燕山大學
YANSHAN UNIVERSITY

编译原理实验指导书

Compilers Principles Experiment Instruction Book

陈贺敏　王林

燕山大学软件工程系

目录

实验 1 词法分析

1.1 实验目的

（1）理解有穷自动机及其应用。

（2）掌握 NFA 到 DFA 的等价变换方法、DFA 最小化的方法。

（3）掌握设计、编码、调试词法分析程序的技术和方法。

1.2 实验任务

编写一个程序对输入的源代码进行词法分析，并打印分析结果。（以下两个任务二选一）

1、借助词法分析工具 GNU Flex，编写一个对使用 C--语言书写的源代码进行词法分析（C--语言的文法参见附录 A），并使用 C 语言完成。

2、自己编写一个 PL/0 词法分析程序，语言不限。

1.3 实验内容

1.3.1 实验要求

你的程序要能够查出源代码中可能包含的词法错误：

词法错误（**错误类型 A**）：（以输入 C--源代码为例）即出现 C--词法中未定义的字符以及任何不符合 C--词法单元定义的字符；

1.3.2 输入格式

1、如果你选择的是任务 1，那么你的程序的输入是一个包含 C--源代码的文本文件，程序需要能够接收一个输入文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 即可获得以 test1 作为输入文件的输出结果。

2、如果你选择的是任务 2，那么你的程序输入是一个包含 PL/0 源代码的文本文件，程序需要能够接收一个输入文件名作为参数，以获得相应的输出结果。

1.3.3 输出格式

实验一要求通过标准输出打印程序的运行结果。对于那些包含词法错误的输入文件，只要输出相关的词法有误的信息即可。在这种情况下，注意不要输出任何与语法树有关的内容。要求输出的信息包括错误类型、出错的行号以及说明文字，

其格式为：Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类（**即词法错误为错误类型 A**），否则将影响你的实验评分。**注意，**输入文件中可能会包含一个或者多个词法错误（但输入文件的同一行中保证不出现多个错误），你的程序需要将这些错误全部

报告出来，每一条错误提示信息在输出中单独占一行。

对于那些没有任何词法错误的输入文件，你的程序需要打印每一个词法单元的名称以及与其对应的词素，无需打印行号。词法单元名与相应词素之间以一个冒号和一个空格隔开。每一条词法单元的信息单独占一行。

表 1-1 词法单元的例子

词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	Pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个"之间，除"以外的任何字符	"core dumped"

表 1-1 给出了一些常见的词法单元、非正式描述的词法单元的模式，并给出了一些示例词素（以下若无特殊说明，词素就是最小词法单位）。下面说明上述概念在实际中是如何应用的。在 C 语句

printf(“Total = %d\n”,score);

中，printf 和 score 都是和词法单元 id 的模式匹配的词素，而“Total = %d\n”则是一个和 literal 匹配的词素。

在很多程序设计语言中，下面的类别覆盖了大部分或所有的词法单元：

1）关键字。一个关键字的类型就是该关键字本身。

2）运算符。它可以表示单个运算符，也可以像表 1-1 中的 comparison 那样，表示一类运算符。

3）标识符。一个表示所有标识符的词法单元。

4）常量。一个或多个表示常量的词法单元，比如数字和字面值字符串。

5）界符。每一个标点符号有一个词法单元，比如左右括号、逗号和分号。

1.3.4 测试环境

任务 1：你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基

本上不会出现兼容性方面的问题。注意，实验一的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

1.3.5 提交要求

1) 任务 1：Flex 以及 C 语言（任务 2 可以是 Java、C、Python 等任意语言，但需针对 PL/0 语言）的可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

1.3.6 样例

此部分样例是以**任务 1** 为例给出的，请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

说明：选择任务 2 的同学，可以参考任务 1 的样例，自行设计测试样例，以反映出正确输出。

样例 1：

输入（行号是为标识需要，并非样例输入的一部分，后同）：

```
1  int main()
2  {
3      int i = 1;
4      int j = ~i;
5  }
```

输出：

这个程序存在词法错误。第 4 行中的字符“~”没有在我们的 C--词法中被定义过，因此你的程序可以输出如下的错误提示信息：

```
Error type A at Line 4: Mysterious character "~".
```

样例 2：

输入：

```
1  int inc()
2  {
3      int   i;
4      i =100;
5  }
```

输出：

这个程序非常简单，也没有任何词法错误，因此你的程序需要输出每一个词法单元信息：

TYPE: int

ID: inc

LP: (

RP:)

LC: {

TYPE: int

ID: i

SEMI: ;

ID: i

RELOP: =

INT: 100

SEMI: ;

RC: }

1.4 实验指导

词法分析和语法分析这两块，可以说是在整个编译器当中被自动化得最好的部分。也就是说，即使没有任何的理论基础，在掌握了工具的用法之后，也可以在短时间内做出功能很全很棒的词法分析程序和语法分析程序。当然这并不意味着，词法分析和语法分析部分的理论基础并不重要。恰恰相反，这一部分被认为是计算机理论在工程实践中最成功的应用之一，对它的介绍也是编译理论课中的重点。但本节指导内容的重点不在于理论而在于工具的使用。

本节指导内容将主要介绍词法分析工具 GNU Flex。如前所述，完成实验一并不需要太多的理论基础，只要看完并掌握了本节的大部分内容即可完成实验一。

1.4.1 词法分析概述

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时它也要有能力报告相应的错误。词法分析程序是编译器所有模块中唯一读入并处理输入文件中每一个字符的模块，它使得后面的语法分析阶段能在更高抽象层次上运作，而不必纠结于字符串处理这样的细节问题。

高级程序设计语言大多采用英文作为输入方式，而英文有个非常好的性质，就是它比较容易断词：相邻的英文字母一定属于同一个词，而字母与字母之间插入任何非字母的字符（如空格、运算符等）就可以将一个词断成两个词。判断一个词是否符合语言本身的词法规范也

相对简单，一个直接的办法是：我们可以事先开一张搜索表，将所有符合词法规范的字符串都存放在表中，每次我们从输入文件中断出一个词之后，通过查找这张表就可以判断该词究竟合法还是不合法。

正因为词法分析任务的难度不高，在实用的编译器中它常常是手工写成，而并非使用工具生成。例如，我们下面要介绍的这个工具 GNU Flex 原先就是为了帮助 GCC 进行词法分析而被开发出来的，但在 4.0 版本之后，GCC 的词法分析器已经一律改为手写了。不过，实验一如果选择使用工具来做，而词法分析程序生成工具所基于的理论基础，是计算理论中最入门的内容：**正则表达式（Regular Expression）**和**有限状态自动机（Finite State Automata）**。

一个正则表达式由特定字符串构成，或者由其它正则表达式通过以下三种运算得到：

1) **并运算（Union）**：两个正则表达式 r 和 s 的并记作 r | s，意为 r 或 s 都可以被接受。

2) **连接运算（Concatenation）**：两个正则表达式 r 和 s 的连接记作 rs，意为 r 之后紧跟 s 才可以被接受。

3) **Kleene 闭包（Kleene Closure）**：一个正则表达式 r 的 Kleene 闭包记作 r*，它表示：| r | rr | rrr | …。

有关正则表达式的内容在课本的理论部分讨论过。正则表达式之所以被人们所广泛应用，一方面是因为它在表达能力足够强（基本上可以表示所有的词法规则）的同时还易于书写和理解；另一方面也是因为判断一个字符串是否被一个特定的正则表达式接受可以做到非常高效（在线性时间内即可完成）。比如，我们可以将一个正则表达式转化为一个 **NFA（即不确定的有限状态自动机）**，然后将这个 NFA 转化为一个 **DFA（即确定的有限状态自动机）**，再将转化好的 DFA 进行化简，之后我们就可以通过模拟这个 DFA 的运行来对输入串进行识别了。具体的 NFA 和 DFA 的含义，以及如何进行正则表达式、NFA 及 DFA 之间的转化等，请参考课本的理论部分。这里我们仅需要知道，前面所述的所有转化和识别工作，都可以由工具自动完成。而我们所需要做的，仅仅是为工具提供作为词法规范的正则表达式。

1.4.2 GNU Flex 介绍

Flex 的前身是 Lex。Lex 是 1975 年由 Mike Lesk 和当时还在 AT&T 做暑期实习的 Eric Schmidt，共同完成的一款基于 Unix 环境的词法分析程序生成工具。虽然 Lex 很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。后来伯克利实验室的 Vern Paxson 使用 C 语言 重写 Lex，并将这个新的程序命名为 Flex（意为 Fast Lexical Analyzer Generator）。无论在效 率上还是在稳定性上，Flex 都远远好于它的前辈 Lex。我们在 Linux 下使用的是 Flex 在 GNU License 下的版本，称作 GNU Flex。

GNU Flex 在 Linux 下的安装非常简单。你可以去它的官方网站上下载安装包自行安装，不过在基于 Debian 的 Linux 系统下，更简单的安装方法是直接在命令行敲入如下命令：

sudo apt-get install flex

虽然版本不一样，但 GNU Flex 的使用方法与课本上介绍的 Lex 基本相同。首先，我们需要自行完成包括词法规则等在内的 Flex 代码。至于如何编写这部分代码我们在后面会提到，现在先假设这部分写好的代码名为 lexical.l。随后，我们使用 Flex 对该代码进行编译：

flex lexical.l

编译好的结果会保存在当前目录下的 lex.yy.c 文件中。打开这个文件你就会发现，该文件本质上就是一份 C 语言的源代码。这份源代码里目前对我们有用的函数只有一个，叫做 yylex()，该函数的作用就是读取输入文件中的一个词法单元。我们可以再为它编写一个 main 函数：

```
1   int main(int argc, char** argv) {
2     if (argc > 1) {
3       if (!(yyin = fopen(argv[1], "r"))) {
4         perror(argv[1]);
5         return 1;
6       }
7     }
8     while (yylex() != 0);
9     return 0;
10 }
```

这个 main 函数通过命令行读入若干个参数，取第一个参数为其输入文件名并尝试打开该输入文件。如果文件打开失败则退出，如果成功则调用 yylex()进行词法分析。其中，变量 yyin 是 Flex 内部使用的一个变量，表示输入文件的文件指针，如果我们不去设置它，那么 Flex 会将它自动设置为 stdin（即**标准输入**，通常连接到键盘）。注意，如果你将 main 函数独立设为一个文件，则需要声明 yyin 为外部变量：extern FILE* yyin。

