=HandleExpGmo(GmoExp->getSubExp1(), str);
        Operand2=HandleExpGmo(GmoExp->getSubExp2(), str);
        /* 分配临时寄存器给 Operand3 */
        Operand3=SettmpReg(gettmpReg());
    }
}
```

图 10-18 GMO 节点分类处理算法



```
        str<<PutInstruction(GmoExp->Op, Operand1, Operand2,Operand3);
    }
    else if(GmoExp 为三元表达式 )
    {
        Operand1=HandleExpGmo(GmoExp->getSubExp1(), str);
        Operand2=HandleExpGmo(GmoExp->getSubExp2(), str);
        Operand3=HandleExpGmo(GmoExp->getSubExp2(), str);
        str<<PutInstruction(GmoExp->Op, Operand1, Operand2,Operand3);
    }
    /* 将生成的指令（或指令序列中最后一条指令）的目标寄存器作为返回值 */
    result=PutInstruction->sequence;
    /* 此时 result 所代表的寄存器里存储的是地址 */
    if(Mem_flag)
    {
        /* 为了生成形如 "ldw $5,-4($15);" 的 SW 指令，此处返回的 result_M 为临时寄存器 */
        .....
        result_M=SettmpReg(gettmpReg());
        .....
        return result_M;
    }
    else
        return result;
}
```

图 10-18（续）

(5) 叶节点的处理

叶节点作为 emoTree 的终端节点，一般由立即数构成，或者是某些其他类型常量，有时也由一个函数调用作为叶节点（表示此处的复杂处理无法展开成 emoTree，只留下扩展处理接口）。在实际处理中需要设计算法所取得的立即数类型作判别，再用目标平台对应的处理方式截取或扩展成可用立即数。处理叶节点的流程比较简单，但是由于个别数据类型在不同平台的数据表示格式差异较大，需要进行单独处理，尤其是浮点数。本节只给出叶节点处理过程 HandleLeaf() 中所有涉及的部分立即数类型的描述对照表（见表 10-5），具体的分支匹配语句就不再给出了。

表 10-5 叶节点类立即数符号对照表

立即数符号	说明
opIntConst	整数常量
opFltConst	浮点常量
opStrConst	字符串常量
opLongConst	长整型常量
opFuncConst	函数调用接口
opSpecImmConst	特殊立即数接口
.....	

10.3 本章小结

因此，不管从何种角度分析，中间表示或者说中间代码的生成都可以看成是反编译的“中间点”，本章我们重点讨论了在经典反编译器中，中间代码生成的实际应用，并且以在一个“基于反编译技术实现的二进制翻译器”的构造过程中，中间表示的生成的具体方法为实例，以实践的方式说明汇编指令是如何转换成中间表示的。



## 反编译的推进 1——数据类型恢复

在笔者所著的《编译与反编译技术》一书的第 13 章中，我们已经对有关数据的恢复技术进行了较为细致的讲述，从理论层面对数据流分析给出了系统的总结，同时又基于 IA64 体系架构讨论了针对“寄存器和内存操作”“数组和 struct 数据类型”“union 数据类型”“存取操作码对应的数据类型”等具体情况的数据类型恢复技术。

本书将从几个新的视角，针对反编译过程中数据类型恢复的一些其他问题进行具体探讨，同时对另外一类较为特殊的数据类型恢复问题给出实例说明，即函数返回值的数据类型恢复。针对常规的数据类型恢复，我们将从基于指令语义的基本数据类型分析和基于过程的数据类型分析技术两方面分别切入；而针对函数返回值类型的恢复，则通过基本块划分、基本块数据流分析等基本知识的介绍，引出传统和改进的函数类型恢复方法。所有的介绍都是基于笔者参与开发的 ITA 系统的具体实例，具有较高的可操作性和可重用性。

### 11.1 基本数据类型的分析和恢复

编译器执行类型分析的目的是为了保证程序执行的正确性。好的类型分析能够在编写代码时就发现程序中存在的错误，这样可以减少在编译、配置和运行中出现的错误。一些高级语言要求在编程的过程中为所有的变量定义类型，这样编译器就可以在编译过程中检查类型的一致性。而有的语言没有这个要求，这样不仅需要在运行时进行类型检查，还需要在编译时进行静态类型检查。

在 ITA 二进制翻译系统中，通过提升指令的语义，最终把可执行程序的语义用高级语言 C 表示。当把指令提升到中间语言表示形式时，寄存器转换为变量，指令的行为被简单的中间语言语句的行为代替，变量必须依靠自己的类型信息属性来规范它的存储格式。本



章首先对类型的一些基本概念进行说明, 然后讨论基于指令语义提升的基本数据类型分析方法, 最后重点论述基于过程分析的基本数据类型分析技术。

### 11.1.1 数据类型分析的相关概念

在高级语言中数据的类型是非常重要的概念, 它增强了程序的可读性, 有助于区别指针和常数, 在封装知识和实现面向对象编程等方面也起着重要的作用。高级语言中的类型分为两类: 基本类型和聚合类型。例如在 C 语言中, 基本类型包括 integer (int, long, long long, unsigned int 等)、char、float (float, double)、boolean、pointer (指针); 聚合类型包括 struct、array、union、enum。

在机器代码级的表示形式上, 一个聚合类型的数据将被拆成多个基本类型的数据, 其操作数的具体类型由机器指令的语义来决定。这一特性给二进制翻译中聚合类型的恢复带来了困难, 而且在大多数情况下想要完全恢复出聚合类型几乎是不可能的。

在 ITA 系统中, 当指令的语义被提升到更高级别的语言表示时, 以下四类数据需要恢复其类型信息:

- 1) 寄存器变量。
- 2) 临时变量。
- 3) 数据段中的数据。
- 4) 过程的参数和本地栈内容。

#### 1. ITA 系统中数据类型分析的依据

对于库函数, 除了可变参函数, 其他函数的参数个数是固定的, 参数的类型和返回值的类型也是固定的, 这样可以通过已知的库函数参数和返回值的类型来恢复部分数据的类型信息。

大部分数据的类型信息是根据指令的语义来提取的。除了少数指令以外, 一般情况下整型寄存器存放整数, 浮点寄存器存放浮点数, 但是由于没有专门的整数乘指令, 因此在 IA64 体系结构下进行整数乘运算时常常将整数存放在浮点寄存器里。使用“变量重命名”的方法可以用来区分这类特殊现象。对于指针变量来说, 在不能确定指针指向的类型时, 可以将其看成是 unsigned long long 的无符号长整型数据。

#### 2. ITA 系统中基本数据类型分析的重要性

基本数据类型分析是指仅仅根据单条指令的语义来分析数据的类型。反编译的过程即将一条汇编语句提升到中间表示语句, 然后从中间表示语句提升到 C 语句, 如下所示:

汇编语句 → RTL → HRTL → C 语句

在这一转换过程中, 指令中的操作数最终转换成 C 语言中的变量。为了使 C 语句正确反映指令的行为, 必须提取指令操作数的类型信息, 然后将其转换成 C 语言中变量的类型。例如:

```
ld8 r36=[r14]  →  r36= *((int64*)(*(unsigned int64*)&r14)) (11-1)
```



关于式(11-1)的具体含义将在下文中解释。

基本数据类型分析的目的是界定指令操作数的存储格式和占有内存的大小,从而正确地反映指令的语义。传统反编译中的数据类型恢复是指恢复高级程序所定义的变量类型,比如整数、浮点数、数组、结构等。这是一项非常困难的工作,因为高级语言中变量的显式定义信息在经过优化编译后的目标代码中已经消失了。高级语言中对变量名的访问在目标代码中被转换为对存储单元的访问;高级语言的数据类型到机器内存单元的映射关系是一种  $n:1$  关系 ( $n \geq 1$ ),而从存储单元到高级语言数据类型的映射关系是  $1:n$  的关系。

### 3. ITA 系统中基本数据类型和高级 C 语言数据类型的区别和联系

为了描述指令,必须对指令的操作数正确界定,所以产生了指令操作数类型的概念,也就是基本数据类型的概念。从机器的角度看,指令操作数是没有类型概念的,对于机器来说所有数据都是 0/1 串,指令通过不同的动作实现指令的语义。但为了提升指令的语义必须对指令操作数的特性进行界定,这样就有了指令操作数类型的概念,这一类型是由指令的本质决定的。例如:

addl r14=72, r1 (11-2)

r14=r1 + 72 (11-3)

式(11-2)中加法指令的语义是先对立即数 72 进行符号扩展,转换成 64 位的数据,然后与寄存器 r1 的内容进行整数相加,最后把结果放在寄存器 r14 中。而我们用 C 语言中的赋值语句来描述这条指令时(见式(11-3)),原来的寄存器转换为变量,变量 r1、r14 的类型规定为 64 位整型,这样该指令的语义与 C 语言赋值语句的语义在执行上本质是一样的。当然,并不是所有的指令都可以用 C 语言提供的语法规则来简单描述,指令操作数的类型与 C 语言中的基本类型也不存在一一对应关系。但是我们可以利用 C 语言提供的语法规则描述指令中操作数的类型。少数更复杂的指令可以用函数来实现。表 11-1 说明指令操作数类型与 C 语言中基本类型的区别与联系。

表 11-1 IA 64 指令集中指令操作数类型与 C 语言类型关系表

指令操作数分类	位数大小	对应 C 语言的基本数据类型	用途举例说明
boolean	1	boolean	谓词寄存器变量的定义
unsigned integer64	64	unsigned long long	地址变量的描述或寄存器变量的定义
integer64	64	long long	指令整型寄存器参数类型
integer32	32	int	整型寄存器参数或立即数类型
unsigned integer32	32	unsigned int	同上
char	8	char	立即数
float64	64	double	浮点寄存器参数类型
float	32	float	同上
floatmax	82	floatmax	同上
地址	64	Char*、integer*、float* 等	整型寄存器参数类型

注: integer64 表示 64 位整数, integer32 表示 32 位整数。



### 11.1.2 基于指令语义的基本数据类型分析

在 ITA 系统的实现中, 指令先提升到中间语言, 从中间语言再提升到 C 语言。在这个过程中如何转换数据类型的信息并不是问题的关键。为了更简洁地说明问题, 在这里先假设从指令直接提升到 C 语言。

基本数据类型恢复的困难来自以下三个因素:

1) 在机器级别上对数据的操作, 其实是对数据的二进制码进行操作, 此时数据并没有类型的概念。但在 C 语言描述中, 同一操作数在不同的指令中可能有不同的类型。比如一个寄存器的值在 `addl` 指令中是整数, 在 `ld` 指令中表示地址。

2) 编译器可能会重新分配寄存器, 以存放不同的数据。

3) 由于指令的操作数在 C 语言中被转换为变量, 而 C 程序中的变量必须有明确的类型定义, 因此, 在本节中我们假设翻译后生成的目标 C 程序中 64 位的整型寄存器变量为 `long` 型, 82 位的浮点寄存器变量为 `floatmax` 类型, 1 位的谓词寄存器为 `int` 型 (因为在标准 C 中没有布尔型) 等。指令被翻译为 C 语句后, 如果变量在语句中的类型与变量定义的类型不一致, 则通过强制类型转换使二者一致。

C 语言中的 “&” 和 “\*” 指针运算符为描述指令操作数类型的多变性提供了手段, 下面对其进行说明。

#### 1. “&” 和 “\*” 运算符

指针是变量的内存地址。指针变量则是专门用来存放指向某一给定类型的数据的内存地址的变量。“&” 和 “\*” 是用于指针操作的运算符。

指针操作符 “&” 用来返回操作数的地址。例如:

```
m=&count
```

该语句将变量 `count` 的内存地址赋给变量 `m`。这个地址是该变量在计算机内部的存储单元位置。它与 `count` 的值毫不相干。& 操作可以理解为 “取地址”。所以, 上面的赋值语句可以理解为 “`m` 取 `count` 的地址”。假定 `count` 的内存地址为 2000, 执行上述赋值语句后, 则 `m` 的值为 2000。

运算符 “\*” 是对 “&” 的一个补充, 它是返回位于这个地址内的变量的值的单目运算符。例如, 如果 `m` 中装有变量 `count` 的内存地址, 则 “`q = *m`” 表示将变量 `count` 的值赋给变量 `q`。若 `count` 的值为 100, 则 `q` 也为 100, 这是因为数值 100 存储在内存地址 2000 中, 而这一地址又存储在 `m` 中。\* 运算符可以理解为 “取地址中的值”。

对这两个操作符的理解是理解下文中指令描述方法的基础, 通过这两个操作符可以解决同一变量在不同表达式中的类型不一致, 以及变量定义的类型与引用时的类型不一致的问题。

#### 2. 普通算术指令的描述

下面给出两个普通算术指令的描述示例。



## 例 11.1

```
addl r1 = imm22, r3 (11-4)
```

```
r1 = r3 + (long long)imm22 (11-5)
```

addl 指令的语义是把立即数 imm22 和寄存器 r3 的内容相加, 结果放在寄存器 r1 中。根据指令 addl 的语义, r1 和 r3 的类型在 C 语言中假设定义为 long long 型, 立即数 imm22 的类型为 int 型。在式 (11-5) 式中我们把 imm22 先转换成 64 位整型, 然后相加。

## 例 11.2

```
zxt4 r1 = r3 (11-6)
```

```
tmp = (unsigned int)(*(unsigned long long*)&r3)
*(unsigned long long*)&r1 = (unsigned long long)(tmp) (11-7)
```

式 (11-6) 中 zxt4 指令的语义是把寄存器 r3 的低 32 位零扩展成 64 位, 把结果放在寄存器 r1 中。该指令用 C 语言描述如式 (11-7) 所示。r1 和 r3 是 long long 型的变量, 临时变量 tmp 定义为 unsigned int 型。式 (11-7) 的第一行, 取 r3 的低 32 位, 将其看成是无符号 int 型数。接下来把该值赋给临时变量 tmp, \*(unsigned int64\*)&r3 的目的是把变量 r3 存储的内容转换为 unsigned long long 型的整数; 式 (11-7) 的第二行是把结果 tmp 放在变量 r1 中, 左端由于在指令 zxt4 中 r1 的类型为 unsigned long long 型, 与 C 中 r1 的定义类型 long long 不一致, 所以要将变量 r1 转换为 unsigned long long 型, 方法是先对存放变量 r1 的地址转换成存放 unsigned long long 型的, 再获得 r1 的数据, 即 \*(unsigned int64\*)&r1; 式 (11-7) 的第二行赋值语句的右部, 是将 unsigned int 型的 tmp 零扩展为 64 位, 也就是将 tmp 转换为 unsigned long long 型。

## 3. 内存读写指令

IA64 指令系统是基于寄存器的, 即指令操作数全部是寄存器和立即数。在内存读写指令中, 其中一个寄存器的内容要么为地址, 要么是一个地址偏移。在源机器上, 这个地址为直接寻址。但当把指令转换为 C 语句时, 寄存器被转换成相应的存放地址的变量。实际上, 这个地址的内容才是我们读写的内容。这样从 C 语言的角度分析, 目标程序在目标机上运行时采用的是间接寻址的方式, 这给确定该操作数的类型也带来难度。

## 例 11.3 内存读指令。

```
ld8 r36 = [r14] (11-8)
```

```
r36 = *((long long*)(*(unsigned long long*)&r14)) (11-9)
```

式 (11-8) 中, 指令 ld8 的语义是从寄存器 r14 存放的地址中读取 8 字节的内容存放到寄存器 r36 中。式 (11-9) 中先将 r14 转换成存放 unsigned long long 类型的地址, 再进一步转换为存放 long long 型的地址变量, 最后把读取的内容存放在 r36 中。

## 例 11.4 内存写数据。

```
st8 [r37] = r36 (11-10)
```

```
*((long long*)(*(unsigned long long*)&r37)) = r36 (11-11)
```



指令 st8 的语义是将寄存器 r36 的内容, 放在寄存器 r37 内容表示的地址中。r36 的定义类型假设是 long long 型,  $(\text{long long}^*)(*(\text{unsigned long long}^*)&\text{r37})$  是把赋值语句左端的 r37 转换为存放 long long 型数据的地址,  $*(\text{long long}^*)(*(\text{unsigned long long}^*)&\text{r37})) = \text{r36}$  是表示把 r36 的内容存放在该地址处。

#### 4. 转移指令

转移指令可分为过程内转移和过程调用指令。对于一般的过程内转移指令, 操作数是过程代码段内的地址。在转换为 C 语言表示时, 用 goto 语句来描述它的语义, 指令后的地址将被一标号代替, 用来指定转移指令要跳转的位置 (注: 在此不讨论间接跳转指令), 如例 11-5 所示。而过程调用指令的操作数也是一个地址, 该地址在转换为 C 语言后将会被已恢复的过程名所代替。对每一个过程或函数, 必须恢复它的形参和实参。例 11-6 是一过程调用语句的恢复情况。

例 11.5 过程内直接跳转语句分析。

br.cond.dpnt.many 40000000000002c30 (11-12)

goto A12; (11-13)

例 11.6 过程调用语句恢复。

br.call.sptk.many b0=40000000000001140 (11-14)

bt\_\_Z8fillDeckP7bITAard(r39); (11-15)

以上是将 IA64 中指令转换成 C 语言时操作数的处理方法, 特别介绍了怎样应用 C 语言提供的强大类型转换来准确地描述指令的语义。对于语义很复杂的指令或者描述比较繁琐的指令, 可以用函数来实现它的语义。

### 11.1.3 基于过程的数据类型分析技术

ITA 系统采用的二进制翻译技术没有考虑指令与指令操作数之间的关系。换句话说, 翻译过程局限于单条指令, 没有考虑各指令操作数之间的联系。在过程中某一寄存器经编译优化后可能存放多个变量, 如果这些变量的类型不一致, 在 C 语言表示中就需要引入强制类型转换以保证语义的正确性。本节通过对过程 HRTL 表示上的数据流分析、变量重命名以及类型推导, 从过程的角度来进行分析, 以获得变量的类型信息。该技术方案首先根据数据流分析和集合划分理论, 对 HRTL 表示中的变量进行重命名; 然后利用类型推导规则确定每个新变量在定义或引用处的类型; 最后对每一个重命名后的变量利用格 (lattice) 理论推导出它在最终的 C 语言表示中的声明类型。

#### 1. 变量重命名技术

在现代体系结构的指令系统中, 大部分指令的操作数是寄存器。寄存器是非常珍贵的资源, 为了有效地利用寄存器, 编译器引入了寄存器分配策略, 该策略使得在一个过程中同一寄存器可能存放多个不相关的数据。但是这一做法却导致在 ITA 系统生成的过程代码



中,一个变量可能在不同的位置代表几个毫无关系的数据。特别是,当这些数据有不同的类型时,C代码中会出现过多的强制类型转换,或者在浮点与整型的转换中产生错误。为了解决这些问题,我们引入了变量重命名技术,该技术在过程数据流分析的基础上使用集合划分理论来实现。

### (1) 过程变量的 DU-chains 和 UD-chains

在中间语言 HRTL 级获得过程的 DU-chains 有以下几个难点:

1) 参数的恢复,包括实参和形参,以及返回值的正确恢复。

2) 在 IA64 指令系统中,引入了谓词寄存器,从而取代了部分 if-else 语句。但该做法却导致在生成过程的 CFG 时,把条件语句看成是简单的赋值语句。这就使得在查找带有谓词的赋值语句时要考虑前面的谓词,换句话说,就是要将赋值语句看成是条件语句。

3) 对带有软件流水特征的代码的处理。在 ITA 系统中,采用模拟寄存器旋转的方法消除软件流水,本来是依赖硬件实现的寄存器自动旋转,我们在 HRTL 级利用简单的赋值语句模拟实现。在进行数据流分析时,必须考虑这些添加的表达式。

### (2) 消除寄存器重用

假设同一寄存器变量存在两个或两个以上的定义 (definition), 如果它们的引用 (use) 链中至少有一个相同的引用, 就假设这些定义所赋的值的类型相同。也就是说, 在消除寄存器重用后, 这些定义共享一个变量名。我们采用集合划分的概念来说明该问题。集合  $R$  的一个划分  $S$  是指根据一定的原则将  $R$  分成一些两两不相交的子集。由  $R$  的这些非空子集构成的集族  $S$  称为集合  $R$  的一个划分 (partition)。也就是说, 如果  $S$  是  $R$  的一个划分, 则  $S$  的元素是两两不相交的, 且  $\bigcup S = R$ 。

消除寄存器重用现象就是对一确定的寄存器变量的所有定义构成的集合求其划分, 该划分中同一元素中的所有定义有相同的变量名, 其形式定义如下。

假设  $R$  是过程中对应一个寄存器变量所有定义的集合。 $S$  是  $R$  的一个划分。 $Use(d)$  为定义  $d$  的所有引用组成的集合,  $S_1, S_2, \dots, S_n$  为  $S$  的元素。满足:

- $R = S_1 \cup S_2 \cup \dots \cup S_n$ 。
- $S = \{S_1, S_2, \dots, S_n\}$ 。
- $S_1, S_2, \dots, S_n$  两两不相交, 也就是,  $\forall S_i \in S, \forall S_j \in S$ , 若  $S_i \neq S_j$ , 则  $S_i \cap S_j = \emptyset$ 。
- $\forall S_i \in S, \forall S_j \in S, S_i \neq S_j, \forall d_m \in S_i, \forall d_n \in S_j$ , 存在:  $Use(d_m) \cap Use(d_n) = \emptyset$ 。

由此将得到集合  $S_1, S_2, \dots, S_n$  的同一变量的不同定义, 每一集合的定义用相同的变量名表示, 而不同的集合 (如  $S_1$  和  $S_2$ ) 用不同的变量表示。

### (3) 实例

图 11-1 是一个过程的 CFG, 假设对变量  $X$  重命名。通过数据流分析得到变量  $X$  的 DU-chains 如下:

$$d_1: \{(X < B2, 1>), <B4, 1>, <B5, 1>\}$$



$$d_2: \{(X \langle B3, 1 \rangle), \langle B5, 1 \rangle\}$$

$$d_3: \{(X \langle B5, 2 \rangle), \langle B6, 1 \rangle\}$$

$d_1$  是变量  $X$  的一条 DU-chain, 在基本块  $B2$  中对变量  $X$  的定义存在两个对它的引用, 分别是基本块  $B4$  和  $B5$  中第一条语句对  $X$  的引用。对  $d_2$  的解释是在  $B3$  中第 1 条语句变量  $X$  的定义在  $B5$  的第 1 条语句中存在引用。同理  $d_3$  是表示在  $B5$  中第 2 条语句对  $X$  的定义在  $B6$  的第 1 条语句中存在对它的引用。那么集合  $Use(d_1)$  包含元素  $\langle B2, 1 \rangle$  和  $\langle B5, 1 \rangle$ 。

我们利用集合划分的理论对变量  $X$  进行重命名, 因为:

- $R = \{d_1, d_2, d_3\}$ 。
- 因为  $Use(d_1) \cap Use(d_2) = \langle B5, 1 \rangle$ ,  $S_1 = \{d_1, d_2\}$ 。
- $S_2 = \{d_3\}$ 。
- $S = \{S_1, S_2\} = \{\{d_1, d_2\}, \{d_3\}\}$ 。

所以将  $d_1$  和  $d_2$  中变量  $X$  的引用和定义用新的变量名  $X0$  代替,  $d_3$  中的引用和定义用  $X1$  代替。同理对  $Y$  和  $Z$  进行重命名, 得到图 11-2。

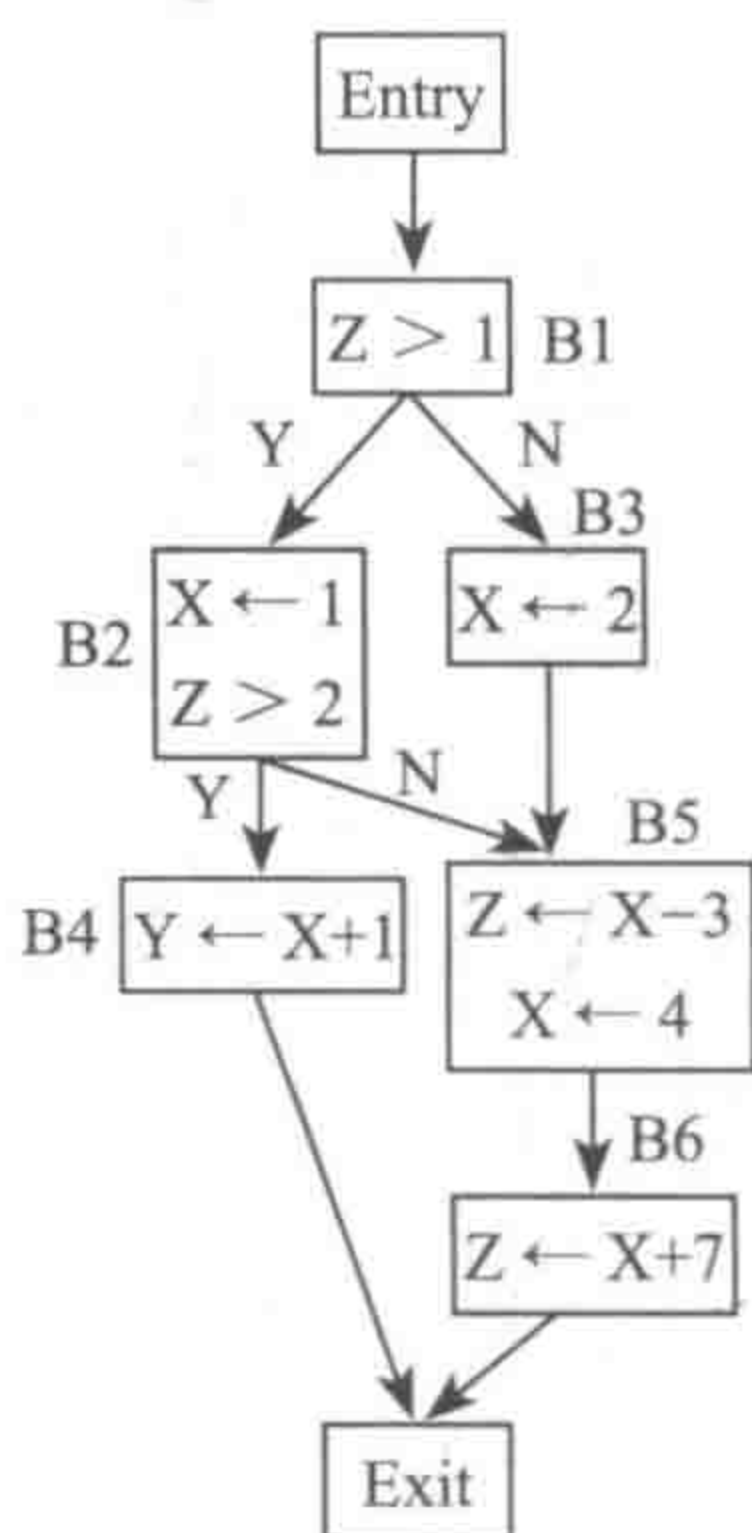


图 11-1 过程的 CFG

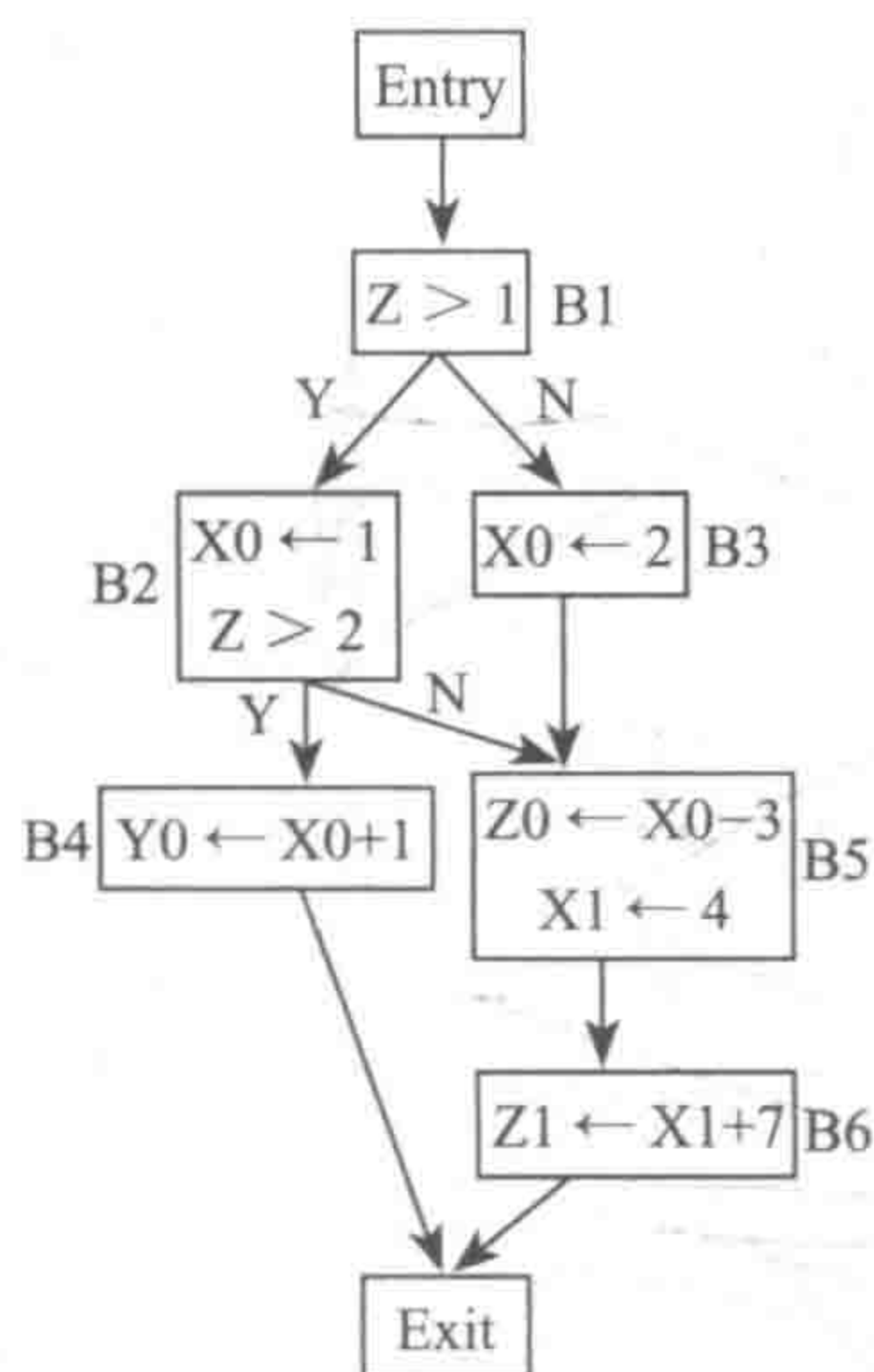


图 11-2 对图 11-1 变量重命名后过程的 CFG

## 2. 变量类型推导的规则

当利用 DU-chains、UD-chains 和集合的划分概念对中间语言 RTL 表示的过程消除变量重用后, 再利用数据类型推导规则对每个变量的每个位置的具体类型进行推导。下面是几条具体的规则。

**规则 1:** 对于  $r = \text{exp}$  赋值语句, 如果  $\text{type}(\text{exp}) = A$ , 那么  $\text{type}(r) = A$ 。

例:  $r32 = r33 + 100$ , 若  $r33 + 100$  的类型为 unsigned int64, 根据规则 1,  $r32$  的类型为 unsigned int64。

**规则 2:** 对于表示 store 的语句  $m[r_i] = r_j$ , 该语句为把  $r_j$  的值存到  $r_i$  表示的地址位置,



如果  $\text{type}(r_j) = A$ , 那么  $\text{type}(r_i) = A^*$ 。

例:  $r32 = r33 + 100$ ,  $m[r32] = r34$ 。假设  $\text{type}(r34) = \text{int}$ , 根据规则 1,  $\text{type}(r32) = \text{unsigned int64}$ , 11.1.2 节说明任何不知道具体类型的地址变量的类型默认为  $\text{unsigned int64}$ , 所以由规则 2 可得  $\text{type}(r32) = \text{int}^*$ 。

规则 3: 对于  $\text{ControlTransfer } r$ , 该语句是将控制流转移到程序代码段的某一位置, 所以  $\text{type}(r) = \text{pi}$  (指令的地址)。在 RTL 表示中, 把该语句转换为一标号, 供跳转语句寻址。

规则 4:  $r = \text{call } X$ , 如果函数的返回值类型为  $A$ , 那么  $\text{type}(r) = A$ 。

根据 IA 64 指令系统的特点, 以及二进制翻译采用的技术, 又存在以下的推导规则:

规则 5: 在 IA64 指令系统中,  $r1$  寄存器是指向全局数据段的指针。ITA 二进制翻译系统中, 用一个数组保存数据段的内容,  $r1$  指向了数组的首地址, 有  $\text{type}(r1) = \text{unsigned int64}$ 。那么, 如果  $r32 = r1$  或者  $r32 = r1 + 100$ , 则  $\text{type}(r32) = \text{unsigned int64}$ 。

规则 6: ITA 系统使用一个数组把原二进制代码中栈的内容映射到目标机上。 $\%afp$  和  $\%vfp$  分别表示栈指针和帧指针,  $\text{type}(\%afp) = \text{unsigned int64}$ ,  $\text{type}(\%vfp) = \text{unsigned int64}$ 。这样产生以下规则。

如果有  $r[38] := \%afp + 40$ , 则  $\text{type}(r38) = \text{unsigned int64}$ 。

规则 7: 语句  $r55 = m[r54]$  的语义是从内存中读数据, 则  $\text{type}(r54) = \text{unsigned int64}$ 。

如果有下列语句序列:

- 1)  $r52 = \%afp + 48$
- 2)  $r53 = m[r52]$
- 3)  $r54 = r53 + 8$
- 4)  $r55 = m[r54]$

在语句 2 的位置不能确定  $r53$  的具体类型, 但根据语句 4 可以确定  $\text{type}(r54) = \text{unsigned int64}$ , 而  $r54 = r53 + 8$ , 因此有  $\text{type}(r53) = \text{unsigned int64}$ 。

### 3. 格理论在变量类型推导中的应用

在编译和反编译中涉及的数据类型形成一个层次关系。比如, 对于 64 位的数据实体, 它的类型可能是  $\text{double}$ , 也可能是  $\text{long long}$ 。可以认为类型  $\text{size64}$  有两个子类型, 即  $\text{double}$  和  $\text{long long}$ 。这些类型之间的关系可以用一个格来表示。利用前面变量的推导规则对过程中的每个新变量进行类型推导, 得到该变量类型的一个集合。下面先对格的基本概念加以阐释, 然后对变量的类型集合利用格理论确定它的最终类型。

#### (1) 格的概念及相关性质

图 11-3 是一个简单的格, 表示为  $\langle X, R, \sqcup, \sqcap \rangle$ , 格上的所有结点构成集合  $X$ , 即  $X = \{a, b, c, d\}$ 。结点之间的边表示结点间的关系  $R$ ,  $R$  是集合  $X$  上的偏序关系。如图 11-3 所示, 结点  $a$  和结点  $b$  之间有连线且  $a$  在  $b$  的上方, 记为  $(a, b) \in R$ , 或  $a R b$ 。 $\sqcup$ 、 $\sqcap$  是定义在集

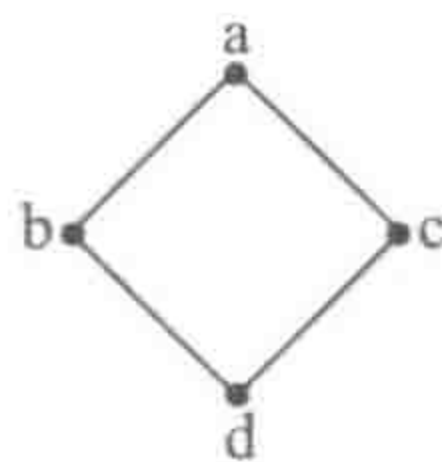


图 11-3 一个简单的格



合  $X$  上的两个运算符, 且满足:

- |  |     |
|--|-----|
| 1) $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ | 结合律 |
| 2) $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$ |     |
| 3) $a \sqcup b = b \sqcup a$                       | 交换律 |
| 4) $a \sqcap b = b \sqcap a$                       |     |
| 5) $a \sqcup a = a$                                | 幂等律 |
| 6) $a \sqcap a = a$                                |     |
| 7) $a \sqcap (a \sqcup b) = a$                     | 吸收律 |
| 8) $a \sqcup (a \sqcap b) = a$                     |     |

格有如下性质:

- 1) 格是一偏序集, 也就是说  $\langle X, R \rangle$  是一偏序集,  $R$  是偏序关系。
  - 若  $a \in X$ , 那么  $(a, a) \in R$ 。
  - 若  $a, b \in X, a \neq b, (a, b) \in R$ , 那么  $(b, a) \notin R$ 。
  - 若  $a, b, c \in X, (a, b) \in R, (b, c) \in R$ , 那么  $(a, c) \in R$ 。
- 2)  $\forall a, b \in X, \exists c \in X, c R a$  且  $c R b$ 。
- 3)  $\forall a, b \in X, \exists d \in X, a R d$  且  $b R d$ 。
- 4)  $\forall a, b \in X, a R b \Leftrightarrow a \sqcup b = a$ 。
- 5)  $\forall a, b \in X, a R b \Leftrightarrow a \sqcap b = b$ 。

为了进一步将格的概念应用到数据类型推导的过程, 阐述如下几个概念:

**定义 11.1** 如果  $a, b \in X, (a, b) \in R$  或  $(b, a) \in R$ , 就说  $a$  和  $b$  是可比的 (comparable); 如果  $a, b \in X, (a, b) \notin R$  且  $(b, a) \notin R$ , 则称  $a$  和  $b$  是不可比的 (incomparable)。

**定义 11.2** 假设  $X_1 \subseteq X, \forall a, b \in X_1, a, b$  是可比的, 称  $R$  为集合  $X_1$  上的全序。

**定义 11.3** 假设  $X_1 \subseteq X, \exists a, b \in X_1, a, b$  是不可比的, 集合  $X_1$  中的元素存在冲突。

**定义 11.4** 格  $\langle X, R, \sqcup, \sqcap \rangle$  的链 (chain)  $C$  是  $X$  的一个子集, 存在关系:

- 1)  $C \subseteq X$
- 2)  $\forall a, b \in C$ , 那么  $a R b$  或  $b R a$ 。

**定义 11.5**  $S \subseteq X$  是  $X$  的子格, 当且仅当  $\forall x, y \in S: (x \sqcup y \in S)$  且  $(x \sqcap y \in S)$ 。

## (2) 数据类型推导中格的定义及说明

大小为 64 位的数据可能表示的是数据的地址 (pointer), 也可能为 double 或 long long 型。也就是说 size64 的子结点有 pointer (指针)、long long、double 等。例如, 假设有一寄存器 ( $r_x$ ) 被三条语句调用: 64 位的 move 指令、位逻辑指令 and 以及逻辑左移指令。在 move 指令前, 不能确定  $r_x$  的类型, move 之后能判断  $r_x$  是 64 位数据; 在 and 指令后, 可以确定  $r_x$  是整型; 而在逻辑左移指令后可以确定  $r_x$  为 unsigned int64 型的数据。这些类型之间的关系可以用图 11-4 的格来表示。



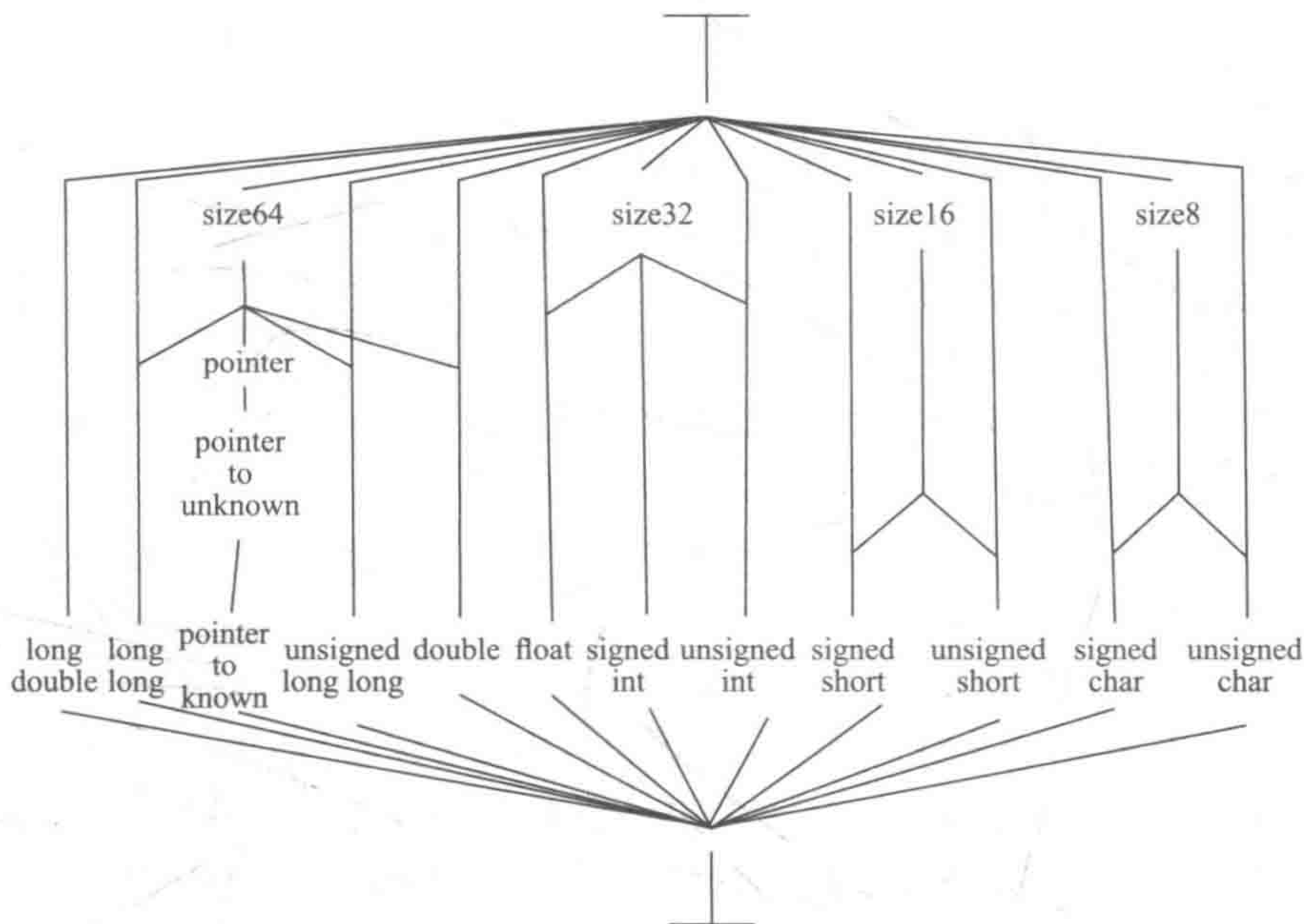


图 11-4 二进制翻译中数据类型的格表示

将图 11-4 所表示的格  $L$  表示为  $\langle T, R, \sqcup, \sqcap \rangle$ ,  $T$  是图中所有数据类型的集合, 包括“ $T$ ”和“ $\perp$ ”。格的最高上界“ $T$ ”表示没有类型的信息, 类型推导就是一个将数据的类型自顶向下尽量定位到靠近格底端“ $\perp$ ”的过程。如果数据的类型定位到底端的“ $\perp$ ”表示推导出的数据类型发生冲突。其他的定义同图 11-4。推导的类型发生冲突是常见的现象, 比如, 对于以下 RTL 语句:

$r32 = f12 / f23;$  (11-16)

$r30 = r32;$  (11-17)

由语句 (11-16) 可推导,  $r32$  为浮点型 (如  $\text{float64}$ ), 而由语句 (11-17) 又可以推导出  $r32$  为整型  $\text{int64}$  (假设  $\text{type}(r30) = \text{int64}$ ), 这就产生了矛盾。我们可以通过强制类型转换来解决这个矛盾。

在推导变量的类型时, 将得到变量类型的一个集合, 也就是说类型并不唯一。这个集合越小则越有价值, 而且当  $R$  在这个集合上是全序关系时最有价值。但在很多情况下这个集合中的类型彼此存在冲突, 如一个变量被推导出既是  $\text{int64}$ , 又是  $\text{float64}$ 。对于这些情况的处理, 在下节中将会详细叙述。

### (3) 基于格的变量类型分析

对于在 HRTL 表示中的一个变量, 根据它的 DU-chains 推导出的类型可能不止一个, 这些类型构成类型全集的一个子集, 该子集的元素可能存在关系  $R$ , 也可能相互冲突。我们只能取其中一个作为低级  $C$  中对变量的声明, 因此必须有对类型的取舍原则。

假设推导出的类型构成集合  $M$ , 满足  $T \notin M$ ,  $\perp \notin M$ ,  $M \subseteq T$ 。把  $M$  的情况归结为以下



三种:

- (1)  $\langle M, R, \sqcup, \sqcap \rangle$  是  $L$  的子格。
- (2)  $\langle M \cup \{\perp\}, R, \sqcup, \sqcap \rangle$  构成  $L$  的子格。
- (3)  $\langle M \cup \{\top\} \cup \{\perp\}, R, \sqcup, \sqcap \rangle$  构成  $L$  的子格。

分别对应图 11-5 中三个示例。

1)  $\langle M, R, \sqcup, \sqcap \rangle$  是  $L$  的子格。

这是类型推导最希望得到的结果, 也是在推导过程中经常出现的情况。这时将  $\sqcap M$  作为  $C$  表示中变量的声明类型, 也就是把最大下界 (the great low bound) 作为变量的定义。如下面两条 RTL 语句:

- ① `r14 = %vfp`
- ② `m[r14] = truncs(64,32,r32)`

由①推出  $\text{type}(r14)=\text{pointer}$ ; 由②推出  $\text{type}(r14)=\text{int}^*$ ; 集合  $\{\text{pointer}, \text{int}^*\}$  中  $\text{pointer}$  是  $\text{int}^*$  的父结点, 即  $(\text{pointer}, \text{int}^*) \in R$ , 所以在低级  $C$  中把 `r14` 声明成  $\text{int}^*$  数据, 生成的  $C$  语句如下:

```
int32* r14;
r14=_virtuals;
*r14=(int32)(r32);
```

2)  $\langle M \cup \{\perp\} \cup \{\top\}, R, \sqcup, \sqcap \rangle$  构成  $L$  的子格。

这时将  $\sqcup M$  作为低级  $C$  中变量的定义类型, 也就是把最小上界 (the least upper bound) 作为变量的类型。如下例:

- ① `r16=virtual+32`
- ⋮
- ② `r28=m[r16]`
- ⋮
- ③ `f10=m[r16]`
- ⋮

由①可以得到  $\text{type}(r16)=\text{pointer to unknown}$ , 由②得到  $\text{type}(r16)=\text{long long}^*$ , 由③得到  $\text{type}(r16)=\text{double}^*$ 。所以集合  $M$  为  $\{\text{pointer to unknown}, \text{double}^*, \text{long long}^*\}$ 。

在图 11-4 中没有  $\text{double}^*$  和  $\text{long long}^*$ , 它们被结点  $\text{pointer to known}$  代替。所以有关系  $\text{double}^* \sqcup \text{long long}^* = \text{pointer to unknown}$ 。所以 `r16` 最终的类型为  $\text{type}(r16)=\text{pointer to unknown}$ 。在  $C$  代码中 “ $\text{pointer to unknown}$ ” 可以被类型 “ $\text{unsigned long long}$ ” 代替。最终的  $C$  代码表示如下:

```
long long virtual[];
```

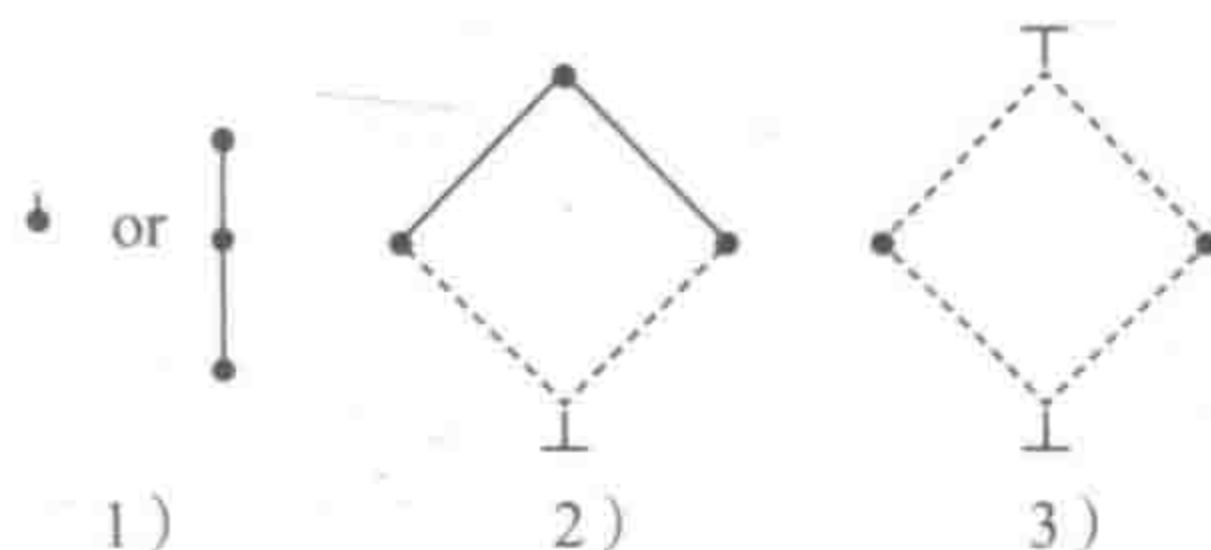


图 11-5 变量类型分析中的三个子格示意图



```

unsigned long long r16;
long long r28;
double f10;
:
r16=virtual + 32;
:
r28=((long long*)r16);
:
f10=((double*)r16);
:

```

3)  $\langle M \cup \{\perp\} \cup \{\top\}, R, \sqcup, \sqcap \rangle$  构成  $L$  的子格。

对于同一定义，在不同的引用中可能得出的类型存在冲突，也就是说，推导出的类型不止一个，它们之间的关系如图 11-5 中的示例 3 所示。这时可以把其中的任何一个作为低级 C 中变量的类型。但是为了减少赋值语句右端的强制类型转换，一般会将从变量的定义赋值语句推导出的类型作为变量的最终类型。用下面的例子说明这类问题的解决方法。

①  $r2 := f4 / f6$

:

②  $f5 := r2$

:

③  $r2007 := r2$

$r2$ 、 $r2007$  为整型寄存器， $f4$ 、 $f5$ 、 $f6$  为浮点寄存器，“/f”表示浮点除。由①推导出  $\text{type}(r2)=\text{long double}$ ，而③推导出  $\text{type}(r2)=\text{int64}$ ；集合  $\{\text{long double}, \text{int64}\}$  中两个类型是冲突的。我们处理的方法是把从①式中推导出的类型作为变量的类型，生成的低级 C 如下所示：

```

long double f4, f5, f6, r2;
int64 r2007;
r2=f4/f6;
:
f5=r2;
:
r2007= *(int64*)&r2;

```

## 11.2 函数类型恢复

函数之所以称为函数，是因为它与一般的语句片段有着本质的区别，函数除了拥有函数体外还必须具有如下信息：函数名、参数和相应的函数类型。前面三章分别阐述了库函数识别技术、用户函数与库函数同名的区分技术以及内嵌数学库函数恢复技术，却无与函数类型恢复相关的论述。本节将介绍反编译中的函数类型恢复。



### 11.2.1 问题引入

函数类型的恢复依赖于特定体系结构在函数返回值方面的约定,按照给定的规则进行恢复。传统的函数类型恢复方法是从调用者角度基于函数返回值的恢复而实施的,当对源程序编译所使用的优化级别比较高时这一恢复方法将失效。所以接下来尝试从调用者和被调用者相结合的角度进行函数类型恢复,反过来再根据恢复出的函数类型进行返回值恢复。

在介绍 ITA 反编译系统中基于控制流和数据流分析的函数类型恢复技术并给出相应的恢复算法之前,有必要明确一下此系统的工作机理:在该系统中,前端提供了二进制文件解码器,从输入的 ELF64 可执行文件中提取指令流;然后使用解码器将该指令流解码成源机器指令序列;随后语义映射器基于源机器的语义说明将每条 IA64 指令翻译成第一级中间表示 RTL;RTL 到 HRTL 通用分析器应用控制转换信息、过程约定以及其他的信息对前面所产生的 RTL 语句进行分析,去除同机器特性相关的部分,得到第二级中间表示 HRTL;最后使用 C 代码生成器将 HRTL 语句转换为 C 程序。

本节所论述的函数类型恢复技术是 RTL 到 HRTL 转换过程中的重要内容,对与给定函数相关的 RTL 语句序列进行控制流分析并把其划分为一系列的基本块,随后依靠对基本块进行数据流分析所得的结果最终确定此函数的函数类型。

### 11.2.2 函数类型的恢复

汇编代码级的函数返回值并不像高级语言(如 C 语言中 `return`)那样显式返回某个类型的值,而是被调用函数根据约定在返回之前将欲返回的值放到指定位置,之后调用函数再到指定位置取该返回值。返回位置根据返回值类型而不同,常为具体的寄存器或栈位置。IA64 约定整型返回值用通用寄存器 `r8` 存放,浮点返回值用寄存器 `f8` 存放;如果需要返回的是一个结构体,则有以下两种情况:①当结构体的大小超过 32 字节时则使用寄存器 `r8`,`r8` 中存放的是指向该结构体的指针;②若结构体的大小小于或等于 32 字节时,则根据需要使用 `r8 ~ r11` 这四个寄存器中的多个来存放返回值。根据存放函数返回值使用的是通用寄存器还是浮点寄存器即可确定函数的返回类型,通用寄存器对应整型,浮点寄存器对应浮点型。(注:为了更清晰地阐述,本节不考虑 `r8` 和 `f8` 以外的情况。)

函数类型恢复可分为两种情况来考虑:①库函数函数类型的恢复;②用户函数函数类型的恢复。

对库函数函数类型的恢复比较简单,只要能识别出库函数的函数名,再根据库函数发布时所声明的函数名与函数类型之间的对应关系即可确定库函数的类型。

对用户函数函数类型的恢复可以从两个不同的角度进行分析:①从调用者(caller)的角度对被调函数进行函数类型恢复;②从被调用者(callee)的角度对被调函数进行函数类型恢复。无论从 caller 或 callee 哪个角度来对函数类型进行恢复,都离不开对基本块的分析。



## 1. 基本块的划分

对于一个给定的程序，可以把它划分为一系列的基本块。划分算法如下：

1) 求出程序中各个基本块的入口语句，它们是：

- ① 程序的第一条语句；
- ② 能由条件转移语句或无条件转移语句转移到的语句；
- ③ 紧跟在条件转移语句后面的语句。

2) 对以上求出的每一入口语句，构造其所属的基本块。它由该入口语句到另一个入口语句（不包含此语句），或到一转移语句（包含此语句），或到一停止语句（包含此语句）之间的语句序列组成。

3) 凡未被纳入某一基本块的语句都是程序中控制流程无法到达的语句，从而也是不会被执行的语句，可以把它们从程序中删除。

给定程序按此方法划分得出一个基本块集合，可以通过构造称为控制流图的有向图把控制信息加入基本块集合中。流图的节点是基本块，首节点的入口语句是程序的第一条语句。如果在某个执行序列中 B2 紧跟在 B1 之后，则从 B1 到 B2 有一条有向边，即如果：

1) 从 B1 的最后一条语句有条件或无条件转移到 B2 的第一条语句；或者

2) 按程序的次序，B2 紧跟在 B1 之后，并且 B1 不是结束于无条件转移，则说 B1 是 B2 的前驱，而 B2 是 B1 的后继。由此可以得到给定程序的控制流图 (CFG)。

## 2. 基本块的数据流分析

对基本块进行数据流分析通常要涉及如下变量集合：LiveIn、LiveOut、UseunDef、Def 等。

令 B 为一个基本块，定义 LiveIn(B) 为在基本块 B 入口处为活跃变量的集合；LiveOut(B) 为在基本块 B 出口处为活跃变量的集合。LiveIn 和 LiveOut 并不是相互独立的，令 S(B) 为控制流图中基本块 B 的后继的集合，V(B) 为控制流图中基本块 B 的前驱的集合，则有

$$\text{LiveOut}(B) = \bigcup_{i \in S(B)} \text{LiveOut}(i)$$

$$\text{LiveIn}(B) = \bigcup_{j \in V(B)} \text{LiveOut}(j)$$

即，一个变量在基本块的出口处是活跃的，仅当它在本基本块的某个后继的入口处为活跃的，如果基本块没有后继，则其 LiveOut 为空；一个变量在基本块的入口处是活跃的，仅当它在本基本块的某个前驱的出口处为活跃的，如果基本块没有前驱，则其 LiveIn 为空 (main() 函数除外)。

令 UseunDef(B) 为 B 中被定值之前要引用的变量的集合。UseunDef(B) 是一个集合常量，这个集合由基本块 B 中的语句唯一确定。如果  $v \in \text{UseunDef}(B)$ ，则  $v \in \text{LiveIn}(B)$ ，即

$$\text{UseunDef}(B) \subseteq \text{LiveIn}(B)$$

令 Def(B) 为 B 中被定值的变量的集合。Def(B) 也是一个集合常量，它由基本块 B 中的语句唯一确定。如果一个变量 v 在基本块 B 的出口处活跃且 v 不属于 Def(B)，则它在 B 的入口处也是活跃的，即

$$\text{LiveIn}(B) \supseteq \text{LiveOut}(B) - \text{Def}(B)$$



于是,有

$$\text{LiveIn}(B) = \text{UseunDef}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

虽对基本块进行数据流分析时要用到变量集合 LiveIn、LiveOut、UseunDef 和 Def,但对本节要解决的函数类型恢复问题来说,变量集合 LiveIn、LiveOut 可以略去不加考虑,仅仅考虑变量集合 UseunDef 和 Def 对问题解决的影响,且基本块的变量集合 UseunDef 和 Def 是由基本块自身唯一确定的。

### 3. 传统函数类型恢复法

令  $\text{recursiveUseunDef}(B)$  为基本块  $B$  的后继递归基本块(后继递归止于 call 块和 compcall 块)中对返回位置变量使用先于定值的变量集合。可以基于控制流图对相应块的 UseunDef 集进行“ $\cup$ ”操作来求得,它由块自身和块间关系唯一确定。

确定一函数的类型,依赖于此函数是否有返回值。这就要对 IA64 规定的返回值的位置变量进行分析,当一函数调用块  $B$  进行函数调用返回后,如果  $\text{recursiveUseunDef}(B)$  为空,则认为此函数无返回值(而不管源代码中函数是否有返回值)。对于这种分析方法,直到所有对该函数的调用都被分析后才能确定函数是否有返回值。最后根据函数有无返回值和函数返回位置变量的类型恢复函数的类型。函数类型恢复算法如图 11-6 所示。

<p>功能: 判定被调函数的函数类型</p> <p>输入: 调用块 CallBB, 被调函数 Callee</p> <p>输出: 被调函数 Callee 的函数类型</p> <p>算法步骤:</p> <ol style="list-style-type: none"> <li>(1) 定义一个集合变量 useundef</li> <li>(2) 执行操作 <math>\text{useundef} = \text{recursiveUseunDef}(\text{CallBB})</math></li> <li>(3) 如果 useundef 集合为空,则设置函数 Callee 的函数类型为 void,退出</li> <li>(4) 根据集合 useundef 中排在第一位的位置变量信息 <math>Q</math> 设置被调函数 Callee 的函数类型,如 <math>Q</math> 是 r8 的表示形式,则设置函数 Callee 的函数类型为 int;如 <math>Q</math> 是 f8 的表示形式,则设置函数 Callee 的函数类型为 float,退出</li> </ol>
---

图 11-6 传统函数类型恢复算法

根据以上给出的函数类型恢复算法进行实例验证,测试用例为 returncallee.c,其源程序及翻译后所对应的 RTL 分块表示如图 11-7、图 11-8 所示。

经验证,对 returncallee.c 编译采用 GCC 0 级优化时函数 add4 和 add2 的函数类型都可正确恢复为 int 型,因为在函数 add4 和 add2 调用块的后继块中都有对 r8 的引用先于定值的情况存在,即调用块的 recursiveUseunDef 变量集合不为空,且处于第一位的位置变量为 r8。

但是,当对程序进行编译时选用高优化级别的情况就不同了。如对 returncallee.c 编译时采用 GCC 1 级优化,则翻译后所对应的 RTL 分块表示如图 11-9 和图 11-10 所示,对此如仍按照上述算法进行函数类型的恢复,则只能正确恢复出函数 add4 的 int 型函数类型,函数 add2 的类型却被识别成 void 型。



```

/*procedure returncallee.c*/
int add2(int a,int b)
{
    return a+b;
}
int add4(int a ,int b int c,int d)
{
    return add2(a+b,c+d);
}
int main()
{
    printf("Fifty five is %d\n",add4(10,12,19));
    return 0;
}

```

图 11-7 测试例子 returncallee

```

/*add 函数相关代码段 */
40000000000000882 r[8]<64i>:= CALL bt_add4(r[36]<64i>,
                                         r[37]<64i>,r[38]<64i>,r[39]<64i>)
Call BB(0x83d73b8):
40000000000000890:0    *64* r[1]:= r[32]
40000000000000891:1    *64* r[14]:= r[8]
40000000000000892:2    *64* r[15]:= r[1]+72
400000000000008a0:0    *64* r[36]:= m[r[15]]
400000000000008a1:1    *64* r[37]:= r[14]
400000000000008a2:2    *64* r[32]:= r[1]
...
400000000000008b2:2 CALL printf(r[36]<64i>,r[37]<64i>)

/* add2 函数相关代码段 */
40000000000000812 r[8]<64i>:= CALL bt_add2(r[40]<64i>,
                                         r[41]<64i>)
Ret BB (0x8d965c0):
40000000000000820:0    *64* r[1]:= r[36]
40000000000000821:1    *64* r[14]:= r[8]
40000000000000822:2    *64* r[8]:= r[14]
...
40000000000000842 RET

```

图 11-8 基于 caller 分析法 (GCC 0 级优化)

#### 4. 改进后的函数类型恢复法

基于上述函数类型恢复算法，对 GCC 0 级优化的程序都能正确恢复函数的类型，然而对高优化级别程序的恢复效果却不太令人满意。究其原因是因为此算法是仅从 caller 的角度进行分析，以函数返回值恢复方法为基础而实施，函数返回值的不正确恢复导致函数类型也不能被正确恢复。



```

/*add 函数相关代码段*/
400000000000007b2 r[8]<64i>:= CALL bt_add4(r[36]<64i>,
                                         r[36]<64i>,r[37]<64i>,r[38]<64i>)
Call BB(0x83d73b8):
400000000000007c0:0      *64* r[1]:= r[32]
400000000000007c1:1      *64* r[36]:= r[8]
400000000000007c2:2      *64* r[35]:= r[1]+72
400000000000007d0:0      *64* r[35]:= m[r[15]]
400000000000007d1:1
400000000000007d2 CALL printf(r[35]<64i>,r[36]<64i>)

/* add2 函数相关代码段*/
40000000000000752 CALL bt_add2(r[39]<64i>,r[40]<64i>)

Ret BB (0x83d93e8):
40000000000000760:0      *64* r[1]:= r[36]
...
40000000000000772 RET

```

图 11-9 基于 caller 分析法 (GCC 1 级优化)

```

/*add 函数相关代码段*/
CALL BB (0x83d92c0):
...
40000000000000732:2      *64* r[39]:= r[32]+r[33]
40000000000000740:0      *64* r[40]:= +r[34]++r[35]
40000000000000741:1      *64* r[36]:= r[1]
...
40000000000000752 CALL bt_add2(r[39]<64i>,r[40]<64i>)
Ret BB (0x83d93e8):
40000000000000760:0      *64* r[1]:= r[36]
...
40000000000000772 RET

/* add2 函数相关代码段*/
40000000000000720:0      *64* r[8]:= r[32]+r[33]
40000000000000721:1
40000000000000722 RET

```

图 11-10 基于 callee 分析法 (GCC 1 级优化)

一个函数是否具有返回值，以及它的类型应该由其自身决定，即应从 callee 的角度来分析：令  $\text{recursiveDef}(B)$  为基本块  $B$  及基本块  $B$  的前驱递归基本块（前驱递归止于 call 块和 compcall 块的后继）中对返回位置变量定值的变量集合，其也可通过对控制流图和相应块的 Def 集执行  $\cup$  操作求得，它由块自身和块间关系唯一确定。依据此函数返回块集合对其中每一个返回块  $BB$  的  $\text{recursiveDef}(BB)$  集进行累积并操作，如果累积变量集合不空则根据其中处于第一位的位置变量设置相应的函数类型，否则不做任何操作。但是，随着优化级别的提高，单一从 callee 角度来分析并恢复函数的类型同仅仅从 caller 角度来分析具有



同样的不足之处，它只能正确恢复出函数 add2 的函数类型（如图 11-10 所示）。

综合考虑基于 caller 和 callee 两种分析方法的优势与不足，把两种分析方法糅合在一起，各取所长、互补其短，改进算法如图 11-11 所示。

功能：判定被调函数的函数类型

输入：调用块 CallBB，被调函数 Callee

输出：被调函数 Callee 的函数类型

算法步骤：

- (1) 定义一布尔变量 isSet = FALSE，定义一个集合变量 def 并初始化为空
- (2) 定义一个基本块集 B，使 B 为被调函数 Callee 的返回块集合
- (3) 对 B 中的每一个基本块  $B_i$  执行操作
 
$$\text{def} = \cup \text{recursiveDef}(B_i)$$
- (4) 如果 def 集不空，根据集合 def 中排在第一位的位置变量信息 Q 设置被调函数 Callee 的函数类型（如 Q 是 r8 的表示形式，则设置函数 Callee 的函数类型为 int；如 Q 是 f8 的表示形式则设置函数 Callee 的类型为 float），并置 isSet = TRUE
- (5) 如果 isSet == TRUE，转 (10)
- (6) 定义一个集合变量 useundef
- (7) 行操作 useundef = recursiveUseunDef(CallBB)
- (8) 如果 useundef 集为空，则设置函数 Callee 的函数类型为 void，转 (10)
- (9) 根据集合 useundef 中排在第一位的位置变量信息 Q 设置被调函数 Callee 的函数类型，如 Q 是 r8 的表示形式，则设置函数 Callee 的函数类型为 int；如 Q 是 f8 的表示形式则设置函数 Callee 的类型为 float
- (10) 退出

图 11-11 改进后的函数类型恢复算法

应用改进的函数类型恢复算法对例子 returncallee.c 的 GCC 0 ~ 2 级优化程序进行函数类型恢复测试，结果显示函数 add4 和 add2 的函数类型都得以正确恢复。可知上述函数类型恢复算法是行之有效的。

除此之外，我们还把改进的“函数类型恢复算法”应用于 ITA 反编译系统，并在一些更具代表性的测试集上进行了测试，如 NAS 测试集中的 8 个测试用例、Fortran78 Test Suite 测试集中的 192 个测试用例，以及 SPEC2000 测试集中 gzip、bzip、mcf、parser、crafty、gap、vortex、twolf、art、equake、ammp 等，翻译前后的运行结果完全相同。这些程序，尤其是 SPEC2000 中的 11 个测试程序规模较大，其中含有大量的函数调用，且函数类型各异。对它们应用所述的函数类型恢复算法进行处理，得到了正确的运行结果，这就验证了上述改进型函数类型恢复算法的正确性和可行性。

### 11.3 本章小结

对于本章的总结，我们从两个方面分别进行，一个是基本数据类型的恢复，另一个是函数类型的恢复。

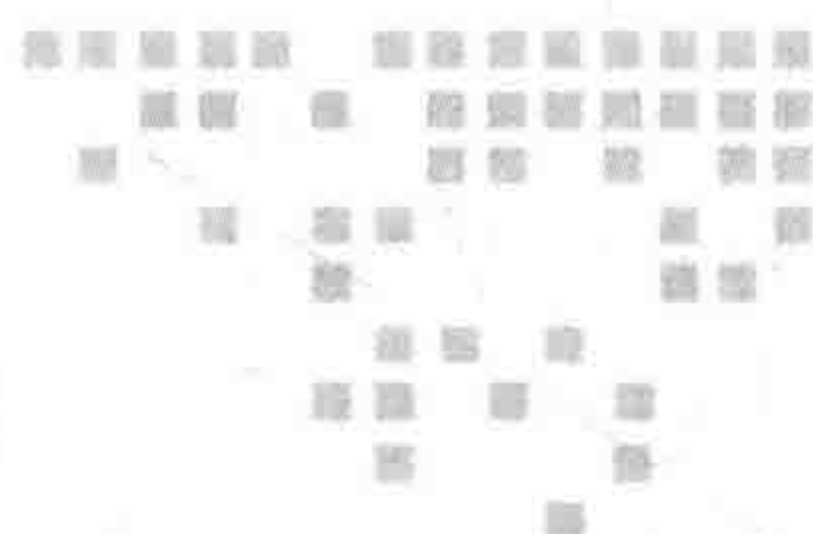


从基本数据类型的恢复方面看,我们解决了二进制翻译中的数据类型恢复问题。对一个过程的 HRTL 中间表示进行分析,主要用到了三种关键技术。首先利用集合理论和数据流分析技术对过程 HRTL 表示中的所有变量重新命名,使得没有联系的数据实体用不同的变量表示。然后利用变量类型的推导规则收集新的变量的类型信息,一般构成一个集合。最后利用格理论确定变量的一个类型,该类型将作为变量在 C 表示中的最终声明类型。对 ITA 系统的测试结果表明该方案是有效的。

从函数类型的恢复方面看,传统的函数类型恢复法是仅从 caller 的角度,以 useundef 变量集合为基础实施函数类型的恢复;改进的函数类型恢复算法是从 caller 和 callee 相结合的角度,以 useundef 和 def 两个变量集合为基础实施函数类型的恢复。分别以 caller 和 callee 为出发点进行分析,一个为正向流分析法,一个为反向流分析法,不论哪一个都有不足之处,只有将两者有机地结合起来才能达到实用的目的。但由于优化级别的不同和机器体系结构的差异,可能会存在某些细节需要特殊考虑,这也是以后要继续研究的内容。

数据类型分析是逆向工程的难点。由于高级程序经编译后数据类型信息在低级代码中已经消失,因此恢复数据的类型特别是高级数据类型几乎是不可能的。本章提出的根据数据在过程中的应用来推导数据类型的方法,虽然在 ITA 系统中已经得到有效的应用,但仅限于恢复数据的基本类型。这里还有许多问题有待进一步完善。





## 反编译的推进 2——控制流恢复实例

在笔者所著的《编译与反编译技术》一书的第 13 章中，我们已经对有关高级控制流恢复技术进行了较为细致的讲述，从理论层面介绍了高级控制流恢复技术，同时也给出了识别 `if...then...else`、`switch`、`while`、`do...while`、`for` 等高级语言控制结构的方法。可以说，基本涵盖了在控制流恢复过程中使用到的多种结构化算法。

本书将从新的切入点出发，并配合两个较为具体的实例，深入讨论如何解决控制流恢复过程中所遇到的实际问题。

第一个例子是基于《编译与反编译技术》一书 12.2.4 节中曾经给出的可以描述汇编指令的描述语言（简称 ASDL）来部分解决间接跳转问题，以实现跳转表的恢复。该实例针对以间接跳转指令形式出现的多分支跳转，给出了一种基于关键语义子树的间接跳转目标解析方案。这部分内容详见 12.1 节。

第二个例子则是通过大胆构造一种名为“功能块”的分析目标，进而在解决“间接转移指令目标地址的确定”问题上做出一种有效的尝试。使用程序执行路径的逆向构造思路，部分解决了针对静态控制流恢复中难以处理间接跳转而导致控制流恢复不完整的问题。这部分内容详见 12.2 节和 12.3 节。

两个示例中所用到的方法，都在笔者平时的科研实践中被证明具有实际价值，具备一定的可操作性和可重用性。

### 12.1 基于关键语义子树的间接跳转目标解析

间接跳转是一种以寄存器或内存单元为间接目标的跳转形式，可以以精简的语句灵活



地实现跳转。但是，该形式在带来灵活和精简的同时，也为静态分析和代码挖掘带来了困难。由于间接跳转多用于利用跳转表实现的多分支跳转，所以，实现跳转表的识别和恢复将对代码挖掘率的提高发挥重要作用。

本节针对现有的跳转表识别方法局限于特定模板或模式的问题，从语义角度抽取跳转表的特征，并在中间表示层上完成跳转目标的自动计算，从而实现与平台、编译器及优化级别均无关的间接跳转目标解析，文中详细阐述了设计思路并结合实例对整个处理过程进行了说明。

### 12.1.1 问题的提出

代码与数据的区分问题既是静态二进制翻译中的基本问题，也是难点问题，原因在于在冯·诺依曼体系结构下代码与数据是以相同形式存放的。当某段数据恰巧与特定机器指令编码相同时，很可能被识别为指令，从而造成解码结果的错误。或者当指令被误当成数据而没有被解码时，则会造成解码结果的不完整。上述两种情况中的任何一种发生，都将无法满足静态二进制翻译对于正确性和完整性的要求。

为了准确地实现代码从数据中的剥离，行进递归（Recursive Traversal）策略被广泛地应用于解码阶段。其主要思路是以程序的主入口点为起点，根据控制流信息选择下一条需要解码的指令位置。优点在于可以保证控制流所访问过的路径中，代码与数据的区分是正确的。

但是，间接跳转目标的不确定性给行进递归策略提出了挑战。所谓间接跳转，即一种在指令中不直接给出跳转目标地址，而是给出跳转目标的存储位置（寄存器或内存单元）的跳转形式。优点在于可以以精简的语句灵活地实现跳转，缺点则是难以静态地获知跳转的目标，从而无法指导解码工作继续正确进行，并由此造成无法构建完整的控制流图，进而使得很多基于控制流图的后续分析工作都失去了意义。

间接跳转多用于多分支跳转的实现，最典型的即为实现高级语言中的 switch-case 语句。在 switch-case 语句中，当分支较多且表达式候选值的分布相对紧凑时，一般会以跳转表的形式存储所有的跳转目标或某个特定地址到跳转目标的偏移，并利用间接跳转指令的形式实现对目标的跳转。

因此，如果可以实现对跳转表的恢复，预测出跳转目标地址，就可以达到提高代码挖掘率的目的。

图 12-1a 中展示了一个典型的 switch-case 的 C 语言程序。b、c、d 是该程序在不同平台及不同优化级别下所生成的指令序列。可以看出，对于同样的 C 语言程序，在不同的平台上生成的指令序列各不相同，即使在同一体系结构下，由于编译器的种类不同、编译器优化级别的不同，生成的指令序列也不同。



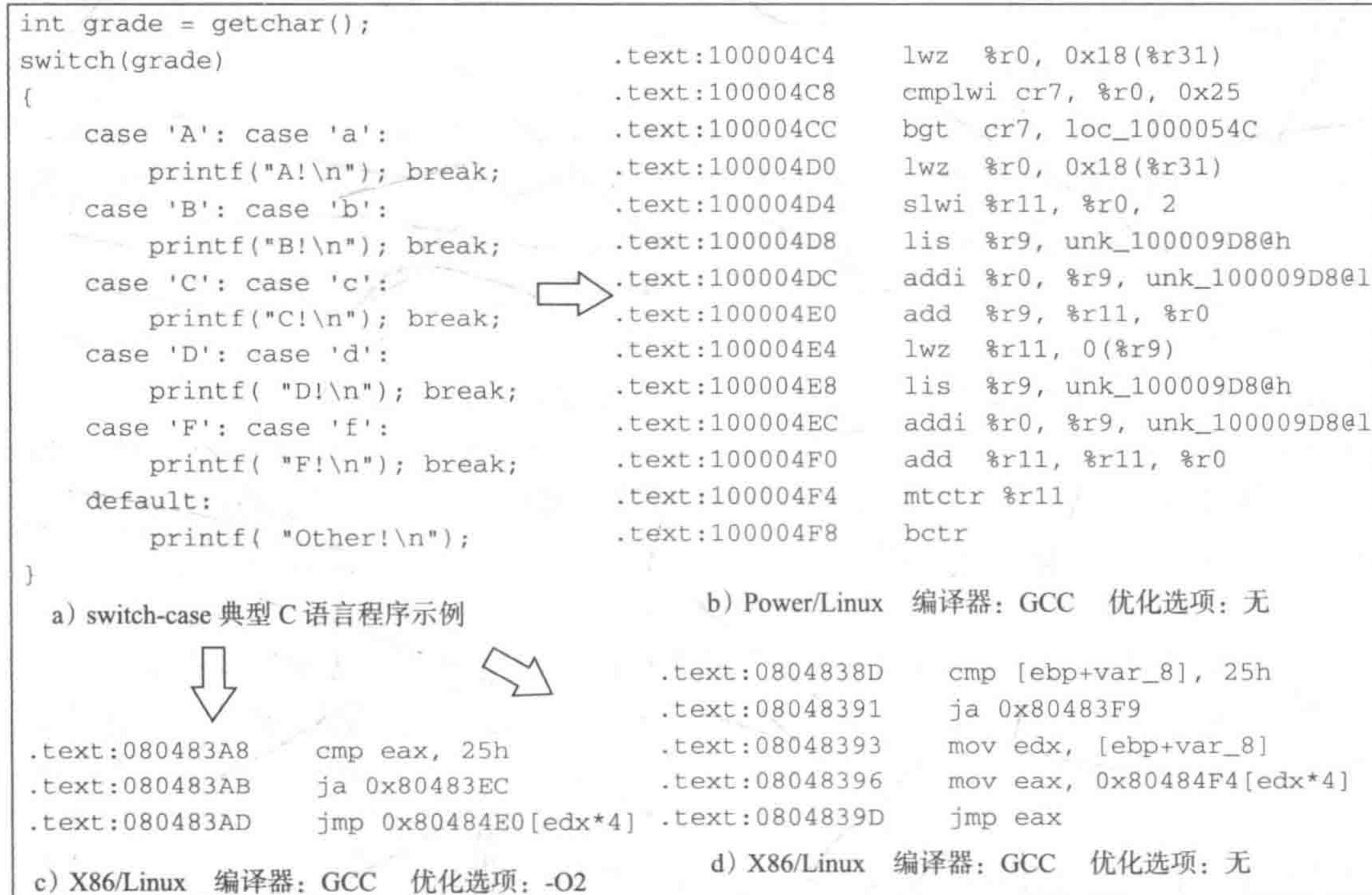


图 12-1 switch-case 的 C 程序及生成代码举例

### 12.1.2 相关工作

正是由于跳转表恢复的重要性，在静态反编译和静态代码分析领域存在着很多相关研究，主要包括如下四种方法。

#### 方法 1: 特定指令模板匹配

对于多分支跳转，同一编译器针对某一特定平台会以相对固定的指令序列加以实现。因此，传统的跳转表识别方法会为某些已知的编译器设定固定的指令序列模板，根据匹配结果计算跳转表信息。该方法的优点在于实现简单且效率高，因此经常被应用于针对某些特定平台程序的静态分析工具中。

反编译工具 dcc 是一款将 80286 上 DOS 可执行文件反编译成 C 代码的工具。该工具就是将 DOS 上的几种编译器所生成的指令模板列举出来用于跳转表的识别。

还有学者在其博士论文中设计了一种模式语言，专门用于描述跳转表结构的指令模板，在一定程度上实现了对多种已知跳转表结构的识别和自动恢复，同时也使得对新跳转表结构模板的添加变得方便、简单。

#### 方法 2: 规范模式匹配

这种方法基于切片和表达式置换，首先将间接跳转指令所在的过程进行反汇编，然后



将反汇编结果转换成规范模式，与事先定义好的规范模式进行匹配，识别出所属模式后，则按照该模式对应的处理方式跳转表信息的获取。

昆士兰大学的 Cifuentes 等人归纳出三种规范模式，并将该方法应用于多源多目标的二进制翻译系统 UQBT 中。后来 Mike 又将其应用到反编译系统 Boomerang 中。另有学者将其应用于等价代码挖掘和二进制代码重用领域。B.De Sutter 也提出了类似的方法，将产生目标地址的程序切片与已定义的切片进行模式匹配，并通过上界的测试和索引值的标准化获得跳转表中的元素个数。

该方法优于方法 1 的地方在于忽略了无关指令并精简了表示形式，且不限定于特定的编译器和特定的指令序列，从而更利于实现平台无关性。但是，由于该方法的成功与否仍然取决于规范模式的定义，而规范模式正是根据现有的编译器代码生成模式提取而得到的，同样没有解决应对未知编译器的问題。

### 方法 3：扩展线性扫描

为了处理嵌入在 text 段中的跳转表，Benjamin Schwarz 等人提出了扩展线性扫描算法。通过观察，他们发现了与跳转表相关的两个特性：一是跳转表项所在的内存位置可重定位，且表项的值是指向 text 段的；二是典型现代架构的指令集中并不允许大量具有上述特性的指令相邻，因而定义了上界（Kmax）。利用上述特性，他们首先将  $N$  条连续的可重定位的 text 段地址中倒数第  $N-K_{\max}$  条指令标记为数据；然后进行线性扫描直至做标记的位置，在此向前检验所有未完整解码的指令并删除。检查最后解码正确的指令，假设有  $m$  ( $0 \leq m \leq K_{\max}$ ) 个地址满足上述条件，那么一定有  $K_{\max}-m$  处未标记为可重定位的 text 段地址应该被标记为数据。

该方法思路新颖，为跳转表的识别提供了新的解决途径。但是，当跳转表的第一个  $K_{\max}$  地址碰巧可以解码成完整的指令时，上述方法则无法正确区分。

### 方法 4：语义图匹配

有学者提出了一种利用语义图解析间接跳转的方法。通过分析间接跳转指令在二进制程序中的常用形式，提取出了相关指令序列表达的语义，采用语义图对其语义进行刻画和存储，并为待分析的指令序列构造语义图，然后调用图匹配算法来判断考察的语义是否与预期语义相匹配。

这种语义图可以有效滤除不相关指令带来的干扰，但是跳转表的识别仍然局限于已定义的语义图形式。从文中描述的例子可以看出，其语义图的定义也是根据已知编译器为间接跳转生成跳转表的常用指令序列提取而得，不同的指令模式需要重新定义不同的语义图，因此无法灵活地应对未知形式。

## 12.1.3 跳转表的语义特征

尽管同一高级语言程序可能被编译成风格各异的指令序列，但是其表达的语义始终是相同的。因此可以提取相关代码片段所表达的语义特征，以此实现跳转表的恢复。



依靠跳转表来实现多分支跳转的指令序列可以根据其主要完成的工作分成两部分：

1) 验证部分，即用于实现索引值的有效性验证的指令片段。当然，验证某些值的无效性是验证有效性的另一种等价形式。

2) 计算部分，即根据有效索引值实现跳转目标计算的指令片段。可以把这部分当作一个函数，以有效索引值为输入，以对应的跳转目标为输出。

其中，验证部分代码通过当前索引值与边界值的比较，判断其是否落在有效范围内，如果超出有效范围，则执行默认分支或紧邻 switch 后的语句，否则将当前索引值输入计算部分，由后者经过特定的转换计算出该索引值所对应的分支地址。

上述过程可以表示成如下两种形式，其中  $F(index)$  代表用于根据索引值计算对应跳转地址的函数，也就是由计算部分代码所构成的转换函数：

形式 A:

```
if (index 不属于有效区间)
    goto other;
else
    goto F(index);
```

形式 B:

```
if (index 属于有效区间)
    goto F(index);
else
    goto other;
```

对于不同的编译器或同一编译器的不同优化级别所生成的代码而言，区别之处主要在于根据索引值计算目标地址的方式即  $F(index)$  函数的具体实现形式不同。上文中提到的特定指令模板或规范模式匹配，实质上也是相当于定义了具体的某几种或某几类  $F(index)$  函数，与生成代码的计算部分代码进行匹配，再根据匹配成功的  $F(index)$  函数种类按既定方式完成跳转表的识别和信息提取。

实际上，真正的计算都是由生成代码自己来完成的，所以只需关注最终生成的指令序列中用于计算部分的指令片段，并根据该部分代码自动生成其对应的  $F(index)$ ，再将有效的索引值输入  $F(index)$  中即可计算出跳转目标。这样就无需关心生成代码的编译器及优化选项的问题，从而摆脱编译器和优化选项相关性的制约。

而关于平台相关性，经过语义映射阶段，不同指令集中的指令已被转换成相同的中间表示语义树，我们选择在语义树层进行分析，从而避免了在指令层分析的平台相关性。

研究发现，无论形式 A 或形式 B 的语义特征，都可以转换成如图 12-2b 的语义树形式。其中实线节点表示为固定的原子操作，对所有的二进制文件都一样；而虚线节点则表示该节点在不同的程序中都存在，但有着不同的表现形式，它可以表示任意的表达式。

值得一提的是，边界的界定需要由取值上界和取值下界来共同完成，所以  $i_{bound}$  实际上指的是取值上界  $i_{High}$  和取值下界  $i_{Low}$ ，二者均为非负整数且  $i_{High}$  不小于  $i_{Low}$ 。另外，在某些情况下，下界直接默认为 0 不再进行验证，所以图 12-2b 中的条件子树可以是一棵界定上界的子树，也可以是分别用于界定上下界的两棵子树，且无前后顺序要求。

另外， $index$  与  $F(index)$  的隶属关系决定了  $index$  所代表的表达式一定是其对应的  $F(index)$  所代表的表达式的子表达式。



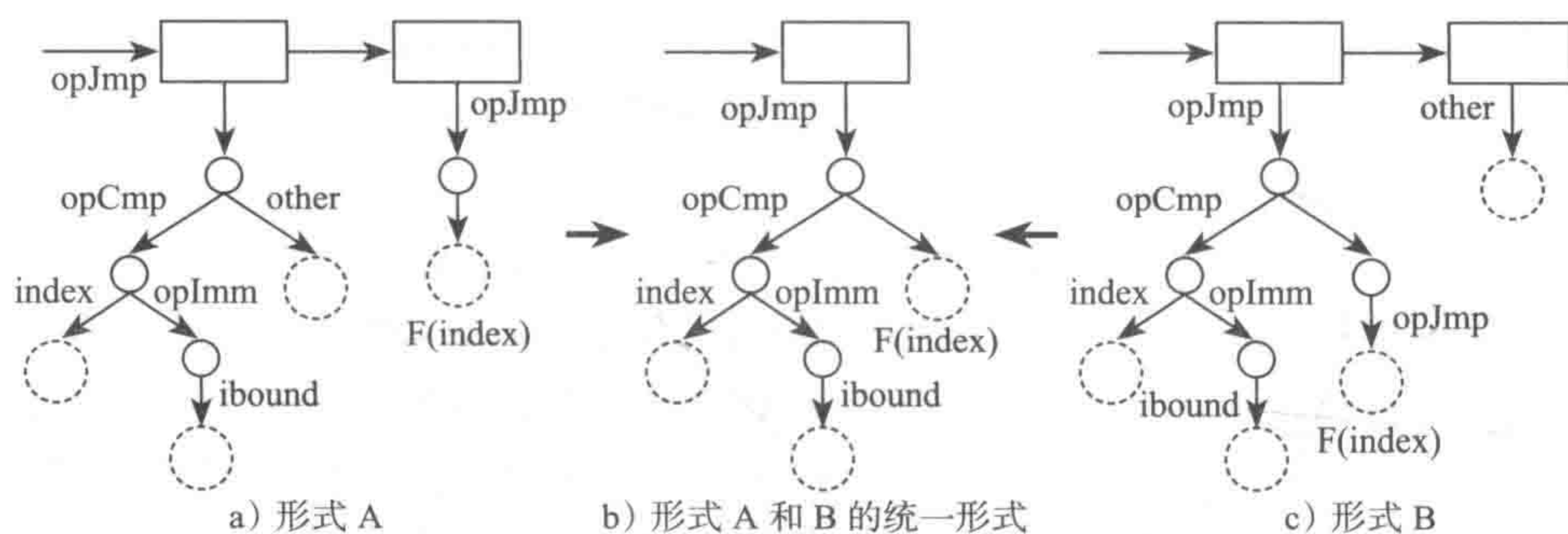


图 12-2 跳转表语义特征的原子语义表示图

可以看出，只要将索引值、索引值的取值上下界及跳转目标计算函数确定了，就可以完成跳转表的恢复。为此，用于存储跳转表信息的数据结构 SWITCH\_INFO 可定义如下：