将这个 main 函数单独放到一个文件 main.c 中（你也可以直接放入 lexical.l 中的用户自定义代码部分，这样就可以不必声明 yyin；你甚至可以不写 main 函数，因为 Flex 会自动给你配一个，但不推荐这么做），然后编译这两个 C 源文件。我们将输出程序命名为 scanner：

gcc main.c lex.yy.c -lfl -o scanner

注意编译命令中的“-lfl”参数不可缺少，否则 GCC 会因为缺少库函数而报错。之后我们

就可以使用这个 scanner 程序进行词法分析了。例如，想要对一个测试文件 test.cmm 进行词法分析，只需要在命令行输入：

./scanner test.cmm

，就可以得到你想要的结果了。

1.4.3 Flex：编写源代码

以上介绍的是使用 Flex 创建词法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是你所编写的 Flex 源代码，它完全决定了你的词法分析程序的一切行为。接下来我们介绍如何编写 Flex 源代码。

Flex 源代码文件包括三个部分，由“%%”隔开，如下所示：

```
1  {definitions}
2  %%
3  {rules}
4  %%
5  {user subroutines}
```

第一部分为**定义部分**，实际上就是给某些后面可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为：

name definition

其中 name 是名字，definition 是任意的正则表达式（正则表达式该如何书写后面会介绍）。例如，下面的这段代码定义了两个名字：digit 和 letter，前者代表 0 到 9 中的任意一个数字字符，后者则代表任意一个小写字母、大写字母或下划线：

```
1  …
2  digit [0-9]
3  letter [_a-zA-Z]
4  %%
5  …
6  %%
7  …
```

Flex 源代码文件的第二部分为**规则部分**，它由正则表达式和相应的响应函数组成，其格式为：

pattern {action}

其中 pattern 为正则表达式，其书写规则与前面的定义部分的正则表达式相同。而 action

则为将要进行的具体操作，这些操作可以用一段 C 代码表示。Flex 将按照这部分给出的内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex 默认只将其拷贝到标准输出，想要修改这个默认行为的话只需要在所有规则的最后加上一条“.”（即匹配任何输入）规则，然后在其对应的 action 部分书写你想要的行为即可。

例如，下面这段 Flex 代码在遇到输入文件中包含一串数字时，会将该数字串转化为整数值并打印到屏幕上：

```
1  …
2  digit [0-9]
3  %%
4  {digit}+ { printf("Integer value   %d\n", atoi(yytext)); }
5  …
6  %%
7  …
```

其中变量 yytext 的类型为 char*，它是 Flex 为我们提供的一个变量，里面保存了当前词法单元所对应的词素。函数 atoi()的作用是把一个字符串表示的整数转化为 int 类型。

Flex 源代码文件的第三部分为**用户自定义代码部分**。这部分代码会被原封不动地拷贝到 lex.yy.c 中，以方便用户自定义所需要执行的函数（之前我们提到过的 main 函数也可以写在这里）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在前面的定义部分（即 Flex 源代码文件的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会一并拷贝到 lex.yy.c 的最前面。

下面通过一个简单的例子来说明 Flex 源代码该如何书写 1。我们知道 Unix/Linux 下有一个常用的文字统计工具 wc，它可以统计一个或者多个文件中的（英文）字符数、单词数和行数。利用 Flex 我们可以快速地写出一个类似的文字统计程序：

```
1  %{
2      /*  此处省略#include 部分  */
3        int chars = 0;
4        int words = 0;
5        int lines = 0;
6  %}
7  letter [a-zA-Z]
8  %%
```

```
9   {letter}+ { words++; chars+= yyleng; }
10  \n { chars++; lines++; }
11  . { chars++; }
12  %%
13  int main(int argc, char** argv) {
14      if (argc > 1) {
15        if (!(yyin = fopen(argv[1], "r"))) {
16            perror(argv[1]);
17            return 1;
18        }
19      }
20      yylex();
21      printf("%8d%8d%8d\n", lines, words, chars);
22      return 0;
23  }
```

其中 yyleng 是 Flex 为我们提供的变量，你可以将其理解为 strlen(yytext)。我们用变量 chars 记录输入文件中的字符数、words 记录单词数、lines 记录行数。上面这段程序很好理解：每遇到一个换行符就把行数加一，每识别出一个单词就把单词数加一，每读入一个字符就把字符数加一。最后在 main 函数中把 chars、words 和 lines 的值全部打印出来。需要注意的是，由于在规则部分里我们没有让 yylex()返回任何值，因此在 main 函数中调用 yylex()时可以不套外层的 while 循环。

真正的 wc 工具可以一次传入多个参数从而统计多个文件。为了能够让这个 Flex 程序对多个文件进行统计，我们可以修改 main 函数如下：

```
1   int main(int argc, char** argv) {
2       int i, totchars = 0, totwords = 0, totlines = 0;
3       if (argc < 2) { /* just read stdin */
4           yylex();
5           printf("%8d%8d%8d\n", lines, words, chars);
6           return 0;
7       }
8       for (i = 1; i < argc; i++) {
```

```
9         FILE *f = fopen(argv[i], "r");
10        if (!f) {
11            perror(argv[i]);
12            return 1;
13        }
14        yyrestart(f);
15        yylex();
16        fclose(f);
17        printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
18        totchars += chars; chars = 0;
19        totwords += words; words = 0;
20        totlines += lines; lines = 0;
21    }
22    if (argc > 1)
23      printf("%8d%8d%8d total\n", totlines, totwords, totchars);
24    return 0;
25  }
```

其中 yyrestart(f)函数是 Flex 提供的库函数，它可以让 Flex 将其输入文件的文件指针 yyin 设置为 f（当然你也可以像前面一样手动设置令 yyin = f）并重新初始化该文件指针，令其指向输入文件的开头。

1.4.4 Flex：书写正则表达式

Flex 源代码中无论是定义部分还是规则部分，正则表达式都在其中扮演了重要的作用。那么，如何在 Flex 源代码中书写正则表达式呢？我门下面介绍一些规则：

1) 符号“.”匹配除换行符“\n”之外的任何一个字符。

2) 符号“[”和“]”共同匹配一个字符类，即方括号之内只要有一个字符被匹配上了，那么方括号括起来的整个表达式都被匹配上了。例如，[0123456789]表示 0~9 中任意一个数字字符，[abcABC]表示 a、b、c 三个字母的小写或者大写。方括号中还可以使用连字符“-”表示一个范围，例如[0123456789]也可以直接写作[0-9]，而所有小写字母字符也可直接写成[a-z] 。如果方括号中的第一个字符是“^”，则表示对这个字符类取补，即方括号之内如果没有任何一个字符被匹配上，那么被方括号括起来的整个表达式就认为被匹配上了。例如，[^_0-9a-zA-Z]表示所有的非字母、数字以及下划线的字符。

3) 符号“^”用在方括号之外则会匹配一行的开头，符号“$”用于匹配一行的结尾，符号“<<EOF>>”用于匹配文件的结尾。

4) 符号“{”和“}”含义比较特殊。如果花括号之内包含了一个或者两个数字，则代表花括号之前的那个表达式需要出现的次数。例如，A{5}会匹配 AAAAA，A{1,3}则会匹配 A、AA 或者 AAA。如果花括号之内是一个在 Flex 源代码的定义部分定义过的名字，则表示那个名字对应的正则表达式。例如，在定义部分如果定义了 letter 为[a-zA-Z]，则{letter}{1,3}表示连续的一至三个英文字母。

5) 符号 “*” 为 **Kleene 闭包**操作，匹配零个或者多个表达式。例如{letter}*表示零个或者多个英文字母。

6) 符号 “+” 为**正闭包**操作，匹配一个或者多个表达式。例如{letter}+表示一个或者多个英文字母。

7) 符号“?”匹配零个或者一个表达式。例如表达式-?[0-9]+表示前面带一个可选的负号的数字串。无论是*、+还是?，它们都只对其最邻近的那个字符生效。例如 abc+表示 ab 后面跟一个或多个 c，而不表示一个或者多个 abc。如果你要匹配后者，则需要使用小括号“(”和“)” 将这几个字符括起来：(abc)+。

8) 符号“|”为选择操作，匹配其之前或之后的任一表达式。例如，faith | hope | charity 表示这三个串中的任何一个。

9) 符号“\”用于表示各种转义字符，与 C 语言字符串里“\”的用法类似。例如，“\n”表示换行，“\t”表示制表符，“*” 表示星号，“\\” 代表字符 “\” 等。

10) 符号 “"”（英文引号）将逐字匹配被引起来的内容（即无视各种特殊符号及转义字符）。例如，表达式"..."就表示三个点而不表示三个除换行符以外的任意字符。

11) 符号 “/” 会查看输入字符的上下文，例如，x/y 识别 x 仅当在输入文件中 x 之后紧跟着 y，0/1 可以匹配输入串 01 中的 0 但不匹配输入串 02 中的 0。

12) 任何不属于上面介绍过的有特殊含义的字符在正则表达式中都仅匹配该字符本身。

下面我们通过几个例子来练习一下 Flex 源代码里正则表达式的书写：

1) 带一个可选的正号或者负号的数字串，可以这样写：[+-]?[0-9]+。

2) 带一个可选的正号或者负号以及一个可选的小数点的数字串，表示起来要困难一些，可以考虑下面几种写法：

a) [+-]?[0-9.]+会匹配太多额外的模式，像 1.2.3.4；

b) [+-]?[0-9]+\.?[0-9]+会漏掉某些模式，像 12.或者.12；

c) [+-]?[0-9]*\.?[0-9]+会漏掉 12.；

d) [+-]?[0-9]+\.?[0-9]*会漏掉.12；

e) [+-]?[0-9]*\.?[0-9]*会多匹配空串或者只有一个小数点的串；

f) 正确的写法是：[+-]?([0-9]*\.?[0-9]+|[0-9]+\.)。

3) 假设我们现在做一个汇编器，目标机器的 CPU 中有 32 个寄存器，编号为 0…31。汇编源代码可以使用 r 后面加一个或两个数字的方式来表示某一个寄存器，例如 r15 表示第 15 号寄存器，r0 或 r00 表示第 0 号寄存器，r7 或者 r07 表示第 7 号寄存器等。现在我们希望识别汇编源代码中所有可能的寄存器表示，可以考虑下面几种写法：

g) r[0-9]+可以匹配 r0 和 r15，但它也会匹配 r99999，目前世界上还不存在 CPU 能拥有一百万个寄存器。

h) r[0-9]{1,2}同样会匹配一些额外的表示，例如 r32 和 r48 等。

i) r([0-2][0-9]?|[4-9]|(3(0|1)?))是正确的写法，但其可读性比较差。

j) 正确性毋庸置疑并且可读性最好的写法应该是：r0 | r00 | r1 | r01 | r2 | r02 | r3 | r03 | r4 | r04 | r5 | r05 | r6 | r06 | r7 | r07 | r8 | r08 | r9 | r09 | r10 | r11 | r12 | r13 | r14 | r15 | r16 | r17 | r18 | r19 | r20 | r21 | r22 | r23 | r24 | r25 | r26 | r27 | r28 | r29 | r30 | r31，但这样写的话可扩展性又非常差，如果目标机器上有 128 甚 至 256 个寄存器呢？

1.4.5 Flex：高级特性

前面介绍的 Flex 内容已足够帮助完成实验一的词法分析部分。下面介绍一些 Flex 里面的高级特性，能让你在使用 Flex 的过程中感到更方便和灵活。这部分内容是可选的，跳过也不会对实验一的完成产生负面的影响。

yylineno 选项：

在写编译器程序的过程中，经常会需要记录行号，以便在报错时提示输入文件的哪一行出现了问题。为了能记录这个行号，我们可以自己定义某个变量，例如 lines，来记录词法分析程序当前读到了输入文件的哪一行。每当识别出“\n”，我们就让 lines = lines + 1。

实际上，Flex 内部已经为我们提供了类似的变量，叫做 yylineno。我们不必去维护 yylineno 的值，它会在每行结束自动加一。不过，默认状态下它并不开放给用户使用。如果我们想要读取 yylineno 的值，则需要在 Flex 源代码的定义部分加入语句“%option yylineno”。

需要说明的是，虽然 yylineno 会自动增加，但当我们在词法分析过程中调用 yyrestart() 函数读取另一个输入文件时它却不会重新被初始化，因此我们需要自行添加初始化语句 yylineno = 1。

输入缓冲区：

课本上介绍的词法分析程序其工作原理都是在模拟一个 DFA 的运行。这个 DFA 每次读

入一个字符，然后根据状态之间的转换关系决定下一步应该转换到哪个状态。事实上，实用的词法分析程序很少会从输入文件中逐个读入字符，因为这样需要进行大量的磁盘操作，效率较低。更加高效的办法是一次读入一大段输入字符，并将其保存在专门的输入缓冲区中。

在 Flex 中，所有的输入缓冲区都有一个共同的类型，叫做 YY_BUFFER_STATE。你可以通过 yy_create_buffer()函数为一个特定的输入文件开辟一块输入缓冲区，例如：

```
1  YY_BUFFER_STATE bp;
2  FILE* f;
3  f = fopen(…, "r");
4  bp = yy_create_buffer(f, YY_BUF_SIZE);
5  yy_switch_to_buffer(bp);
6  …
7  yy_flush_buffer(bp);
8  …
9  yy_delete_buffer(bp);
```

其中 YY_BUF_SIZE 是 Flex 内部的一个常数，通过调用 yy_switch_to_buffer()函数可以让词法分析程序到指定的输入缓冲区中读字符，调用 yy_flush_buffer()函数可以清空缓冲区中的内容，而调用 yy_delete_buffer()则可以删除一个缓冲区。

如果你的词法分析程序要支持文件与文件之间的相互引用（例如 C 语言中的#include），你可能需要在词法分析的过程中频繁地使用 yyrestart()切换当前的输入文件。在切换到其他输入文件再切换回来之后，为了能继续之前的词法分析任务，你需要无损地保留原先输入缓冲区的内容，这就需要使用一个栈来暂存当前输入文件的缓冲区。虽然 Flex 也有提供相关的函数来帮助你做这件事情，但这些函数的功能比较弱，建议自己手写更好。

Flex 库函数 input：

Flex 库函数 input()可以从当前的输入文件中读入一个字符，这有助于你不借助正则表达式来实现某些功能。例如，下面这段代码在输入文件中发现双斜线“//”后，将从当前字符开始一直到行尾的所有字符全部丢弃掉：

```
1  %%
2  "//" {
3  char c = input();
4  while (c != '\n') c = input();
5  }
```

Flex 库函数 unput：

Flex 库函数 unput(char c)可以将指定的字符放回输入缓冲区中，这对于宏定义等功能的实现是很方便的。例如，假设之前定义过一个宏#define BUFFER_LEN 1024，当在输入文件中遇到字符串 BUFFER_LEN 时，下面这段代码将该宏所对应的内容放回输入缓冲区：

```
1  char* p = macro_contents("BUFFER_LEN"); // p = “1024”
2  char* q = p + strlen(p);
3  while (q > p) unput(*--q); // push back right-to-left
```

Flex 库函数 yyless 和 yymore：

Flex 库函数 yyless(int n)可以将刚从输入缓冲区中读取的 yyleng-n 个字符放回到输入缓冲区中，而函数 yymore()可以告诉 Flex 保留当前词素，并在下一个词法单元被识别出来之后将下一个词素连接到当前词素的后面。配合使用 yyless()和 yymore()可以方便地处理那些边界难以界定的模式。例如，我们在为字符串常量书写正则表达式时，往往会写成由一对双引号引起来的所有内容\"[^\"]*\"，但有时候被双引号引起来的内容里面也可能出现跟在转义符号之后的双引号，例如"This is an \"example\""。那么如何使用 Flex 处理这种情况呢？方法之一就是借助于 yyless 和 yymore：

```
1  %%
2  \"[^\"]*\" {
3      if (yytext[yyleng - 2] == '\\') {
4          yyless(yyleng - 1);
5          yymore();
6      } else {
7          /* process the string literal */
8      }
9  }
```

Flex 宏 REJECT：

Flex 宏 REJECT 可以帮助我们识别那些互相重叠的模式。当我们执行 REJECT 之后，Flex 会进行一系列的操作，这些操作的结果相当于将 yytext 放回输入之内，然后去试图匹配当前规则之后的那些规则。例如，考虑下面这段 Flex 源代码：

```
1  %%
2  pink { npink++; REJECT; }
3  ink { nink++; REJECT; }
```

```
4    pin { npin++; REJECT; }
```

这段代码会统计输入文件中所有的 pink、ink 和 pin 出现的个数，即使这三个单词之间互有重叠。

Flex 还有更多的特性，感兴趣的读者可以参考其用户手册。

1.4.6 词法分析提示

如果你选择任务 1，为了完成实验一，首先需要阅读 C--语言文法（附录 A），包括其文法定义和补充说明。除了 INT、FLOAT 和 ID 这三个词法单元需要你自行为其书写正则表达式之外，剩下的词法单元都没有难度。

阅读完 C--语言文法，对 C--的词法有大概的了解之后，就可以开始编写 Flex 源代码了。在敲入所有的词法之后，为了能检验你的词法分析程序是否工作正常，你可以暂时向屏幕打印

当前的词法单元的名称，例如：

```
1    %%
2    "+" { printf(“PLUS\n”); }
3    "-" { printf(“SUB\n”); }
4    "&&" { printf(“AND\n”); }
5    "||" { printf(“OR\n”); }
6    …
```

为了能够报告错误类型 A，你可以在所有规则的最后增加类似于这样的一条规则：

```
1 %%
2 …
3    .    {
4             printf("Error type A at Line %d: Mysterious characters \'%s\'\n",
5                    yylineno, yytext);
6       }
```

完成 Flex 源代码的编写之后，使用前面介绍过的方法将其编译出来，就可以自己书写一些小规模的输入文件来测试你的词法分析程序了。一定要确保你的词法分析程序的正确性！如果词法分析这里出了问题没有检查出来，到了后面语法分析发现了前面的问题再回头调试，那将增加许多不必要的麻烦。为了使你在编写 Flex 源代码时少走弯路，以下是几条建议：

1) 留意空格和回车的使用。如果不注意，有时很容易让本应是空白符的空格或者回车

变成正则表达式的一部分，有时又很容易让本应是正则表达式一部分的空格或回车变成 Flex 源代码里的空白符。

2) 正则表达式和其所对应的动作之间，永远不要插入空行。

3) 如果对正则表达式中的运算符优先级有疑问，那就不要吝啬使用括号来确保正则表达式的优先级确实是你想要的。

4) 使用花括号括起每一段动作，即使该动作只包含有一行代码。

5) 在定义部分我们可以为许多正则表达式取别名，这一点要好好利用。别名可以让后面的正则表达式更加容易阅读、扩展和调试。

6) 在正则表达式中引用之前定义过的某个别名（例如 digit）时，时刻谨记该别名一定要用花括号“{”和“}”括起来。

实验 2 自顶向下的语法分析程序

2.1 实验目的

（1）熟练掌握 LL(1)分析表的构造方法。

（2）掌握设计、编制和调试典型的语法分析程序，进一步掌握常用的语法分析方法。

2.2 实验任务

根据 LL(1)分析法自己编写一个语法分析程序，语言不限，文法不限。

2.3 实验内容

2.3.1 实验要求

你的程序应具有通用性，能够识别由词法分析得出的词法单元序列是否是给定文法的正确句子（程序），并能够输出分析过程和识别结果。

2.3.2 输入格式

1、一个包含源代码的文本文件，此时需要和词法分析程序进行对接，通过一次扫描同时完成词法分析和语法分析；

2、通过实验一中借助 Flex 工具或自己编写的词法分析程序，得到相应的词法单元序列，然后将其作为此次语法分析程序的输入。

注：两种输入格式，二选其一即可。

2.3.3 输出格式

实验二要求通过标准输出打印程序的运行结果（包括 First 集、Follow 集、LL(1)分析表），此外，要求可以保存 LL(1)分析表。

你的程序需要输出语法分析过程和相应的分析结果（即此串是否为 LL(1)文法的句子）。

2.3.4 提交要求

1) 可以是 Java、C、Python 等任意语言编写的可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