```
struct SWITCH_INFO {
    Exp*      pIndex;    // 指向存储索引值的数据单元的指针
    Exp*      pIndexFunc; /* 指向 F(index) 表达式即跳转目标存储单元的指针 */
    unsigned int iHigh;  // 索引值取值的上限
    unsigned int iLow;   // 索引值取值的下限
};
```

#### 12.1.4 基于关键语义子树的间接跳转目标解析及翻译

根据上文的描述可知，如果在中间表示语义树上完成对  $F(\text{index})$  函数的自动生成，那么就可以实现平台、编译器及优化级别均无关的跳转表恢复，从而满足系统对该模块的要求。为了实现上述目的，本节阐述一种基于关键语义子树的间接跳转目标解析策略，主要分三步完成。

**步骤 1：**提取关键语义子树。所谓关键语义子树（Critical Semantic Subtree, CSS），即以二进制文件生成的语义树为基础，忽略其中对跳转表的生成使用均无影响的语句，最终提取出的一棵形如图 12-2b 的子树。CSS 提取成功的同时也意味着跳转表恢复所需的各种信息，如  $\text{index}$  的边界值、 $F(\text{index})$  等也已获得。

**步骤 2：**计算间接跳转目标。根据 CSS 得到的相关信息，在该部分要做的主要工作就是将有效的索引值一一代入  $F(\text{index})$ ，计算出对应的跳转目标。该部分的难点问题就是如何将语义子树表示的  $F(\text{index})$  转换为可以自动完成对输入值进行计算的函数体。

**步骤 3：**对同目标的跳转分支进行聚合。经过对跳转目标的预取，建立索引值与跳转目标的映射关系，在一般情况下，跳转目标个数都会少于索引值的个数，也就是说会有多个索引值对应同一跳转目标的情况。为了减少生成代码的数量，可将映射同一目标地址的索引值进行归类合并。下面将对上述 3 个步骤进行详细介绍。

##### 1. 关键语义子树的提取

###### (1) 提取的位置和时机

为了保证提取算法的正确性和有效性，在设计提取算法之前，首先应该对提取的位置



和时机进行选取。

首先是提取位置的选取。由于跳转最终是由间接跳转指令实现的，在跳转之前必然已经完成索引值的验证并将计算得出的跳转目标放入寄存器或内存单元中以供间接跳转指令使用。因此只需从间接跳转指令开始依控制流从后向前切片，可大大提高子树提取的效率。

其次是提取时机的选取。当解码模块识别出间接跳转指令后，此时的解码尚未完成，并没有建立相对完整的控制流图，不易于实现沿控制流从后向前切片。因此，采取继续解码直至所有控制流可到位置均解码完成为止。除了对控制流向未知的基本块标记为 Unkown 外，可以构建相对完整的控制流图。另外，此时语义映射也已完成，语义树已经生成，由上文可知，在语义树层进行相关处理更利于平台无关性的实现。

## (2) 提取路径的选择算法

在阐述提取路径的选择算法之前，首先对基本块的定义及类型特点进行简单介绍，以便于相关内容的理解。

所谓基本块 (Basic Block)，是指程序中一段顺序执行的语句序列，程序控制流无中止且无分支 (出口除外) 地从其入口指令 (第一条指令) 顺序执行到出口指令 (最后一条指令)。

根据基本块中出口指令的类型，可以把基本块分为七类，具体的分类及类型描述见表 12-1。

表 12-1 基本块分类及类型描述

基本块类型	类型描述
One-Way	出口指令为无条件跳转到可知的目标地址，具有一条出边
Two-Way	出口指令为条件跳转到可知的目标地址，具有两条出边
N-Way	出口指令为索引 / 间接跳转到可知的目标地址， $n$ 个分支地址存放在索引表中，具有 $n$ 条出边
Call	出口指令为过程调用，有两条出边：一条指向过程调用后的指令，另一条指向被调过程
Return	出口指令为过程的返回指令，不存在出边
FallThrough	基本块紧邻的下一条指令地址为某个分支指令的跳转目标。只有一条出边
Unkown	出口指令为索引或间接跳转或调用未知目标，只有一条出边

选择提取路径时，由基本块的定义及类型的特点可知，同一基本块的指令只需按其在基本块内的顺序由后向前依次进行处理即可，而处于基本块开始位置的指令在选择下一条处理指令时则需要根据其所在基本块的入边情况进行区分处理。具体实现流程如下。

### 算法 12.1 提取路径选择算法

算法目的：确定切片路径，选择下一条需要分析的指令

输入：CurInstr —— 指向切片处理刚刚结束的指令的指针，初始指向当前处理的间接跳转指令

CurBB —— 指向 CurInstr 所在的基本块的指针

IsNot —— 间接跳转所在路径是否为某条件跳转的 false 分支，可用于标识是索引值无效性验证 (形式 A) 还是有效性验证 (形式 B)，初始为 false



输出: PreInstr —— 查找成功后为待分析的指令, 否则为 NULL

查找成功根据实际情况对 IsNot 置位

1. 如果 CurInstr 不是基本块 CurBB 的入口指令, 按照从后向前的顺序依次选取同一基本块内紧邻的前驱指令设为 PreInstr, 并返回
2. 如果 CurBB 入边个数为 0, PreInstr 设为 NULL, 返回
3. 如果入边个数为 1, 则选择唯一入边基本块 PreBB 加入当前路径并转 9
4. 如果入边基本块中包含 FallThrough 类型的 PreBB, 选择此基本块加入当前路径并转 9
5. 如果入边基本块中包含 One-Way 类型的 PreBB, 选择此基本块加入当前路径并转 9
6. 如果入边基本块中包含 Two-Way 类型的 PreBB, 且 CurBB 为 PreBB 的 false 分支, 选择此基本块加入当前路径并转 9
7. 对 CurBB 的所有入边进行检验, 如果所有入边都被处理过, PreInstr 设为 NULL, 返回。否则, 选择其第一条未处理过的入边基本块 PreBB 加入当前路径
8. 如果 PreBB 为 Two-Way 类型, 且 CurBB 为 PreBB 的 true 分支, 则将 IsNot 置为 true
9. 获取 PreBB 的出口指令设为 PreInstr, 返回

### (3) 关键语义子树提取算法

由间接跳转目标解析所需的信息可得出, 关键语义子树提取的主要目的有三个: 确定上下界、确定索引值 index 并确定索引值计算函数  $F(index)$ 。

根据上文的分析, 以间接跳转指令为提取的起点, 按照路径选择模块提供的提取路径, 沿控制流从后向前依次查找用于边界检验的操作, 同时对代表跳转目标存储位置的语义树进行子树传播, 直至有效的边界检验操作确定完毕, 从而记录下该部分指令序列对索引值进行操作的整个过程并融合到代表跳转目标存储位置的语义树中。

其中, 子树传播是指沿控制流从后向前查找定值语义树, 并判断其是否为当前跳转目标语义树的某子树定值, 如果是的话, 则将其左子树在当前跳转目标语义树中出现的所有位置用其右子树替换。

边界检验在中间表示层上表示为比较操作, 根据其验证的边界类型而选择不同的比较操作。在形式 A 中, 由于是索引值无效性验证 (此时 IsNot 为 false), 所以检验上界时多使用大于类比较操作 (大于、大于等于), 检验下界时则多使用小于类比较操作 (小于、小于等于), 而在形式 B 中则恰恰相反。

由于并非所有的比较操作都是用于边界检查, 所以需要对其进行验证。根据索引值 index 与  $F(index)$  的关系, 当比较操作的变量部分是跳转目标语义树的一部分则将其选为备选索引值, 而且:

- 1) 当上下界验证均存在且二者所在的指令所提取出的备选索引值相同, 则认为提取成功。
- 2) 当上下界验证均存在但二者所在的指令所提取出的备选索引值不同, 则以上界得到



的备选索引值为准,并将下界置为0,认为提取成功。

3) 当只存在上界验证,则将下界置为0,认为提取成功。

4) 其他情况均为提取失败。

当在当前路径提取失败后,选择新的路径重新开始提取,如果所有的路径都处理过而仍未找到索引值检验相关信息,则认为此次提取失败。

关键语义子树提取算法基本流程描述如下。

### 算法 12.2 关键语义子树提取算法

算法目的: 提取与跳转表实现相关的关键节点构建关键语义子树

输入: CurInstr —— 当前间接跳转指令

CurBB —— CurInstr 所在基本块

输出: pSwitchInfo —— 存储跳转表相关信息的结构体

1. 初始化临时变量 E\_Td、E\_High、ihigh、E\_Low、ilow、E\_Cmp 均为空
2. 获取 CurInstr 的目标子树, 设为 E\_dest
3. 获取当前切片路径上的下一条指令设为 PreInstr。如果 PreInstr 不是 CurBB 的指令, 将 E\_dest 赋值给 E\_Td 并转 6
4. 如果 PreInstr 不是定值语句, 转 3
5. 如果 PreInstr 的左子树 lhs 为 E\_dest 的子树, 用 PreInstr 的右子树 rhs 替换 lhs 在 E\_dest 中的所有出现。转 3
6. 如果 PreInstr 不是比较操作, 转 9
7. 如果 IsNot 为 True 且 PreInstr 为小于类比较操作, 或 IsNot 为 False 且 PreInstr 为大于类比较操作, 获取 PreInstr 比较操作的左子树赋值给 E\_High, 获取 PreInstr 比较操作的右子树赋值给 ihigh, 并转 10
8. 如果 IsNot 为 False 且 PreInstr 为小于类比较操作, 获取 PreInstr 比较操作的左子树赋值给 E\_Low, 获取 PreInstr 比较操作的右子树赋值给 ilow 并转 10
9. 如果 PreInstr 为定值语句, 且 PreInstr 的左子树 lhs 为 E\_Td、E\_High、E\_Low 或 E\_Cmp 的子树, 用 PreInstr 的右子树 rhs 替换 lhs 在上述四者中的所有出现
10. 如果 E\_Low 不为空且为 E\_Td 的子树, 将 E\_Low 赋值给 E\_Cmp, 如果 E\_High 不为空且为 E\_Td 的子树, 将 E\_High 赋值给 E\_Cmp
11. 如果 E\_Cmp 不为空, 且与 E\_Low、E\_High 均相等, 转到 15
12. 获取当前切片路径上的下一条指令 PreInstr, 如果 PreInstr 不为空, 转 6
13. 如果 E\_Cmp 或 E\_High 为空, 转 16
14. 如果 E\_Low 和 E\_Cmp 不相等, ilow 置为 0
15. 设置 pSwitchInfo\_iHigh = ihigh, pSwitchInfo\_iLow = ilow, pSwitchInfo\_pIndex = E\_Cmp, pSwitchInfo\_pIndexFunc = E\_Td, 退出



16. 为 CurBB 选择新的切片路径, 并获取下一条指令 PreInstr, 如果 PreInstr 不为空, 将 E\_dest 赋给 E\_Td 并转 6。否则, 退出

#### (4) 实例说明

下面以图 12-1 的例子为实例具体讲解算法实现过程。

对图 12-1d 中的语句经过语义映射模块转换后得到语义树如图 12-3。鉴于篇幅有限, 图 12-3 中只画出了前驱基本块的最后一条指令, 即 0x8048391 处的条件跳转指令, 根据第 3 章的处理策略, 跳转的判定条件已经以条件子树的形式融合在该指令中。

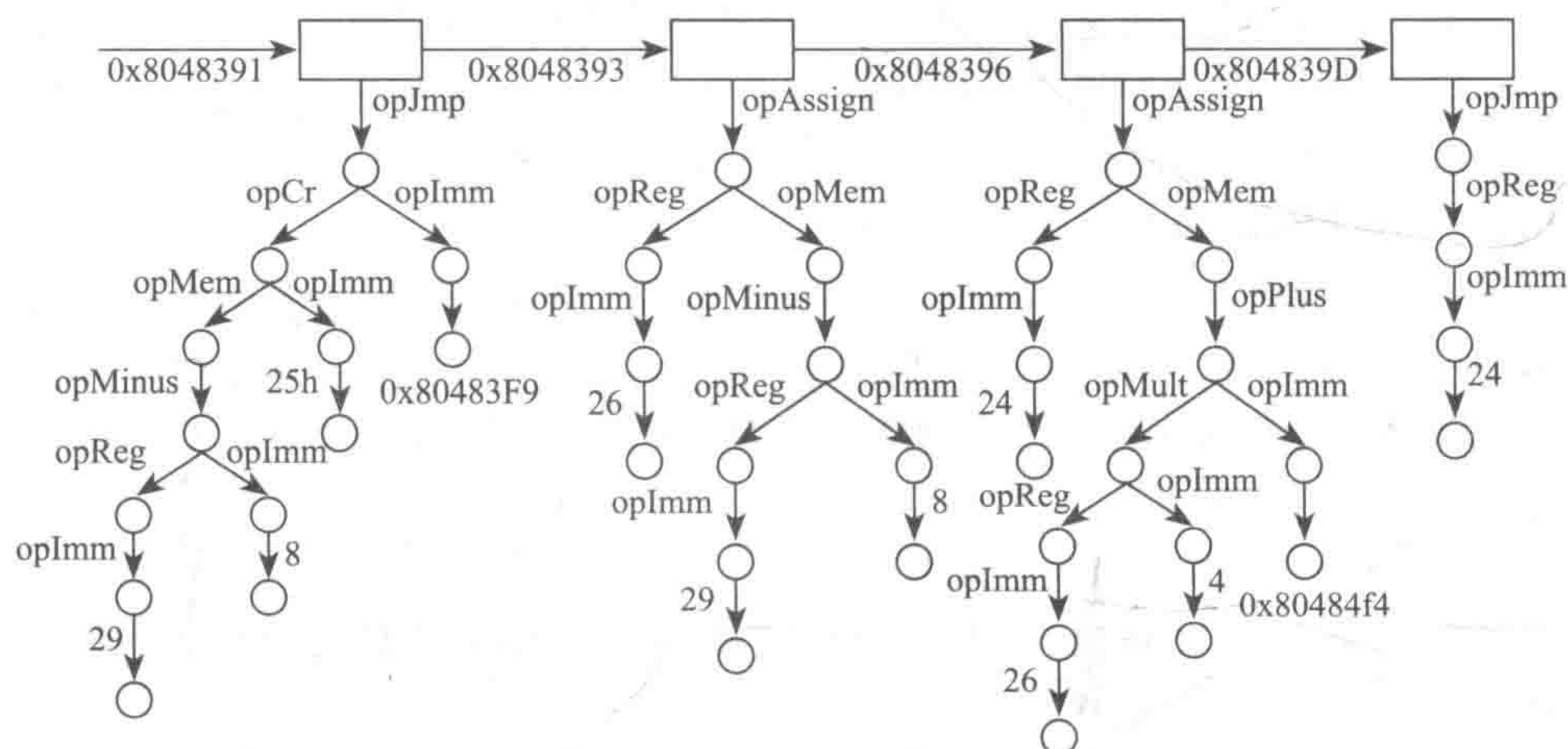


图 12-3 图 12-1d 的代码片段生成的语义树

根据提取算法获取间接跳转指令即 0x804839D 处指令的语义树, 其目标子树即为 E\_dest。在此例中, 初始状态下 E\_dest 如图 12-4 中阴影部分。

获取 0x804839D 处指令的前驱 0x8048396 处指令后, 发现该指令对 E\_dest 定值, 用其右子树替换 E\_dest 得到当前的 E\_dest, 如图 12-5 中阴影部分。

获取 0x8048396 处指令的前驱 0x8048393 处指令后, 发现 0x8048393 处指令对图 12-5 的阴影部分的子树 (即方框部分) 定值, 将该子树替换成 0x8048393 处指令的右子树, 此时, E\_dest 变成了如图 12-6 中阴影部分。

此时已经到达本基本块的第一条指令, 然后获取本基本块的前驱基本块, 根据算法获取条件子树的左子树为备选 index, 如图 12-7 中阴影部分。

不难发现此时的备选 index 为图 12-6 中 E\_dest 的一部分, 因此可以认为提取过程已经完成, 生成的关键语义子树 (CSS) 如图 12-8 所示。

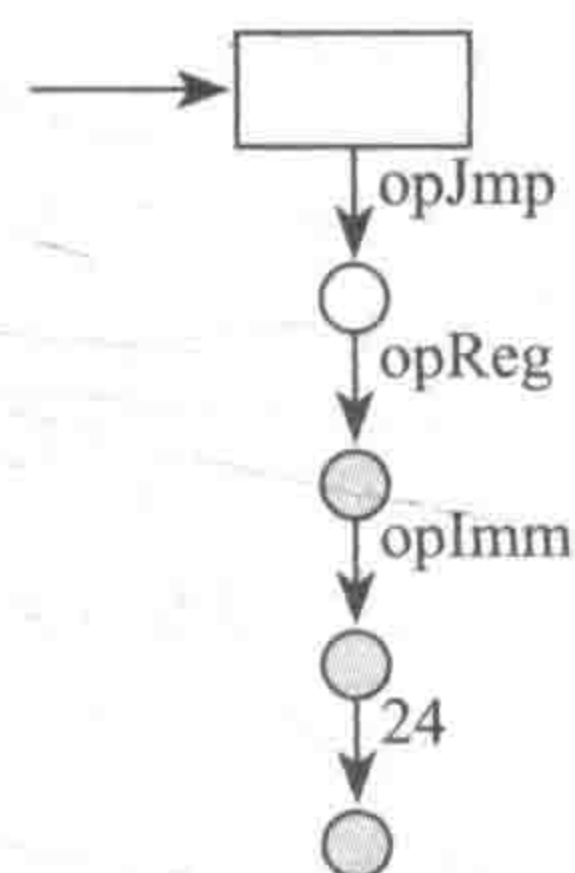


图 12-4 关键语义子树提取实例 (a)



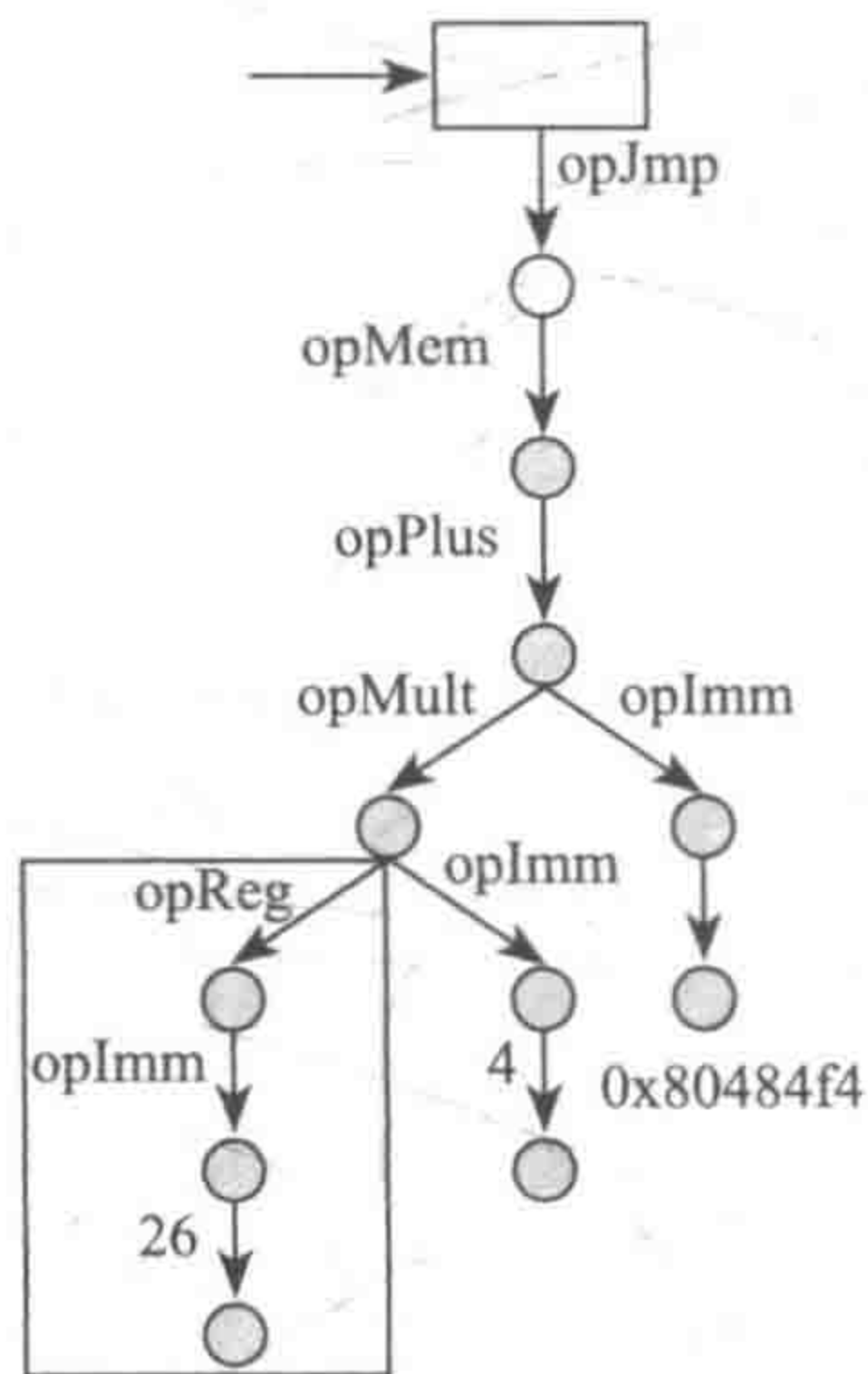


图 12-5 关键语义子树提取实例 (b)

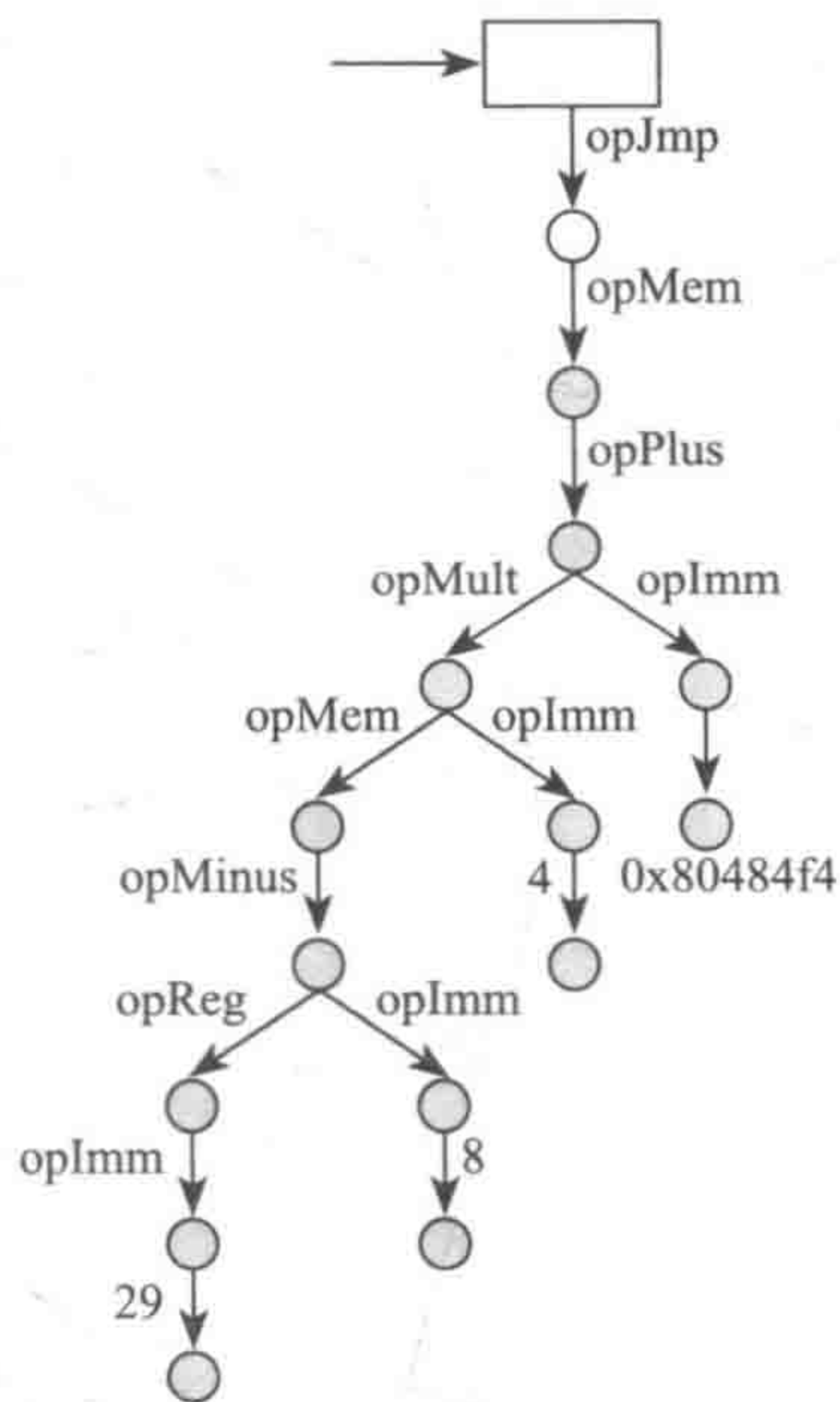


图 12-6 关键语义子树提取实例 (c)

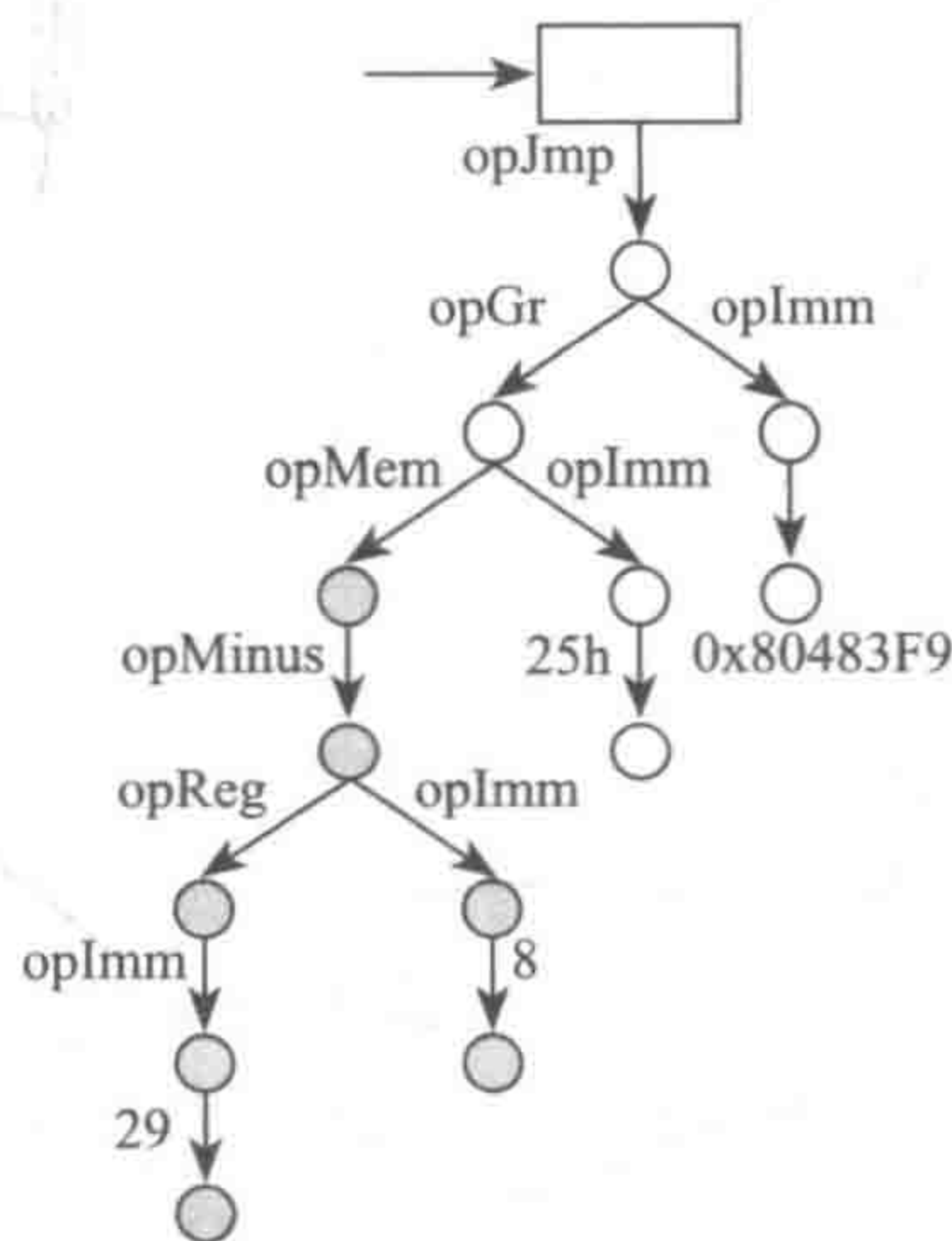


图 12-7 关键语义子树提取实例 (d)

根据 CSS 可以得到图 12-1d 中 pSwitchInfo 的所有成员信息，pIndex 为指向图 12-8 中 index 子树的指针，pIndexFunc 为指向 F(index) 子树的指针，iHigh=25h，iLow 默认为 0。可以看出，在提取关键语义子树的过程中，提取成功的同时也完成了相关信息的获取。

2. 间接跳转目标的预取

(1) 跳转目标预取算法

提取出关键语义子树 (CSS) 后，即可以根据 CSS 获取计算跳转目标的相关信息。从



存储跳转表信息的结构体 pSwitchInfo 中取得 pIndexFunc 成员即指向 F(index) 子树的指针，用于跳转目标的计算。

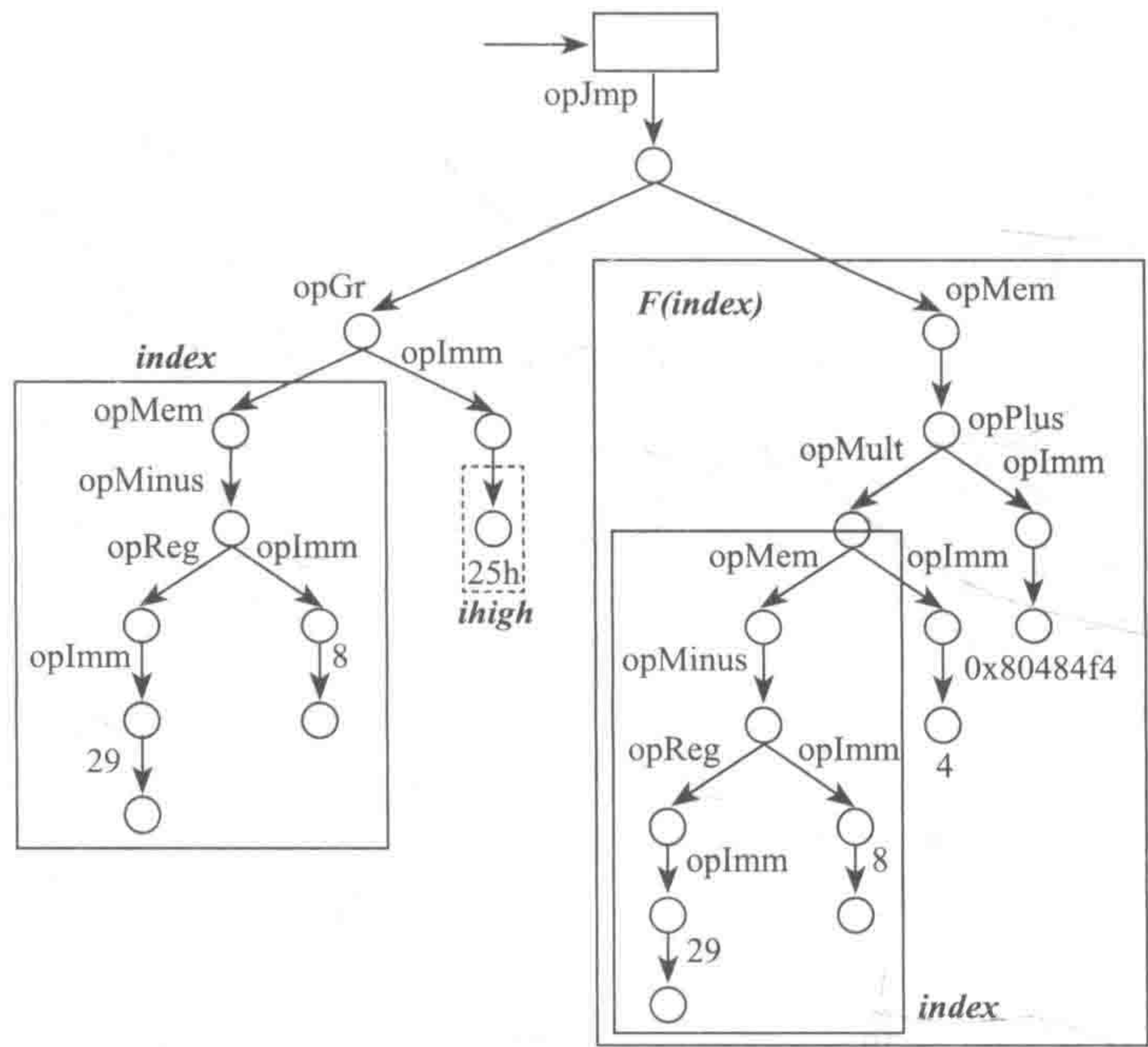


图 12-8 关键语义子树提取实例 (e)

第一步，对代表索引值存储单元的子树部分模糊化。

所谓模糊化，即忽略掉代表索引值存储单元的子树部分的具体组合形式，取而代之一个简单的 Terminal 类节点，并用 opIndex 作为索引，用于标识索引值。

该操作的提出依据在于此时索引值的上下界均已知，索引值必为该范围内的非负整数，只需穷举有效索引值进行下一步计算即可，不必关心其具体存储位置。

第二步，对 F(index) 子树的节点类型进行处理，以保证所有信息在获取过程中与上下文无关。

所谓与上下文无关，是指在对节点进行处理时，针对同一文件，无论程序执行上下文如何改变均得出同样的结果。这将使得后续的计算过程只需针对 F(index) 语义树进行，而不必关心程序执行情况。

该操作主要是针对 F(index) 语义树中寄存器类存储单元而提出的。由于大部分寄存器用于存储临时变量，所以在大多数情况下，很难甚至无法说出某个寄存器在某一时刻的具体内容。因此，需要对 F(index) 语义树中寄存器类存储单元进行子树传播，使得整个 F(index) 语义树仅包含索引值、立即数、各种运算操作和内存取值操作。

尽管本过程与 CSS 提取过程的处理目的不同，但二者处理方法和处理路径是基本相同的，我们选择将二者结合在一起进行，避免大量重复处理。

根据寄存器用途的不同，分别采取不同的处理方法：



1) 程序计数器 PC: 在一般情况下, PC 用于指向当前执行指令的下一条指令。在我们的设计中, 进入一条指令时, PC 默认为当前指令的地址, 对控制流有影响的指令其 PC 值的变化情况会在语义描述时进行显性模拟。因此, 对于  $F(index)$  中出现的 PC 会在 CSS 提取及无关化处理中由其所在指令的地址决定并替换。

2) 返回参数寄存器: 对于存储返回参数的寄存器出现在  $F(index)$  语义树中的情况, 如果在无关化处理中遇到函数调用指令, 则认为该寄存器的值由调用函数定值, 需要对调用函数进行分析; 否则与 CSS 提取时子树传播的处理方式相同。

3) 通用寄存器: 在遇到函数调用指令时, 默认为调用函数对通用寄存器无影响而直接跳过。遇到其他指令时, 处理方式同 CSS 提取过程中子树传播的处理方式相同。

**第三步**, 自顶向下依次对  $F(index)$  子树的节点进行分析, 由下向上进行计算。

语义转换模块是要将识别出来的节点类型在目标指令集中找到语义等价的对应元素, 而本部分则是将识别出来的节点类型直接转换成具体的运算和数值。

例如, 对 `opPlus` 类型的节点, 语义转换模块会将其翻译成 “add” 指令操作符输出, 而在本部分则是将两个子树的值计算出来后直接相加并输出和。

经过第二步的处理后,  $F(index)$  语义树仅包括四种节点, 在计算时采取不同处理:

- 1) 索引值: 即 `opIndex`, 直接返回当前正在参与计算的索引值数值即可。
- 2) 立即数: 直接将其数值返回。
- 3) 运算类操作: 如 `opPlus`、`opMinus` 等, 将此类节点的子树计算所得值进行对应的操作后, 将计算结果返回上一层。
- 4) 内存取值操作: 即 `opMem (M[])` 节点, 由 `opMem` 节点的定义可知, 其子树经计算后必然是一个地址值, 不妨设为  $A$ 。在文件装载时, 装载模块会为待分析的二进制文件构建内存映像, 并记录下虚拟地址与物理地址的关系, 由此可以将  $A$  转换得出其在内存中的真正位置  $A'$ 。再根据环境变量配置文件获取位宽及存储大小端的定义, 最终将  $A'$  中的内容读取出来返回。其中, 位宽决定读取数值时的长度 (以字节为单位), 存储大小端决定读取数值的方式。

图 12-9 给出了跳转目标计算部分的代码片段。

本质上讲, 计算过程就是要完成某个表达式求值的过程。如果将存储单元集到数集的映射函数  $\sigma$  所构成的集合称为状态集合, 那么  $\sigma(X)$  则代表着状态  $\sigma$  下存储单元  $X$  的值或内容。在讨论状态  $\sigma$  下表达式  $e$  求值过程时, 可以用序偶  $\langle e, \sigma \rangle$  来表示状态  $\sigma$  下表达式  $e$  等待求值, 并把该序偶与数集间的求值关系定义为:

$$\langle e, \sigma \rangle \rightarrow n$$

它表示状态  $\sigma$  下表达式  $e$  的求值结果是  $n$ 。

此时, 求值关系的形式规范可以由一组规则给出:

- 1) 索引值求值:  $\langle index, \sigma \rangle \rightarrow i$ , 其中,  $i$  为当前参与计算的索引值,  $index$  是状态  $\sigma$  下令  $\sigma(index) = i$  的存储单元。
- 2) 立即数求值:  $\langle n, \sigma \rangle \rightarrow n$ , 显然, 任一立即数的求值结果就是本身。



```

int BB::getDestForind(Exp *exp, int index, UserProc* proc)
{
    int result;
    .....
    switch(op)
    {
        case opIndex:
            result=index;
            break;
        .....
        case opPlus:
            op1=getDestForind(b->getSubExp1(), index, proc);
            op2=getDestForind(b->getSubExp2(), index, proc);
            result=op1+op2;
            break;
        case opMinus:
            op1=getDestForind(b->getSubExp1(), index, proc);
            op2=getDestForind(b->getSubExp2(), index, proc);
            result=op1-op2;
            break;
        .....
        case opMem:
            result=prog->readNative4(getDestForind(b->getSubExp1(), index, proc));
            break;
        .....
    }
    return result;
}

```

图 12-9 跳转目标计算部分的代码片段

3) 运算类操作求值:  $\frac{\langle e_0, \sigma \rangle \rightarrow n_0 \quad \langle e_1, \sigma \rangle \rightarrow n_1}{\langle e, \sigma \rangle \rightarrow n}$ , 这代表的是一类求值规则, 其中,  $e_0$  与  $e_1$  为  $e$  的两个子表达式, 且  $e$  由  $e_0$  与  $e_1$  经过某种运算得到,  $n$  是  $n_0$  与  $n_1$  经过对应运算后所得结果。

如  $\frac{\langle e_0, \sigma \rangle \rightarrow n_0 \quad \langle e_1, \sigma \rangle \rightarrow n_1}{\langle e_0+e_1, \sigma \rangle \rightarrow n}$  代表的是加法运算的求值规则, 其中  $n$  是  $n_0$  与  $n_1$  相加的结果。

4) 存储单元求值:  $\langle X, \sigma \rangle \rightarrow \sigma(X)$ , 根据定义, 在一个状态下存储单元的求值结果就是它的内容。

可以看出, 这组规则与上文中对求值过程直观的非形式化描述非常接近。在一般情况下, 规则由实线上方的前提和实线下方的结论构成, 当规则前提为空时, 实线可以省略。

## (2) 实例说明

在计算跳转目标之前, 首先将 pIndexFunc 所指向的表达式中用于表示 index 的子表达式用 opIndex 节点代替, 如图 12-10。

转换成图 12-10 后, 对树中节点类型进行检验, 只有 opIndex、立即数常量、算术运算以及存储器寻址和立即寻址, 没有未知内容的寄存器存在。因此, 省略未知寄存器内容切



片的步骤，直接进行  $F(index)$  计算。由边界值可知，索引值的有效范围为 0 到 25h，将该区间内的所有整数——代入计算即可。以  $index=2$  为例，说明计算的过程。

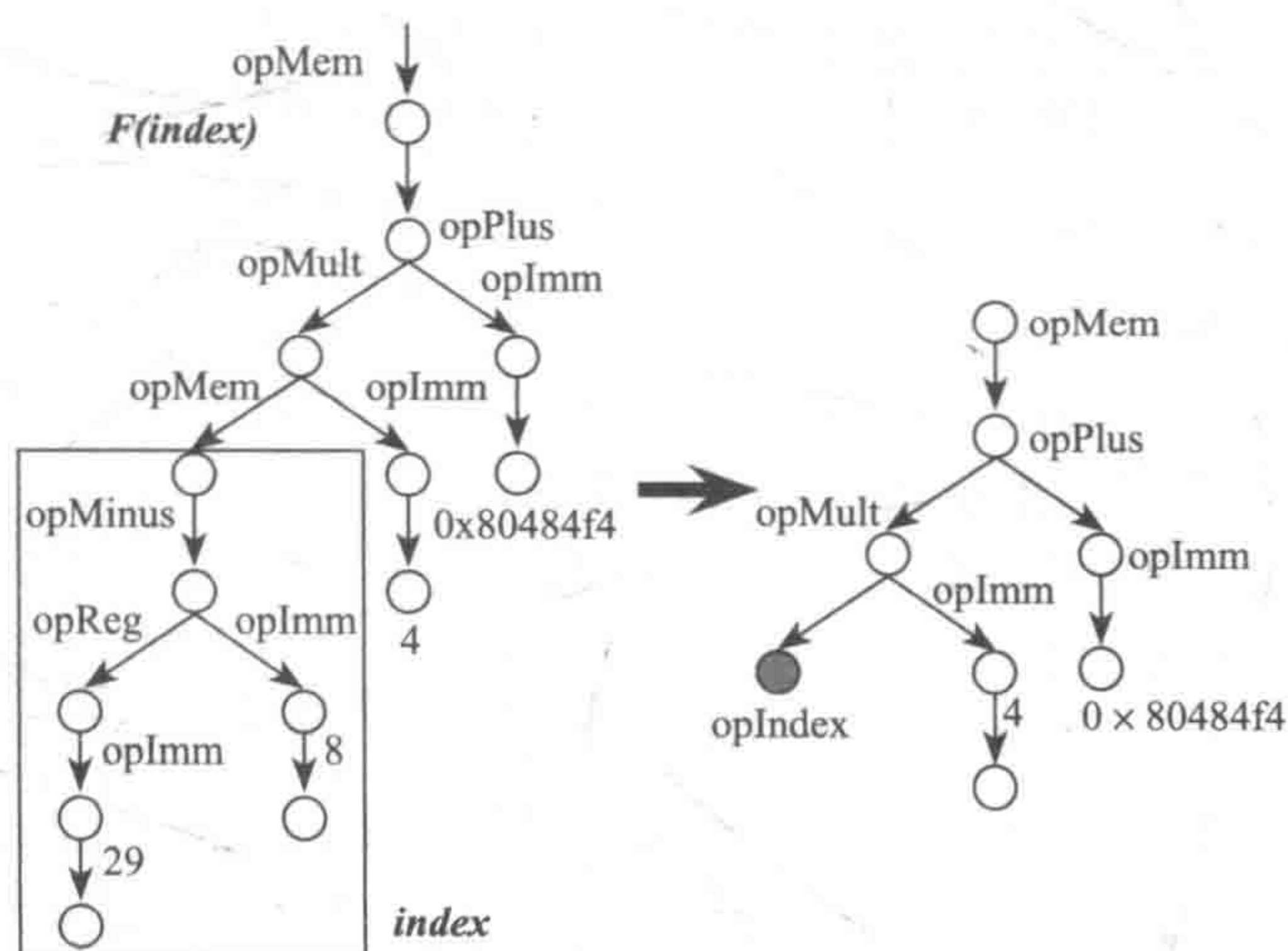


图 12-10  $F(index)$  子树模糊化举例

计算模块由顶向下判断节点类型并进行计算，从树的角度看，可以看成是以深度优先的原则按照由根至子树的顺序进行，同一层次的子树则按由左至右顺序进行。首先遇到的节点为  $opMem$ ，则其子表达式必然可以计算得出某个地址，因此将其子树  $sub_1$  输入计算模块，等得到计算结果后再调用存储器取值函数  $readNative$  进行读取。

$$Result = readNative(i_1); \quad (12-1)$$

对  $sub_1$  的根节点进行判断后发现为  $opPlus$  类型，则需要依次将左右子树  $sub_2$  和  $sub_3$  输入计算模块进行计算，得出结果后再相加。

$$i_1 = i_2 + i_3; \quad (12-2)$$

对  $sub_2$  的根节点进行判断后发现为  $opMult$  类型，则需要依次将左右子树  $sub_4$  和  $sub_5$  输入计算模块进行计算，得出结果后再相乘。

$$i_2 = i_4 * i_5; \quad (12-3)$$

$sub_4$  是  $opIndex$  类型节点，将当前  $index=2$  作为计算结果返回上一层。

$$i_4 = index = 2; \quad (12-4)$$

$sub_5$  是  $opImm$  类型节点，将其数值作为计算结果返回上一层。

$$i_5 = 4; \quad (12-5)$$

此时  $i_4$  和  $i_5$  均为已知，代入式 (12-3) 得到  $i_2=8$ ，返回上一层。再计算  $sub_3$ ，判断为  $opImm$  类型，取出其数值作为结果返回上一层。

$$i_3 = 0x80484f4; \quad (12-6)$$

此时  $i_2$  和  $i_3$  均已知，代入式 (12-2) 得出  $i_1$  为  $0x80484FC$ ，由  $readNative$  函数读出该位置的数据为  $0x80483C3$ ，由此得出，索引值为 2 时对应的跳转目标为  $0x80483C3$ 。

该计算过程可以利用上文中定义的规则用如下推导实例直观地表示出来：



$$\begin{array}{c}
 \frac{\langle index, \sigma \rangle \rightarrow 2 \quad \langle 4, \sigma \rangle \rightarrow 4}{\langle index * 4, \sigma \rangle \rightarrow 8} \quad \langle 0x80484F4, \sigma \rangle \rightarrow 0x80484F4 \\
 \hline
 \langle (index * 4) + 0x80484F4, \sigma \rangle \rightarrow 0x80484FC \\
 \hline
 \langle m[(index * 4) + 0x80484F4], \sigma \rangle \rightarrow 0x80483C3
 \end{array}$$

### 3. 同目标跳转分支的聚合

#### (1) 分支聚合的提出

经过间接跳转目标预取后，会为每一个有效索引值与其对应的跳转目标建立一个映射关系  $\langle index_i, Addr_i \rangle$ ，为了完整地展现所有的跳转关系，翻译时就要将每个对应关系都表现出来。如果采用单独处理策略，每一个对应关系都会生成一组比较和条件跳转指令的组合，由此翻译得到的程序代码膨胀率大，执行效率大大下降。

但是，在实际情况中，多数的索引值都会共享同一跳转目标。如图 12-1d 的例子中，索引值的有效范围为 0~25h，而其中有 28 个索引值都是指向代表 default 的分支地址 0x80483F9，且这 28 个索引值有 26 个是连续的。因此，如果以跳转目标为划分标准，将具有相同跳转目标的分支聚合成一组进行翻译，将有效提升翻译代码的质量。

另外，对于连续的索引值指向同一跳转目标的情况，如果只记录上下界的值并由此界定一个区间，生成代码时只需判定当前索引值是否落入此区间就可以判断其是否应该跳转到该区间所对应的跳转目标，进而避免了一一比较所带来的代码膨胀。

#### (2) 数据结构与算法流程

根据上述分析设计了数据结构 CaseLimit 和 CaseInfo 用于表示分支的信息。

```

struct CaseLimit{
    int Case_low;           // 取值下界
    int Case_high;          // 取值上界
};

struct CaseInfo{
    ADDRESS m_Dest;         // 跳转目标地址
    std::list<CaseLimit*> m_CaseLimit; // 对应 m_Dest 的所有索引值取值区间
};

```

其中，CaseLimit 用于表示一个区间，该区间代表了一个连续的取值范围，在该范围内的所有非负整数（包括区间上下界）都为有效索引值，且对应的跳转目标相同。CaseInfo 中包含了用于表示本分支所对应的跳转目标地址的成员 m\_Dest，以及链表 m\_CaseLimit，用于记录对应本分支的所有索引值区间。

进行同目标跳转分支的聚合时，按照索引值的大小次序依次进行。由目标预取算法可知 index 取值是从 pSwitchInfo.iLow 开始依次递增，所生成的跳转目标链表 dest 也是按此顺序存储的。

取当前映射关系中的地址部分  $Addr_i$ ，判断该地址是否已被解码过：

1) 如果  $Addr_i$  没有被解码，则为其生成以  $Addr_i$  为 m\_Dest 的分支信息，并初始化唯一



的取值区间为  $[index, index]$ ，同时将该地址列为新解码地址输入解码模块进行解码。

2) 如果  $Addr_i$  已被解码，则现有的 Case 信息集合中必然已有针对此地址的 CaseInfo，取出该 CaseInfo 中具有最大上界值的区间，判断当前的 index 值是否恰好与该区间的最大上界值连续（即为最大上界值加 1），连续的话则将该区间的最大上界值更新为当前的 index 值；否则，为 CaseInfo 生成新的取值区间，此时该区间只包含一个元素，即初始化的上下界均为 index。

如此依次进行，直至所有的跳转目标全部处理完毕。

除此之外，由于在上述过程中产生了新解码的代码片段，需要对原有控制流图进行修订，将间接跳转所在基本块 BB 的 Unkown 类型修改为 N-Way 类型，并将各个新解码的地址列为 BB 的出边。

同目标跳转分支聚合的具体算法流程见图 12-11。

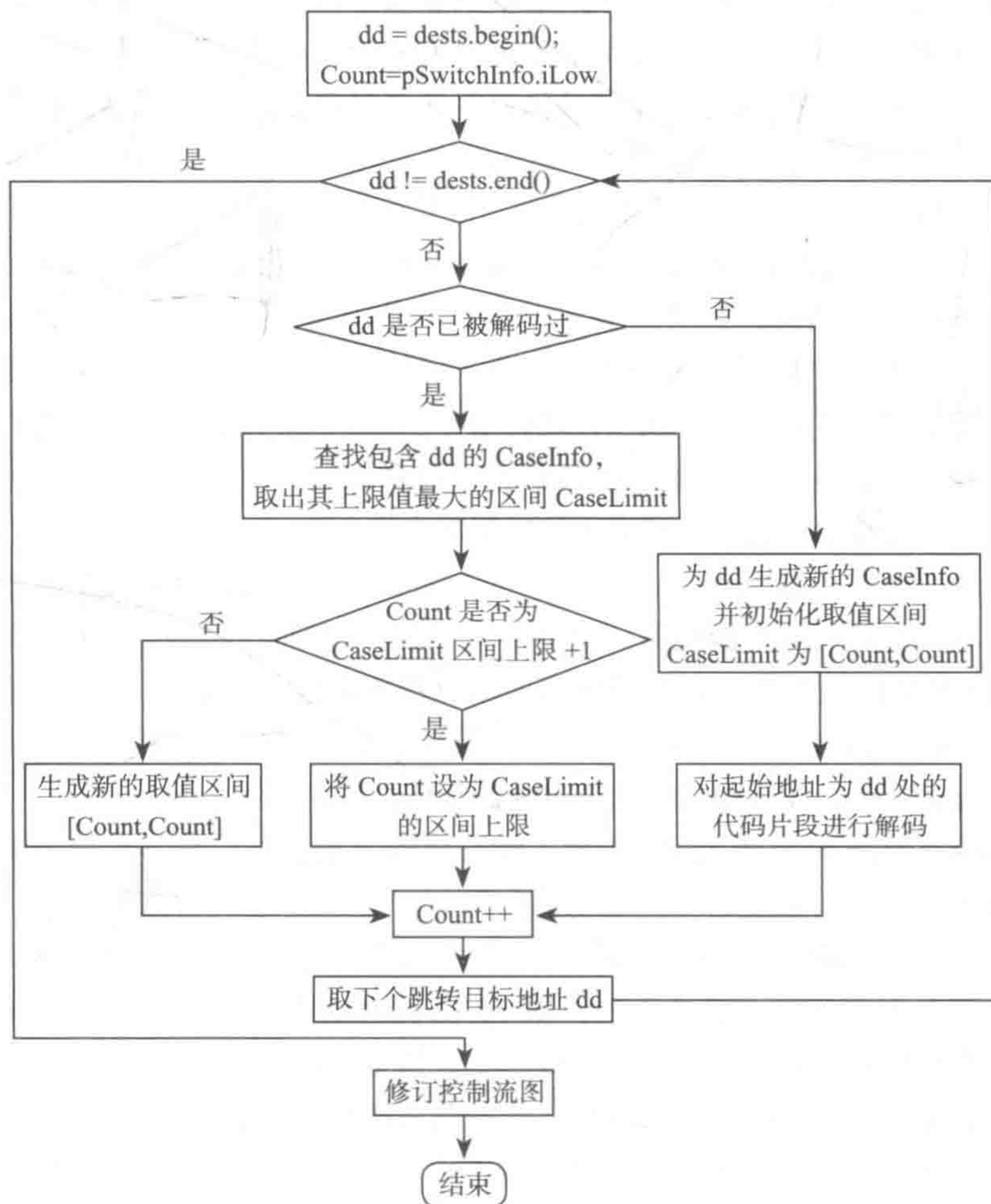


图 12-11 分支聚合算法流程



## 12.2 功能块概念的引入

程序分析是对计算机程序行为进行自动分析的过程。基本块在程序分析中起着重要作用且应用广泛，如程序编译、程序优化、逆向工程、程序正确性验证、软件安全分析、程序等价判定和核心模块筛选等。针对间接转移指令的目标地址确定，基于基本块的程序分析可以解决少数问题，但它无法处理大多数的情况。

### 12.2.1 分析单位

程序分析经常使用的四个分析单位为：基本块、扩展型基本块、轨迹树和区间。

基本块是程序指令的一个线性序列，它只拥有一个入口和一个出口，当线性指令序列的第一条指令被执行时，此指令序列的最后一条指令也必须被执行。

基本块概念的提出至少已经四十年，基本块在程序分析领域应用广泛。例如：程序分析使用的控制流和数据流分析技术是基于基本块而实施的；程序验证的有些应用同样是基于基本块进行的；基于基本块判定程序优化前后的语义等价；使用符号执行和定理证明对两个基本块或函数进行比较，发现两个二进制文件语义上的不同；基于基本块确定软件的核心部分；基于基本块改善指令的发射等。

近来有人提出扩展型基本块 (Extended Basic Block) 的概念，学术界对其还没有形成统一的定义。其中一种说法为，扩展型基本块是由一组基本块组成的超块，它仅有一个入口但有多个出口，且不包含控制流图中的汇合点。另外一种说法为，扩展型基本块是由不包含标号的连续指令序列组成，它可以包含跳转指令。

轨迹树 (Trace Tree) 是一组基本块的集合，拥有一个入口和多个出口，可以包含控制流图中的汇合点。实践中，轨迹树通常比扩展型基本块大。

当把程序划分为基本块集合，并用节点表示基本块，用边表示基本块间的控制流，则程序可以表示成一个有向的控制流图。给定一个节点  $h$ ，区间 (interval)  $I(h)$  是一个单入口的最大子图， $h$  是区间的入口节点并且所有的闭合路径都包含节点  $h$ 。 $h$  被称为区间头或者头节点。一个区间仅包含一个入口节点，但可能有多个出口节点。划分区间时只有当一节点的所有直接前驱都已经在区间中，此节点才能加入区间。

某程序的控制流图  $G$  如图 12-12 所示，针对此图分别按基本块、扩展型基本块、轨迹树和区间划分如下。

由图 12-12 可知，基本块的划分很显然；依据参考文献 [46] 和 [49] 给出的区间划分算法，对控制流图  $G$  的区间划分是唯一的；图 12-12 同时给出了图  $G$  扩展型基本块的一种划分，仅为了对比四种划分的不同；关于图  $G$  的轨迹树也仅给出一部分，用于对比分析。

由区间的定义及划分原则知区间可以包含控制流图中的汇点，如区间  $\{3, 4, 5, 6, 7\}$  包含汇点 7。扩展型基本块不包含控制流图中的汇点，而轨迹树可以包含控制流图中的汇点且它为程序的执行轨迹，因此，在一般情况下扩展型基本块、轨迹树和区间的大小关系为



$L_{\text{扩展型基本块}} \leq L_{\text{区间}} < L_{\text{轨迹树}}$ , 其中  $L_{\text{扩展型基本块}}$  为扩展型基本块的大小,  $L_{\text{轨迹树}}$  为轨迹树的大小,  $L_{\text{区间}}$  为区间的大小, 且大小定义为包含的节点个数。

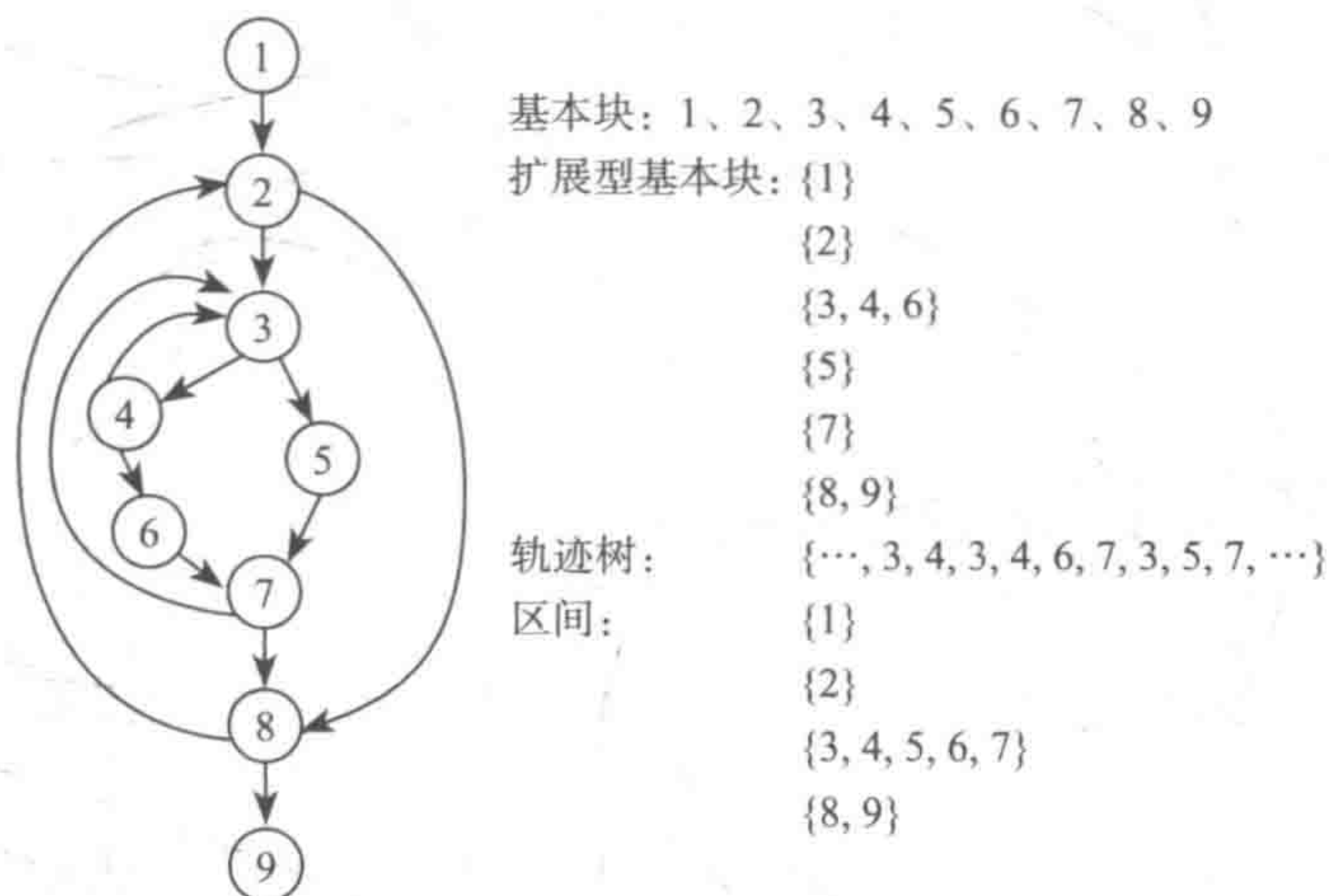


图 12-12 某程序的基本块、扩展型基本块、轨迹树和区间划分

由于针对扩展型基本块没有明确的划分准则, 而轨迹树的轨迹大小又不确定, 因此下面的论述只针对基本块和区间。

### 12.2.2 基于基本块的分析

后续分析都是针对程序 switch.c (统计录入成绩的等级数量) 进行的, 此程序包含 switch 分支结构, 如下所示。首先, 给出基本块和区间的划分算法。

```
main()    // 程序 switch.c
{
    int grade;
    int aCount = 0, bCount = 0, cCount = 0, dCount = 0, fCount = 0;
    printf("Enter the letter grades.\n");
    printf("Enter the EOF character to end input.\n");
    while ( ( grade = getchar() ) != EOF) {
        switch (grade) {    /* 在 while 中嵌套使用 switch */
            case 'A': case 'a': /* 当 grade 取值为 A 或 a 时 */
                ++aCount;
                break;
            case 'B': case 'b': /* 当 grade 取值为 B 或 b 时 */
                ++bCount;
                break;
            case 'C': case 'c': /* 当 grade 取值为 C 或 c 时 */
                ++cCount;
                break;
            case 'D': case 'd': /* 当 grade 取值为 D 或 d 时 */
                ++dCount;
                break;
            case 'F': case 'f': /* 当 grade 取值为 F 或 f 时 */
                ++fCount;
```



```

        break;
    case '\n': break;
    default: /* 当输入为其他字母时 */
        printf("Incorrect letter grade entered. Enter a new grade.\n "); break;
    }
}
printf("\nTotals for each letter grade are:\n");
printf(" A: %d\n B: %d\n C: %d\n D: %d\n F: %d\n ", aCount, bCount, cCount,
dCount, fCount);
return 0;
}

```

---

### 算法 12.3 把三地址指令序列划分成基本块

输入：一个三地址指令序列。

输出：输入序列对应的一个基本块列表，其中每个指令恰好被分配给一个基本块。

Begin

1. 首先确定首指令（某个基本块的第一条指令）的集合，所使用的规则如下：

- 1) 序列的第一条指令是首指令；
- 2) 任意一个条件或无条件转移指令的目标指令是首指令；
- 3) 紧跟在一个条件或无条件转移指令之后的指令是首指令。

2. 每个首指令对应的基本块包括了从它自己开始，直到下一条首指令（不含）或者序列的结束指令之间的所有指令。

End

---

### 算法 12.4 把控制流图划分成区间

输入：一个程序的控制流图  $G$ 。

输出：输出控制流图  $G$  对应的区间集合。

Begin

1. 建立一个头节点列表  $H$ ，用  $G$  的入口节点对其初始化。

2. 对  $H$  中的节点  $h$ ，按以下步骤构建区间  $I(h)$ 。

- 2.1 把节点  $h$  添加为  $I(h)$  的第一个节点。
- 2.2 对  $G$  中的任一节点  $b$ ，只要它的所有直接前驱都已在  $I(h)$  中，则添加节点  $b$  到  $I(h)$ 。
- 2.3 重复 2.2 操作直到没有节点能添加到  $I(h)$ 。

3. 构造区间集合，按以下步骤构建。

- 3.1 把所有不在  $H$  中且不在  $I(h)$  中而其直接前驱处于  $I(h)$  中的节点添加到  $H$ 。
- 3.2 把  $I(h)$  添加到区间集合。

4. 选择  $H$  中未处理过的下一个节点重复步骤 2、3 和 4，直到  $H$  中没有未被处理的节点，结束。

End

---



### 1. 能胜任的情况

在 IA64 上使用 GCC 编译器, 采用 -O0 优化选项编译 switch.c 生成可执行程序 a.out, 使用基于控制流的递归扫描反汇编器处理 a.out 生成的汇编代码如图 12-13 所示, 为了节省空间, 用序号替代指令对应的内存地址。

B1	L001: alloc r34=ar.pfs,10,4,0	B8	L045: adds r15=8,r35;;
	L002: mov r35=r12		L046: ld4 r14=[r15]
	L003: adds r12=-32,r12		L047: shladd r15=r14,3,r0
	L004: mov r33=b0;;		L048: addl r14=88,r1;;
	L005: adds r14=-12,r35;;		L049: ld8 r14=[r14];;
	L006: st4 [r14]=r0		L050: add r15=r15,r14
	L007: adds r14=-8,r35;;		L051: ld8 r14=[r15];;
	L008: st4 [r14]=r0		L052: add r14=r14,r15
	L009: adds r14=-4,r35;;		L053: mov b6=r14
	L010: st4 [r14]=r0		L054: br.few b6;;
	L011: mov r14=r35;;		...
	L012: st4 [r14]=r0	B9	L085: addl r14=96,r1;;
	L013: adds r14=4,r35;;		L086: ld8 r36=[r14]
	L014: st4 [r14]=r0		L087: mov r32=r1
B2	L015: addl r14=72,r1;;	B10	L088: br.call.sptk.many <printf>;
	L016: ld8 r36=[r14]		L089: mov r1=r32
	L017: mov r32=r1	B11	L090: br.few L025
	L018: br.call.sptk.many <printf>;		L091: addl r14=104,r1;;
B3	L019: mov r1=r32;;	B12	L092: ld8 r36=[r14]
	L020: addl r14=80,r1		L093: mov r32=r1
B4	L021: ld8 r36=[r14]		L094: br.call.sptk.many<printf>;
	L022: mov r32=r1		L095: mov r1=r32
B5	L023: br.call.sptk.many <printf>;		L096: adds r15=-12,r35
	L024: mov r1=r32		L097: adds r16=-8,r35
	L025: mov r32=r1		L098: adds r17=-4,r35
	L026: br.call.sptk.many <getchar>;		L099: mov r18=r35
B6	L027: mov r1=r32		L100: adds r19=4,r35;;
	L028: mov r14=r8		L101: addl r14=112,r1;;
	L029: adds r15=-16,r35;;		L102: ld8 r36=[r14]
	L030: st4 [r15]=r14		L103: ld4 r37=[r15]
	L031: adds r16=-16,r35;;		L104: ld4 r38=[r16]
	L032: ld4 r14=[r16];;		L105: ld4 r39=[r17]
	L033: cmp4.eq p7,p6=-1,r14		L106: ld4 r40=[r18]
	L034: (p06) br.cond.dptk.few L036		L107: ld4 r41=[r19]
B7	L035: br.few L091		L108: mov r32=r1
	L036: adds r15=-16,r35;;	B13	L109: br.call.sptk.many <printf>;
	L037: ld4 r14=[r15]		L110: mov r1=r32
	L038: adds r15=-10,r14		L111: mov r14=r0;;
	L039: adds r16=8,r35;;		L112: mov r8=r14
	L040: st4 [r16]=r15		L113: mov.i ar.pfs=r34
	L041: adds r16=8,r35;;		L114: mov b0=r33
	L042: ld4 r16=[r16];;		L115: mov r12=r35
	L043: cmp4.ltu p6,p7=92,r16		L116: br.ret.sptk.many b0;;
	L044: (p06) br.cond.dptk.few L085		

图 12-13 反汇编 a.out 生成的部分代码及基本块划分



按照算法 12.3 对生成代码的基本块划分见图 12-13，共划分成 13 个基本块，生成代码对应的控制流图如图 12-14 所示。

由于基本块 B8 中的指令 L054 为间接跳转指令，基本块 B8 的后继基本块无法被确定，因此，针对此程序构建的控制流图不完整。

为构建程序的完整控制流图，需要确定基本块 B8 的后继基本块，根据算法 12.3 提供的规则 2 即要获得指令 L054 的跳转目标地址，也即寄存器 b6 的值。

获取指令 L054 使用的寄存器 b6 的值，最直接的办法是使用 Mark Weiser 提出的程序切片技术，获得程序中与寄存器 b6 在 L054 处状态相关的所有指令，即切片 (L054, b6) 如图 12-15 所示。

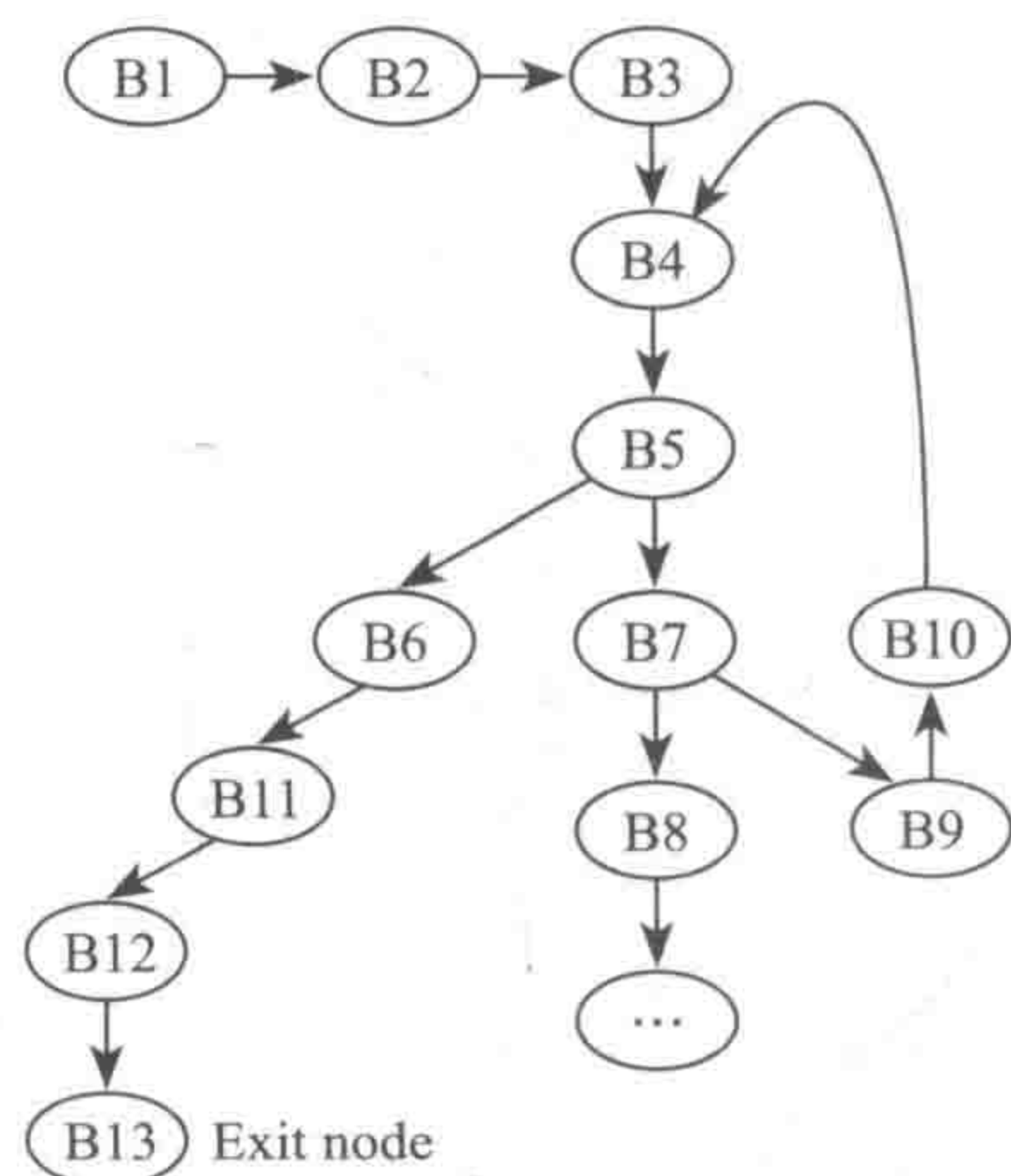


图 12-14 图 12-13 中代码对应的控制流图

```

L002:  mov r35=r12
      ...
L045:  adds r15=8,r35;;
L046:  ld4 r14=[r15]
L047:  shladd r15=r14,3,r0
L048:  addl r14=88,r1;;
L049:  ld8 r14=[r14];;
L050:  add r15=r15,r14
L051:  ld8 r14=[r15];;
L052:  add r14=r14,r15
L053:  mov b6=r14
L054:  br.few b6;;
  
```

图 12-15 切片 (L054, b6) 对应的指令

由切片 (L054, b6) 知，L054 处的 b6 依赖于：L045 处寄存器 r35 的值和 r35+8 所指内存单元存储的值、L047 处寄存器 r0 的值和 L048 处寄存器 r1 的值。而寄存器 r35 存储的是寄存器 r12 的值，因在指令 L054 所处的过程体中 r35 仅在 L002 处被定值一次。寄存器 r0 是常量寄存器，存储的值总是 0；寄存器 r1 是专用寄存器，存储的是全局数据指针，在程序中保持不变；寄存器 r12 是专用寄存器，存储的是栈指针，在当前过程中保持不变。因此，基本块 B8 包含的指令完全可决定寄存器 b6 在 L054 处的值。

## 2. 不能胜任的情况

同样使用 GCC 编译器，采用 -O2 优化选项编译 switch.c 生成的代码 b.out 不同于 0 级优化代码 a.out，反汇编 b.out 生成的汇编代码见图 12-16。

按照算法 12.3 对生成代码的基本块划分见图 12-16，共划分成 11 个基本块，生成代码对应的控制流图见图 12-17。

由于基本块 B6 中的指令 L035 为间接跳转指令，基本块 B6 的后继基本块无法被确定，因此，对此程序构建的控制流图不完整。



L001: alloc r39=ar.pfs,14,8,0		L030: ld8 r14=[r14];;	
L002: addl r40=80,r1		L031: add r15=r15,r14	
L003: mov r32=r1		L032: ld8 r14=[r15];;	B6
L004: mov r38=b0;;		L033: add r14=r14,r15	
L005: ld8 r40=[r40]	B1	L034: mov b6=r14	
L006: mov r34=r0		L035: br.few b6;;	
L007: mov r35=r0		...	
L008: mov r36=r0		L044: addl r40=96,r1	
L009: mov r37=r0		L045: mov r32=r1;;	B7
L010: br.call.sptk.many <printf>;		L046: ld8 r40=[r40]	
L011: mov r33=r0		L047: br.call.sptk.many <printf>;	B8
L012: mov r1=r32;;		L048: br.few L016	
L013: addl r40=88,r1;;	B2	...	
L014: ld8 r40=[r40]		L051: addl r40=104,r1	
L015: br.call.sptk.many <printf>;		L052: ld8 r40=[r40]	B9
L016: mov r1=r32		L053: br.call.sptk.many <printf>;	
L017: addl r15=24,r1		L054: mov r1=r32	
L018: mov r32=r1;;	B3	L055: mov r41=r34	
L019: ld8 r14=[r15]		L056: mov r42=r35	
L020: ld8 r40=[r14]		L057: mov r43=r36	
L021: br.call.sptk.many <getchar>;		L058: mov r44=r37	B10
L022: cmp4.eq p7,p6=-1,r8		L059: mov r45=r33;;	
L023: mov r1=r32	B4	L060: addl r40=112,r1	
L024: (p07) br.cond.spnt.few L051		L061: ld8 r40=[r40]	
L025: adds r8=-10,r8;;		L062: br.call.sptk.many <printf>;	
L026: addl r14=120,r1		L063: mov r1=r32	
L027: dep.z r15=r8,3,32	B5	L064: mov r8=r0	
L028: cmp4.ltu p6,p7=92,r8		L065: mov.i ar.pfs=r39	B11
L029: (p06) br.cond.spnt.few L044		L066: mov b0=r38	
		L067: br.ret.sptk.many b0;;	

图 12-16 反汇编 b.out 生成的部分代码及基本块划分

同样为了构建程序的完整控制流图，需要确定基本块 B6 的后继基本块，即获取寄存器 b6 的值。寄存器 b6 在 L035 处的切片 (L035, b6) 如图 12-18 所示。

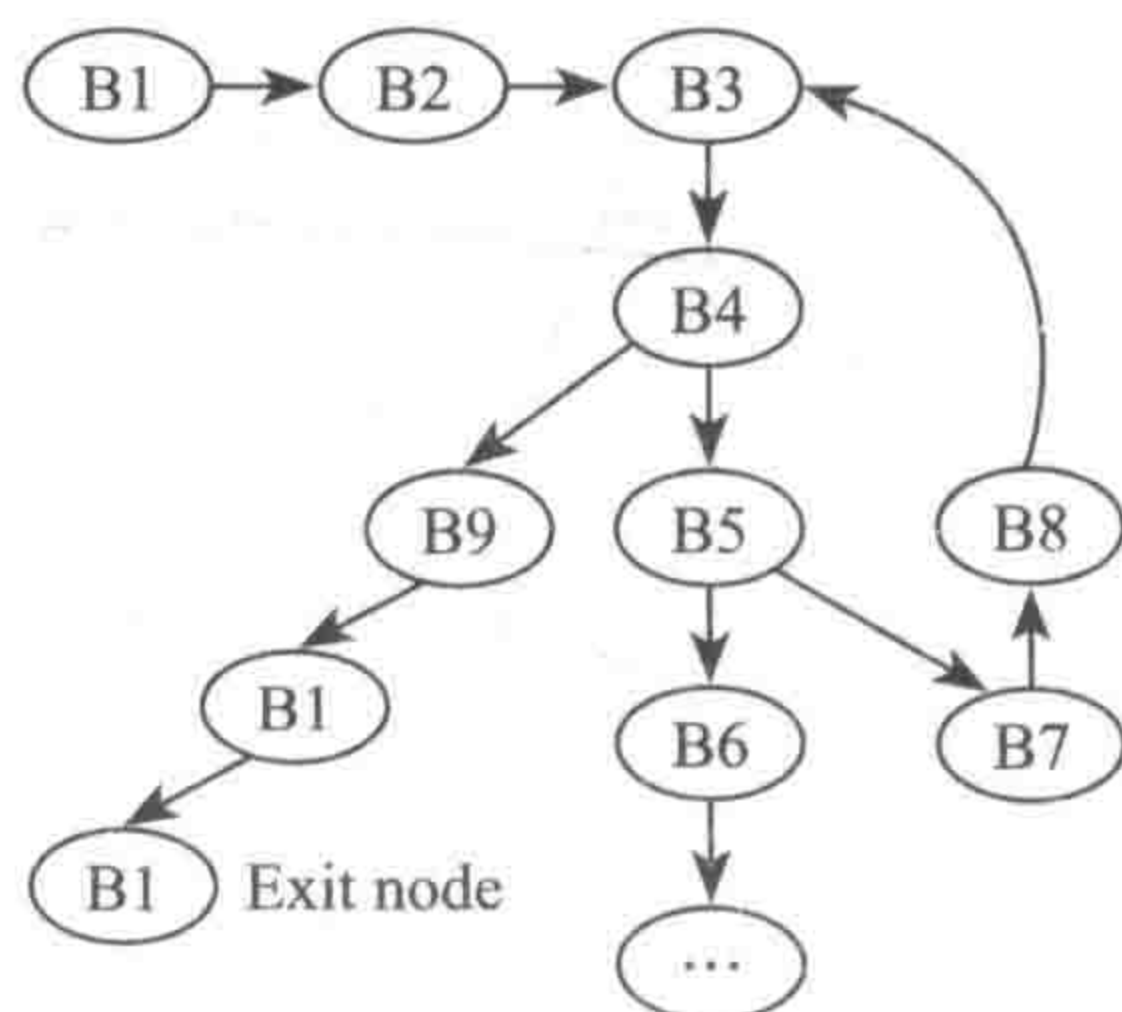


图 12-17 图 12-16 代码对应的控制流图