2.3.5 样例

此部分输出样例可参考课本例题中对输入串的分析过程表（教材 P94-95 的表 4.4 和表 4.5）。

2.4 实验指导

词法分析的下一阶段是语法分析，语法分析是编译原理的核心部分，其在编译器模型中的位置如图 2-1 所示。语法分析的作用是识别从词法分析器获得的一个由词法单元序列组成的串是否是给定文法的正确句子。目前语法分析常用的方法有自顶向下分析和自底向上分析

两大类。此次实验采用自顶向下的 LL(1)分析方法。

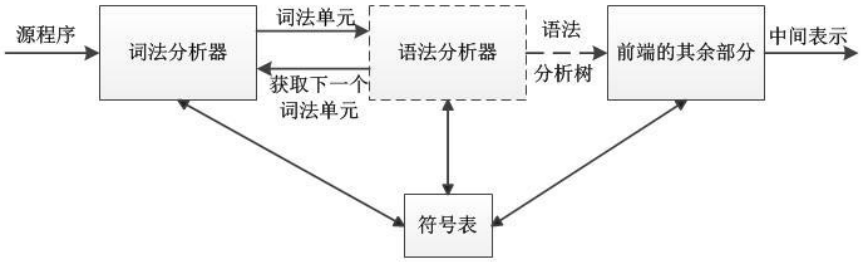


图 2-1 编译器模型中语法分析器的位置

2.4.1 LL(1)分析法概述

所谓 LL(1)分析法，就是指从左到右扫描输入串（源程序），同时采用最左推导，且对每次直接推导只需向前看一个输入符号，便可确定当前所应当选择的规则。实现 LL(1)分析的程序又称为 LL(1)分析程序或 LL(1)分析器。

一个文法要能进行 LL(1)分析，那么这个文法应该满足：无二义性，无左递归，无左公因子。LL(1)的语法分析程序包含了三个部分，总控程序，预测分析表函数，先进先出的语法分析栈。

2.4.2 预测分析程序

LL(1)预测分析程序的总控程序在任何时候都是按 STACK 栈顶符号 X 和当前输入符号 a 来决定做何种操作的。其具体工作过程可用示意图 2-2 表示。其中：“#”句子括号即输入串的括号；“S”文法的开始符号；“X”存放当前栈顶符号的工作单元；“a”存放当前输入符号 a 的工作单元。

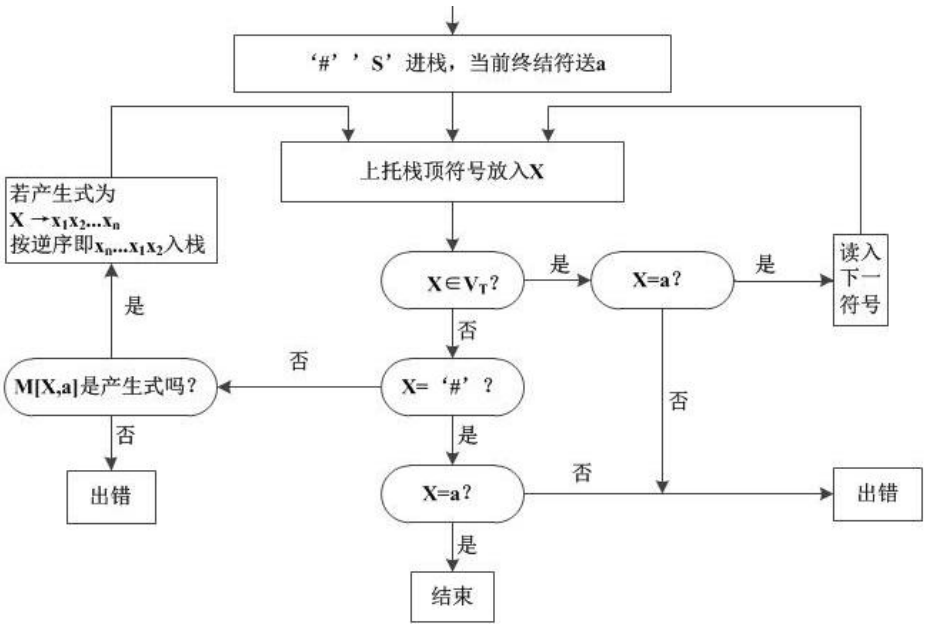


图 2-2 预测分析程序的框图

从图 2-2 可以看出，对于任何（X，a），总控程序每次都执行下述三种可能的动作之一：

1）若 X = a = ‘#’，则宣布分析成功，停止分析过程。

2）若 X = a≠ #’，则把 X 从 STACK 栈顶弹出，让 a 指向下一个输入符号。

3）若 X 是一个非终结符，则查看预测分析表 M。若 M[A，a]中存放着关于 X 的一个产生式，那么，首先把 X 弹出 STACK 栈顶，然后，把产生式的右部符号串按反序一一弹出 STACK 栈（若右部符号为ε，则不推什么东西进 STACK 栈）。若 M[A，a]中存放着“出错标志”，则调用出错诊断程序 ERROR。

2.4.3 构造 LL(1)分析表

对于此次语法分析程序中用到的分析表，可根据自己实际情况，选择以下一项实现分析算法中分析表的构造：

1）直接输入根据已知文法构造的分析表 M;

2）输入文法的 FIRST(α)和 FOLLOW(U)集合，由程序自动生成文法的分析表 M；

3）输入已知文法，由程序自动构造文法的分析表 M。

实验 3 基于 LR(0)方法的语法分析

3.1 实验目的

（1）掌握 LR(0)分析表的构造方法。

（2）掌握设计、编制和调试典型的语法分析程序，进一步掌握常用的语法分析方法。

（3）理解语法分析在编译程序中的作用。

3.2 实验任务

编写一个程序对输入的源代码进行语法分析，并打印分析结果。（以下两个任务二选一）

1、借助语法分析工具 GNU Bison，编写一个对使用 C--语言书写的源代码进行语法分析（C--语言的文法参见附录 A），并使用 C 语言完成。

2、自己编写一个基于 LR(0)方法的语法分析程序。语言不限，文法不限。

此时可根据自己的实际情况，选择以下一项实现分析算法中分析表的构造：

（1）直接输入根据已知文法构造的 LR(0)分析表；

（2）输入已知文法的项目集规范族和转换函数，由程序自动生成 LR(0)分析表；

（3）输入已知文法，由程序自动生成 LR(0)分析表。

3.3 实验内容

3.3.1 实验要求

任务 1：你的程序要能够查出源代码中可能包含的语法错误：语法错误（**错误类型 B**）。

任务 2：你的程序应具有通用性，能够识别由词法分析得出的词法单元序列是否是给定文法的正确句子（程序），并能够输出分析过程和识别结果。

3.3.2 输入格式

1、如果你选择的是**任务 1**，那么你的程序的输入是一个包含 C--源代码的文本文件，程序需要能够接收一个输入文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 即可获得以 test1 作为输入文件的输出结果。

2、如果你选择的是**任务 2**，那么你的程序输入是一个包含待分析词法单元序列的文本文件，程序需要能够接收一个输入文件名作为参数，以获得相应的输出结果。

3.3.3 输出格式

实验三要求通过标准输出打印程序的运行结果。

任务 1：对于那些包含语法错误的输入文件，只要输出相关的语法有误的信息即可。在这种情况下，注意不要输出任何与语法树有关的内容。要求输出的信息包括错误类型、出错的行号以及说明文字，

其格式为：Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类（**即语法错误为错误类型 B**），否则将影响你的实验评分。**注意，**输入文件中可能会包含一个或者多个语法错误（但输入文件的同一行中保证不出现多个错误），你的程序需要将这些错误全部报告出来，每一条错误提示信息在输出中单独占一行。

对于那些没有任何语法错误的输入文件，则需要输出相应的语法分析结果。

你的程序需要将构造好的语法树按照先序遍历的方式打印每一个结点的信息，这些信息包括：

1） 如果当前结点是一个语法单元并且该语法单元没有产生ε（即空串），则打印该语法单元的名称以及它在输入文件中的行号（行号被括号所包围，并且与语法单元名之间有一个空格）。所谓某个语法单元在输入文件中的行号是指该语法单元产生出的所有词素中的第一个在输入文件中出现的行号。

2） 如果当前结点是一个语法单元并且该语法单元产生了ε，则无需打印该语法单元的信息。

3） 如果当前结点是一个词法单元，则只要打印该词法单元的名称，而无需打印该词法单元的行号。

a) 如果当前结点是词法单元 ID，则要求额外打印该标识符所对应的词素；

b) 如果当前结点是词法单元 TYPE，则要求额外打印说明该类型为 int 还是 float；

c) 如果当前结点是词法单元 INT 或者 FLOAT，则要求以十进制的形式额外打印该数字所对应的数值；

d) 词法单元所额外打印的信息与词法单元名之间以一个冒号和一个空格隔开。

每一条词法或语法单元的信息单独占一行，而每个子结点的信息相对于其父结点的信息来说，在行首都要求缩进 2 个空格。具体输出格式可参见后续的样例。

任务 2：你的程序需要输出语法分析过程（包括 **LR(0)分析表**和**分析过程表**，并能够保存 LR(0)分析表）和相应的分析结果（即此串是否为 LR(0)文法的句子）。

3.3.4 测试环境

任务 1：你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

4) GNU Bison version 2.5.

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实验三的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

3.3.5 提交要求

1) 任务 1：以 Flex、Bison 以及 C 语言编写的可被正确编译运行的源程序。

任务 2：可以是 Java、C、Python 等任意语言编写的可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

3.3.6 样例

此部分样例是以**任务 1** 为例给出的，请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

说明：选择任务 2 的同学，可以参考教材 P125 页中表 6.1 和表 6.2 相关内容，自行设计测试样例，以反映出正确输出。

样例 1：

输入（行号是为标识需要，并非样例输入的一部分，后同）：

```
1  int main()
2  {
3      float a[10][2];
4      int i;
5      a[5,3] = 1.5;
6      if (a[1][2] == 0)   i = 1 else i = 0;
7  }
```

输出：

这个程序存在两处语法错误。其一，虽然我们的程序中允许出现方括号与逗号等字符，但二维数组正确的访问方式应该是 a[5][3]而非 a[5,3]。其二，第 6 行的 if-else 语句在 else 之前少了一个分号。因此你的程序可以输出如下的两行错误提示信息：

Error type B at Line 5: Missing "]".

Error type B at Line 6: Missing ";".

样例 2：

输入：

```
1  int inc()
2  {
3      int    i;
4      i = i + 1;
5  }
```

输出：

这个程序非常简单，也没有任何词法或语法错误，因此你的程序需要输出如下的语法树结点信息（左侧行号是为标识需要，并非程序输出的一部分）：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: inc
8          LP
9          RP
10       CompSt (2)
11         LC
12         DefList (3)
13           Def (3)
14             Specifier (3)
15               TYPE: int
16             DecList (3)
17               Dec (3)
18                 VarDec (3)
19                   ID: i
20             SEMI
21         StmtList (4)
22           Stmt (4)
23             Exp (4)
24               Exp (4)
25                 ID: i
26               ASSIGNOP
27               Exp (4)
28                 Exp (4)
```

```
29                      ID: i
30                  PLUS
31                  Exp (4)
32                      INT: 1
33              SEMI
34          RC
```

3.4 实验指导

词法分析和语法分析这两块，可以说是在整个编译器当中被自动化得最好的部分。也就是说，即使没有任何的理论基础，在掌握了工具的用法之后，也可以在短时间内做出功能很全很棒的词法分析程序和语法分析程序。当然这并不意味着，词法分析和语法分析部分的理论基础并不重要。恰恰相反，这一部分被认为是计算机理论在工程实践中最成功的应用之一，对它的介绍也是编译理论课中的重点。但本节指导内容的重点不在于理论而在于工具的使用。

本节指导内容将主要介绍语法分析工具 GNU Bison。如前所述，完成实验三并不需要太多的理论基础，只要看完并掌握了本节的大部分内容即可完成实验三。

3.4.1 语法分析概述

语法分析是词法分析的下一阶段。语法分析程序的主要任务是读入词法单元流、判断输入程序是否匹配程序设计语言的语法规范，并在匹配规范的情况下构建起输入程序的静态结构。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。

语法分析与词法分析有很多相似之处：它们的基础都是形式语言理论，它们都是计算机理论在工程实践中最成功的应用，它们都能被高效地完成，它们的构建都可以被工具自动化完成。目前绝大多数实用的编译器在语法分析这里都是使用工具帮助完成的。

正则表达式难以进行任意大的计数，所以很多在程序设计语言中常见的结构（例如匹配的括号）无法使用正则文法进行表示。为了能够有效地对常见的语法结构进行表示，人们使用了比正则文法表达能力更强的上下文无关文法（Context Free Grammar 或 CFG）。然而，虽然上下文无关文法在表达能力上要强于正则语言，但在判断某个输入串是否属于特定 CFG 的问题上，时间效率最好的算法也要 O(n3)，这样的效率让人难以接受。因此，现代程序设计语言的语法大多属于一般 CFG 的一个足够大的子集，比较常见的子集有 LL(k)文法以及 LR(k)文法。判断一个输入是否属于这两种文法都只需要线性时间。

上下文无关文法 G 在形式上是一个四元组：终结符号（也就是词法单元）集合 T、非终结符号集合 NT、初始符号 S 以及产生式集合 P。产生式集合 P 是一个文法的核心，它通

过产生式定义了一系列的推导规则，从初始符号出发，基于这些产生式，经过不断地将非终结符替换为其它非终结符以及终结符，即可得到一串符合语法规约的词法单元。这个替换和推导的过程可以使用树形结构表示，称作**语法树**。事实上，语法分析的过程就是把词法单元流变成语法树的过程。尽管在之前曾经出现过各式各样的算法，但目前最常见的构建语法树的技术只有两种：自顶向下方法和自底向上方法。我们下面将要介绍的工具 Bison 所生成的语法分析程序就采用了自底向上的 LALR(1)分析技术（通过一定的设置还可以让 Bison 使用另一种被称为 GLR 的分析技术，不过对该技术的介绍已经超出了我们讨论的范围）。而其它的某些语法分析工具，例如基于 Java 语言的 JTB 生成的语法分析程序，则是采用了自顶向下的 LL(1)分析技术。当然，具体的工具采用哪一种技术这种细节，对于工具的使用者来讲都是完全屏蔽的。和词法分析程序的生成工具一样，工具的使用者所要做的仅仅是将输入程序的程序设计语言的语法告诉语法分析程序生成工具，虽然工具本身不能帮助直接构造出语法树，但我们可以通过在语法产生式中插入语义动作这种更加灵活的形式，来实现一些甚至比构造语法树更加复杂的功能。

3.4.2 GNU Bison 介绍

Bison 的前身为基于 Unix 的 Yacc。令人惊讶的是，Yacc 的发布时间甚至比 Lex 还要早。Yacc 所采用的 LR 分析技术的理论基础早在 50 年代就已经由 Knuth 逐步建立了起来，而 Yacc 本身则是贝尔实验室的 S.C. Johnson 基于这些理论在 75 年到 78 年写成的。到了 1985 年，当时在 UC Berkeley 的一个研究生 Bob Corbett 在 BSD 下重写了 Yacc，后来 GNU Project 接管了这个项目，为其增加了许多新的特性，于是就有了我们今天所用的 GNU Bison。

GNU Bison 在 Linux 下的安装非常简单，你可以去它的官方网站上下载安装包自行安装，基于 Debian 的 Linux 系统下更简单的方法同样是直接在命令行敲入如下命令：

```
sudo apt-get install bison
```

虽说版本不一样，但 GNU Bison 的基本使用方法和课本上所介绍的 Yacc 没有什么不同。首先，我们需要自行完成包括语法规则等在内的 Bison 源代码。如何编写这份代码后面会提到，现在先假设这份写好的代码名 syntax.y。随后，我们使用 Bison 对这份代码进行编译：

```
bison syntax.y
```

编译好的结果会保存在当前目录下的 syntax.yy.c 文件中。打开这个文件你就会发现，该文件本质上就是一份 C 语言的源代码。事实上，这份源代码里目前对我们有用的函数只有一个，叫做 yyparse()，该函数的作用就是对输入文件进行语法分析，如果分析成功没有错误则返回 0，否则返回非 0。不过，只有这个 yyparse()函数还不足以让我们的程序跑起来。前面说过，语法分析程序的输入是一个个的词法单元，那么 Bison 通过什么方式来获得这些词

法单元呢？事实上，Bison 在这里需要用户为它提供另外一个专门返回词法单元的函数，这个函数的名字正是 yylex()。

函数 yylex()相当于嵌在 Bison 里的词法分析程序。这个函数可以由用户自行实现，但因为我们之前已经使用 Flex 生成了一个 yylex()函数，能不能让 Bison 使用 Flex 生成的 yylex()函数呢？答案是肯定的。

仍以 Bison 源代码文件 syntax.y 为例。首先，为了能够使用 Flex 中的各种函数，需要在 Bison 源代码中引用 lex.yy.c：

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

随后在使用 Bison 编译这份源代码时，我们需要加上“-d”参数：

```
bison -d syntax.y
```

这个参数的含义是，将编译的结果分拆成 syntax.tab.c 和 syntax.tab.h 两个文件，其中.h 文件里包含着一些词法单元的类型定义之类的内容。得到这个.h 文件之后，下一步是修改我们的 Flex 源代码 lexical.l，增加对 syntax.tab.h 的引用，并且让 Flex 源代码中规则部分的每一条 action 都返回相应的词法单元，如下所示：

```
1  %{
2      #include "syntax.tab.h"
3      …
4  %}
5  …
6  %%
7  "+" { return PLUS; }
8  "-" { return SUB; }
9  "&&" { return AND; }
10 "||" { return OR; }
11 …
```

其中，返回值 PLUS 和 SUB 等都是在 Bison 源代码中定义过的词法单元（如何定义它们后文会提到）。由于我们刚刚修改了 lexical.l，需要重新将它编译出来：

```
flex lexical.l
```

接下来是重写我们的 main 函数。由于 Bison 会在需要时自动调用 yylex()，我们在 main 函数中也就不需要调用它了。不过，Bison 是不会自己调用 yyparse()和 yyrestart()的，因此这两个函数仍需要我们在 main 函数中显式地进行调用：

```
1  int main(int argc, char** argv)
2  {
3      if (argc <= 1) return 1;
```

```
    FILE* f = fopen(argv[1], "r");
    if (!f)
    {
        perror(argv[1]);
        return 1;
    }
    yyrestart(f);
    yyparse();
    return 0;
}
```

现在我们有了三个 C 语言源代码文件：main.c、lex.yy.c 以及 syntax.tab.c，其中 lex.yy.c 已经被 syntax.tab.c 引用了，因此我们最后要做的就是把 main.c 和 syntax.tab.c 放到一起进行编译：

gcc main.c syntax.tab.c -lfl -ly -o parser

其中“-lfl”不要省略，否则 GCC 会因缺少库函数而报错，但“-ly”这里一般情况下可以省略。现在我们可以使用这个 parser 程序进行语法分析了。例如，想要对一个输入文件 test.cmm 进行语法分析，只需要在命令行输入：

./parser test.cmm

就可以得到你想要的结果。

3.4.3 Bison：编写源代码

我们前面介绍了使用 Flex 和 Bison 联合创建语法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是你所编写的 Flex 源代码和 Bison 源代码文件，它们完全决定了所生成的语法分析程序的一切行为。Flex 源代码如何进行编写前面已经介绍过了，接下来我们介绍如何编写 Bison 源代码。

同 Flex 源代码类似，Bison 源代码也分为三个部分，其作用与 Flex 源代码大致相同。第一部分是定义部分，所有词法单元的定义都可以放到这里；第二部分是规则部分，其中包括具体的语法和相应的语义动作；第三部分是用户函数部分，这部分的源代码会被原封不动地拷贝到 syntax.tab.c 中，以方便用户自定义所需要的函数（main 函数也可以写在这里，不过不推荐这么做）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在定义部分（也就是 Bison 源代码的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会被一并拷贝到 syntax.tab.c 的最前面。

下面我们通过一个例子来对 Bison 源代码的结构进行解释。一个在控制台运行可以进行

整数四则运算的小程序，其语法如下所示（这里假设词法单元 INT 代表 Flex 识别出来的一个整数，ADD 代表加号+，SUB 代表减号 ，MUL 代表乘号*，DIV 代表除号/）：