```

L018: mov r32=r1;;
L021: r8=br.call.sptk.many <getchar>;
L022: cmp4.eq p7,p6=-1,r8
L023: mov r1=r32
L024: (p07) br.cond.spnt.few L051
L025: adds r8=-10,r8;;
L026: addl r14=120,r1
L027: dep.z r15=r8,3,32
L028: cmp4.ltu p6,p7=92,r8
L029: (p06) br.cond.spnt.few L044
L030: ld8 r14=[r14];;
L031: add r15=r15,r14
L032: ld8 r14=[r15];;
L033: add r14=r14,r15
L034: mov b6=r14
L035: br.few b6;;
  
```

图 12-18 切片 (L035, b6) 对应的指令



由切片 (L035, b6) 可知, L035 处的 b6 依赖于 L018 处寄存器 r1 的值和 L021 处寄存器 r8 的值。因为寄存器 r8 用于存储函数的返回值, 因此 L021 处对 r8 的赋值是隐式显化。切片中存在两条关键的指令: ① 指令 L024, 如果谓词寄存器 p7 为真则基本块 B5 和 B6 将都不被执行, 然而对谓词寄存器 p7 的赋值受寄存器 r8 控制; ② 指令 L029, 如果谓词寄存器 p6 为真则基本块 B6 将不被执行, 然而对谓词寄存器 p6 的赋值同样受寄存器 r8 控制。因此, 本质上 L021 处寄存器 r8 的值控制着指令 L035 执行与否, 基本块 B6 包含的指令 L030~L035 无法决定寄存器 b6 在 L035 处的值; 由基本块 B3、B4、B5 和 B6 组成的整体虽可决定寄存器 b6 的值, 但包含了一些不相关指令。

按照算法 12.4 对图 12-17 给出的控制流图进行区间划分, 可划分为 {B1, B2} 和 {B3, B4, B9, B10, B11, B5, B6, B7, B8}。如果用包含基本块 B6 的区间来确定 B6 的后继基本块, 它与切片 (L035, b6) 相比更大, 包含了太多的无用指令。对于更复杂的程序, 问题可能更严重。

### 12.2.3 功能块

计算机上运行的程序, 无论大 (整个程序或者程序模块) 或者小 (函数或者程序片段) 必包含三个部分: 输入数据、执行代码和输出数据。如无明显的输入输出数据, 代码执行前后的内存状态依然可认为是其输入数据和输出数据, 由三部分组成一个功能明确的程序片段。

#### 1. 功能块的定义

在程序分析领域, 经常使用的程序分析单位是基本块, 但基本块的划分是机械的、硬性的 (如算法 12.3), 就代码完成的功能而言基本上没有任何意义, 因为它不能给出一个具有明确意义的功能描述。

功能块是针对特定程序点感兴趣变量的最小相关语句序列, 它可包含一至多个基本块或由一至多个基本块和一至几个基本块的部分组成, 拥有一个入口和一到多个出口。功能块是基于描述代码功能而提出的程序划分概念, 更有利于对代码的分析。

由功能块组成的程序控制流图定义为: 节点表示功能块, 边表示功能块间的控制流。当功能块中不包含转移指令时用如图 12-19a 所示节点表示; 当功能块中包含转移指令时用如图 12-19b 所示节点表示, 从节点发出的边序列 1, 2, ..., n 表示功能块包含的转移指令对应的控制流, 且边在序列中的先后顺序对应于转移指令在功能块中所处的先后位置, 如边 1 对应的转移指令在功能块中的位置要先于边 2 对应的转移指令, 处于节点正下方的边对应功能块结尾处的控制流。

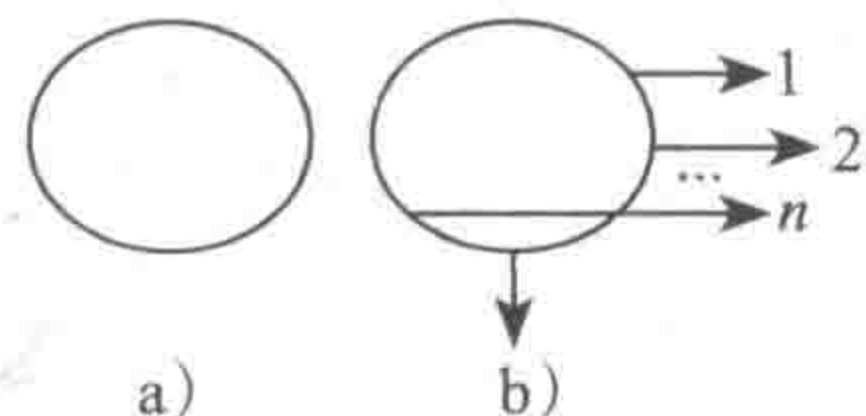


图 12-19 功能块的节点表示

#### 2. 功能块的划分算法

介绍功能块划分算法之前, 先给出四个概念。



**定义 12.1** 兴趣点是构建功能块所围绕的指令。对切片图 12-15 和图 12-18 而言兴趣点分别为指令 L054 和指令 L035。

**定义 12.2** 功能块控制指令是控制功能块具体执行的指令，此指令与功能块的输入数据相关。对兴趣点 L054 而言，功能块控制指令为比较指令 L043；对兴趣点 L035 而言，功能块控制指令为比较指令 L028。

**定义 12.3** 控制变量是功能块控制指令中起关键作用的变量。例如：功能块控制指令 L028 对应的控制变量是 r8。

**定义 12.4** 功能块输入变量是提供功能块输入数据的变量。

---

#### 算法 12.5 构建给定兴趣点的功能块

输入：某程序、兴趣点。

输出：关于输入兴趣点的功能块，功能块输入变量及输入数据取值范围。

Begin

1. 首先根据兴趣点确定功能块控制指令。
2. 从兴趣点开始对感兴趣的变量进行切片，直到与特殊寄存器（如 r0、r1、r12 等）相关的简单定值指令（如 `mov r35=r12`、`mov r1=r32`、`mov r32=r1`）。
3. 对步骤 2 获得的切片统计与特殊寄存器相关的简单定值指令，对这些指令进行删除处理。
4. 步骤 3 处理后的切片如果不包含功能块控制指令，则转 7。
5. 针对不在兴趣点到功能块控制指令指令序列中的其他剩余切片指令，对仅与控制变量相关的指令实施删除处理。
6. 根据功能块控制指令提取控制变量的取值范围，并把控制变量定义为功能块输入变量，并转 8。
7. 以功能块控制指令为基点，向前向后检查有无与控制变量所存数据（也可理解为内存单元）相关的等价指令序列（如图 12-13 中的指令序列 L041 ~ L042 和指令序列 L045 ~ L046）。如果存在等价的指令序列，根据功能块控制指令提取控制变量的取值范围，从切片头部删除此指令序列并把控制变量的取值范围赋给将删除指令序列中最后被定值的变量（如把指令 L043 中控制变量 r16 的取值范围赋给指令 L046 中被定值的变量 r14），并把此变量定义为功能块输入变量；否则，定义功能块输入变量为 NULL。
8. 剩余的切片指令构成功能块，退出。

End

---

图 12-13 所示程序指令序列中基本块 B8 的部分指令序列 L047~L054 构成一个功能块，完成计算指令 L054 中寄存器 b6 值的功能，见图 12-20a。分析图 12-18 知：指令 L035 处寄存器 b6 的值实质上由寄存器 r8 的值控制，而寄存器 r8 在指令 L035 之前最后出现于指令 L028 中且指令 L028 和 L029 共同组成一个条件跳转模块，另外，指令序列 L030~L035 数



据依赖于指令 L026 和 L027。因此，图 12-16 中基本块 B5 的一部分和基本块 B6，即指令序列 L026 ~ L035 组成一个关于指令 L035 处寄存器 b6 的功能块，完成计算指令 L035 中寄存器 b6 值的功能，见图 12-20b。上述分析得到的功能块与使用算法 12.5 构建出的功能块相同。

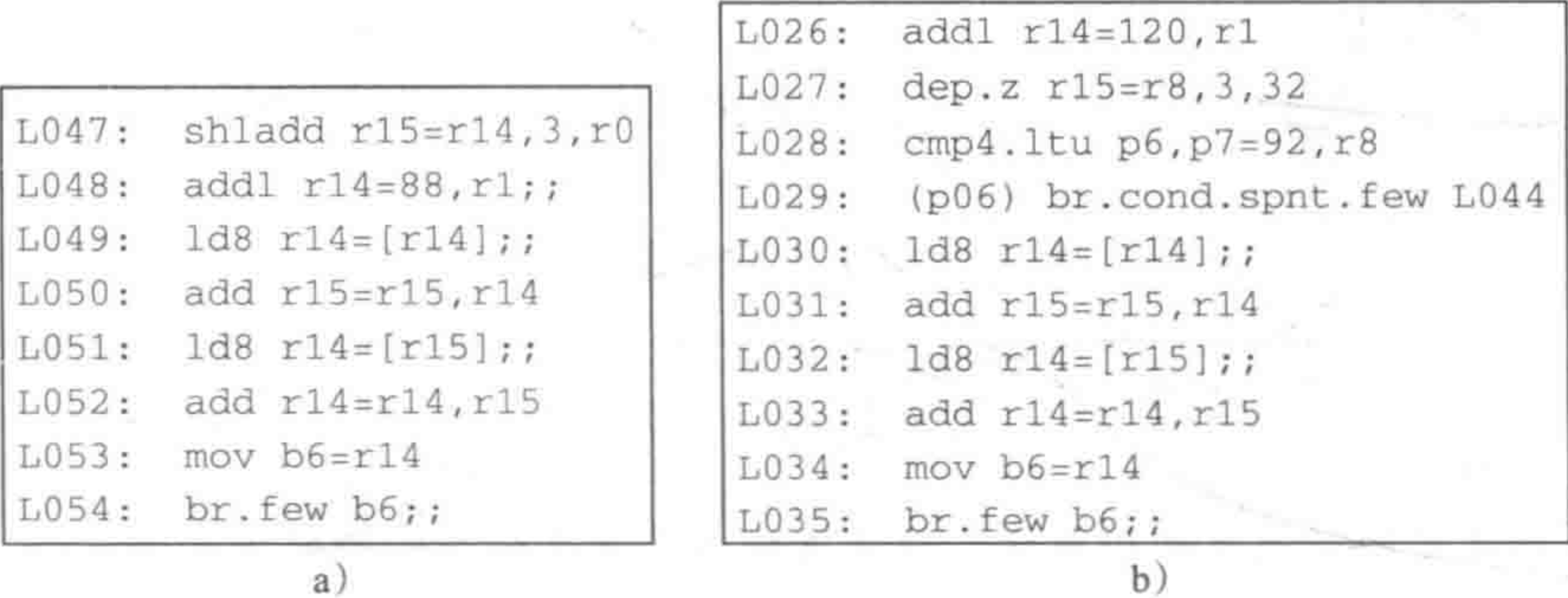


图 12-20 功能块

图 12-20a 中的功能块输入变量为寄存器 r14，其对应的数据取值范围为 [0, 92]；图 12-20b 中功能块的输入变量为寄存器 r8，其对应的数据取值范围为 [0, 92]。对两功能块而言输出数据都是寄存器 b6 的值。

12.2.4 针对功能块的验证

在功能块首部（第一条指令执行之前）恰当地设置输入变量的值，然后执行功能块，重复上述操作直至输入数据被全部遍历，即可获得功能块的完备输出数据。图 12-20a 中功能块的输出数据见图 12-21a，图 12-20b 中功能块的输出数据见图 12-21b。

引入功能块并调试获得间接跳转指令的目标地址集合（如图 12-21 所示）后，利用取得的目标地址信息指导反汇编器重新反汇编可执行程序 b.out 和 a.out，可分别得到图 12-22 和图 12-23 所示的汇编代码。

图 12-22 和图 12-23 对应的完整控制流图分别见图 12-24a 和图 12-24b，它们分别覆盖了各自程序的所有指令代码。

包含非 switch 结构间接跳转的某一程序片段如图 12-25 所示，针对兴趣点 Ln+11 使用算法 12.5 处理此程序片段，得到关于兴趣点 Ln+11 处寄存器 b6 的功能块，即指令序列 Ln+8 ~ Ln+11，功能块输入变量为 NULL。分析程序可知，此间接跳转的目标地址由指令序列 Ln+4 ~ Ln+7 决定，按能够到达此功能块的程序路径调试执行即可获得间接跳转指令 Ln+11 的目标地址。

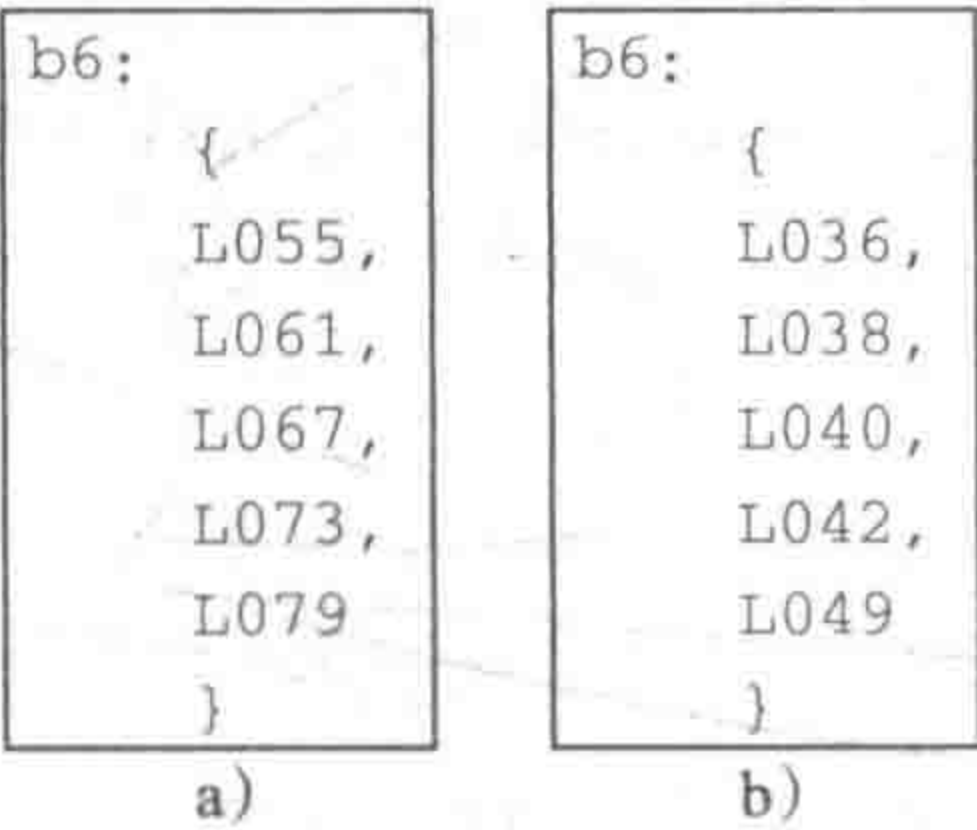


图 12-21 功能块输出数据



使用 GCC 编译器编译程序 switch.c, 除使用 -O0 和 -O2 优化选项编译生成的代码外, 针对采用 -O1、-O3 和 -O4 优化选项编译生成的代码, 使用 12.2.3 节介绍的基于功能块的分析方法进行分析, 都能得到相应程序的完整控制流图。

应用功能块构建算法 12.5 处理四个测试集: 基准测试集 SPEC2006、IEEE 浮点测试软件、Fortran78 Test Suite 测试集和自编测试集的 1800 个测试用例, 间接转移指令对应的功能块都能被成功构建。

L001: alloc r39=ar.pfs,14,8,0		L036: adds r34=1,r34	B8
L002: addl r40=80,r1		L037: br.few L017	
L003: mov r32=r1		L038: adds r35=1,r35	B9
L004: mov r38=b0;;		L039: br.few L017	
L005: ld8 r40=[r40]	B1	L040: adds r36=1,r36	B10
L006: mov r34=r0		L041: br.few L017	
L007: mov r35=r0		L042: adds r37=1,r37	B11
L008: mov r36=r0		L043: br.few L017	
L009: mov r37=r0		L044: addl r40=96,r1	
L010: br.call.sptk.many <printf>;		L045: mov r32=r1;;	B12
L011: mov r33=r0		L046: ld8 r40=[r40]	
L012: mov r1=r32;;		L047: br.call.sptk.many <printf>;	
L013: addl r40=88,r1;;	B2	L048: br.few L016	B13
L014: ld8 r40=[r40]		L049: adds r33=1,r33	B14
L015: br.call.sptk.many <printf>;		L050: br.few L017	
L016: mov r1=r32	B3	L051: addl r40=104,r1	
L017: addl r15=24,r1		L052: ld8 r40=[r40]	B15
L018: mov r32=r1;;		L053: br.call.sptk.many <printf>;	
L019: ld8 r14=[r15]	B4	L054: mov r1=r32	
L020: ld8 r40=[r14]		L055: mov r41=r34	
L021: br.call.sptk.many <getchar>;		L056: mov r42=r35	
L022: cmp4.eq p7,p6=-1,r8		L057: mov r43=r36	
L023: mov r1=r32	B5	L058: mov r44=r37	B16
L024: (p07) br.cond.spnt.few L051		L059: mov r45=r33;;	
L025: adds r8=-10,r8;;	B6	L060: addl r40=112,r1	
L026: addl r14=120,r1		L061: ld8 r40=[r40]	
L027: dep.z r15=r8,3,32		L062: br.call.sptk.many <printf>;	
L028: cmp4.ltu p6,p7=92,r8		L063: mov r1=r32	
L029: (p06) br.cond.spnt.few L044		L064: mov r8=r0	
L030: ld8 r14=[r14];;		L065: mov.i ar.pfs=r39	B17
L031: add r15=r15,r14	B7	L066: mov b0=r38	
L032: ld8 r14=[r15];;		L067: br.ret.sptk.many b0;;	
L033: add r14=r14,r15			
L034: mov b6=r14			
L035: br.few b6;;			

图 12-22 反汇编 b.out 生成的代码及功能块划分



L001: alloc r34=ar.pfs,10,4,0		L045: adds r15=8,r35;;		L089: mov r1=r32	B16
L002: mov r35=r12		L046: ld4 r14=[r15]	B8	L090: br.few L025	
L003: adds r12=-32,r12		L047: shladd r15=r14,3,r0		L091: addl r14=104,r1;;	
L004: mov r33=b0;;		L048: addl r14=88,r1;;		L092: ld8 r36=[r14]	B17
L005: adds r14=-12,r35;;		L049: ld8 r14=[r14];;		L093: mov r32=r1	
L006: st4 [r14]=r0		L050: add r15=r15,r14	B9	L094: br.call.sptk.many<printf>;	
L007: adds r14=-8,r35;;		L051: ld8 r14=[r15];;		L095: mov r1=r32	
L008: st4 [r14]=r0		L052: add r14=r14,r15		L096: adds r15=-12,r35	
L009: adds r14=-4,r35;;		L053: mov b6=r14		L097: adds r16=-8,r35	
L010: st4 [r14]=r0	B1	L054: br.few b6;;		L098: adds r17=-4,r35	
L011: mov r14=r35;;		L055: adds r15=-12,r35		L099: mov r18=r35	
L012: st4 [r14]=r0		L056: adds r14=-12,r35	B10	L100: adds r19=4,r35;;	
L013: adds r14=4,r35;;		L057: ld4 r14=[r14];;		L101: addl r14=112,r1;;	B18
L014: st4 [r14]=r0		L058: ldds r14=1,r14		L102: ld8 r36=[r14]	
L015: addl r14=72,r1;;		L059: st4[r15]=r14		L103: ld4 r37=[r15]	
L016: ld8 r36=[r14]		L060: brfew L025		L104: ld4 r38=[r16]	
L017: mov r32=r1		L061: adds r15=-8,r35	B11	L105: ld4 r39=[r17]	
L018: br.call.sptk.many <printf>;		L062: adds r14=-8,r35;;		L106: ld4 r40=[r18]	
L019: mov r1=r32;;	B2	L063: ld4 r14=[r14];;		L107: ld4 r41=[r19]	
L020: addl r14=80,r1		L064: adds r14=1,r14		L108: mov r32=r1	
L021: ld8 r36=[r14]		L065: st4[r15]=r14		L109: br.call.sptk.many <printf>;	
L022: mov r32=r1	B3	L066: br.few L025		L110: mov r1=r32	
L023: br.call.sptk.many <printf>;	B4	L067: adds r15=-4,r35	B12	L111: mov r14=r0;;	
L024: mov r1=r32		L068: adds r14=-4,r35		L112: mov r8=r14	
L025: mov r32=r1	B5	L069: ld4 r14=[r14];;		L113: mov.i ar.pfs=r34	B19
L026: br.call.sptk.many <getchar>;		L070: adds r14=1,r14		L114: mov b0=r33	
L027: mov r1=r32		L071: st4[r15]=r14		L115: mov r12=r35	
L028: mov r14=r8		L072: brfew L025		L116: br.ret.sptk.many b0;;	
L029: adds r15=-16,r35;;	B6	L073: mov r15=r35	B13		
L030: st4 [r15]=r14		L074: mov r14=r35;;			
L031: adds r16=-16,r35;;		L075: ld4r14=[r14];;			
L032: ld4 r14=[r16];;	B7	L076: adds r14=1,r14	B14		
L033: cmp4.eq p7,p6=-1,r14		L077: st4[r15]=r14			
L034: (p06) br.cond.dptk.few L036		L078: br.few L025	B15		
L035: br.few L091		L079: adds r15=4,r35			
L036: adds r15=-16,r35;;		L080: adds r14=4,r35			
L037: ld4r14=[r15]		L081: ld4 r14=[r14];;			
L038: adds r15=-10,r14		L082: adds r14=1,r14			
L039: adds r16=8,r35		L083: st4[r15]=r14			
L040: st4[r16]=r15		L084: br.few L025			
L041: adds r16=8,r35		L085: addl r14=96,r1;;			
L042: ld4r16=[r16];;		L086: ld8 r36=[r14]			
L043: cmp4.ltu p6,p7=92,r16		L087: mov r32=r1			
L044: (p06)br.cond.dptk.few L085		L088: br.call.sptk.many <printf>;			

图 12-23 反汇编 a.out 生成的代码及功能块划分

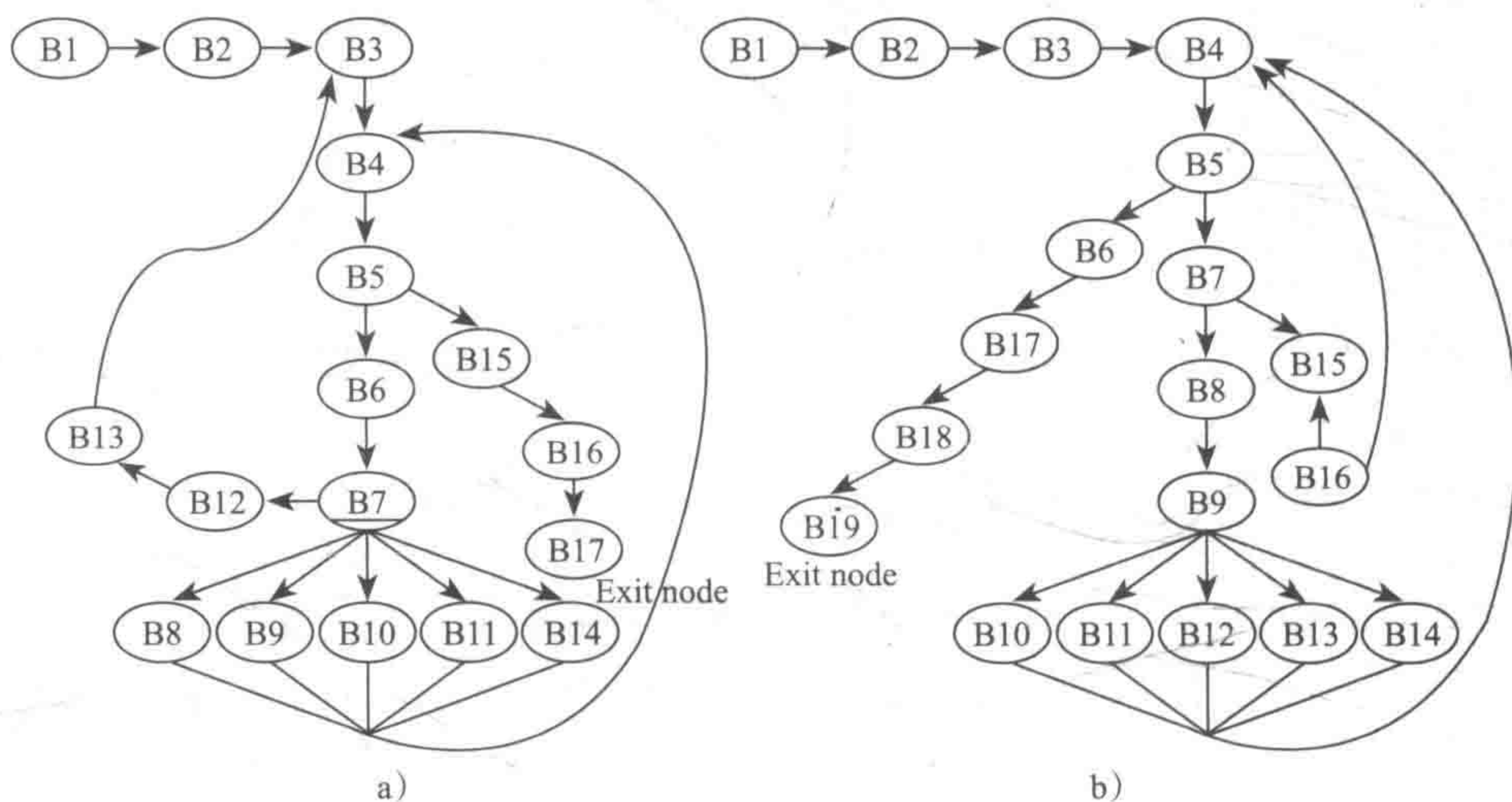


图 12-24 完整控制流图



```

Ln:      mov r35=r12;;
...
Ln:      adds r14=-92,r35;;
Ln+1:    ld4 r14=[r14];;
Ln+2:    cmp4.eq p6,p7=0,r14
Ln+3:    (p06) br.cond.dptk.few Ln+4
Ln+4:    adds r15=-80,r35
Ln+5:    addl r14=80,r1;;
Ln+6:    ld8 r14=[r14];;
Ln+7:    st8 [r15]=r14
Ln+8:    adds r14=-80,r35;;
Ln+9:    ld8 r14=[r14];;
Ln+10:   mov b6=r14
Ln+11:   br.few b6;;

```

图 12-25 非 switch 结构间接跳转对应的程序片段

## 12.3 基于功能块的间接转移指令目标地址的确定

由于基于基本块的程序分析不能很好地解决间接转移指令目标地址确定问题，12.2.3 节提出了基于功能块的程序分析方法。功能块被成功划分后，针对如何利用获得的功能块、功能块输入变量和输入数据取值范围提取程序信息的过程前文并没有给出详细的介绍，本章将给出基于功能块解决间接转移指令目标地址确定问题的具体方法。

### 12.3.1 程序控制流图构建方法中存在的问题

如何从二进制程序构建完整的程序控制流图，是一个备受计算机众多领域关注的问题，如逆向工程、恶意代码检测、二进制探测、程序轮廓信息提取和程序验证等。为了构建程序的完整控制流图，需要解决间接转移指令目标地址确定问题。

EEL (Executable Editing Library) 对二进制程序进行了控制流重建研究，其目的是为了在缺失源程序的情况下仍然可以编辑程序。当间接转移指令的目标地址不能静态获得时，EEL 使用切片技术确定目标地址，并在程序中插入一些代码实现对目标地址的转换功能，以保证程序重新编译后可以正确执行。然而，哪些间接转移指令的目标地址不能静态获得并没有被指出。虽然针对某些机器（如 SPARC）使用 EEL 比较精确地重建了程序的控制流图，但它并不能很好地应用于复杂的体系结构和先进的编译技术。

反编译器 dcc 最先使用扩展型寄存器复制传播技术，它针对 Intel 80286 体系结构实现从可执行程序到 C 程序的恢复工作。翻译器 Asm2c 针对 SPARC 体系结构完成汇编程序到 C 程序的翻译工作，它使用程序切片和扩展型寄存器复制传播技术计算获得需要的控制流图。同 EEL 相比，Asm2c 和 dcc 都不能应用于其他体系结构，且都没有对重建的控制流图的完整性进行特殊处理。



Theiling 则假设任何体系结构使用的跳转地址都可以直接通过指令计算获得, 并且它拒绝使用间接跳转指令。按其介绍的方法处理间接跳转时, 他建议可以使用未知的代码来重建一个近似的控制流图。

Cifuentes 和 Emmerik 提出基于跳转表获得跳转地址的分析方法, 即使用反向切片和复制传播技术获得间接跳转目标地址对应的表达式, 然后将获得的表达式和 switch 语句对应的模板进行匹配, 最终依据跳转表提取间接跳转对应的目标地址。此方法完全依赖于编译器的版本, 通用性差, 并且程序中的间接跳转并不完全依赖于跳转表 (如在目标地址运行时才可计算获得的 goto 语句、涉及指针算术运算的指针函数调用、汇编级手工嵌入的间接跳转)。

Kästner 和 Wilhelm 介绍了一种自顶向下的构建策略, 将二进制程序构建成过程和基本块, 要求过程间不能存在重叠的代码区域、过程间或者过程内不能存在数据、间接跳转可能的目标地址都有对应的显式标号。然而, 即使各个过程的控制流图是完全不重叠的, 编译器生成的过程一般都共用入口和出口点, 因此, 他们提出的自顶向下构建策略因为限制条件太多而不具有通用性。

在程序验证领域, Myreen 通过将程序转换为尾递归函数, 从而提出基于语义的控制流重建技术。

Kinder 和 Veith 给出了在构建程序控制流图时, 解决间接跳转目标问题的分析框架。他们提出利用一种工作链表算法来动态地扩展程序的控制流, 此方法可以得到程序精确近似的控制流图。然而, 由于此框架只适用于过程内部分析, 需要不断为其提供新的过程, 因此, 它无法处理递归过程对应的汇编代码。

Flexeder 和 Mihaila 使用 Kinder 和 Veith 提出的解决间接跳转的分析框架, 并对它进行扩展使其可以处理过程调用。但他们只跟踪了寄存器和内存单元的值, 而完全忽略了堆的使用, 因此, 针对某些间接调用指令时将无法处理。

数据流分析是针对给定的程序静态计算程序变量信息的技术, 它可以辅助解决间接跳转问题, 但由于数据流分析依赖于精确的控制流图, 然而, 在间接跳转的目标确定问题未解决之前不可能拥有程序的精确控制流图, 解决间接跳转遇到了自相矛盾的境况, 就如“鸡和蛋”的问题。之前的研究指出, 数据流分析可以加大反汇编覆盖的代码范围, 但是针对未解决的控制流后继未能给出肯定的解决方法。

### 12.3.2 无法处理的代码

针对间接转移指令的目标地址确定, 使用程序切片、复制传播和数据流分析等传统技术只能解决部分问题。本节给出一个使用上述分析技术无法处理的代码示例。

测试集 Fortran78 Test Suite 中程序 FM317.f 经编译器 f77 使用优化选项 -O2 编译生成的代码片段如图 12-26 所示, 它含有一条间接调用指令 `br.call.sptk.many b0=b6`。

为了获得间接调用指令的目标地址, 即寄存器 b6 的值, 使用切片分析技术处理上述代



码片段得到相应的切片 (400000000000565c, b6), 见图 12-27。

```

4000000000005630 <ff324_>:
4000000000005630:      [MMI]      alloc r36=ar.pfs,6,5,0
4000000000005636:                      ld8 r14=[r32],8
400000000000563c:                      mov r35=b0
4000000000005640:      [MFI]      mov r34=r1
4000000000005646:                      nop.f 0x0
400000000000564c:                      mov r37=r33;;
4000000000005650:      [MIB]      ld8 r1=[r32]
4000000000005656:                      mov b6=r14
400000000000565c:                      br.call.sptk.many b0=b6;;

```

图 12-26 特殊间接调用对应的程序片段

```

4000000000005636:  ld8 r14=[r32],8
4000000000005656:  mov b6=r14
400000000000565c:  br.call.sptk.many b0=b6;;

```

图 12-27 切片 (400000000000565c, b6) 对应的指令

使用表达式复制传播技术处理切片 (400000000000565c, b6) 代码, 得到  $b6 = m[r32]$ , 其中  $m[]$  表示取内存操作。获得结果  $b6 = m[r32]$  是因为指令 `ld8 r14=[r32], 8` 属于 `imm_base_update_form` 型取内存操作, 偏移量 8 只有等到取内存操作完成后才会起作用, 用它来更新刚才使用的内存地址, 使得  $r32 = r32 + 8$ 。

解决间接转移指令目标地址确定问题是为了完备的代码挖掘, 即在指令解码阶段使用间接转移指令的目标地址来定位后继的指令位置以继续解码。表达式  $b6 = m[r32]$  中的  $r32$  是当前过程的形参, 它的值需要追溯到此过程对应的调用点来获得, 然而这是过程恢复和过程间分析完成的工作, 在指令解码阶段绝对不可能获得上述需要的信息。另外, 由表达式  $b6 = m[r32]$  可知寄存器  $b6$  的值要从内存的某个存储单元中获得, 而在程序未执行的前提下从内存中取出的数据的有效性是无法保证的, 从而使得对间接转移指令目标地址的确定更加困难。

### 12.3.3 程序执行路径的逆向构造

基于制导信息提取系统 CGIPS (在《编译与反编译技术》一书的 11.3.8 节中讲解 ITA 静态反编译框架的扩展 ITA-E 时提到了 CGIPS, 即 Control and Guide Information Picking System), 我们在这个实例中采取直接执行间接转移指令的方式, 来获得间接转移指令对应的所有目标地址。但一个程序可能包含多个间接转移指令, 且随机分布, 因此获得间接转移指令的目标地址需要遍历整个程序, 这一工作非常繁琐且时空消耗巨大, 因为它需要创建很多快照点。

为了有效降低获取间接转移指令目标地址的时空消耗和复杂度, 需要获取一条与具体间接转移指令直接相关的程序执行路径, 然后, 严格按照此路径执行程序取得间接转移指



令的目标地址。

要获取间接转移指令对应的执行路径，一个显然的方法是根据程序的控制流图来选择一条路径，但由于间接跳转和间接调用的存在致使此时程序的控制流图不完整，从而导致针对某些间接转移指令无法获得相应的执行路径。因此，我们提出执行路径逆向构造技术来获取间接转移指令对应的执行路径。

### 1. 间接点的提取

继续论述之前需要明确一点：引入执行路径逆向构造技术的目的是解决间接转移指令目标地址的确定问题，逆向构造执行路径前需要确切知道待处理的间接转移指令所处的位置，及逆向构造将涉及的汇编指令。使用线性扫描反汇编工具 objdump 处理待翻译程序可获得满足上述要求的汇编代码。

编译器编译程序时，基本上都把可执行的代码组织到代码段 .text 的合适位置处。不同编译器使用的总的组织策略是相同的，如编译器 GCC、icc 和 f77 对 IA64 可执行程序代码段的组织都符合如图 12-28 所示模板：例程 \_start 是程序执行的入口函数，它的功能是做一些启动设置并调用库函数 \_\_libc\_start\_main；例程 gmon\_initializer 的功能是初始化执行剖析系统；例程 \_\_libc\_csu\_init 调用共享库函数 \_init() 进行初始化；例程 \_\_libc\_csu\_fini 调用共享库函数 \_fini() 进行收尾处理。

真正与高级语言程序相对应的执行代码都被放置在例程 gmon\_initializer 和 \_\_libc\_csu\_init 之间。然而，不同编译器在具体处理时又各不相同，如编译器 GCC、icc（10.0 和 11.1 版）和 f77 都把真正的执行代码置于例程 \_\_do\_jv\_register\_classes 和 \_\_libc\_csu\_init 之间；而 8.0 版本的 icc 编译器却把真正的执行代码置于例程 \_Z11\_\_icrt\_initv 和 \_\_libc\_csu\_init 之间，且从例程 gmon\_initializer 到 \_Z11\_\_icrt\_initv 之间的代码组织完全不同于上述几个编译器编译对生成代码的组织情况。

```
<_start>:
...
<gmon_initializer>:
...
...
<main>:
...
...
<__libc_csu_init>:
...
<__libc_csu_fini>
...
...
```

图 12-28 .text 段组织模式

经统计发现，IA64 可执行程序使用如图 12-29 所示的间接转移指令模板进行间接跳转或间接调用。其中，b\* 表示转移寄存器 b5、b6、b7 三者中的任意一个。

```
br.few b*;
br.cond.dptk.many b*;
br.cond.dptk.few b*;
```

a)

```
br.call.dptk.few b0=b*;
br.call.dptk.many b0=b*;
br.call.sptk.few b0=b*;
br.call.sptk.many b0=b*;
```

b)

图 12-29 间接转移指令模板

根据实际可执行代码的存放位置和间接转移指令模板，就可定位那些真正阻碍静态二进制翻译的间接转移指令的位置。下面给出间接点提取算法。



**算法 12.6 提取可执行程序中间接转移指令所处的位置**

输入：使用 objdump 处理可执行程序获得的汇编代码。

输出：间接转移指令及其对应的指令地址。

Begin

1. 在汇编代码中搜索字符串 “\_\_do\_jv\_register\_classes”，如果存在此字符串则定义字符串变量  $start = \text{“__do\_jv\_register\_classes”}$ ，否则定义字符串变量  $start = \text{“_Z11\_icrt\_initv”}$ 。另外，定义字符串变量  $end = \text{“__libc\_csu\_init”}$ 。

2. 在汇编代码中查找字符串变量  $start$  和  $end$  各自对应例程的首地址  $Addr_{start}$  和  $Addr_{end}$ 。

3. 针对处于地址  $Addr_{start}$  和  $Addr_{end}$  之间的所有指令，将它及其对应的指令地址按原有先后顺序存储到指令表  $Table_{inst}$  中。

4. 使用图 12-29 给出的间接转移指令模板对  $Table_{inst}$  中的所有指令进行匹配搜索，对搜索到的间接转移指令，将此指令和指令地址作为一个整体记录到间接转移指令表  $LIST_{ITI}$  中。

End

## 2. 逆向构造相关问题与分析

程序执行路径逆向构造的主要思想是：从程序中感兴趣的一点开始，沿着程序控制流的反方向构造一条从兴趣点到程序主入口点的路径，然后将此路径反序得到从程序主入口点到兴趣点的执行路径。为方便介绍，定义 5 个与控制流相关的概念。

**定义 12.5** 控制流源头是控制流流出的指令。

**定义 12.6** 控制流目的是控制流直接到达的指令。

**定义 12.7** 控制流代码是体现控制流转移的代码片段。

**定义 12.8** 控制流分支点是控制控制流走向的指令，如 IA64 的比较指令。

**定义 12.9** 母体比较指令——因为条件跳转指令受谓词寄存器的控制，而谓词寄存器的值又由比较指令决定，因此，定义控制条件跳转指令执行与否的比较指令为此条件跳转指令的母体比较指令。

给出程序执行路径逆向构造算法之前，先介绍与其相关的工作。

### (1) 间接转移指令间的关系

程序可能不仅包含间接跳转，还包含间接调用。同一条执行路径上相邻间接转移指令间存在如下四种关系：间接跳转 A 先于间接跳转 B，见图 12-30a；间接跳转 C 先于间接调用 D，见图 12-30b；间接跳转包含在间接调用的目标函数体中，见图 12-30c；间接调用 J 包含在间接调用 I 的目标函数体中，见图 12-30d。

针对图 12-30a 所示关系，在成功逆向构造间接跳转 B 对应的执行路径前必须先确定间接跳转 A 的目标地址，因为逆向构造间接跳转 B 对应执行路径到达指令 x 时，可能找不到以指令 x 为控制流目的的控制流源头，而此时以指令 x 为控制流目的的控制流源头很可能就是间接跳转 A。针对图 12-30c 所示关系，在逆向构造间接跳转 G 对应的执行路径之前必



须先确定间接调用 F 的目标地址，当逆向构造间接跳转 G 对应执行路径到达过程 F 的首部时，可能因为没有对过程 F 的直接调用指令导致找不到以过程 F 为控制流目的的控制流源头，而此时以过程 F 为控制流目的的控制流源头可能恰好就是过程 E 中的间接调用指令。针对图 12-30b、d 所示关系的分析同上，不再赘述。

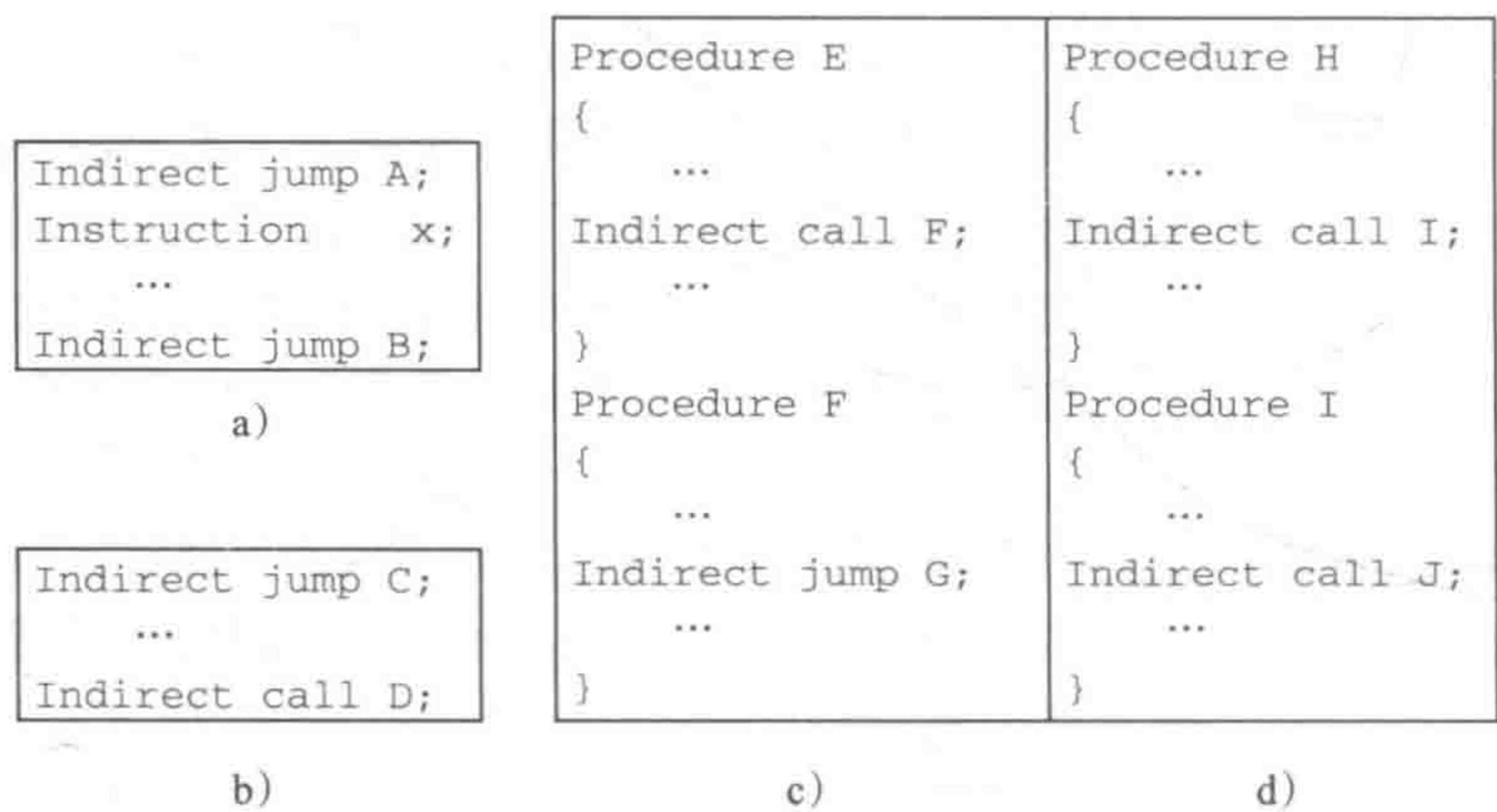


图 12-30 相邻间接转移指令间的关系

由上述分析知，为提取间接转移指令的目标地址需要设计一个间接转移指令制约关系表 ITIRT，当逆向构造间接转移指令 n 对应的执行路径而受制于未处理的间接转移指令 m 时，就把这一制约关系记录到 ITIRT 中，并停止针对间接转移指令 n 进行的执行路径逆向构造，见图 12-31a。当处理待翻译二进制程序到达一定阶段时，如果间接转移指令 x 不受制于任何间接转移指令或者制约指令 x 的间接转移指令都已处理完毕（即指令 x 对应的制约链表为空），则重新为间接转移指令 x 逆向构造一条执行路径，同时在 ITIRT 中删除受制于间接转移指令 x 的制约关系，见图 12-31b。

(2) 控制结构

逆向构造执行路径之前，必须解决如何处理控制流的问题。C 语言使用的 if...else、for、while、do...while、switch、goto、break、continue 和函数调用等控制结构对应的控制流代码如图 12-32 所示，控制结构 for 和 while 对应的控制流代码相同，控制结构 goto、break 和 continue 对应的控制流代码相同。

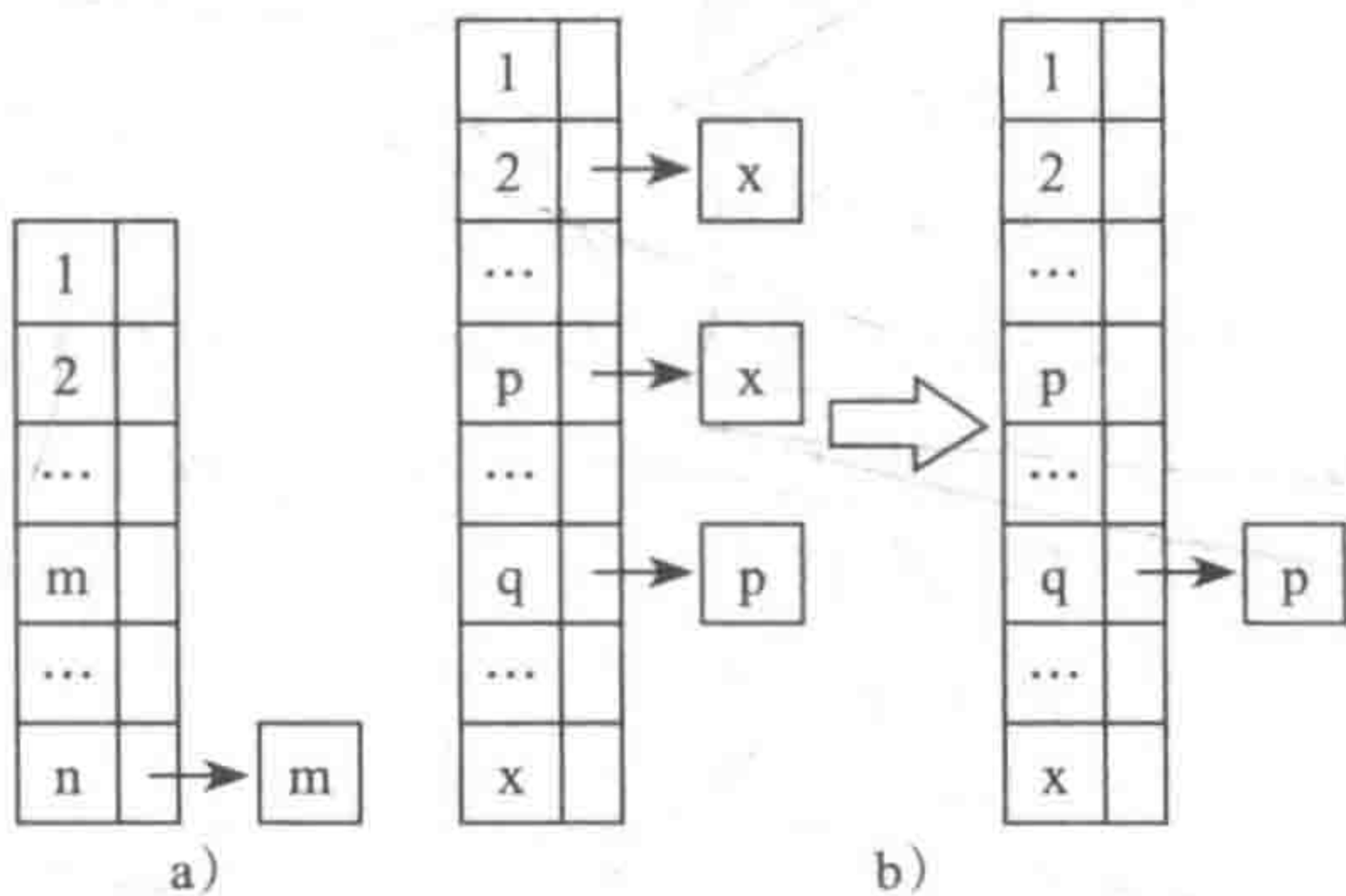


图 12-31 间接转移指令制约关系表 ITIRT

图 12-32a 所示 if...else 的控制流代码对应两个执行路径，分别为：

Path1=Lr → Lr+1 → Lr+2 → Ls+1 → Is<sub>2</sub> → Lt → Lt+1  
Path2=Lr → Lr+1 → Lr+2 → Is<sub>1</sub> → Ls → Lt → Lt+1



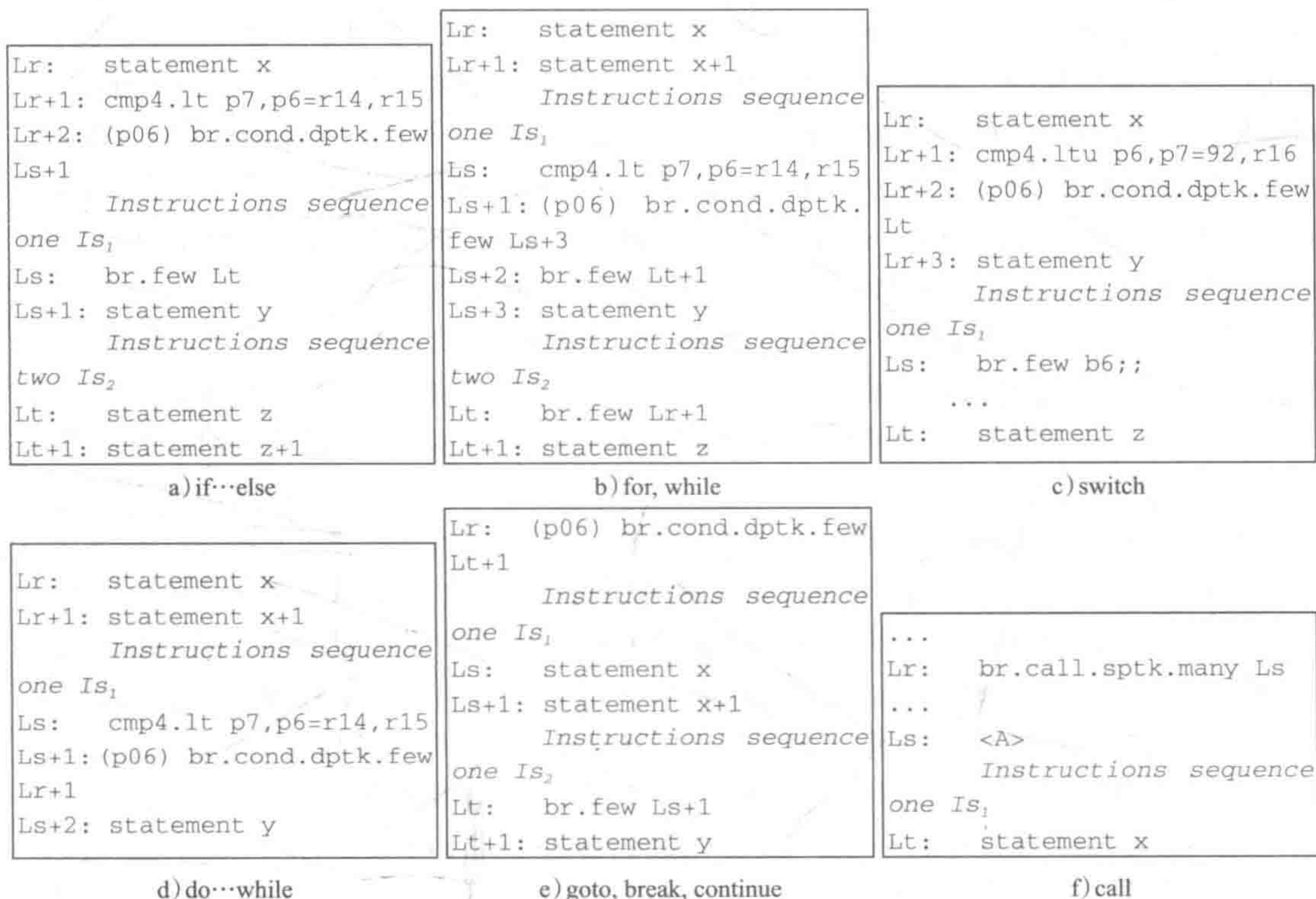


图 12-32 C 语言控制结构对应的控制流代码

其中指令 Lr+2 和 Ls 实现控制流的转移，且谓词寄存器存储的是布尔值。由于 Lr+2 受谓词寄存器 p6 的控制，当 p6 的值为真时执行指令 Lr+2，否则不执行指令 Lr+2 而执行它下面的一条指令。当 p6 取假值时，虽然由谓词寄存器 p6 控制的指令 Lr+2 不被执行，但由于需要判断 p6 值的真假，因此指令 Lr+2 仍然要存在于执行路径 Path2 中。从指令 Lt+1 开始沿着控制流的反方向依次寻找以当前指令为控制流目的的控制流源头，找到后把此指令加入到构造路径中，并设置它为当前指令重复上述操作，直至指令 Lr。针对图 12-32a 所示的控制流代码逆向构造的路径为 Path3=Lt+1 → Lt → Is<sub>2</sub> → Ls+1 → Lr+2 → Lr+1 → Lt，把逆向构造的路径 Path3 反序得到逆向构造的执行路径 Path4=Lr → Lr+1 → Lr+2 → Ls+1 → Is<sub>2</sub> → Lt → Lt+1，显然逆向构造的执行路径 Path4 等同于执行路径 Path1。

图 12-32b 所示 for 或者 while 的控制流代码对应的执行路径为 Path5=Lr → (Lr+1 → Is<sub>1</sub> → Ls → Ls+1 → Ls+3 → Is<sub>2</sub> → Lt)\* → Lr+1 → Is<sub>1</sub> → Ls → Ls+1 → Ls+2 → Lt+1，其中 (Lr+1 → Is<sub>1</sub> → Ls → Ls+1 → Ls+3 → Is<sub>2</sub> → Lt)\* 表示此部分子路径在 Path5 中可以连续重复出现多次或者一次都不出现。从指令 Lt+1 开始进行执行路径逆向构造，得到逆向构造路径 Path6=Lt+1 → Ls+2 → Ls+1 → Ls → Is<sub>1</sub> → Lr+1 → Lr，把路径 Path6 反序得到逆向构造的执行路径 Path7=Lr → Lr+1 → Is<sub>1</sub> → Ls → Ls+1 → Ls+2 → Lt+1，对比执行路径 Path5 和 Path7 可知路径 Path7 属于路径 Path5，即 Path7 ⊆ Path5。



执行路径  $\text{Path8} = \text{Lr} \rightarrow (\text{Lr}+1 \rightarrow \text{Is}_1 \rightarrow \text{Ls} \rightarrow \text{Ls}+1)^+ \rightarrow \text{Ls}+2$  对应于图 12-32d 所示的 do...while 控制流代码, 其中  $(\text{Lr}+1 \rightarrow \text{Is}_1 \rightarrow \text{Ls} \rightarrow \text{Ls}+1)^+$  表示此部分子路径在 Path8 中可以连续重复出现多次, 且必须出现一次。从指令  $\text{Ls}+2$  开始进行执行路径逆向构造, 得到逆向构造路径  $\text{Path9} = \text{Ls}+2 \rightarrow \text{Ls}+1 \rightarrow \text{Ls} \rightarrow \text{Is}_1 \rightarrow \text{Lr}+1 \rightarrow \text{Lr}$ , 把路径 Path9 反序得到逆向构造的执行路径  $\text{Path10} = \text{Lr} \rightarrow \text{Lr}+1 \rightarrow \text{Is}_1 \rightarrow \text{Ls} \rightarrow \text{Ls}+1 \rightarrow \text{Ls}+2$ , 对比执行路径 Path8 和 Path10 可知路径 Path10 属于路径 Path8, 即  $\text{Path10} \subseteq \text{Path8}$ 。

图 12-32e 给出了控制结构 goto、break 和 continue 对应的控制流代码, 除控制流转移指令 Lt 外, 其他代码只是为了展示程序段的基本框架而存在。从指令  $\text{Lt}+1$  开始进行执行路径逆向构造, 得到逆向构造路径  $\text{Path11} = \text{Lt}+1 \rightarrow \text{Lr}$ , 它对应的执行路径为  $\text{Path12} = \text{Lr} \rightarrow \text{Lt}+1$ 。

图 12-32f 所示的控制流代码对应于控制结构 call, 其中语句 x 属于过程 A。当程序执行到过程调用指令 Lr 时, 则直接执行此指令进行过程调用, 并把对程序的控制转移到 Ls 处继续执行, 因此其执行路径为  $\text{Path13} = \text{Lr} \rightarrow \text{Ls} \rightarrow \text{Is}_1 \rightarrow \text{Lt}$ 。从指令 Lt 开始进行执行路径逆向构造, 当逆向构造路径到达过程 A 的首指令 Ls 时, 因为以指令 Ls 为控制流目的的控制流源头可能有多, 即对过程 A 的调用点可能存在多个, 因此继续逆向构造执行路径之前需要从多个控制流源头中选择一个, 但针对图 12-32f 所示代码进行逆向构造得到的执行路径为  $\text{Path14} = \text{Lr} \rightarrow \text{Ls} \rightarrow \text{Is}_1 \rightarrow \text{Lt}$ , Path14 等同于执行路径 Path13。

因为图 12-32c 所示 switch 控制流代码含有本文所致力解决的间接跳转指令, 所以对它的讨论安排在本小节的最后。针对图 12-32c 给出的控制流代码, 间接跳转指令 Ls 对应的执行路径为  $\text{Path15} = \text{Lr} \rightarrow \text{Lr}+1 \rightarrow \text{Lr}+2 \rightarrow \text{Lr}+3 \rightarrow \text{Is}_1 \rightarrow \text{Ls}$ , 而从指令 Ls 开始进行执行路径逆向构造得到的执行路径为  $\text{Path16} = \text{Lr} \rightarrow \text{Lr}+1 \rightarrow \text{Lr}+2 \rightarrow \text{Lr}+3 \rightarrow \text{Is}_1 \rightarrow \text{Ls}$ , 显然 Path16 等同于 Path15。

由上述分析知, 针对 C 语言所使用的控制流代码进行执行路径逆向构造所获得的执行路径是真实的程序执行路径。

### (3) 条件冗余

多路径分析的作用之一在于确定某条语句或若干条语句的集合 (称为程序检测点集合) 在不同的执行路径下是否满足一定的需求, 如判断某条语句是否存在漏洞或者是否存在恶意行为等。此时在程序分析过程中, 部分分支条件的取值并不会影响程序检测点的判断条件, 因此产生了条件冗余。

基于功能块的描述及对 SPEC2006 基准测试集的程序实例的分析, 可知功能块是一个功能相对完整的代码片段, 它同程序中其他代码的联系很少, 针对功能块而逆向构造的执行路径, 其上的大部分分支条件的取值都不会影响功能块的执行, 因此这些分支条件对功能块而言是冗余的。

下面对前面给出的执行路径进行分析。图 12-32a 所示控制流代码对应两条执行路径 Path1 和 Path2, 选择路径 Path1 或者 Path2 执行都可到达后面的功能块, 因此由指令  $\text{Lr}+1$



和  $Lr+2$  组成的分支条件是冗余的；对比执行路径 Path5 和逆向构造的执行路径 Path7，可知  $Path7 \subseteq Path5$  且由指令  $Ls$  和  $Ls+1$  组成的分支条件是冗余的；对比执行路径 Path8 和逆向构造的执行路径 Path10，可知  $Path10 \subseteq Path8$  且由指令  $Ls$  和  $Ls+1$  组成的分支条件同样是冗余的；使用功能块划分算法 12.5 处理图 12-32c 给出的控制流代码，可知执行路径 Path15 和逆向构造的执行路径 Path16 都属于间接跳转指令  $Ls$  对应的功能块，由于要遍历设置功能块的输入数据而重复执行功能块以获取间接跳转指令的所有目标地址，因此由指令  $Lr+1$  和  $Lr+2$  组成的分支条件不属于条件冗余集合。

针对属于条件冗余集合的分支条件只要恰当地设置控制变量的值，就能保证程序按指定的路径执行，并且不影响最终到达功能块并执行功能块。

#### (4) 路径中的环

从程序中感兴趣的某一点开始进行执行路径逆向构造，构造过程中可能会遇到路径环的问题，从而导致逆向构造陷入无穷尽的循环中，比如图 12-33 所示 SPEC2006 测试集中测试用例 h264ref 对应的部分汇编代码，对其处理就会遇到路径环问题。

以过程 A 的首指令  $Lw$  为兴趣点进行程序主入口点到  $Lw$  执行路径的逆向构造，针对图 12-33 给出的代码片段，以指令  $Lr+4$  为控制流目的的控制流源头包括指令  $Lr+2$  和指令  $Lu$ ，针对指令  $Lr+4$ ，如果选择指令  $Lr+2$  作为  $Lr+4$  的控制流源头，便可以成功逆向构造一条执行路径  $Path17=Lr \rightarrow Lr+1 \rightarrow Lr+2 \rightarrow Lr+4 \rightarrow Is_1 \rightarrow Ls \rightarrow Ls+1 \rightarrow Ls+3 \rightarrow Is_2 \rightarrow Lt \rightarrow Lw$ ，但如果每次处理指令  $Lr+4$  时都选择指令  $Lu$  作为  $Lr+4$  的控制流源头则导致路径环的生成，逆向构造的路径形如  $Path18=Lw \rightarrow (Lt \rightarrow Is_2 \rightarrow Ls+3 \rightarrow Ls+1 \rightarrow Ls \rightarrow Is_1 \rightarrow Lr+4 \rightarrow Lu \rightarrow Is_3)^+$ ，并且会无穷尽地循环下去，而期望得到的执行路径永远无法被成功逆向构造。

```

Lr:      statement x1
Lr+1:    cmp4.lt p6,p7=r15,r14
Lr+2:    (p06) br.cond.dptk.few Lr+4
Lr+3:    br.few Lv
Lr+4:    statement x2
          Instructions sequence one Is1
Ls:      cmp4.eq p6,p7=0,r14
Ls+1:    (p06) br.cond.dptk.few Ls+3
Ls+2:    br.few Lu+1
Ls+3:    statement x3
          Instructions sequence one Is2
Lt:      br.call.sptk.many Lw
          Instructions sequence one Is3
Lu:      br.few Lr+4
Lu+1:    statement x4
          Instructions sequence one Is4
Lv:      statement x5
          Instructions sequence one Is5
Lw:      <A>

```

图 12-33 h264ref 的汇编代码片段

在逆向构造过程中，当遇到路径环时则停止对当前路径的构造，并回退到最近拥有多个控制流源头的指令并重新选择它的控制流源头继续路径的逆向构造，如果没有未遍历的控制流源头可供选择则继续向上回退。除了一般指令的控制流源头为紧邻的上一条指令外，指令通常拥有多个控制流源头，如图 12-33 中的指令  $Lr+4$ ，另外在程序中对某一过程的调用点也可能存在多个，因此，逆向构造执行路径的过程中即使遇到路径环的问题，最终也能成功逆向构造一条执行路径。

解决路径环问题需要在逆向构造执行路径的过程中记录哪些指令拥有多个控制流源头，并建立一个与之对应的控制流源头链表 CFSLT，如图 12-34 所示，方便回退操作。



(5) 内存一致性

多路径分析系统需要在控制流分支点处创建当前进程的快照，并记录决定控制流走向的参数变量的值，之后可以把运行的进程恢复到先前创建的快照状态，同时改变参数变量的值从而改变程序的执行路径。然而，在控制流分支点处仅依靠改变决定控制流走向的参数变量的值，从而改变程序执行路径的做法非常不充分，因为原始的参数变量的值可能被拷贝到了其他内存位置，也可能参与了某些计算并且计算结果被程序所使用。因此，如果只改变参数变量的值，则与它的原始数值相关的数据还存在于程序的数据段中，这样可能导致程序执行非法操作或者执行不可能存在的路径。

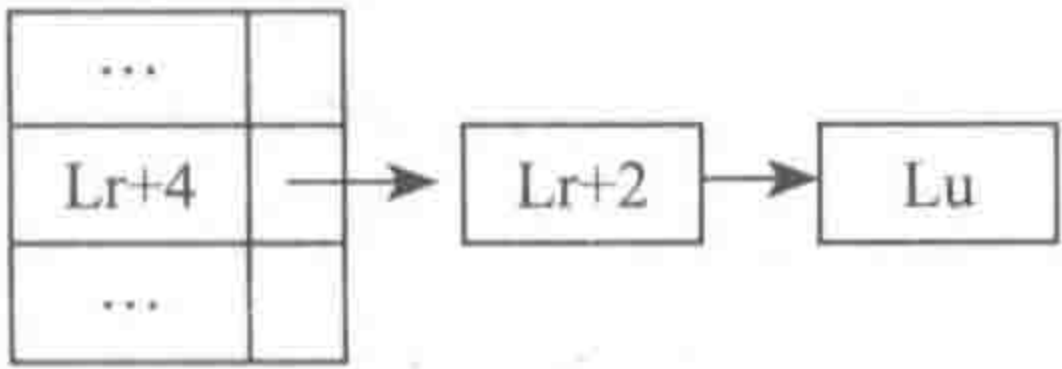


图 12-34 控制流源头链表 CFSLT

逆向构造获得的程序执行路径中同样存在大量的控制流分支点，仅靠逆向构造执行路径时记录决定控制流走向的参数变量的值，以使程序严格按照给定路径执行并获取期望信息的做法是不可行的，正确的做法应该是，无论参数变量的值何时被改写，都要更新与参数变量相关的所有数据单元，即维持内存的一致性。

保持内存一致性需要记录那些依赖于参数变量的内存位置以及相应的依赖关系。相关文献中给出了此方面的详细介绍，有兴趣的读者可以自行查找。

3. 执行路径逆向构造算法

前面介绍了程序执行路径逆向构造所面临的五个问题，同时对这些问题进行了深入分析并给出相应的解决方法，基于此，以下将给出执行路径逆向构造算法。

控制流分支点（即条件跳转指令的母体比较指令）决定程序的执行路径，因此逆向构造执行路径的过程中需要创建一个控制流分支点参数变量取值表 CFIVT，见表 12-2，按指令出现的先后顺序记录控制流分支点处的参数变量取值情况。其中，第一列记录控制流分支点指令的地址，第二列记录参数变量，第三列给出此变量可能的取值范围，第四列记录此变量具体的取值。另外，比较指令的表示形式为 `cmp4.crel p1,p2 = a,b`，根据 `crel` 指代内容的不同比较指令可分为如表 12-3 所示类型，针对比较指令的不同类型其参数变量 `b` 的取值范围也不相同。例如针对比较指令 `cmp4.lt p6,p7 = 56,r14` 可知，参数变量 `r14` 的取值范围为 `[-128,127]`，如果 `r14` 取值于 `[-128,56]` 则比较式子 `56<r14` 不成立，导致 `p6=0` 且 `p7=1`，而如果 `r14` 取值于 `[57,127]` 则比较式子 `56<r14` 成立，导致 `p6=1` 且 `p7=0`。

表 12-2 控制流分支点参数变量取值表 CFIVT

Inst_addr	Input variable	Range of Input value	Selected Input value
...	...	...	...
m	r14	[-127,128]	[68,68]
...	...	...	...



表 12-3 参数变量  $b$  的取值范围

crel	Compare Relation (a rel b)	Value Range
eq	$a == b$	$[-128, 127]$
ne	$a != b$	$[-128, 127]$
lt	$a < b$	$[-128, 127]$
le	$a \leq b$	$[-127, 128]$
gt	$a > b$ signed	$[-127, 128]$
ge	$a \geq b$	$[-128, 127]$
ltu	$a < b$	$[0, 127]$ 或 $[2^{32}-128, 2^{32}-1]$
leu	$a \leq b$	$[1, 128]$ 或 $[2^{32}-127, 2^{32}]$
gtu	$a > b$ unsigned	$[1, 128]$ 或 $[2^{32}-127, 2^{32}]$
geu	$a \geq b$	$[0, 127]$ 或 $[2^{32}-128, 2^{32}-1]$

### 算法 12.7 执行路径逆向构造算法

输入：间接转移指令  $I_{\text{indirect}}$  和指令表  $\text{Table}_{\text{inst}}$ 。

输出：从程序主入口点到指令  $I_{\text{indirect}}$  的一条执行路径。

Begin

1. 把间接转移指令  $I_{\text{indirect}}$  的地址赋给指令指针  $I_{\text{pointer}}$  和  $I_{\text{pointer\_cmp\_or\_call}}$ ，同时置  $\text{flag}_{\text{long\_jump}}=0$ 。
2. 从指令表  $\text{Table}_{\text{inst}}$  中提取指针  $I_{\text{pointer}}$  对应的指令  $I_{\text{current}}$ ，并把  $I_{\text{current}}$  记录到指令表  $\text{Table}_{\text{inst\_current}}$  中，如果  $I_{\text{current}}$  为程序主入口点的首指令则转 11，当发现指令表  $\text{Table}_{\text{inst\_current}}$  中出现重复指令时，则说明逆向构造遇到了路径环，这时需要对指令表  $\text{Table}_{\text{inst\_current}}$  和 CFIVT 进行回退，回退到最近拥有多控制流源头的指令并重新从 CFSLT 中选择其控制流源头 CFS 继续路径的逆向构造，如果没有未遍历的控制流源头可供选择则继续向上回退，找到后把  $I_{\text{pointer}}$  设置为指向 CFS，同时置  $\text{flag}_{\text{long\_jump}}=1$ ，转 2。
3. 分析指令  $I_{\text{current}}$ ，如果  $I_{\text{current}}$  为一般指令进行如下操作：把  $I_{\text{pointer}}$  设置为指向紧邻  $I_{\text{current}}$  的上一条指令，同时如果  $\text{flag}_{\text{long\_jump}}==1$  则置  $\text{flag}_{\text{long\_jump}}=0$  并转 2。
4. 如果  $I_{\text{current}}$  为条件跳转指令，提取控制指令  $I_{\text{current}}$  是否执行的谓词寄存器  $PQ$ ，并置  $\text{flag}_{\text{base\_comp}}=1$  表示需要寻找到母体比较指令后继续处理，如果  $\text{flag}_{\text{long\_jump}}==0$  置  $PQ=0$ ，否则置  $PQ=1$ ，最后把  $I_{\text{pointer}}$  设置为指向紧邻  $I_{\text{current}}$  的上一条指令并转 2。
5. 如果  $I_{\text{current}}$  为比较指令，判断谓词寄存器  $PQ$  是否属于  $I_{\text{current}}$ ，如果  $PQ$  属于指令  $I_{\text{current}}$  且  $\text{flag}_{\text{base\_comp}}==1$  则根据  $I_{\text{current}}$ 、 $PQ$  的值和表 12-3 共同决定此控制流分支点输入数据的值并把相关的信息存储到 CFIVT 中，同时置  $\text{flag}_{\text{base\_comp}}=0$ ，最后置  $I_{\text{pointer\_cmp\_or\_call}}$  为  $I_{\text{current}}$  的指令地址，并把  $I_{\text{pointer}}$  设置为指向紧邻  $I_{\text{current}}$  的上一条指令并转 2。
6. 如果  $I_{\text{current}}$  为无条件跳转指令而不是间接跳转指令，则寻找  $I_{\text{pointer}}$  和  $I_{\text{pointer\_cmp\_or\_call}}$  之间指令对应的所有非此区间的控制流源头，并把它们存储到 CFSLT 中，然后选择一个控制流源头 CFS 继续逆向构造，并把  $I_{\text{pointer}}$  设置为指向 CFS，同时置  $\text{flag}_{\text{long\_jump}}=1$ ，转 2；如果  $I_{\text{current}}$  为间接跳转指令，则转 10。



7. 如果  $I_{current}$  为间接调用指令, 则转 10。

8. 如果  $I_{current}$  为某过程的首指令, 则寻找  $I_{current}$  对应的所有控制流源头并把它们存储到 CFSLT 中。如果 CFSLT 中指令  $I_{current}$  对应控制流源头链表不空, 则选择一个控制流源头 CFS 继续逆向构造, 并把  $I_{pointer}$  设置指向 CFS, 同时置  $flag_{long\_jump}=1$ , 转 2; 否则, 说明当前过程依赖于间接调用或者它为不可达的无用过程, 等待间接过程处理完后再次处理, 置  $flag_{return}=1$ , 退出。

9. 如果  $I_{current}$  为过程调用指令, 同时置  $I_{pointer\_cmp\_or\_call}$  为  $I_{current}$  的指令地址, 最后把  $I_{pointer}$  设置为指向紧邻  $I_{current}$  的上一条指令并转 2。

10. 在间接转移指令制约关系表 ITIRT 中记录指令  $I_{indirect}$  依赖于指令  $I_{current}$ , 终止当前  $I_{indirect}$  对应的执行路径逆向构造, 置  $flag_{return}=2$ , 退出。

11. 把控制流分支点输入数据取值表 CFIVT 中存储的项反序重新存储到 CFIVT 中, 即得到间接转移指令  $I_{indirect}$  对应的执行路径, 退出。

End

## 算法 12.8 总控算法

输入: 使用 objdump 处理可执行程序获得的汇编代码。

输出: 程序中间接转移指令对应的目标地址。

Begin

1. 调用算法 12.6 提取程序中所有间接转移指令, 并初始化间接转移指令制约关系表 ITIRT。

2. 从 ITIRT 表中选择一个间接转移指令  $I_{indirect}$ , 选择标准是  $I_{indirect}$  不受制于任何间接转移指令或者制约它的间接转移指令都已处理完毕, 同时清空控制流分支点输入数据取值表 CFIVT、控制流源头链表 CFSLT 和指令表  $Table_{inst\_current}$ , 置  $flag_{return}=0$ , 如果成功选择一个  $I_{indirect}$  则针对间接转移指令  $I_{indirect}$  调用执行路径逆向构造算法——算法 12.7, 否则转 8。

3. 如果  $flag_{return}=1$ , 标志  $I_{indirect}$  最后处理, 转 2。

4. 如果  $flag_{return}=2$ , 转 2。

5. 如果  $flag_{return}=0$ , 调用特定路径的控制执行算法——算法 12.9。

6. 记录间接转移指令  $I_{indirect}$  对应的目标地址到图 12-35 所示的 ITITAT 表, 以供算法 12.7 寻找控制流源头操作使用。

7. 转 2。

8. 退出。

End



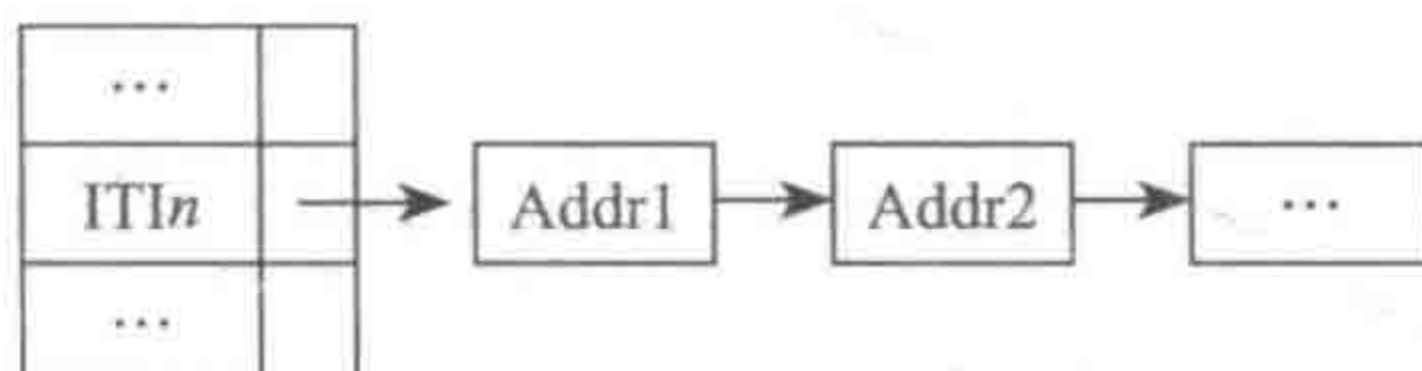


图 12-35 间接转移指令对应目标地址表 ITITAT

注：ITI 表示间接转移指令，Addr 表示目标地址

研究执行路径逆向构造技术的目的是一次性获得间接转移指令的全部目标地址，另外，算法 12.7 中步骤 8 和步骤 10 都要求一个总的控制程序来协调处理间接转移指令，并严格按照逆向构造生成的执行路径执行程序。算法 12.8 实现了针对各个间接转移指令逆向构造执行路径并执行此路径以获取相应目标地址的控制功能。

### 12.3.4 逆向构造执行路径的控制执行

12.3.3 节介绍了执行路径逆向构造算法，使用算法 12.7 处理间接转移指令都能成功逆向构造一条执行路径，间接转移指令处于无用过程体中的情况除外。虽然间接转移指令对应的执行路径被成功构造，但要获得间接转移指令的目标地址还需要严格按照构造的路径执行程序。本节将给出特定路径的控制执行算法。

#### 算法 12.9 特定路径的控制执行算法

输入：某一个执行路径对应的 CFIVT 表和间接转移指令  $I_{\text{indirect}}$ 。

输出：间接转移指令对应的目标地址。

Begin

1. 使用算法 12.5 处理间接转移指令  $I_{\text{indirect}}$ ，得到  $I_{\text{indirect}}$  对应的功能块、功能块控制指令、功能块输入变量和输入数据取值范围，如果功能块首指令领先于功能块控制指令，则根据功能块输入变量和输入数据取值范围更新 CFIVT 表中功能块控制指令对应表项的最后一项内容，同时把此表项的指令项调整为功能块首指令，如果功能块首指令落后于功能块控制指令，则在 CFIVT 表中功能块控制指令对应表项之后为功能块首指令添加一项，同时设置功能块的首指令为断点 BP0、间接转移指令  $I_{\text{indirect}}$  为断点 BP1。
2. 设置指针 PCFIVT 指向 CFIVT 表的第一个元素， $n=2$ 。
3. 提取指针 PCFIVT 指向元素存储的指令地址，在此地址设置断点 BPn，并提交程序运行。
4. 如果到达断点 BP0，则转 8。
5. 到达断点 BPn 后，根据指针 PCFIVT 指向元素存储的变量参数的值设置此变量，并一致性更新与变量参数相关的所有数据。
6. 设置指针 PCFIVT 指向下一元素，置  $n=n+1$ 。
7. 提取指针 PCFIVT 指向元素存储的指令地址，在此地址设置断点 BPn，并继续执行，转 4。
8. 提取指针 PCFIVT 指向元素最后一项存储的输入数据范围。
9. 如果输入数据遍历设置完毕转 11，否则，根据获得的输入数据范围遍历设置输入变



量，并继续执行。

10. 到达断点 BP1 时，记录间接转移指令  $I_{\text{indirect}}$  对应的目标地址值，同时转移控制流到断点 BP0 并转 9。

11. 退出。

End

步骤 1 根据功能块控制指令、功能块输入变量和输入数据取值范围，来更新 CFIVT 表中功能块控制指令对应表项的内容或者添加功能块首指令对应的 CFIVT 表项，其目的是为了在步骤 9 中遍历设置输入变量的值从而循环执行功能块以获得指令  $I_{\text{indirect}}$  对应的全部目标地址。步骤 10 只是简单记录  $I_{\text{indirect}}$  对应的目标地址，间接转移指令  $I_{\text{indirect}}$  与其全部目标地址的对应关系在算法 12.8 中被正式记录于间接转移指令对应的目标地址表 ITITAT。

### 12.3.5 针对程序执行路径逆向构造的验证

根据 12.3.3 节和 12.3.4 节及 12.2.3 节提出的间接点提取、程序执行路径逆向构造、特定路径控制执行和功能块划分技术，提取间接转移指令目标地址的流程图见图 12-36。针对待处理二进制程序实施线性扫描反汇编，将得到的汇编代码提交给算法 12.8，由算法 12.8 控制实施对间接转移指令目标地址的提取：首先调用算法 12.6 提取程序中含有的间接转移指令信息并返回；接着调用算法 12.7 得到与间接转移指令直接相关的执行路径并返回；然后调用算法 12.9 控制执行算法 12.7 生成的执行路径，其中，算法 12.9 首先调用算法 12.5 构建间接转移指令对应的功能块，然后根据生成的功能块及相关信息更新调整执行路径，之后严格按照生成的执行路径控制程序执行，并最终获得间接转移指令对应的目标地址。

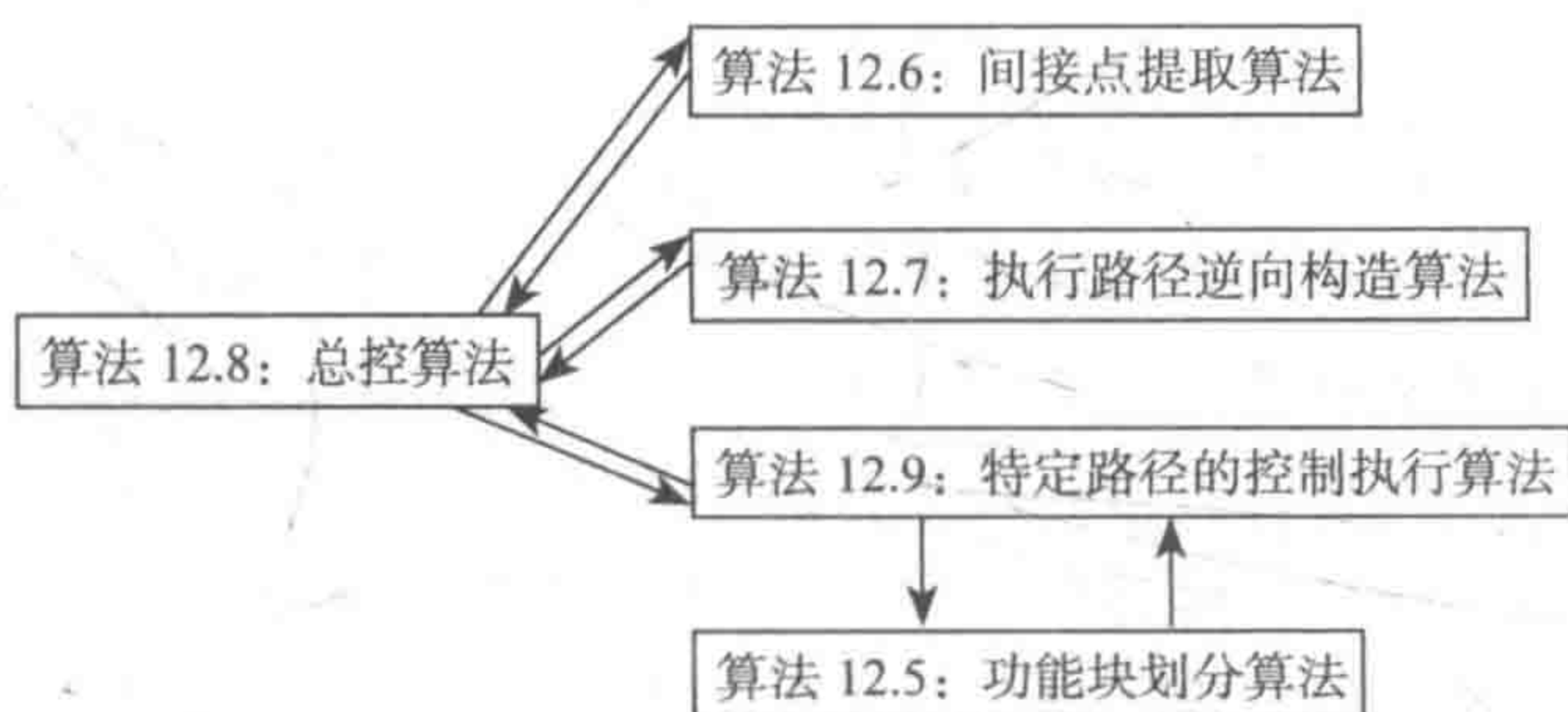


图 12-36 提取间接转移指令目标地址的流程图

使用 12.2.2 节给出的程序 switch.c 展示确定间接转移指令目标地址的

整个过程。采用 -O0 优化选项使用 GCC 编译 switch.c 生成可执行程序 a.out，使用线性扫描反汇编器处理 a.out 生成的汇编代码如图 12-37 所示。

首先将图 12-37 所示汇编代码提交给算法 12.8，使用算法 12.6 提取出单一的一个间接跳转指令 L054；然后把间接跳转指令 L054 提交给算法 12.7 进行执行路径逆向构造，构造获得的执行路径如图 12-38a 所示；随后使用算法 12.9 控制执行间接跳转指令 L054 对应的执行路径，算法 12.9 首先调用算法 12.5 构建 L054 对应的功能块，见图 12-20a，然后根据生成的功能块及相关信息更新调整 CFIVT 表，见图 12-38b，之后严格按照 CFIVT 表控制



程序执行，最终获得间接跳转指令 L054 对应的目标地址，见图 12-38c。

L001: alloc r34=ar.pfs,10,4,0	L040: st4 [r16]=r15	L079: adds r15=4,r35
L002: mov r35=r12	L041: adds r16=8,r35;;	L080: adds r14=4,r35;;
L003: adds r12=-32,r12	L042: ld4 r16=[r16];;	L081: ld4 r14=[r14];;
L004: mov r33=b0;;	L043: cmp4.ltu p6,p7=92,r16	L082: adds r14=1,r14
L005: adds r14=-12,r35;;	L044: (p06) br.cond.dptk.few L085	L083: st4 [r15]=r14
L006: st4 [r14]=r0	L045: adds r15=8,r35;;	L084: br.few L025
L007: adds r14=-8,r35;;	L046: ld4 r14=[r15]	L085: addl r14=96,r1;;
L008: st4 [r14]=r0	L047: shladd r15=r14,3,r0	L086: ld8 r36=[r14]
L009: adds r14=-4,r35;;	L048: addl r14=88,r1;;	L087: mov r32=r1
L010: st4 [r14]=r0	L049: ld8 r14=[r14];;	L088: br.call.sptk.many <printf>;
L011: mov r14=r35;;	L050: add r15=r15,r14	L089: mov r1=r32
L012: st4 [r14]=r0	L051: ld8 r14=[r15];;	L090: br.few L025
L013: adds r14=4,r35;;	L052: add r14=r14,r15	L091: addl r14=104,r1;;
L014: st4 [r14]=r0	L053: mov b6=r14	L092: ld8 r36=[r14]
L015: addl r14=72,r1;;	L054: br.few b6;;	L093: mov r32=r1
L016: ld8 r36=[r14]	L055: adds r15=-12,r35	L094: br.call.sptk.many <printf>;
L017: mov r32=r1	L056: adds r14=-12,r35;;	L095: mov r1=r32
L018: br.call.sptk.many <printf>;	L057: ld4 r14=[r14];;	L096: adds r15=-12,r35
L019: mov r1=r32;;	L058: adds r14=1,r14	L097: adds r16=-8,r35
L020: addl r14=80,r1	L059: st4 [r15]=r14	L098: adds r17=-4,r35
L021: ld8 r36=[r14]	L060: br.few L025	L099: mov r18=r35
L022: mov r32=r1	L061: adds r15=-8,r35	L100: adds r19=4,r35;;
L023: br.call.sptk.many <printf>;	L062: adds r14=-8,r35;;	L101: addl r14=112,r1;;
L024: mov r1=r32	L063: ld4 r14=[r14];;	L102: ld8 r36=[r14]
L025: mov r32=r1	L064: adds r14=1,r14	L103: ld4 r37=[r15]
L026: br.call.sptk.many <getchar>;	L065: st4 [r15]=r14	L104: ld4 r38=[r16]
L027: mov r1=r32	L066: br.few L025	L105: ld4 r39=[r17]
L028: mov r14=r8	L067: adds r15=-4,r35	L106: ld4 r40=[r18]
L029: adds r15=-16,r35;;	L068: adds r14=-4,r35;;	L107: ld4 r41=[r19]
L030: st4 [r15]=r14	L069: ld4 r14=[r14];;	L108: mov r32=r1
L031: adds r16=-16,r35;;	L070: adds r14=1,r14	L109: br.call.sptk.many <printf>;
L032: ld4 r14=[r16];;	L071: st4 [r15]=r14	L110: mov r1=r32
L033: cmp4.eq p7,p6=-1,r14	L072: br.few L025	L111: mov r14=r0;;
L034: (p06) br.cond.dptk.few L036	L073: mov r15=r35	L112: mov r8=r14
L035: br.few L091	L074: mov r14=r35;;	L113: mov.i ar.pfs=r34
L036: adds r15=-16,r35;;	L075: ld4 r14=[r14];;	L114: mov b0=r33
L037: ld4 r14=[r15]	L076: adds r14=1,r14	L115: mov r12=r35
L038: adds r15=-10,r14	L077: st4 [r15]=r14	L116: br.ret.sptk.many b0;;
L039: adds r16=8,r35;;	L078: br.few L025	

图 12-37 包含间接跳转指令的某程序汇编代码

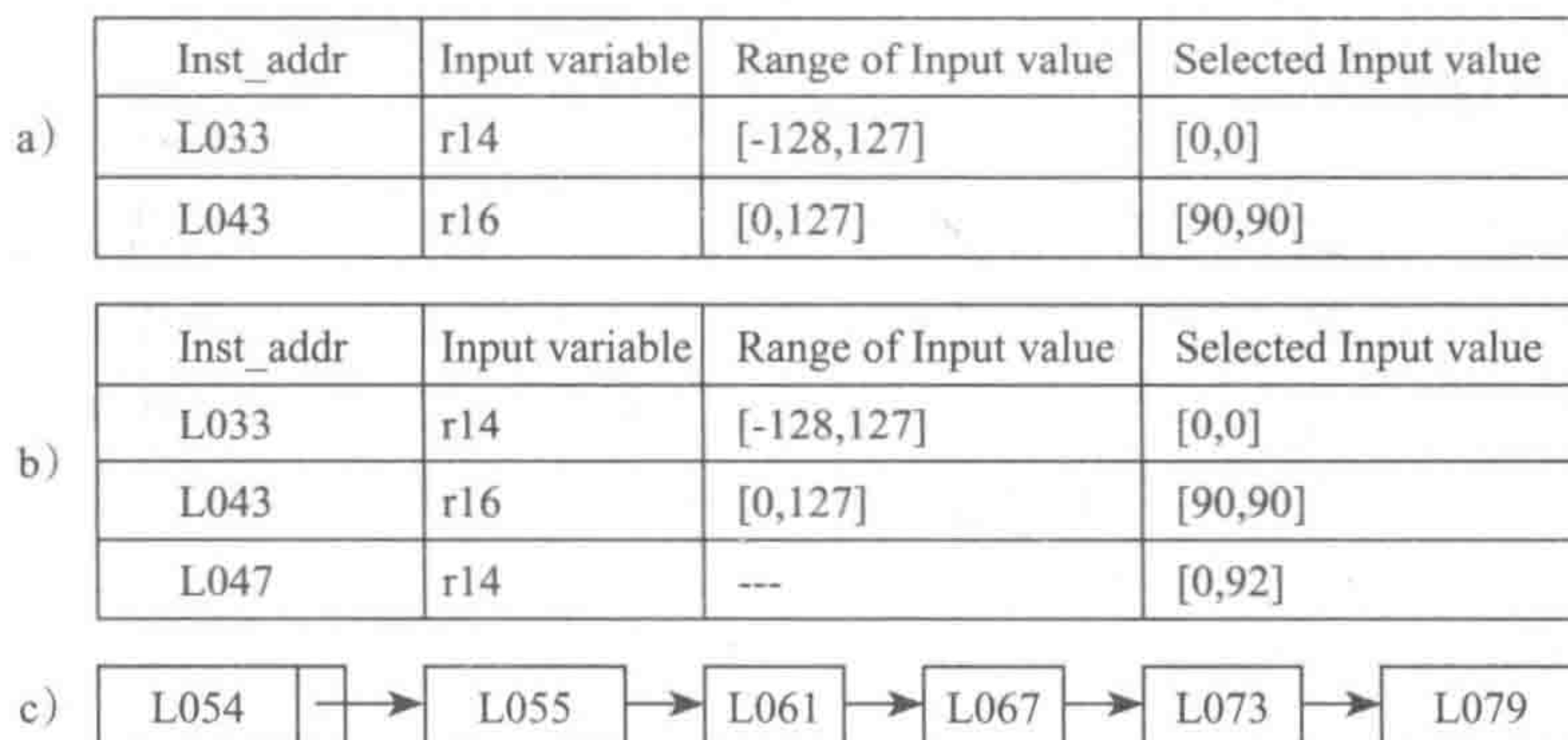


图 12-38 间接跳转指令 L054 处理信息图



因为测试用例较多，表 12-4 只给出了针对部分测试用例的处理结果。间接转移指令数是间接跳转和间接调用指令数的和。测试用例 libquantum.g2 包含一条间接跳转指令，间接跳转指令对应的功能块可以被成功划分，但由于此间接跳转指令处于无用过程体 quantum\_objcode\_run 中，最终导致逆向构造执行路径失败。

表 12-4 部分测试用例处理结果

测试用例名称	间接跳转指令数	间接调用指令数	间接转移指令数	成功处理的指令数	成功率
bzip2.g2	4	20	24	24	100%
bzip2.i2	2	20	22	22	100%
milc.g2	5	4	9	9	100%
milc.i2	0	3	3	3	100%
sjeng.g2	19	1	20	20	100%
sjeng.i2	0	1	1	1	100%
sphnx3.g2	2	7	9	9	100%
sphnx3.i2	0	8	8	8	100%
h264ref.g2	12	367	379	379	100%
h264ref.i2	0	372	372	372	100%
libquantum.g2	1	0	1	0	0%
gobmk.g2	10	30	40	40	100%
gobmk.i2	6	45	51	51	100%
gromacs.f2	33	274	307	307	100%

联合应用执行路径逆向构造技术和功能块划分技术处理四个测试集（基准测试集 SPEC2006、IEEE 浮点测试软件、Fortran78 Test Suite 测试集和自编测试集）的 1800 个测试用例，包含间接转移指令的功能块都能成功划分，间接转移指令对应的执行路径都能成功构造。因为间接转移指令导致没有被成功翻译的测试用例（如 bzip2.i1、bzip2.i2、gobmk.i2、gcc.g2、gcc.i1、gcc.i2、gromacs.f1、gromacs.f2、FM013.f0、FM257.f2、Fm912.f2 等），使用执行路径逆向构造技术、功能块划分技术和特定路径的控制执行技术进行处理可获得间接转移指令的所有目标地址。

## 12.4 本章小结

在本章的 12.1 节描述的实例中针对以间接跳转指令实现的多分支跳转，提出了一种基于关键语义子树的间接跳转目标解析及翻译的方法，该方法从语义角度分析了利用跳转表实现的多分支跳转代码的特征，在语义树的基础上抽取与实现跳转相关的关键节点，形成关键语义子树（CSS），完成索引值对应跳转目标的自动计算，从而实现了与指令集、编译器及优化选项均无关的跳转表恢复方法。

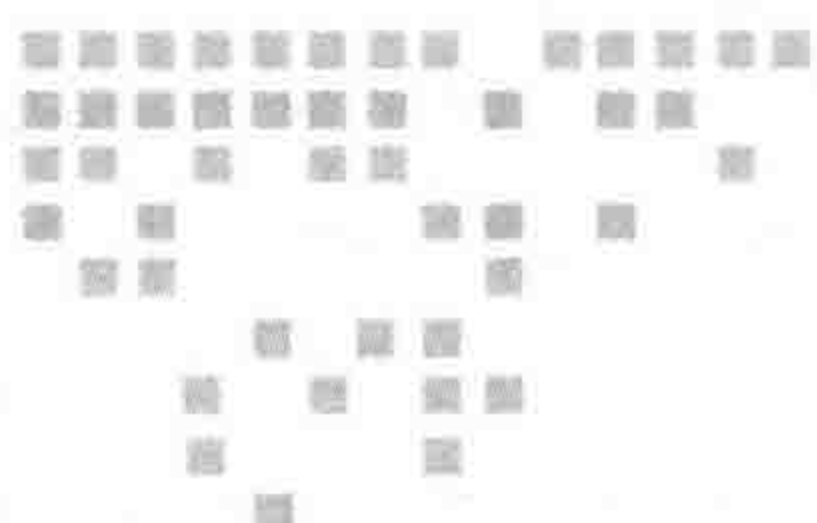
在本章 12.2 节和 12.3 节描述的实例中，首先解析了由于程序分析领域经常使用的基



本块、扩展型基本块、轨迹树和区间都不适合于提取间接转移指令的目标地址，以致对这类程序很难构造完整的控制流图。并据此提出了基于功能块的程序分析方法，有利于间接转移指令目标地址的提取，为解决静态二进制翻译面临的间接转移指令目标地址确定问题奠定了基础。

接下来，我们又介绍了关于间接转移指令目标地址确定问题的研究现状，众多研究基本上都是使用程序切片、复制传播和数据流分析等传统技术确定间接转移指令的目标地址，却只能部分解决此问题。其原因是在解码过程中遇到间接转移指令时，因为不知道间接转移指令对应的目标地址而导致反汇编解码失败。基于假设：“当解码间接转移指令之前已经知道间接转移指令所对应的目标地址，则递归扫描反汇编算法所面临的问题将不复存在”，我们在制导信息提取系统 CGIPS 框架内采取直接执行间接转移指令的策略，来获得间接转移指令对应的所有目标地址。为了降低获取间接转移指令目标地址的时空消耗，在本章的 12.2 节和 12.3 节描述的实例中提出了程序执行路径逆向构造技术和特定路径的控制执行技术，另外配合使用此前提出的基于功能块的程序分析技术可获得间接转移指令的所有目标地址。经四个测试集共 1800 个测试用例的验证，使用该技术可以一次性获得间接转移指令对应的全部目标地址。





## 反编译的推进 3——过程定义恢复

### 13.1 过程分析概述

#### 13.1.1 过程抽象

在进行程序设计时，可以将一段经常需要使用的代码封装起来，需要使用时可以直接调用，这就是程序中的函数。

库函数指的是用于支持程序正常运行的函数，通常包括入口函数及其所依赖的函数、各种语言的标准化函数的实现，以及用户提供的用于支撑其程序运行的函数集合。

本节着重研究如何对静态链接的库函数进行识别。静态链接的库函数一般以目标文件集合的形式存储在编译器或用户提供的库文件中，每个库函数的表现形式是一组编译过的目标代码。这些目标代码由链接器在程序创建的过程中与用户程序的目标代码链接在一起，共同生成一个可执行文件。

静态库函数识别指的是将可执行文件中静态链接的库函数代码部分识别出来，用规范的标识符代替具体的库函数目标代码，为后续的逆向分析和程序恢复工作奠定基础的过程。

库函数按照其适用范围可分为系统库函数和用户库函数。系统库函数一般指操作系统提供的一系列应用编程接口函数，以及各种语言的标准库函数集合；而用户库函数则指的是由用户提供的为满足特定需求而编写的专用函数集合。无论是系统库函数还是用户库函数，都只在指定范围、特定语言环境下才有意义。针对用户库函数识别进行的研究较少，本节在这方面进行尝试与探索，并总结出一种有效的库函数识别技术。

库函数特征指的是能够唯一标识某个库函数的函数属性的集合。

现有的库函数特征多种多样，每一种特征都有其相应的特征提取算法。按照模式特征



进行库函数识别，可以减少识别中所需要进行比较的内容，提高识别效率。

库函数识别过程存在一些无法避免的问题，这些问题将会影响到最后的识别结果。

### 13.1.2 调用约定分析

函数调用约定是指当一个函数被调用时，函数的参数会被传递给被调用的函数，以及返回值会被返回给调用函数。函数的调用约定就是描述参数是怎么传递和由谁平衡堆栈的，当然还有返回值。

几种类型：\_\_stdcall, \_\_cdecl, \_\_fastcall, \_\_thiscall, \_\_pascal。

参数传递顺序：

1) 从右到左依次入栈：\_\_stdcall, \_\_cdecl, \_\_thiscall, \_\_fastcall。

2) 从左到右依次入栈：\_\_pascal。

调用堆栈清理：

1) 调用者清除栈。

2) 被调用函数返回后清除栈。

常用描述：

(1) \_\_cdecl

\_\_cdecl 是 C Declaration 的缩写 (declaration, 声明)，表示 C 语言默认的函数调用方法：所有参数从右到左依次入栈，这些参数由调用者清除，称为手动清栈。被调用函数不会要求调用者传递多少参数，调用者传递过多或者过少的参数甚至完全不同的参数都不会产生编译阶段的错误。

1) 参数是从右向左传递的，也是放在堆栈中。

2) 堆栈平衡是由调用函数来执行的。

3) 函数的前面会加一个前缀 “\_”。

下面来看看具体的反汇编代码，这是从 C 反汇编的代码中截取的一部分代码，如表 13-1 所示。

表 13-1 C 语句及其对应的汇编指令

C 语句	对应的汇编指令
int c = 0;	00401088 C7 45 FC 00 00 00 00 mov dword ptr [ebp-4],0
c = sumExample(2, 3);	0040108F 6A 03 push 3 00401091 6A 02 push 2

从上面的两个 push 操作中就可以知道参数是从右向左传递的。另外这里也回答了前面的问题，即为什么参数会被扩展为 4 字节，因为堆栈的操作都是对一个字进行操作的，所以参数都是 4 字节。

00401093 E8 7C FF FF FF call @ILT+15(\_Max) (00401014)



这里就是调用函数操作了。在进行 `call` 操作之后，会自动将 `call` 的下一条语句作为函数的返回地址保存在栈中，也就是上面的“00401098”。在地址 00401014 处我们可以看到这样的一小段代码

```
@ILT+0(_Max):
00401005 E9 26 00 00 00 jmp _sumExample (00401030)
```

这里可以看出程序在编译之后会在函数前面加上前缀“\_”。

```
00401098 83 C4 08 add esp,8
```

这里就是平衡堆栈操作了。可以看出是由调用者进行的。

```
0040109B 89 45 FC mov dword ptr [ebp-4],eax
```

保存值是由 `eax` 寄存器返回的，从这里就可以看出来。

```
12:
13: return 0;
0040109E 33 C0 xor eax,eax
14: }
3: int __cdecl sumExample(int a, int b)
4: {
00401030 55 push ebp
00401031 8B EC mov ebp,esp
00401033 83 EC 40 sub esp,40h // 为局部变量预留空间
00401036 53 push ebx
00401037 56 push esi
00401038 57 push edi
00401039 8D 7D C0 lea edi,[ebp-40h]
0040103C B9 10 00 00 00 mov ecx,10h
00401041 B8 CC CC CC CC mov eax,0CCCCCCCCh
00401046 F3 AB rep stos dword ptr [edi]
```

上面的一段代码就是函数的开端了，也就是 `function prolog`。通过一些寄存器来对它们进行保存，也就像中断发生后需要保护现场一样。

```
5: return (a + b);
00401048 8B 45 08 mov eax,dword ptr [ebp+8]
0040104B 03 45 0C add eax,dword ptr [ebp+0Ch]
6: }
0040104E 5F pop edi
0040104F 5E pop esi
00401050 5B pop ebx
00401051 8B E5 mov esp,ebp
00401053 5D pop ebp
00401054 C3 ret
```

这里就是函数收尾，也就是 `function epilog`。

经过上面的分析，对 `__cdecl` 调用约定应该有一个比较清晰的认识了，但是还应该想想



为什么不在被调函数内进行堆栈平衡呢？在这里应该要考虑类似于 `scanf` 和 `printf` 这样的函数，应该明白这两个函数的参数都是可变的，如果参数不固定的话，在被调用函数内就无法知道参数究竟使用了多少字节，所以为了实现可变参数，必须要在被调函数执行之后才知道参数究竟用了多少字节，从而需由调用者来进行堆栈平衡操作。

## (2) `__stdcall`

被 `__stdcall` 关键字修饰的函数，其参数都是从右向左通过堆栈传递的（`__fastcall` 前面的参数由寄存器 `ecx`、`edx` 传递），函数调用在返回前要由被调用者清理堆栈。

这个关键字主要见于 Microsoft Visual C、C++。GNU 的 C、C++ 是另外一种修饰方式，即 `__attribute__((stdcall))`。

Win32 API 函数绝大部分都是采用 `__stdcall` 调用约定的。WINAPI 其实也只是 `__stdcall` 的一个别名而已。

```
#define WINAPI __stdcall
```

还是与上面一样，在函数的前面用 `__stdcall` 作为修饰符。此时函数将会采用 `__stdcall` 调用约定。

```
int __stdcall sumExample (int a, int b);
```

`__stdcall` 调用约定的主要特征是：

- 1) 参数是从右往左传递的，也是放在堆栈中。
- 2) 函数的堆栈平衡操作是由被调用函数执行的。
- 3) 在函数名的前面用下划线修饰，在函数名的后面用 “@” 来修饰并加上栈需要的字节数的空间（`_sumExample@8`）。

```
main 函数
push 3
push 2
```

这两个 `push` 可以说明函数的参数是由右向左传递的。

```
call _sumExample@8 // 调用函数
mov dword ptr [c], eax //eax 寄存器保存函数的返回值，此时将返回值赋值给局部变量 c
```

再来看看函数的代码。

函数的开端与 `__cdecl` 调用约定是相同的。

```
mov eax, dword ptr [a]
add eax, dword ptr [b]
```

函数的收尾也是与 `__cdecl` 调用约定是相同的。

另外在最后面将对堆栈进行平衡操作。

```
ret 8 // 两个 4 字节的参数
```



main 函数

```
0040108F 6A 03 push 3
00401091 6A 02 push 2
00401093 E8 81 FF FF call @ILT+20(_sumExample) (00401019)
FC mov dword ptr [ebp-4],eax
```

sumExample 函数

```
5: return (a + b);
00401048 8B 45 08 mov eax,dword ptr [ebp+8]
0040104B 03 45 0C add eax,dword ptr [ebp+0Ch]
00401054 C2 08 00 ret 8 // 堆栈平衡操作
```

因为栈的清理（堆栈平衡操作）是由被调用函数执行的，所以使用 `__stdcall` 调用约定生成的可执行文件要比 `__cdecl` 的小，因为每次函数调用都要产生堆栈清理的代码。函数具有可变参数，如 `wsprintf` 函数，与前面的 `printf` 一样都必须使用 `__cdecl` 调用约定，因为只有调用者知道每一次函数调用的参数数量，因此也只有调用者才能够执行堆栈清理操作。

### (3) `__fastcall`

这是一种快速调用方式。规定将前两个（或若干个）参数由寄存器传递，其余参数还是通过堆栈传递（从右到左）。

不同编译器编译的程序规定的寄存器不同。在 Intel 386 平台上，使用 `ecx` 和 `edx` 寄存器。

`__fastcall` 方式无法用作跨编译器的接口。

`__fastcall` 见名知其意，其特点就是快。`__fastcall` 函数调用约定表明了参数应该放在寄存器中，而不是在栈中，VC 编译器采用调用约定传递参数时，最左边的两个不大于 4 字节（DWORD）的参数分别放在 `ecx` 和 `edx` 寄存器。当寄存器用完的时候，其余参数仍然以从右到左的顺序压入堆栈。像浮点值、远指针和 `__int64` 类型总是通过堆栈来传递的。

下面来看看使用测试的源代码。

```
#include <stdio.h>
int __fastcall sumExample(int a, int b, int c)
{
    return (a + b + c);
}
double __fastcall sumExampled(double a, double b)
{
    return (a + b);
}
int main()
{
    int c = 0;
    double d = 0.0;
    c = sumExample(2, 3, 5);
    d = sumExampled(2.3, 2.5);
    return 0;
}
```

反汇编的代码片段是：



```

15: int c = 0;
004010C8 C7 45 FC 00 00 00 00 mov dword ptr [ebp-4],0
16: double d = 0.0;
004010CF C7 45 F4 00 00 00 00 mov dword ptr [ebp-0Ch],0
004010D6 C7 45 F8 00 00 00 00 mov dword ptr [ebp-8],0
17: c = sumExample(2, 3, 5);
004010DD 6A 05 push 5
004010DF BA 03 00 00 00 mov edx,3
004010E4 B9 02 00 00 00 mov ecx,2
004010E9 E8 26 FF FF FF call @ILT+15(@sumExample@8) (00401014)
004010EE 89 45 FC mov dword ptr [ebp-4],eax
18:
19: d = sumExampled(2.3, 2.5);
004010F1 68 00 00 04 40 push 40040000h
004010F6 6A 00 push 0
004010F8 68 66 66 02 40 push 40026666h
004010FD 68 66 66 66 66 push 66666666h
00401102 E8 FE FE FF FF call @ILT+0(@sumExampled@16) (00401005)
00401107 DD 5D F4 fstp qwordptr [ebp-0Ch][1]

```

## 13.2 库函数恢复

通过分析可执行文件的符号表,可以实现系统库函数调用的快速识别。而对于静态等方式编译的可执行文件,静态库函数识别方法主要有两种——基于特征数据库的模式匹配方法和基于函数签名的库函数识别方法。下面分别介绍这几种方法的基本思想和各自的优缺点。

### 13.2.1 快速库函数调用识别方法

对良性 PE 程序而言,识别其对 API 函数的调用,通过分析导入地址表 (Import Address Table, IAT) 即可以实现。当编译器生成库函数调用代码时,大都会采用动态链接机制。编译器在对源程序进行编译生成可执行程序的过程中,会在 PE 文件头中生成一个导入表,导入表中保存的是源程序所调用的库函数的函数名及其所在动态链接库文件的文件名等信息,这些信息都将在实际链接的时候被用到。当编译好的可执行程序实际执行时,Windows 装载器会将所需的动态链接库文件装入内存,并在调用库函数的指令与该函数的实际装载地址之间建立联系,从而实现该程序对库函数的调用。图 13-1 给出了一个示例程序对这一过程进行的说明。

从图 13-1 中可以看出,当一个 PE 程序调用到动态链接库中的 API 函数时,相应的 CALL 指令中的目标地址并不是直接指向动态链接库文件中相应函数的地址,而是跳转到代码段中的一条 JMP 指令处执行。JMP 指令的目标地址存储在 .idata 或者 .text 段中,该地址实际上是导入地址表 (IAT) 的一个入口地址。





图 13-1 通过导入表实现的库函数调用目标识别示例

## 13.2.2 基于特征数据库的模式匹配方法

### 1. 基本思路

这种方法的基本思路是提取库函数目标代码的某些属性作为模式特征，将提取出的模式特征统一存放至一个特征数据库中；识别时使用相同的提取方法提取待识别函数的特征，在特征数据库中进行模式匹配，若匹配成功，则输出该库函数名，否则，返回错误值指明该函数无法识别。

特征数据库的建立过程及函数的匹配识别过程通常是相互独立的，库函数目标代码可从库文件中提取，得到目标代码后即可进行特征提取和特征入库等工作。总的来说，特征的选取以及数据库中数据的组织形式是决定这一类方法优劣的关键因素。

### 2. 识别方法

基于特征数据库的识别方法一般分为以下几个步骤：

- 1) 通过分析总结库函数目标代码的规律确定一种能够唯一标识一个库函数的特征，如目标代码的前  $n$  字节、函数长度、基本块数目等。
- 2) 建立特征数据库。即对库文件进行分析，按照步骤 1 中确定的函数特征，对每个库函数的目标代码进行特征提取，并将获得的库函数特征按照特定格式存储至特征数据库中。
- 3) 对待识别的目标代码进行特征提取。即采用与步骤 2 相同的特征提取方法对目标代码中的每个函数进行处理，获得待识别函数的特征信息集合。
- 4) 特征匹配。即在特征数据库中搜索与待识别函数具有相同特征的库函数，若能够找到，则输出该库函数的名称；若无法找到，则识别失败。

为了提高识别的效率，通常采用多级索引的方式来建立特征数据库。将函数的一些基本属性——如函数长度、指令条数等——作为特征数据库的低级索引，将显著提高数据库查找的速度。



国内外很多研究成果都是基于这种方法实现的, 这些方法的不同之处大多体现在函数特征的选取方面: 华中科技大学的许向阳等提出以操作码序列作为库函数特征; 中国科技大学的胡政和陈凯明则提出在函数特征中加入可变操作码, 并且将指令条数作为特征之一, 以便于识别 C++ 模板函数; 哈尔滨理工大学的邱景提出了基于基本块划分的库函数识别技术, 以基本块的各种属性以及块间关系作为函数的特征。

以上提到的方法虽然在具体实现上各有不同, 但核心思想区别不大, 在对单一体系结构下的系统库函数进行识别时效果良好。然而, 随着库函数特征数据库的不断扩大, 识别的速度会严重降低, 系统资源的使用量也将无法控制。而且, 将不同种类、不同版本编译器下的库函数特征存放在同一个数据库中, 将增大造成识别冲突的概率, 导致函数识别率和识别准确率降低。

### 13.2.3 基于函数签名的库函数识别方法

#### 1. 基本思路

这种方法最早是 1995 年由澳大利亚昆士兰大学反编译研究小组提出的, 其基本思路如下:

- 1) 从库文件中提取库函数目标代码的模式特征, 将每个函数表示为一个模式。
- 2) 选择一种优化的散列函数对提取出的模式进行散列处理, 将得到的散列值存储在一个表中, 即对库函数进行签名。通常每一种编译器对应一张表, 表以签名文件的形式存在。
- 3) 根据可执行文件的启动代码来判断其编译器类型等信息, 最终决定使用哪一个或哪几个签名文件。
- 4) 对待识别的函数进行处理得到其散列值, 以确定该函数是否在表中。若在表中, 则返回库函数名称, 否则识别失败。

昆士兰大学的反编译器 dcc 中的库函数识别模块就采用了这种设计思路, 其函数签名部分使用了最小完美散列 (minimal perfect hashing), 具体算法将在后面进行详细介绍。

#### 2. 识别方法

反编译工具 IDA 使用的库文件快速识别与鉴定技术 FLIRT 是迄今为止应用最广泛的基于函数签名的库函数识别技术, 其识别方法具有一定的代表性。本节通过介绍其识别步骤来具体讲解基于函数签名的识别方法。

FLIRT 提取函数开头的 32 字节作为模式特征, 辅以函数长度、外部符号引用等属性作为辅助特征; 使用特定算法对函数特征进行签名处理, 并将得到的签名保存至特定的签名文件中——通常每种编译器对应一个签名文件; 进行函数识别时, 只需加载对应的签名文件, 即可完成识别工作。

FLIRT 有以下主要特点:

- 1) 识别算法所需要的信息保存在一个签名文件中。每个函数表现为一个模式, 模式是一个函数开头的 32 字节, 其中所有变数字节用符号标记。



2) 将签名文件的创建分为两个阶段: 库的预处理和签名文件创建。这使得具体签名文件的生成与输入库文件的格式无关, 扩展了适用范围。同时, 压缩所创建的签名文件以减少磁盘空间。

3) 签名文件可由适当的库文件自动生成, 不需要用户干涉, 而且, 对于不同的编译器厂商需要生成不同的签名文件。

4) 根据目标文件的启动代码 (starting code) 来确定使用哪一个签名文件, 之后提取目标文件中的函数模块, 在该签名文件中进行模式匹配以确定该函数是否为库函数, 并根据匹配结果加以标识。

5) 为了降低短函数的错误识别概率, 必须完全地记录任何一个对外部名字的引用。对于级联函数, 推迟至第一趟扫描完毕后再进行识别。

6) 使用树结构存储函数特征, 减少了识别过程中内存的使用量, 同时提高了模式匹配查找的速度。

结合上述特点, 使用 FLIRT 进行库函数识别的具体步骤如下:

1) 针对各种类型及各种版本的编译器生成相应的库文件签名, 存储至特定位置供识别时使用。

2) 对待识别文件的启动代码进行分析, 确定其使用的编译器类型及版本, 加载相应的签名文件。

3) 对待识别文件中的每一个函数模块进行特征提取, 将得到的特征与签名文件中存储的库函数特征进行比较, 输出识别结果。

签名机制避免了不必要的匹配, 大大提高了函数识别的效率。IDA 使用启动签名识别的方法自动选择需要加载的签名文件, 其原理即根据不同编译器的特征识别出该文件使用的编译器, 并加载该编译器对应的签名文件。由于系统函数库与编译器联系紧密, 该方法在识别系统库函数时无疑是行之有效的——但是, 在软件中使用领域函数库并不会影响到最终生成的可执行文件的编译器特征, 因此启动签名机制无法有效识别并自动加载领域函数库签名。面对未知的二进制文件, 手动选择需要加载的领域函数库签名显然是不现实的。因此这种方法在对领域库函数进行识别时存在一定的缺陷。

### 3. 签名算法及其应用

反编译器 dcc 使用最小完美散列函数对从目标代码中提取出的模式进行签名, 即计算模式的散列值并将其存储于一张散列表中。在介绍具体的签名算法之前, 先给出几个明确定义。

**定义 13.1 (散列函数)** 给定一个由  $m$  个单词组成的集合  $W$ , 其中每一个单词均是由从有序字母表  $\Sigma$  中取出的若干个字母组成的有限字符串。散列函数  $h: W \rightarrow I$  是一个映射, 它将集合  $W$  映射到一个指定的整数区间  $I$  上, 如  $[0, k-1]$ , 其中  $k$  是一个整数而且通常  $k \geq m$ 。散列函数对给定的单词进行计算, 得到一个地址 (区间  $I$  中的一个整数), 这些单词和地址 (称为散列值) 的对应关系被存储在一张表中, 该表称为散列表。



不同的单词经过散列函数计算后可能得到相同的地址,这种情况称为冲突。完美散列函数指的是不会产生冲突的散列函数,其严格定义如下。

**定义 13.2 (完美散列函数)** 若一个散列函数  $h$  满足对于任意的  $x_i$  和  $x_j$  属于  $W$ , 当且仅当  $i=j$  时才有  $h(x_i)=h(x_j)$ , 则称其为完美散列函数 (Perfect Hash Function, PHF); 若同时还满足  $k=m$ , 则称  $h$  为最小完美散列函数 (Minimal Perfect Hash Function, MPHf); 若一个最小完美散列函数同时满足对于  $x_i < x_j$ , 有  $h(x_i) < h(x_j)$ , 即满足“保序性”, 则称其为保序最小完美散列函数 (Order Preserving Minimal Perfect Hash Function, OPMPHF)。

以上性质都是针对某一个键值集合  $W$  的, 即所得散列函数对于另外的键值集合可能不再满足以上性质, 因此在对新的静态集合进行散列运算前, 需要重新构造该集合相应的完美散列函数。dcc 采用了一种利用随机图 (Random Graph) 以随机线性时间度生成最小完美散列函数的算法。下面针对这一算法进行详细介绍。

### (1) 主体思想

根据定义 13.1 及定义 13.2, 所生成的最小完美散列函数应当能够将  $m$  个单词一一映射到整数区间  $[0, m-1]$  上。算法首先建立一个拥有  $n$  个顶点、 $m$  条边的随机图, 其中每个单词对应一条边, 顶点个数  $n = O(m)$ 。这种基于随机图的算法将生成如下形式的最小完美散列函数:

$$h(w) = (g(f_1(w)) + g(f_2(w))) \bmod m$$

其中  $w \in W$ , 且有  $f_i: W \rightarrow \{0 \dots n-1\}$ ,  $g: \{0 \dots n-1\} \rightarrow \{0 \dots m-1\}$ 。函数  $f_i$  是一组辅助散列函数, 用于将集合  $W$  中的字符串 (单词) 映射到整数区间  $[0, n-1]$  上; 而函数  $g$  则将整数映射至区间  $[0, m-1]$  上, 以最终实现最小完美散列。按照此种形式, 对于一个给定的无向图  $G = (V, E)$ ,  $|E| = m$ ,  $|V| = n$ , 算法所要做的就是寻找一个函数  $g: V \rightarrow [0, m-1]$ , 使得函数  $h: E \rightarrow [0, m-1]$  是一个一一映射的函数, 其中函数  $h$  定义如下:

$$h(e = \{u, v\} \in E) = (g(u) + g(v)) \bmod m$$

换句话说, 算法需要对图中的每个顶点进行赋值, 使得对于图中的每条边, 该边的两个顶点值的和模  $m$  (图中边的数量) 的结果是一个区间  $[0, m-1]$  上的独一无二的整数。如果考虑任意可能产生的随机图类型, 这个赋值过程不一定总能够实现, 然而若图  $G$  是一个无环 (环是指从某一个顶点出发并且最终回到该顶点的一条通路) 图, 则可对该图进行深度优先遍历, 在遍历的同时完成赋值, 这样就一定能够实现赋值过程。赋值的具体实现方法在后文给出。

上述算法执行结束后, 就生成了从集合  $W$  到整数区间  $[0, m-1]$  的一一映射, 而此时的函数  $h$  即为所生成的最小完美散列函数。

### (2) 具体实现

算法的具体实现分为两个阶段: 映射阶段和赋值阶段。

在映射阶段, 输入的单词集合被映射到图  $G = (V, E)$ 。其中  $V = \{0 \dots n-1\}$ ,  $n$  的值待图生成之后确定;  $E = \{\{f_1(w), f_2(w)\} : w \in W\}$ , 其中  $f_i: W \rightarrow \{0 \dots n-1\}$ 。需要注意的是



这个随机图中不能有环出现, 因此随机图生成之后需要进行环检测, 直到生成合适的图为止。图的生成过程如图 13-2 所示。

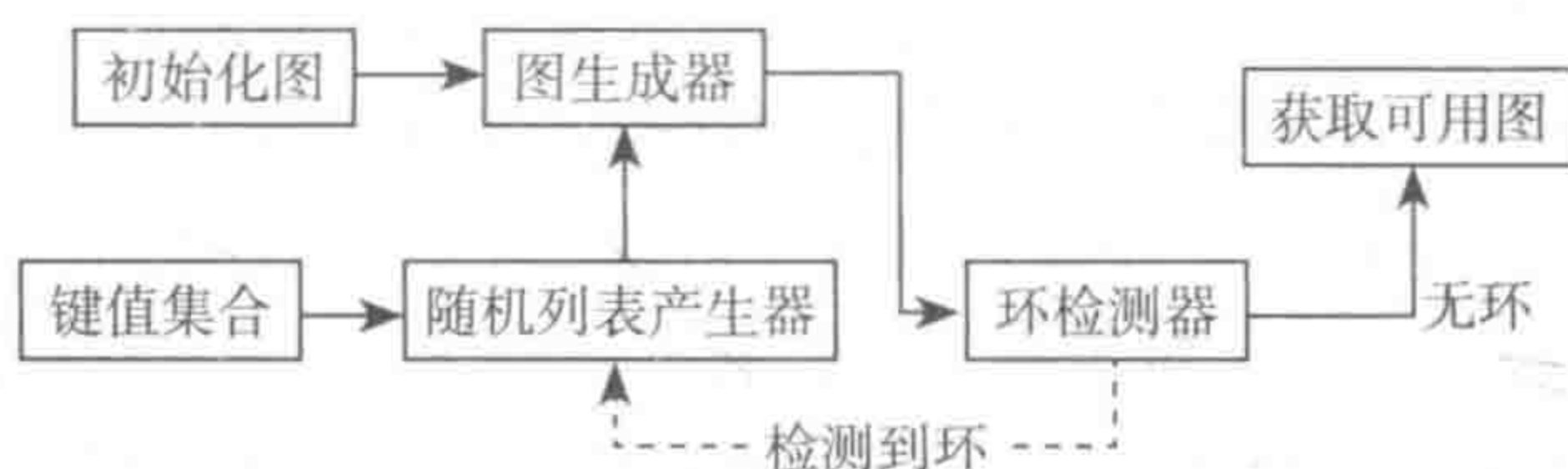


图 13-2 随机图生成过程

映射阶段使用到的辅助函数  $f_1$  和  $f_2$  应该是两个相互独立的随机函数, dcc 中这两个函数的形式如下:

$$f_1(w) := \left( \sum_{i=1}^{|w|} T_1(i, w[i]) \right) \bmod n$$

$$f_2(w) := \left( \sum_{i=1}^{|w|} T_2(i, w[i]) \right) \bmod n$$

其中  $|w|$  表示单词  $w$  的长度,  $w[i]$  则表示  $w$  中的第  $i$  个字符。  $T_1$  和  $T_2$  是两个随机整数列表, 可根据字符和该字符出现的位置得到一个模  $n$  的随机整数。算法的目的就此转化为寻找两个合适的随机表  $T_1$  和  $T_2$  使得图  $G$  是一个无环图, 由于没有确定的方法来解决这一问题, dcc 不断重复生成随机表, 并对对应的图进行检测, 直到产生一个有效图为止。

综上所述, 映射阶段的算法流程可粗略表示为如图 13-3 所示。

有效图成功生成后, 算法就进入赋值阶段。对于生成的拥有  $m$  条边的无环图  $G$ , 图中的每一条边  $e=(u,v) \in E$  都与集合  $W$  中的某个单词  $w$  唯一对应, 此时有  $f_1(w)=u$  且  $f_2(w)=v$ 。若  $w$  是集合  $W$  中的第  $i$  个单词, 则令  $h(e=(f_1(w), f_2(w)))$  等于  $i-1$ 。接下来对整个图进行深度优先遍历 (DFS): 对于每个连通分量, 任取一个顶点  $v$  作为开始顶点, 令  $g(v)=0$ , 从  $v$  开始进行 DFS; 若顶点  $j$  与顶点  $i$  邻接, 且连接  $i, j$  两点的边  $e=(i, j)$  的值为  $h(e)$ , 则令  $g(j)=(h(e)-g(i)) \bmod m$ 。对整个图的深度优先遍历结束后, 所有的顶点就都完成了赋值。

赋值阶段的算法流程可粗略表示为如图 13-4 所示。

赋值阶段结束之后, 函数  $g$  作用于图中所有顶点的值都完成了计算, 此时函数  $h(e=\{u,v\} \in E) = (g(u) + g(v)) \bmod m$  已经是集合  $W$  上的最小完美散列函数了。由于图  $G$  中没有环存在, 因此对于所有顶点  $v_i$ ,  $g(v_i)$  都只计算了一次, 这足以证明本算法生成的函数  $h$  是最小完美散列函数。

```

输入 键值集合 W
输出 随机图 G
do
    E := ∅
    随机生成表 T1、T2
    for each W ∈ W // 共有 m 个
    do
        f1(w) := (∑i=1|w| T1(i, w[i])) mod n
        f2(w) := (∑i=1|w| T2(i, w[i])) mod n
        将边添加至图 G
    endfor
while 图 G 中检测到环
Output G
  
```

图 13-3 随机图映射算法



```

function: traverse (u: vertex)
    visited[u] = TRUE
    for each w 邻接于 u
    do
        if not visited[w]
        then
             $g(w) := (h(e = u, w) - g(u)) \bmod m$ 
        endif
    endfor
end

visited[] := false // 将所有顶点置为未访问
for each  $v \in V$ 
do
    if not visited[v]
    then
         $g[v] := 0$ 
        traverse(v)
    endif
endfor

```

图 13-4 图顶点赋值算法

本算法的时间复杂度是与集合  $W$  中单词的数量  $m$  线性相关的。

### (3) 实际应用

对库文件进行处理并提取出每个库函数的模式特征之后,使用本算法对这些模式进行签名。dcc 中模式特征是由库函数目标代码的前 23 字节组成的固定长度的十六进制字符串,其中可重定位代码用通配符 (0xF4) 表示,这种长度固定、字符取值范围较小的键值显然可以进行最小完美散列处理。

由 dcc 函数模式的格式可知,用于进行最小完美散列签名的字母表  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ ,集合  $W$  由  $m$  个长度为 46 的字符串组成,其中的每一个字符均取自字母表  $\Sigma$ , $m$  的值则与具体的库文件有关。按照前面介绍的算法对这些键值进行处理,生成最小完美散列函数  $h$ ,并将每一个库函数的函数名及其散列值一同存储在签名文件中。

在进行函数匹配识别时,首先对待识别函数进行预处理提取出其目标代码的前 23 字节,然后使用相同的散列函数计算待识别函数的散列值,接下来就可以通过简单的查找快速判断出该函数是否存在于签名文件的散列表中。

需要注意的是,输入键值的唯一性是能够生成最小完美散列函数的基本前提。然而,以目标代码的前 23 字节作为库函数模式特征时,可能出现几个不同的库函数拥有相同特征的情况,这将导致  $W$  中存在若干个完全相同的字符串,也就是所谓的重复关键字 (duplicate key)。一旦有重复关键字出现,最小完美散列函数将无法生成。为了避免这种情况,dcc 引入了冲突排斥机制——即对于拥有相同模式特征的  $n$  个库函数,由用户决定舍弃其中  $n-1$  个,被舍弃的库函数不进入后续的签名过程。例如,对于下列两个函数:



```
__ntohs  0FB744240486C4C3.....
__htons  0FB744240486C4C3.....
```

这两个函数的模式特征完全相同，由生成签名文件的用户决定舍弃其中的一个，这样只有被选中的函数会进入集合  $W$  并最终连同其散列值存储于签名文件中。这种做法将导致所有拥有该特征的待识别函数均被识别为保留的那个函数，这无疑是不精确的。然而，这些特征相同的函数大多拥有相同的功能实现，因此将它们识别为其中的一个一般并不会影响程序辅助分析人员对汇编代码的理解。为了兼顾识别效率，dcc 选择了这种措施，在识别准确性与识别速度两方面进行了折中。

4. 两种函数识别方法的比较

通过对以上基于特征数据库和基于函数签名两种识别方法进行比较，可以总结出它们各自的优缺点。基于特征数据库的函数识别方法其优点在于思路较为简洁，可以依赖已有的数据库软件实现数据库建立与查询；缺点在于随着特征数据库规模的扩大，识别速度将显著降低，而且产生识别冲突的几率也将明显提高。基于函数签名的识别方法优点在于减少了需要进行匹配的模式的数量，而且使用散列签名的方法进行匹配查找能够极大地提高查找效率；其缺点在于实现方法较为复杂，而且根据启动代码来选择函数签名文件的方法在进行用户库函数识别时并不适用。表 13-2 综合考虑了这两种方法的特点，并给出了其适用范围。

表 13-2 两种函数识别方法对比

	基于特征数据库	基于函数签名
实现难度	简单	复杂
能否使用已有的数据库软件	能	不能
识别速度	慢	快
识别冲突	较多	较少
系统库函数识别	适用于在单一体系结构下、对指定编译器编译的可执行文件进行库函数识别	可对任意可执行文件直接进行识别
用户库函数识别	适用于任何可以得到库文件的用户库函数	无法自动加载用户库签名，无法自动识别

13.2.4 库函数参数的恢复

应用程序中往往使用了大量的库函数调用，通过库函数及其参数的恢复能够直接在目标代码中调用目标平台的库函数，有利于提高翻译后程序执行的效率。库函数名的恢复技术已经很成熟，在指令解码阶段我们可以利用现有的库函数恢复技术恢复出库函数调用的函数名。库函数的参数和返回值的名称和类型都可以通过预先分析源平台操作系统和编译器的函数库得到，可以把分析得到的每一个库函数信息设计成一个标签：<函数名，返回值类型，参数列表>，在库函数恢复时得到函数名后，通过查询库标签得到参数和返回值信息。但是有些库函数的参数必须经过一些特殊的分析才能得到。



### 1. 库函数格式串参数的分析

由于使用格式串参数的库函数的参数个数是不确定的，甚至某些情况下这些格式串在使用前是临时生成的，在静态翻译时无法得知参数的个数和类型。

在 x86\_64 平台上调用库函数时，参数按传递方式大体可分为两类：浮点类，除浮点以外的其他类别（统一简称为整型类）。整型类能通过 6 个通用寄存器和栈传参，浮点类能通过 8 个 xmm 寄存器和栈传参，并且任何参数在传递时都占用 64 位。所以在分析库函数参数时只需分析出参数在列表中的位置和参数属于两类传递方式中的哪一种，就能够将参数提取出来，然后传递给需要调用的目标机器库函数。图 13-5 给出了库函数的格式串参数分析算法。算法的流程如下。

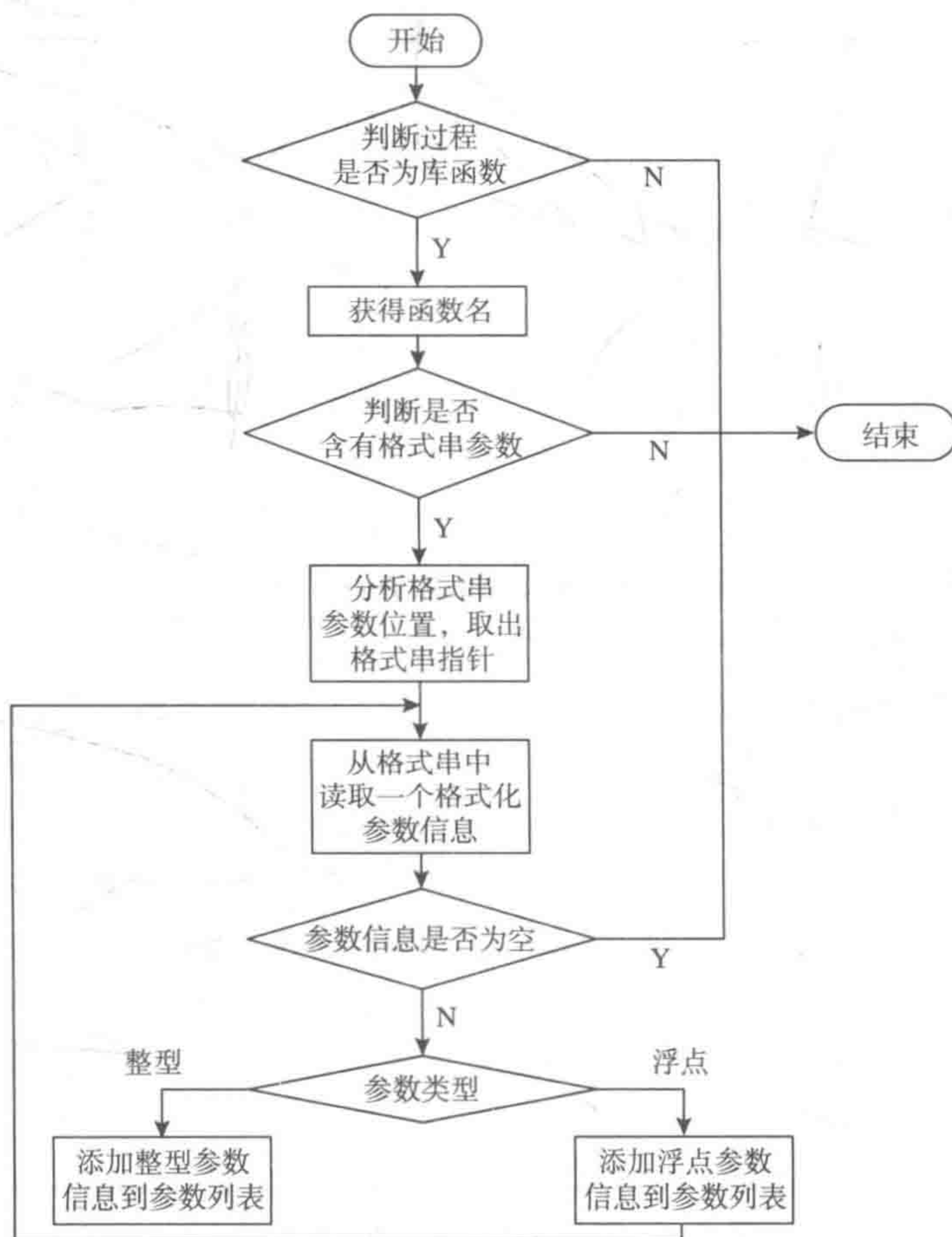


图 13-5 库函数格式串参数分析流程图

1) 判断给定的过程地址是否为库函数，若不是则转入步骤 4；若是则通过地址获得函数名，转入步骤 2。



2) 判断该库函数是否含有格式串参数, 如果含有, 则分析格式串参数在参数列表中的位置, 并利用已恢复的部分参数列表信息获得格式串指针, 转入步骤 3; 如果不含有则转入步骤 4。

3) 从格式串中读取一个格式化参数, 并判断读取的参数信息是否为空, 如果为空, 转入步骤 4; 如果不为空则分析该参数类型, 并将该参数类型信息添加到库函数的参数列表中, 然后转入步骤 3 开始处继续读取判断。

4) 格式串参数分析结束。

该算法的具体实现如图 13-6 所示。

```
Void LibFunAnalysis(uAddress FunAddress) // 库函数格式串参数分析
{
    if(islib(FunAddress)) // 判断是否为库函数
    {
        pFunName=getFunName(FunAddress); // 通过地址获得函数名
        switch(pFunName) // 根据函数名, 判断格式串在库函数参数中的位置
        {
            case "printf": case "scanf":
                formatposition=0; // 格式串为第一个参数
                break;
            case "sprintf": case "fprintf": case "sscanf": case "fscanf":
                formatposition=1; // 格式串为第二个参数
                break;
        }
        char *pFormat=getformat(formatposition); // 根据格式串位置, 得到格式串指针
        int ParaNum=formatposition; // 记录参数列表中位置, 初始为格式串的位置
        while (ParaType != getNextParaType(pFormat)) // 从格式串中读取一个带有 % 的格式化符号,
        { // 并返回格式符号类型 ParaType
            ParaNum++;
            switch(ParaType) // 根据格式符号类型为该次库函数调用
            { // 的参数列表添加参数位置和类型信息
                case "INTEGER": // 通过整型类型传参
                    AddPara("Int", ParaNum); // 添加参数到参数列表
                    break;
                case "FLOAT": // 通过浮点类型传参
                    AddPara("Flt", ParaNum); // 添加参数到参数列表
                    break;
            }
        }
    }
}
```

图 13-6 库函数格式串参数分析算法

## 2. 库函数参数恢复算法设计

通过库函数参数分析得到库函数的参数列表后, 就能够在生成调用目标机器库函数的



代码前插入其参数恢复代码，这些参数恢复代码是通过分析参数列表并进行源机器到目标机器参数转换生成的。目标机器使用整型和浮点参数混合编码的参数传递方式，即通过寄存器传参的整型和浮点的个数之和不超过6个，大于6个时使用栈来传参。目标机器分别提供了6个用于整型传参的寄存器和6个用于浮点传参的寄存器，并且在使用寄存器传递参数时，库函数参数列表中的第 $i$ 个参数必须保存在用于整型传参或用于浮点传参的第 $i$ 个寄存器中。根据源机器和目标机器的传参方式分析，构造了库函数的参数恢复算法，如图13-7所示，算法的流程如下。

1) 进行预处理，包括获得参数列表中参数的个数，初始化用于记录已处理的源机器整型传参寄存器个数  $iRegNum$  和浮点传参寄存器个数  $fRegNum$ ，设置将要处理的参数在参数列表中的序号从0开始等。

2) 判断当前参数序号是否小于参数总个数，若是，转入步骤3执行；若否，转入步骤9执行。

3) 获得当前参数序号对应的参数类型。判断参数序号是否小于6，若小于6则需要将该参数保存在目标机器的相应传参寄存器中，转入步骤4执行；若不小于6则需要保存在目标机器的栈中，转入步骤5执行。

4) 对参数类型进行判断，如果为整型，则将源机器中第  $iRegNum$  个整型传参寄存器的值保存在目标机器的第  $i$  个整型传参寄存器中，源机器整型传参寄存器计数  $iRegNum$  加1，转入步骤8；如果为浮点类型，则将源机器中第  $fRegNum$  个浮点传参寄存器的值保存在目标机器的第  $i$  个浮点传参寄存器中，源机器浮点传参寄存器计数  $fRegNum$  加1，转入步骤8。

5) 对参数类型进行判断，如果为整型，则转入步骤6执行；如果为浮点，则转入步骤7执行。

6) 判断参数在源机器中的传递方式，如果通过寄存器传递，则将源机器中第  $iRegNum$  个整型传参寄存器的值保存在目标机器栈中， $iRegNum$  加1，转入步骤8；如果通过栈传递，则判断已处理的浮点参数在源机器中是否有通过栈传递的，若有则将源机器栈中第  $(i-14)$  个参数的值保存在目标机器栈中， $iRegNum$  加1，转入步骤8；若没有则将源机器栈中第  $(iRegNum-6)$  个参数的值保存在目标机器栈中， $iRegNum$  加1，转入步骤8。

7) 判断参数在源机器中的传递方式，如果通过寄存器传递，则将源机器中第  $fRegNum$  个浮点传参寄存器的值保存在目标机器栈中， $fRegNum$  加1，转入步骤8；如果通过栈传递，则判断已处理的整型参数在源机器中是否有通过栈传递的，若有则将源机器栈中第  $(i-14)$  个参数的值保存在目标机器栈中， $fRegNum$  加1，转入步骤8；若没有则将源机器栈中第  $(fRegNum-8)$  个参数的值保存在目标机器栈中， $fRegNum$  加1，转入步骤8。

8) 当前参数处理完毕，参数序号加1，转入步骤2进行下一个参数的处理。

9) 参数列表处理完毕，结束。

针对该算法的具体实现，如图13-7所示。



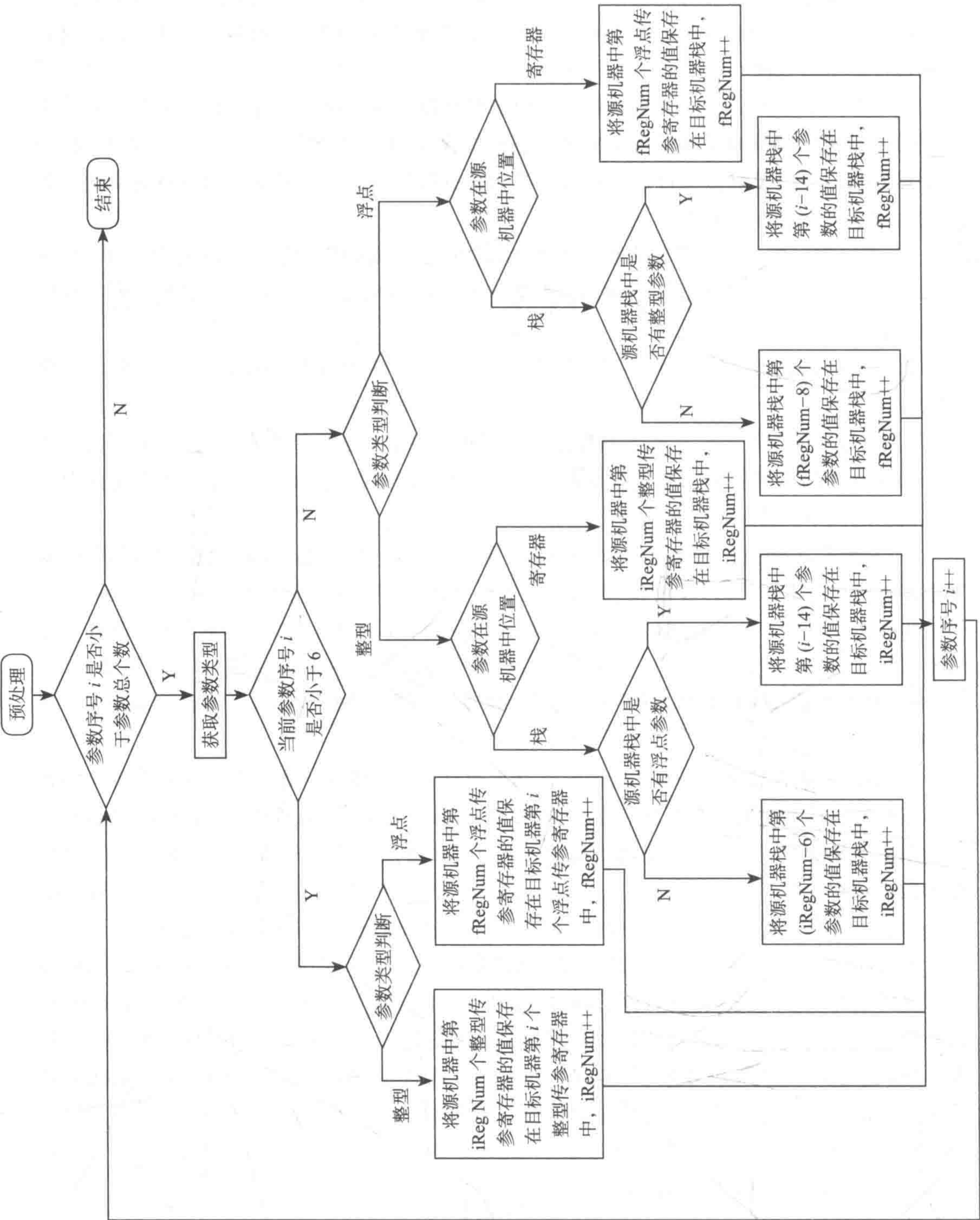


图 13-7 库函数参数恢复流程图



### 3. 临时生成的格式串参数的处理

如果库函数使用的格式串是在程序运行中临时生成的,则在静态分析程序时就得不到格式串的实际值,因此无法分析后续参数个数和类型。针对这种情况,笔者设计了针对使用临时生成的格式串参数的库函数包装处理方法。首先在遇到此类库函数调用时,将对该库函数的调用替换为一个包装函数。在包装函数开始处使用上述参数分析和参数恢复算法对传入的参数进行分析和恢复,然后调用真正的库函数实现原有功能,最后根据库函数返回值设置包装函数的返回值。

包装函数的开始和结尾部分屏蔽了源和目标机器传参的差异,对包装函数的调用完全实现了原有库函数的功能。但是,经过统计分析,发现大多数库函数的格式串参数在程序运行时是固定不变的,如果对所有使用格式串参数的库函数调用都使用上述处理方式,无疑会降低程序的执行效率。那么,能否找出一种方法区别临时生成的格式串参数的库函数和使用固定格式串参数的库函数调用呢?

经过对两种格式串参数不同用例的分析,可以发现,使用固定的格式串参数的程序通常将格式串保存在 `.rodata` 数据段中,而使用临时生成格式串的程序则将格式串保存在 `.data` 数据段和运行时的栈中。针对这种明显的区别,就可以在遇到使用格式串参数的库函数调用时,首先对格式串的地址进行分析判断,如果该地址属于 `.rodata` 数据段,则格式串是固定不变的,能够通过静态分析完成库函数参数的恢复;否则,则说明该格式串有可能发生变化,这时再使用库函数包装的方式进行处理。这种处理方式与对所有使用格式串参数的库函数均使用包装处理的方式相比,具有更好的效率优势。

## 13.2.5 隐式库函数调用

### 1. 隐式库函数调用方法

为了更好地研究隐式 API 函数调用行为的检测方法,本节选取了恶意代码常用的 API 函数隐式调用方法中的四类进行详细分析,并针对这四种方法分别给出检测的方法。

#### (1) 硬编码形式的库函数调用

对于相同版本的 Windows 操作系统,库函数模块的装载地址相对固定。因此,一部分恶意程序选择不通过导入表,而是在程序中直接调用库函数的装载地址,以达到调用库函数的目的。这种调用方法被称为“硬编码”(hard-coded)式的调用形式。

第一个 Windows 95 病毒 Boza,即 W95/Boza.A,对库函数的调用就是采取硬编码形式。该病毒使用的是 Windows 95 英文发布版中 API 函数的装载地址,将所有需要调用的库函数地址硬编码到程序代码里。例如,对 API 函数 `GetCurrentDirectoryA()` 的调用就是通过地址 `0xBFF77744` 的调用实现的。

硬编码是一种最简单的 API 函数隐式调用方法。但是由于同一个 API 函数的实际装载地址在不同版本的操作系统中并不完全相同,因此该方法的实用性并不强。所以,Boza 并



不能称为真正的 Windows 95 兼容病毒，因为它与许多版本的 Windows 95 操作系统并不兼容。

硬编码形式的库函数调用实现起来比较简单，检测也相对容易。检测的主要策略是首先建立库函数的硬编码地址库，然后在分析过程中根据调用目标地址（即硬编码地址）通过查表即可获得所调用库函数的相关信息。

(2) 同名函数实现的库函数调用

定义与库函数同名的用户自定义函数（以下简称同名函数），是恶意代码用来隐式调用 API 函数的另一种方法。该方法通过定义一个数据与代码混杂的用户自定义函数（该函数与将被调用的库函数同名）来迷惑反汇编算法，以达到隐藏其对库函数的调用目的。下面以 Win32.Bozano 为例对该方法以及该方法对反汇编算法造成的影响进行说明。

Bozano 为每一个将被其调用到的 API 函数定义了一个同名函数，每个同名函数的结构相同。图 13-8a 中给出了 Bozano 的部分源代码，该段代码实现了对 API 函数 CreateFileMappingA() 的调用。

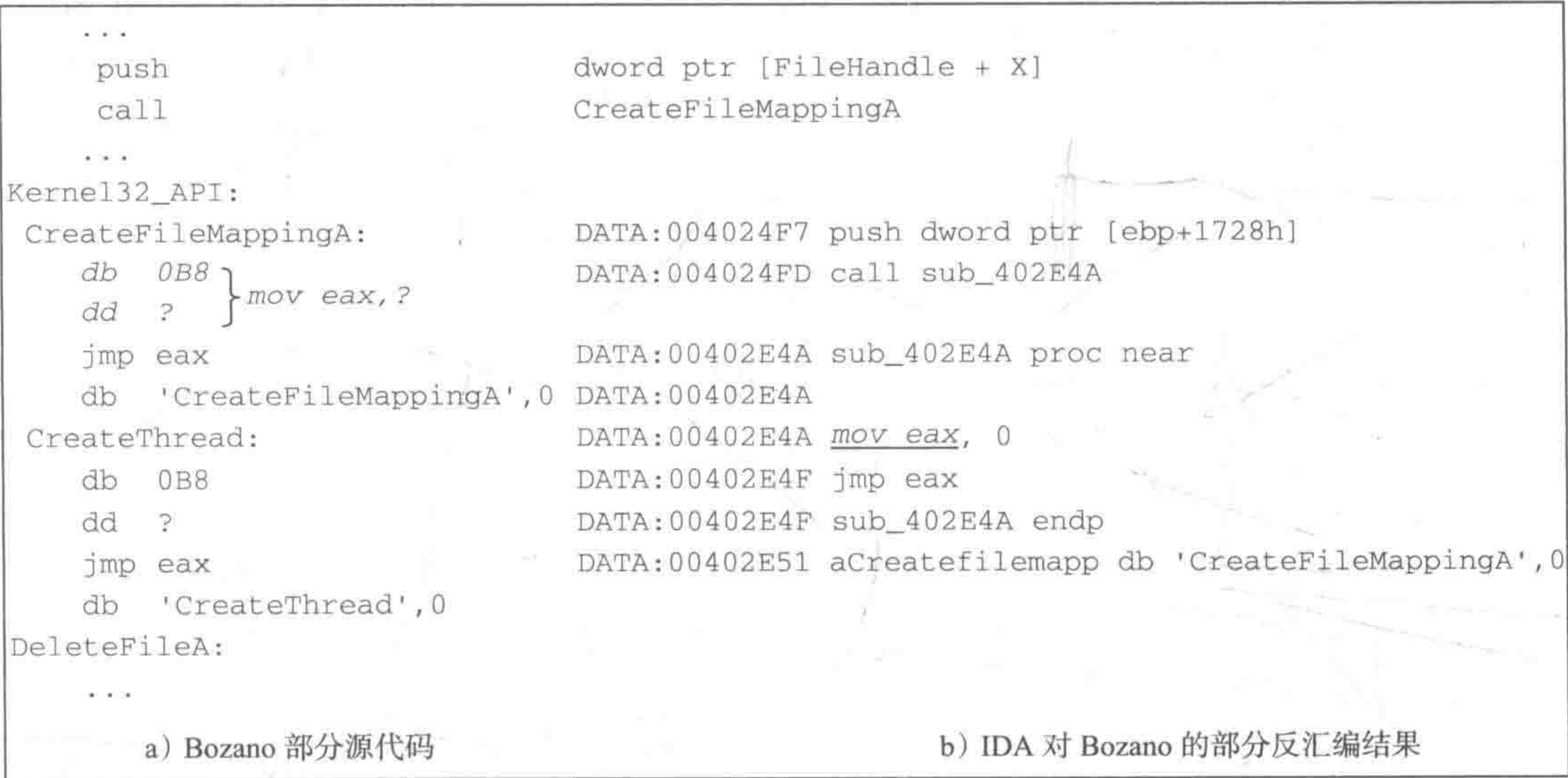


图 13-8 Bozano 源代码与 IDA 反汇编结果

从图 13-8a 中可以看出，每个同名函数中包含以下四部分内容：

- 1) 一个单字节数据，其值为十六进制常数 0B8。
- 2) 一个双字数据，该数据未赋初值，仅分配了相应的存储单元。
- 3) 一条无条件跳转指令，跳转的目标地址存放在寄存器 `eax` 中。
- 4) 一个字符串，该字符串的内容为 API 函数的函数名的 ASCII 码。

需要说明的是，不同的同名函数结构相同，仅仅是第四个数据项有所区别。第一个数据其实对应于 MOV 指令的操作码，并表明指令的目的操作数是寄存器 `eax`。在代码执行的



过程中，第二个数据将被动态地填入相应库函数的实际装载地址。

经过动态跟踪发现，Bozano 病毒调用库函数的过程分两步实现。第一步，Bozano 首先在内存中搜索 Kernel32.dll 的基址，然后从标号 Kernel32\_API 处开始，以固定的偏移量依次取出所有预先存放在同名函数中的字符串数据，即 API 函数的函数名。每取出一个 API 函数名，Bozano 都会从 Kernel32.dll 的基址处开始，按照函数名搜索出该函数的实际装载地址。接下来，Bozano 会将搜索到的地址存放到其同名函数中，即图 13-8a 代码中“?”所在的位置。

第二步，当调用某个 API 函数时，Bozano 将首先调用预先定义好的该函数的同名函数。在同名函数执行时，前两个数据实际上将被当作一条 MOV 指令执行，该指令完成的工作是将第一步中存入的相应 API 函数的实际装载地址传送到寄存器 eax 中。接下来执行指令 jmp eax，跳转到 API 函数地址处执行。由于同名函数中并没有对栈顶做修改，因此当 API 函数执行完毕返回时，程序将直接返回到调用同名函数的 CALL 指令后的下一条指令处执行，从而完成了对 API 函数的一次调用。

由以上分析可见，Bozano 通过定义 API 同名函数，并在该函数中使用一条隐含的 MOV 指令和一条 JMP 指令，从而实现了对 API 函数的隐式调用。这种方法能够很容易地迷惑使用模板匹配识别库函数代码的检测方法，使得检测工具错误地将对库函数的调用识别成对用户自定义函数的调用。

对待此类特殊的库函数隐式调用方法，首先需要能够识别出其调用形式，然后针对其特殊的调用形式，设计特定的隐式库函数调用行为识别方法。

### (3) 指令隐含操作实现库函数调用

在 x86 指令集中，CALL 指令是一类包含隐含操作的指令。在正常情况下，CALL 指令的语义其实是分两个执行步骤来实现的：

- 1) 将 CALL 指令的下一条指令的地址压栈，然后执行步骤 2。
- 2) 无条件跳转到 CALL 指令的目标地址处。

步骤 1 执行的操作给恶意程序编写者提供了将某些数据或者是代码隐藏在内存栈中的机会，并且由于没有使用 PUSH 等栈操作指令，这些被隐藏的内容也不容易被分析人员察觉到。

实际上，CALL 指令的隐含压栈操作确实给了恶意代码可乘之机。病毒 Win32.Velost.1186 就是使用此类方法的恶意代码之一。该病毒通过使用连续的隐含压栈操作，帮助其实现对 API 函数的重定位，并最终实现对 API 函数的隐式调用。

图 13-9 和图 13-10 分别给出了 Win32.Velost.1186 病毒的部分源代码。病毒首先执行图 13-9 中的代码。从图 13-9 中可以看出，该病毒执行了一系列的 CALL 指令，通过该指令隐含的压栈操作，将所有需要进行重定位的 API 函数的函数名存储在内存中一块连续的区域，寄存器 esi 指向最后存入的函数名所在位置。



```
Kn1ApiCount=20
...
RelocalKn1Api:
    sub    esp,size ApiAddressList+10h;// 在栈中存放 API 的地址
    mov    edi,esp
    call   PushKn1ApiStr19;// 将函数名压入栈中
    db     'ExitProcess',0
    PushKn1ApiStr19:
        call PushKn1ApiStr18;// 将函数名压入栈中
        db     'RegisterServiceProcess',0
    PushKn1ApiStr18:
        ...
    PushKn1ApiStr01:
        call PushKn1ApiStr00
        db     'LoadLibraryA',0
    PushKn1ApiStr00:
        mov    ecx,Kn1ApiCount ;// 共需要定位 Kn1ApiCount 个 Kn132Api 函数
```

图 13-9 Win32.Velost.1186 部分源码——API 函数名压栈

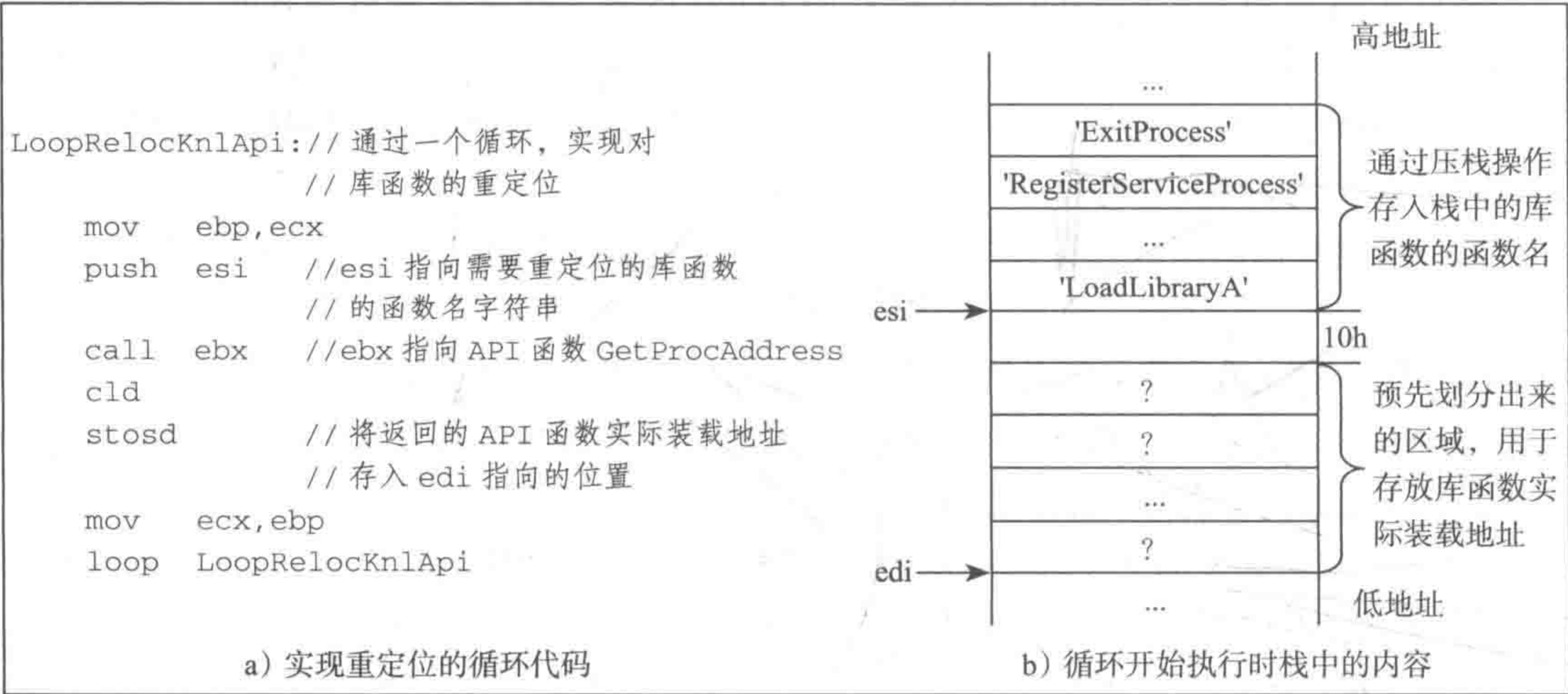


图 13-10 Win32.Velost.1186 部分源码及相应内存状态——循环实现重定位

接下来，病毒将执行图 13-10a 中的代码。该段代码实际执行的操作是通过一个 LOOP 循环，多次调用函数 GetProcAddress，得到由 esi 指向的函数名所对应的 API 函数的实际装载地址，然后将该地址存入到寄存器 edi 指向的栈单元中。图 13-10b 给出了这段循环代码开始执行时栈中的内容。

IDA 对 Win32.Velost.1186 病毒中数据与指令混杂的情况处理得并不理想。图 13-11 比较了同一段二进制代码的手工反汇编结果与 IDA 得到的错误的反汇编结果。结果表明，由于无法正确地区分数据与指令，IDA 将本为一个数据和一条指令的内容解码成 7 条指令和 1 个数据。本应把字符串数据的二进制码“0x69 0x62 0x72 0x61 0x72 0x79 0x41”解码成一



条 IMUL 指令，而作为字符串结束标志的字节“00”被错误地当成了下一条指令的开始位置，从而导致 MOV 指令被错误地解码成 ADD 指令。

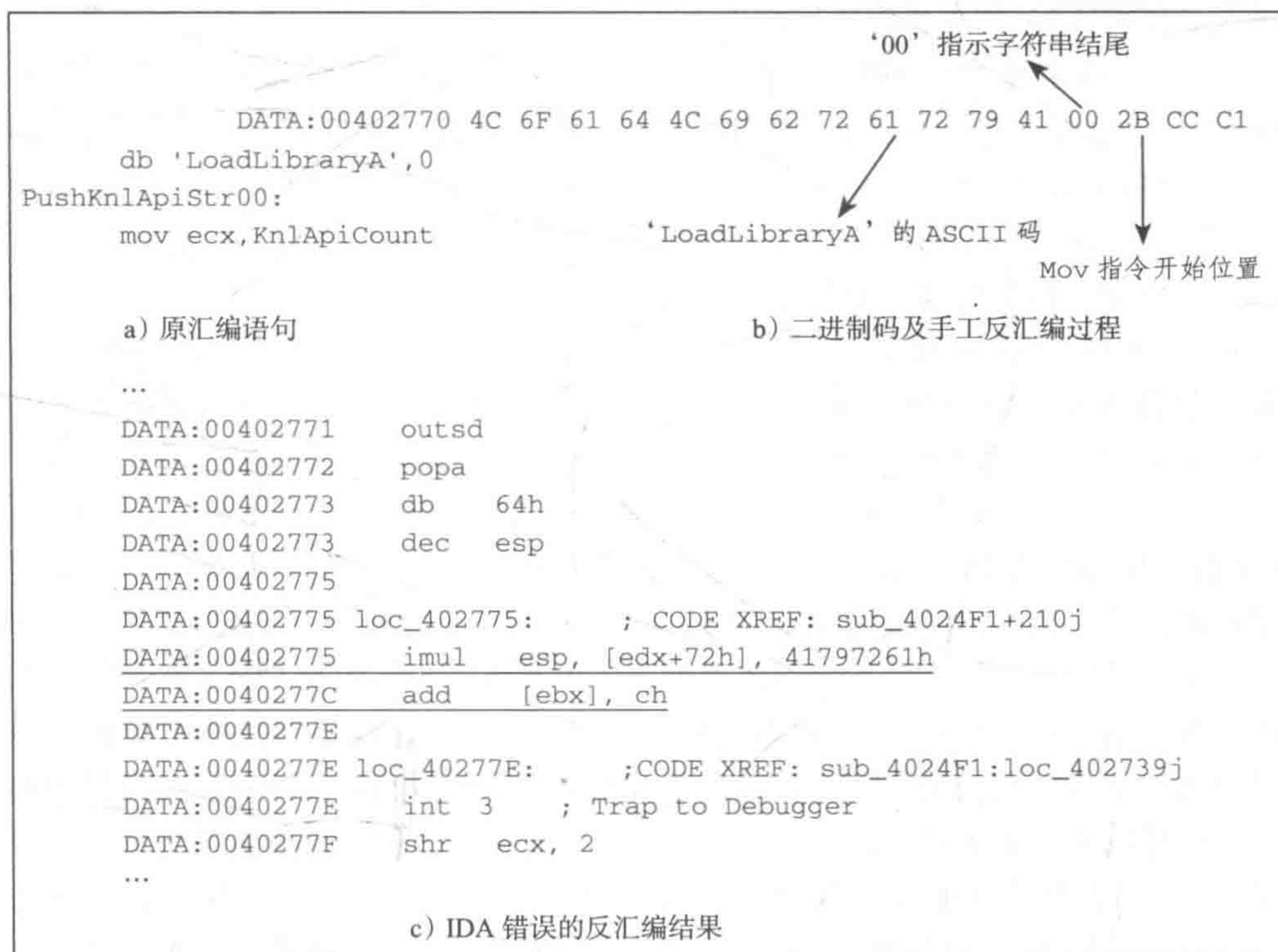


图 13-11 IDA 对病毒 Velost 错误的反汇编结果分析

从以上分析可以看出，利用指令的隐含操作实现库函数调用实现起来并不复杂，但是要想通过对可执行代码进行反汇编从而识别出其中隐含的库函数调用行为却比较困难。造成这一局面的根本原因是反汇编算法无法正确地区分数据与指令。如果反汇编算法得到的反汇编结果本身就存在错误，那么，要想通过对反汇编结果的分析识别其中隐含的库函数行为将会更加困难。

#### (4) 数组实现的库函数调用

通过使用若干数组来实现对 API 函数的隐式调用是目前恶意代码最常用的一种 API 调用方法。这种方法比上面提到的方法更加普遍，也更加难以发现。在此方法中，数组通常用来存储库函数的函数名、重定位地址以及偏移量等信息。

下面以 Win32.Aztec 为例，对使用数组实现的库函数调用方法进行分析。图 13-12 显示了 Win32.Aztec 的部分源代码。

对图 13-12 中的源代码进行分析可知，Aztec 中定义了两个数组：@@Namez 和 @@Offsetz。其中，数组 @@Namez 中预先存放有 Aztec 需要调用的所有 API 函数的函数名，每个函数名为数组的一项。数组 @@Offsetz 是偏移量数组，偏移量初始化为全零，但是每个偏



移量都对应有一个类似函数名的标号。该数组在运行时将被用来存放 API 函数实际装载地址相对于文件装载位置的偏移量。

进一步分析 Aztec 的源代码可以发现, Aztec 首先在内存中搜索 Kernel32.dll 的基址。接下来, 依次从函数名数组 @@Namez 中取出一个 API 函数名字符串。每取出一个 API 函数名字符串, Aztec 都会从 Kernel32.dll 的基址处开始, 按照函数名搜索出该函数的实际装载地址的相对偏移量, 然后将偏移量放入偏移量数组 @@Offsetz 中, 并且保证每个 API 函数在函数名数组 @@Namez 和偏移量数组 @@Offsetz 中的相应序号不变。在实际调用 API 函数时, Aztec 通过“基址 + 偏移量标号”的方式实现对相应 API 函数的调用。

```

virus_start label byte
aztec:
    ...
    lea edi,[ebp+@@Offsetz]
    lea esi,[ebp+@@Namez]
    ...
    push eax
    call [ebp+_FindFirstFileA]
    ...
@@Namez          label byte
@FindFirstFileA  db    "FindFirstFileA",0
@FindNextFileA   db    "FindNextFileA",0
    ...
@@Offsetz        label byte
_FindFirstFileA  dd     00000000h
_FindNextFileA   dd     00000000h
    ...

```

图 13-12 Aztec 的部分源代码

在分析 Aztec 的源代码时, 通过调用指令中的偏移量标号即可以简单地判断出所调用的是哪个 API 函数。但是在编译 Aztec 后生成的可执行代码中, 调用指令中的偏移量标号已经被编译成为一个相对地址, 并且该地址处对应的二进制代码为全零, 见图 13-13 所示。因此, 分析人员无法直接从二进制代码中获知被调用 API 函数的信息。通过这种方式, Aztec 实现了其对 API 函数调用行为的隐藏, 从而迷惑了分析人员和相关的防护软件。

AZTEC 源代码 :	call [ebp+_GetWindowsDirectoryA]
IDA 反汇编结果 :	CODE: 00401077      call      ss:dword_4015A8[ebp]
	// 偏移量标号被替换成实际偏移量
	...
	CODE: 004015A8      dword_4015A8      dd      0
	// 偏移量处值为全 0

图 13-13 IDA 对 Aztec 的部分反汇编结果及分析

实际上, 还存在一些其他的使用数组实现的隐式库函数调用方法。这些方法与 Aztec 的实现方法之间存在或多或少的差别。但是, 通过对此类恶意程序的分析可以发现, 此类程序有一个共同的特点, 即始终存在一个字符串数组用来存放要调用的 API 函数的函数名。因此, 只要能够识别二进制代码中的函数名数组, 并且能够正确地辨别恶意程序对 API 函数的具体调用方式, 就可以分析出恶意程序对 API 函数的隐式调用行为。

## 2. 隐式库函数调用检测技术设计

前面介绍了恶意程序常用的四种库函数隐式调用方法, 并对其实现过程和特点进行了



分析。本节将首先对这四种调用方式的共性进行研究，然后在此基础上归纳和剖析隐式库函数调用的实现机理，为下一步设计与实现库函数的隐式调用识别方法提供依据。

#### (1) 隐式调用方法的共同特征

不同编程人员写出来的汇编程序千差万别。但是从前面介绍的四种恶意程序常用的库函数隐式调用方法中可以发现，不同的库函数隐式调用方法仍然存在一些共同的特征，概括来说有以下几点：

1) 在不使用导入表的情况下，恶意程序需要自行实现对库函数的重定位。

从前文中可以知道，恶意程序要想实现其恶意功能，必须调用系统提供的库函数，包括 DLL 函数以及一些底层的系统函数等。但是出于隐蔽性考虑，恶意程序往往又需要避免使用导入表来实现对库函数的调用。因此，要想实现其恶意目的，恶意代码编写者必须在代码中自行实现对库函数的重定位。

2) 要自行实现对库函数的重定位，恶意程序中需要提供库函数的函数名信息。

要实现对库函数的重定位，恶意程序中必须提供能够指定到具体函数的库函数信息，比如库函数的函数名、API 函数在 DLL 中的序号等。由于动态链接库文件随着操作系统版本的不同会发生变化，其中不同版本的操作系统中相同序号对应的库函数可能并不是同一个函数，而对于库函数函数名的变化则较少，所以恶意程序常常会使用函数名来实现对所调用 API 函数的重定位。换句话说，恶意程序要想对 API 函数进行重定位，程序中就必然会出现 API 函数的函数名信息。

3) 恶意程序获取库函数装载地址大多通过一段循环结构的代码来实现。

用循环结构代码来实现库函数重定位符合编程人员的编程习惯。从编程习惯来说，定位不同库函数的过程是相似的，符合循环结构代码的特征，因而大多数编程人员会采用循环结构代码来实现库函数的重定位。如果不采用循环结构，而是在每次调用库函数时进行库函数的重定位，那么每调用一次库函数前都需要一段实现重定位的代码，并且还有可能对多次调用的同一个库函数编写多段相同的重定位代码。这样造成的后果必然是代码量的增加。对于追求代码体积小、隐蔽性强的恶意程序编写者来说，这无疑是一个不希望看到的结果。

4) 恶意程序会对库函数的函数名采取一定的隐藏手段，但是隐藏手段一般不会过于复杂。

为了防止安全防护软件察觉恶意程序中库函数的函数名信息，恶意代码编写者可能会采用一定的手段来隐藏库函数的函数名，比如将函数名字符串中的每个字符的 ASCII 码值加上一个固定的差值，使得二进制代码分析工具分析出的字符串与原函数名并不相同，从而迷惑分析工具。但是，由于隐藏的手段越复杂，相应的编程代码量就越大，所以，出于隐蔽性考虑，恶意程序对库函数函数名的隐藏手段一般不会太复杂，理论上通过分析应该能够予以解决。

将上述特征归纳为以下假设条件，即：

**假设一：**恶意程序通过循环代码自行实现对库函数的重定位。



**假设二：**二进制可执行代码中存在函数名字符串信息，且不对函数名加密。

假定以上假设条件对于所有的恶意程序都成立，本章在以上假设的基础上研究隐式调用行为的检测策略。

## (2) 隐式库函数调用行为检测策略

隐式库函数调用行为检测的目标包括两部分：

- 1) 从二进制代码中识别隐式库函数调用行为。
- 2) 确定隐式库函数调用行为所调用的目标函数。

由硬编码实现的 API 函数隐式调用的检测思路是，预先将恶意程序常用的库函数在某些操作系统上的实际装载地址与函数名之间的对应关系存入一个二维映射表中，然后根据调用地址查表即可确定被调用的目标库函数。通过定义同名函数实现的隐式库调用，由于其调用方法较特殊，因此可以首先识别其调用方式，然后再实现对调用函数的识别。主要方法可以是首先通过指令序列库中的指令模板识别调用形式，然后从同名函数体中恢复 API 函数的函数名，接着将对同名函数的调用替换成对相应库函数的调用，最后将同名函数体从反汇编结果生成的中间代码中删除，从而完成对隐式 API 调用行为的检测。

通过指令隐含操作实现库函数调用的恶意代码，仍然需要准备一个数组用于存放重定位后的库函数实际装载地址。因此，指令隐含操作实现的隐式库函数调用也可以看作通过数组实现的隐式库函数调用。而检测数组型隐式库函数调用行为的主要策略是识别可执行程序中的相关数组，通过挖掘数组之间的联系，确定调用地址与函数名之间的对应关系，从而实现对数组型隐式库函数调用行为的检测。由于使用数组实现库函数隐式调用的情况更为普遍，其检测方法也较其他几种复杂，所以本节将重点介绍数组型隐式库函数调用行为的检测方法。

具体而言，检测数组型隐式库函数调用行为的检测策略由以下几步实现：

- 1) 构建库函数函数名库，存放恶意程序常用的库函数函数名。

构建库函数函数名库的原因，主要是为了保证从二进制可执行程序中识别出来的函数名信息的可用性。在库函数函数名库中，库函数函数名字符串按字母顺序排列。当一个字符串从可执行程序中识别出来时，会首先在函数名库中查找该字符串，判断该字符串是否存在。如果存在，则表示该字符串对应的库函数可能在该程序中被调用，说明该字符串数据识别成功，并将该库函数名及其所在地址保存到相关数据结构中。

- 2) 构建指令序列库，用于实现对部分隐式调用方法的识别。

在某些隐式调用方法中存在一些特殊的指令序列，其可以用于识别特殊的库函数隐式调用形式。以 Bolzano 为例，作为库函数调用行为的关键指令序列为：

```
Mov    eax, 0;
Jmp     eax
```

可将该指令序列作为识别同名函数型隐式调用方式的特征指令序列模板，存入指令序列库中。在反汇编阶段，判断可执行程序中是否存在该指令序列。如果存在该指令序列，



则接下来对指令序列后的二进制码进行预分析,以判断是否为字符串数据,如果是则还需要进一步分析该字符串是否能够在库函数函数名库中找到。只有当分析出来的字符串数据存在于函数名库中时,才能最终认定当前指令序列实现了同名函数形式的隐式库函数调用,并且调用的目标函数是 `jmp` 指令后存放的字符串数据所对应的库函数。

3) 通过对调用指令相关地址的预分析,发掘调用目标地址与函数名之间的对应关系。

对调用指令相关地址的预分析主要完成两方面的工作。首先,对调用指令的调用目标地址与调用指令所在地址之间的距离进行分析。如果两地址之间的距离小于某一阈值(字符串长度应有一个限制,可设为 50),则接下来再对两地址之间的二进制代码进行分析,判断其是否为一个字符串数据。如果字符串数据确实存在,且该字符串对应某库函数的函数名,则可初步判定该处使用了指令隐含操作以实现隐式库函数调用。

4) 若以上条件皆不满足,则继续对调用指令的目标地址处的二进制代码进行预分析,判断是否该地址处的二进制代码为全零,或者是否能恢复出一个 32 位地址。当以上两种情况满足时,则可初步判定该可执行程序使用了数组实现的库函数调用方法。

要想通过循环代码实现对库函数的重定位,恶意程序必须要保证各个数组之间存在一定的内在联系。通过对一定数量的恶意代码的反汇编代码分析可以发现,同一个库函数在函数名数组与函数装载地址(或偏移量)数组中的相对序号相同。从恶意代码编写的角度来看,这种对应关系使得通过指针的依次移动即可实现对数组元素的取出和赋值操作;从恶意代码分析的角度来看,这种对应关系可以有助于根据调用地址(或偏移量)在相应数组中的相对序号,从函数名数组中找出该地址(或偏移量)对应的库函数的函数名信息。具体实现细节和相关算法将在下面介绍。

### 3. 隐式库函数调用检测技术实现

以下将首先介绍隐式库函数调用行为的检测框架,然后重点介绍数组型隐式库函数调用行为的检测方法。

#### (1) 隐式库函数调用行为检测框架

图 13-14 给出了隐式库函数调用行为检测的基本框架图。从框架图中可以看出,检测框架的输入是初始的反汇编结果,输出的是已检测隐式库函数调用行为的反汇编结果。该结果将被用于构建含有准确的函数调用信息的库函数调用图,为下一步可执行程序函数调用行为安全性分析做准备。

由于数组型隐式库函数调用方法在恶意程序中的使用更为普遍,因此下面将重点以数组型隐式库函数调用行为的检测方法为例,介绍隐式库函数调用行为检测技术的具体实现及相关算法。

#### (2) 基于模板匹配的调用方式识别

经过大量恶意程序的分析后可以发现,存在一些常用的指令序列以实现对数组的初始化功能。以两数组实现库函数调用的病毒 Win32.Aztec 为例,该恶意程序在重定位 API 地址之前使用两条连续的地址传送指令 `LEA` 来获取两数组的首地址:



```
LEA    edi, [ebp+addr1]
LEA    esi, [ebp+addr2]
```

因此，作者实现的 REMARQUE 原型系统将这类指令序列作为特殊的指令模板存入指令序列库，在反汇编过程中对该类指令序列进行识别。为了保证数组首地址识别的正确性，还需要对数组首地址处的二进制代码进行分析。

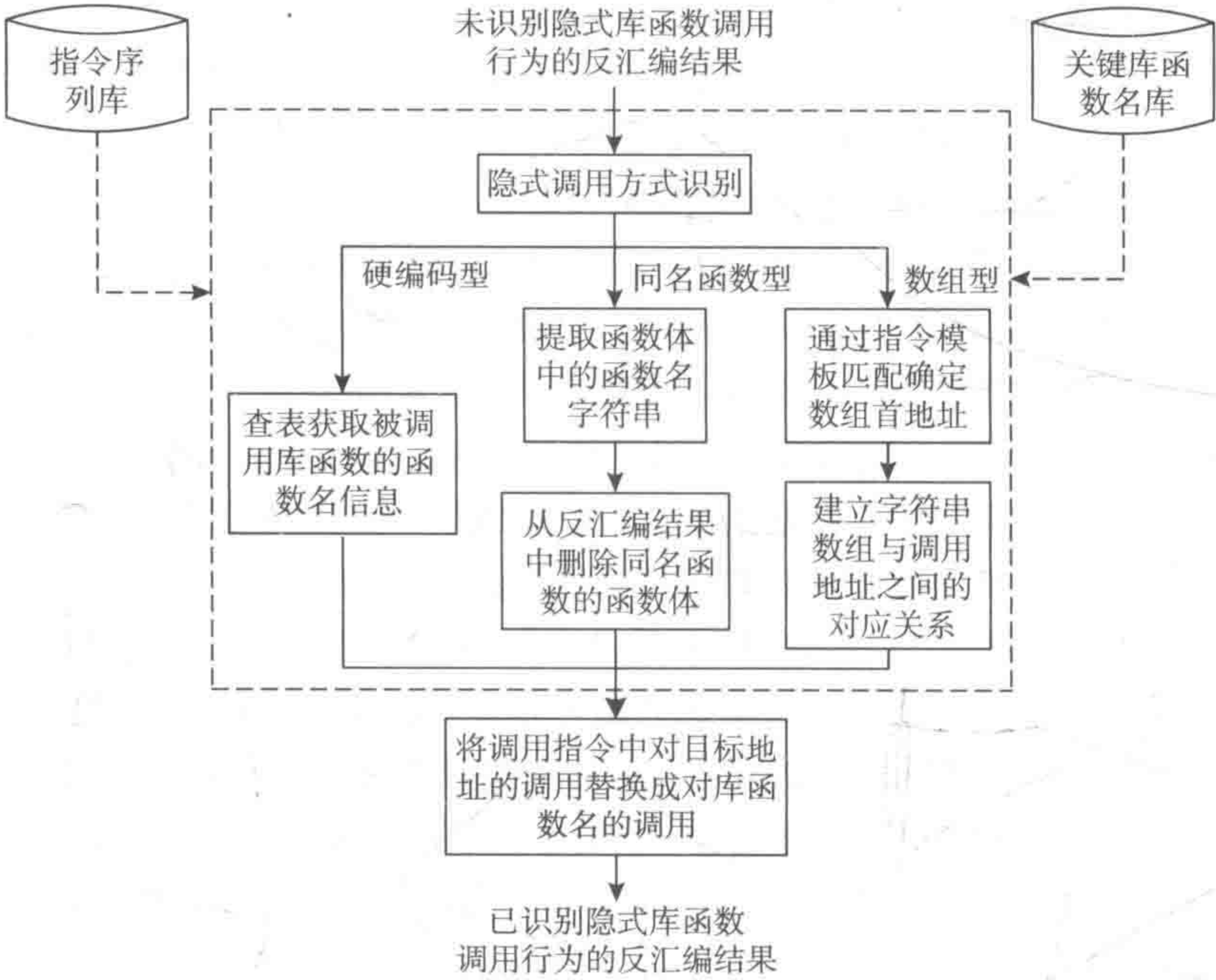


图 13-14 隐式库函数调用行为检测框架

设两条 LEA 指令的源操作数指向的地址分别为 addr1 和 addr2。只有当 addr1 和 addr2 满足以下条件时，才能认定 addr1 为偏移量数组首地址，addr2 为函数名数组首地址：

**条件一** 从地址 addr2 处开始取出连续的若干字节，且使得取出的若干字节中除了最后一字节外其余字节都不为 0，则取出的若干字节分别对应于某个 API 函数名字符串中每个字符的 ASCII 码。

**条件二** 从地址 addr1 处开始的若干字节的值都为零。

除了由双数组实现的隐式库函数调用方法外，还存在由三数组实现的隐式库函数调用方法。由三数组实现的隐式库函数调用行为同样可以通过两条 LEA 指令构成的指令模板进行识别。若设三数组分别为数组 A、数组 B 和数组 C，数组 A 中存放库函数的函数名字符串，数组 B 将在重定位过程结束后用于存放库函数的实际装载地址，数组 C 中则存放的是数组 B 中每一项元素的虚拟地址，三数组之间同样存在“同一库函数在不同数组中对应序号相同”的特征，如图 13-15 所示。



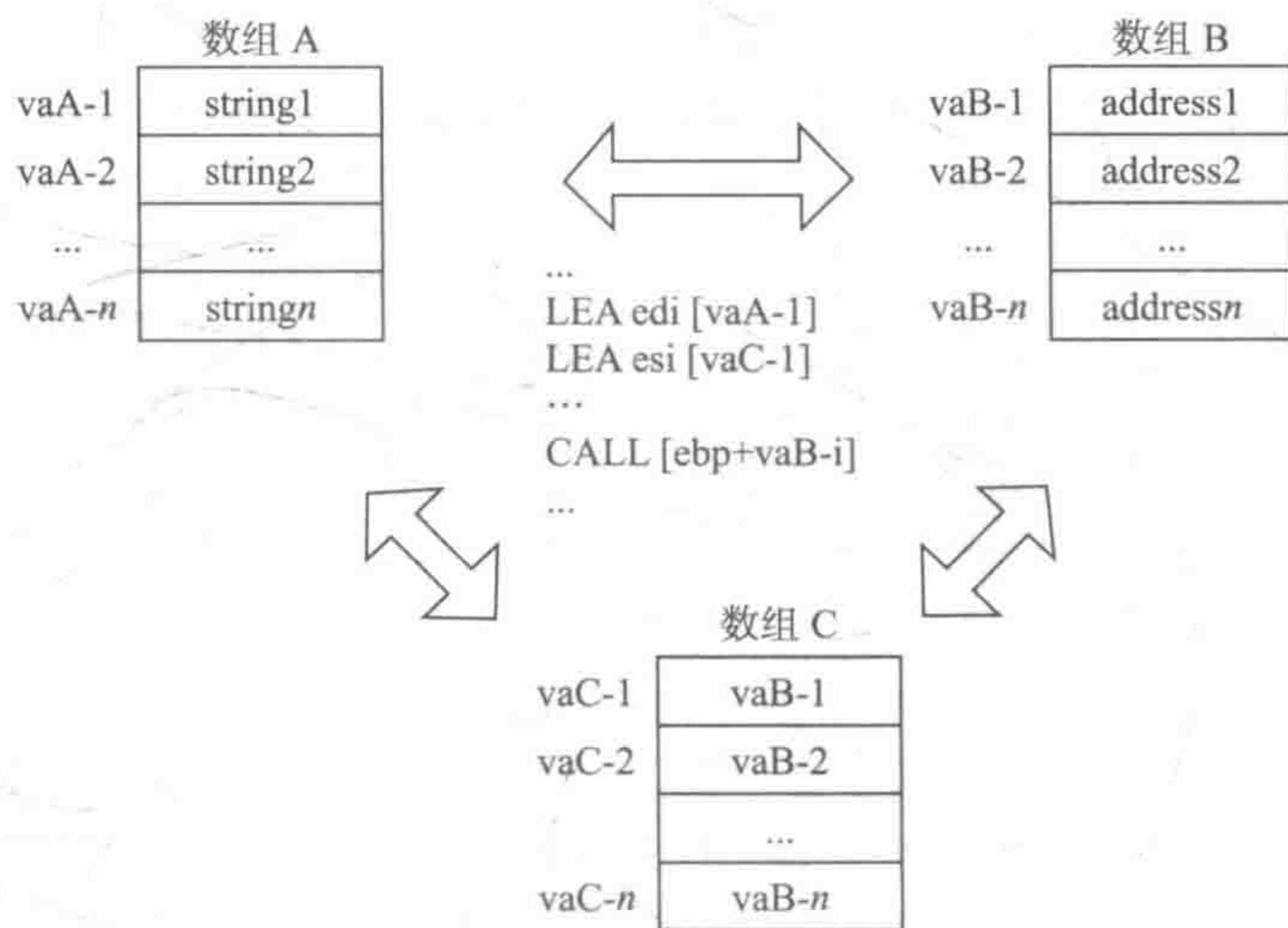


图 13-15 三数组实现的隐式库函数调用

因此，为了能够识别双数组和三数组形式的隐式库函数调用行为，还应该在以上条件一和条件二的基础上再添加一个条件。

**条件三** 从地址 `addr1` 处开始的连续的 4 字节处存放的是一个虚拟地址，且该虚拟地址处的内容为全零。

当条件一和条件二满足时，可判断当前程序使用的是双数组型隐式库函数调用；当条件一和条件三满足时，可判断当前程序使用的是三数组型隐式库函数调用。

### (3) 建立数组间映射关系

从便于恶意代码编写的角度来看，为了方便对 API 函数进行重定位，恶意程序编写者往往会使用循环结构代码来获取 API 函数的实际装载地址。从程序安全性分析的角度来看，只要恶意程序采用循环结构实现 API 函数重定位，就说明数组之间存在固定的对应关系。

经过对使用数组实现隐式 API 函数调用的恶意程序的分析可以发现，同一 API 函数在 API 函数名数组中的序号与其实际装载地址的偏移量在偏移量数组中的序号相同。图 13-16 给出了调用地址到 API 函数名之间的对应关系示意图。

因此，在知道数组首地址的基础上，根据调用指令 `CALL` 的目标地址，可以计算此次调用的 API 函数在偏移量数组中的序号（偏移地址数组的每一项大小为 4 字节）；根据该序号取出 API 函数名数组中相应的字符串，即可得到此次调用的目标 API 函数的函数名（函数名数组中每一项都以字节“00”结尾）。

按照以上分析思路，图 13-17 给出的算法流程将用于建立调用地址到函数名的映射关系。算法的输

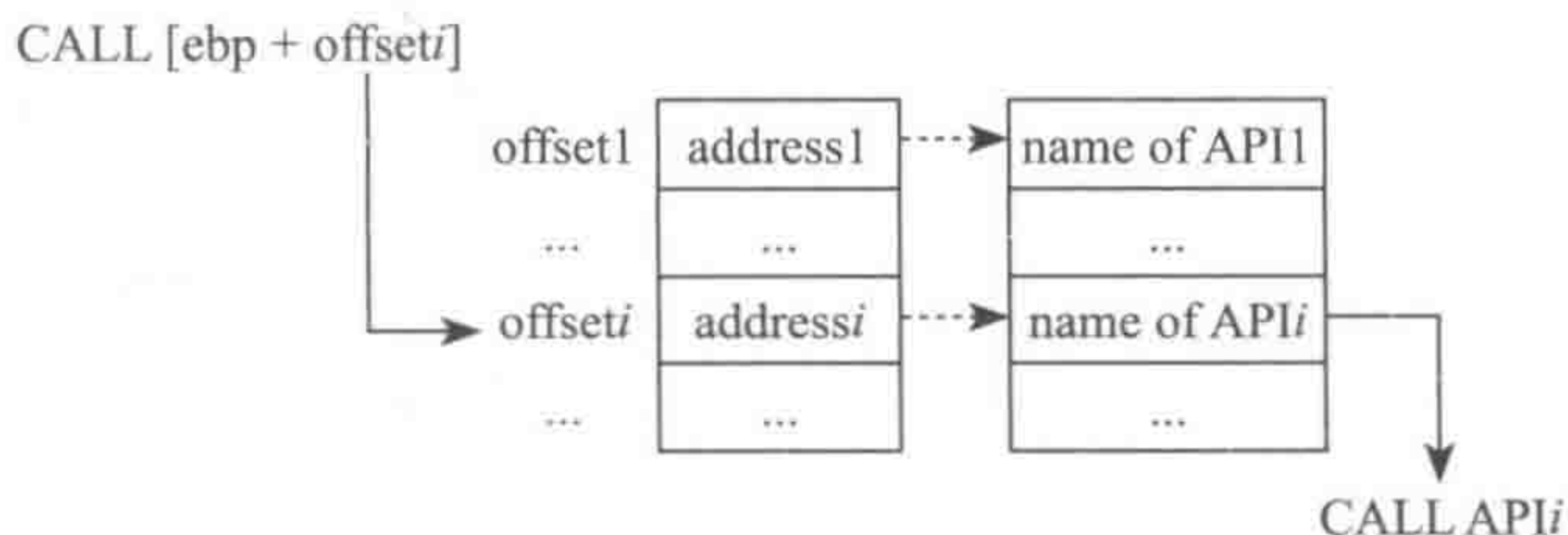


图 13-16 调用地址到 API 函数名之间的对应关系示意图



人为数组 A 和数组 B 的首地址，其中，数组 A 为函数名数组，数组 B 为库函数实际装载地址数组或者是相关偏移量数组；输出为函数名到调用目标地址之间的对应关系表。

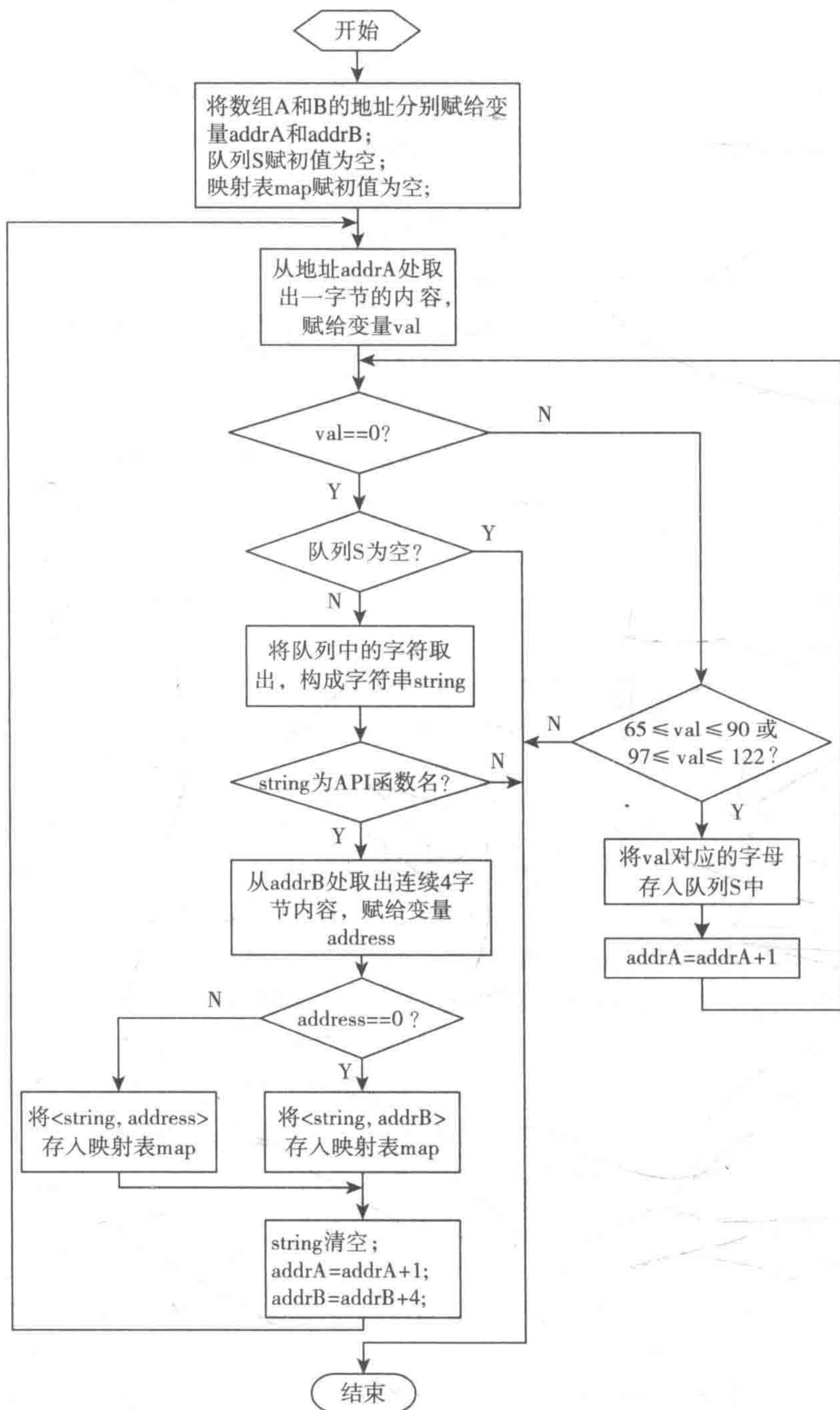


图 13-17 建立调用地址到函数名映射关系的算法流程



然而，图 13-17 给出的分析算法虽然可以有效检测双数组或是三数组型隐式库函数调用行为，但是却无法分析单数组型隐式库函数调用行为。

单数组型隐式库函数调用方法一般会将库函数的实际装载地址存放在数组中，而用于搜索库函数实际装载地址的字符串信息则采用一定的方法隐藏在代码中，比如像 Win32.Velost.1186 病毒那样，利用 CALL 指令的隐含操作在内存中动态形成一个函数名字符串数组。

检测此类型的隐式调用的目标函数需要三步骤来实现。首先，在识别调用形式的基础上，从可执行程序中恢复字符串数据，将恢复出来的字符串数据存储在字符串数组 A 中，数组 A 的每一元素为一个库函数的函数名。

第二步，从可执行程序中恢复出库函数的装载地址数组。由于单数组实现的隐式库函数调用方法对数组的操作不具备典型特征，因此，需要对 CALL 指令的目标地址进行分析，以恢复出地址数组。分析的方法是首先对反汇编结果进行扫描，用数组 B 记录所有调用目标不明的 CALL 指令的目标地址（或目标地址中的偏移量）；接下来，对这些目标地址（或偏移量）按照从小到大的顺序重新排序，则有数组首元素  $B_1$  为数组中元素的最小值， $B_n$  为数组中元素的最大值（此处假设每个库函数调用行为都已经被识别出来，因此，此处只考虑数组 B 中元素个数与数组 A 中元素个数相等的情况）。

第三步，建立数组间对应关系。由于利用指令压栈操作实现的字符串数组，在程序执行读取数组元素操作时采用的是先进后出的读取方式，因此，当调用地址为数组 B 中的第一个元素时，此次调用的库函数的函数名应该存储在数组 A 的末尾处；当调用地址为数组 B 中的最后一个元素时，此次调用的库函数的函数名应该存储在数组 A 的第一个元素的位置。基于以上考虑，在建立数组 A 和数组 B 之间的对应关系时，应该采取相反的读取方向依次读取两数组中的元素，从而保证建立关系的正确性。

#### （4）隐式调用的目标库函数恢复

按照以上算法，即可得到调用指令的目标地址与库函数的函数名之间的映射关系。接下来，可根据该映射表，将反汇编结果中对库函数的隐式调用替换成对库函数的函数名的显式调用形式。

仍以 AZTEC 病毒为例，IDA 的反汇编结果与 REMARQUE 的分析结果比较见图 13-18。分析结果显示，采用 IDA 无法分析出隐式库函数调用行为的调用目标。采用以上策略和相关算法后，REMARQUE 原型系统能够正确地识别出库函数调用行为，并且检测出被调用的目标函数的函数名。

AZTEC 源代码： call [ebp+_FindFirstFileA] IDA 反汇编结果： CODE:004010D2 call ss:dword_401584[ebp] REMARQUE 分析结果： 4010d2: CALL.Evod FindFirstFileA
---

图 13-18 IDA 反汇编结果与 REMARQUE 分析结果的比较

### 13.3 用户自定义过程的数据恢复

由于用户自定义过程的参数和变量具有不可完全恢复性，而过程执行时的语义则是完全可以恢复的，因此在反编译的一项重要应用——二进制翻译方面，在源平台上的任何操



作都可以在目标平台上模拟。在由源机器指令中间表示到目标机器指令的完全语义等价转换过程中,针对源机器指令中使用的数据进行恢复处理。指令中对数据的寻址分为三类:立即数寻址,寄存器寻址,内存寻址。

1) 立即操作数通常蕴涵在指令的中间表示语义中,无需专门处理。

2) 寄存器操作数能够通过将源机器的寄存器直接映射到目标机器的同类型寄存器或在目标机器内存中以模拟的方式保证数据操作的正确性。

3) 内存操作数通常分为两部分:一部分是静态数据,以有组织的数据段的形式存在;另一部分是动态数据,存在于程序动态运行时的堆和栈中。由于堆中的数据是运行时用户调用源机器内存申请库函数动态得到的,不需对其进行数据恢复,因此可以通过直接调用目标平台内存申请函数实现其数据访问功能。

针对寄存器操作数、数据段中的数据和栈内数据的处理,将在下面详细阐述。

### 13.3.1 基于语义映射的数据恢复

源机器程序执行时对于寄存器和数据段中的数据访问,通常是分别直接给出相应的寄存器号和计算出的内存地址。在程序的每次执行时,针对程序相同的输入,这些寄存器号和内存地址是固定不变的。因此,在翻译到目标机器上时就可以采用固定映射的方式,使得映射后指令在语义上是完全等价的。

#### 1. 寄存器映射

源机器的寄存器可以被直接映射到目标机器的寄存器中,也可以映射到目标机器的固定内存中,为了减少频繁访存的开销,在目标机器寄存器较多的情况下一般都是映射到寄存器中,而目标机器寄存器不够时则多数选择将被程序频繁访问的寄存器映射到目标机器的寄存器中。假设我们的目标机器采用 RISC 架构,提供了丰富的寄存器资源,其数量整体超过了 x86\_64 寄存器的个数。因此本节采用直接寄存器映射的方式。

综上所述,x86\_64 中寄存器的使用比较复杂。下面我们介绍具体的映射情况。

##### (1) 通用寄存器

x86\_64 中共有 16 个通用寄存器,其中 callee-saved 寄存器有 6 个分别是 %rbx、%rbp 和 %r12 ~ %r15,如果这些寄存器映射到目标机器寄存器不是 callee-saved 类型,则在调用到库函数时,在库函数内部难免会对这些寄存器的值进行修改,当库函数返回到 caller 时,这些寄存器的值可能已被改变。而在 x86\_64 中,caller 调用库函数前并不保存这些寄存器的值,所以,应当将这类寄存器映射到目标机器的 callee-saved 寄存器,以保证子过程调用前后这些寄存器的值不会被改写。由于 %rbp 在源机器中是可选的帧指针寄存器,所以将其对应到目标机器上相应的可选帧指针寄存器。源机器的其他 5 个 callee-saved 寄存器也被映射到目标机器相应的 callee-saved 类型寄存器中。

x86\_64 通用寄存器中除了 callee-saved 类型的寄存器就是 caller-saved 寄存器。对应 caller-saved 寄存器,由于 caller 在调用 callee 前已经将自己需要的 caller-saved 寄存器保护



起来,这些操作在目标机器上也会被模拟,所以源机器的 caller-saved 寄存器可以映射到目标机器中两类寄存器的任一种。通常先对一些有特殊用途的寄存器进行映射,首先是把 `rsp` 映射到目标机器上有同样功能的栈指针寄存器中;为了减少调用目标机器库函数传参时寄存器间的数据传送次数,分别将 `x86_64` 平台的 6 个整型传参寄存器映射到目标平台的 6 个传参寄存器中;将用于过程调用时保存整型返回值的寄存器 `%rax` 与目标机器保存返回值的寄存器相对应;剩下的源平台的两个暂时寄存器 `%r10` 和 `%r11` 简单地映射到目标机器的两个暂时寄存器中。

实现了寄存器之间的直接映射后,仍需针对源和目标平台对寄存器操作中存在的差异进行相应的处理。`x86_64` 可以访问一个 64 位存储空间的整个 64 位数据,也可以访问它的低 32 位、低 16 位、低 8 位,甚至能够访问某些寄存器的第二字节数据。而在目标机器上只能使用整个 64 位的数据。假设源机器指令的操作数是某寄存器的第二字节数据,针对这种情况,可做以下特殊处理:

- 如果指令的功能是引用操作,首先将该寄存器的值复制到一个临时寄存器中,使用目标机器的寄存器字节清零指令将临时寄存器中第二字节以外的其余字节清零,然后再将该临时寄存器内的数据右移 8 位。此时,对该临时寄存器的引用就等同于源机器对源寄存器的第二字节的数据引用。
- 如果指令的功能是修改操作,则在执行与引用操作同样的一系列处理后,将对源寄存器第二字节的修改操作直接作用到临时寄存器上,然后将临时寄存器左移 8 位,用字节清零指令将除第二字节外的其余字节清零,将源寄存器的第二字节清零,最后将源寄存器和临时寄存器异或,得到的数据保存在源寄存器中。

## (2) 标志寄存器

`x86_64` 中使用扩展到 64 位的标志寄存器 `RFLAGS`,其包含了 9 个标志,可以分为两组:运算结果标志和状态控制标志。其中经常用到的标志位是 `CF`、`PF`、`AF`、`ZF`、`SF`、`IF`、`DF`、`OF`。对于运算结果标志,每条加减运算和逻辑运算都或多或少地对部分标志位产生影响,即影响标志位的定值,对标志位的定值通常是经过硬件自动完成的。而另外一些指令通过特定的标志位状态的判断来决定程序的下一步操作,即标志位引用。

在指令解码阶段生成中间表示时为能够产生标志位定值和对标志位引用的指令添加了标志位修改和引用操作的中间表示,因此当这些中间表示翻译到目标机器上运行时需要在目标机器上保存标志位的信息。由于目标机器的 RISC 架构并不包含标志位寄存器,因此需要指派专门的寄存器或内存空间来模拟标志位信息。由于运算和逻辑指令使用的频繁性,使用内存模拟将会大大降低程序的效率,因此使用一个临时寄存器来保存经常用到的 8 个标志位信息,每个标志位占用寄存器的一字节,如果要对某个标志位进行修改或引用,则通过 `and` 或 `or` 操作设置或提取寄存器相对应字节的数据,从而完成了寄存器的模拟。通常一次标志位修改和引用只需要一到两个目标机器寄存器操作指令就能完成。

## (3) 其他寄存器

对于指令指针寄存器,因为在指令解码时已经将使用指令指针寄存器偏移寻址的地址



分析转化为固定地址，所以无需在目标机器上模拟；对于段寄存器和控制寄存器，由于我们处理的是 x86\_64 的应用程序，一般不会遇到对这些寄存器的访问，所以无需处理；而对于 x86\_64 中用于浮点操作的 xmm 寄存器，则可以直接对应到目标机器的浮点寄存器中。

## 2. 数据段映射

由于翻译的程序是 x86\_64 平台下的 ELF64 格式可执行文件，而 ELF64 格式可执行文件可以看作程序头部描述的一系列段的集合，因此这些段在程序执行时一般会被映射到内存空间中。通常程序访问的静态数据分别保存在只读数据段 .rodata、初始化数据区 .data 和未初始化数据段 .bss 中。

在 ELF64 文件的程序头部中为每个段保存段描述符，段描述符中记录了段类型、在文件中的偏移量、需要被映射到的虚拟地址和段的大小等信息。根据段描述能够将相应的段数据提取出来，由于段在源平台二进制程序中被映射到固定的虚拟空间，程序对段数据的访问也通过虚拟地址给出，所以，如果将提取出的段数据直接映射到目标机器可执行程序相同的虚拟空间中，就可以让翻译得到的目标代码根据虚拟地址直接访问这些数据。

目标机器的最低 4GB 空间可提供给用户程序自由映射，虽然限制了翻译的程序不能使用高于 4GB 的虚拟空间，但幸运的是，目前大部分用户程序都不会用到高于 4GB 的虚拟地址。

因此，在我们实现的翻译器中，通过分析待翻译的源平台 ELF64 可执行文件，将 .rodata 和 .data 数据段数据提取出来并分别保存到两个数据文件中，然后在生成目标机器可执行文件时，通过编写链接脚本文件，将这两个段数据和 .bss 段大小作为目标文件的特定数据段，这样当目标文件被执行时，这些数据段会被自动加载到相应的内存空间中，从而保证了翻译生成的目标指令访问静态数据的正确性。

### 13.3.2 基于栈帧平衡的数据恢复

当过程调用传递的参数超过一定数量时需要使用栈来传参，在子过程中若要访问这些参数，可以通过栈指针加偏移的形式。在 x86\_64 平台上，当 caller 将参数入栈后调用 callee 时，操作系统自动在栈内压入一个 8 字节的返回地址，该地址在 callee 返回时自动弹出。而目标机器在过程调用时使用硬件寄存器保存返回地址的值，在过程调用时不会自动改变栈指针的值。因此，在目标平台上模拟的栈指针的值到参数的偏移就会比源平台上的栈指针到参数的实际偏移少 8 字节，当目标平台仍然使用源平台给出的偏移值访问通过栈传递的参数时就会出现差错。

出错的原因是机器特性带来的栈帧偏差，因此本节提出了一个栈帧平衡的解决方法，在 callee 的开始和结束处添加一个调整堆栈的动作，分别使栈指针减 8 或加 8。从而使得目标机器上的栈帧与源机器上的栈帧保持平衡，保证目标机器上对形式参数和局部变量访问时地址的正确性。



目标平台上提供的过程调用时自动保存返回地址的寄存器是一个 callee-saved 类型的寄存器。如果在 callee 中没有将该寄存器保存起来，而恰恰 callee 中又调用了其他用户自定义过程，这样在 callee 中被调用的过程正确返回时，callee 自身的返回地址却被覆盖，无法保证正确返回到 caller 中。因此，需要在 callee 开始处将该返回地址寄存器中的值保存起来，而前面为了平衡栈帧在 callee 开始处对目标机器栈指针进行了加 8 操作，由于返回地址寄存器也保存了 8 字节的返回地址，所以完全可以用将返回地址寄存器的数据压栈的动作替代上述栈指针的调整。

因此，最终的栈帧平衡方案是：在目标平台上遇到用户自定义的过程调用时，在 callee 的开始处额外地添加一个将返回地址寄存器的值压栈的动作，在 callee 返回前，应添加一个从栈中弹出一个 8 字节数据到返回地址寄存器的动作。

## 13.4 用户函数与库函数同名的区分

针对绝大部分二进制可执行程序，应用库函数快速识别技术都能得到令人满意的正确结果，但是当二进制可执行程序包含与库函数同名的用户函数时，就可能会出现问題。如果用户编写的同名函数与相应的库函数等价，将其识别成库函数没有问题；但如果两者实现的功能不等价，若仍把其识别成库函数就会导致无法预测的后果，因此，有些软件将此作为一种反反编译的混淆策略。迄今为止在反编译领域对用户函数与库函数同名的区分问题进行的研究相对较少。

### 13.4.1 函数同名问题

#### 1. 函数同名问题描述

如果用户函数与系统库函数同名，则会出现两种情况：两者功能等价或者两者的输出结果不同。对于反编译而言，第一种情况因为功能等价，所以把同名的用户函数当成相应的库函数处理完全可以；但对于第二种情况，如果仍把其当成相应的库函数进行处理就改变了程序的原本语义，这样可能导致对整个程序的移植失败，因为对此函数调用的返回结果与预期结果不一致。下面使用两个具体的例子加以说明。

**例 13.1** 用户函数 *floor* 与数学库函数 *floor* 功能等价。

```
#include "math.h"
double floor(double t)
{
    double m;
    if(t>0)          m=(int)t;
    else if(t<0)     m=(int)(t-1);
    else             m=t;
    return m;
}
```



```

}
int main()
{
    double a,b;
    printf("input a= ");
    scanf("%lf",&a);
    b=floor(a);
    printf("***** %lf\n",b);
    return 1;
}

```

此程序的输入输出结果对如下，格式为“<输入数据、输出数据>”：

<35.784, 35.000000>, <0, 0.000000>, <-6.138, -7.000000>, ...

对此程序的可执行程序进行反编译后仅得到一个 main.c 源程序文件，因为把用户函数 *floor* 处理成了数学库函数 *floor*，但得到的结果仍然为：

<35.784, 35.000000>, <0, 0.000000>, <-6.138, -7.000000>, ...

**例 13.2** 用户函数 *floor* 与数学库函数 *floor* 的功能不等价。

```

#include "math.h"
double floor(double t)
{
    double m;
    m=t+1.5;
    return m;
}
int main()
{
    double a,b;
    printf("input a= ");
    scanf("%lf",&a);
    b=floor(a);
    printf("***** %lf\n",b);
    return 1;
}

```

此程序的输入输出结果对格式为“<输入数据、输出数据>”：

<35.784, 37.284000>, <0, 1.500000>, <-6.138, -4.638000>, ...

对此程序的可执行程序进行反编译后也仅得到一个 main.c 源程序文件，同样把用户函数 *floor* 处理成了数学库函数 *floor*，但得到的结果却与源程序的输出结果大不一样：

<35.784, 35.000000>, <0, 0.000000>, <-6.138, -7.000000>, ...

## 2. 函数同名问题分析

按上文描述，参考库函数快速识别技术无法把与库函数同名的用户函数同相应的库函



数区分开来,且无法保证待反编译的二进制可执行程序包含的同名用户函数同库函数功能等价,如例 13.2 所示,所以才会导致这样的问题。

表面上看来,原因在于反编译系统无法将与库函数同名的用户函数识别出来;但从更深的层面看,真正的原因是现有的库函数识别技术存在严重的缺陷。

为了解决该问题,研究了 IA64 体系结构关于动态链接实现的约定,结合实例分析了惰性绑定机制的实现过程,进而研究了 ELF64 中 .plt 段的布局规律,同时分析了 .plt、.got、.IA\_64.pltoff、.rela.IA\_64.pltoff 和 .dynstr 等段之间的相互联系,提出了由 .plt 段中的 *import stub* 地址反向推出对应的函数名的步骤和方法,并给出了反编译系统中所使用的库函数名的识别算法,该算法可在二进制文件解码阶段建立库函数的 *import stub* 地址与函数名之间的一一映射关系,构建名字-值对并将其加入反编译系统符号表,以达到通过调用指令中的 *import stub* 地址就能够找到函数名的目的。下面通过一个例子加以说明。

**例 13.3** 包含库函数 *sin* 和用户函数 *cos* 的示例程序 *cos\_sin-e.c*。

```
#include <math.h>
float cos(float i);
int main()
{
    float b;
    b = sin(0.5);
    b = cos(0.6);
    printf("cos 0.6 is %f\n", b);
    return 0;
}
float cos(float i)
{
    i = i * i;
    return i;
}
```

本节对程序 *cos\_sin-e.c* 的各种操作,都是对其在 IA64 机上使用 GCC 编译器生成的二进制可执行码而进行的。

在此程序的 *objdump* 代码中,函数 *sin*、*cos* 和 *printf* 与相应的函数调用语句对比关系如下:

```
...
br.call.sptk.many b0=40000000000000560 <sin>;
...
br.call.sptk.many b0=400000000000008f0 <cos>;
...
br.call.sptk.many b0=40000000000000540 <printf>;
...
```

此程序对应的动态链接符号如下:

DYNAMIC SYMBOL TABLE:



```

0000000000000000      DF *UND* 000000000000000b0 GLIBC_2.2  printf
0000000000000000      DF *UND* 00000000000000040 GLIBC_2.2  sin
400000000000008f0    g  DF .text 00000000000000060      Base  cos
0000000000000000    w  D  *UND* 00000000000000000 _Jv_RegisterClasses
0000000000000000      DF *UND* 00000000000000280 GLIBC_2.2  __libc_start_main
0000000000000000    w  D  *UND* 00000000000000000      __gmon_start__

```

另外，该程序对应的 .plt 段内容组织如下：

```

400000000000004c0 <.plt>:
400000000000004c0:      ***
400000000000004c6:      ***
***
400000000000005b6:      ***
400000000000005bc:      ***

```

基于以上三组数据进行分析可得，库函数 `sin` 和 `printf` 的调用地址分别为 `0x40000000000000560` 和 `0x40000000000000540`，两者都处于 .plt 段的地址范围之内，在动态链接符号表中两者对应的地址都为零，正好符合引入的库函数识别技术的识别范围，于是构建名字-值对 `<sin, 0x40000000000000560>`、`<printf, 0x40000000000000540>` 并将其加入反编译系统符号表；但对于用户函数 `cos`，由于其在动态链接符号表中对应的地址是 `0x400000000000008f0`，此地址不为零且函数 `cos` 的调用地址同样为 `0x400000000000008f0`，所以反编译系统就直接将构建的名字-值对 `<cos, 0x400000000000008f0>` 加入系统符号表中。

在随后解码过程中，解码 `main` 函数时会先后遇到与函数 `sin`、`cos`、`printf` 相应的函数调用，于是先把这些函数调用地址依次存入函数调用链中，在 `main` 函数解码完成后根据函数调用链再对相应的被调函数函数体进行解码。解码之前要先判断该函数是否已被解码过，如果已解码就不再对其进行解码；另外，还要判断该函数是不是库函数，即需不需要对此函数的函数体进行解码，如果是库函数就不再对其进行解码。

此时的反编译系统对库函数的判定依据是：根据函数调用地址到系统符号表中查找匹配的名字-值对，从而得到被调用函数的函数名，然后再根据函数名判断此函数是不是库函数。针对例 13.3 而言，函数调用地址 `0x40000000000000560`、`0x400000000000008f0` 和 `0x40000000000000540` 对应的函数名字-值对分别为 `<sin, 0x40000000000000560>`、`<cos, 0x400000000000008f0>` 和 `<printf, 0x40000000000000540>`，因此可知 `main` 函数调用的三个函数是 `sin`、`cos` 和 `printf`，从而根据函数名判定这三个函数都是库函数，进而不再对其函数体进行解码。这样库函数 `sin` 和 `printf` 得到了正确的识别，而用户函数 `cos` 却被错误地识别成了系统库函数。

把与库函数同名的用户函数识别成库函数的主要原因是：对库函数的判定仅依据函数名进行，而编译器在处理与库函数同名的用户函数时又将其放入了动态链接符号表中。

### 13.4.2 函数同名解决实例

根据之前描述的库函数识别技术和 ELF64 格式二进制文件组织策略的具体分析，可知



要解决用户函数与库函数同名的区分问题，应把突破点放在以下两方面：1) 与库函数同名的用户函数和库函数在 .text 段中所处位置的差别；2) 二进制文件中的动态链接符号表。

依据与库函数同名的用户函数和库函数在 .text 段中所处位置的差别，完全可以解决同名问题。因为编译器 GCC 在处理函数时有明确的标准：库函数都使用动态链接进行链接处理，对库函数的调用地址都处于 .plt 段的地址范围之内；用户函数都使用静态链接进行链接处理，对函数的调用地址都大于 .plt 段的段尾地址。因此，对于由 GCC 编译生成的二进制可执行程序可以按照调用地址与 .plt 段地址之间的关系进行用户函数和库函数的区分。对由 icc 编译器编译生成的二进制可执行程序处理起来就很麻烦，因为 icc 是 Intel 公司针对 Intel 系列机器发布的编译器，由于商业竞争的原因它不像编译器 GCC 那样标准，而是采取了很多深度的编译优化策略。例如，icc 在处理库函数时对小部分库函数像 GCC 那样采用动态链接的处理策略，而对大部分的库函数则采用静态链接的处理策略，更是对一部分的库函数则直接在静态链接的基础上又采用内嵌优化策略来提高程序的运行速度。虽然编译器 icc 在处理库函数时极不规范，但并不是无章可循，由 icc 编译生成的二进制可执行程序中用户函数代码和库函数代码都是集中存储的，且在两种代码交界处有一个标志性符号 “DwArFil\_HiGh\_Pc”，据此可对由 icc 编译生成的二进制可执行程序按照函数调用地址、.plt 段地址以及标志性符号 “DwArFil\_HiGh\_Pc” 对应地址之间的关系进行用户函数和库函数的区分：当函数调用地址处于 .plt 段尾地址和标志性符号 “DwArFil\_HiGh\_Pc” 对应地址之间时，此函数为用户函数；否则为库函数。

上述方案深度依赖于 ELF64 文件中的符号表，特别是标志性符号 “DwArFil\_HiGh\_Pc”。当 ELF64 文件经过 strip 等瘦身工具处理后，其中绝大多数符号信息都不复存在，包括符号 “DwArFil\_HiGh\_Pc”，仅留下了动态链接符号表中的少量信息，因为这些符号信息在动态链接的时候需要使用。庆幸的是动态链接符号表中保存了库函数及与库函数同名的用户函数的一些信息，下面基于二进制文件中的动态链接符号表给出一种通用的用户函数与库函数同名的区分算法。

### 1. 基于动态链接符号表的函数同名问题分析

本节对用户函数与库函数同名问题的分析仍是基于例 13.3 程序 cos\_sin-e.c 进行的。使用反汇编工具 objdump 显示的动态链接符号表信息与符号表存储结构是一一对应的。例 13.3 示例程序经 icc 编译器编译生成的二进制可执行程序对应的信息组如图 13-19 所示，经 GCC 编译器编译生成的二进制可执行程序对应的信息组如图 13-20 所示。

结合符号表存储结构及图 13-19 和图 13-20 给出的动态链接符号表信息，可以很明显地看出：*st\_name* 列对应的是函数及例程的名字；*st\_value* 列对应的是与函数或例程相关的地址；*st\_info* 列对应的是与函数或例程相关的符号类型信息和绑定属性；*st\_shndx* 列表明 *st\_name* 列给出的符号是在哪个段中定义的，“\*ABS\*”代表此符号是绝对符号且它的值为 0xFFFF1，“\*UND\*”代表此符号是未定义的符号且它的值为 0，对于其他项（如 .text）所具有的值要视具体情况而定，但肯定不为 0；*st\_size* 列表明与函数或例程符号相关的空间



大小值；“区分标志”列给出的信息表明相关函数或例程所处的位置信息，GLIBC\_2.2 表示对应的函数或例程来自动态链接库文件，Base 表示对应的函数或例程来自于该 ELF64 文件的 .text 段。

1. 函数 sin、cos 和 printf 与相应的函数调用语句对比关系如下：

```
...
br.call.sptk.many b0=4000000000000920    <sin>;
...
br.call.sptk.many b0=40000000000001240    <cos>;
...
br.call.sptk.many b0=40000000000000900    <printf>;
...
```

2. 对应的动态链接符号表与符号表数据成员对应关系如下：

DYNAMIC SYMBOL TABLE:

st_value	st_info	st_shndx	st_size	区分标志	st_name
40000000000001380	g	DF .text	0000000000000020	Base	__ashldi3
60000000000002b10	g	DO *ABS*	0000000000000000	Base	__DYNAMIC
00000000000000000		DF *UND*	00000000000000b0	GLIBC_2.2	printf
40000000000001ac0	g	DF .text	00000000000001c0	Base	__umoddi3
40000000000001580	g	DF .text	0000000000000180	Base	__udivdi3
00000000000000000		DF *UND*	0000000000000040	GLIBC_2.2	sin
***					
40000000000001240	g	DF .text	0000000000000060	Base	cos
***					
40000000000001340	g	DF .text	0000000000000040	Base	__muldi3
00000000000000000		DF *UND*	00000000000001e0	GLIBC_2.2	__cxa_finalize
00000000000000000		DF *UND*	0000000000000280	GLIBC_2.2	__libc_start_main
00000000000000000	w	D *UND*	0000000000000000		__gmon_start__

3. 对应的 .plt 段内容组织如下：

```
40000000000000840 <.plt>:
40000000000000840:    ***
40000000000000846:    ***
***
40000000000000a16:    ***
40000000000000a1c:    ***
```

图 13-19 例 13.3 示例程序经 icc 编译对应的信息组

遗憾的是从动态链接符号表中得不到“区分标志”列显示的信息，推测此列信息可能是由其他列中的信息综合得到的。虽然如此，仍可以根据 *st\_name*、*st\_shndx* 和 *st\_size* 三列信息最终判定出哪些函数为库函数，哪些函数是用户函数。比如对用户函数 cos 的识别工作继续往下进行，当 main 函数解码完成并依据函数调用链对被调用地址 0x400000000000008f0 或 0x40000000000001240 所对应的 cos 函数体进行是否解码判定时，根据系统符号表中的名字-值对。



<cos, 0x400000000000008f0> 或 <cos, 0x40000000000001240>

1. 函数 sin、cos 和 printf 与相应的函数调用语句对比关系如下:

```

...
br.call.sptk.many b0=40000000000000560    <sin>::
...
br.call.sptk.many b0=400000000000008f0    <cos>::
...
br.call.sptk.many b0=40000000000000540    <printf>::
...

```

2. 对应的动态链接符号表与符号表数据成员对应关系如下:

DYNAMIC SYMBOL TABLE:

st_value	st_info	st_shndx	st_size	区分标志	st_name
0000000000000000	DF	*UND*	00000000000000b0	GLIBC_2.2	printf
0000000000000000	DF	*UND*	0000000000000040	GLIBC_2.2	sin
400000000000008f0	g	DF .text	0000000000000060	Base	cos
0000000000000000	w	D *UND*	0000000000000000		_Jv_RegisterClasses
0000000000000000	DF	*UND*	0000000000000280	GLIBC_2.2	__libc_start_main
0000000000000000	w	D *UND*	0000000000000040		__gmon_strat__

3. 对应的 .plt 段内容组织如下:

```

400000000000004c0 <.plt>:
400000000000004c0:    ***
400000000000004c6:    ***
***
400000000000005b6:    ***
400000000000005bc:    ***

```

图 13-20 例 13.3 示例程序经 GCC 编译对应的信息组

得到与其相对应 *st\_shndx* 和 *st\_size* 值, 如果函数调用地址不在 .plt 段的地址范围中且 *st\_shndx* 和 *st\_size* 的值都大于 0, 则此函数为用户函数同时对其函数体进行解码, 否则此函数为库函数。

## 2. 基于动态链接符号表的同名问题区分算法

基于上文的分析, 以下将给出基于动态链接符号表的用户函数与库函数同名的区分算法。在给出区分算法之前先定义并建立一个与动态链接符号表相对应的不完全符号表 Table\_Dynamic, 该表包含的数据成员有 *st\_name*、*st\_shndx* 和 *st\_size*, 具体组织形式如图 13-21 所示。

用户函数与库函数同名的区分算法描述如图 13-22 所示。

在系统中应用上述同名区分算法后, 系统对库函数的识别能力得到极大的提高, 有效消除了与库函数同名的用户函数对翻译正确性的影响。正确识别了所有与库函数同名的用户函数, 像 *sin*、*cos*、*sqrt*、*log*、*exp*、

st_name	st_shndx	st_size
Printf	*UND*	00000000000000b0
sin	*UND*	0000000000000040
cos	.text	0000000000000060

图 13-21 不完全符号表 Table\_Dynamic 组织结构



*pow* 及 *random* 等；另外，对测试集 IEEE 浮点测试软件中测试用例 *paranoia* 包含的用户函数 *random* 进行了成功的识别。

```

输入：函数调用地址 Addr、系统符号表 Table_Sys 和符号表 Table_Dynamic
输出：与地址 Addr 相应的函数是否为库函数，是库函数返回 1，否则返回 0
begin
    根据调用地址 Addr 在系统符号表 Table_Sys 中查找与其匹配的名字-值对，从而得到要调用的函数名；
    根据得到的函数名在符号表 Table_Dynamic 中查找与其相关的 st_shndx 和 st_size 值；
    依据 Addr、st_shndx 和 st_size 的值进行判定
    if 编译器为 gcc then
        begin
            if 地址 Addr 处于 .plt 段的地址范围中 then
                返回 1；
            else begin
                返回 0；
            end
        end
    else if 编译器为 icc then
        begin
            if 地址 Addr 不在 .plt 段的地址范围中且 st_shndx 和 st_size 的值都大于 0 then
                返回 0；
            else begin
                返回 1；
            end
        end
    else begin
        返回 1；
    end
end
end

```

图 13-22 用户函数与库函数同名的区分算法

综上所述，本节首先指出库函数快速识别技术在区分库函数和与库函数同名的用户函数方面存在的缺陷，并在深入分析库函数快速识别技术、GCC 和 icc 编译器在处理库函数及与库函数同名的用户函数时所采用的编译策略的基础上，提出了判定库函数和同名用户函数的区分算法。在设计算法的过程中考虑到各种可能出现的情况，另外还特别考虑了 *strip* 等瘦身工具的使用对区分算法的影响。

## 13.5 本章小结

本章首先介绍了过程抽象和调用约定分析，然后介绍了库函数恢复方面的方法和技术，紧接着介绍了用户自定义过程的数据恢复技术，最后剖析了用户函数和库函数同名的区分技术。



## 反编译在信息安全方面的应用实践

### 14.1 反编译在信息安全中的应用

静态分析与动态分析在信息安全领域中有着诸多应用。静态分析主要包括特征码扫描技术、启发式分析技术以及反编译技术。其中，反编译技术在静态分析中占有举足轻重的地位。

Cristina Cifuentes 在其博士论文中将反编译划分为 7 个阶段，如图 14-1 所示，包括语法分析、语义分析、中间代码生成、控制流图生成、数据流分析、控制流分析以及高级代码生成。对于恶意代码检测而言，高级代码生成阶段可以忽略。Khaled Yakdan 等人设计了 REcompile 反编译框架，对 Cristina Cifuentes 的控制流分析算法进行扩展，包括复制退出基本块以减少 goto 语句的数量，通过添加循环路径中节点到循环起始节点的可达性判断提高循环结构的识别率。

由于反编译技术能够从语义角度对代码行为进行刻画，所以是当前静态分析技术的研究热点。

J. Bergeron 等人提出了一种基于反编译的静态分析方法。该方法首先对二进制代码进行反汇编，并转换为高级中间表示形式，通过控制流分析与数据流分析对中间表示进行优化，然后基于控制流图与数据流图采用静态切片提取可疑代码片段，最后对提取得到的可疑代码片段进行分析并判断是否存在恶意行为。

不同于正常代码分析，恶意代码为了提高自身的生存能力，经常采用混淆技术影响静态分析，阻碍反汇编等分析过程。Andreas

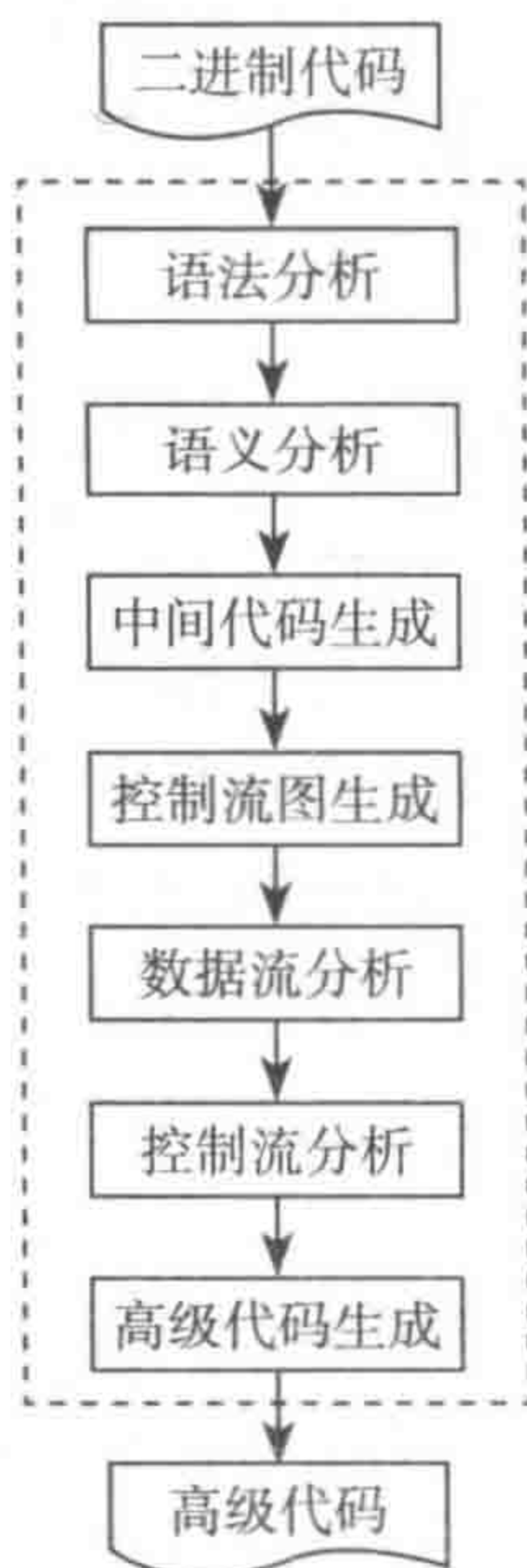


图 14-1 反编译流程图



Moser 等人分析了静态分析方法在恶意代码检测中的局限性。例如,借助不透明常量,可以将无条件跳转指令与调用指令的目标地址用不透明常量表示,实现控制流混淆;将寄存器中的常量地址替换为不透明常量,实现数据存储混淆;将 define-use 链中的常量替换为不透明常量,实现数据使用混淆。因此,为了得到准确的分析结果,需要围绕混淆技术进行专门的研究。

### 14.1.1 反编译技术的优势

静态分析方法不运行被分析程序,而是通过检测分析程序代码来确定程序执行特征。静态分析方法用于信息安全领域,实现对代码的逆向分析,理解代码的实现意图。

作为主要的静态分析方法,反编译技术主要具备以下优势:

1) 反编译技术能够对程序进行全面的分析。因为在分析过程中,反编译技术可以不受限于程序的某一条执行路径,能够实现对程序的所有执行路径进行分析。而动态分析仅能够对与所选择分支相对应的执行动作进行分析。

2) 反编译技术无需程序的执行。如果被分析代码是恶意代码,那么该代码的执行可能影响系统安全,反编译技术则能够在程序执行之前完成对程序的分析,而程序不执行则不会对系统造成潜在的威胁。

3) 反编译技术具有更细的分析粒度。反编译分析技术不仅能够发现程序中是否含有恶意行为,而且能够对程序中含有的恶意行为进行定位,从而方便安全检测人员对程序进行分析。

4) 反编译技术具有更高的执行效率。同执行监控、污点跟踪、多路径分析等动态分析技术相比,反编译技术在分析过程中占用的空间更少、分析速度更快,具有更高的执行效率。

### 14.1.2 代码恶意性判定

代码恶意性判定即恶意代码检测,其是反编译技术在信息安全领域中的重要应用。恶意代码行为阻断、入侵检测、攻击预警等工作都以代码恶意性判定为基础。但是,判定一个程序是否具有恶意性是一项较为困难的任务,Fred Cohen 已经证明了不存在一种检测方法能够检测出所有的恶意程序;也不存在一种方法可以检测出某一个特定恶意程序的所有变体。传统的进行代码恶意性判定的方法主要有特征码方法、基于程序完整性的方法、基于程序行为的方法,以及基于程序语义的方法。近年来又出现了许多新型的检测方法,如基于数据挖掘的方法、基于生物免疫的检测方法以及基于人工智能的方法等。各种判定方法的侧重点有所区别,有的侧重于提取判定依据,为判定提供全面、准确的支撑,有的侧重于设计判定模型,能够高效地区分恶意代码和正常代码。

反编译技术通过语义提升的方法将机器指令流变换为语义等价的高级表示形式,从而针对使用高级表示形式的指令流进行数据流分析和控制流分析,达到充分理解可执行程序



行为的目的。因此反编译技术能够非常全面地了解 and 掌握目标程序在执行过程中将要执行的操作行为,这对于分析目标程序中是否包含威胁计算机系统安全的恶意代码有很大的帮助。与以往基于特征码匹配和模型检测的方法相比,基于反编译逆向分析技术的恶意代码检测方法有着非常明显的优势:

1) 基于特征码匹配的恶意代码检测方法必须基于一套恶意代码特征库,通过在目标程序中查找和匹配已知的机器指令序列来检测恶意代码。但是恶意代码的作者往往能够通过对于源代码进行变形的办法生成恶意代码变体,从而有效地避开基于模板匹配的检测工具,达到有效隐藏自己的目的。基于反编译逆向分析技术的恶意代码检测方法检测的目标不是指令序列,而是指令版段执行的行为,从而有效地检测出系统中存在的具有相似行为的恶意代码及其变体。

2) 基于模型检测的恶意代码检测方法虽然也是针对代码行为进行检测,但是对于经过加密的恶意代码则无能为力。基于反编译逆向分析的恶意代码检测方法能够将恶意代码中自带的解密程序进行分析,将其提升为容易被人理解的高级表示形式,从而便于技术人员利用相关信息对加密代码进行分析,有效地确认并定位加密代码中存在的恶意行为。

### 14.1.3 代码敏感行为标注

代码敏感行为标注是反编译技术在信息安全领域中的深入应用。传统的安全软件在保障计算机安全方面具有非常重要的地位,对于具有恶意行为的目标程序,传统安全软件的处理方法却显得非常“鲁莽”,即对于已经确认的恶意行为予以查杀操作,而对于可疑行为仅仅给出非常粗浅的警告信息。但是在对安全性和健壮性有很高要求的计算机系统当中,操作者不仅需要消除系统中存在的每一个已经确认的安全隐患,还需要对可疑但未经确认的代码进行精确定位,并且希望安全软件提供能够有效帮助操作人员分析相关代码版段的信息。通过在反向编译的结果中标注可疑代码能够帮助操作人员达到这个目的,并且可以依据可疑代码版段与恶意行为模型的吻合程度来给出相应的警告级别,从而帮助操作人员构建更加稳定和安全的计算机系统。

在反编译分析的基础上,能够对代码敏感行为进行标注。恶意行为模型库中记录了基于反编译结果描述的恶意行为特征。可疑程序经过反编译模块的处理生成与反编译各层对应的程序分析结果,再根据恶意行为库中的行为特征,采用对应的检测机制,对分析结果进行标注。

### 14.1.4 恶意代码威胁性评估

恶意代码威胁性评估是反编译技术在信息安全领域中的最新应用。恶意代码威胁性评估即通过静态分析、动态分析等方法对恶意代码进行深入、全面、准确的分析,刻画其在实际计算环境中对信息系统、用户所构成威胁的大小。在恶意代码分析领域中,二进制代码恶意性判定同恶意代码威胁性评估都是其中的重要内容,但是两者在分析角度、分析粒



度、分析依据等内容上存在区别。

二进制代码恶意性判定：

- 1) 恶意性判定主要考虑二进制代码是否存在恶意行为，而无需考虑恶意行为的实现方式。
- 2) 恶意性判定的依据相对较为客观，主观因素参与程度有限。

二进制恶意代码威胁性评估：

1) 威胁性评估主要考虑恶意代码行为的实现方式，具体行为的不同实现方式对威胁性评估有影响。

2) 在威胁性评估过程中针对恶意代码类型的不同，评估指标也随之变化，在评估指标的设置上允许一定程度的主观因素参与。

3) 威胁性评估在评估过程中有独特的性质。例如，对于评估恶意代码某种关键属性的威胁程度，虽然恶意代码可能存在多种实现方式实现该属性，但是该属性的威胁程度最终由能力最强的实现方式所决定。

恶意代码威胁性评估是恶意代码分析中的一项重要内容，在信息系统攻击预警以及信息系统攻击响应方面均有重要应用。在攻击预警方面，根据分析得到的恶意代码威胁性，有助于及时对威胁性较高的恶意代码发布预警信息，指导用户开展针对性防护；在攻击响应方面，依据恶意代码威胁性评估结果，有助于选取针对性措施作出响应，通过应急响应人员部署、安全设备升级加以应对。

借助于恶意代码威胁性评估，可以实现有限的人力、财力、物力的精准投放，提高网络空间安全保障的效率与质量。美国国防部计算机应急响应小组（DoD-CERT）、联邦计算机安全事件响应中心（FedCIRC）、美国国家基础设施保护中心（NIPC）等单位的任务之一就是对新兴恶意代码的威胁提供预警，而其基础正是来自于对恶意代码威胁性准确并且科学的评估。

## 14.2 反编译在恶意代码分析中的应用

### 14.2.1 基于文件结构的恶意代码分析

文件结构主要指感染或者捆绑恶意代码的正常文件的代码结构信息。当处于非执行状态时，恶意代码依附于正常文件，当处于执行状态时，控制流要流经恶意代码。感染正常文件或者捆绑正常文件的目的都在于执行恶意代码。

Peter Szor 在对病毒的分析过程中认为代码从文件末段开始执行、段首部存在可疑属性、PE 可选首部大小不正确等文件结构信息可以用来判断该文件存在异常。

Ashish Saini 等人将可疑段数作为恶意代码检测的一项重要指标。作者分析认为每个段都有两个重要参数，即 Virtual Size 与 Raw Data Size。其中 Virtual Size 是段装载进内存的空间尺寸，而 Raw Data Size 是段在磁盘中所占空间尺寸。如果上述两个参数的差距过大，那么认为该段可疑。



以文件结构的异常作为恶意代码检测的依据不失为一种有效方法,但是漏报率与误报率较高,且无法揭示恶意代码的具体行为。所以,在恶意代码分析过程中,文件结构多与其他特征相结合进行恶意代码检测。

### 14.2.2 基于汇编指令的恶意代码分析

汇编指令是二进制代码最底层的表示形式。每条汇编指令由操作码和操作数组成,操作码指明了指令所要进行操作的类型,揭示了汇编指令的行为,因此,当前的汇编级恶意代码分析主要围绕汇编指令的操作码进行。根据对操作码分析的组织模式,主要分为以下三种:

#### 1. 操作码频率

基于操作码频率的分析并未涉及操作码序列,其思路是统计单条操作码的频率,然后借助统计学方法进行恶意代码检测,研究重点放在了采用何种统计模型进行异常检测。

Daniel Bilar 等人使用操作码的频率分布进行恶意代码检测。其关注的操作码包括 398 条 IA32 指令操作码,作者统计了正常代码以及 7 类恶意代码中指令的频率(包括内核模式 rootkit、用户模式 rootkit、黑客工具、僵尸代码、木马、病毒、蠕虫),通过卡方分布验证了操作码的频率分布能够区分正常代码与恶意代码。

Mamoun Alazab 等人提出了一种将在惩罚样条中使用 HS 核的多元逻辑回归模型同操作码频率相结合的特征选择技术检测混淆恶意代码。作者首先对二进制代码进行反汇编并对操作码频率进行统计,然后使用基于最大似然函数的赤池信息量准则进行特征选择,最后采用在惩罚样条中使用 HS 核的多元逻辑回归模型进行恶意代码判定。

虽然操作码频率可以用于恶意代码检测,但是单条操作码在代码中出现的频率非常高,所以误报率较高,同时由于单条操作码表达能力有限,难以细致刻画恶意代码的行为特征。

#### 2. 操作码序列

操作码序列以若干条操作码为特征,基于操作码序列的分析主要关注于操作码的  $n$ -gram 特征,其本质是利用操作码序列的频率特征进行恶意代码检测。由于操作码序列对代码行为的刻画能力比单条操作码高,所以基于操作码序列的恶意代码检测准确率较单条操作码有所提高。

根据操作码序列的长度,可以进一步分为定长操作码序列与变长操作码序列。

Muazzam Siddiqui 等人提出了使用数据挖掘技术通过识别关键指令操作码序列进行恶意代码检测的方法。作者借助 IDA 获取反汇编代码,以操作码序列在样本集合(包括正常代码和恶意代码)中出现的频率作为主要的特征选择标准,进而从中筛选出 1134 条序列,并对上述序列在样本文件中出现的频率进行了统计。作者将操作码序列定义为条件跳转、无条件跳转等分支指令以及函数边界之前的连续指令,因此,该文中操作码序列是可变长的。最后,作者使用逻辑回归、神经网络、决策树作为分类器进行恶意代码判定。



Robert Moskovitch 等人同样借助 IDA 获取反汇编代码, 然后提取  $n$ -gram 操作码序列, 即  $n$  条连续的操作码, 文中分别提取了 1-gram、2-gram、3-gram、4-gram、5-gram、6-gram, 然后对每种  $n$ -gram 分别计算 TF、TFIDF, 之后分别采用 DF(Document Frequency)、GR(Gain Ratio)、FS(Fisher Score) 等方法进行特征选择, 最后采用神经网络、决策树、原始贝叶斯、boosted 决策树、boosted 原始贝叶斯进行分类。在实验结果中, 2-gram 操作码序列取得了较好的效果, 同时  $n$  为 1 时操作码序列不具有代表性,  $n$  较大时又降低了准确率。

操作码序列在恶意代码检测中的作用也是有所区别的, 可以通过加权体现不同操作码序列的作用。Igor Santos 等人提出了一种基于操作码序列出现频率检测恶意代码变体的方法, 并且对操作码之间的相关性进行了挖掘, 得到每条操作码序列的权重。作者首先选择操作码序列的长度, 之后对每条长度为  $n$  的操作码序列计算 TF, 得到由操作码序列频率组成的向量  $S = (o_1, o_2, o_3, \dots, o_{n-1}, o_n)$ , 再对操作码序列出现的频率赋以权值, 得到由加权操作码序列频率组成的向量  $\vec{v} = (wtf_1, wtf_2, wtf_3, \dots, wtf_{n-1}, wtf_n)$ 。对于将被分析的两个程序提取操作码序列作为特征建模向量, 通过余弦相似度计算相似度。实验结果显示长度为 2 的操作码在判断恶意代码变体时优于长度为 1 的操作码, 并且长度为 1 和 2 的操作码联合起来能够有效判断恶意代码, 其中选择相似度的阈值是一个非常关键的问题。

但是, Asaf Shabtai 等人对使用固定规模  $n$ -gram 以及可变规模  $n$ -gram 进行恶意代码检测的效果进行了比较, 结果显示使用不同规模的  $n$ -gram 在性能上并没有明显改进。

### 3. 操作码图

虽然汇编指令操作码组织为操作码序列的方式较为直观, 也有学者基于控制流图将操作码组织为操作码图的形式进行检测。

Blake Anderson 等人通过动态方法获取代码指令序列, 将指令序列建模为基于马尔科夫链的加权有向图, 即操作码图, 图中节点代表指令, 边的权重代表马尔科夫链中状态的转移概率。然后采用图核方法比较指令序列图的相似性, 最后将借助支持向量机进行恶意代码的判定。基于 Blake Anderson 等人所提出的操作码图, Neha Runwal 等人提出了一种新型相似性评估算法进行变形恶意代码检测。

当前, 操作码序列是汇编指令级行为分析的主要特征, 其刻画粒度优于操作码频率, 分析效率优于操作码图。但是, 无论是变长操作码序列还是定长操作码序列都针对的是连续指令序列, 如果恶意代码采用插入垃圾字节等混淆技术, 对检测准确率的影响较大。

## 14.2.3 基于流图的恶意代码分析

流图特征主要包括控制流与信息流, 两者都是编译过程中的重要结构。其中控制流对代码的执行逻辑结构进行刻画, 并且可以同汇编指令、系统调用相结合刻画代码行为。信息流对代码中关键信息所涉及的路径进行提取, 对准确定位恶意行为关联代码具有重要作用。



## 1. 控制流

控制流分析通过捕获程序的控制流语义来对恶意代码进行分析与检测。控制流分析主要通过通过对控制流图的分析对恶意代码进行检测。控制流图是有向图，图中节点为基本块，边表示图中基本块的转移方向。基于控制流图的恶意代码检测原理主要基于子图同构的误用检测，即将分析得到的控制流图同恶意代码的特征控制流图进行比较。因此基于控制流的恶意代码检测问题转化为子图同构检测问题。对此，当前解决方法主要包含两类。

一类是采用经典图匹配算法进行子图同构的判定。

Anju S.S 等人将中间语言表示的控制流图进行优化：包括连续流指令块的节点合并，无条件跳转指令同跳转目标基本块的合并，不可达节点的删除。并将恶意代码的检测转化为子图同构的判断。

由于原始过程间控制流图粒度较粗，Danilo Bruschi 等人通过对节点与边进行标记对图进行扩展，节点根据指令的性质进行标记，边通过所连接节点之间的流的关系进行标记。子图同构通过 VF2 算法进行实现。

另一类是将子图同构问题进行简化。由于图的比较难以在多项式时间内完成，所以研究人员采用多种方法将基于子图同构问题的控制流图误用检测进行简化。

针对变形恶意代码，Shahid Alam 等人提出了标注控制流图（Annotated Control Flow Graph, ACFG）和差别滑动窗口与加权控制流（Sliding Window of Difference and Control Flow Weight, SWOD-CFWeight）技术。ACFG 实现了一种在不影响检测准确率情况下的控制流图快速匹配，对于反汇编得到的汇编代码用中间表示语言 MAIL 进行描述，然后根据 MAIL 描述的代码对控制流图进行标注即得到标注控制流图（ACFG），然后采用 ACFG 作为特征通过恶意代码特征库进行比对。ACFG 能够处理具有控制流图规模较小的恶意代码，并且包含更多信息，因此能够比 CFG 提供更多的信息。SWOD-CFWeight 从一定程度上缓解了操作码频率变化对检测的影响，因为不同的编译器、编译器优化、混淆技术的使用都会影响操作码频率。差别滑动窗口（SWOD）的大小可以变化，从而使得反病毒工具可以选择适当的参数值来优化恶意代码检测。加权控制流则能够捕获程序的控制流语义来帮助实时检测变形恶意代码。

Silvio Cesare 等人提出了两种方法来将控制流图转化为特征向量，然后通过特征向量的近似匹配完成恶意代码检测。一种方法是通过  $k$  子图来刻画控制流图， $k$  子图即控制流图中具有  $k$  个节点的子图。将控制流图中每个规模为  $k$  的子图作为可能的代码特征，并将该图的邻接矩阵以字符串的形式进行存储。另一种方法是通过  $q$ -gram 刻画控制流图， $q$ -gram 即字符串中所有长度为  $q$  的序列所构成的滑动窗口。字符串中每个可能的  $q$ -gram 即认为长度为  $q$  的字符串，所构成的滑动窗口都表示一个特征。通过将控制流图转化为特征向量，控制流图的比较即转换为特征向量距离的比较。

Christopher Kruegel 等人基于控制流图采用了流图着色的方法识别蠕虫变体的相似性。其核心思路是通过  $k$  子图识别控制流图中的公共结构。作者采用规范图标号（canonical



graph labeling) 技术加速子图匹配, 将子图临界矩阵的行向量依次连接, 即将子图匹配问题转换为字符串匹配问题。采用图着色技术解决控制流比较中对基本块中指令的忽略, 根据基本块中指令类型对节点着色, 只有当对应节点连通且具有相同颜色的情况下才判断子图匹配。最后通过一组启发式规则完成蠕虫的检测。

## 2. 信息流

无论是人工分析还是自动分析, 代码规模都是检测过程中必须考虑的问题。信息流采用污点追踪技术对敏感数据进行定位, 通过污点传播收集相关操作, 能够有效降低代码分析规模。基于信息流的恶意代码检测主要通过污点跟踪、程序切片加以实现, 最后通过启发式规则完成恶意代码的判定。

Yin Heng 等人提出的 Panorama 系统能够捕获系统级的信息流。由此得到的信息能够帮助分析人员详细地了解被测程序如何同敏感信息进行交互, 从而使分析人员能够获取这个程序的真实意图。Panorama 在对被分析样本进行分析的过程中, 基于键盘输入、网络输入以及硬盘写入等污点源, 应用数据和地址的污点传播, 进而构建污点图刻画代码行为。

Yin Heng 等人提出的检测方法主要针对显式信息流, 李佳静等人针对隐式信息流设计了一套隐式流敏感的类型推导和检查规则, 刻画了条件语句和循环语句对类型环境的影响。之后借助反向切片技术实现了间接跳转语句分析算法, 最后采用基于隐式流敏感的类型推导规则判断信息流的安全性。

### 14.2.4 基于系统调用的恶意代码分析

系统调用是代码向操作系统内核请求服务的方式。恶意代码对内存、进程、线程等内核对象的操作均需要借助系统调用加以实现。因此系统调用是刻画代码行为的一种重要方式, 根据系统调用在恶意代码检测过程中不同的组织方式, 可以分为系统调用序列与系统调用图。

#### 1. 系统调用序列

与基于操作码序列的恶意代码检测类似, 基于系统调用序列的恶意代码检测主要通过将系统调用建模为  $n$ -gram 的形式进行分析, 其实质是基于系统调用序列的异常检测。

Stephanie Forrest 等人将恶意代码检测建模为免疫系统, 作者假设系统调用短序列相对稳定, 能够对“自我”即正常行为进行建模。正常行为被定义为运行进程中长度为 5、6、11 的系统调用, 然后为每个敏感进程创建正常行为数据库。通过对进程的行为进行实时监控, 如果系统调用序列在正常行为数据库中则无需处理, 否则判断为异常调用, 判断进程存在异常。

机器学习算法被广泛应用于基于系统调用序列的恶意代码检测。

Surekha Mariam Varghese 等人将特定进程的系统调用轨迹表示为序列, 每条系统调用轨迹都是一组不同序列的集合, 对每条序列构建系统调用频率表。进程执行轨迹中单条系



统调用的频率用于确定特定进程的异常状态,输入执行轨迹的系统调用频率用于确定并匹配正常行为。贝叶斯模型使用系统调用频率以及先验概率分布来计算异常分值,当异常分值高于某一阈值即判定为异常。

Zhang Boyun 等人提出了一种基于粗糙集的支持向量机方法进行恶意代码检测,通过监控恶意样本的执行,作者跟踪 API 系统调用,采用滑动窗口方法提取系统调用序列。由于提取到的序列规模较大,所以作者采用粗糙集理论消除冗余特征,最后使用支持向量机对恶意代码进行分类。

Yin Chuanhuan 等人通过监控进程获得系统调用轨迹,正常轨迹用于构建定长模式来表示正常进行,任何偏离正常模式的轨迹都被看作异常。采用滑动窗口从系统调用轨迹中提取出长度为  $k$  的唯一序列,在文中作者选择  $k=10$ 。最后使用基于高阶马尔科夫核的支持向量机进行恶意行为检测。

ByungRae Cha 等人将会话作为行为单元,一种会话即一种行为模式,由于变长模式难以处理,ByungRae Cha 等人试图由变长系统调用序列构建定长行为模式。首先作者以系统调用的数量、系统调用的类型以及系统调用序列为属性建模模式向量,其中调用序列的最大规模为 40,使用 Soundex 算法将变长系统调用序列转换为定长系统调用序列。最后使用反向传播神经网络进行异常行为的判断。

## 2. 系统调用图

图可以作为刻画代码行为的关键数据结构,但是子图同构问题的时间复杂性较高,在基于控制流图的恶意代码检测中已经提到,研究人员主要采用两种思路解决子图同构问题,与之类似,针对系统调用图的恶意代码检测也遵循类似思路。

一类是根据代码分析特点,对经典图匹配算法进行优化并进行系统调用图的匹配。

Youngee Park 等人通过监控系统调用以及系统调用参数,对同族恶意代码的所有实例构建内核对象行为图(KOBG),然后通过聚类得到加权公共行为图,在公共行为图中指明内核对象的作用、属性,以及内核对象之间的依赖关系,并进一步从公共行为图中提取唯一子图,称作热路径(Hot Path)。进行恶意代码检测时,针对被测代码构造的内核对象行为图,如果包含某类恶意代码公共行为图中的热路径,并且同公共行为图的相似性超过某一阈值,即判定为恶意代码。

Ammar Ahmed E. Elhadi 等人通过将 API 调用同操作系统资源相结合的方式来表示图中节点的方式来构建 API 调用图,不同于传统系统调用中节点均为系统调用,操作系统资源亦可作为节点。图中边表示不同节点之间的依赖关系,包括顺序、数据、声明以及调用四种类型。图匹配算法基于加强的图编辑距离算法,该算法通过贪心算法选择最优公共子图的方式降低复杂度。

刘星等人提出了一种基于函数调用图的恶意代码相似性分析方法,作者将两个恶意代码函数调用图之间所有匹配路径中匹配成本最小的匹配路径定义为最优匹配路径,最优匹配路径的匹配成本即两个恶意代码函数调用图的相似性距离 SDMFG。进而采用 Kuhn-



Munkres 算法通过求解最大权匹配实现 SDMFG 的求解。

另一类是将图匹配问题等价转换为其他形式的数据结构进行比较。

Jusuk Lee 等人将系统调用图简化为代码图作为检测的语义特征。其所提架构主要包括代码分析器、代码图生成器以及图分析器。代码分析器将二进制代码转换为有向的系统调用图。代码图生成器将系统调用图简化为代码图。图分析器则用来评估代码图的相似性。在系统调用图中，API 调用表示节点，有向边表示程序调用关系，通过将 API 调用分为 128 组（32 组对象 × 4 种行为）来简化系统调用图，使用邻接矩阵来存储代码图，代码图表示为  $128 \times 128$  的邻接矩阵，每个代码图占用 16KB。最后采用图的交与并进行图的比较和评估图的相似性以完成恶意代码检测。

Parvez Faruki 等人通过将 API 调用图建模为 API Call-gram 进行未知恶意代码的检测。作者首先采用 IDA 进行反汇编，从反汇编文件生成控制流图，再从控制流图得到 API 调用图，API 调用转换为特征向量 API Call-gram，即转化为 API 调用序列模式。最后采用机器学习算法进行恶意代码的检测。

Mojtaba Eskandari 等人使用 CFG 表示程序的控制结构，通过提取的 API 调用对 CFG 进行扩展并刻画其语义行为，扩展得到的 CFG 称为 API-CFG。作者将所有调用的 API 均存储在 API 库中，每个 API 映射到一个全局唯一的数字 API-Id，反汇编构造完成 CFG 后对 CFG 进行遍历，对相关的边用 API-Id 进行标注，即获得 API-CFG。为了降低复杂度，将 API-CFG 转换为二元组“（位置编号，调用名称）”表示的特征向量。最后通过机器学习方法完成恶意代码检测。

白莉莉等人基于代码控制流图提取 API 图，再进一步提取出安全相关的关键系统调用图（Critical API Graph, CAG）。最后将 CAG 同恶意行为库中恶意代码的关键系统调用图（CAG）进行子图匹配，实现恶意代码的检测。为了简化子图匹配的复杂度，作者将子图匹配问题简化为在 CAG 中查找对应节点是否存在连通关系的问题。

## 14.3 恶意代码与反编译技术的对抗

### 14.3.1 混淆

代码混淆（Code Obfuscation）技术，也称为代码迷惑技术，最初是一种防止拥有版权的软件发布被恶意逆向分析的有效手段。程序员采用混淆技术可以将源代码转换为与之功能上等价，但是逆向分析难度大幅提高的目标代码，这样逆向分析人员便难以对程序进行分析，也就难以得到源代码所采用的算法、数据结构等关键信息。但是目前越来越多的恶意代码编写者通过这种技术逃避采用特征码

检测技术的恶意代码分析工具的检测，给恶意代码的检测带来了巨大的挑战。图 14-2 描述了代码混淆的基本概念。

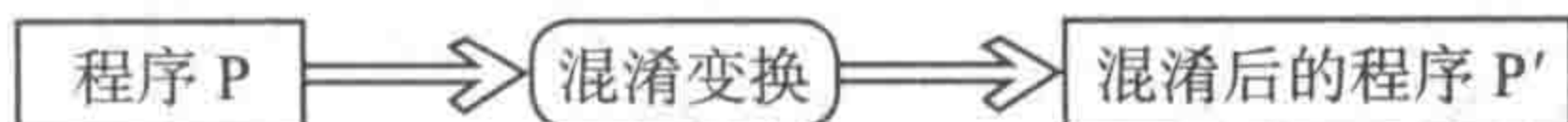


图 14-2 代码混淆示意图



根据混淆对象可以将混淆技术分为以下四类：

### 1. 外形混淆

外形混淆主要包括删除注释、删除源代码的结构信息以及改名。代码注释中往往包含关于程序的功能、算法、输入输出等多种信息，源代码的结构信息则包含方法和类等信息结构。删除注释和源代码的结构信息之后不但使攻击者难以阅读和理解，还可以减小程序的规模，提高程序装载和执行的效率。改名包括对程序中的变量名、常量名、类名、方法名等标识符作词法上的变换。编程人员在编写代码时精心选择这些名称以对变量、方法等的功能位置加以说明，所以改变这些名称能够阻止攻击者对程序的理解。

改名的方法有很多种，如 Hashing 改名、名字交换、重载归纳等。Hashing 改名就是简单地将原来的名字替换为一个不相关的名字。名字交换是将原来程序中所有的名字集中，再随机地分发给变量、常量、类、方法等。这种方法比较隐蔽，攻击者往往不易察觉。重载归纳（Overload Induction）是 PreEmptive 公司的专利技术。它利用了程序不只是依靠名字还可利用方法的参数和返回类型来区分，以及不同命名空间中相同的名字不会引起冲突的特性，将程序中的名字尽可能用相同的名称来代替。

外形混淆是保护软件的第 1 道防线，也是当前最为成熟的混淆技术。几乎每种混淆工具都不同程度地采用了相同或类似的外形混淆方法。

### 2. 控制混淆

根据对控制流的计算、聚合、顺序三个不同方面的影响，控制混淆被分为三类。控制计算混淆通过插入新的代码（冗余或者无用代码），改变源程序的算法。控制聚合混淆将原本逻辑上一体的计算分解，或者将原来逻辑不同的计算加以合并。控制顺序混淆则随机化计算执行的顺序。

对于控制混淆而言，一定量的计算开销是不可避免的。

#### （1）计算混淆

计算混淆可以进一步分为三类：① 在不影响实际运算的情况下将实际控制流隐藏于无关语句之后；② 在不存在与之对应的高级语言的对象代码级别上引入代码序列；③ 删除实际的控制流抽象或者引入可疑控制流抽象。具体来说包括插入无用代码和扩展循环条件、将可归约流图转变为不可归约流图、添加冗余操作数等具体方法，下面以代码并行化为例进行说明。

代码的自动并行化是用于提高多处理器运行程序性能的一种重要的编译优化。但是在混淆中应用并行化的目的不是提高性能，而是混淆控制流。为此，通常采取下面两种操作来达到此目的：① 创建隐蔽进程，运行无用任务；② 将连续的程序代码段分割为多个并行执行的代码段。

如果应用程序运行在单处理器上，代码并行化混淆会有一个较大的执行时间开销。在很多情况下这是可以接受的，而且其抗攻击能力很强：对并行程序进行静态分析非常困难，



因为可能的执行路径的数量随着执行进程数的增长呈几何级数提高。逆向分析人员发现较之顺序程序而言，一个并行程序的分析难度大幅增长。

一个代码段在不包含数据依赖关系的情况下非常容易进行并行化，如图 14-3 所示。

而对于包含数据依赖关系的代码段，则可通过插入适当的同步原语，如 `wait` 和 `start` 以达到分割源代码段、实现并行化的目的。这样，程序仍将会按顺序执行，但是控制流会从一条分支转移到另一条分支，如图 14-4 所示。这样做的目的是提高分析人员对程序控制流的分析难度。

### (2) 聚合混淆

编程人员通过抽象来克服内在的编程复杂性。在程序的多个层次上都可以进行抽象，过程抽象是其中最重要的一种。所以，混淆过程和方法调用对于混淆者来说是很重要的。

在实际应用过程中可以通过多种方法来达到此目的，如函数内嵌和外提、函数交叉、函数克隆以及循环变换等，也可以将这些方法聚合起来使用，实现聚合混淆。事实上，这些方法背后的基本思想都是一样的：① 原本属于一个方法的代码（假设其逻辑上也属于一个整体）应该加以分割并分散到程序的各个部分；② 看起来逻辑上无关的代码应该聚合起来，放到一个方法中。

### (3) 顺序混淆

编程人员尽可能地以局部化原则来组织其代码。其基本思想在于如果程序中的两条语句（或者函数等）逻辑相关，并且其在源代码中的物理位置也相邻，那么程序易于阅读和理解。这种局部性体现在源代码的各个层次：表达式、语句、基本块，还有方法中的基本块、类中的方法、文件中的类等，所有的空间局部性都能为逆向分析人员提供逆向分析的线索。因此，如果可能的话，应该将前面提到的源代码中的各语法单位位置随机化放置。对于某些类型的项（如类中的方法），由于其数据结构的特点，可以很容易地改变其位置。而对于其他情况（如基本块中的语句），则需要进行依赖关系分析以确定能够对哪些语句进行乱序放置。

顺序混淆的代价不高，但是其抗分析能力很强。例如，当将一个基本块中语句的放置随机化后，在最终的代码中找不到原来顺序的任何痕迹。

最后要特别注意，在控制混淆中“不透明谓词”的应用十分常见，关于不透明谓词将在 14.4.4 节详细说明，此处不再赘述。

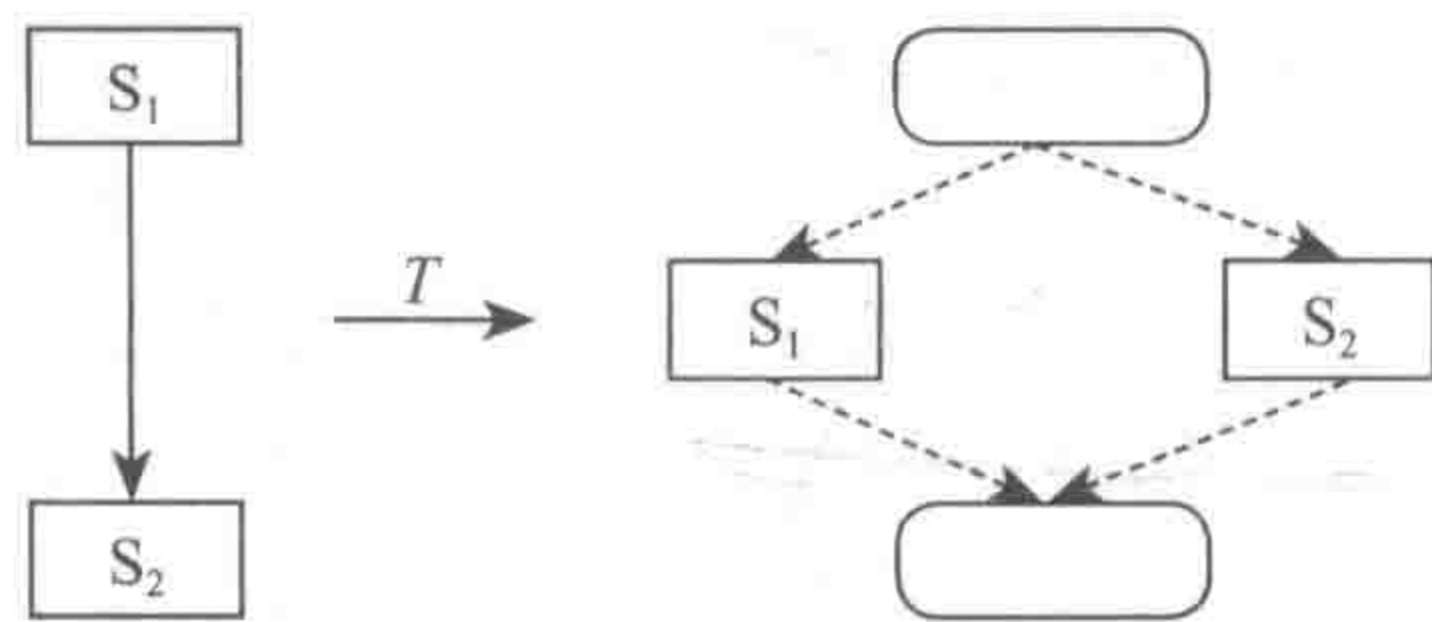


图 14-3 代码并行化示意图（无数据依赖关系）

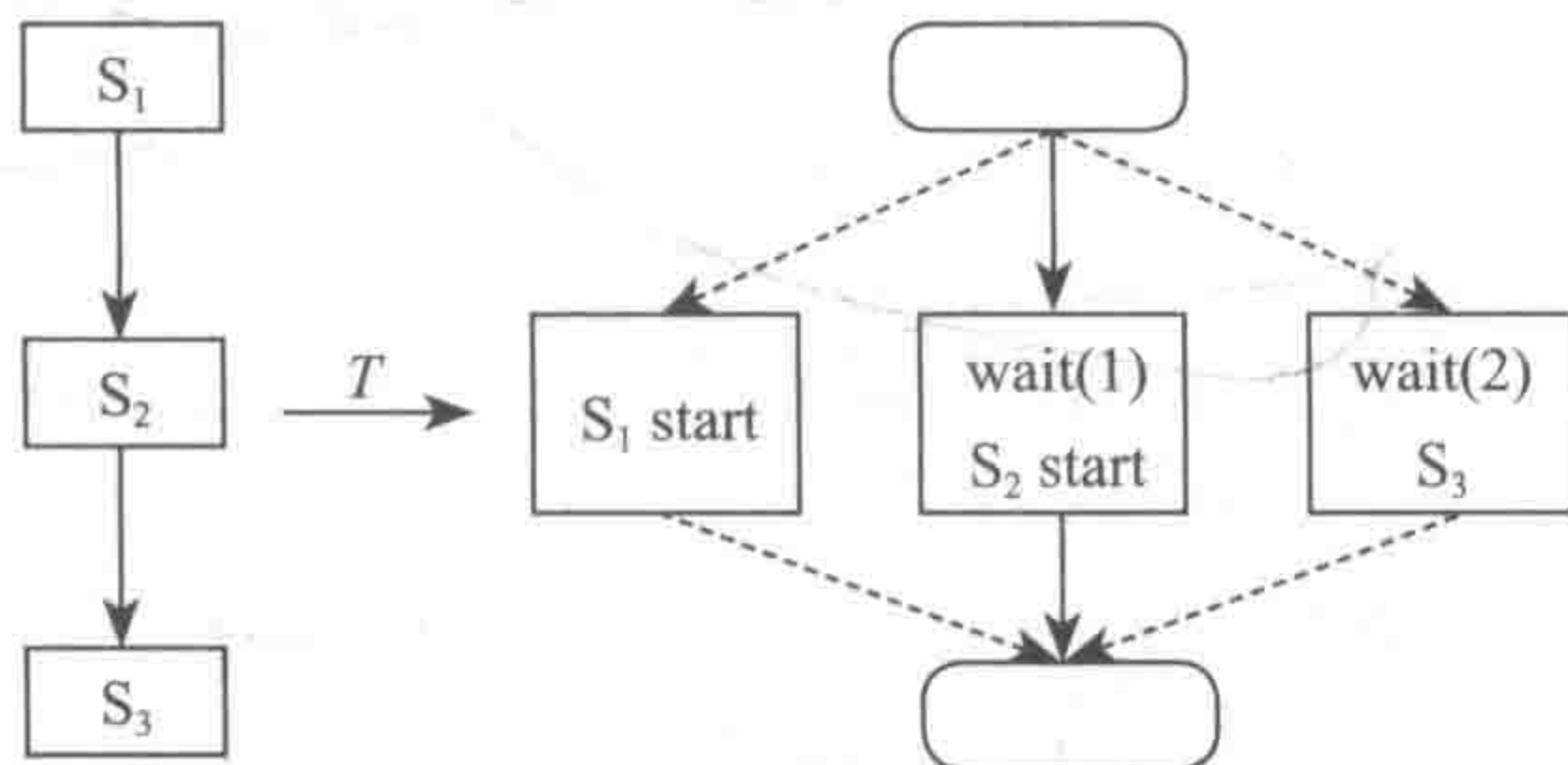


图 14-4 代码并行化示意图（存在数据依赖关系）



### 3. 数据混淆

数据混淆的对象是程序中的数据结构。数据混淆中也可以根据对数据存储和编码、聚合或者顺序进行混淆进一步分为三类。

#### (1) 存储、编码混淆

在很多情况下都采用一种“自然”方式来存储程序中特殊的数据项。例如，为了遍历数组中的元素，一般都会分配一个大小适当的局部整型变量作为遍历变量。分配其他类型变量当然也是可以的，但是实际应用中并不常见，而且可能会导致效率的降低。

此外，对位数据通常也存在“自然”解释，即一个变量往往保存基于该变量类型的内容。例如，一个16位的整型变量“0000000000001100”表示整数值12。但是，这仅是在通常情况下对该数据作出的解释，其他解释方式也是可以的。

存储混淆变换就是选择一种不常见的存储方式来存储动态数据或者静态数据。类似地，编码混淆则试图选择对于常见数据类型而言不常见的编码方式。存储混淆和编码混淆通常一起使用，当然也可以分开使用。

一些简单的存储混淆是将某一个变量从一种专用的存储类型变换为一种更加通用的类型。这种混淆变换结合其他混淆变换会更加有效。

例如在Java语言中，一个整型变量可以提升为一个整型对象。类似的操作同样可以应用于其他具有对应的“封装”类的标量类型。图14-5就是一个例子。

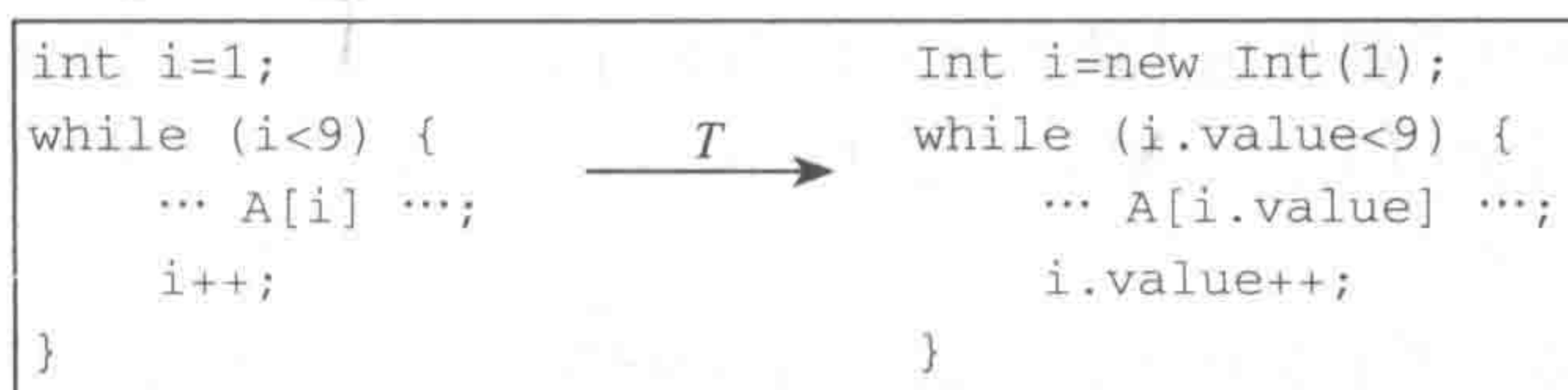


图 14-5 存储类型变换

还可以改变一个变量的生存期。最简单的变换是将一个局部变量转换为一个全局变量，该全局变量在相互独立的过程调用之间共享。例如，如果一个过程P和Q都引用了一个局部整型变量，而且P和Q不能够同时运行，那么该变量就可以成为这两个过程之间的全局共享变量，如图14-6所示。

#### (2) 聚合混淆

不同于命令式语言，面向对象语言更加面向数据而不是面向控制。换句话说，在一个面向对象的程序中，控制是围绕数据结构组织的。这意味着对面向对象的应用程序进行逆向分析的一个重要内容就是存储程序中所用到的数据结构。所以对于混淆而言则需要尽可能地隐藏这些数据结构。

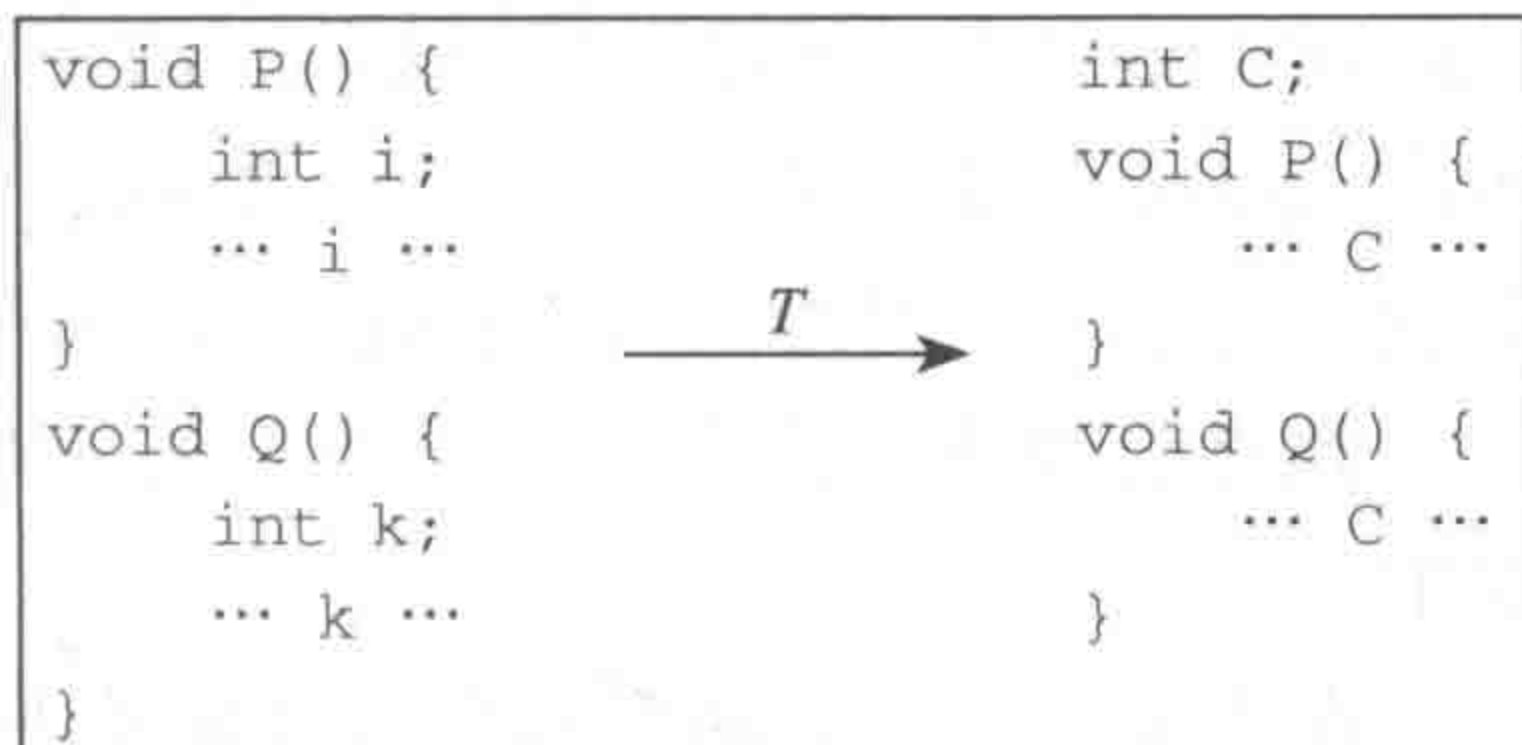


图 14-6 变量生存期变换



在大多数面向对象编程语言中，只有两种方式来聚合数据：以数组为单位聚合数据或者以对象为单位聚合数据。

### (3) 顺序混淆

在控制混淆中混淆计算所执行的顺序是一种常见的混淆形式。类似地，对源代码中的声明进行随机化也是一种常见的混淆形式。与之前不同的是，此处是对方法以及类中变量和方法中形式参数的顺序进行随机化。在对方法中形式参数的顺序进行随机化的过程中，实参顺序也要进行相应的重新排序。这种混淆的强度虽然较低，但是抗分析性比较好。

在很多情况下，也可以对一个数组中的元素进行重排序。例如通过不透明函数  $f(i)$  可以将原数组中的第  $i$  个元素映射到重新排序的数组中的位置，如图 14-7 所示。

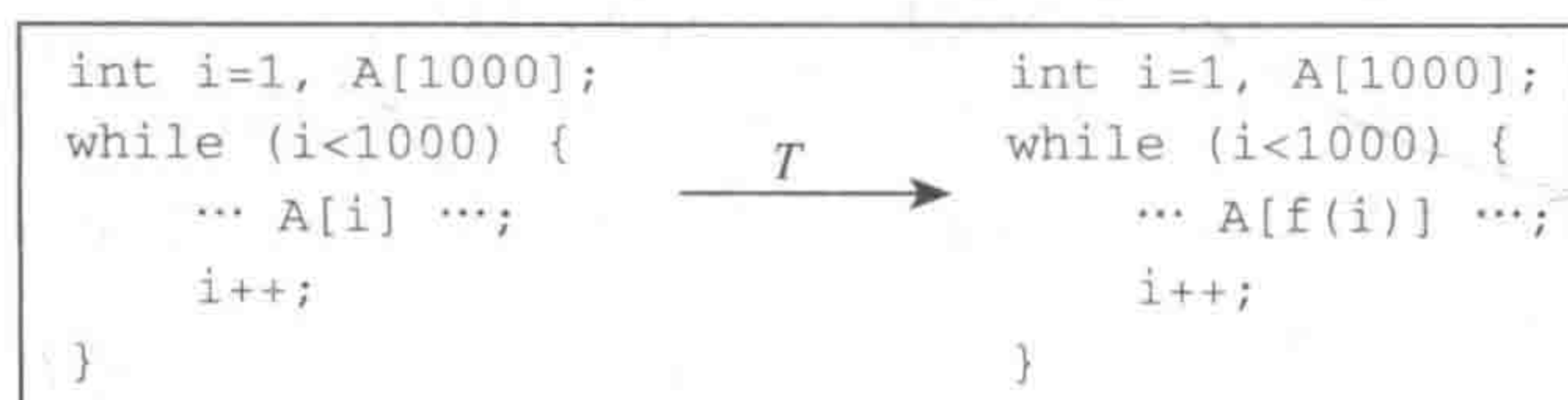


图 14-7 顺序混淆示例

## 4. 预防混淆

预防混淆不同于控制混淆和数据混淆。控制混淆和数据混淆的目的在于迷惑程序或者分析人员。但是预防混淆则是用来降低各种已知的自动反混淆技术的分析能力（内在预防混淆），或者利用当前各种反混淆器和反编译器中的弱点（目标预防混淆）来实现混淆的目的。

### (1) 内在预防混淆

通常内在预防混淆具有较低的混淆强度以及较高的抗分析能力。所以在实际应用过程中，内在预防混淆多与其他混淆变换相结合，以提高其他混淆变换的抗分析能力。

### (2) 目标预防混淆

例如，反编译器 mocha 对于 return 后面的指令不进行反编译，而反逆向分析工具 Hosemocha 就是专门针对此缺陷，故意将代码放在 return 语句后面，从而使反编译失效；PreEmptive 公司开发的针对 .NET 的混淆器 Dotfuscator 也有针对反编译器 ILDASM 的预防混淆功能。该类混淆对于特定的反编译工具是非常有效的。

到目前为止还没有一个全能型的、对每一种反编译工具皆有效的工具，其局限性是明显的。

在实际应用中，通常都会综合使用上面这些混淆技术以达到最优的混淆效果。

## 14.3.2 多态

虽然代码加密能够避免反汇编技术对恶意代码主体的分析，但是也存在问题。恶意代码为了执行，必然需要对代码主体进行解码，因此只要发现代码中存在解码代码，就可认



为该段代码是可疑的。为了克服代码加密带来的问题,基于解密代码变异的多态技术也得到了广泛的应用。多态恶意代码能够对它们的每一个变种的解密代码进行变换,产生大量的不同形式的解密代码,甚至采用不同的加密方法对恶意代码主体代码进行加密,即完成同一功能但采用了不同的方法。多态的特点是随机性和变化性。多态变换的基本模型如图 14-8 所示。

其中:

1) 指令位置变换是指部分指令的相对顺序对于程序的执行效果不产生影响,通过改变它们的执行顺序可以得到不同的代码。见图 14-9,该指令的位置就是任意的,可以有 6 种变化。

2) 寄存器变换指在不影响程序执行效果的基础上,可以在代码中随机地选择能够使用的寄存器,形如:

```
mov reg, 1234
mov[0x5678], reg
```

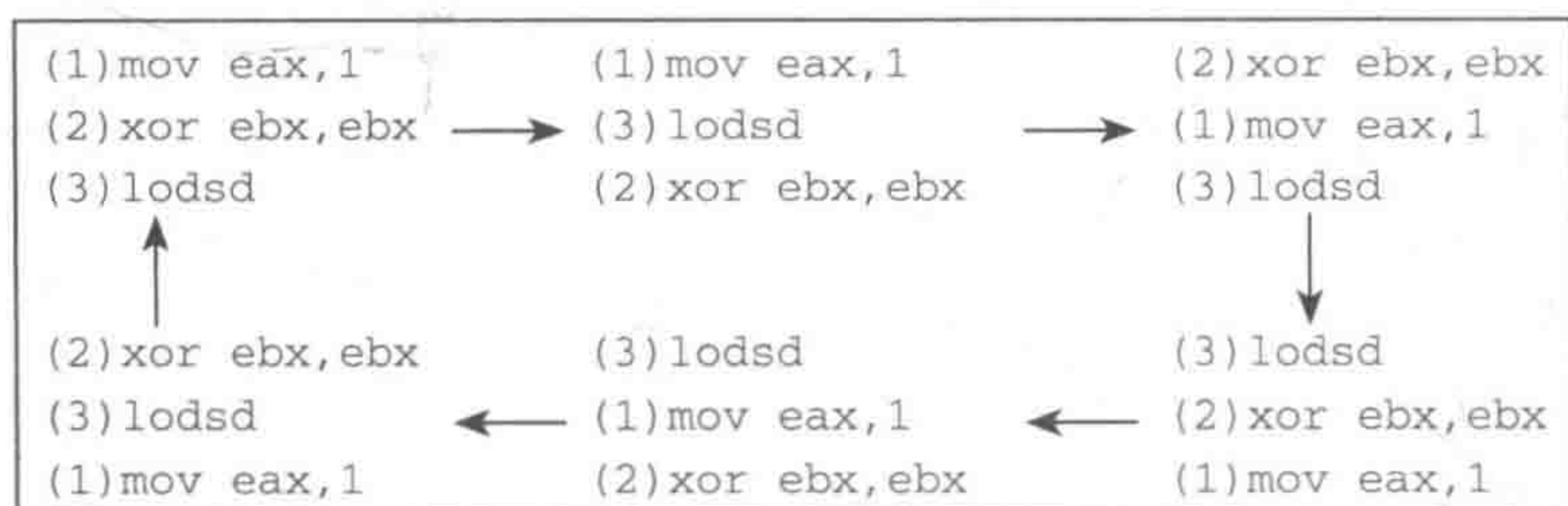


图 14-9 简单的指令位置变换例子

其中 reg 可以在 eax、ebx、ecx、edx 等通用寄存器之间进行随机选择。

3) 指令扩展变换是指可以将一条指令替换为能够完成同样功能的多条等价指令。如图 14-10 所示。

4) 指令收缩变换可以将多条指令替换为一条等价指令,如图 14-11 所示。

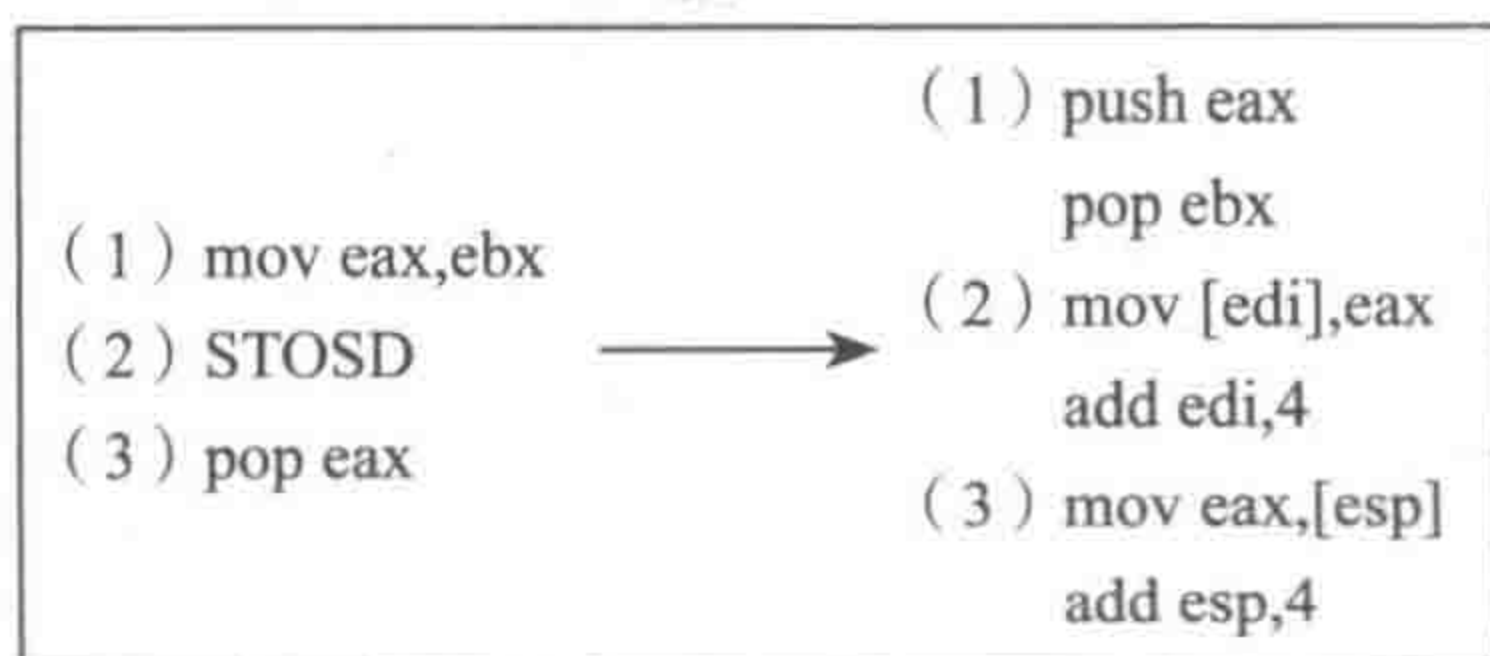


图 14-10 简单的指令扩展变换例子

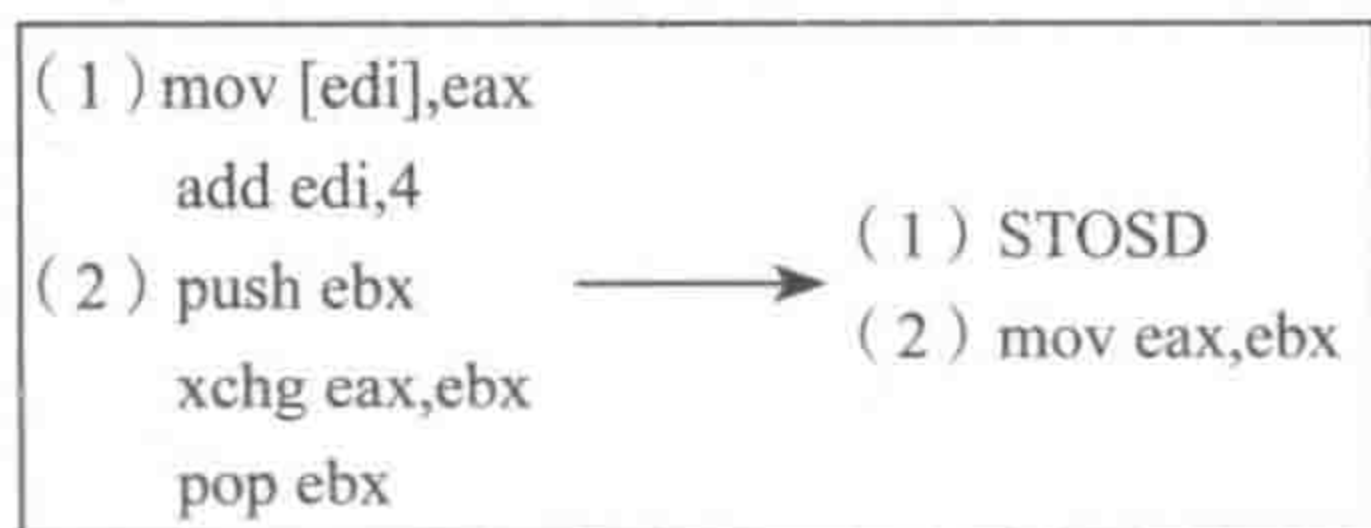


图 14-11 简单的指令收缩变换例子

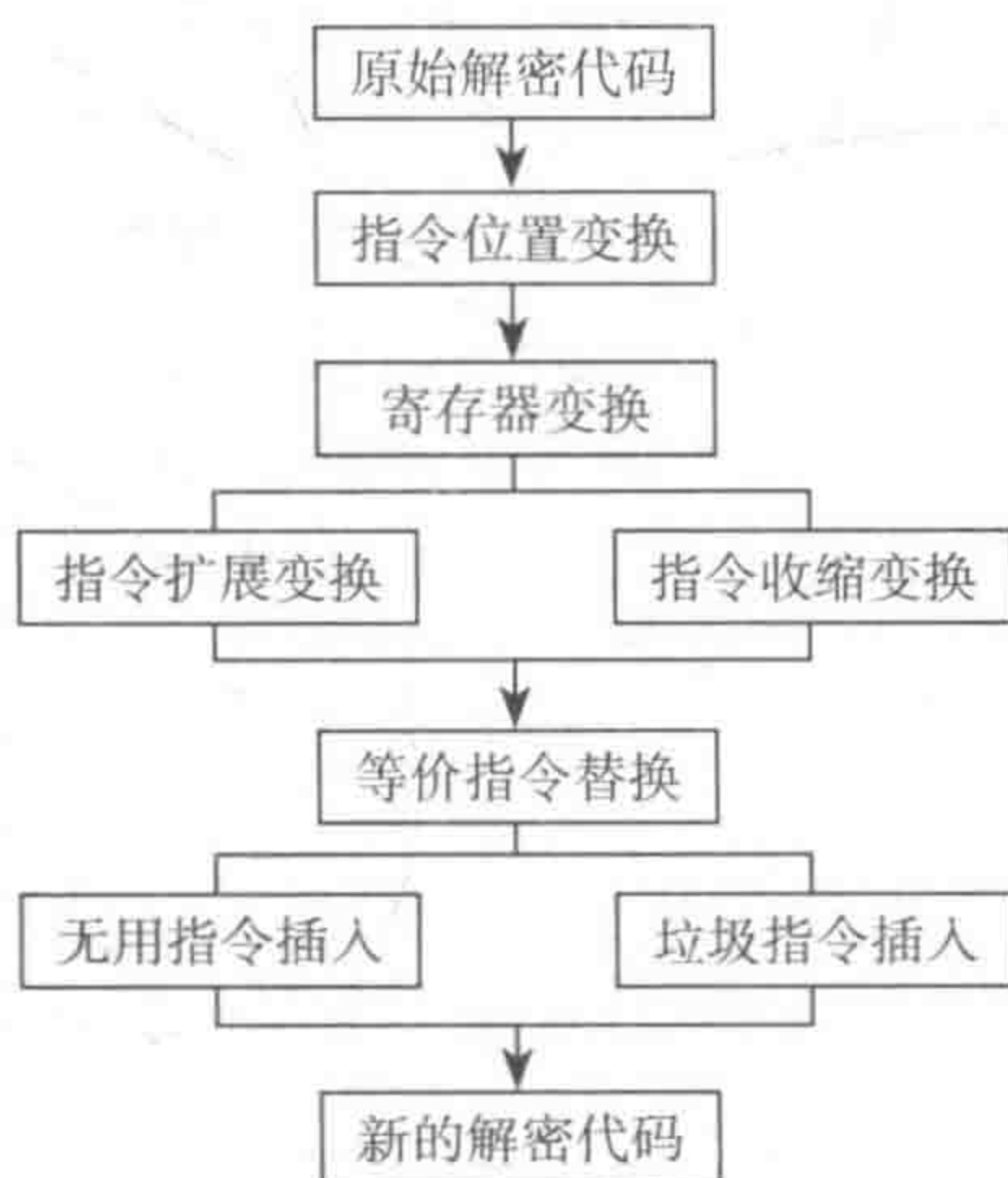


图 14-8 多态变换基本模型



- 5) 等价指令替换指将一条指令替换为一条等价指令，如图 14-12 所示。
- 6) 无用指令插入指在程序代码中插入不影响解密代码的指令。常见的如 NOP 指令插入，由于 NOP 指令没有任何功效，所以计算机程序的功能没有任何改变。但是 NOP 指令不能无止尽地任意添加，如果 NOP 指令出现的频率超过了正常编译器编译得到程序的限度，就容易被杀毒软件检测出来。
- 7) 垃圾指令的执行并不影响程序的执行效果，可以有单字节、双字节的垃圾指令。如图 14-13 所示。

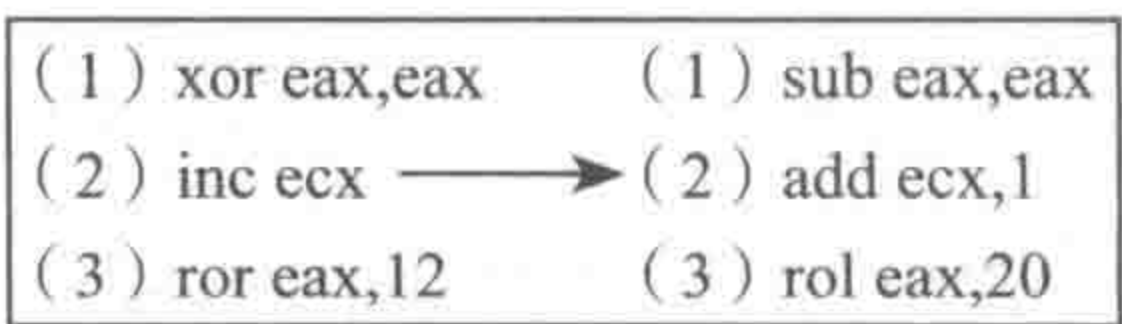


图 14-12 等价指令替换的例子

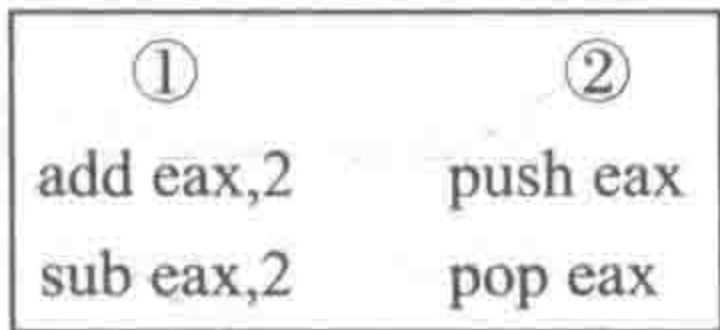


图 14-13 垃圾指令示例

以上是多态变换模型的各个功能模块的主要功能，多态变换的目的是改变程序解密模块的特征码，干扰杀毒软件的扫描。为了检测多态的病毒，反病毒软件将相应的病毒主体解密模块集成到自己的产品中，用以动态地检测加密的病毒主体。由于每一个多态病毒的所有实例都带有相同的加密病毒主体，因此，基于解密得到病毒主体内容的检测是可行的。

14.3.3 变形

为了提高恶意代码的抗检测能力和生存能力，变形技术也得到了大量应用，可以将变形技术看作多态技术的发展。变形技术在多态技术的基础上更进一步，在每次应用过程中不仅对解密代码，并且对恶意代码主体也进行变换，使不同恶意代码实例的代码完全不同。变形后生成的恶意代码变种在表现形式及原代码主体内容上都不一样。多态技术主要对恶意代码中的解密代码进行变换，变形技术还对恶意代码主体进行变换。变形技术可以动态生成更复杂的变化过程，包括对重要代码版段的动态加密、解密，语义等价下的代码和数据替换，还包括代码混淆，这些都使得分析人员受累于代码复杂的变换和还原过程，从而无法找到真正的恶意代码解码代码与代码主体。

换句话说，变形技术使恶意代码的“外形”发生了变化但没有改变其行为。如果说对抗多态技术还可以通过虚拟机等待恶意代码被还原之后检测特征值，那么变形技术则使得该种检测技术完全失效。变形的基本模型如图 14-14 所示。

在变形过程中首先对二进制恶意代码进行反汇编，

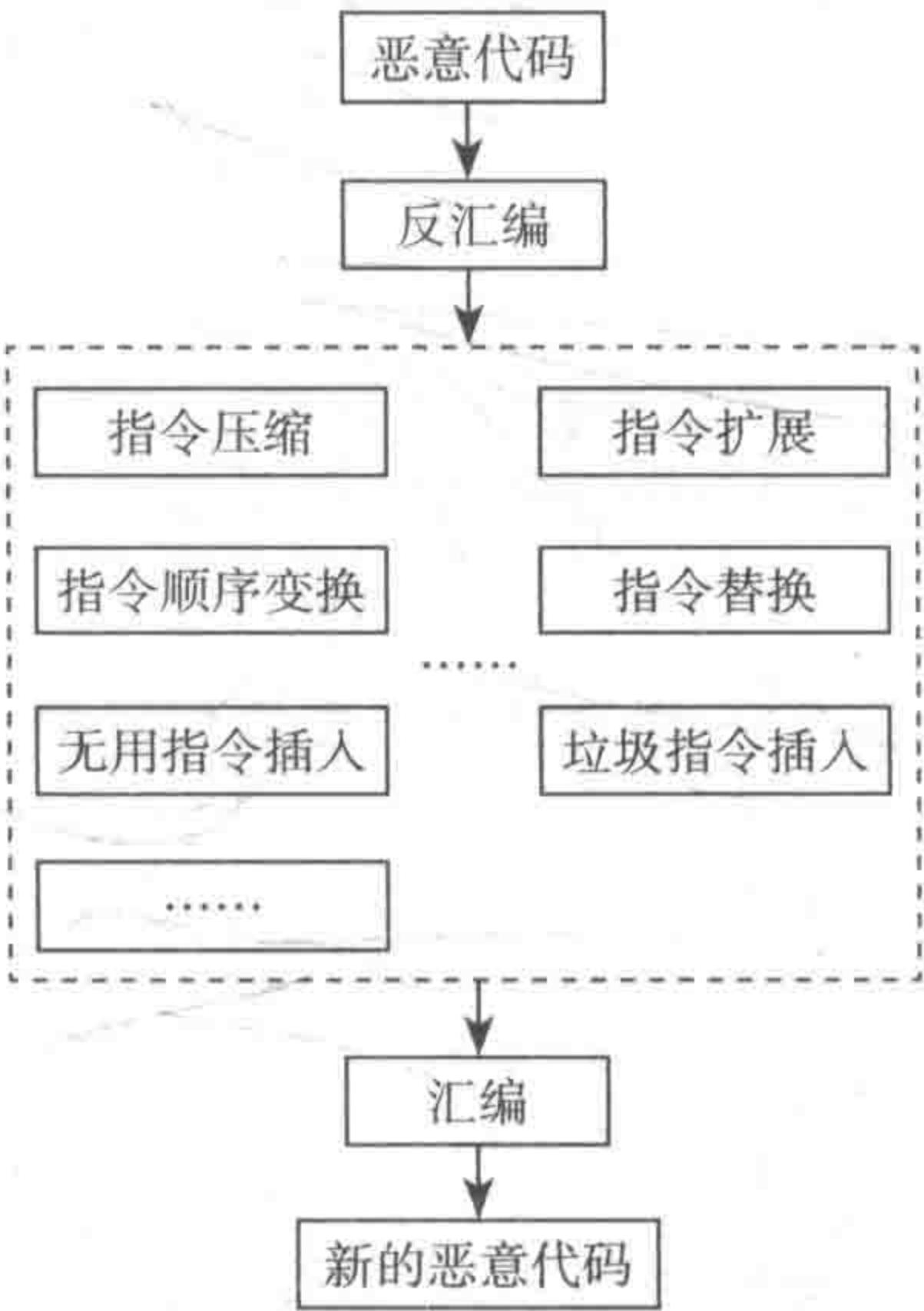


图 14-14 变形变换基本模型



得到恶意代码的反汇编指令或者中间表示,然后在反汇编指令或者中间表示的基础上进行指令压缩、指令扩展、指令顺序变换、指令替换、无用指令插入以及垃圾指令插入等变换,最后对变换后的反汇编指令或者中间表示进行汇编,使之生成新的可执行恶意代码。

由于涉及对源代码的反汇编以及对变形完成代码的再次汇编,所以变形的恶意代码编写比较困难,并且一般的变形代码体积比较大,因此综合利用加密多态技术的恶意代码变种较多。多态变形技术虽然改变了程序的外部表现形式,但是对于程序的控制流程没有改变,因此可以对程序运行的结构和行为进行分析,以检测恶意代码的存在。

#### 14.3.4 加壳

加壳是指在原二进制文件(如可执行文件、动态链接库)的基础上附加一定的代码、数据,并对原文件进行加密或压缩,然后修改原文件的运行参数,使其被加载执行时程序流程发生改变,即首先执行附加的代码,再执行原文件的代码。在原二进制文件中附加的代码(称为壳)先于被保护的程序运行,在运行过程中对原有程序的代码及数据进行相应的还原、解密等操作。

加壳修改了原始程序的执行文件的组织结构,从而能够比原始程序的代码提前获得控制权,但不会影响原始程序的正常运行。下面简单说明一般“壳”的装载过程。

1) 获取壳自身所需要使用的 API 函数地址。如果用 PE 编辑工具(如 LordPE)查看加壳后生成的文件(packed binary),会发现未加壳的文件和加壳后的文件的输入表不一样,加壳后的输入表一般所引入的 DLL 和 API 函数很少,甚至只有 Kernel32.dll 以及 GetProcAddress 这个 API 函数。壳实际上还需要其他 API 函数来完成它的工作,为了隐藏这些 API 函数,它一般只在代码中用显式链接方式动态加载它们。

2) 解密原始程序的各个区块的数据。壳出于保护原始程序代码和数据的目的,一般都会加密原始程序文件的各个区块。在程序执行时外壳将会对这些区块数据解密,以让程序能正常运行。壳一般是按区块加密的,那么在解密时也按区块解密,并且把解密的区块数据按照区块的定义放在合适的虚拟内存位置。如果加壳时用到了压缩技术,那么在解密之前需要解压缩。

3) 重定位。文件执行时将被映像到指定内存地址中,这个初始内存地址称为基地址(ImageBase)。当然这只是程序文件中声明的,程序运行时不能保证系统一定满足此要求。对于可执行文件(EXE)的程序文件来说,Windows 操作系统会尽量满足。例如某 EXE 文件的基地址为 0x400000,而运行时 Windows 操作系统提供给程序的基地址也同样是 0x400000,在这种情况下就不需要进行地址“重定位”了。不过对于动态链接库(DLL)来说,Windows 操作系统一般无法保证为每一次 DLL 运行时都提供相同的基地址。这时“重定位”就很重要了,此时壳中也需要提供进行“重定位”的代码,否则原程序中的代码无法正常运行。

4) HOOK API。程序文件中输入表的作用是让 Windows 操作系统在程序运行时提供



API 的实际地址给程序使用。在程序的第一行代码执行之前，Windows 操作系统就完成了这个工作。壳一般都修改了原始程序文件的输入表，然后模仿 Windows 操作系统的工作来填充输入表中相关的数据。在填充过程中，外壳就可填充 HOOK API 的代码的地址，这样就可间接地获得程序的控制权。

5) 跳转到程序原入口点 (OEP)。从这时起壳就把控制权交还给原始程序了。加壳不仅可以改变原文件的执行流程和特征，并且可以有效改变原文件的静态特征。加上外壳后，原文件代码在磁盘文件中一般是以加密或压缩后的形式存在，只在执行时从内存中还原，这样可以有效地防止程序被静态反编译，同时也可以防止破解者对程序文件进行非法修改。

为可执行文件或者动态链接库加壳的目的在于压缩原文件的体积或防止原文件被破解、篡改。常用的加壳软件可分为两类：一类以压缩为目的，主要用于减少 PE 文件的体积，如 ASPack、UPX 等；另一类以加密为目的，主要用于保护 PE 文件所蕴含的信息，如 ASProtect 等。

### 14.3.5 虚拟执行

“虚拟机保护”是一种新兴的软件保护技术，起源于俄罗斯的著名软件保护软件“VmProtect”，以此软件为开端引发了软件保护壳领域的新革命。各大软件保护壳开发团队都希望将虚拟机保护这一新颖的技术加入和应用到自己的产品中，以增强技术优势和安全性能，虚拟机保护可以算作代码混淆技术的新思路 and 突破性进展，这一技术同样可被恶意代码作者用来对抗反编译技术对恶意代码的分析。

基于虚拟机保护的恶意代码的编写思想一般是通过设计独立的指令系统，构造现实生活中并不存在的计算机及其汇编器和调试器，将恶意代码中的关键代码转换为该虚拟机识别的指令，在执行过程中由虚拟机解释器完成关键代码指令的解释执行。虚拟机保护的总体流程如图 14-15 所示。

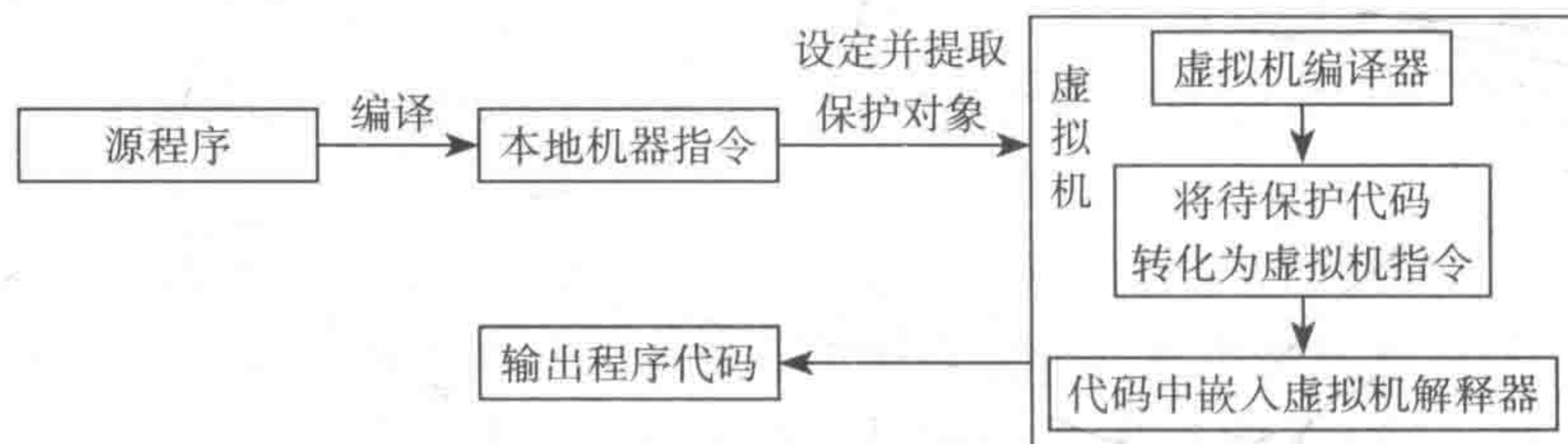


图 14-15 虚拟机保护框架图

在使用虚拟机保护时，首先把恶意代码的源程序转化为本地机器指令，根据设定的保护对象将需要保护的恶意代码中关键代码送入虚拟机中，虚拟机编译器将其转化为虚拟机指令，然后将转化后的代码与虚拟机解释器一起与原恶意代码相结合得到基于虚拟执行技术的恶意代码。



## 14.4 反编译框架针对恶意行为的改进

### 14.4.1 条件跳转混淆

条件跳转混淆是一种广泛采用的混淆技术。其主要特点如下：

- 实现灵活方便，能够添加到汇编代码的任意位置。
- 混淆效果明显，当前各主要反汇编引擎对采用条件跳转混淆技术混淆后的程序进行反汇编均无法得到正确结果。
- 实现代价相对较低，相对于其他混淆技术而言，条件跳转混淆技术的时间、空间开销都很低。

由于条件跳转混淆使逆向分析工具的反汇编引擎无法得到正确的反汇编结果，从而直接导致控制流、数据流等分析过程无法正常展开，后续的恶意代码判断更无从谈起。所以需要在反汇编阶段破解条件跳转混淆以保证后续分析的顺利进行。

#### 1. 条件跳转混淆的主要形式

根据条件跳转指令放置的位置，条件跳转混淆的实现形式进一步分为四类，即典型形式、多层顺序嵌套形式、多层乱序嵌套形式以及嵌入冗余代码形式，下面将分别进行介绍。

##### (1) 典型形式

首先介绍条件跳转混淆的典型实现形式，如图 14-16 所示，后面三种形式都是基于典型形式作进一步改进得到的。

图 14-16 所示汇编代码段中 (2)、(3) 处的 `jz`、`jnz` 是条件跳转指令，在反汇编过程中，不管反汇编引擎采用线性扫描算法还是行进递归算法，当遇到条件跳转指令时都不会发生控制流的转移。然而由于这两条条件跳转指令的跳转条件形成互补关系，而且其跳转目的地址相同，从而导致 `jz`、`jnz` 这两条条件跳转指令的组合实际上相当于一条件无跳转指令。因此在实际的执行过程中，两条连续的条件跳转指令在跳转条件互补且跳转目的地址相同的情况下一定会导致控制流的转移。这时如果在最后一条跳转指令后面插入垃圾数据就会导致反汇编错误。所插入的垃圾数据通常是一条多字节指令的操作码，如在图 14-16 所示 (4) 处的 `thunkcode` 中放入 `0e8h`，由于 `0e8h` 是 `call` 指令的操作码，这样对 `0e8h` 进行解码时就会将它后面的四字节，也就是 `maliciouscode` 的前四字节看作 `call` 指令的调用目标地址，从而导致反汇编出错，达到隐藏恶意代码的目的。

(1)	<code>normalcode</code>
(2)	<code>jz label</code>
(3)	<code>jnz label</code>
(4)	<code>db thunkcode</code>
(5)	<code>label:</code>
(6)	<code>maliciouscode</code>

图 14-16 条件跳转混淆典型形式

##### (2) 多层顺序嵌套形式

对于 Intel IA32 体系结构下的指令集合，其指令结构决定了反汇编过程中存在一种自修复现象：即使反汇编过程中在某条指令处出现了错误，经过若干条指令后，它将自动恢复到正确的反汇编结果，即重新同步到正确的指令序列上来，这种反汇编错误在有限的几条指令后自动修复的现象称为反汇编过程的自修复现象。所以典型形式的条件跳转混淆可能



仅仅造成几条指令的反汇编错误，错误传播的范围比较小。但是下面提到的多层顺序嵌套的条件跳转混淆则能够使更多条指令的反汇编得到错误的结果，使得错误传播范围扩大。

多层顺序嵌套有两种形式，分别如图 14-17 中 A、B 所示。A 中 (2)、(3) 处的条件跳转指令对和 (4) 处的垃圾数据形成混淆，从而能够隐藏 (6) 处所表示的部分恶意代码。紧接着，(7)、(8) 处的条件跳转指令对和 (9) 处的垃圾数据又形成混淆，从而能够隐藏 (11) 处所表示的部分恶意代码。类似地还可以通过增加条件跳转指令对数来形成更多的混淆，以达到所期望的混淆效果。对于 B 中形式，则是在两条条件互补且跳转地址相同的条件跳转指令 (2)、(7) 之间再嵌套入一组混淆，即 (3)、(4)、(5) 的指令组合，以防止被轻易地识别。

### (3) 多层乱序嵌套形式

在上面的形式中，可以简单地将条件跳转到跳转目的地址之间的所有字节用“090h”填充来破解混淆。例如，如图 14-17 的 A 中，将 (3) 与 (5) 之间的所有字节均用“090h”予以填充，这样解码得到指令为空操作指令 nop，而不会影响反汇编结果。对于 B 而言，则可通过多遍填充来解决。但是对于条件跳转混淆的第三种形式——多层乱序嵌套，上面所述填充字节的方式则无能为力：

多层乱序嵌套也有两种形式，分别如图 14-18 中 A、B 所示。以 A 为例，实际的执行顺序为执行至 (2) 或 (3) 处，控制流会跳转至 (10) 处 label1 表示的代码块；接着执行至 (12) 或 (13) 处，这时控制流会向后跳回至 (5) 处 label2 表示的代码块。如果采用简单的填充

A:		B:	
(1)	normalcode	(1)	normalcode
(2)	jz label1	(2)	jz label1
(3)	jnz label1	(3)	jz label2
(4)	db thunkcode1	(4)	jnz label2
(5)	label1:	(5)	db thunkcode2
(6)	maliciouscode1	(6)	label2:
(7)	jz label2	(7)	jnz label1
(8)	jnz label2	(8)	db thunkcode1
(9)	db thunkcode2	(9)	label1:
(10)	label2:	(10)	maliciouscode1
(11)	maliciouscode2		.....
	.....		

图 14-17 多层顺序嵌套

A:		B:	
(1)	normalcode	(1)	normalcode
(2)	jz label1	(2)	jz label1
(3)	jnz label1	(3)	jnz label1
(4)	db thunkcode1	(4)	db thunkcode1
(5)	label2:	(5)	label2:
(6)	maliciouscode2	(6)	maliciouscode2
(7)	jz label3	(7)	jz label3
(8)	jnz label3	(8)	jz label4
(9)	db thunkcode3	(9)	jnz label4
(10)	label1:	(10)	thunkcode3
(11)	maliciouscode1	(11)	label4:
(12)	jz label2	(12)	jnz label3
(13)	jnz label2	(13)	thunkcode4
(14)	db thunkcode2	(14)	label1:
(15)	label3:	(15)	maliciouscode1
(16)	maliciouscode3	(16)	jz label2
		(17)	jnz label2
		(18)	thunkcode2
		(19)	label3:
		(20)	maliciouscode3

图 14-18 多层乱序嵌套



方法,那么(3)和(10)之间的所有字节都将被填充,其中当然也包括(6)处所表示的恶意代码,这样就无法得到完整的反汇编结果,从而可能导致后续的恶意代码分析与识别不够准确。对B可进行类似分析。

#### (4) 嵌入冗余代码形式

最后,还可以通过嵌入冗余代码的形式来使分析者耗费更多的精力,而且也可以提高条件跳转混淆的识别难度。下面仅对采用对称指令来表示冗余代码的情况进行说明,在实际应用中冗余代码可以有多种实现形式。其混淆形式如图14-19所示。

```
(1)    normalcode1
(2)    jz    label
(3)    对称的代码。例如, push eax, pop eax; dec ebx, inc ebx 等
(4)    jnz   label
(5)    db   thunkcode
(6)label:
(7)    maliciouscode
```

图 14-19 冗余代码形式

对称代码的定义是:执行效果互相抵消的两个指令所组成的指令对。例如, push eax 和 pop eax, inc ebx 和 dec ebx 都是对称指令。

通过测试,发现采用线性扫描算法的反汇编器 objdump 与采用行进递归算法的 IDA 对用上面提到的四种混淆形式进行混淆的代码的反汇编结果正确率都不能令人满意。

## 2. 针对条件跳转混淆的分析

针对前面提到的四种条件跳转混淆形式,进行以下分析:

### (1) 条件跳转混淆形式分析

在条件跳转混淆的典型情况中,可以很轻松地使采用线性扫描算法的反汇编引擎得到错误的反汇编结果,这是因为线性扫描算法在解码第二条条件跳转指令之后会顺序地从下一字节处开始进行下一条指令的解码,而无论该字节是数据还是指令。所以如果放在该字节处的数据恰好是某条多字节指令的操作码,那么就可能会将该字节连同后面几字节的内容作为一条完整的指令进行解码,从而得到错误的反汇编结果。

对于采用行进递归算法的反汇编引擎而言,虽然在处理控制转移指令时会进行控制流的转移,看起来似乎能够轻易化解前面的条件跳转混淆,但是实际上,采用行进递归算法的反汇编器得到的结果依然是错误的。这是因为,在进行静态分析时无法准确确定条件跳转指令的跳转条件是否成立,所以行进递归算法遇到条件跳转指令时不可能像处理无条件跳转指令那样,将控制流转移到两条条件跳转指令共同的目的地址处,然后再继续进行反汇编。实际分析过程是,在处理条件跳转时并不进行控制流的转移,而是与线性扫描算法一样,也是顺序地从条件跳转指令的下一字节处开始继续解码。

所以采用条件跳转混淆技术的恶意代码既能够使采用线性扫描算法的反汇编器一筹莫展,也能够使采用行进递归算法的反汇编器无能为力。



针对上述分析,笔者设计并实现了一种改进算法。该算法基于行进递归反汇编算法,能够有效地识别并破解上面所提到的四种条件跳转混淆形式。在改进算法中,首先创建一个链表保存解码时遇到的条件跳转指令的指令地址、指令长度、条件跳转类型。在反汇编过程中,当第一次遇到某条条件跳转指令时先将其放入链表,然后像修改之前一样继续进行顺序解码;当再次遇到条件跳转指令时也将该指令放入链表中,这时要对链表中的条件跳转指令的指令类型进行逐条判断,对于指令类型互补的条件跳转,再判断这两条条件跳转目的地址是否相同,如果相同,这时就认为这两条条件跳转构成了条件跳转混淆。

在作出条件跳转混淆的判断后,需要对程序的控制流进行修改,此时控制流不能够再顺序地从条件跳转指令的下一字节处开始解码,而是需要将控制流转移至两条条件跳转指令共同的目的地址处,从该地址开始新一轮的反汇编。

需要说明的是,上面举例时所用到的互补条件跳转指令对是 `jz`、`jnz`,但是实际上在 `x86` 指令集中存在一组类似的指令,有符号数的互补条件跳转指令列表如表 14-1 所示,无符号数的互补条件跳转指令列表如表 14-2 所示。对于无符号数的条件跳转指令而言,为与有符号数相区别,将无符号数的大于改称为高于,无符号数的小于改称为低于。在实际应用中,恶意代码的编写者可以利用任意两条互补的条件跳转指令来对 `jz`、`jnz` 进行替换构造混淆,从而达到避免模板匹配检测的目的。所以在判断互补的条件跳转时需要将表 14-1 和表 14-2 中所列的所有情况均考虑在内。

多层顺序嵌套形式的处理情况类似,在图 14-17 中解码到 (3) 处指令 `jnz label1` 后,控制流将转移到从 (5) 处开始的 `label1` 代码块,从而不会对 (4) 处的垃圾数据 `thunkcode1` 进行解码。接着从 `label1` 处开始的解码与典型情况的处理相同,在解码到 (8) 处的指令 `jnz label2` 后,控制流转移到从 (10) 处开始的 `label2` 代码块,也能够避开 (9) 处的垃圾数据 `thunkcode2`。图 14-17 中 B 的代码情况与之类似,从而解决条件跳转指令多层顺序嵌套形式的混淆。

在多层乱序嵌套形式中,以图 14-18 中的 A 为例,当执行至 (13) 处的指令 `jnz label2` 后,控制流向后跳回至 (5) 处 `label2` 表示的代码块,接着分析到 (8) 处 `jnz label3` 后,控制流跳至从 (15) 开始的 `label3` 代码块。由于在反汇编过程中不对认定的垃圾数据进行填充,所以不会出现解码不完整的情况。对图 14-18 的处理情况类似。

表 14-1 有符号数的互补条件跳转指令列表

指令	等价形式	判断条件	说明
<code>JE</code>	<code>JZ</code>	<code>ZF=1</code>	两个数相等
<code>JNE</code>	<code>JNZ</code>	<code>ZF=0</code>	两个数不等
<code>JL</code>	<code>JNGE</code>	<code>SF ≠ OF</code>	第 1 个数小于第 2 个数
<code>JLE</code>	<code>JNG</code>	<code>SF ≠ OF 或 ZF=1</code>	第 1 个数小于或等于第 2 个数
<code>JG</code>	<code>JNLE</code>	<code>SF=OF 且 ZF=0</code>	第 1 个数大于第 2 个数
<code>JGE</code>	<code>JNG</code>	<code>SF=OF</code>	第 1 个数大于或等于第 2 个数



表 14-2 无符号数的互补条件跳转指令列表

指令	等价形式	判断条件	说明
JE	JZ	ZF=1	两个数相等
JNE	JNZ	ZF=0	两个数不等
JB	JNAE/JNB	CF=1	第1个数低于第2个数
JBE	JNA	CF=1 或 ZF=1	第1个数低于或等于第2个数
JA	JNBE	CF=0 且 ZF=0	第1个数高于第2个数
JAE	JNB/JNC	CF=0	第1个数高于或等于第2个数

## (2) 条件跳转混淆冗余代码形式分析

至于在两个条件跳转之间添加冗余代码的情况,则相对复杂一些。因为如果存在垃圾数据,那么必须在解码至第2条条件跳转之前利用 x86 指令的自修复特性重新同步到正确的指令,而且对垃圾数据误解码得到的指令不能够对程序中用到的堆栈、主要寄存器的内容进行修改,否则随意添加的垃圾数据将可能影响到代码的功能实现,产生意想不到的效果。所以认为在这种情况下恶意代码编写者添加垃圾数据的难度是非常大的,因此假设冗余代码中不存在垃圾数据。在这条假设下,两条条件跳转指令之间的冗余代码都能够被正确解码,冗余代码的作用仅仅是增加对反汇编结果的分析难度,而不影响反汇编的正确性,因此仍可沿用上面的分析过程。

但是还存在这样一种正常的编码情况,如图 14-20 所示,即两条条件跳转之间代码段 normalcode2 根据编程需要改变了标志位。这样即便两条条件跳转指令的条件互补且跳转地址相同,也不需要将控制流转移到第2条条件跳转指令的目标地址,而是继续从第2条条件跳转指令的下一字节开始进行顺序解码。

对于这种情况,则需要进一步对算法加以完善:首先判断两条条件跳转指令是否相邻,如果相邻的话,且跳转条件互补、目的地址相同,那么一定是条件跳转混淆。而对于不相邻的情况,中间插入的代码有可能是冗余的,但也有可能是正常的,这时就需要作出进一步判断,冗余代码的判断是一个非常复杂的问题,我们只就其中的对称代码形式进行判断。

```

(1)  normalcode1
(2)  jz   label
(3)  normalcode2
    (normalcode2 可能改变标志位)
(4)  jnz  label
(5)  normalcode3
(6) label:
(7)  normalcode4

```

图 14-20 正常代码

对于图 14-20 中的情况,通过一个新创建的链表来存储两条条件跳转指令中间遇到的指令。首先将遇到的第一条指令放入链表中,随后放入的指令则要先进行判断,看是否执行互逆的操作,其中对于 push、pop 等堆栈操作指令还需要考虑操作数是否相同,如果在链表中存在执行互逆操作的指令,那么不将该指令放入链表,同时删除链表中的互逆指令,否则将其放入链表,然后继续对下一指令进行判断。最后在处理到第2条条件跳转时,对该链表的长度进行判断,如果为 0 则表示中间遇到的为冗余指令,否则认为是正常指令。需要特别指出的是,冗余代码的具体实现形式纷繁复杂,我们所描述的算法仅能够判断对



称代码形式的冗余，它仅仅是实际冗余代码的一个子集。

最后给出改进算法主要部分的伪码表示，如图 14-21 所示。

```

if (instr ∈ CJI) {
    将 instr 的信息放进 pbrstll
    if (nextAddr 与 instr 的地址相同) sequen = true
    else sequen = false
    计算并存储当前条件跳转指令的下一条指令地址到 nextAddr
    if (pbrstll 的长度 > 1) {
        在 pbrstll 中查找 sinst
        if (sinst 存在并且其条件与 instr 互补) {
            if (sequen = true) {flag = true, sequen = false}
            else
                {if (predstll 为空 (flag = true
                else flag = false}
            }
        }
    }
}

```

图 14-21 改进算法伪码表示

在图 14-21 中，CJI 表示条件跳转指令集合；instr 表示刚解码的指令；sinst 表示链表 pbrstll 中与 instr 具有相同目的地址的指令；nextAddr 表示紧跟在条件跳转指令后面的指令地址；pbrstll 表示存储所遇到条件跳转指令信息的链表；predstll 表示存储在两条条件跳转指令之间的某种类型指令的链表；sequen 表示指明两条条件跳转指令是否相邻的标志位，如果为 true，那么所遇到两条条件跳转指令是相邻的，否则不相邻；flag 表示控制流是否转移的标志位，如果为 true，控制流不会转移，否则控制流将会转移到条件跳转的目标地址处。

### 14.4.2 指令重叠混淆

指令重叠作为一种主要的混淆技术，由 Fred Cohen 首先提出，虽然他并未对该混淆技术命名，但是出现了利用跳转指令回跳到之前指令内部字节处的思路。Cullen Linn 对指令重叠进行了定义，但是他认为能够满足指令重叠的指令相对较少，而且由于汇编指令的自修复现象，所以他对指令重叠的混淆效果并不看好。后来，Charles LeDoux 提出了指令嵌入方法实现了对指令重叠的形式的改进，即对于指令 I 与 J，指令 J 的所有字节都包含在指令 I 的后  $k$  字节中。通过这种方式，可以在某条指令的首部有意插入若干字节作为操作码，将原指令作为所插入操作码的操作数。然后再添加一条越过添加字节直接跳转到原指令的跳转指令。Charles LeDoux 提出的方法大大方便了指令重叠混淆的应用场景。Christopher Jämthagen 也提出了一种指令重叠的新方法，该方法利用空操作指令 (nop) 的多字节编码方式，通过构造封装指令实现指令重叠。在恶意代码中，指令重叠混淆与其他混淆技术相结合，诱导反汇编算法出错，提高分析难度，本节通过分析指令重叠混淆的原理，针对指



令重叠的抗混淆反汇编算法进行研究。

### 1. 指令重叠混淆的实现形式

Fred Cohen 提出的指令重叠的思想是, 将跳转指令的跳转目标地址指向特定的操作代码, 从而使跳转时的地址位于一条之前已经执行过的指令中间, 进而实现了对跳转目标地址处的代码的复用, 即作为另一条指令的操作码。

Cullen Linn 对指令重叠作出了定义, 对于两条相邻的指令  $I$  与  $J$ , 必须满足以下条件:

- 1) 执行流程不能从  $I$  流向  $J$ 。
- 2) 对于  $k > 0$ , 指令  $I$  的后  $k$  字节必须与指令  $J$  的前  $k$  字节相同。

Michael Sikorski 在其著作中提出了两种指令重叠混淆的实例, 分别如图 14-22 与图 14-23 所示, 本节对指令重叠混淆的分析主要以 Michael Sikorski 提出的实例为分析对象进行展开。

在图 14-22 中, 字节 EB、FF 构成指令 jmp-1, 字节 FF、C0 构成指令 inc eax。对于线性扫描反汇编算法, 解析完 EB、FF 字节后, 判断其为 jmp-1 指令, 此时控制流

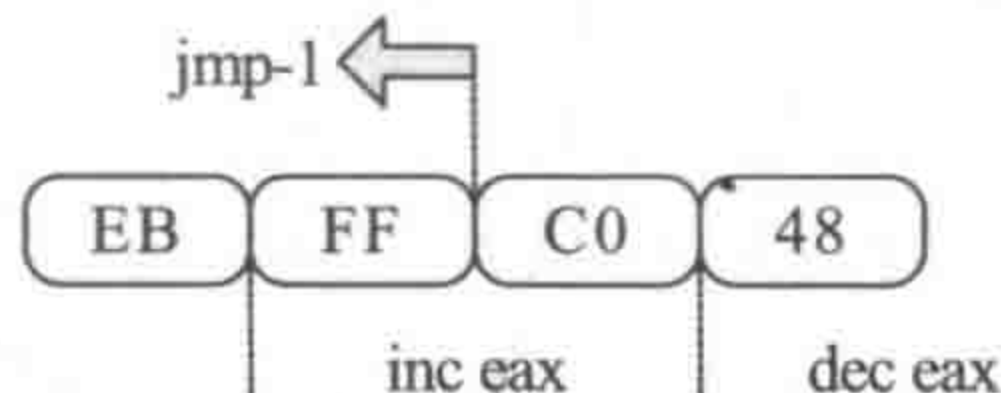


图 14-22 跳转目标地址为当前指令内部字节的情况

将从下一字节, 即 C0 开始解析, 并将其解析为 ror 指令。对于行进递归反汇编算法, 会将字节 FF 与字节 C0 的地址同时放入待解析队列, 分别从上述两个地址开始解析, 但实际 C0 处起始的路径并不会执行, 从而造成分析路径的冗余。

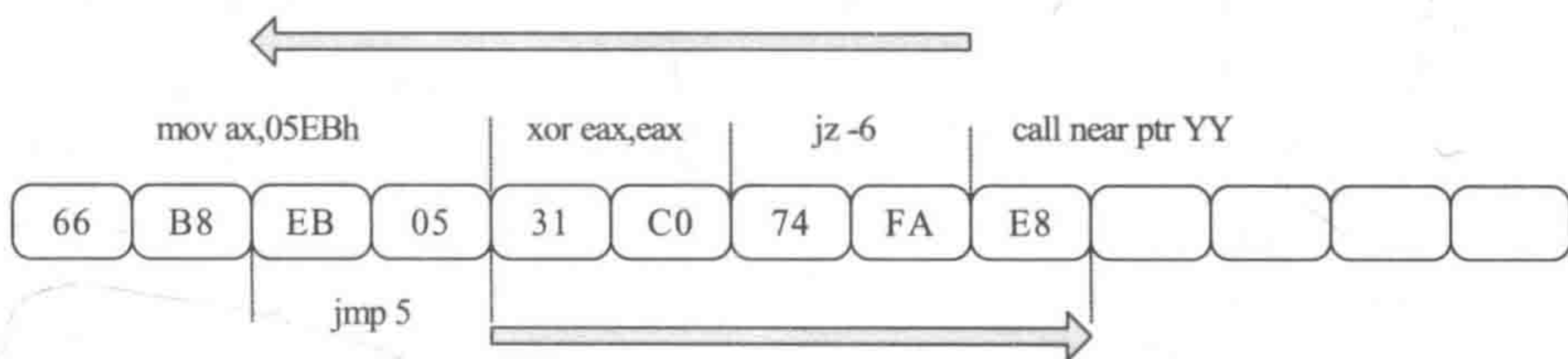


图 14-23 跳转目标地址为其他指令内部字节的情况

在图 14-23 中, 字节 EB、05 是指令 mov ax, 05EBh 的操作数, 同时字节 EB、05 又构成指令 jmp 5。对于线性扫描反汇编算法, 会陆续解析得到指令序列 mov ax, 05EB、xor eax, eax、jz short near XX、call near ptr YY (XX 为字节 EB 处的地址, YY 为字节 E8 后所指示的调用目标地址)。对于行进递归反汇编算法, 由于行进递归反汇编算法不将条件跳转指令看作控制流转移, 所以依然会得到与线性扫描反汇编相同的错误指令序列。

从对上面两个例子可以看出, 指令重叠混淆能够有效地影响反汇编算法的正确率, 进而影响恶意代码的分析效果。

### 2. 问题分析

通过对上节指令重叠的实现技术进行分析不难发现, 指令重叠混淆主要利用无条件跳



转指令或者条件跳转指令回跳至某一指令内部，并将该处的字节复用为操作码开始后续执行。其实现特点归纳如下：

### (1) 指令重叠混淆需要利用跳转指令

指令重叠需要将控制流转移到某条指令内部，完成这一控制转移需要利用跳转指令。根据跳转目标地址的区别，可以将其分为跳转目标地址在当前指令内部的 jump-current 型、跳转目标地址在之前已解析指令内部的 jump-backward 型、跳转目标地址在之后未解析指令内部的 jump-forward 型。本章主要关注 jump-current 型与 jump-backward 型指令重叠，对于 jump-forward 型的情况在后面进行说明。

在汇编语言中，跳转指令分为无条件跳转指令与条件跳转指令。

其中无条件跳转指令 jmp 主要有 3 种实现形式：

1) 短跳转 (Short Jmp)：跳转范围在  $EIP_{current} - 128$  到  $EIP_{current} + 127$  之间。只能跳转到 256 字节的范围内，对应机器码为 EB rel8。

2) 近跳转 (Near Jmp)：跳转至当前代码段中的某条指令（即 CS 寄存器当前指向的段），对应机器码为 E9 rel16/32。

3) 远跳转 (Far Jmp)：可跳转至具有相同权限的不同段的指令任意地址，有时指段间跳转，对应机器码为 EA ptr 16:16/32。

$EIP_{current}$  指示 EIP 寄存器的当前值，rel8、rel16 以及 rel32 指示跳转目标指令距当前指令的相对偏移。

短跳转和近跳转指令中包含的操作数都是相对于 EIP 的偏移，远跳转指令中包含的是目标的绝对地址。短跳转指令由于涉及字节少，使用较为方便，上述例子中的指令重叠所采用的跳转指令也主要是短跳转，但是通过选择字节以及借助编码过程中的标号，近跳转与远跳转亦可用作实现指令重叠。

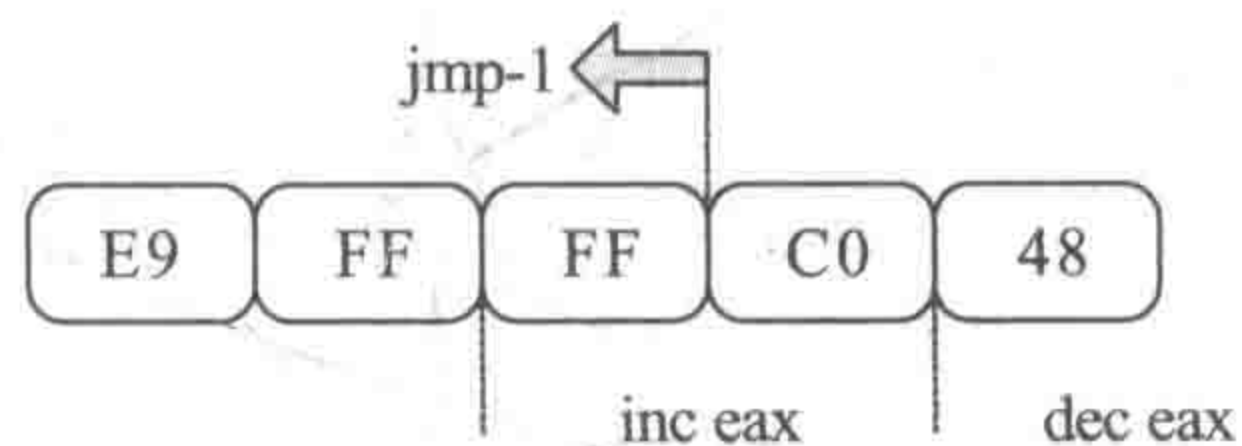


图 14-24 利用近跳转实现图 14-22 所示的指令重叠

图 14-24 展示了将图 14-22 中的短跳转替换为近跳转的 jump-current 型指令重叠。

图 14-25 展示了利用借助标号的远跳转所实现的图 14-23 中所示的 jump-backward 型指令重叠。

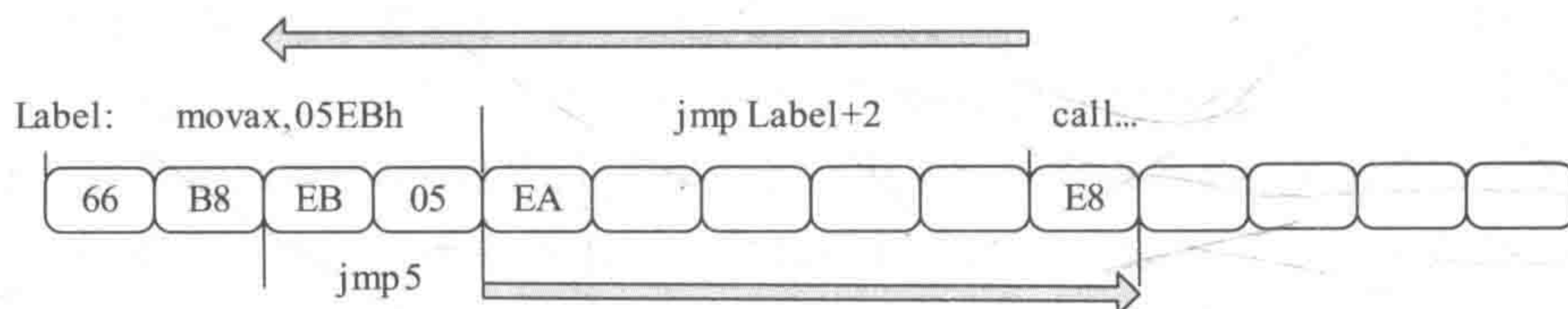


图 14-25 借助标号的远跳转实现图 14-23 所示的指令重叠

从图 14-24、图 14-25 可以看出，无条件跳转指令的跳转范围对于指令重叠混淆的构造没有影响。



相对于条件跳转而言,基于无条件跳转的指令重叠构造起来更加灵活。如图 14-23 中所示,为了触发条件跳转 jz 的跳转条件,需要构造指令置 ZF 标志位为 0。但是如果采用无条件跳转,则无需额外构造标志位置位指令。图 14-26 展示了用无条件跳转指令实现同图 14-23 功能相同的 jump-backward 型指令重叠。

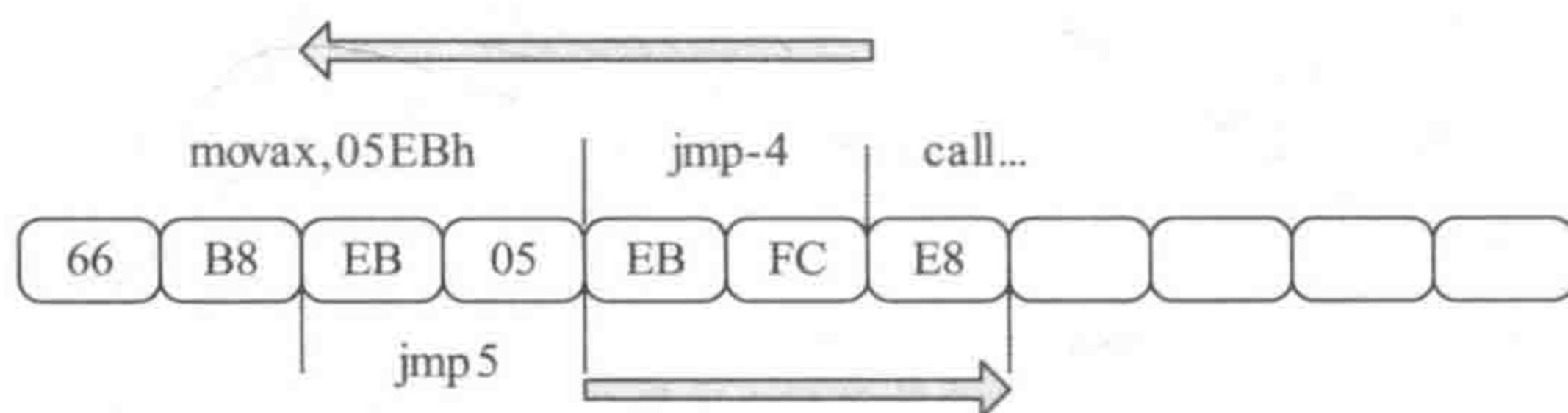


图 14-26 用无条件跳转实现图 14-23 所示的指令重叠

## (2) 指令重叠混淆的控制转移方式主要为回跳

指令重叠之所以成为影响分析的一种方法,在于其将之前已解析指令中间的若干字节作为一条新指令的前若干字节进行分析,而能够实现该目标,则需要利用跳转指令将控制流转移到之前已解析指令或者当前指令内部,控制转移方向均向后,即 jump-current 型与 jump-backward 型指令重叠。

对于控制转移方向为后续未解析指令的情况,即 jump-forward 型指令重叠,如图 14-27 所示,本章将其归为垃圾指令插入类混淆。该混淆亦会影响线性扫描反汇编算法的汇编准确率,并在同条件跳转相结合的情况下影响行进递归反汇编算法的正确率。

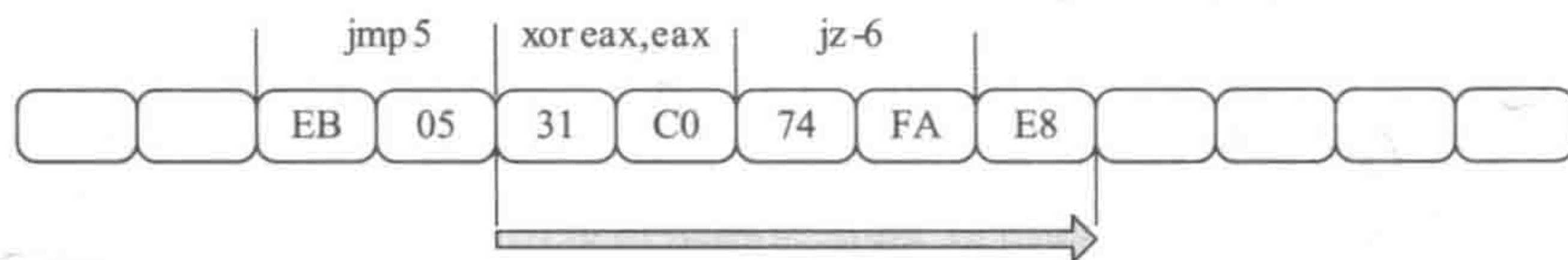


图 14-27 跳转目标地址为前方指令内部字节的情况

因此,本章关注的指令重叠混淆的控制转移方式主要为回跳。由于指令重叠混淆需要对相邻指令的操作码、操作数进行精心构造,所以跳转范围有限,通常在跳转指令所在基本块内。在此我们假设指令重叠混淆在同一基本块中实现,所以回跳指令的目标地址多小于回跳指令的地址。

根据对指令重叠实现特点的分析,基于行进递归反汇编算法设计了针对指令重叠的抗混淆反汇编算法。该算法通过对控制转移指令的类型、控制转移指令的转移方向、指令重叠的地址范围进行分析实现对指令重叠混淆的判断。

在反汇编过程中,遇到控制转移指令时,此处的控制转移指令包括无条件跳转指令、条件跳转指令(传统行进递归算法在处理条件跳转指令时不进行控制流转移)、调用指令以及返回指令,进一步对控制转移指令的类型进行判断。如果控制转移指令为跳转指令,则对跳转指令的跳转目标地址进行分析;如果跳转目标地址位于之前已解析指令或者当前指



令的内部，即跳转方向向后且跳转目标地址未在已解析指令起始地址集合中，则判断该段代码采用了指令重叠混淆技术。此时，仅将跳转目标地址放入转移地址集合，无需将跳转指令的下一条指令地址放入转移地址集合。之后则参考原始行进递归算法继续进行递归解析，用伪代码表示如图 14-28 所示。

```

if(instrJUMP)
// 判断当前指令是否为跳转指令
    if(addrjumpTarget-addrnext<0)
// 判断跳转指令是否为回跳指令
        if(addrjumpTarget∈ADDR)
// 判断回跳指令的目的地址是否为已解码指令的首地址
        {
// 仅将回跳指令的目的地址入队列
            insertTargetqueue(addrjumpTarget)
        }

```

图 14-28 预处理伪代码

其中  $\text{instr}_{\text{JUMP}}$  为属于跳转指令集合 JUMP 中的指令， $\text{addr}_{\text{jumpTarget}}$  为当前跳转指令的目标地址， $\text{addr}_{\text{next}}$  为当前跳转指令的下一条指令地址，如果两者差值小于 0，即表示当前跳转指令为回跳指令。ADDR 为已解析指令的起始地址集合。

在实现过程中，如果 ADDR 保存所有已解析指令集合，那么占用空间较大，且搜索判断过程耗时也会随之增加。根据假设，即指令重叠混淆在同一基本块中实现，所以在实现过程中 ADDR 仅保存当前基本块中已解析指令的起始地址。

针对指令重叠的抗混淆反汇编算法伪代码如图 14-29 所示。

对于分析范围内的地址  $\text{addr}$ ，首先判断该地址起始的指令是否已经完成解析，如果已完成解析则直接跳出函数。否则解析以  $\text{addr}$  为起始地址的指令，并设置解析标志，同时将已解码的地址存入已解析指令地址集合 ADDR。

接着判断当前解析指令是否为控制转移指令：① 如果为非控制转移指令，按顺序对当前指令的下一条指令进行解析，直到对指定代码区域中的所有代码完成解析。② 如果当前解析指令为控制转移指令，进一步判断被分析指令是否为跳转指令，如果为跳转指令，则用跳转目标地址同下一条指令地址之差判断是否为回跳指令，再进一步判断跳转目标地址是否在已解析指令地址集合 ADDR 中，如果上述条件均满足，则判断为指令重叠混淆，此时将回跳指令的目标地址存储在可能的控制流后继的集合 T 中。否则按照原始行进递归算法，将控制转移的目标地址以及控制转移指令的下一条指令地址存储在可能的控制流后继的集合中。

此处需要指出的是，指令重叠混淆中回跳指令的目标地址位于已解析指令的内部，而已解析指令地址集合 ADDR 所存储的均为已解析指令的起始地址，所以虽然回跳指令的目标地址指向已解析的地址范围，但是指令重叠混淆中回跳指令的目标地址并未在已解析指令集合 ADDR 中。



```

global startAddr, endAddr;
proc robustDisasmRec(addr)
{
    while (startAddr ≤ addr ≤ endAddr)
    {
        if (addr has been visited already)
            return;
        instr = DecodeInstr(addr);
        addr.visited = true;
        insertAddr(addr); // 将已解析指令的起始地址存入集合
        if(instr is a control transfer instruction)
        {
            if(instrJUMP)
            {
                if(addrjump - addrnext < 0)
                {
                    if(addrjumpTarget ∉ ADDR)
                    {
                        insertTargetqueue (addrjumpTarget)
                    }
                }
            }
            else
            {
                insertTargetqueue(instrtarget);
                insertTargetqueue(instrnext);
            }

            for each (target ∈ T)
                robustDisasmRec(target);
            return;
        }
        else
            addr += Length(instr);
    }
}

```

图 14-29 针对指令重叠的抗混淆反汇编算法伪代码

### 14.4.3 子程序异常返回

子程序调用通常使用 CALL 指令，在执行 CALL 指令时会将当前指令（也就是本条 CALL 指令）的下一条指令的地址（Addr1）放入栈顶，以便在子程序完成时继续向下执行。在子程序执行结束时，会用 RET 指令返回主程序，这时是将当前栈的栈顶取出（正常应该是 Addr1），继续执行主程序的操作。

利用子程序异常返回来干扰反汇编器工作属于控制结构混淆范畴，这种方法修改了程序的控制结构，但不会改变程序实际执行的内容。许多汇编器在调用指令后紧跟的是一条有效指令。但是，我们假设若干字节的无用数据被插入在此处。在 x86 指令集中 CALL 指令是由跳转和压栈动作组成，RET 指令的返回地址是当前的栈顶。但是通过分支过程能够修改栈顶，子程序异常返回的混淆技术就是利用了这个特点。



图 14-30 是通过子程序异常返回来干扰反汇编的一种典型情况。在主程序中插入无用数据 (E8)，然后在子程序返回之前调整堆栈，将真正的返回地址 (exit) 放入栈顶。

对于混淆情况，当程序执行到 CALL 指令调用子程序 (sub) 时，会自动将当前指令的下一条指令的地址 (Addr1) 放入堆栈中，在 sub 结束之前使用的 RET 指令会将当前的栈顶取出 (正常情况应该是调用该过程的 CALL 指令自动压入堆栈的 Addr1)，由于子程序在执行 ret 之前使用 push exit 调整了堆栈，因此此时取出的值应该是 exit，并将 exit 放入了 PC (程序计数器) 中，使得程序能够从该地址开始执行，这样就保证了程序能够跨过垃圾数据，使程序正确执行。

正常情况	混淆情况
sub: ..... ..... ret main: ..... call sub exit: .....	sub: ..... push exit 调整堆栈 ret main: ..... call sub Addr1: db db 0E8h 垃圾数据 exit: .....

图 14-30 子程序异常返回干扰反汇编的一种典型情况

IDA 对上例的混淆情况反汇编结果为：

```
.text:00401000 sub_401000 proc near ;
.....
.text:00401015 push 401034h
.text:0040101A retn