```
Calc → ε
  | Exp
Exp → Factor
  | Exp ADD Factor
  | Exp SUB Factor
Factor → Term
  | Factor MUL Term
  | Factor DIV Term
Term → INT
```

这个小程序的 Bison 源代码为：

```
%{
    #include <stdio.h>
%}

/* declared tokens */
%token INT
%token ADD SUB MUL DIV

%%
Calc : /* empty */
   | Exp { printf("= %d\n", $1); }
   ;
Exp : Factor
   | Exp ADD Factor { $$ = $1 + $3; }
   | Exp SUB Factor { $$ = $1 - $3; }
   ;
Factor : Term
   | Factor MUL Term { $$ = $1 * $3; }
   | Factor DIV Term { $$ = $1 / $3; }
   ;
Term : INT
   ;
 %%
#include "lex.yy.c"
int main() {
   yyparse();
```

```
27  }
28  yyerror(char* msg) {
29      fprintf(stderr, "error: %s\n", msg);
30  }
```

这段 Bison 源代码以“%{”和“%}”开头，被“%{”和“%}”包含的内容主要是对 stdio.h 的引用。接下来是一些以%token 开头的词法单元（终结符）定义，如果你需要采用 Flex 生成的 yylex()的话，那么在这里定义的词法单元都可以作为 Flex 源代码里的返回值。与终结符相对的，所有未被定义为%token 的符号都会被看作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次。

第二部分是书写产生式的地方。第一个产生式左边的非终结符默认为初始符号（你也可以通过在定义部分添加%start X 来将另外的某个非终结符 X 指定为初始符号）。产生式里的箭头在这里用冒号“:”表示，一组产生式与另一组之间以分号“;”隔开。产生式里无论是终结符还是非终结符都各自对应一个属性值，产生式左边的非终结符对应的属性值用$$表示，右边的几个符号的属性值按从左到右的顺序依次对应为$1、$2、$3 等。每条产生式的最后可以添加一组以花括号“{”和“}”括起来的语义动作，这组语义动作会在整条产生式归约完成之后执行，如果不明确指定语义动作，那么 Bison 将采用默认的语义动作{ $$ = $1 }。语义动作也可以放在产生式的中间，例如 A→B { … } C，这样的写法等价于 A→BMC，M→ε{ … }，其中 M 为额外引入的一个非终结符。需要注意的是，在产生式中间添加语义动作在某些情况下有可能会在原有语法中引入冲突，因此使用的时候要特别谨慎。

在这里你可能有疑问：每一个非终结符的属性值都可以通过它所产生的那些终结符或者非终结符的属性值计算出来，但是终结符本身的属性值该如何得到呢？答案是：在 yylex()函数中得到。因为我们的 yylex()函数是由 Flex 源代码生成的，因此要想让终结符带有属性值，就必须回头修改 Flex 源代码。假设在我们的 Flex 源代码中，INT 词法单元对应着一个数字串，那么我们可以将 Flex 源代码修改为：

```
1   …
2   digit [0-9]
3   %%
4   {digit}* {
5           yylval = atoi(yytext);
6           return INT;
7       }
8   …
9   %%
10  …
```

其中 yylval 是 Flex 的内部变量，表示当前词法单元所对应的属性值。我们只需将该变量的 值赋成 atoi(yytext)，就可以将词法单元 INT 的属性值设置为它所对应的整数值了。

回到之前的 Bison 源代码中。在用户自定义函数部分我们写了两个函数：一个很简单的只调用了 yyparse()的 main 函数以及另一个没有返回类型并带有一个字符串参数的 yyerror()的函数。yyerror()函数会在你的语法分析程序每发现一个语法错误时被调用，其默认参数为“syntax error”。默认情况下 yyerror()只会将传入的字符串参数打印到标准错误输出上，而你也可以自己重新定义这个函数，从而使它打印一些别的内容，例如上例中我们就在该参数前面多打印了“error: ”的字样。

现在，编译并执行这个程序，然后在控制台输入 10-2+3，然后输入回车，最后输入 Ctrl+D 结束，你会看到屏幕上打印出了计算结果 11。

3.4.4 Bison：属性值的类型

在上面的例子中，每个终结符或非终结符的属性值都是 int 类型。但在我们构建语法树的过程中，我们希望不同的符号对应的属性值能有不同的类型，而且最好能对应任意的类型而不仅仅是 int 类型。下面我们介绍如何在 Bison 中解决这个问题。

第一种方法是对宏 YYSTYPE 进行重定义。Bison 里会默认所有属性值的类型以及变量 yylval 的类型都是 YYSTYPE，默认情况下 YYSTYPE 被定义为 int。如果你在 Bison 源代码的“%{”和“%}”之间加入#define YYSTYPE float，那么所有的属性值就都成为了 float 类型。那么如何使不同的符号对应不同的类型呢？你可以将 YYSTYPE 定义成一个联合体类型，这样你就可以根据符号的不同来访问联合体中不同的域，从而实现多种类型的效果。

这种方法虽然可行，但在实际操作中还是稍显麻烦，因为你每次对属性值的访问都要自行指定哪个符号对应哪个域。实际上，在 Bison 中已经内置了其它的机制来方便你对属性值类型的处理，一般而言我们还是更推荐使用这种方法而不是上面介绍的那种。

我们仍然还是以前面的四则运算小程序为例，来说明 Bison 中的属性值类型机制是如何工作的。原先这个四则运算程序只能计算整数值，现在我们加入浮点数运算的功能。修改后的语法如下所示：

Calc → ε
  | Exp
Exp → Factor
  | Exp ADD Factor
  | Exp SUB Factor
Factor → Term
  | Factor MUL Term
  | Factor DIV Term

Term → INT

| FLOAT

在这份语法中，我们希望词法单元 INT 能有整型属性值，而 FLOAT 能有浮点型属性值，其他的非终结符为了简单起见，我们让它们都具有双精度型的属性值。这份语法以及类型方案对应的 Bison 源代码如下：

```
%{
    #include <stdio.h>
%}

/* declared types */
%union {
  int type_int;
  float type_float;
  double type_double;
}

/* declared tokens */
%token <type_int> INT
%token <type_float> FLOAT
%token ADD SUB MUL DIV

/* declared non-terminals */
%type <type_double> Exp Factor Term

%%
Calc : /* empty */
  | Exp { printf("= %lf\n, $1); }
  ;
Exp : Factor
  | Exp ADD Factor { $$ = $1 + $3; }
  | Exp SUB Factor { $$ = $1 - $3; }
  ;
Factor : Term
  | Factor MUL Term { $$ = $1 * $3; }
  | Factor DIV Term { $$ = $1 / $3; }
  ;
Term : INT { $$ = $1; }
```

```
33      | FLOAT { $$ = $1; }
34      ;
35
36  %%
37  …
```

首先，我们在定义部分的开头使用%union{…}将所有可能的类型都包含进去。接下来，在%token 部分我们使用一对尖括号<>把需要确定属性值类型的每个词法单元所对应的类型括起来。对于那些需要指定其属性值类型的非终结符而言，我们使用%type 加上尖括号的办法确定它们的类型。当所有需要确定类型的符号的类型都被定下来之后，规则部分里的$$、$1 等就自动地带有了相应的类型，不再需要我们显示地为其指定类型了。

3.4.5 Bison：语法单元的位置

实验要求中需要你输出每一个语法单元出现的位置。你当然可以自己在 Flex 中定义每个行号和列号以及在每个动作中维护这个行号和这个列号，并将它们作为属性值的一部分返回给语法单元。这种做法需要我们额外编写一些维护性的代码，非常不方便。Bison 有没有内置的位置信息供我们使用呢？答案是肯定的。

前面介绍过 Bison 中每个语法单元都对应了一个属性值，在语义动作中这些属性值可以使用$$、$1、$2 等进行引用。实际上除了属性值之外，每个语法单元还对应了一个位置信息，在语义动作中这些位置信息同样可以使用@$、@1、@2 等进行引用。位置信息的数据类型是一个 YYLTYPE，其默认的定义是：

```
1  typedef struct YYLTYPE {
2      int first_line;
3      int first_column;
4      int last_line;
5      int last_column;
6  }
```

其中的 first_line 和 first_column 分别是该语法单元对应的第一个词素出现的行号和列号，而 last_line 和 last_column 分别是该语法单元对应的最后一个词素出现的行号和列号。有了这些内容，输出所需的位置信息就比较方便了。但注意，如果直接引用@1、@2 等将每个语法单元的 first_line 打印出来，你会发现打印出来的行号全都是 1。

为什么会出现这种问题？主要原因在于，Bison 并不会主动替我们维护这些位置信息，我们需要在 Flex 源代码文件中自行维护。不过只要稍加利用 Flex 中的某些机制，维护这些信息并不需要太多的代码量。我们可以在 Flex 源文件的开头部分定义变量 yycolumn，并添加如下的宏定义 YY_USER_ACTION：

```
1  %locations
2  …
3  %{
4      /*  此处省略#include 部分  */
5      int yycolumn = 1;
6      #define YY_USER_ACTION \
7          yylloc.first_line = yylloc.last_line = yylineno; \
8          yylloc.first_column = yycolumn; \
9          yylloc.last_column = yycolumn + yyleng - 1; \
10         yycolumn += yyleng;
11 %}
```

其中 yylloc 是 Flex 的内置变量，表示当前词法单元所对应的位置信息；YY_USER_ACTION 宏表示在执行每一个动作之前需要先被执行的一段代码，默认为空，而这里我们将其改成了对位置信息的维护代码。除此之外，最后还要在 Flex 源代码文件中做的更改，就是在发现了换行符之后对变量 yycolumn 进行复位：

```
1  …
2  %%
3  …
4  \n { yycolumn = 1; }
```

这样就可以实现在 Bison 中正常打印位置信息。

3.4.6 Bison：二义性与冲突处理

Bison 有一个非常好用但也很恼人的特性：对于一个有二义性的文法，它有自己的一套隐式的冲突解决方案（一旦出现归约/归约冲突，Bison 总会选择靠前的产生式；一旦出现移入/ 归约冲突，Bison 总会选择移入）从而生成相应的语法分析程序，而这些冲突解决方案在某些场合可能并不是我们所期望的。因此，我们建议在使用 Bison 编译源代码时要留意它所给的提示信息，如果提示文法有冲突，那么请一定对源代码进行修改，尽量把所有冲突全部消解掉。

前面那个四则运算的小程序，如果它的语法变成这样：

```
Calc → ε
  |  Exp
Exp → Factor
  |  Exp ADD Exp
```

| Exp SUB Exp

…

虽然看起来好像没什么变化（Exp→Exp ADD Factor | Exp SUB Factor 变成了 Exp→Exp ADD Exp | Exp SUB Exp），但实际上前面之所以没有这样写，是因为这样做会引入二义性。例如，如果输入为 1 - 2 + 3，语法分析程序将无法确定先算 1 - 2 还是 2 + 3。语法分析程序在读到 1 - 2 的时候可以归约（即先算 1 - 2）也可以移入（即先算 2 + 3），但由于 Bison 默认移入优先于归约，语法分析程序会继续读入+ 3 然后计算 2 + 3。

为了解决这里出现的二义性问题，要么重写语法（Exp→ Exp ADD Factor | Exp SUB Factor 相当于规定加减法为左结合），要么显式地指定算符的优先级与结合性。一般而言，重写语法是一件比较麻烦的事情，而且会引入不少像 Exp→Term 这样除了增加可读性之外没任何什么实质用途的产生式。所以更好的解决办法还是考虑优先级与结合性。

在 Bison 源代码中，我们可以通过“%left”、“%right”和“%nonassoc”对终结符的结合性进行规定，其中“%left”表示左结合，“%right”表示右结合，而“%nonassoc”表示不可结合。例如，下面这段结合性的声明代码主要针对四则运算、括号以及赋值号：

```
1  %right ASSIGN
2  %left ADD SUB
3  %left MUL DIV
4  %left LP RP
```

其中 ASSIGN 表示赋值号，LP 表示左括号，RP 表示右括号。此外，Bison 也规定任何排在后面的算符其优先级都要高于排在前面的算符。因此，这段代码实际上还规定括号优先级高于乘除、乘除高于加减、加减高于赋值号。在实验一所使用的 C--语言里，表达式 Exp 的语法便是冲突的，你需要模仿前面介绍的方法，根据 C--语言的文法补充说明中的内容为运算符规定优先级和结合性，从而解决掉这些冲突。

另外一个在程序设计语言中很常见的冲突就是嵌套 if-else 所出现的冲突（也被称为悬空 else 问题）。考虑 C--语言的这段语法：

```
Stmt → IF LP Exp RP Stmt
     | IF LP Exp RP Stmt ELSE Stmt
```

假设我们的输入是：if (x > 0) if (x == 0) y = 0; else y = 1;，那么语句最后的这个 else 是属于前一个 if 还是后一个 if 呢？标准 C 语言规定在这种情况下 else 总是匹配距离它最近的那个 if，这与 Bison 的默认处理方式（移入/归约冲突时总是移入）是一致的。因此即使我们不在 Bison 源代码里对这个问题进行任何处理，最后生成的语法分析程序的行为也是正确的。

但如果不处理，Bison 总是会提示我们该语法中存在一个移入/归约冲突。有没有办法把这个冲突去掉呢？

显式地解决悬空 else 问题可以借助于算符的优先级。Bison 源代码中每一条产生式后面都可以紧跟一个%prec 标记，指明该产生式的优先级等同于一个终结符。下面这段代码通过定义一个比 ELSE 优先级更低的 LOWER_THAN_ELSE 算符，降低了归约相对于移入 ELSE 的优先级：

```
1  …
2  %nonassoc LOWER_THAN_ELSE
3  %nonassoc ELSE
4  …
5  %%
6  …
7  Stmt : IF LP Exp RP Stmt %prec LOWER_THAN_ELSE
8       | IF LP Exp RP Stmt ELSE Stmt
```

这里 ELSE 和 LOWER_THAN_ELSE 的结合性其实并不重要，重要的是当语法分析程序读到 IF LP Exp RP 时，如果它面临归约和移入 ELSE 这两种选择，它会根据优先级自动选择移入 ELSE。通过指定优先级的办法，我们可以避免 Bison 在这里报告冲突。

前面我们通过优先级和结合性解决了表达式和 if-else 语句里可能出现的二义性问题。事实上，有了优先级和结合性的帮助，我们几乎可以消除语法中所有的二义性，但我们不建议使用它们解决除了表达式和 if-else 之外的任何其它冲突。原因很简单：只要是 Bison 报告的冲突，都有可能成为语法中潜在的一个缺陷，这个缺陷的来源很可能是你所定义的程序设计语言里的一些连你自己都没有意识到的语法问题。表达式和二义性这里，我们之所以敢使用优先级和结合性的方法来解决，是因为我们对冲突的来源非常了解，除此之外，只要是 Bison 认为有二义性的语法，大部分情况下这个语法也能看出二义性。此时你要做的不是掩盖这些语法上的问题，而是仔细对语法进行修订，发现并解决语法本身的问题。

3.4.7 Bison：源代码的调试

以下这部分内容是可选的，跳过不会对实验三的完成产生负面影响。

在使用 Bison 进行编译时，如果增加-v 参数，那么 Bison 会在生成.yy.c 文件的同时帮我们多生成一个.output 文件。例如，执行

```
bison -d -v syntax.y
```

命令后，你会在当前目录下发现一个新文件 syntax.output，这个文件中包含 Bison 所生

成的语法分析程序对应的 LALR 状态机的一些详尽描述。如果你在使用 Bison 编译的过程中发现自己的语法里存在冲突，但无法确定在何处，就可以阅读这个.output 文件，里面对于每一个状态所对应的产生式、该状态何时进行移入何时进行归约、你的语法有多少冲突以及这些冲突在哪里等等都有十分完整的描述。

例如，如果我们不处理前面提到的悬空 else 问题，.output 文件的第一句就会是：

```
1  state 112 conflicts: 1 shift/reduce
```

继续向下翻，找到状态 112，.output 文件对该状态的描述为：

```
1  State 112
2
3  36 Stmt : IF LP Exp RP Stmt .
4  37 | IF LP Exp RP Stmt . ELSE Stmt
5
6  ELSE shift, and go to state 114
7
8  ELSE [reduce using rule 36 (Stmt)]
9  $default reduce using rule 36 (Stmt)
```

这里我们发现，状态 112 在读到 ELSE 时既可以移入又可以归约，而 Bison 选择了前者，将后者用方括号括了起来。知道是这里出现了问题，我们就可以以此为线索修改 Bison 源代码或者重新修订语法了。

对于一个有一定规模的语法规范（如 C--语言）而言，Bison 所产生的 LALR 语法分析程序可以有一百甚至几百个状态。即使将它们都输出到了.output 文件里，在这些状态里逐个寻找潜在的问题也是挺费劲的。另外，有些问题，例如语法分析程序在运行时刻出现“Segmentation fault”等，很难从对状态机的静态描述中发现，必须要在动态、交互的环境下才容易看出问题所在。为了达到这种效果，在使用 Bison 进行编译的时候，可以通过附加-t 参数打开其诊断模式（或者在代码中加上#define YYDEBUG 1）：

```
bison -d -t syntax.y
```

在 main 函数调用 yyparse()之前我们加一句：yydebug = 1;，然后重新编译整个程序。之后运行这个程序你就会发现，语法分析程序现在正像一个自动机，一个一个状态地在进行转换，并将当前状态的信息打印到标准输出上，以方便你检查自己代码中哪里出现了问题。以前面的那个四则运算小程序为例，当打开诊断模式之后运行程序，屏幕上会出现如下字样：

```
1  Starting parse
```

```
2  Entering state 0
3  Reading a token:
```

如果我们输入 4，你会明显地看到语法分析程序出现了状态转换，并将当前栈里的内容打印出来：