.text:0040101B start proc near
.text:0040102E call sub_401000
.text:00401033 call near ptr 1CA810A2h(反汇编出错)
.text:00401038 xor [eax+0], al(反汇编出错)
.....
.text:0040105A start ends
```

通常情况下，反汇编器认为 CALL 指令之后的内容也是有效指令，所以会将执行不到的垃圾数据 (E8) 和其后面的指令进行组合识别为其他指令 (call near ptr 1CA810A2h)。又因为 x86 指令集是变长指令集，所以指令的开始位置可以从任意地址开始。反汇编器是根据当前指令的长度来计算下一条指令的开始地址，如果当前指令识别出错，指令长度就可能不正确，导致下一条指令解码出错 (xor [eax+0], al)，这样就出现解码连续出错的情况。

而线性扫描和行进递归两种算法都不能对子程序异常返回的混淆情况进行处理，需要设计新算法解决这种情况。

针对压栈后返回的情况，如果利用线性扫描算法，不能对过程进行划分，从而无法跳过垃圾数据。我们选择以行进递归算法为基础，能够对过程进行划分，在划分的同时能够记录过程的调用关系，以便后期使用。首次对主程序解码时子程序还未解码，不能得到子程序实际返回地址，如果存在异常返回的情况，依然会将垃圾数据识别为指令。因此，需



要在全部子程序解码完成之后再次对主程序解码。

首次解码如图 14-31 所示, 针对每个过程 (Proc), 添加虚拟栈 (T) 记录对堆栈的操作。在过程开始解码之前, 将调用该过程的 CALL 指令的下一条指令的地址 (Addr1) 压入 T; 在解码过程中, 在识别出堆栈操作指令时, 同时在 T 上做相应的动作; 在当前过程解码完成时, 则以 Addr1 的值为索引填表 M, 表示该调用的子程序已解码, 并且记录实际的返回地址。比较 T 的栈顶是否与 Addr1 相同。若相同, 实际返回地址就是 Addr1, 在 M 中填入“0”, 表示子程序正常返回; 否则, 实际返回地址是 T 的栈顶值 (Addr2), 将 Addr2 填入 M, 将调用 Proc 的过程的开始地址放入二次解码堆栈 (DT) 中。在调用的所有子程序解码完毕之后, 开始二次解码。

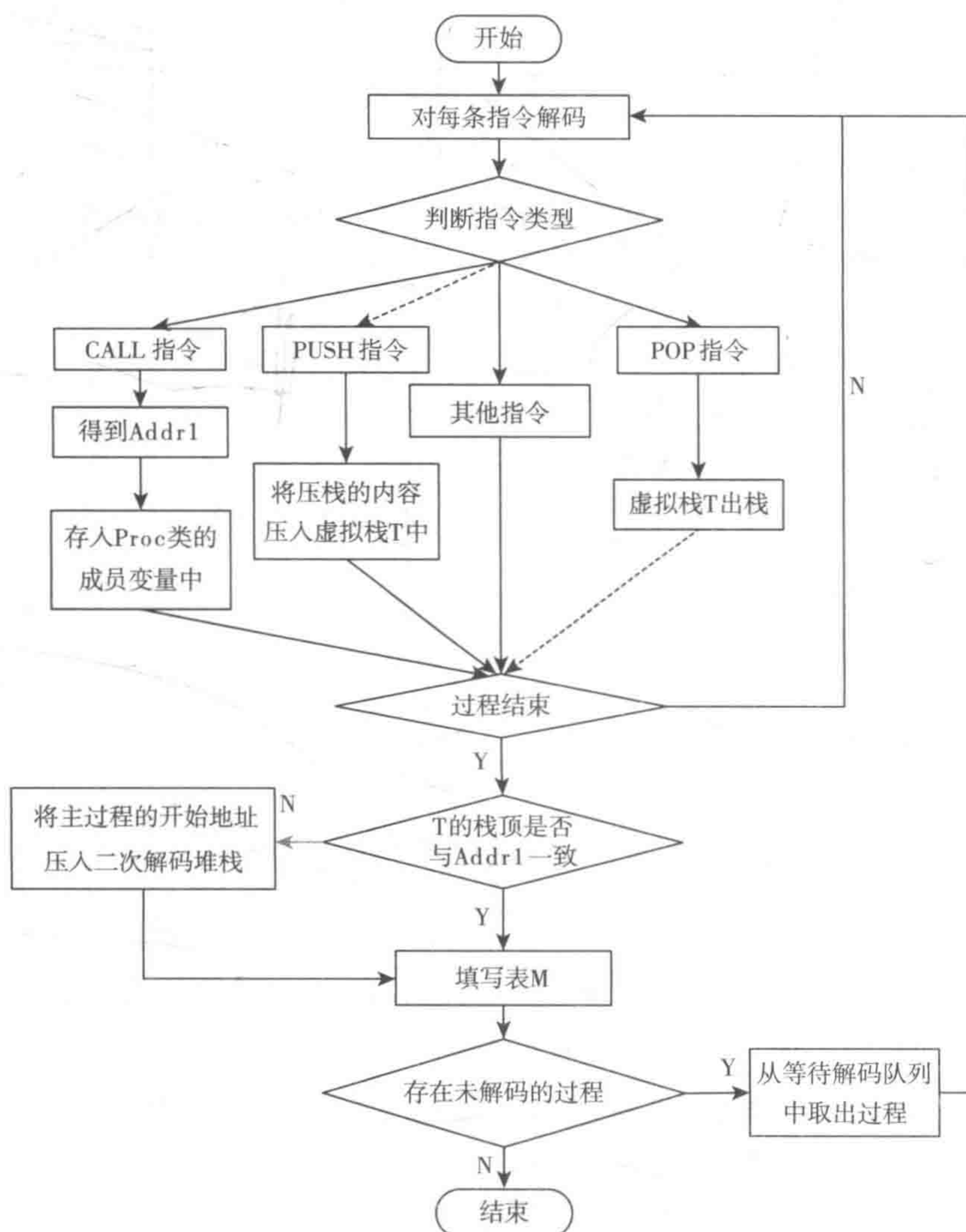


图 14-31 首次解码流程图



二次解码过程如图 14-32 所示，从 DT 中取出地址开始解码，当发现当前指令是 CALL 指令时，用 Addr1 值查表 M，若实际返回地址为“0”，继续连续解码；否则，下一条应该解码的指令的开始地址就是实际返回地址（Addr2）。本过程解码完毕之后，从 DT 中取出新的过程开始地址解码，直到 DT 为空，反汇编结束。

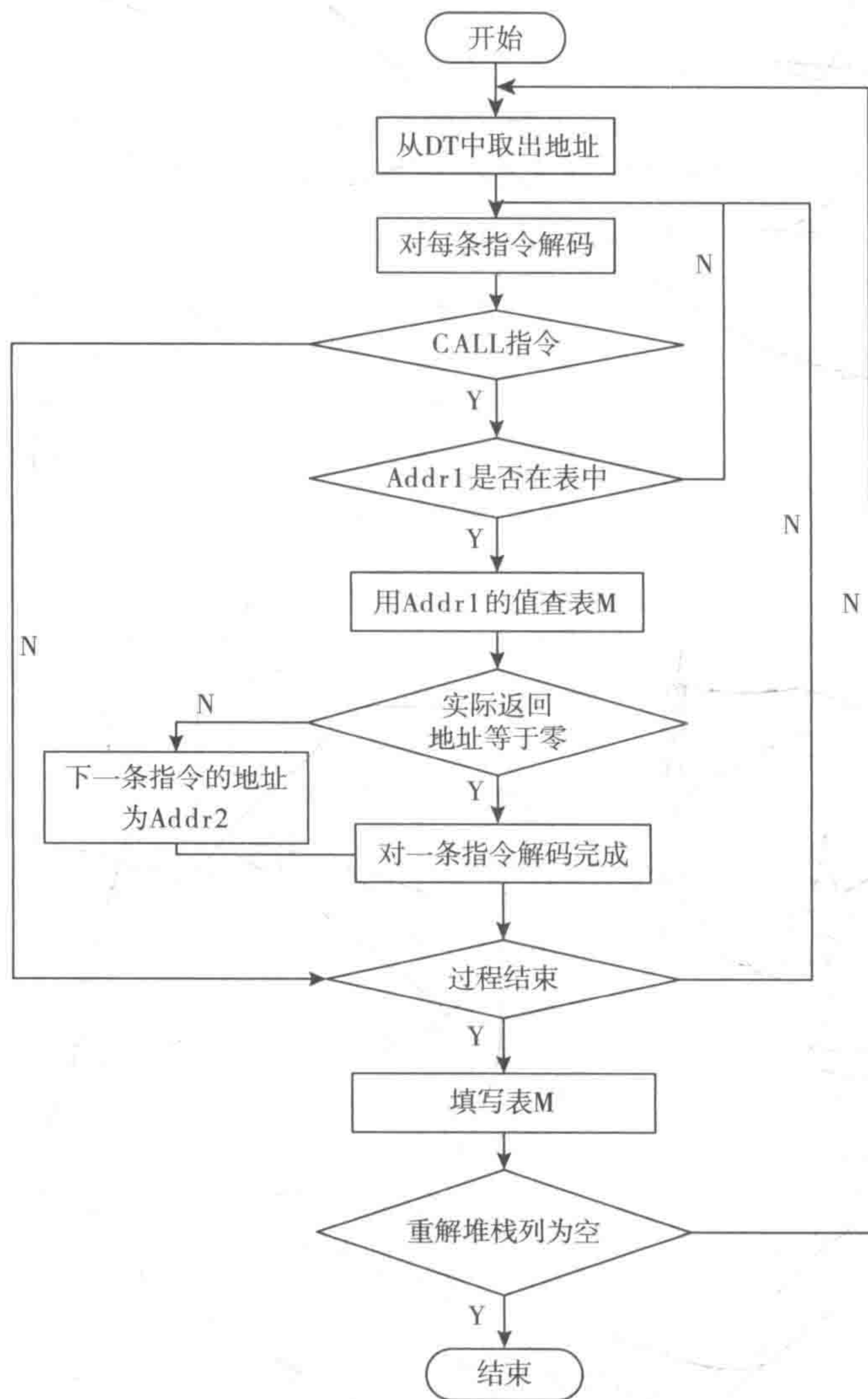


图 14-32 二次解码流程图

由于可能产生指令连续解码出错，因此可能造成将垃圾数据识别为指令，同时将垃圾数据之后的有效指令识别错误的情况。又由于 x86 指令集具有自动恢复的特点，在对“几条”指令解码出错之后，就会开始正确地解码。在绝大多数情况下，这“几条”解码出错的指令会在二次解码时得到修正，但这里有两种特殊情况需要进行处理。



**情况 1:** 在首次解码时, 如果 CALL 指令因为混淆没有被正确识别 (call sub2), 因此相应的子程序没有解码。该过程存在异常返回的情况, 没有被发现。如图 14-33 所示。

情况 1:	IDA 结果:
main:	.text:00401025 start:
.....	.....
call sub1	
db 0E8h	.text:00401038 call sub_401000
exit:	.text:0040103D call near ptr 3FE92Ah
call sub2	.....
db 0E8h	.text:00401042 dw 0E8FFh
.....	.....
end main	.text:0040105E_text ends

图 14-33 特殊情况 1

对此种情况, 在二次解码查表 M 时, 若 M 中没有对应的项, 说明该过程没有进行解码, 需要对其子程序重新解码, 来判断是否存在异常返回的情况。

处理方法: 中断本过程的二次解码, 先对子程序解码, 待子程序解码完毕, 填写 M, 再重新对主程序进行解码, 这时表 M 中就会有对应该过程的项, 解码正确完成。

**情况 2:** 由于在首次解码时, 因为混淆没有正确识别过程结束指令 (如 RET 指令), 过程之间就会发生过程重叠的情况 (sub2 与 sub1 重叠)。如图 14-34 所示。

情况 2:	IDA 结果:
sub1:	.text:00401000 sub_401000 proc near
.....	.....
call sub2	.text:00401015 call sub_401021
db 0E8h	.text:0040101A call near ptr 40506387h
exit2:	.text:0040101F add bl, al
push exit	.....
ret	
sub2:	.text:00401021 sub_401021 proc near
.....	.....
push exit2	.text:00401034 push 40101Bh ;
ret	.text:00401039 retn
main:	.text:0040103A start proc near
.....	.....
call sub1	.text:0040104D call sub_401000
db 0E8h	.text:00401052 call near ptr 1DA810C1h
exit:	.text:00401057 xor [eax+0], al
.....	.....
end main	.text:0040106D start endp

图 14-34 特殊情况 2

处理方法: 增加条件判断, 当前过程为 sub1, 被调用过程为 sub2, 判断 sub2 是否



在 sub1 中,若是就对 sub 1 进行划分,将属于 sub2 的内容从 sub1 中删除,使过程的划分正确。

#### 14.4.4 不透明谓词混淆

混淆的方法很多,使用不透明谓词进行混淆是其中比较难处理的一种。所谓不透明谓词就是其输出对混淆代码编写者易于判断而对攻击者来说难于推导的谓词。使用这样的谓词 P,在程序中某一点的输出经常保持恒为 false 或者 true,我们将其分别记作 PF 和 PT。不难看出,利用不透明谓词进行条件测试,它总是产生同样的结果。但是,要想通过逆向分析技术推测到此结果却非常困难。不透明谓词主要用来对控制流信息进行混淆。图 14-35 给出了利用不透明谓词混淆控制流信息的原理。语句 A、B、C 按照次序执行。通过不透明谓词增加一个恰当的控制条件,将程序转化成如图 14-35a 或者 b 的形式。表面上这个控制条件决定了 B 是否执行,但实际上由于混淆代码编写者对于不透明谓词的值是确定的,因此增加的干扰控制并不会影响到 B 的执行。但是对于逆向分析而言,对不透明谓词的分析难度是很大的,难以获知其取值固定这一特征,从而无法正确识别控制流信息。

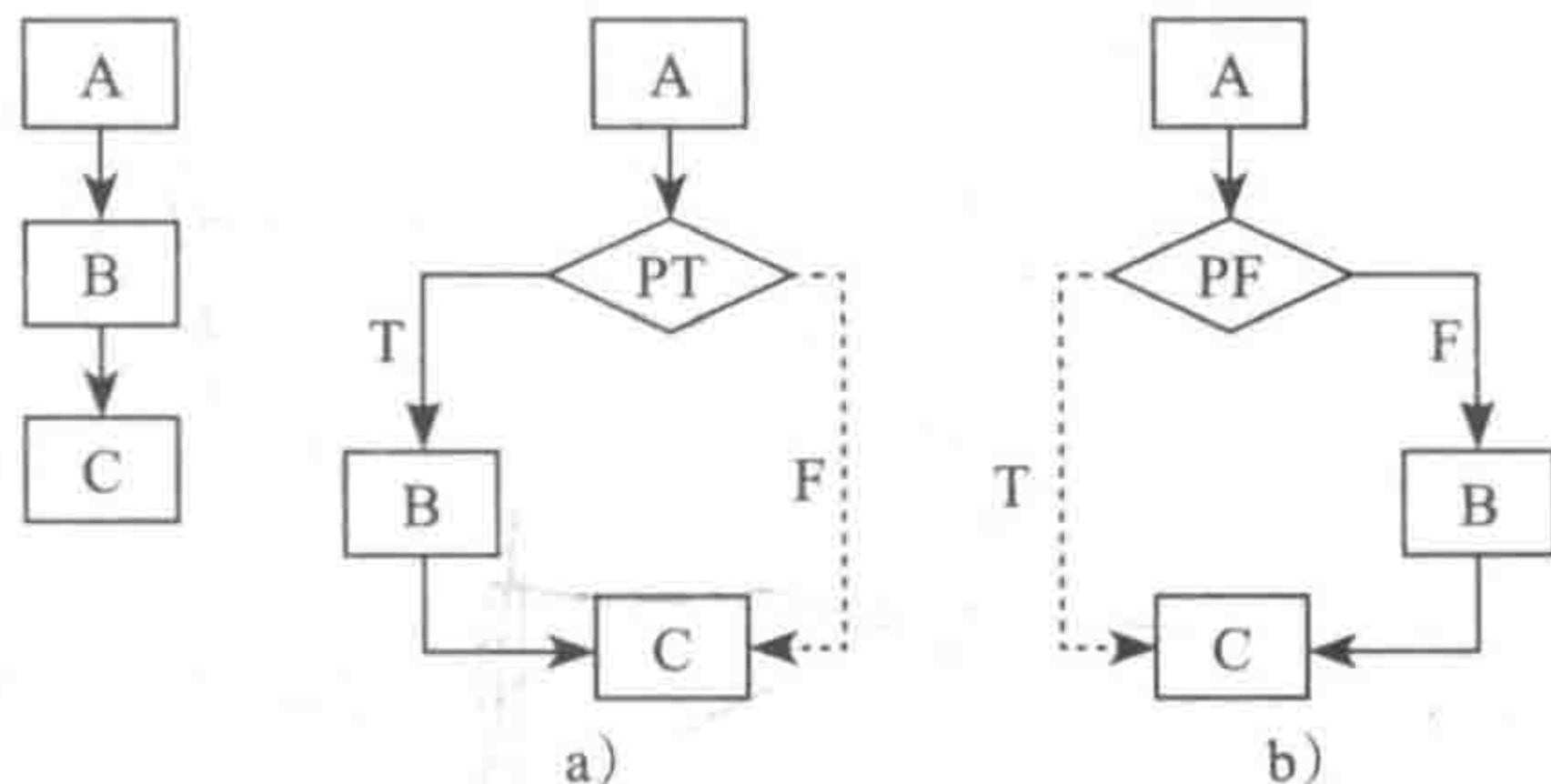


图 14-35 利用不透明谓词混淆控制流

而且,利用不透明谓词将二进制代码中的无条件跳转语句或顺序语句转换成以不透明谓词为条件的选择语句后,又为垃圾数据的插入增加了机会。转换后的条件选择语句有两个分支,这两个分支表面上看起来都是可能的执行路径,但是其中一个分支,如图 14-35a、b 中虚线所示,代表了永远不会执行的混淆路径。那么在这些混淆分支上,就可以插入垃圾数据而不会影响到程序的实际功能。采用行进递归算法的反汇编引擎遵循条件跳转有两个后继的假设,对两条执行路径均进行解码,从而受到垃圾数据的欺骗,导致反汇编结果出错。

不透明谓词通过增加混淆路径达到用于混淆控制流的目的,所以在逆向分析的过程中所要做的工作就是消除混淆分支,将结果加以简化。彻底解决不透明谓词混淆非常困难,不透明谓词主要集中于两种具体形式的不透明谓词混淆,即基于变量和基于简单代数恒等式的不透明谓词混淆形式。

##### 1. 不透明谓词

###### (1) 基于变量的不透明谓词

对基于变量的不透明谓词,以 switch 为例进行分析。在图 14-36 所示代码中,i 已被赋予一个常量 1,在实际执行过程中只会执行 case1 中的语句。但是在经过静态分析后得到的控制流图中,仍会有 case2 ~ case8 以及 default 各自所代表的执行路径,即增加了 8 条混



淆路径,从而在进行流图分析的过程中导致逆向分析工具的分析结果不准确、耗费分析人员较多的时间和精力。而且如果再进一步地将图 14-36 中 case2 ~ case8 以及 default 的 8 条分支中的 printf 语句改为垃圾数据,那么在反汇编过程中对该段代码的分析就将出现很多错误,后续分析的准确性更加无法保障。

图 14-36 以 switch 为例进行了说明,在实际应用中基于变量的不透明谓词混淆形式同样适用于 if...else...以及 while 等其他需要执行路径选择的情况。

## (2) 基于代数恒等式的不透明谓词

对基于代数恒等式的不透明谓词形式,以图 14-37 中的代码为例进行分析。图 14-37 中表达式 expr1、expr2 可以是常数 1、99、1001 等,也可以是变量 i、u、x 等,还可以是一个计算表达式 1+2、x+100 等。其中如果 expr1 与 expr2 是相同的表达式,那么即构成代数恒等式。

当 expr1、expr2 相同时,例如  $9==9$ 、 $12*6/3==12*6/3$ 、 $x==x$ 、 $v+99==v+99$  等代数恒等式的比较结果恒为真,所以在执行过程中仅会执行 if 分支中的指令。类似地,如果 expr1、expr2 不同,即代数恒等式永远不成立时,执行过程只会执行 else 分支,而不会执行 if 分支。

## 2. 问题分析

针对上面提到的基于变量的不透明谓词以及基于代数恒等式的不透明谓词,分别分析如下。

### (1) 基于变量的不透明谓词混淆分析

在基于变量的不透明谓词中,决定程序执行路径的分支变量即是不透明变量。由于分支变量的值其实已经确定,不再变化,所以只要取得该值即可确定执行哪条分支。这里需要注意的是对分支变量的定义不一定与分支判断语句相邻,中间可能间隔有其他语句。比如图 14-36 中的 `j=othernumber1` 和 `k=othernumber2`。

对此,采用程序切片技术来解决这个问题,即对不透明变量进行切片。具体到本例,即对分支变量 i 做切片得到它的常量值,再根据其常量值进行控制流流向的判断。其中需要特别提出的是在分析基于变量的不透明谓词混淆的过程中,我们基于中间表示 RTL(Register Transfer Lists, 寄存器传递列表)进行程序切片。

```
i=1;
j=othernumber1;
k=othernumber2;
switch(i){
case 1:
    printf("Case 1:\n");break;
case 2:
    printf("Case 2:\n");break;
case 3:
    printf("Case 3:\n");break;
...
case 8:
    printf("Case 8:\n");break;
default:
    printf("Case default:\n");
}
```

图 14-36 基于变量的不透明谓词

```
if (expr1==expr2){
    printf("if statement.\n");
}
else{
    printf("else statement.\n");
}
```

图 14-37 基于代数恒等式的不透明谓词



基于具有一定抽象度的中间表示 RTL 进行切片是比直接基于汇编指令进行切片更好的选择。在具体实现中,分以下三个步骤对基于变量的不透明谓词混淆进行分析(见图 14-38):

Step1: 对分析程序采用行进递归算法进行反汇编,同时对每个基本块新建栈并存储已解码指令的中间表示(即 RTL)。

Step2: 当反汇编到 switch 语句时,获取相应的分支变量。通过新建栈进行反向切片得到分支变量的最初定义。反向切片的终止条件为遇到常量值或者分析至过程的起始位置。

Step3: 用切片得到的结果对 switch 语句的分支进行选择。在后面的反汇编过程中仅对与切片结果相对应的分支进行解码,并修改相应基本块,将 *n*-way 的基本块类型改为 fall-through 基本块类型。

if...else...结构以及 while 结构可以进行类似的处理。

最后需要进一步说明的是,在对基于变量的不透明谓词进行程序切片操作时,之所以放到反汇编阶段进行分析,是因为此阶段完整的控制流图尚未生成,可以根据分析结果直接对控制流图进行化简,尽早处理基于变量的不透明谓词,在后续阶段则可集中精力进行反汇编阶段难于分析的混淆形式的处理。

## (2) 基于代数恒等式的不透明谓词混淆分析

对于基于代数恒等式的不透明谓词,我们进一步把它分解为下面两种情况。

1) *expr1*、*expr2* 均为常量或者常量表达式的常量代数恒等式,如  $9==9$ 、 $12*6/3==12*6/3$  等。

2) *expr1* 或 *expr2* 中包含变量或者变量表达式的变量代数恒等式,如  $x==x$ 、 $v+99==v+99$  等。

### 常量代数恒等式

在基于代数恒等式的不透明谓词中,如果恒等式两边是常量或者常量表达式的形式,分析过程仍在反汇编阶段进行。这是因为如果在反汇编阶段之后的流分析阶段进行分析,那么由于此时已进行了部分代码优化工作,一些汇编代码所传递的信息被删除,所以无法准确地识别汇编指令,从而不利于分析。

图 14-39 展示了一组采用常量代数恒等式混淆的测试实例。在 *expr1* 与 *expr2* 相等的情况下(以  $1==1$  为例),得到图 14-39 中 A 段所示的反汇编结果;如果 *expr1* 与 *expr2* 不等(以  $1==2$  为例),则得到图 14-39 中 B 段所示的反汇编结果。



图 14-38 基于变量的不透明谓词混淆分析流程

```

A:
    if (1==1){...
00411A4E  mov     eax,1
00411A53  test    eax,eax
00411A55  je      main+47h (416D77h)

B:
    if (1==2){...
00411A4E  xor     eax,eax
00411A50  je      main+42h (416D72h)
  
```

图 14-39 常量代数恒等式



从上面的反汇编结果可以看到, 基于常量代数恒等式的不透明谓词主要涉及三条汇编指令, 即 `test`、`xor` 和 `je`。执行时首先根据 `test`、`xor` 指令对操作数进行比较并修改相应的标志位, 然后 `je` 指令根据标志位对控制流进行选择。下面对这三条指令作简要介绍。

**xor**: 两个操作数进行逻辑“异或”操作, 结果存入目的操作数中, 当两个操作数相同时将寄存器的内容清零。

**test**: 两个操作数进行逻辑“与”操作, 设置 ZF、SF、PF 标志位。`test` 指令常用于测试一个数的某一位或几位是否为 1。是的话, 则置 ZF=0, 否则置 ZF=1。

**je**: 如果上一指令计算结果等于 0 则进行跳转。即 ZF=1 时进行跳转, 否则不跳转。

在后续的分析过程中通过测试可以发现, 当两个操作数相同时, `test` 指令与 `or` 指令的操作结果相同, `xor` 指令与 `sub` 指令的操作结果相同, 从而造成无法准确地对指令进行识别。这就是为什么把基于常量的代数不等式的情况放至反汇编阶段进行分析的原因, 因为此时不需要考虑优化过后的情况, 能够准确地对汇编指令的类型加以识别。

当两个操作数 `expr1` 与 `expr2` 相同时: 对于 `test` 指令, ZF 标志位置 0, 此时, `je` 条件不满足, 不会执行 `else` 分支。

当两个操作数 `expr1` 与 `expr2` 不同时: 对于 `xor` 指令, ZF 标志位置 1, `je` 条件满足, 此时不会执行 `if` 分支。

通过上面对基于常量代数恒等式的不透明谓词混淆的分析, 提出了如下解决方法。

首先从保存有解码指令的栈中弹出条件跳转指令 `je` 之前的一条指令, 判断其指令类型及两个操作数是否相等, 根据结果分别设置相应的标志位。在生成控制流图时, 根据标志位改变基本块的类型并设置相应下一条开始解码的地址。

具体而言, 在判断指令类型时:

- 如果 `je` 之前的一条指令类型为 `test`, 并且该指令的两个参数相等, 那么不把 `else` 分支的起始地址放入待解码的地址序列, 不创建以该地址为起始地址的基本块。
- 如果 `je` 之前的一条指令类型为 `xor`, 并且该指令的两个参数相等, 那么不把 `if` 分支的起始地址放入待解码的地址序列, 不创建以该地址为起始地址的基本块。

对于 `expr1`、`expr2` 为常量表达式的情况, 反汇编结果如图 14-40A、B 所示, 在反汇编的过程中会将常量运算表达式直接计算出结果, 从而将再次转变为常量的情况, 其处理方法与前面相同。

### 变量代数恒等式

在基于常量恒等式的不透明谓词混淆的反汇编结果中, 可以发现根据常量恒等式成立与否分别采用 `test-je` 组合 (常量代数恒等式成立的情况) 或者 `xor-je` 组合 (常量代数恒等式不成立的情况) 来实现路径的转移。而通过对基于变量代数恒等式的不透明谓词混淆反汇编结果

A: 恒等式成立	
<code>if (1000-999==1000-999){...</code>	
00411A4E	<code>mov eax,1</code>
00411A53	<code>test eax,eax</code>
00411A55	<code>je main+36h (411A66h)</code>
B: 恒等式不成立	
<code>if (1000-999==1000-998){...</code>	
00411A4E	<code>xor eax,eax</code>
00411A50	<code>je main+31h (411A61h)</code>

图 14-40 常量代数恒等式 (基于常量表达式)



进行分析，可以发现其与前者有所不同。基于变量代数恒等式的不透明谓词混淆反汇编结果不具有明显的指令组合特征，分支转移均是通过 `cmp-jne` 组合来进行控制。如图 14-41A、B 所示，其中 B 中的 `x`、`y` 被赋予不同的值。

图 14-41 中所示的反汇编结果可以看出变量代数恒等式的反汇编代码通用性更强，变量代数恒等式的成立与否与其指令组合形式没有明显关系。显然，像前面处理基于常量恒等式的不透明谓词混淆一样通过代码组合形式来判断执行路径的方法此处不再适用了。

下面简要介绍其中用到的指令。  
**cmp**：比较两个操作数的大小，根据比较结果设置相应的标志位 `CF`、`OF`、`SF`、`ZF`、`AF`、`PF` 等。比较操作是通过减法来进行的，标志位的设置与 `sub` 指令相同。但 `cmp` 指令不改变目的操作数的值。

A: 恒等式成立	
if (x==x){...	
00416D55	mov eax,dword ptr [x]
00416D58	cmp eax,dword ptr [x]
00416D5B	jne main+4Dh (416D7Dh)
B: 恒等式不成立	
if (x==y){...	
00416D5C	mov eax,dword ptr [x]
00416D5F	cmp eax,dword ptr [y]
00416D62	jne main+54h (416D84h)

图 14-41 变量代数恒等式

**jne**：上一指令的计算结果不等于 0 时进行跳转。即 `ZF=0` 时进行跳转，否则不跳转。  
其中 `dword ptr [x]` 表示地址 `x` 指向的一个双字单元。  
处理基于变量代数恒等式的不透明谓词是在反汇编阶段完成之后的数据流分析阶段进行的，因为这时能够利用代码化简后的信息，而且这时控制流图已经生成，在分析过程中可用的信息更多。解决方法的基本思想与之前类似：遇到条件跳转指令 `jne` 后，从存储当前基本块指令的栈中弹出该条件跳转指令前的一条指令，然后根据对指令的操作符的类型以及其两个操作数是否相等的分析结果来设置不同的标志位，最后根据标志位修改控制流图，删除其中判定的混淆分支。

中间表示 SSA 可以大大方便程序切片分析过程。  
对于不涉及变量运算的代数恒等式，对恒等式两边的变量分别进行切片分析，这时的切片是通过对 SSA 进行分析进而直接提取出其定义－引用链信息，然后判断切片结果是否是常量 (`opIntConst`) 或者内存操作数 (`opMemOf`)，如果是的话，停止切片，否则继续切片进行此分析过程，直到得到常量或者内存操作数。算法伪代码如图 14-42 所示。

```
操作数类型为 opSubscript 置标志位 subflag 为 true
if ( subflag ) {
    通过下标取得相应的赋值语句，并进一步获得该赋值语句的右值 rhs
    if ( rhs->getOper( )==opMemOf 或者 rhs->getOper( )==opIntConst ){
        将该右值放至比较变量
    }
    if ( rhs->getOper( )!=opSubscript ) {
        subflag=false;
    }
}
```

图 14-42 变量代数恒等式不透明谓词混淆解决算法伪代码表示 1



对于变量代数恒等式而言同样涉及变量运算表达式的情况,如图 14-43 所示,A、B 分别为恒等式成立与恒等式不成立的情况。

```

A:
    if (x+100==x+100){...
00411A55    mov eax,dword ptr [x]
00411A58    add  eax,64h
00411A5B    mov ecx,dword ptr [x]
00411A5E    add  ecx,64h
00411A61    cmp  eax,ecx
00411A63    jne  main+44h (411A74h)

B:
    if (x+100==x+1){...
00411A55    mov eax,dword ptr [x]
00411A58    add  eax,64h
00411A5B    mov ecx,dword ptr [x]
00411A5E    add  ecx,1
00411A61    cmp  eax,ecx
00411A63    jne  main+44h (411A74h)

```

图 14-43 变量代数恒等式(基于变量表达式)

对于涉及变量运算的代数恒等式,则需要进一步考虑运算类型,针对常见运算进行分析。基于图 14-43 所示算法进行如图 14-44 所示修改。

```

if ( 操作数类型为 opSubscript ) {
    通过下标取得相应的赋值语句,并进一步获得该赋值语句的右值 rhs
    switch ( rhs->getOper() ){
        case opSubscript: 取得对应的赋值语句并对当前右值 rhs
进行替换
        case opMemOf:      将值存储至比较变量
        case opIntConst:   将值存储至比较变量
        case opPlus:       将常量存储至比较变量,表达式替换当
前右值 rhs,
                                对替换后的 rhs 继续进行此分析过程
        case opMinus:      处理过程与 opPlus 类似
        case opMult:       处理过程与 opPlus 类似
        case opDiv:        处理过程与 opPlus 类似
    }
}

```

图 14-44 变量代数恒等式不透明谓词混淆解决算法伪码表示 2

其中 `cmp` 表示需要比较的表达式, `getOper()` 用于取得该表达式的操作符以及获得该表达式的类型。如果操作符类型为 `opSubscript`, 即该语句有下标, 能够利用定义-引用链进一步获得。

对 `expr1` 进行上面操作后, 对 `expr2` 进行类似的操作。



最后分别对 `expr1`、`expr2` 的切片结果进行比较。如果在对 `expr1`、`expr2` 进行切片的过程中涉及表达式的计算，那么需要首先比较运算符，并进而对计算过程中所涉及的常量以及内存操作数进行比较。如果全部相等的话，则判断该基于变量的代数恒等式成立，否则恒等式不成立。

根据恒等式是否成立，从控制流图中删除相应的基本块以及与之相应的出边、入边。

## 14.5 实例分析

本节通过实例介绍一套作者课题组开发的基于反编译的恶意代码检测与标注系统 Radux 的设计与实现。该检测框架建立在反编译平台的基础之上，通过反编译逆向分析提升目标程序的语义信息，从而进一步获取其行为特征。另外，对于某已知恶意软件的新型变体，可以通过反编译逆向分析技术获取相关信息，再与已知的行为特征进行匹配。一旦匹配成功，便可以认为目标可执行程序是恶意的。

### 14.5.1 系统设计

基于反编译的恶意代码检测与标注系统 Radux 主要包括三个模块，分别是：反编译模块、行为检测模块和标注模块。各模块之间的组织流程如图 14-45 所示。

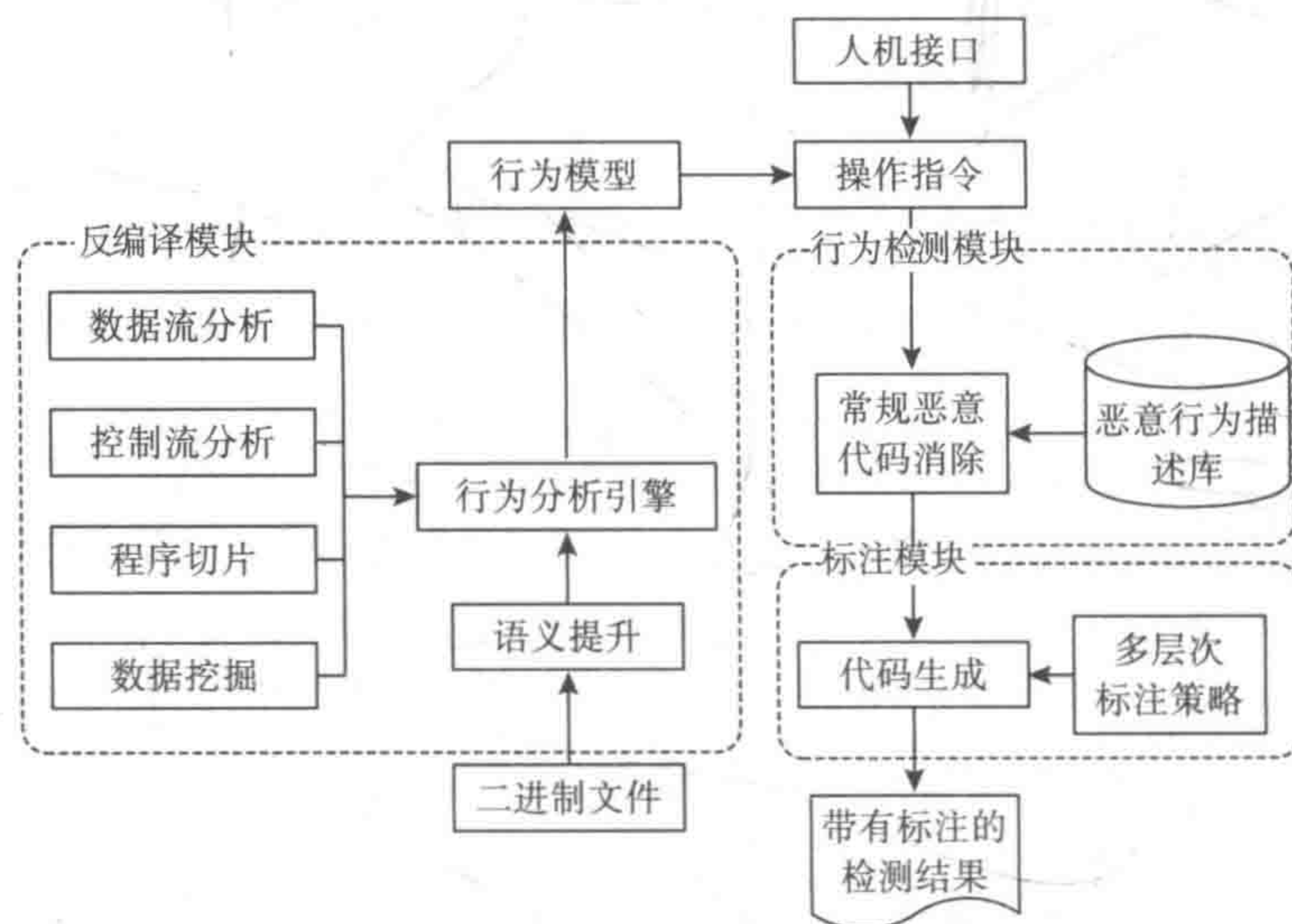


图 14-45 Radux 原型系统模块划分

各个模块的具体功能如下。

#### (1) 反编译模块主要功能

利用反编译引擎收集并分析二进制可执行程序的语义信息，进而对其进行抽象和取舍。在此基础之上，将语义分析的中间表示结果提交给行为分析引擎进行分析，挖掘单元指令



或指令片段之间存在的数据流关系和控制流关系。

### (2) 行为检测模块主要功能

在反编译分析中程序装载与控制流图构建的基础上,对目标代码的函数调用序列、控制流以及程序结构进行分析,完成与预定义的可疑行为特征库的基于加权相似度的模糊识别;通过对模糊理论的研究,将模糊推理应用于可疑代码恶意性的判定中,在行为模糊识别的基础上,依据基于贝叶斯算法所建立的规则库,进行恶意行为识别。

### (3) 标注模块主要功能

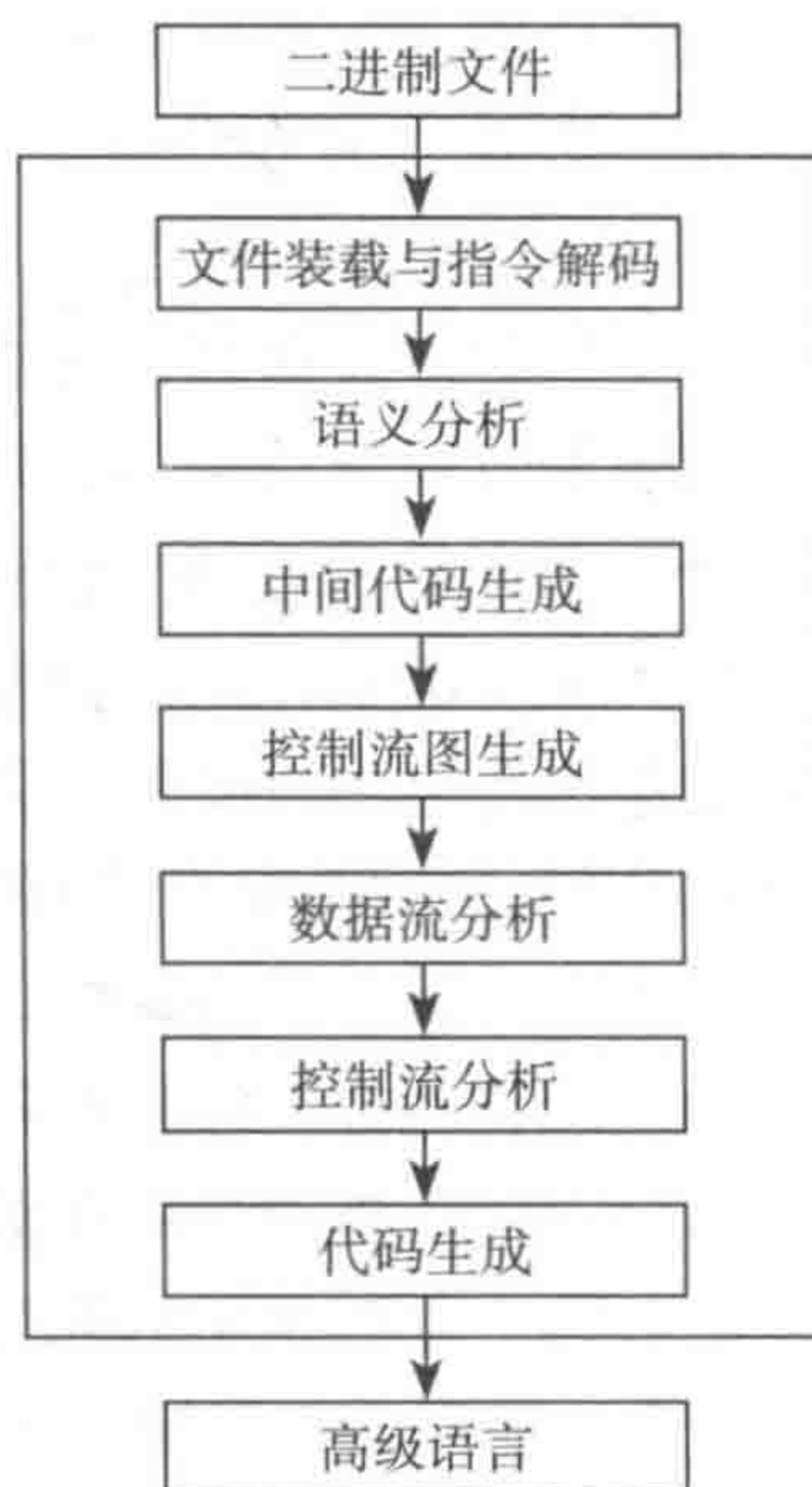
根据反编译不同阶段的检测策略设计相应的标注方案,充分展示不同层次的恶意行为检测结果。需要同时考虑到各层反编译结果与检测结果的特点,以便保证标注的准确性与合理性。设计图形化标注方法,实现多层次标注结果的相互协同,同时增强标注结果的表现能力。针对检测结果的特点,设计描述语言以规范恶意行为描述,以便标注结果的存储、分析、协同与复用。

## 14.5.2 系统模块划分

### 1. 反编译模块

反编译模块的核心为反编译分析引擎,其功能类似于常见反编译器的程序分析功能。可以进一步将该引擎细化为文件装载与指令解码、语义分析、中间代码生成、控制流图生成、数据流分析、控制流分析、代码生成等模块。如图 14-46 所示。

反编译模块的主要功能在于通过对目标可执行程序进行逆向分析和语义提升,将以二进制代码形式存放的可执行程序代码提升为某种高级的、与机器平台无关的中间表示代码。其中指令解码和指令语义映射可以将目标可执行代码等价变换为低级中间表示形式——RTL。为了便于程序的分析,通过数据流分析方法分析关键变量在程序中的生存区间,以及关键变量的定义-引用信息;通过控制流分析方法构建程序控制流图和调用图,并且初步分析控制流图中节点之间的支配关系。基于数据流分析与控制流分析结果,建立关键指令与其依赖指令之间的关联,再结合流图中的区间分析,最终完成高级控制结构(如 IF-THEN-ELSE 和 LOOP-UNTIL)的恢复。代码生成的作用主要体现在标注模块中,但为了体现反编译模块的完整性,故在此处对该模块进行说明。其主要功能是结合反编译分析的中间表示代码,生成带有恶意代码标注的某种高级语言(通常为 C 语言)的程序代码。



下面对反编译模块所涉及的主要子模块进行详细说明。 图 14-46 反编译模块框架结构图



### (1) 文件装载

文件装载模块的功能为：依据用户的命令，装入需要检测的可执行的二进制文件；判断文件类型，主要识别 PE 格式可执行程序；提取文件的符号表和段信息；为解码模块提供必要的信息，包括各个段的描述信息、程序主入口点、主函数位置等。

该模块输入为二进制可执行程序，输出为文件头信息和文件的内存映像。

在处理过程中，文件装载模块首先读入文件内容，根据文件头中的信息判断文件的格式，提取文件的段信息、符号表和导入表等信息。程序流程图如图 14-47 所示。

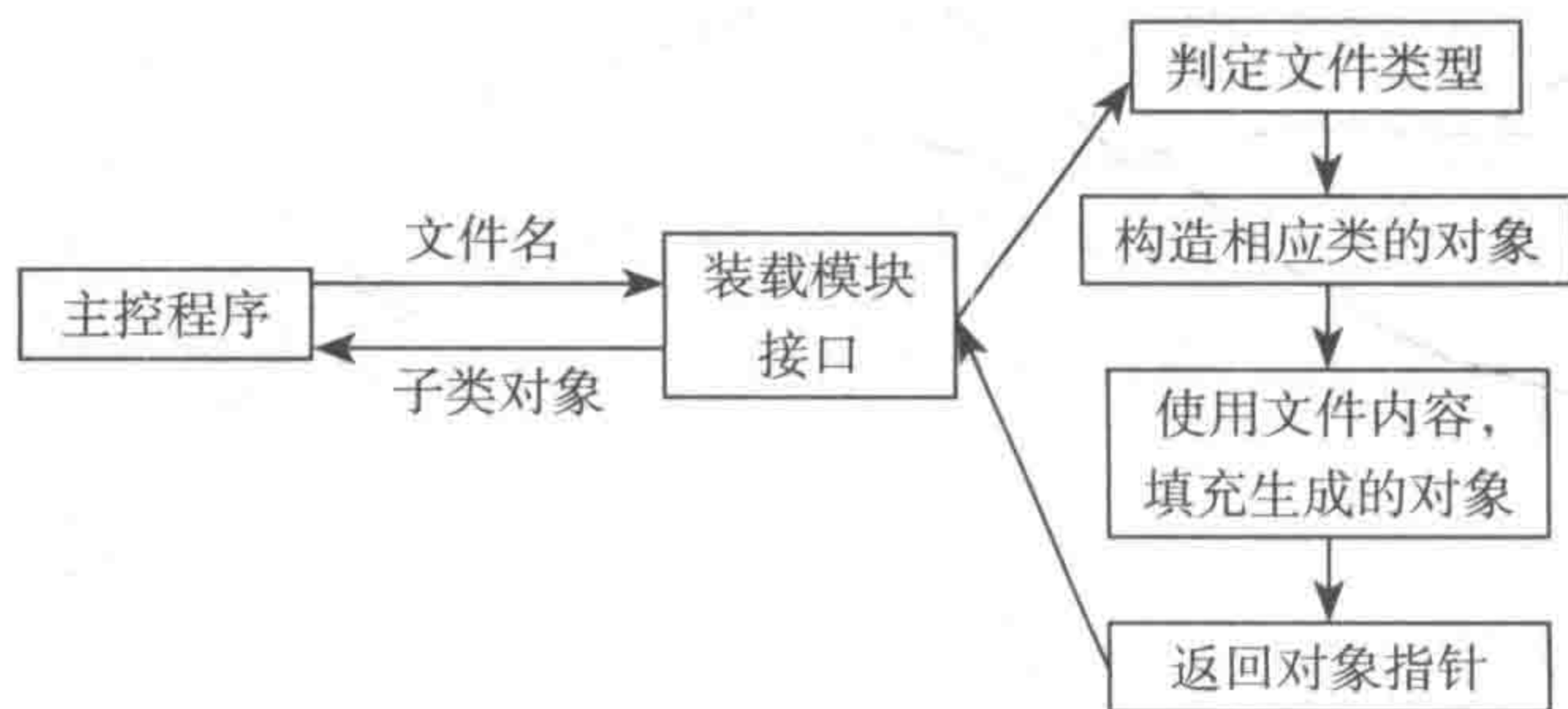


图 14-47 文件装载模块流程图

### (2) 指令解码

指令解码模块的功能为：对程序的主入口点进行试探性解码；从主入口点对指令流进行反汇编处理，完成二进制指令流到汇编表示的转换；同时完成基本块的划分和控制流图的生成。

该模块输入为文件装载之后得到的文件映像和从文件中提取的重要信息，输出为该程序的控制流图。

对一个过程进行解码时，生成一单链表 Ptrl，每解码一条指令便把所识别出的指令链接到该链表尾部。当遇到过程内转移类指令时，把转移的目标地址放到该过程的 targets 队列中，如遇到过程调用指令，则置目标地址于调用队列中，然后创建一个新的基本块并把当前指令链表中的所有指令放到这个基本块中，最后一条指令的类型即为该基本块类型。如果转移类指令的目标地址是在某一已存在的基本块内部，则对该基本块进行划分操作，最终要确保每个基本块的第一条指令是控制流的唯一入口，最后一条指令是唯一出口。增加“顺序解码”的判断保证了解码是基于控制流的。比如当前解码指令是无条件跳转指令，把目标地址放入 targets 队列后，置“顺序解码”为“假”，跳出顺序解码的循环体，从 targets 队列中取队首地址，再进行顺序解码。一个过程解码结束后，从调用队列中取下一过程的入口地址，判断是否是库函数和是否已经解码，如果判断为未解码的用户函数，则进行该过程的解码。直到调用队列为空，对二进制文件的解码才结束。如果指令解码中遇到间接调用指令，还需要作间隙代码分析。指令解码模块流程如图 14-48 所示。



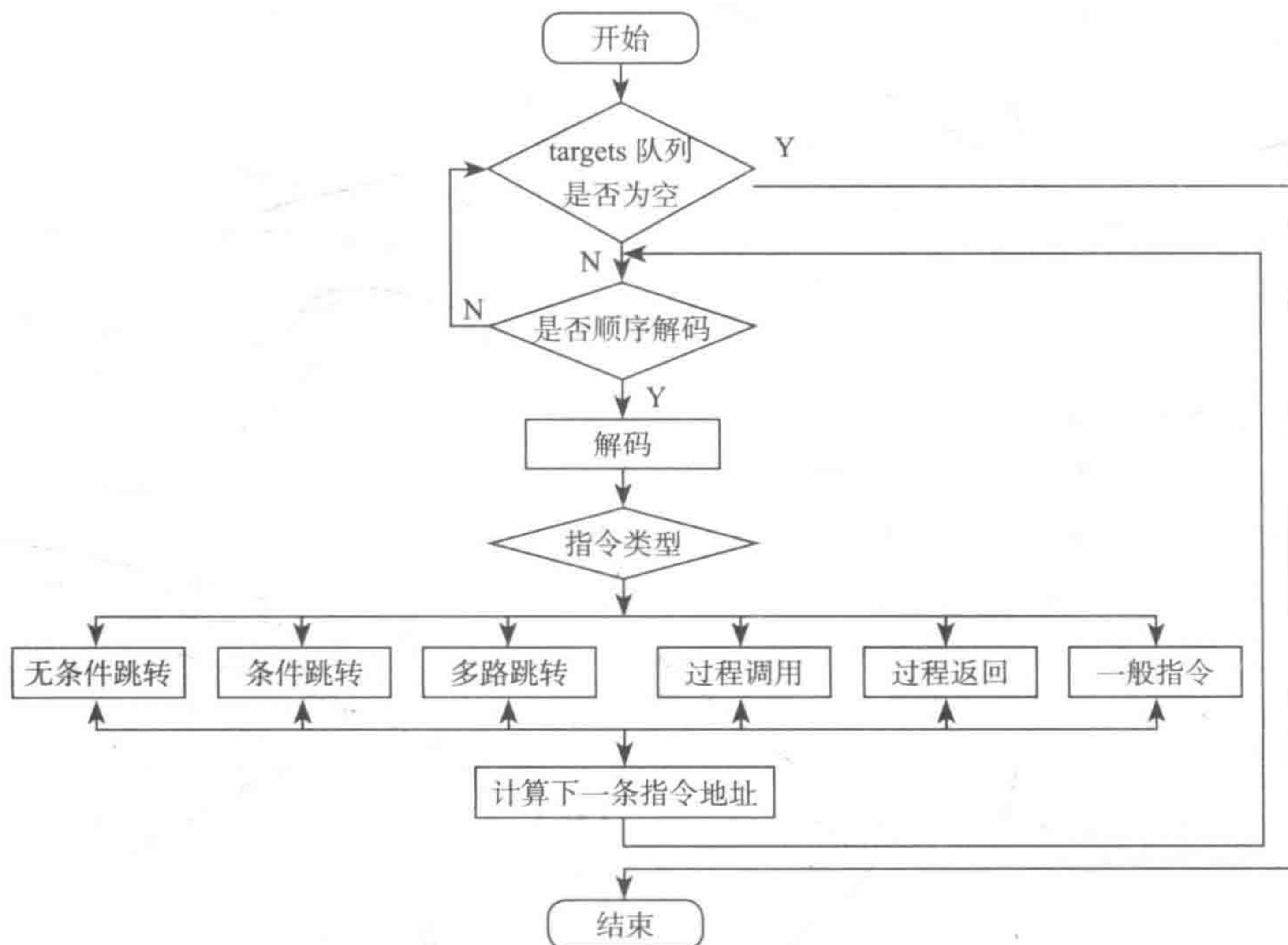


图 14-48 指令解码模块流程图

### (3) 语义映射

基于 SSL 对 IA32 指令集中指令的语义进行一系列描述，建立每条汇编指令到中间表示 RTL 的映射关系。

该模块输入为反汇编器对二进制代码进行反汇编后提交的汇编指令流，主要信息包括汇编指令的名字和参数、指令类型。输出为机器相关的低级中间表示序列 RTL，主要信息包括汇编指令的名字和参数、指令对应的一条或多条 RT 语句、每条赋值语句所操作的位数大小、是否有谓词执行、指令类型等。

语义映射是利用语义描述语言，针对特定的处理器体系结构，将处理器执行指令的过程及结果描述出来，根据描述完成从机器的汇编指令流到中间代码表示的映射，从而达到对机器指令进行语义映射的目的，并为最终转换为高级语言程序奠定基础。语义映射模块流程如图 14-49 所示。以本章系统为例，语义映射根据 IA32 体系结构的特性，使用语义描述语言 SSL 对 IA32 每条指令的功能进行一系列描述，每条指令的功能由一条或多条描述语句来表示。在二进制翻译过程中，装入 SSL 描述文件，通过对 SSL 文件进行分析生成 RTL 模板库，然后根据对二进制文件反汇编后提交的指令名和操作数进行匹配，查找 RTL 模板库获得指令的 RTL 表示，完成汇编指令流向 RTL 序列的转换。



图 14-49 语义映射模块流程图



## 2. 行为检测模块

为了对二进制文件中的行为进行分析、检测，因此将模糊理论应用到可疑行为识别和恶意代码判定中，设计了基于模糊推理的恶意行为检测模块（以下简称“行为检测模块”）。行为检测模块具体由三个模块组成：恶意行为识别模块、规则建立模块、恶意性判定模块，各个模块的关系如图 14-50 所示。

**恶意行为识别模块：**通过提取病毒的函数调用序列来描述可疑行为，对实现功能相似的可疑行为进行归类，并在行为类定义的基础上引入基于函数调用序列匹配的加权相似度算法，设计并实现了基于加权相似度计算的可疑行为模糊识别。

**规则建立模块：**将贝叶斯算法应用于判定系统规则的训练过程，设计并实现了判定规则库的建立，为第三步的模糊推理提供推理的依据。

**恶意性判定模块：**在模糊推理判定系统设计的基础上，采用 Zadeh 合成规则、最大最小法等模糊推理规则，对可疑代码的恶意性进行判定。

下面对行为检测模块所涉及的主要子模块进行详细说明。

### （1）恶意行为识别模块

恶意行为识别模块使用反编译过程中得到的文件结构、指令序列、函数调用、控制流和数据流等信息识别二进制文件中的恶意行为。

在处理过程中，首先，在反编译处理的基础上，提取与可疑行为库进行序列相似度匹配后得到的函数调用；其次，在序列相似度匹配的基础上，实现可疑行为子类的识别；同时，在可疑行为子类识别的基础上，对行为进行归类并在可疑行为类的层面上实现行为类的识别，完成行为检测模块中代码恶意性的判定工作。

#### 1) 可疑行为归类定义。

根据之前对于病毒编写原理和代码特征的分析，将病毒必须或经常需要调用的 API 函数序列提取出来作为可疑行为定义的依据，在定义方面存在两个问题：

① 由于 API 函数本身的类似，存在两个或多个函数能够完成相似的功能。

② 由于行为结果的类似，存在多个行为能够实现相似的功能。

针对问题①主要采用以调用关系图和调用序列相结合的形式来定义可疑行为子类，针对问题②主要采用相似功能行为归类的方式来定义可疑行为类。图 14-51 选取了一个基于可疑行为子类“文件写”的 API 函数调用关系图。而在可疑行为类

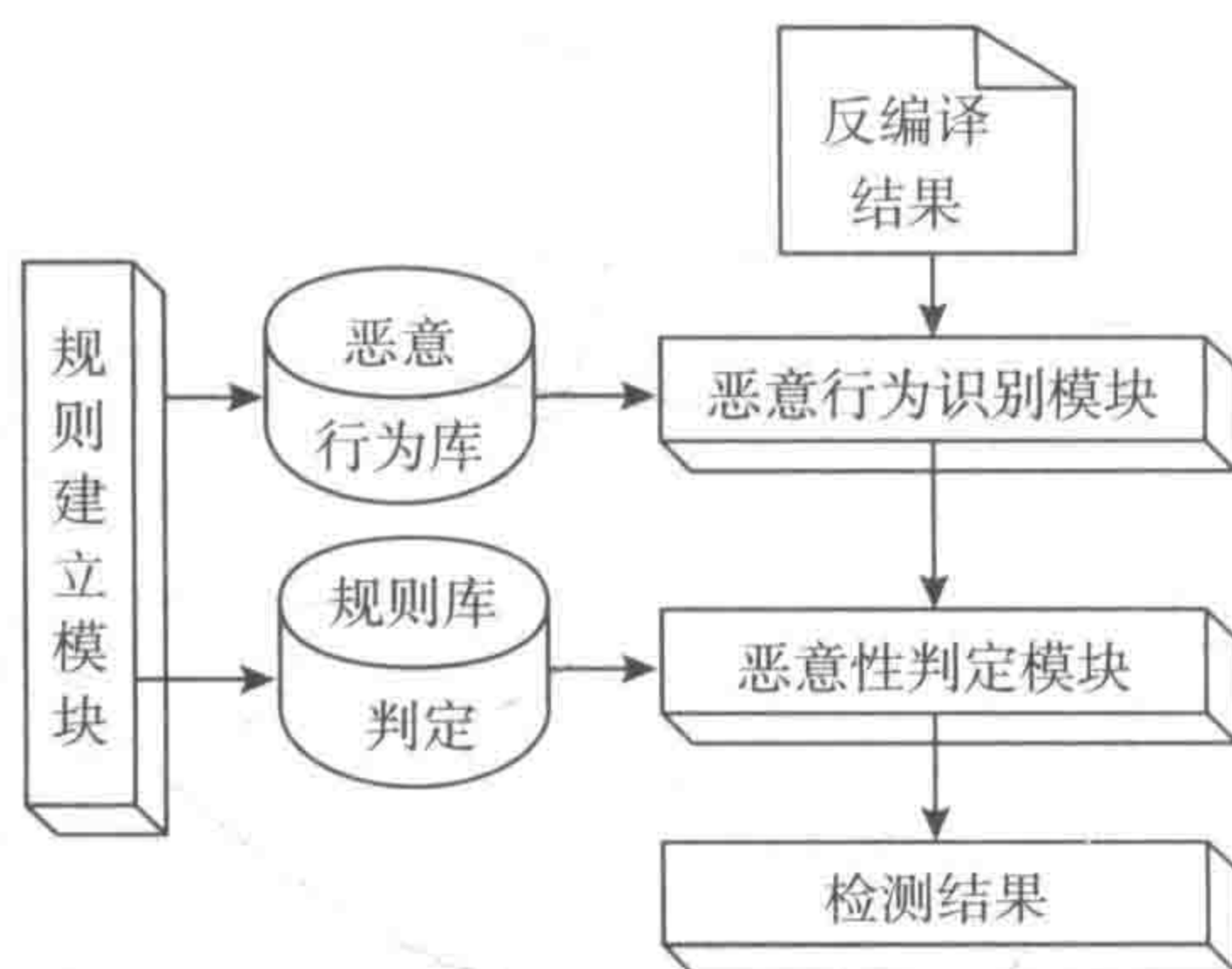


图 14-50 行为检测模块框架结构图

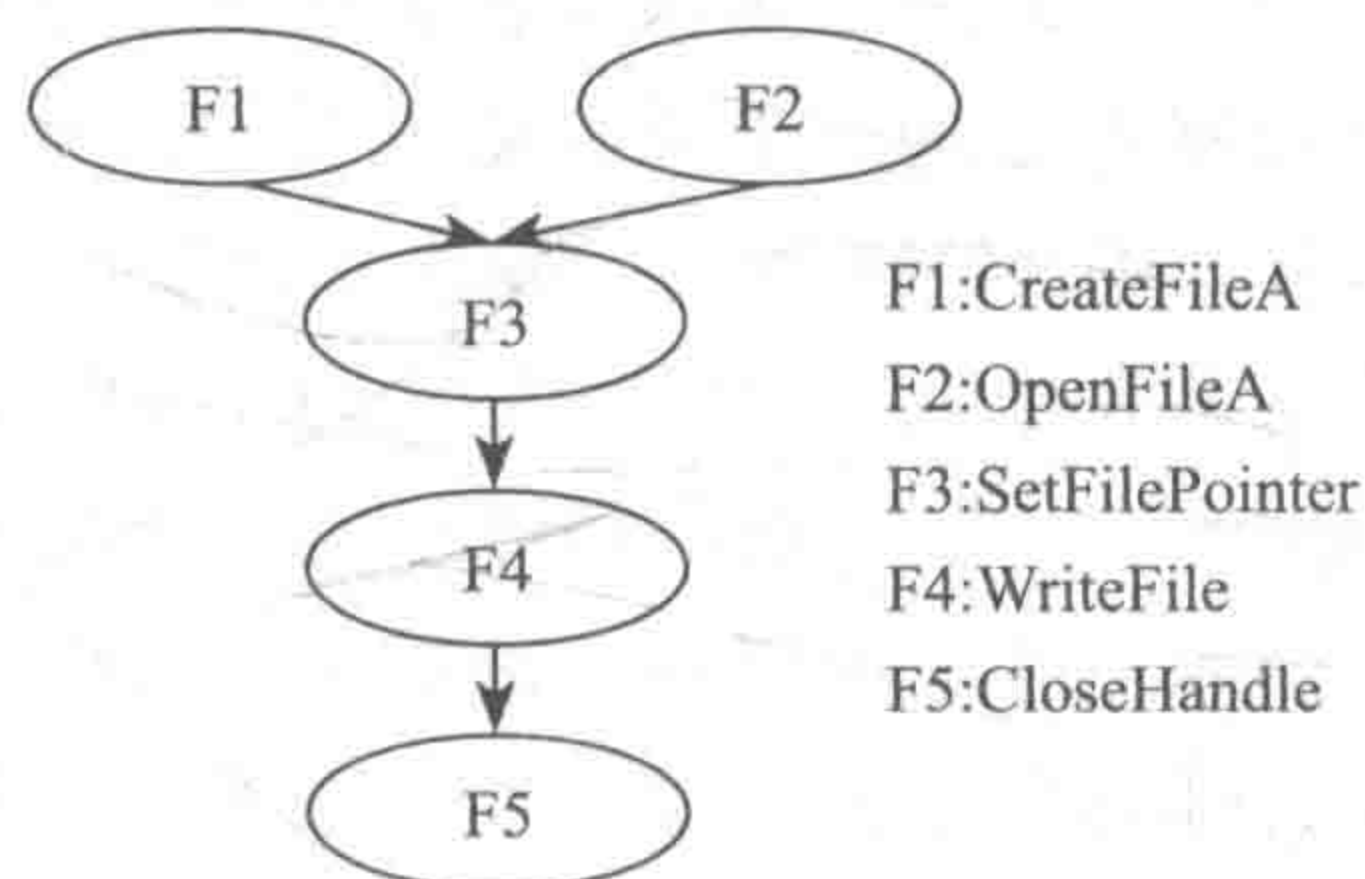


图 14-51 基于“文件写”的 API 函数调用关系图



的定义方面,表 14-3 举例给出了部分基于函数调用的可疑行为归类的一般描述。

## 2) 加权相似度。

相似度是指两个对象之间的相似程度,其取值范围一般为  $0 \sim 1$ 。本章选取相似度算法中最常用的 LD 算法作为计算函数调用序列相似度的基本算法。LD 算法是用  $m \times n$  矩阵来存储距离值。其算法的一般过程为:

① 若 str1 或 str2 的长度为 0,则返回另一个字符串的长度。

② 初始化  $(n+1) \times (m+1)$  的矩阵  $d$ ,并让第一行和列的值从 0 开始增长。

③ 扫描两个字符串 ( $n \times m$  级的),如果  $\text{str1}[i]=\text{str2}[j]$ ,用 temp 记录它为 0,否则 temp 记为 1。

④ 将矩阵元素  $d[i][j]$  赋值为  $d[i-1][j]+1$ 、 $d[i][j-1]+1$ 、 $d[i-1][j-1]+temp$  三者中的最小值。

⑤ 扫描完后,返回矩阵的最后一个值  $d[n][m]$  即为两个字符串的距离。

⑥ 相似度 =  $1 - (\text{距离} / \text{字符串长度最大值})$ 。

下面使用图 14-51 基于“文件写”的 API 函数调用图举例说明算法的步骤。该图其实是由两个函数调用序列 (F1,F3,F4,F5) 和 (F2,F3,F4,F5) 构成。若检测出未知代码甲具备函数调用序列 (F1,F4,F5),未知代码乙具备函数调用序列 (F2,F3,F5)。利用前面介绍的算法,可计算出这两个代码对于该可疑行为的相似度  $\mu$  都为 0.75。代码甲的相似度计算过程如图 14-52 所示。

		F1	F3	F4	F5
	0	1	2	3	4
F1	1				
F4	2				
F5	3				

→

		F1	F3	F4	F5
	0	1	2	3	4
F1	1	0	1	2	3
F4	2	1	1	1	2
F5	3	2	2	1	1

图 14-52 计算序列相似度示意图

但考虑每个函数调用在可疑行为所在的比重是不一样的,存在某个或多个关键函数调用来完成该可疑行为大部分或核心的功能,所以在可疑行为序列的相似度计算中引入加权的概念,即对所定义的某一可疑行为序列 (序列长度为  $n$ ) 中的每个函数调用  $F_i$  ( $i \leq n$ ),按照在行为中的重要性设定相应的初始权值  $\partial_{(F_i)}$ ,并作以下规定:

- $\sum_{i=1}^n \partial_{(F_i)} = 1$ , 即单一行为的所有函数调用的权值相加为 1。

表 14-3 部分可疑行为归类的一般描述

可疑行为类	可疑行为子类
一、可疑文件搜索	01, 类型文件搜索
	02, 目录文件搜索
	03, 资源文件搜索
二、异常文件操作	04, 文件写
	05, 创建文件映射
	06, 修改文件属性
三、内存进程操作	07, 虚拟内存分配
	08, 全局内存分配
	09, 创建进程执行
	.....



- 当目标代码调用序列中出现  $F_i$  函数时,  $\partial'_{(Fi)} = \partial_{(Fi)}$ 。
- 当目标代码调用序列中未出现  $F_i$  函数时,  $\partial'_{(Fi)} = 0$ 。
- 利用点乘运算, 即可计算调整之后的加权相似度  $\mu^* = \mu \cdot \sum_{i=1}^n \partial'_{(Fi)}$ 。

继续图 14-52 的举例进行说明, 在该可疑行为中, 核心函数为完成写操作功能的函数 F4, 即 WriteFile (设其权值为 0.7, 其他三个函数均为 0.1)。则函数调用序列为 (F1,F4,F5) 的未知代码甲的加权相似度为:

$$\mu^* = \mu \cdot \sum_{i=1}^n \partial'_{(Fi)} = 0.75 \times (0.1+0.7+0.1) = 0.675$$

而函数调用序列为 (F2,F3,F5) 的代码乙的加权相似度  $\mu^* = 0.225$ 。这样, 在行为识别中显示出了在相似度上是否包含核心函数的区别。

### 3) 行为模糊识别。

在对可疑行为类定义与加权相似度引入的基础上, 给出基于加权相似度计算的行为模糊识别的设计框架, 以及其在整个原型系统中的位置。基于函数调用的可疑行为类模糊识别的基本框架设计图如图 14-53a 所示, 图 14-53b 行为类定义部分的举例说明。

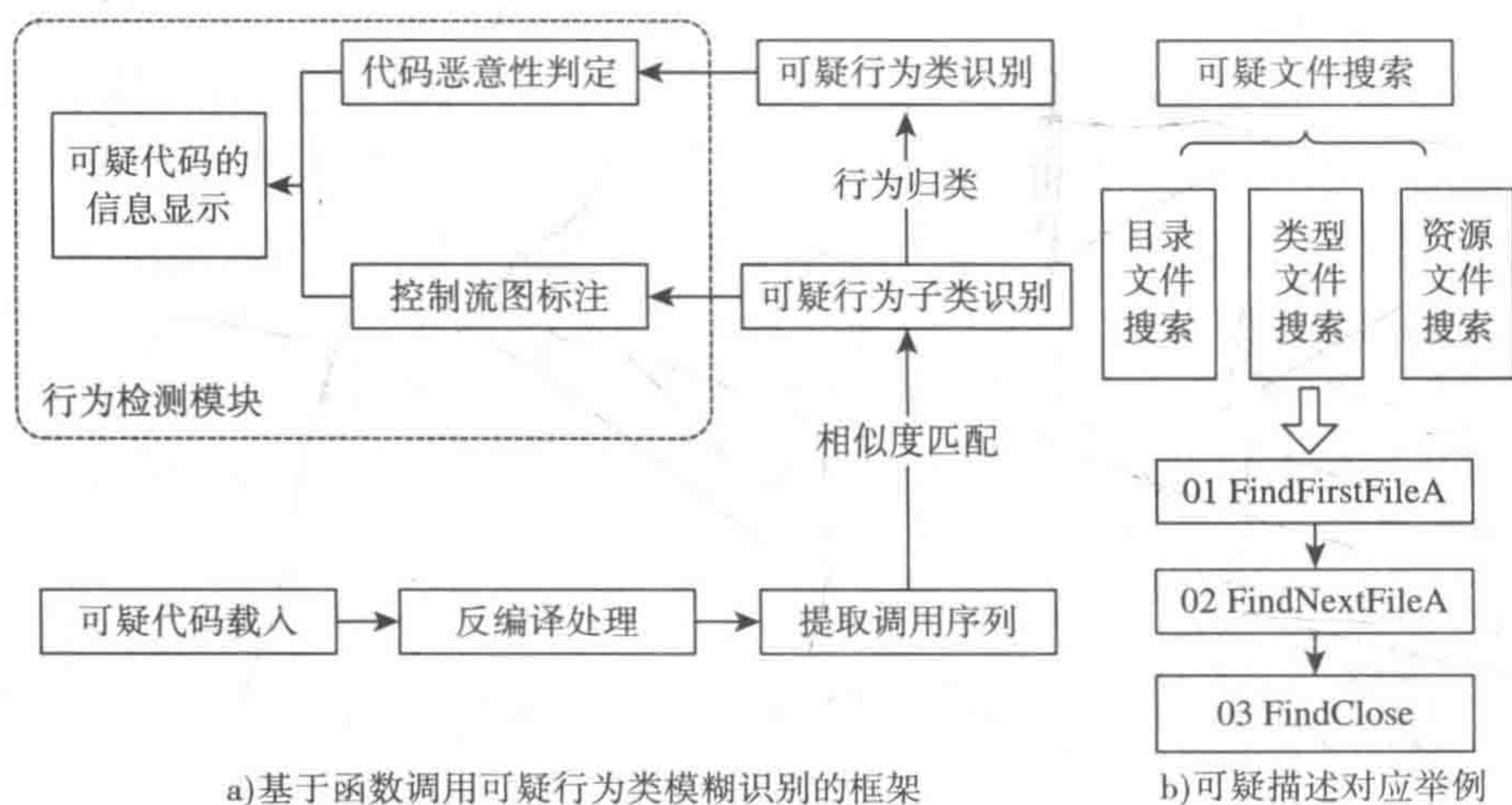


图 14-53 基于函数调用可疑行为类模糊识别的框架图

当可疑代码载入系统后, 首先, 在反编译处理的基础上, 提取与可疑行为库序列相似度匹配后的函数调用; 其次, 在序列相似度匹配的基础上实现可疑行为子类的识别, 用以完成标注模块中控制流图的标注工作; 同时, 在可疑行为子类识别的基础上, 对行为进行归类并在可疑行为类的层面上实现行为类的识别, 用以完成检测模块中代码恶意性的判定工作; 最后, 将之前完成的两部分工作汇总, 在标注模块中显示出加载代码的可疑行为识别和恶意性判定等信息, 便于程序分析人员和病毒检测人员对于目标代码的整体认识和进一步分析。



#### 4) 行为识别实现。

系统在识别过程中共定义 7 种可疑行为类, 包含 22 种可疑行为子类。其中包括基于函数调用匹配的四类, 基于指令序列和控制流图分析的一类, 基于文件结构异常识别的两类。下面以病毒代码 Win32.Apathy.5378 (以下简称“ Apathy”) 为例说明行为识别的整个过程。当 Apathy 载入之后, 经过脱壳、反编译等处理, 进入行为识别模块。

在行为识别模块中, 首先完成行为子类的识别。按照识别的先后顺序, 行为子类的识别分为三个部分的识别:

- ① 基于文件结构异常的可疑特征识别。
- ② 基于指令和控制流分析的函数调用方式行为识别。
- ③ 基于加权相似度识别的 API 函数行为模糊识别。

其中, 第三部分是本章研究的重点, 也是行为识别的核心部分。经过行为子类的识别, 病毒 Apathy 在第一部分被识别出文件头大小设置错误的可疑特征, 在第二部分未被识别出异常的调用方式, 在第三部分的前三类识别情况如表 14-4 所示, 其中序号表示第几类行为的第几个子类行为, 如 1.3 表示第一类行为的第三个子类行为。

在完成行为子类的识别之后, 取每一类中各子类行为识别的最大值作为该类行为模糊识别的隶属度。如表 14-4 所示, 第一类行为 (即“可疑文件搜索”), 三个子类中最大的模糊识别度为第一个子类的识别隶属度, 其为 1, 则第一类行为识别的分量为 1, 之后第二类、第三类依次为 0.675 和 0.225。结合三个部分的七类行为, 最终完成病毒 Apathy 的行为模糊识别, 生成行为识别隶属度构成的序列  $S_{Apa} = (1, 0.675, 0.225, 0, 0, 1, 0)$ 。

表 14-4 行为子类的模糊识别

序号	行为子类名称	模糊识别
1.1	类型文件搜索	1
1.2	目录文件搜索	0.675
1.3	资源文件搜索	0
2.1	文件写	0.675
2.2	创建文件映射	0.675
2.3	修改文件属性	0.225
3.1	虚拟内存分配	0
3.2	全局内存分配	0
3.3	创建进程执行	0.225
.....		

#### (2) 规则建立模块

将样本空间划分为训练集和测试集, 训练集通过恶意行为识别模块生成每个训练程序的由多个行为识别隶属度所组成的序列, 作为判定规则的前件; 再利用贝叶斯算法, 根据该序列在训练集的良好代码集和病毒代码集中各自出现的频率, 计算出行为识别序列的隶属度, 作为判定规则的后件。

判定规则的确定是在恶意行为识别模块的基础上, 主要完成模糊推理中规则库的建立。

该模块的输入为二进制文件中的可疑行为, 输出为生成规则库以及恶意行为库。

一个行为识别序列生成的前件和后件共同组成模糊推理的一条判定规则, 整个训练集的识别序列生成的规则集合就构成判定规则库。这个判定规则库将作为测试集进行测试的判定依据, 生成测试结果, 以完成后期的可疑代码恶意性判定工作。规则库建立流程如图 14-54 所示。



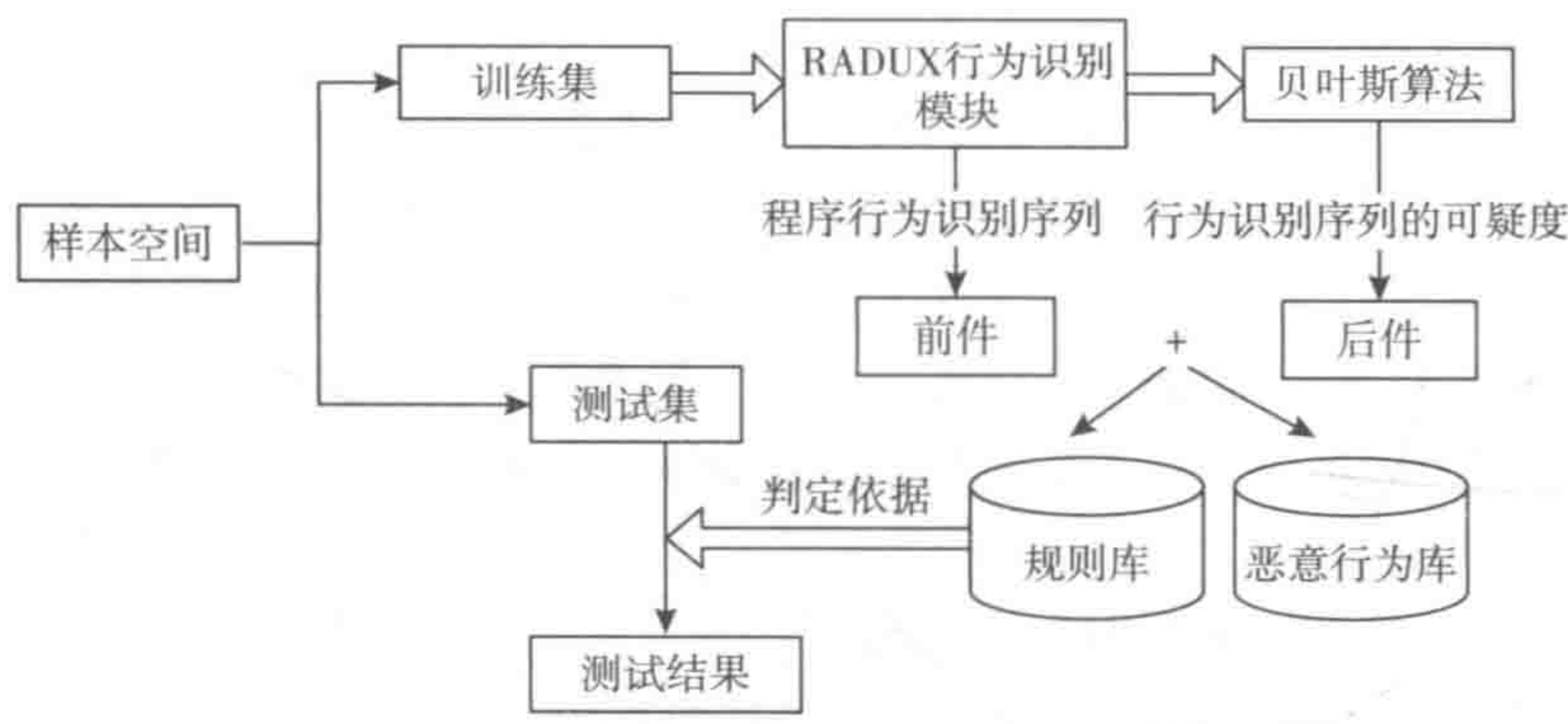


图 14-54 基于贝叶斯算法的规则库建立流程

1) 划分样本空间。

选用的是 Windows XP 首次安装后，机器中 Windows 目录下全部 PE 文件以及正版应用程序中随机采集的各种文件 3583 个，并在网络上经多种途径收集到病毒代码 3572 个，共 7155 个代码构成的样本空间，将样本空间随机按照 9:1 的比例划分为训练样本集  $S_{training}$  和测试样本集  $S_{test}$  两个集合（训练样本集和测试样本集为样本空间中两个不相交的子集  $S_{test} \cap S_{training} = \emptyset$ ），并根据收集来源，将训练样本集划分为病毒代码集  $S_{virus}$  和良性代码集  $S_{benign}$  ( $S_{virus} \cap S_{benign} = S_{training}$ ,  $S_{virus} \cap S_{benign} = \emptyset$ )，具体分类集合见表 14-5 所示。

2) 生成行为识别序列。

将训练样本集  $S_{training}$  载入原型系统，在反编译模块的基础上，通过行为识别模块，依据之前对于可疑行为的归类定义和描述，由加权相似度计算生成目标代码的行为识别序列  $S (Sequence) = (\mu_1, \mu_2, \dots, \mu_m) \in \xi (U)$ ，其中：

- $0 \leq \mu_i \leq 1, 1 \leq i \leq m$ ,  $\mu_i$  表示一个行为类  $i$  的识别隶属度。
- 依据可疑行为定义，共定义七种可疑行为类，即  $m = 7$ 。
- 论域  $U$  为行为识别情况。

通过行为识别，将目标可疑代码转化为一个含有七个分量的行为识别序列  $S (\mu_1, \mu_2, \dots, \mu_m)$ ，其中每一个行为识别序列中的分量为前面经过相似度计算的行为类隶属度。

3) 计算行为序列隶属度。

基于贝叶斯理论的可疑代码判定是一个二值分类问题，即病毒代码与良性代码两类。定义  $C$  为分类集，即 {病毒，良性}，令  $C$  表示病毒代码集， $\bar{C}$  表示良性代码集。基于朴素贝叶斯的分类规则是通过概率统计方法得到的，为病毒代码集和良性代码集都建立相对应的散列表，分别为 hashtable\_virus 和 hashtable\_benign，以存储行为识别序列到序列频率的映射关系，其中散列表中某一行为序列  $S_i (S_i \in S)$  出现的概率为：

$$P (S_i / C) = (S_i \text{ 在 hashtable\_virus 中的序列频率值} / \text{对应散列表的长度})$$

表 14-5 样本空间集合

	样本空间	训练集 $S_{training}$	测试集 $S_{test}$
良性代码	3583	3225	358
病毒代码	3572	3215	357
合计	7155	6440	715



$P(S_i/\bar{C}) = (S_i \text{ 在 hashtable\_benign 中的频率值} / \text{对应散列表的长度})$

对于样本空间中的每一个样本代码，均可以生成其行为识别序列  $S_i$ ，并由于病毒代码集和良性代码集是相互独立的，所以用行为序列的恶意度  $P(C/S_i)$  来表示可疑代码表现行为序列  $S_i$  时，该代码为病毒代码的概率，正常度  $P(\bar{C}/S_i)$  来表示可疑代码表现可疑行为序列  $S_i$  时，该代码为良性代码的概率。

由于  $C$  和  $\bar{C}$  互斥构成一个完备事件，则条件概率  $P(C/S_i)$  为：

$$P(C/S_i) = \frac{P(C) \cdot P(S_i/C)}{P(C) \cdot P(S_i/C) + P(\bar{C}) \cdot P(S_i/\bar{C})} \quad (14-1)$$

由于贝叶斯理论的二值分类性， $P(C/S_i) + P(\bar{C}/S_i) = 1$ 。通过对上式的分析可知：当选定的可疑行为在病毒代码集中出现的概率大于在良性代码集中出现的概率时， $P(C/S_i)$  的值将大于 0.5，反之则小于 0.5。

#### 4) 建立判定规则库。

通过上一步将相同的行为识别序列合并作为规则的前件，而规则的后件以对应序列的恶意度  $P(C/S_i)$  为基础构建，为一个三元组： $(P(C/S_i), P(\tilde{C}/S_i), P(\bar{C}/S_i))$ ，其中，恶意度  $P(C/S_i)$ ，表示该序列对于病毒代码集的隶属度；正常度  $P(\bar{C}/S_i) = 1 - P(C/S_i)$ ，表示该序列对于良性代码集的隶属度；可疑度  $P(\tilde{C}/S_i) = 1 - |P(C/S_i) - P(\bar{C}/S_i)|$ ，表示具备序列  $S_i$  的代码对于可疑集合的隶属度，选取它来刻画该序列恶意度与正常度之间的差别：当经过贝叶斯算法生成的序列恶意度越高或越低，即  $|P(C/S_i) - P(\bar{C}/S_i)|$  的值越大，则序列隶属于病毒代码集或良性代码集的程度越高，具备该行为序列的代码恶意性就越容易判定，所以序列隶属于可疑集合的程度就越小，其可疑度就越小，反之则越大。

将前件与后件结合就确定了判定的一条规则  $R_i$ ，即建立规则映射为：

$$R_i: S_i \rightarrow (P(C/S_i), P(\tilde{C}/S_i), P(\bar{C}/S_i)) \quad (14-2)$$

将式 (14-2) 作为规则库中的一般表示形式。

经过训练测试建立的规则库为了方便读取和修改，在代码编写中以  $m \times n$  矩阵存储， $m$  表示规则库的规则个数， $n$  表示一条规则所具备的分量。其中  $m$  的取值由训练中行为识别所得的序列在完成相同序列合并之后的个数决定，系统经过样本空间的训练共得到 936 个不同序列，即  $m = 936$ ，规则库共由 936 条； $n$  取值等于式 (14-2) 中所给出规则的一般形式中前件和后件总共具备的分量数，所以：

$$n = 7 (\text{七个行为类 } S_i) + 3 (\text{行为序列恶意度 } P(C/S_i), \text{可疑度 } P(\tilde{C}/S_i), \text{正常度 } P(\bar{C}/S_i)) = 10$$

下面举例说明一条规则的确定。设训练集中所有代码均已载入系统完成训练，并生成行为识别序列。继续使用前面行为识别实现中的例子，当病毒 Apathy 生成行为识别序列为  $S_{Apa} = (1, 0.675, 0.225, 0, 0, 1, 0)$  时，经统计该序列在整个测试集中出现次数为：在病毒代码集中出现 167 次，即  $P(S_i/C) = 167/3215$ ；在良性代码集中出现 45 次，即  $P(S_i/\bar{C}) =$



45/3225, 经过贝叶斯算法可得到该序列的恶意度  $P(C/S_i)=0.79$ , 正常度  $P(\bar{C}/S_i)=1-0.79=0.21$ , 隶属于可疑集合的可疑度  $P(\tilde{C}/S_i)=1-|0.79-0.21|=0.42$ 。所以该代码生成的一条规则可表示为  $R_{Apa}=(1, 0.675, 0.225, 0, 0, 1, 0, 0.79, 0.21, 0.42)$ 。

### (3) 恶意性判定模块

恶意性判定模块使用模糊推理技术, 对二进制文件的恶意性进行判断。

判定系统的设计主要依据模糊推理系统, 模糊推理系统是建立在模糊集合、模糊规则和模糊推理等概念基础上的一种推理框架。可疑代码经过模糊推理系统, 根据规则库判定出可疑代码的恶意性, 分析人员可以根据判定的结果对可疑目标代码进行相应的处理。

该模块的输入为二进制文件中含有的可疑行为、规则库、恶意行为库。输出为二进制文件的恶意程度。

结合之前各项工作的研究, 实现基于模糊推理的代码恶意性判定系统的建立。推理系统设计如图 14-55 所示。

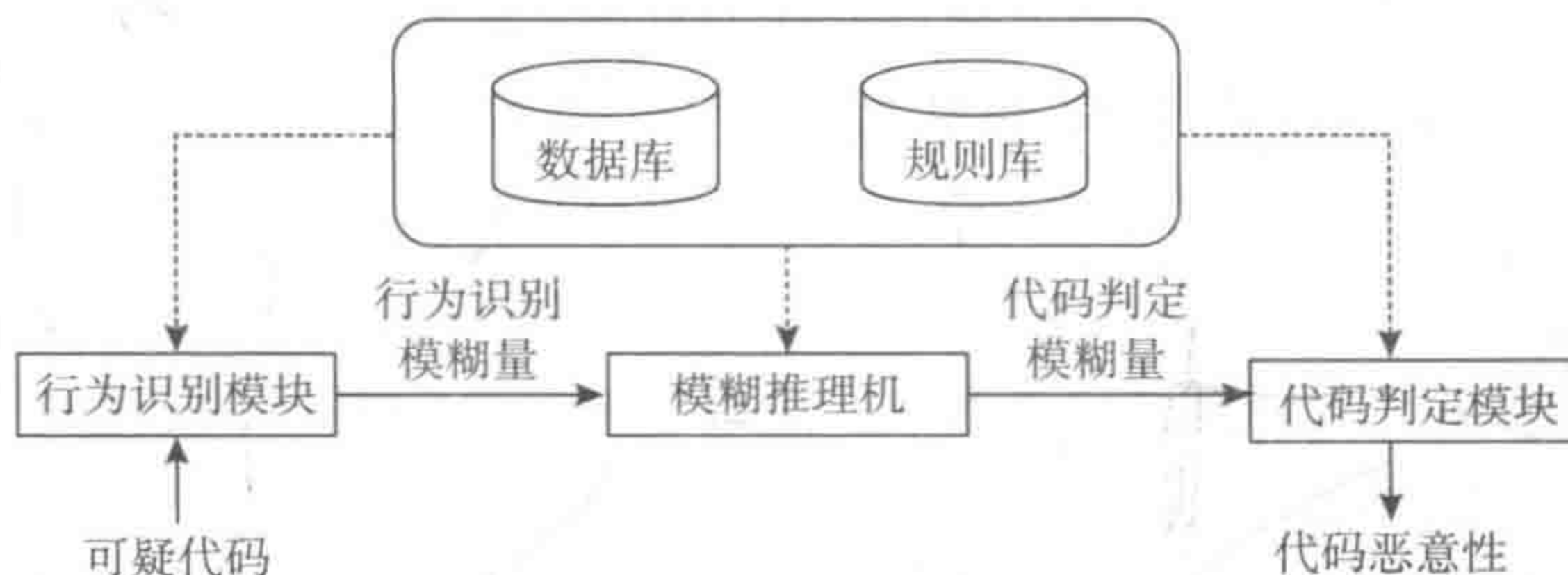


图 14-55 基于可疑行为识别的模糊推理系统框架

在已经确定推理规则和判定规则的情况下, 下面给出模糊推理系统中模糊推理机和代码判定模块两个部分的设计流程。

#### 1) 模糊推理。

如图 14-55 所示, 当可疑代码  $x$  载入判定系统, 经过行为识别模块生成由 7 个行为识别分量  $\mu_i'$  组成的行为识别序列  $S_x(\mu_1', \mu_2', \dots, \mu_7')$ , 其中  $0 \leq \mu_i' \leq 1 (1 \leq i \leq 7)$ 。当进入模糊推理机时, 系统主要完成三步工作:

① 将  $S_x$  与规则库中的所有规则的前件  $S_i (0 \leq i \leq n)$  进行比较, 采用最小最大法计算出与每一条规则前件的贴近期  $\psi(S_i, S_x)$ 。

② 通过比较取得最大贴近期, 确定  $S_x$  与哪一条规则  $R_j$  的前件  $S_j (0 \leq j \leq n)$  最为贴近期。

③ 如式 (14-2) 给出的规则一般表示, 将最为贴近期的规则  $R_j$  作为条件进行模糊推理, 得出一个由三个代码判定隶属度构成的模糊量  $\theta=(\theta_M, \theta_S, \theta_N)$ 。下面给出判定可疑代码恶意性的主要方式和判定过程。

#### 2) 恶意性判定。

基于传统分类理论的未知程序恶意性判定, 如概率统计、数据挖掘等, 一般通过训练



建立判定基础后,计算出测试集中每个程序的恶意度,之后通过设置多个不同的阈值,考察每一个阈值对应的误报率和漏报率,最终比较确定出其中精度更高的阈值来作为未知程序判定的依据。而无论载入的未知程序所得恶意度 $\theta$ 有多大,它对于传统分类的{正常,病毒}两个集合的隶属度,非1即0,如图14-56a所示。

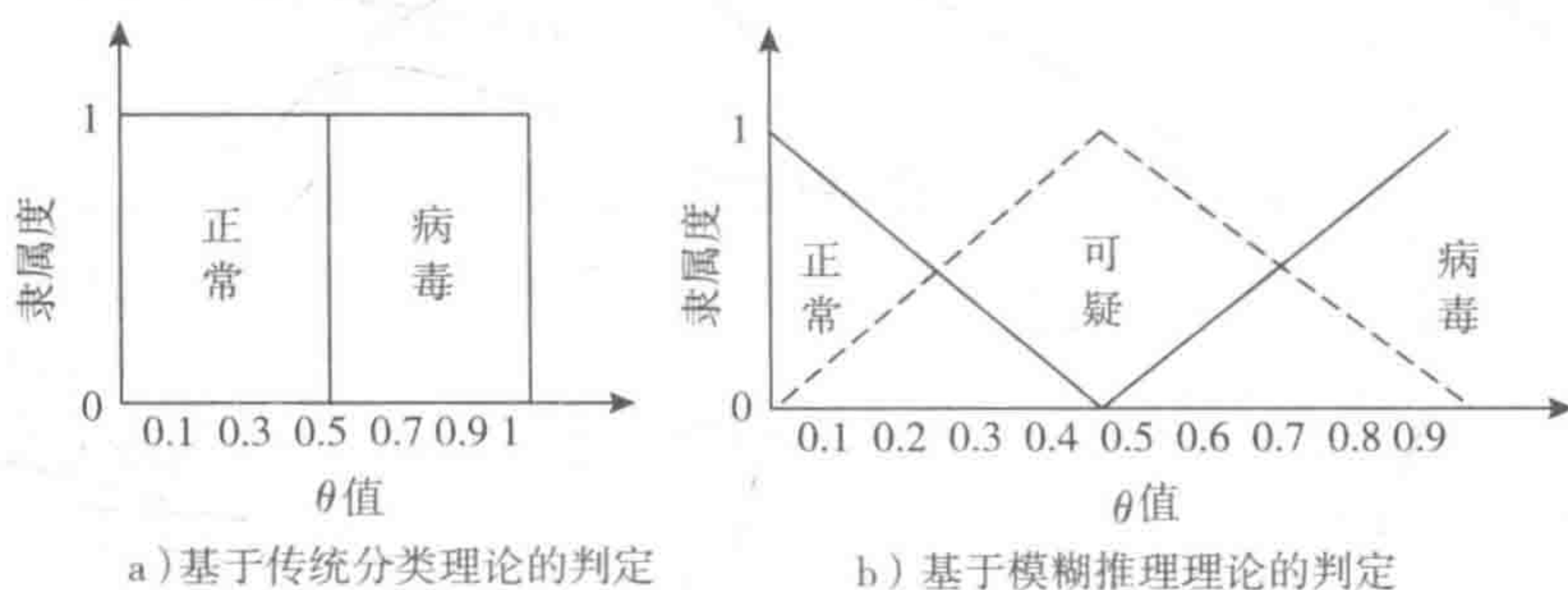


图 14-56 传统分类与模糊理论在代码判定中比较

基于模糊推理理论的可疑代码恶意度判定在判定方式上与传统分类理论判定有着本质的区别。在判定中,主要确定两个论域的模糊表示:论域 $U$ 为行为识别情况,论域 $V$ 为模糊推理系统所得到的代码判定。将论域 $V$ 分为三个模糊子集,即{正常代码集(N—Normal),可疑代码集(S—Suspicious),恶意代码集(M—Malware)},如图14-56b所示。三个模糊子集都选取最常用的三角形隶属函数来表示,其中模糊子集 $N$ 为“正常代码集”,则隶属函数 $\mu_N(V)$ 为:

$$\mu_N(V) = \begin{cases} 1, v < a_1 \\ \frac{a_2 - v}{a_2 - a_1}, a_1 \leq v \leq a_2 \\ 0, v > a_2 \end{cases} \quad (14-3)$$

其中 $a_1 = 0.1$ ,  $a_2 = 0.45$ ,其他模糊子集与之类似。本章所要实现的工作就是实现从论域 $U$ 到论域 $V$ 的模糊推理,以完成对可疑代码的恶意性判定。

模糊推理机生成代码判定的模糊量 $\theta = (\theta_M, \theta_S, \theta_N)$ ,其中的分量分别代表三个模糊子集的隶属度。下面要实现的是代码判定模块的反模糊化工作。反模糊化的方法有很多,其中较常用、有效的为最大隶属度法。在模糊推理输出的结果中,取其隶属度最大的元素作为精确值并执行控制的方法称为最大隶属度法。最大隶属度法在处理过程中有简单、方便、容易实现等优点。

依据最大隶属度法,取模糊量 $\theta$ 中隶属度最大的元素作为代码恶意性进行输出,即 $\theta^* = \text{Max}\{\theta_M, \theta_S, \theta_N\}$ 。

若 $\theta^* = \theta_M$ ,则判定该代码具备明显的恶意性,为恶意代码,系统将建议程序分析人员或用户消除该程序。

若 $\theta^* = \theta_S$ ,则判定该代码具备一定的恶意性,为可疑代码,系统将建议程序分析人员



或用户清除程序中的恶意功能, 或将该程序隔离。

若  $\theta^* = \theta_N$ , 则判定该代码不具备恶意性, 系统将提示为正常代码。

在完成规则库建立之后, 一可疑代码  $x$  载入判定系统中的识别序列为:  $S_x = (1, 0.6, 1, 0.225, 1, 1, 0)$ , 它将经过模糊推理得到代码判定的模糊量, 然后经过恶意性判定得到代码恶意性, 最终完成判定工作。下面就这两部分在系统中的具体实现进行阐述。

### 3) 模糊推理实现。

接下来首先要计算出序列  $S_x$  与规则库中所有规则  $R_i$  前件  $S_i$  的贴近度, 然后找到与规则库构建的矩阵中前七项  $S_i$  具有最大贴近度的一行  $R_i$ , 最后经过模糊推理生成该代码判定的模糊量。

由于规则库中的规则个数比较多, 仅举例说明对较为贴近的 8 条规则进行最大贴近度计算, 规则库为  $8 \times 10$  的矩阵, 即:

$$R = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0.81 & 0.38 & 0.19 \\ 1 & 1 & 1 & 0.225 & 1 & 1 & 0 & 0.94 & 0.12 & 0.06 \\ 1 & 0.675 & 1 & 0 & 1 & 1 & 0 & 0.67 & 0.66 & 0.33 \\ 1 & 0.675 & 1 & 0.225 & 1 & 1 & 0 & 0.84 & 0.32 & 0.16 \\ 0.675 & 1 & 1 & 0 & 1 & 1 & 0 & 0.70 & 0.60 & 0.30 \\ 0.675 & 1 & 1 & 0.225 & 1 & 1 & 0 & 0.56 & 0.96 & 0.44 \\ 0.675 & 0.675 & 1 & 0 & 1 & 1 & 0 & 0.32 & 0.64 & 0.68 \\ 0.675 & 0.675 & 1 & 0.225 & 1 & 1 & 0 & 0.48 & 0.96 & 0.52 \end{pmatrix}$$

其中每一行为一个  $R_i$ , 每个  $R_i$  的前七个分量为一个  $S_i$ ,  $1 \leq i \leq 7$ 。

首先, 计算出  $S_x$  与每一个  $S_i$  的贴近度:

$$\psi(S_1, S_x) = 0.880, \psi(S_2, S_x) = 0.923$$

$$\psi(S_3, S_x) = 0.939, \psi(S_4, S_x) = 0.985$$

$$\psi(S_5, S_x) = 0.818, \psi(S_6, S_x) = 0.861$$

$$\psi(S_7, S_x) = 0.872, \psi(S_8, S_x) = 0.918$$

可得  $\bigvee_{i=1}^8 (\psi(S_i, S_x)) = 0.985 = \psi(S_4, S_x)$ , 此处“ $\vee$ ”表示取最大值, 所以得出可疑代码  $x$  与规则库中的第四条规则的前件最为贴近。

最后, 将按照该条规则和  $x$  所识别的序列作为条件, 进行模糊推理以得出该可疑代码的判定模糊量, 即要实现:

$$\begin{array}{c} R_4: S_4 \rightarrow (P(C/S_4), P(\tilde{C}/S_4), P(\overline{C}/S_4)) \\ \hline S_x \\ \hline (P(C/S_x), P(\tilde{C}/S_x), P(\overline{C}/S_x)) \end{array}$$

根据 Zadeh 合成规则, 首先计算模糊蕴涵算子  $R_a$ 。



规则为“(1, 0.675, 1, 0.225, 1, 1, 0) → (0.81, 0.32, 0.16)”, 则可利用公式(14-3):

$$\mu_R(1,1) = 1 \wedge (1 - \mu_A(1) + \mu_B(1)) = 1 \wedge (1 - 1 + 0.84) = 0.84$$

$$\mu_R(1,2) = 1 \wedge (1 - \mu_A(1) + \mu_B(2)) = 1 \wedge (1 - 1 + 0.32) = 0.32$$

.....

$$\mu_R(3,3) = 1 \wedge (1 - \mu_A(3) + \mu_B(3)) = 1 \wedge (1 - 1 + 0.16) = 0.16$$

.....

$$\mu_R(7,3) = 1 \wedge (1 - \mu_A(7) + \mu_B(3)) = 1 \wedge (1 - 0 + 0.16) = 1$$

从而  $R_a = 0.84/(1, 1) + 0.32/(1, 2) + \dots + 0.16/(3, 3) + \dots + 1/(7, 3)$ , 如果只取隶属度, 且写成  $7 \times 3$  矩阵形式, 则:

$$R_a = \begin{pmatrix} 0.84 & 0.32 & 0.16 \\ 1 & 0.645 & 0.485 \\ 0.84 & 0.32 & 0.16 \\ 1 & 1 & 0.935 \\ 0.84 & 0.32 & 0.16 \\ 0.84 & 0.32 & 0.16 \\ 1 & 1 & 1 \end{pmatrix}$$

然后, 采用“ $\vee - \wedge$ ”合成运算计算出可疑代码  $x$  的恶意性判定的模糊量  $\theta$ :

$$\begin{aligned} \theta = S_x \circ R_a &= (1, 0.6, 1, 0.225, 1, 1, 0) \circ \begin{pmatrix} 0.84 & 0.32 & 0.16 \\ 1 & 0.645 & 0.485 \\ 0.84 & 0.32 & 0.16 \\ 1 & 1 & 0.935 \\ 0.84 & 0.32 & 0.16 \\ 0.84 & 0.32 & 0.16 \\ 1 & 1 & 1 \end{pmatrix} \\ &= (0.84, 0.6, 0.485) \end{aligned}$$

通过结果可以看到经过模糊推理后, 得到的分量  $P(C/S_x) = 0.84$  和  $P(\bar{C}/S_x) = 0.485$  相加已经不再等于 1 了, 这就是模糊推理的特点, 即不再符合传统概率理论的基本性质。

#### 4) 恶意性判定实现。

经过模糊推理机, 系统生成了可疑代码  $x$  的判定模糊量  $\theta = (\theta_M, \theta_S, \theta_N) = (0.84, 0.6, 0.485)$ , 根据关于恶意性判定的设置, 依据最大隶属度法, 取  $\theta^* = \text{Max}\{0.84, 0.6, 0.485\} = 0.84 = \theta_M$ , 所以代码  $x$  相对隶属于恶意程序集合, 则判定程序具有明显的恶意性, 系统将其判定为恶意代码, 并建议程序分析人员或用户进行消除。这样, 系统就实现了一个可疑代码的判定工作。

### 3. 标注模块

根据反编译分析结果, 同时结合行为检测模块的恶意行为检测结果, 设计了双空间三层协同标注的标注方案。所谓双空间是指标注结果分别以图形化的方式和 XML 描述的方式出现, 二者所表现的恶意行为相同, 但各自的任务不同, 前者旨在恶意行为的直观表现,



适用于对个例作具体分析；而后者使用抽象的描述语言对恶意行为进行标记，适用于对分析和检测结果的总结和存储。所谓的三层协同是指在图形化的基础上，对反编译过程中的三层结果进行的可疑行为标注，三层采用不同的标注方式，但又通过图形化标注联系起来，实现标注的系统化，详细的流程如图 14-57 所示。

下面对标注模块所涉及的主要子模块进行详细说明。

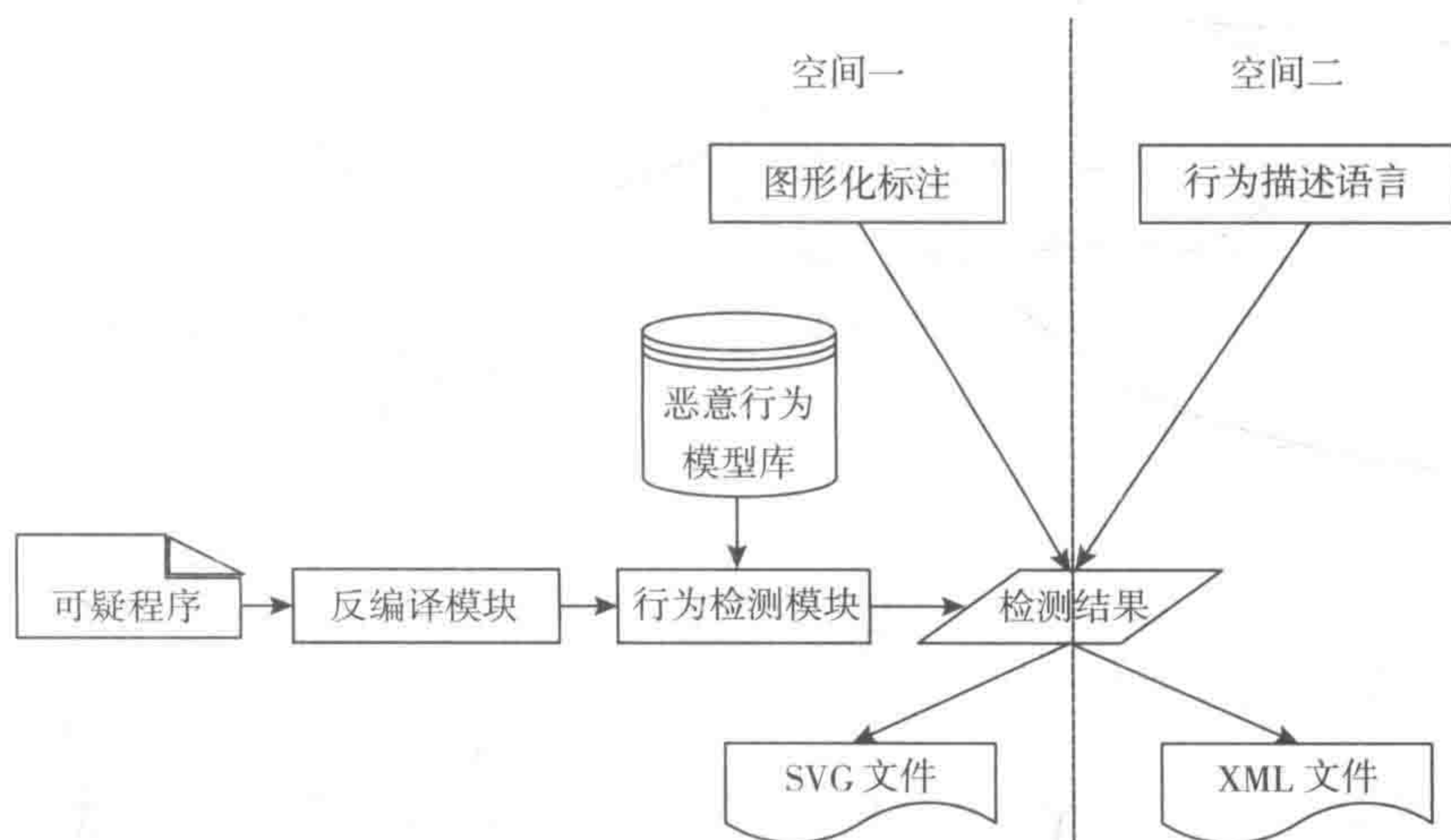


图 14-57 双空间多层次标注系统实现流程

### （1）恶意行为的多层次标注

可疑程序经过反编译模块的处理生成与反编译各层对应的程序分析结果，再根据恶意行为库中的行为特征，应用对应的检测机制，进行多个层次的图形化标注。

在对二进制文件的检测过程中，对二进制文件中含有的指令序列、函数调用、控制流程图等三个分析层面的程序分析结果进行系统化的标注。

该模块的输入为经过反编译得到的指令序列、函数调用、控制流程图等检测信息，以及文件的恶意程度。输出为多层次图形化标注结果。

处理过程如下：

1) 指令序列层。匹配恶意行为库中的模板序列，从汇编代码层面识别可疑的汇编指令行为序列，对于具有恶意行为的标志性序列，标注系统将在所生成的汇编代码上进行颜色与文字的说明。指令序列匹配实现简单，行为定义较直观，但对于混淆后的病毒程序效果不明显，这就需要其他层次标注结果的支持。

2) 控制流程层。根据重构的控制流程信息，生成采用 DOT 语言描述的控制流程图，展示程序控制结构，同时标注流程图匹配的结果。

3) 函数调用层。根据反汇编恢复出的程序关键结构——函数，与行为库中定义的恶意函数序列进行匹配。对于匹配成功的函数，标注系统将在所生成的 C 语言代码上进行详细的标注，该标注以注释的形式出现，所以并不影响 C 语言的正常编译与执行。关于函数匹



配的基本流程如图 14-58 所示。

根据对该三层恶意行为的检测与标注,综合考虑标注结果的特点,而后实现标注系统的构建。

标注系统的框架如图 14-59 所示。这样设计标注系统的框架结构,主要基于以下分析:

1) 三层标注体系能够充分发挥基于反编译技术的恶意行为检测的优势,可从多个角度发掘并分析恶意行为,使得标注更加全面,分析结果更加可靠。

2) 三层标注保证了标注结果的准确和可靠与多种检测方法优势的发挥。

3) 三方面的结果通过图形化的方式联系在一起,给分析带来了很大的便利,同时实现了各层标注结果的协同。

总之,三层次的标注结果相互补充,相互依托,使分析更加准确和全面,结果更加可靠和丰富。行为检测模块被设计为一种具有可扩充能力的模块。恶意行为库包含技术人员合理抽象出的恶意行为的判别特征及描述说明,方便行为检测模块充分利用该库中的信息完成行为分析和检测。

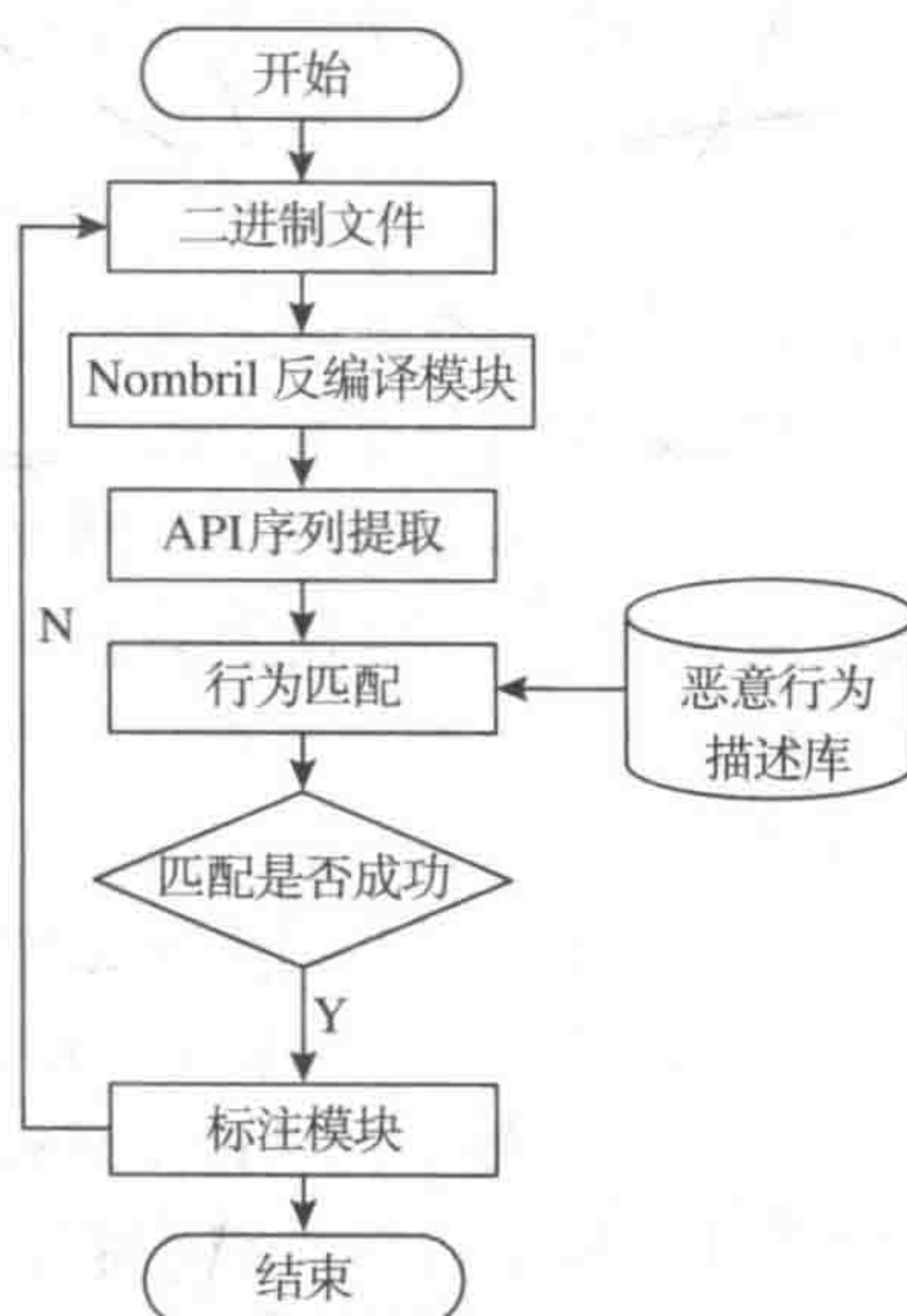


图 14-58 函数序列匹配流程

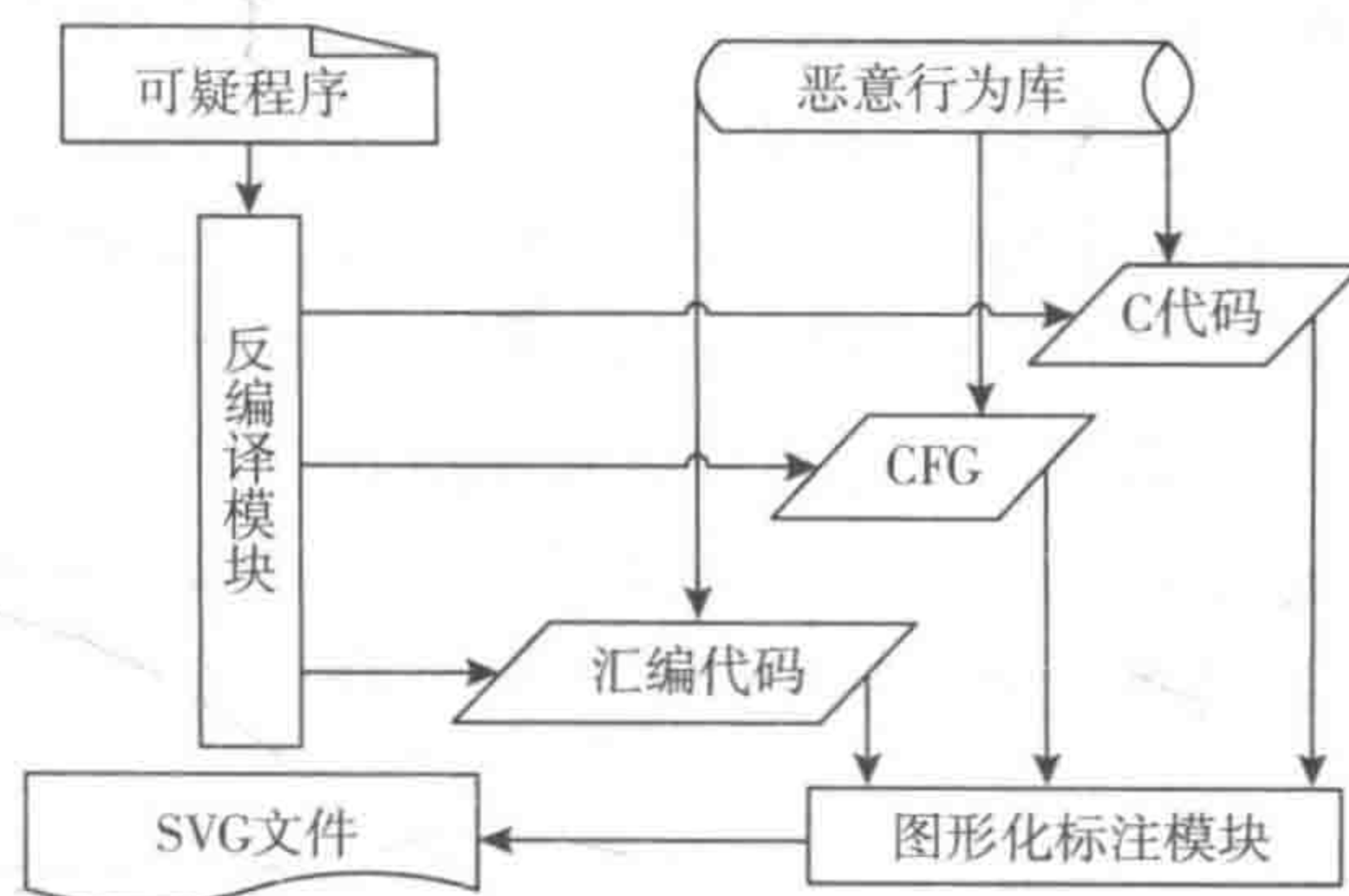


图 14-59 标注系统框架

具体的实现方法将在下面介绍。

首先对汇编层面与高级语言层面的标注实现作简要阐述。

### 汇编层标注实现

在解码完成后,行为检测模块将对汇编代码进行检测并将检测结果进行存储,需要保存的信息如表 14-6 所示。

汇编指令存储某条汇编指令的代码;识别标志位可为 0 或 1,表明该条指令是否被系统检测为恶意行为;行

表 14-6 汇编指令检测结果

存储单元	数据类型
汇编指令	字符串
识别标志位	0 或 1
行为编号	数字
序号	数字
指令地址	十六进制数



为编号代表如果识别标志位为 1，该条指令被判定为恶意行为其在恶意行为库中的编号；序号代表该条指令在所在恶意行为中的序号；指令地址记录该汇编指令的虚拟地址。

在标注模块中利用上述存储结构中的信息就可以生成汇编层的标注结果。具体步骤为：读取存储结构信息，根据行为编号在恶意行为库中获得行为的恶意等级与行为说明，将汇编代码与标注信息同时输出到结果文件中。

### 高级语言层标注的实现

在高级语言层主要的检测方法是恶意 API 函数序列检测。在检测完成后，检测模块将被检测到的 API 函数序列信息保存在临时文件中，然后标注模块在生成 C 语言代码时读取该文件，按照检测结果进行必要的注释，其中包括该函数序列的恶意程度、序列号、基本功能等。

以上两层的标注结果将通过图形化模块与控制流程层的标注结果相结合完成第一空间标注系统的构建。下面将重点阐述图形化模块的实现。

SVG 图形能够完整地呈现被分析程序的控制流程图，并依靠其优良的特性实现在控制流程图上标注被检测到的可疑行为。

在解码的同时，反编译模块会构造程序的调用图和每个过程的控制流图。之后，反编译模块将依赖于图信息对程序进行分析。控制流图是有向的连通图，它表示程序的控制流走向，由节点和节点之间的边组成。节点表示程序中的基本块，边代表节点之间的控制流。

控制流图（CFG）是描述程序控制流的一种图示方法，它反映了程序中语句的执行顺序和函数之间的相互调用关系。因为 CFG 直接表示程序的控制流路径，使得代码分析和优化的形式化更直接和高效，许多程序分析技术和编译技术都使用控制流图来表示程序的信息。图形化模块将在 CFG 上统一三层标注的结果，通过 SVG 的动态交互性可以实现标注结果之间的对应，基本块对应其汇编代码，过程对应其高级 C 代码，这种方法等于是给标注信息添加了结构化的目录，通过 CFG 指导标注结果的观察与研究，可使操作集中且简便。实验表明该方法能够科学地描述被检测到的恶意行为，使得分析更加直观、快捷。其相互关系如图 14-60 所示。

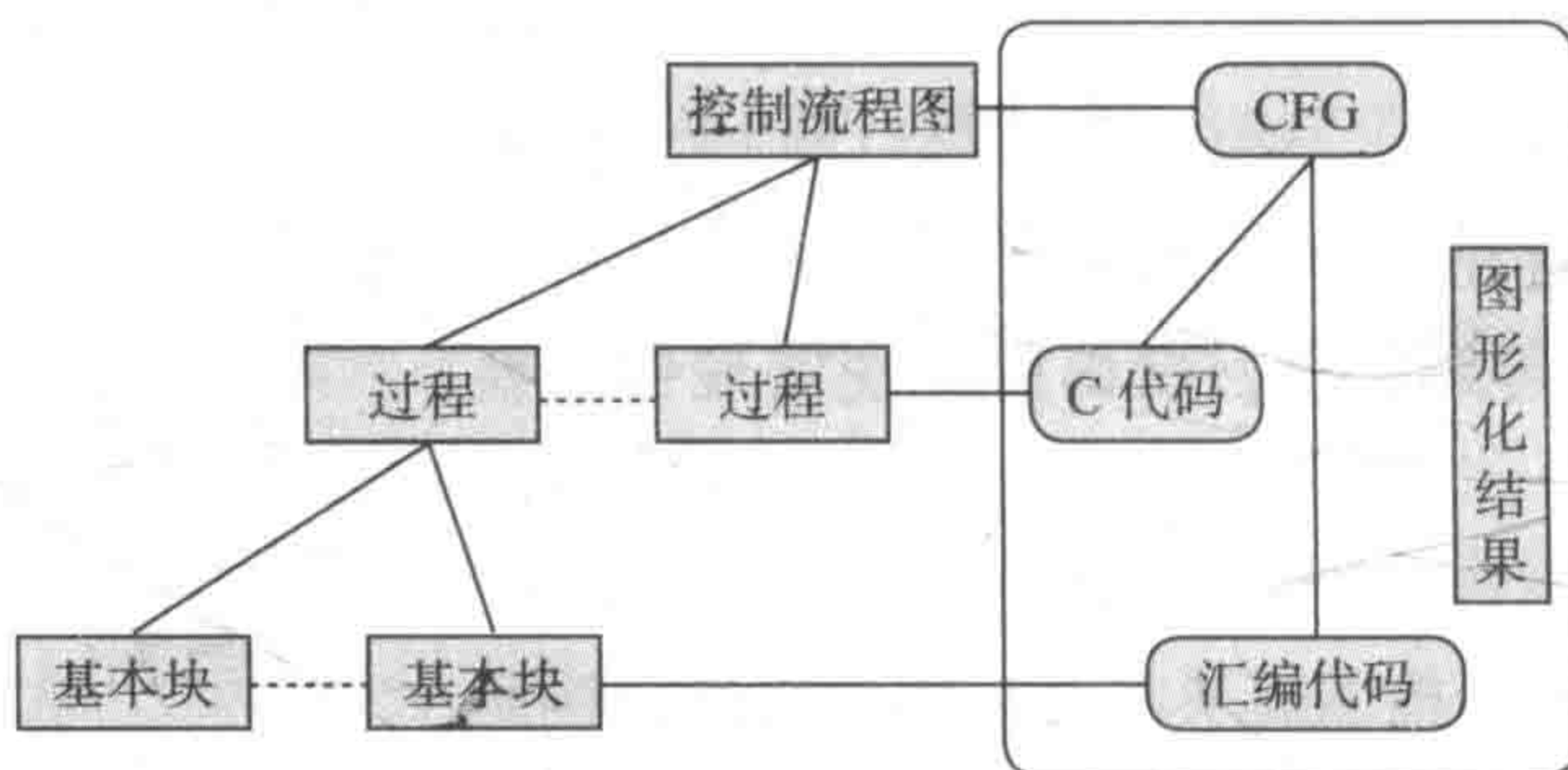


图 14-60 控制流程图上的三层标注

在控制流程图中标注恶意行为的过程不同于汇编层和高级语言层的实现方式，因为控



制流程图的结构与调用关系复杂,而且需要在控制流程图中体现汇编层与高级语言层的识别结果,这就使得为图形化模块重新建立标注结果的存储结构难度很大。经过具体分析我们采用了循环读取基本块结构信息的方法。

系统生成控制流程图后采用图 14-61 的结构存储基本块信息。

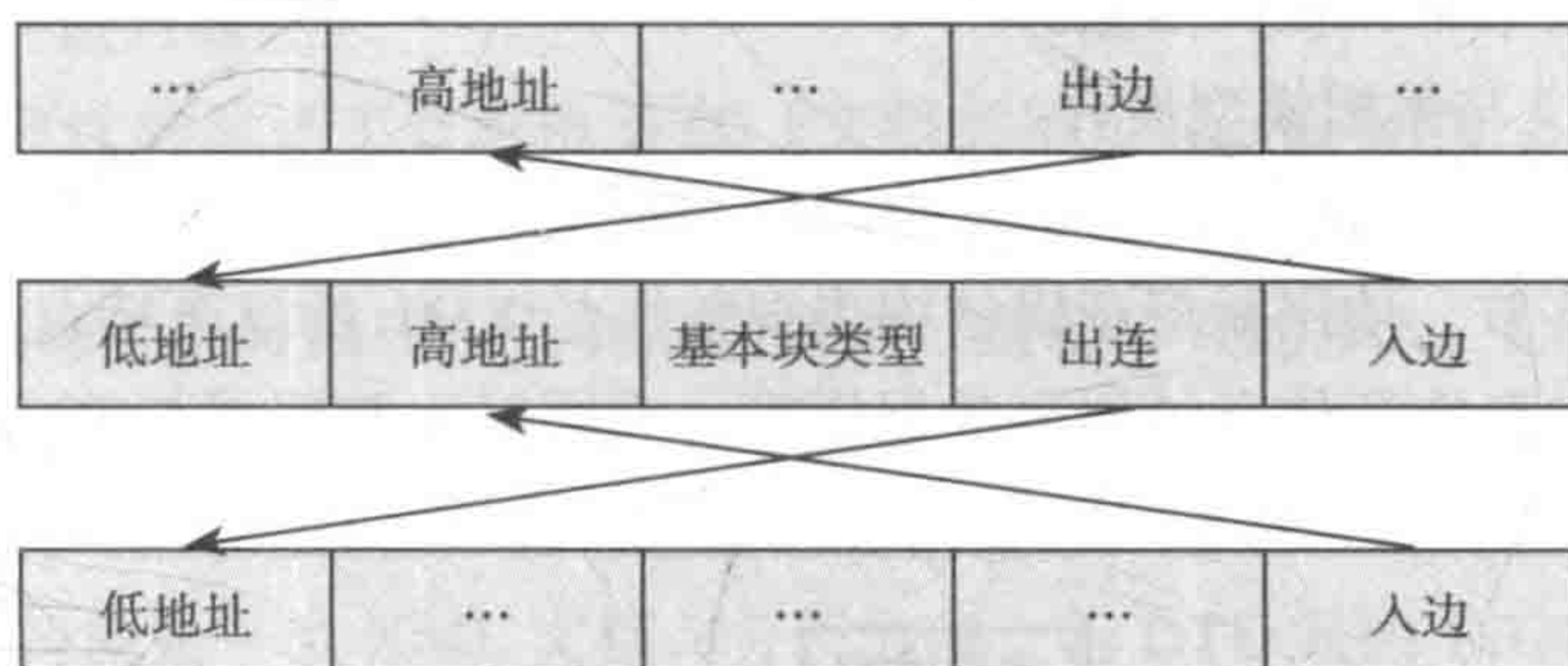


图 14-61 基本块信息的存储结构

高、低地址存储了基本块的虚拟地址;基本块类型记录了基本块的类型信息;入边存储前驱基本块的高地址;出边存储后继基本块的低地址。

有了这些信息我们可以通过循环读取过程信息,而后再在该过程中循环读取基本块信息,在这种双重循环中构造控制流程图。在使用 DOT 语言进行图形化描述以前,需要匹配汇编层和高级语言层标注的结果,将三层结合起来,并转化为 DOT 语言描述,输出到文件中。

该过程的基本处理过程如图 14-62 所示。

利用解码的结果,存储用于图形化和标注的信息,如基本块的类型、出边、入边、低地址、高地址、API 函数名、对应的 RTL 等。在生成 DOT 文件阶段,按基本块循环提取该信息,以每个基本块为单位存储图形语言,并根据恶意行为库匹配恶意行为,可以通过指令序列、函数序列等标志信息匹配。如果匹配成功则从恶意行为库中提取恶意行为描述,实现标注,将带有标注的图形信息输出到 DOT 文件,循环结束则得到完整的控制流程信息。然后使用相关软件读取相关 DOT 文件,生成 SVG 图形,方便可疑行为的分析与程序分析结果的观察。

## (2) 恶意行为描述语言

作者所在课题组设计了一种基于 XML 的

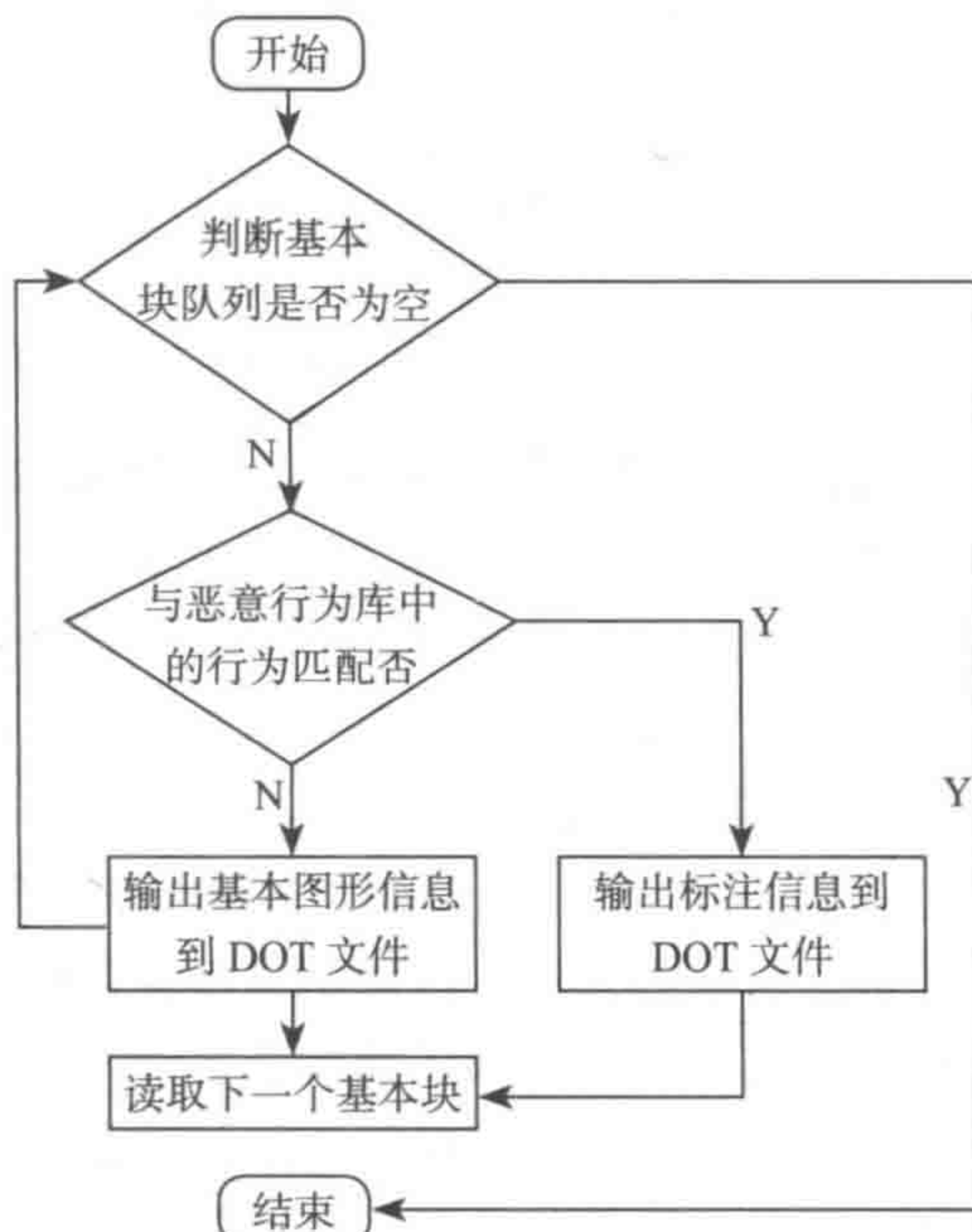


图 14-62 基本块图形化的过程



可疑行为描述语言 SBDL，对病毒特征信息进行统一的描述，使其具有统一的格式与明晰的语义，以规范基于反编译的恶意行为检测结果。

对二进制文件中含有的信息以及恶意行为使用 SBDL 进行描述。

该模块的输入为对二进制文件的检测结果和文件信息，输出为生成使用 SBDL 进行描述的二进制文件检测结果和相关信息。

1) SBDL 的定义与数据模型设计。

设计基于 XML 的可疑行为描述语言 SBDL，首先要构建描述对象的数据模型，这也是整个设计中的重要环节。使用树形逻辑结构表现数据是 XML 的显著特点，因此我们建立了如图 14-63 所示的病毒特征信息的树形数据模型，它同时也起到了与 BNF 等效的语言定义功能。

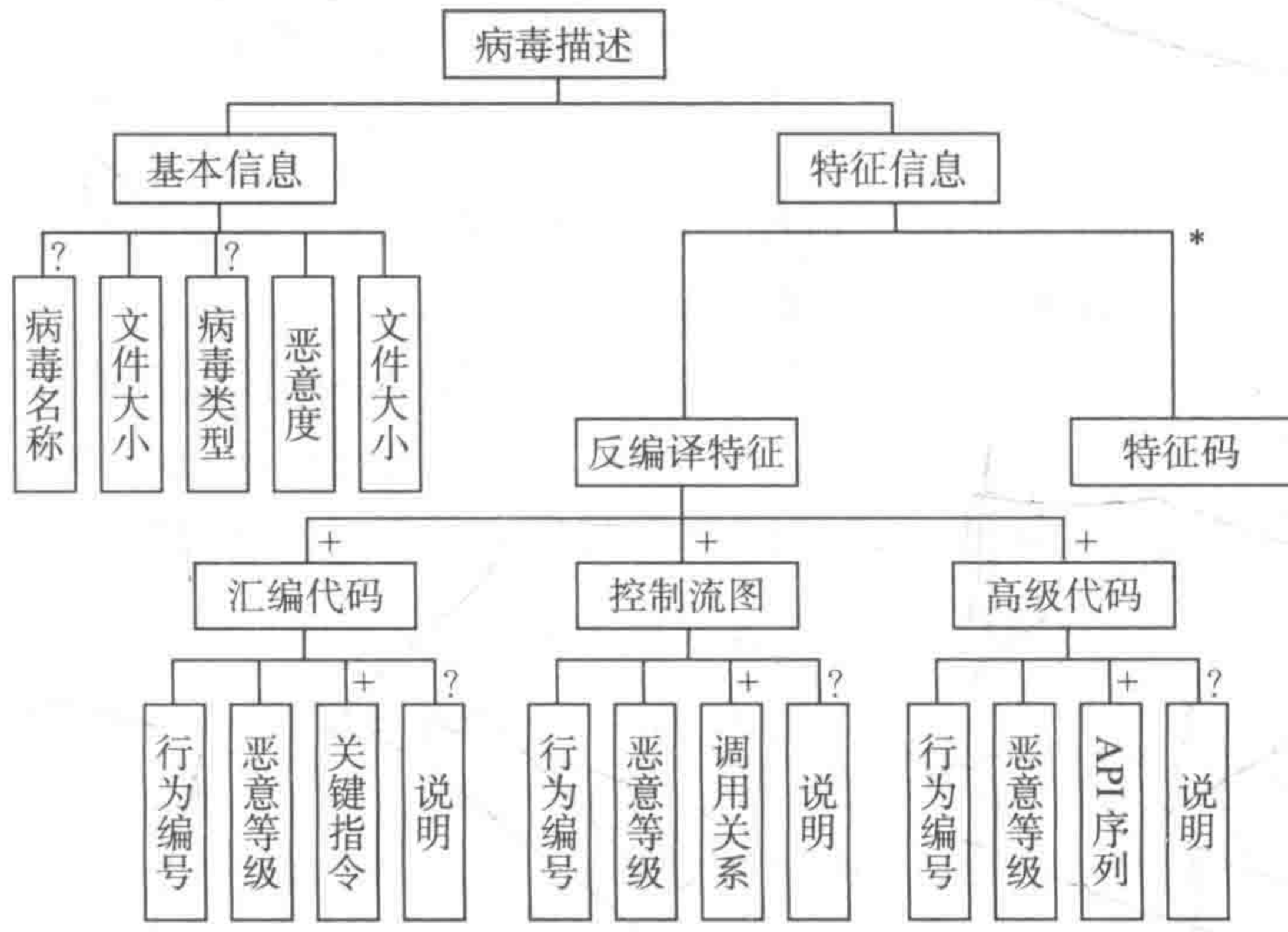


图 14-63 病毒特征信息的树形数据模型

图 14-63 给出的是一种半结构化的数据模型，值得注意的是节点在实例数据中可以不存在，也可以重复出现。图中元素对象右上角所标记的符号借用 XML 的文档类型定义 (Document Type Definition, DTD) 中的使用方法：加号 (+) 表示至少出现一次，可出现多次；问号 (?) 表示最多出现一次；星号 (\*) 说明可出现零次或多次，没有做标注的元素对象表示必须出现且仅出现一次。

从图 14-63 中可以看出，可疑行为描述规范是由基本信息、特征信息元素构成，其中特征信息是核心元素。特征信息包含反编译特征和特征码，其中特征码记录的是一般特征码检测的结果，该结果是辅助系统进行检测的，在此不作过多介绍；反编译特征包括了系统执行后所检测到的可疑行为的描述信息，对应了三层结构的各个方面，它们各自的描述又包含了独立的特征元素。从模型框图可以明显地观察到三层标注结构与前面介绍的三层图形化标注结构是完全对应的，构成了系统的第二空间。



## 2) XML Schema 设计。

满足 XML 编写规则的 XML 文档是良构的文档。由于 XML 的自描述性,我们能够分析并得出 XML 文档所描述的数据具体逻辑结构,但在具体应用中分析比较困难,不容易实现。所以一些辅助机制被设计出来,其不但可以反映 XML 的数据结构,而且也可以对 XML 文档的有效性进行验证。

XML 文档有效性的验证工作可由 DTD (文档类型定义) 或 XML Schema (XML 模式) 来完成。与 DTD 相比,XML Schema 的描述能力要强得多。XML 发布后,应用越来越广泛,出现在信息编码、串行处理、元数据描述中。随着研究的深入,DTD 在描述能力方面存在的局限性愈显突出,使得 XML 不能完成复杂的描述工作,为此 W3C 组织设计了 XML Schema。事实表明,XML Schema 具有更强的表达能力,与 DTD 相比它的优势有:

① Schema 本身就是一个 XML 文档,分析方便;但 DTD 是由 EBNF (扩展巴科斯-瑙尔范式) 书写的,所以需要设计其他分析方法。

② Schema 具有良好的扩展性。例如,Schema 文档能够依靠命名空间访问其他 Schema 文档的内容,这样使得 Schema 易于维护,扩展方便;而 DTD 只有一个文档。

③ Schema 对类型有着丰富的定义,且支持类型继承;而 DTD 类型定义非常单一,不支持继承,表达能力不强。

XML Schema 本身是一个 XML 文档,它运用 XML 的相关语法规则表示数据的内容,因而有更好的可读性和可用性。鉴于以上分析,我们在 SBDL 中采用 XML Schema 验证文档的有效性。XML Schema 中的主要内容包括元素和属性的命名、元素之间的关系以及元素的顺序等。下面是我们设计的 XML Schema 中有代表性的片段:

```
<xs:element name="反编译特征">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="汇编代码" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="控制流图" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="高级代码" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="汇编代码">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="行为编号" type="xs:string"/>
      <xs:element ref="恶意等级" type="xs:string"/>
      <xs:element ref="关键指令" minOccurs="1" maxOccurs="unbounded" type="xs:string"/>
      <xs:element ref="说明" minOccurs="0" maxOccurs="1" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="控制流图">
  <xs:complexType>
```



```
< xs:sequence>
< xs:element ref =" 行为编号 " minOccurs ="0" maxOccurs= "1"/>
...
```

### 3) SBDL 文档的生成。

使用上述模式所定义的病毒描述标记就可以生成病毒描述的 XML 文档。根据 XML 语法规则的要求,文档具备以下要素:

- ① XML 声明,包括版本信息、字符集、命名空间、模式与样式引用信息等。
- ② 病毒描述文档必须以 <病毒描述> 标记作为顶级元素(即根元素)。
- ③ 元素、属性和实体必须满足 Schema 中规定的层次、包含关系。

在具体实现的过程中,依据 XML Schema 描述的文档结构和对文档内容的限制编写相应的 SBDL 文档生成程序,读取检测结果信息便可生成 SBDL 文档,保证了 SBDL 文档的完备性与正确性。

生成 SBDL 文档病毒基本信息部分的程序代码如下:

```
std::ofstream SBDL((outputPath+prog->getRootCluster()->getName()+ "-SBDL.xml").
c_str());
SBDL << "<?xml version=\"1.0\" encoding=\"GB2312\" ?>";
SBDL << "<?xml-stylesheet type=\"text/css\"><\" href=\"<\" \"SBDL.css\"><\"?>";
SBDL << "<SBDL>\n";
SBDL << "<base>\n";
SBDL << " 载入文件的基本信息: \n";
SBDL << "<vname>";
SBDL << " 病毒名称: ";
SBDL << prog->getVName();
SBDL << "</vname>";
SBDL << "<fname>";
SBDL << " 文件大小: ";
SBDL << prog->getFileSize();
SBDL << "</fname>";
...
SBDL << " 恶意度: ";
SBDL << prog->getSusrate();
SBDL << "</malrate>";
SBDL << "</base>\n";
...
```

如代码所示,定义名称为 SBDL 的输出文件流对象,然后将恶意行为信息添加标记后进行输出,生成 SBDL 文档。

### 4) SBDL 文档的验证。

可通过 XML 解析器验证用户所编写的 XML 病毒描述文档是否遵守了病毒描述语言的 Schema 模式。XML 解析器可采用基于对象的文档对象模型(document object model)或基于事件的简单 API(simple API for XML)模型来实现。在实际应用中,编写一个 XML 解析器比较困难,目前可以直接调用一些已有的商业软件对文档进行验证。比较实用的 XML 解



析器主要有：

- ① JAXP (Java API for XML processing), 由 SUN 公司开发。
- ② XML Parser for Java 4J, 由 IBM 公司开发。
- ③ XML Parser for Java v2, 由 Oracle 公司开发。

目前不是所有的 XML 解析器都支持 XML Schema, 但当 XML Schema 规范变成一个稳定的推荐标准时, 其他解析器的新版本必然会支持它。

#### 5) SBDL 文档的显示。

XML 文档着重对数据内容的描述, 不能直接显示。可使用层叠式样式表 (Cascading Style Sheets, CSS) 与可扩展样式单语言 (eXtensible Stylesheet Language, XSL) 两种样式表显示 XML 的病毒描述文档。一个样式表可作用于多个 XML 文档; 而一个 XML 病毒描述文档也可根据不同的使用者分别使用不同的样式表, 呈现出不同的内容。该方法简单、易于实现, 一般用于对 XML 病毒描述文档的查询显示。

可通过编程实现对 XML 文档的修改、添加和提取等操作。利用 XML 文档 DOM 树模型的编程接口可以轻松地访问病毒描述文档中各元素的数据, 配合操作界面可应用于病毒的特征信息的显示, 也可作为病毒识别程序的特征信息库。

### 14.5.3 测试结果与分析

#### 1. 代码恶意性判定

##### (1) 测试对象

可执行程序库包括 VX Heavens 网站上收集的和 Windows XP 首次安装后 Windows 目录下的共 715 个可执行程序 (PE 格式文件), 对这些程序统一进行测试。

##### (2) 测试指标

指标 1: 误报率

指标 2: 检测精度

恶意程序被判定为恶意程序的情况, 称为 TP (True Positive); 正常程序被判定为正常程序的情况, 称为 TN (True Negative); 正常程序被判定为恶意程序的情况, 称为 FP (False Positive); 恶意程序被判定为正常程序的情况, 称为 FN (False Negative)。

测试指标一般包括检测率、误报率以及结合检测率和误报率的整体检测精度, 其计算方式分别为:

$$\text{检测率} = \frac{TP}{TP+FN}$$

$$\text{误报率} = \frac{EP}{TP+FP}$$

$$\text{漏报率} = \frac{FN}{TP+FN}$$



检测精度 =  $\frac{TP+TN}{TP+TN+FP+FN}$

检测结果如表 14-7 所示。

表 14-7 代码恶意性判定结果

AV 软件	TP	TN	FP	FN	检测率	误报率	漏报率	检测精度
ClamWin	284	350	8	73	79.55%	2.23%	20.45%	88.67%
瑞星	341	354	4	16	95.52%	1.12%	4.48%	97.20%
诺顿	343	357	1	14	96.08%	0.28%	3.92%	97.90%
Kaspersky	348	355	3	9	97.20%	0.84%	2.80%	98.32%
Radux	353	353	5	4	98.60%	1.96%	1.40%	98.74%

2. 敏感行为标注

Radux 系统支持对恶意程序行为检测功能，对可执行程序中的行为进行统计和分析。通过对可执行程序的分析，在多个方面对程序进行检测，从而发现程序中含有的恶意行为。

本系统对可执行程序中恶意行为的检测分为三个层次，分别对应汇编层次、中间表示层以及控制流、数据流图层，根据恶意行为库中定义的恶意行为进行识别，具体过程如图 14-64 所示。

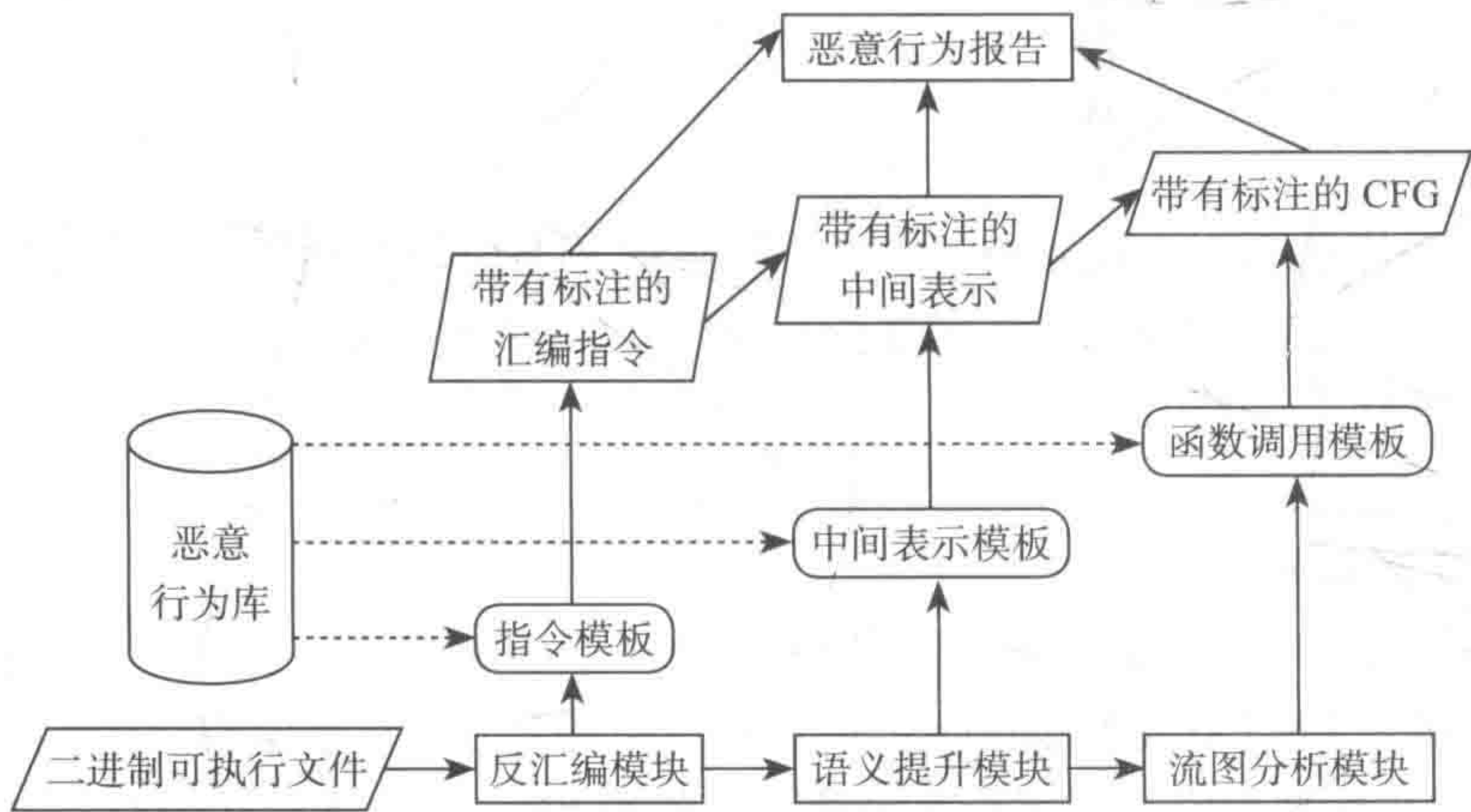


图 14-64 恶意行为检测

系统检测完可执行程序之后，给出程序的反汇编结果、反编译结果以及程序的控制流图和程序行为检测报告。

界面中显示出分析文件结果：反汇编结果、控制流图、二进制流的十六进制显示，如图 14-65 所示。单击控制流图中的黄色标注，可以打开流图中过程所对应的 C 代码，如图 14-66 所示。

还可以获取分析文件的整体反编译结果 C 代码显示，如图 14-67 所示。



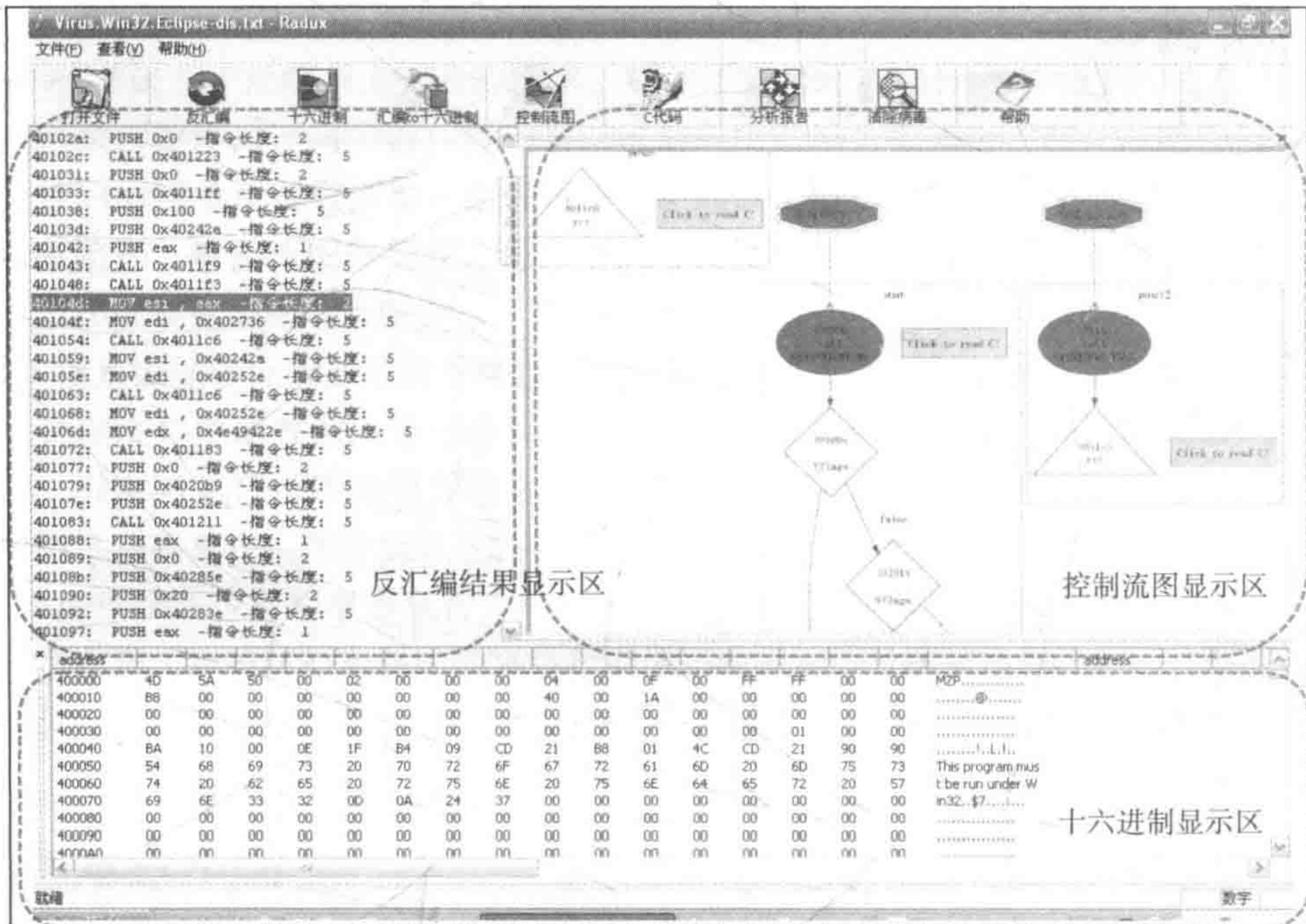


图 14-65 标注区域示例

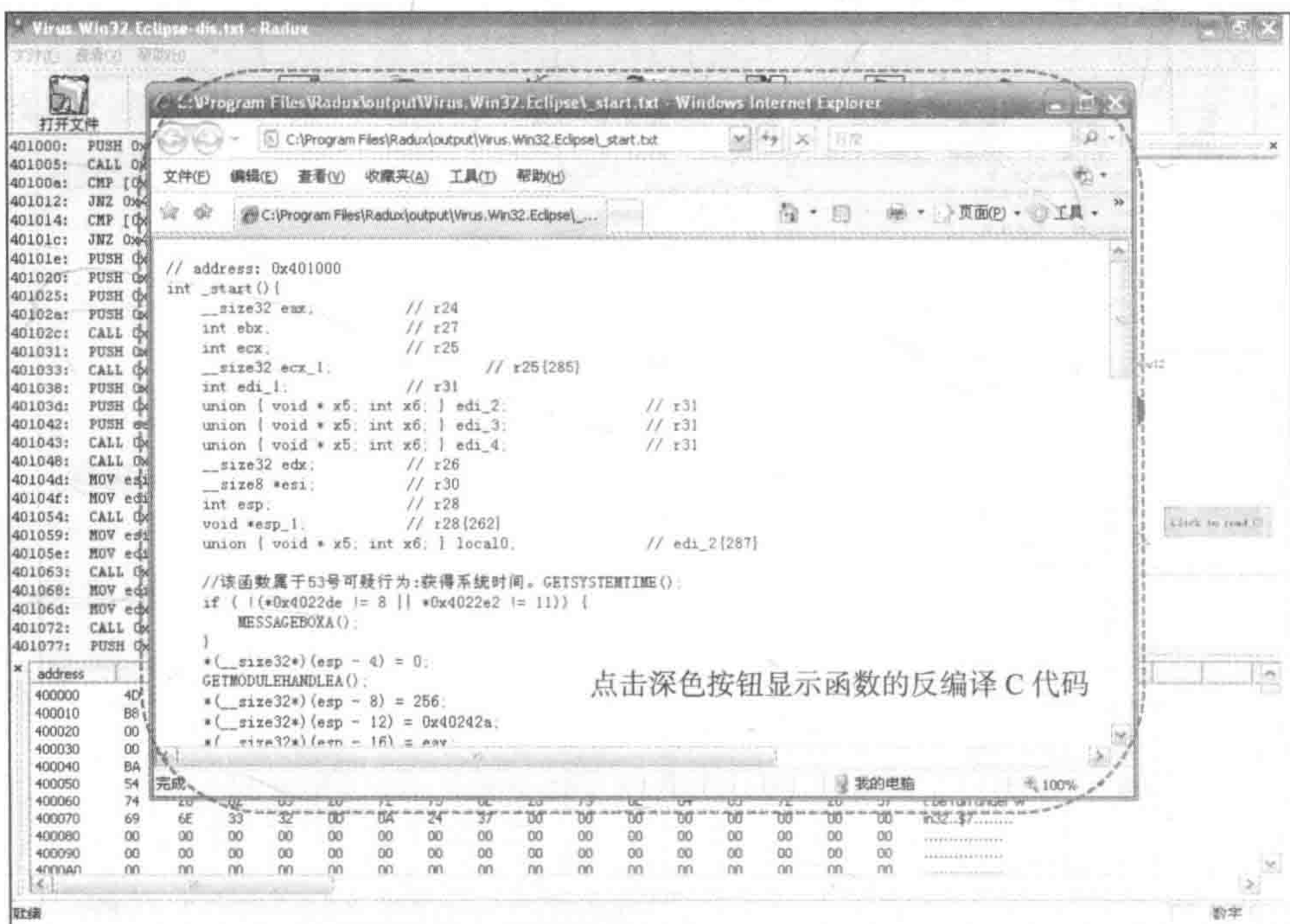


图 14-66 函数的反编译 C 代码示例



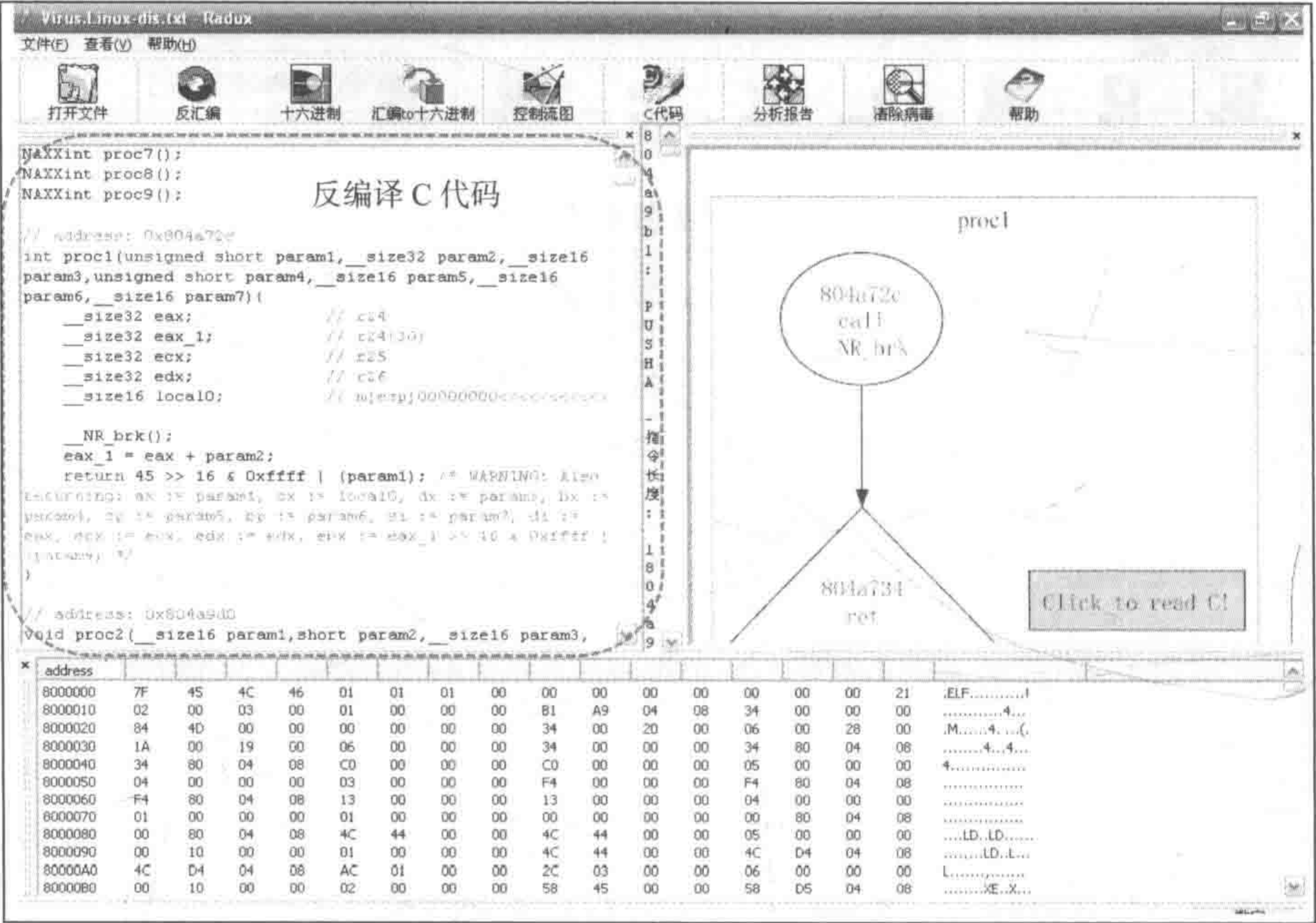


图 14-67 代码整体反编译 C 代码示例

最后，可以获取分析文件的分析报告显示，如图 14-68 所示。

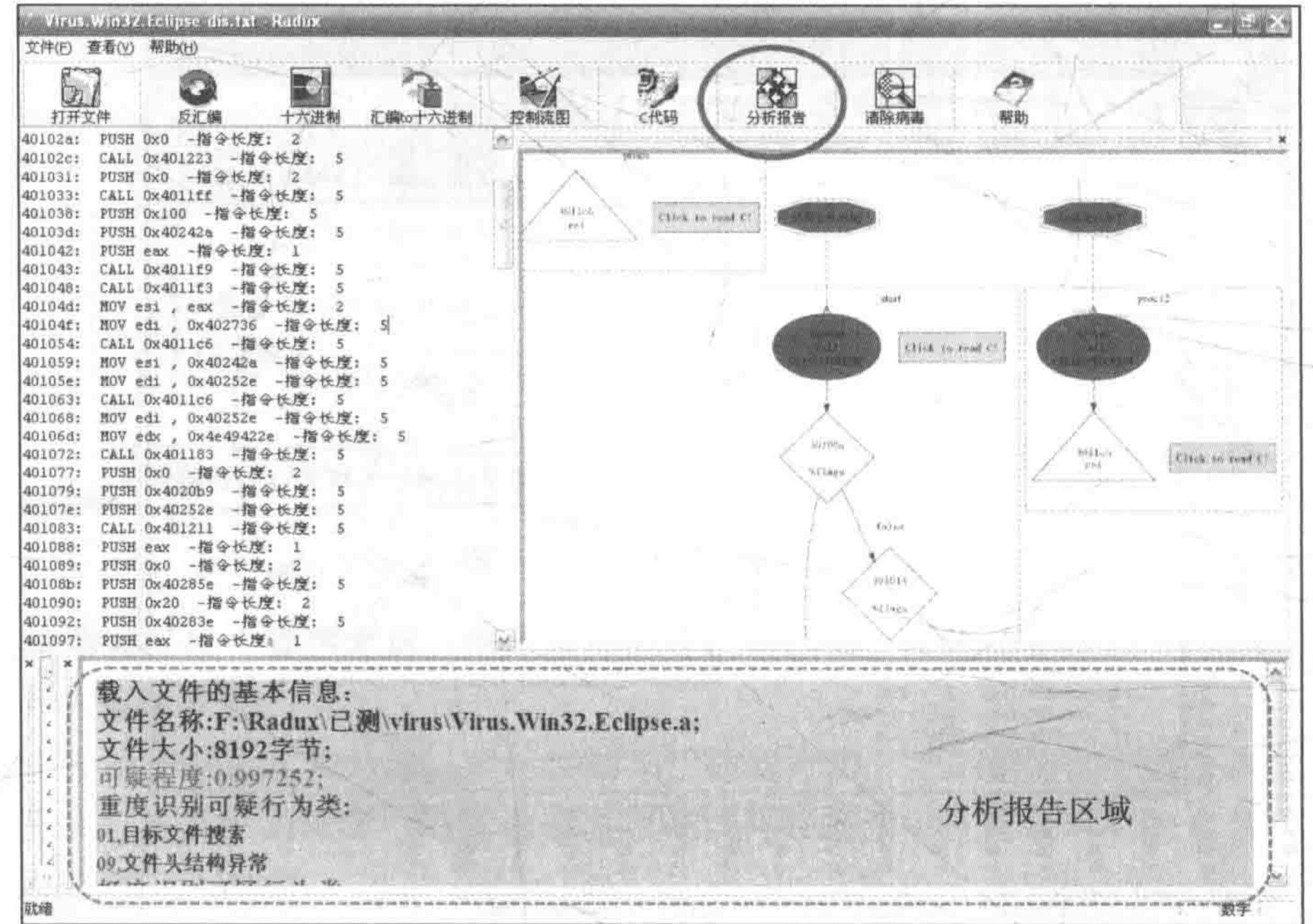


图 14-68 分析报告示例



### 3. 对 Radux 的第三方测试结论

1) 该系统基于反编译逆向分析技术, 构建了反编译逆向分析框架, 实现了对多平台下二进制可执行程序的逆向分析: 首先将二进制码反汇编为汇编码, 在此阶段能够应对恶意代码常使用的混淆及变形技术, 生成正确的中间表示结果。然后在反汇编的基础上能够抽象出控制流图、数据流图等信息, 最后给出可读性更强的 C 代码, 从而使得对可执行文件的分析工作能够更加深入地展开。

2) 该系统能够分析多平台下可疑二进制程序中的恶意行为, 能够在反编译过程中准确获取二进制文件中函数调用等可疑行为信息, 并在此基础上对程序的可疑性给出量化的分析结果, 并对可执行程序是否为恶意程序给出初步的判定结果。该系统采用的基于行为的可执行程序分析技术, 使系统具有较好的应对恶意代码变体的能力。

3) 该系统在分析程序恶意行为的基础上, 在汇编代码层、控制流图层, 以及生成的 C 代码层上实现了对可执行程序中的恶意行为的正确识别以及多层次标注, 标注信息清晰、明确, 方便分析人员快速掌握该可执行程序的相关信息, 以及对恶意代码进行准确定位并分析。

## 14.6 本章小结

作为信息安全中的主要分析方法之一, 反编译有着诸多应用。本章在充分了解不同特征在恶意代码分析中的不同作用以及恶意代码反编译对抗技术的基础上, 介绍了反编译技术针对条件跳转混淆、指令重叠混淆、子程序异常返回以及不透明谓词混淆等典型混淆技术的改进, 最后通过作者课题组开发的基于反编译的恶意代码检测与标注系统 Radux, 介绍了反编译技术在恶意代码判定过程中的应用。



## 参考文献

- [1] Levine J R, Mason T, Brown D. Lex & Yacc[M]. O'Reilly Media, Inc., 1992.
- [2] Brian J Gough. An Introduction to GCC[M]. Network Theory Ltd., 2004.
- [3] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]. Proceedings of the IEEE International Symposium on Code Generation and Optimization, 2004.
- [4] Eagle C. The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler[M]. No Starch Press, 2011.
- [5] Redoc. OllyDbg VS SoftICE: 用户级程序调试秘笈 [J]. 黑客防线, 2006(3):111-114.
- [6] Bellard F. QEMU, A Fast and Portable Dynamic Translator[C]. Proceedings of the Usenix Technical Conference USENIX Association, 2005:41-46.
- [7] Yin H, Song D. Temu: Binary Code Analysis via Whole-system Layered Annotative Execution[R]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3, 2010.
- [8] Ramsey N, Fernández M F. Specifying Representations of Machine Instructions[J]. ACM Transactions on Programming Languages & Systems, 1997, 19(3):492-524.
- [9] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, X Sean Wang. Moving Target Defense—Creating Asymmetric Uncertainty for Cyber Threats[M]. Springer, 2011.
- [10] 陈火旺, 刘春林, 谭庆平, 赵克佳, 刘越. 程序设计语言编译原理 [M]. 3 版. 北京: 国防工业出版社, 2013.
- [11] 陈意云. 编译原理和技术 [M]. 2 版. 合肥: 中国科学技术大学出版社, 1997.
- [12] Randal E Bryant, David R O'Hallaron. Computer Systems: A Programmer's Perspective[M]. Prentice Hall, 2010.
- [13] 李劲华, 陈宇. 编译原理与技术练习解答与实验指导 [M]. 2 版. 北京: 北京邮电大学出版社, 2014.
- [14] 黄贤英, 王柯柯, 刘洁, 曹琼. 编译原理重点难点分析·习题解析·实验指导 [M]. 北京: 机械工业出版社, 2008.
- [15] 张昱, 陈意云. 编译原理实验教程 [M]. 北京: 高等教育出版社, 2009.
- [16] 张幸儿. 计算机编译原理 [M]. 3 版. 北京: 科学出版社, 2008.



- [17] Kenneth C Loudon. 编译原理及实践 [M]. 冯博琴, 冯岚, 等译. 北京: 机械工业出版社, 2000.
- [18] Dick Grune, Henri E Bal, Criel J H Jacobs, Koen G Langendoen. 现代编译程序设计 [M]. 冯博琴, 傅向华, 等译. 北京: 人民邮电出版社, 2003.
- [19] Andrew W Appel. 现代编译原理 C 语言描述 [M]. 赵克佳, 黄春, 沈志宇, 译. 北京: 人民邮电出版社, 2006.
- [20] Charles N Fischer, Richard J LeBlanc. 编译器构造 C 语言描述 [M]. 郑启龙, 姚震, 译. 北京: 机械工业出版社, 2005.
- [21] 张幸儿, 戴新宇. 编译原理: 编译程序构造实践教程 [M]. 北京: 人民邮电出版社, 2010.
- [22] 王博俊, 张宇. 自己动手写编译器、链接器 [M]. 北京: 清华大学出版社, 2015.
- [23] 许畅, 陈嘉, 朱晓瑞. 编译原理实践与指导教程 [M]. 北京: 机械工业出版社, 2015.
- [24] Cifuentes C. Reverse Compilation Techniques [D]. Queenslnd University of Technology, 1994.
- [25] 陈渝, 李明, 杨晔, 等. 源码开放的嵌入式系统软件分析与实践——基于 SkyEye 和 ARM 开发平台 [M]. 北京: 航空航天大学出版社, 2004.
- [26] Ramsey N, Davidson J W, Fernández M F. Design Principles for Machine-Description Languages [R]. Submitted to ACM Transactions on Programming Languages and Systems, 2001.
- [27] Ramsey N, Davidson J W. Specifying Instructions' Semantics using RTL[R]. Technical Report, Department of Computer Science, University of Virginia, 1999.
- [28] G Winskel. 程序设计语言的形式语义 [M]. 宋国新, 等译. 北京: 机械工业出版社, 2004.
- [29] Olinsky R, Lindig C, Ramsey N. Staged Allocation: Engineering the Specification and Implementation of Procedure Calling Conventions [C]. Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages, 2006.
- [30] 申利民. 应用于病毒分析的逆编译 [J]. 微机发展, 2003, 13(1): 8-10.
- [31] 刘宗田, 李力. 8086C 反编译数据类型恢复技术 [J]. 计算机研究与发展, 1992, 31(4): 44-51.
- [32] 刘宗田. 68000C 反编译系统中的变量类型恢复 [J]. 微计算机应用, 1989, 39(6): 16-19.
- [33] Richard Johnsonbaugh. 离散数学 [M]. 石纯一, 金津, 张新良, 等译. 北京: 电子工业出版社, 2005.
- [34] C Cifuentes, M V Emmerik, N Ramsey, et al. Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework[R]. Sun Labs Tech Report TR-2002-105, 2002.
- [35] A V Aho, R Sethi, J D Ullman. 编译原理技术与工具 (英文版) [M]. 北京: 人民邮电出版社, 2001.
- [36] C Kruegel, W Robertson, F Valeur, et al. Static Disassembly of Obfuscated Binaries[C]. Proceedings of the 13th Conference on USENIX Security Symposium, 2004.
- [37] 陈龙, 武成岗, 谢海斌, 等. 二进制翻译中解析多目标分支语句的图匹配方法 [J]. 计算机研究与发展, 2008, 45(10): 1789-1798.
- [38] C Cifuentes, M V Emmerik. Recovery of Jump Table Case Statements from Binary Code[J]. Science of Computer Programming, 2001, 40(2-3): 171-188.
- [39] C Cifuentes, K Gough. Decompilation of Binary Programs[J]. Software-Practice and Experience, 1995, 25(7): 811-829.



- [40] 陈凯明. 逆编译中几项关键技术研究 [D]. 合肥: 合肥工业大学, 2003.
- [41] M V Emmerik. Static Single Assignment for Decompilation[D]. The University of Queensland School of Information Technology and Electrical Engineering, 2007.
- [42] L Jiang, Z Su. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing[C]. Proceedings of International Symposium on Software Testing and Analysis, 2009.
- [43] J Caballero, N M Johnson, S McCamant, et al. Binary Code Extraction and Interface Identification for Security Applications[C]. Proceedings of the 17th Annual Network and Distributed System Security Symposium, 2010.
- [44] B D Sutter, B D Bus, D Bosschere, et al. On the Static Analysis of Indirect Control Transfers in Binaries[C]. Proceedings of International Conference on Parallel and Distributed processing Techniques and Applications, 2000.
- [45] B Schwarz, S Debray, G Andrews. Disassembly of Executable Code Revisited[C]. Proceedings of the Ninth Working Conference on Reverse Engineering , 2002.
- [46] Program Analysis [EB/OL]. [http://en.wikipedia.org/wiki/Program\\_analysis](http://en.wikipedia.org/wiki/Program_analysis). Retrieved 2011-03-14.
- [47] Allen F E. Control flow analysis[C]. Proceedings of ACM SIGPLAN Notices - Proceedings of a Symposium on Compiler Optimization, 1970.
- [48] The-Difference-Between-Extended-Basic-Blocks-and-Traces [DB/OL]. <http://www.masonchang.com/blog/2009/1/13/the-difference-between-extended-basic-blocks-and-traces.html>.
- [49] Extended Basic Block – Wiktionary [DB/OL]. [http://en.wiktionary.org/wiki/extended\\_basic\\_block](http://en.wiktionary.org/wiki/extended_basic_block).
- [50] John Cocke. Global Common Subexpression Elimination[C]. Proceedings of ACM SIGPLAN Notices - Proceedings of a Symposium on Compiler optimization, 1970.
- [51] Mark Weiser. Program slicing[C]. Proceedings of the 5th International Conference on Software Engineering, 1981.
- [52] Evans J S, Trimper G L. Itanium Architecture for Programmers: Understanding 64-bit Processors and EPIC Principles[M]. Prentice Hall Professional, 2003.
- [53] Larus J R, Schnarr E. EEL: Machine-Independent Executable Editing [C]. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1995.
- [54] Theiling H. Extracting Safe and Precise Control Flow from Binaries[C]. Proceedings of the 7th International Conference on Real-Time Systems and Applications, 2000.
- [55] Cifuentes C. Interprocedural Data Flow Decompilation[J]. Journal of Programming Languages, 1996, 4(2):77-99.
- [56] Cifuentes C, Simon D, Fraboulet A. Assembly to High-Level Language Translation[C]. Proceedings of the International Conference on Software Maintenance, 1998.
- [57] Cifuentes C, Emmerik M V. Recovery of Jump Table Case Statements from Binary Code[C]. Proceedings of the 7th International Workshop on Program Comprehension, 1999.
- [58] Kästner D, Wilhelm S. Generic Control Flow Reconstruction from Assembly Code[C]. Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software



- and Compilers for Embedded Systems, 2002.
- [59] Myreen M. Formal Verification of Machine-code Programs [D]. University of Cambridge, 2008.
  - [60] Kinder J, Zuleger F, Veith H. An Abstract Interpretation-based Framework for Control Flow Reconstruction from Binaries [C]. Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2009.1:214-228.
  - [61] Flexeder A, Mihaila B, Petter M, et al. Interprocedural Control Flow Reconstruction[C]. Proceedings of the 8th Asian Conference on Programming Languages and Systems (APLAS), 2010.
  - [62] Balakrishnan G, Reps T. Analyzing Memory Accesses in x86 Executables[M]. Springer Berlin Heidelberg, 2004.
  - [63] Kinder J, Veith H. Jakstab: A Static Analysis Platform for Binaries[C]. Proceedings of the 20th International Conference on Computer Aided Verification, 2008.
  - [64] Sutter B D, Bus B D, Bosschere K D. Link-time Binary Rewriting Techniques for Program Compaction[J]. ACM Transactions on Programming Languages and Systems, 2005, 27(5):882-945.
  - [65] Chen K, Feng D G, Su P R. Exploring Multiple Execution Paths Based on Dynamic Lazy Analysis[J]. Chinese Journal of Computer, 2010, 33(3):193-503.
  - [66] Crandall J R, Wu S F, Chong F T. Architectural Support for Protecting Control Data[J]. ACM Transactions on Architecture and Code Optimization, 2006, 3(4):359-389.
  - [67] Newsome J, Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software[C]. Proceedings of the 12th Annual Network and Distributed System Security Symposium, 2005.
  - [68] Moser A, Kruegel C, Kirda E. Exploring Multiple Execution Paths for Malware Analysis[C]. Proceedings of the IEEE Symposium on Security and Privacy, 2007.
  - [69] 戴超. 二进制恶意代码分析关键技术研究 [D]. 郑州: 解放军信息工程大学, 2016.



## 推荐阅读



### Arduino可穿戴设备开发

作者: Tony Olsson ISBN: 978-7-111-54132-5 定价: 49.00元



### Arduino从入门到精通: 创客必学的13个技巧

作者: John Baichtal ISBN: 978-7-111-54811-9 定价: 59.00元



### Arduino计算机视觉编程

作者: Ozen Ozkaya 等 ISBN: 978-7-111-55126-3 定价: 49.00元



### 深入理解Arduino: 移植和高级开发

作者: Rick Anderson 等 ISBN: 978-7-111-54140-0 定价: 59.00元



编译系统是计算机系统的重要组成部分，理解编译与反编译技术对构建高性能的计算机系统、保障计算机系统的安全性有着重要意义。本书结合作者多年的科研、工程 and 教学实践，创新性地将编译与反编译的内容结合在一本书中论述，使读者从正向和逆向两个角度理解编译系统的构造原理和实现方法，对从事软件开发、信息安全等领域的技术和研究人员极具参考价值。

### 主要特色

- 从正向和逆向两个角度介绍编译系统和编译技术，深化读者对系统的认识和理解以及逆向工程能力。
- 通过大量的案例和实践项目，展现深刻的编译与反编译理论，特别是前沿的理论，在实战中理解理论，将理论与技术有机结合。
- 既涵盖编译与反编译的经典理论和技术，也对该领域最新的研究和技术进展进行介绍，特别是系统地介绍了反编译领域的理论和技术。



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机/网络安全

ISBN 978-7-111-56617-5



9 787111 566175 >

定价: 79.00元