```
 1  Next token is token INT ()
 2  Shifting token INT ()
 3  Entering state 1
 4  Reducing stack by rule 9 (line 29):
 5     $1 = token INT ()
 6  -> $$ = nterm Term ()
 7  Stack now 0
 8  Entering state 6
 9  Reducing stack by rule 6 (line 25):
10     $1 = nterm Term ()
11  -> $$ = nterm Factor ()
12  Stack now 0
13  Entering state 5
14  Reading a token:
```

继续输入其他内容，我们可以看到更进一步的状态转换。

注意，诊断模式会使语法分析程序的性能下降不少，建议在不使用时不要随便打开。

3.4.8 Bison：错误恢复

当输入文件中出现语法错误的时候，Bison 总是会让它生成的语法分析程序尽早地报告错误。每当语法分析程序从 yylex()得到了一个词法单元，如果当前状态并没有针对这个词法单元的动作，那就会认为输入文件里出现了语法错误，此时它默认进入下面这个错误恢复模式：

1) 调用 yyerror("syntax error")，只要你没有重写 yyerror()，该函数默认会在屏幕上打印出 syntax error 的字样。

2) 从栈顶弹出所有还没有处理完的规则，直到语法分析程序回到了一个可以移入特殊符号 error 的状态。

3) 移入 error，然后对输入的词法单元进行丢弃，直到找到一个能够跟在 error 之后的符号为止（该步骤也被称为再同步）。

4) 如果在 error 之后能成功移入三个符号，则继续正常的语法分析；否则，返回前面的步骤二。

这些步骤看起来似乎很复杂，但实际上需要我们做的事情只有一件，即在语法里指定 error 符号应该放到哪里。不过，需谨慎考虑放置 error 符号的位置：一方面，我们希望 error 后面跟的内容越多越好，这样再同步就会更容易成功，这提示我们应该把 error 尽量放在高层的产生式中；另一方面，我们又希望能够丢弃尽可能少的词法单元，这提示我们应该把 error 尽量放在底层的产生式中。在实际应用中，人们一般把 error 放在例如行尾、括号结尾等地方，本质上相当于让行结束符“;”以及括号“{”、“}”、“(”、“)”等作为错误恢复的同步符号：

Stmt → error SEMI

CompSt → error RC

Exp → error RP

以上几个产生式仅仅是示例，并不意味着把它们照搬到你的 Bison 源代码中就可以让语法分析程序能够满足实验三的要求。你需要进一步思考如何书写包含 error 的产生式才能够检查出输入文件中存在的各种语法错误。

3.4.9 语法分析提示

想要做好一个语法分析程序，第一步要仔细阅读并理解 C--语法规范。C--的语法要比它的词法复杂很多，如果缺乏对语法的理解，在调试和测试语法分析程序时你将感到无所适从。

接下来，我们建议你先写一个包含所有语法产生式但不包含任何语义动作的 Bison 源代码，然后将它和修改以后的 Flex 源代码、main 函数等一块编译出来先看看效果。对于一个没有语法错误的输入文件而言，这个程序应该什么也不输出；对于一个包含语法错误的输入文件而言，这个程序应该输出 syntax error。如果你的程序能够成功地判别有无语法错误，再去考虑优先级与结合性该如何设置以及如何进行错误恢复等问题；如果你的程序输出的结果不对，或者说你的程序根本无法编译，那你需要重新阅读前文并仔细检查哪里出了问题。好在此时代码并不算多，借助于 Bison 的.output 文件以及诊断模式等帮助，要查出错误并不是太难的事情。

再下一步需要考虑语法树的表示和构造。语法树是一棵多叉树，因此为了能够建立它你需要实现多叉树的数据结构。你需要专门写函数完成多叉树的创建和插入操作，然后在 Bison 源代码文件中修改属性值的类型为多叉树类型，并添加语义动作，将产生式右边的每一个符号所对应的树结点作为产生式左边的非终结符所对应的树结点的子结点逐个进行插入。具体这棵多叉树的数据结构怎么定义、插入操作怎么完成等完全取决于你的设计。

构造完这棵树之后，下一步就是按照实验要求中提到的缩进格式将它打印出来，同时要求打印的还有行号以及一些相关信息。为了能打印这些信息，你需要写专门的代码对生成的语法树进行遍历。由于还要求打印行号，所以在之前生成语法树的时候你就需要将每个结点第一次出现时的行号都记录下来（使用位置信息@n、使用变量 yylineno 或者自己维护这个行号均可以）。这段负责打印的代码仅是为了实验三而写，后面的实验不会再用，所以我们建议你将这些代码组织到一起或者写成函数接口的形式，以方便后面的实验对代码的调整。

实验 4 语义分析和中间代码生成

4.1 实验目的

（1）熟悉语义分析和中间代码生成过程。

（2）加深对语义翻译的理解。

4.2 实验任务

此次实验任务有两个：首先是**语义分析**任务，即在词法分析和语法分析程序的基础上编写一个程序，对输入的源代码进行语义分析和类型检查，并打印分析结果；然后，在语义正确的基础上，实现一种**中间代码的生成**。

任务 1：审查每一个语法结构的静态语义，即验证语法正确的结构是否有意义。此部分不再借助已有工具，需手写代码来完成。

任务 2：在词法分析、语法分析和语义分析程序的基础上，将输入源代码翻译成中间代码。（A、B 任务二选一）

A) 将 C--源代码翻译为中间代码，理论上中间代码在编译器的内部表示可以选用**树形结构（抽象语法树）**或者**线形结构（三地址代码）**等形式，为了方便检查你的程序，我们要求将中间代码输出成线性结构，从而可以使用我们提供的虚拟机小程序（附录 B）来测试中间代码的运行结果。

B) 编写一个中间代码生成程序，能将算术表达式等翻译成逆波兰、三元组或四元组形式（选择其中一种形式即可）

4.3 实验内容

4.3.1 实验要求

任务 1：语义分析

在本任务中，以 C--语言为例（选择其他语言同样可以事先进行相关假设），可对其做如下假设，你可以认为这些就是 C--语言的特性：

1) **假设 1**：整型（int）变量不能与浮点型（float）变量相互赋值或者相互运算。

2) **假设 2**：仅有 int 型变量才能进行逻辑运算或者作为 if 和 while 语句的条件；仅有 int 型和 float 型变量才能参与算术运算。

3) **假设 3**：任何函数只进行一次定义，无法进行函数声明。

4) **假设 4**：所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名。

5) **假设 5**：函数无法进行嵌套定义。

以上假设 1 至 5 也可视为要求，违反即会导致各种语义错误，不过我们只对后面讨论的 11 种错误类型进行考察。此外，**你可以安全地假设输入文件中不包含注释、八进制数、十六进制数、以及指数形式的浮点数、不包含结构体，也不包含任何词法或语法错误。**

你的程序需要对输入文件进行语义分析（输入文件中可能包含函数、一维和高维数组）并检查如

下类型的错误：

1) **错误类型 1**：变量在使用时未经定义。

2) **错误类型 2**：函数在调用时未经定义。

3) **错误类型 3**：变量出现重复定义，或变量与前面定义过的结构体名字重复。

4) **错误类型 4**：函数出现重复定义（即同样的函数名出现了不止一次定义）。

5) **错误类型 5**：赋值号两边的表达式类型不匹配。

6) **错误类型 6**：操作数类型不匹配或操作数类型与操作符不匹配（例如整型变量与数组变量相加减）。

7) **错误类型 7**：return 语句的返回类型与函数定义的返回类型不匹配。

8) **错误类型 8**：函数调用时实参与形参的数目或类型不匹配。

9) **错误类型 9**：对非数组型变量使用“[…]”（数组访问）操作符。

10) **错误类型 10**：对普通变量使用“(…)”或“()”（函数调用）操作符。

11) **错误类型 11**：数组访问操作符“[…]”中出现非整数（例如 a[1.5]）。

其中，要注意的是关于数组类型的等价机制，同 C 语言一样，只要数组的基类型和维数相同我们即认为类型是匹配的，例如 int a[10][2]和 int b[5][3]即属于同一类型。

任务 2：中间代码生成

在 **A 任务**中，我们对输入的 C--语言源代码文件做如下假设：

1) **假设 1**：不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量。

2) **假设 2**：不会出现类型为结构体或高维数组（高于 1 维的数组）的变量。

3) **假设 3**：任何函数参数都只能为简单变量，也就是说，结构体和数组都不会作为参数传入函数中。

4) **假设 4**：没有全局变量的使用，并且所有变量均不重名。

5) **假设 5**：函数不会返回结构体或数组类型的值。

6) **假设 6**：函数只会进行一次定义（没有函数声明）。

7) **假设 7**：输入文件中不包含任何词法、语法或语义错误（函数也必有 return 语句）。

你的程序需要将符合以上假设的 C--源代码翻译为中间代码，中间代码的形式及操作规范如表 4-1 所示，表中的操作大致可以分为如下几类：

1)标号语句 LABEL 用于指定跳转目标，注意 LABEL 与 x 之间、x 与冒号之间都被空格或制表符隔开。

2)函数语句 FUNCTION 用于指定函数定义，注意 FUNCTION 与 f 之间、f 与冒号之间都被空格或制表符隔开。

3)赋值语句可以对变量进行赋值操作（注意赋值号前后都应由空格或制表符隔开）。赋值号左边

的 x 一定是一个变量或者临时变量，而赋值号右边的 y 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将常数 5 赋给临时变量 t1，可以写成 t1 := #5。

4)算术运算操作包括加、减、乘、除四种操作（注意运算符前后都应由空格或制表符隔开）。赋值号左边的 x 一定是一个变量或者临时变量，而赋值号右边的 y 和 z 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将变量 a 与常数 5 相加并将运算结果赋给 b，则可以写成 b := a + #5。

5)赋值号右边的变量可以添加“&”符号对其进行取地址操作。例如，b := &a + #8 代表将变量 a 的地址加上 8 然后赋给 b。

6)当赋值语句右边的变量 y 添加了“*”符号时代表读取以 y 的值作为地址的那个内存单元的内容，而当赋值语句左边的变量 x 添加了“*”符号时则代表向以 x 的值作为地址的那个内存单元写入内容。

7)跳转语句分为无条件跳转和有条件跳转两种。无条件跳转语句 GOTO x 会直接将控制转移到标号为 x 的那一行，而有条件跳转语句（注意语句中变量、关系操作符前后都应该被空格或制表符分开）则会先确定两个操作数 x 和 y 之间的关系（相等、不等、小于、大于、小于等于、大于等于共 6 种），如果该关系成立则进行跳转，否则不跳转而直接将控制转移到下一条语句。

表 4-1 中间代码的形式及操作规范

语法	描述
LABEL x :	定义标号 x。
FUNCTION f :	定义函数 f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取 y 的地址赋给 x。
x := *y	取以 y 值为地址的内存单元的内容赋给 x。
*x := y	取 y 值赋给以 x 值为地址的内存单元。
GOTO x	无条件跳转至标号 x。
IF x [relop] y GOTO z	如果 x 与 y 满足[relop]关系则跳转至标号 z。
RETURN x	退出当前函数并返回 x 值。
DEC x [size]	内存空间申请，大小为 4 的倍数。
ARG x	传实参 x。
x := CALL f	调用函数，并将其返回值赋给 x。
PARAM x	函数参数声明。
READ x	从控制台读取 x 的值。
WRITE x	向控制台打印 x 的值。

8)返回语句 RETURN 用于从函数体内部返回值并退出当前函数，RETURN 后面可以跟一个变量，也

可以跟一个常数。

9)变量声明语句 DEC 用于为一个函数体内的局部变量声明其所需要的空间，该空间的大小以字节为单位。这个语句是专门为数组变量和结构体变量这类需要开辟一段连续的内存空间的变量所准备的。例如，如果我们需要声明一个长度为 10 的 int 类型数组 a，则可以写成 DEC a 40。对于那些类型不是数组或结构体的变量，直接使用即可，不需要使用 DEC 语句对其进行声明。变量的命名规范与之前的实验相同。另外，在中间代码中不存在作用域的概念，因此不同的变量一定要避免重名。

10)与函数调用有关的语句包括 CALL、PARAM 和 ARG 三种。其中 PARAM 语句在每个函数开头使用，对于函数中形参的数目和名称进行声明。例如，若一个函数 func 有三个形参 a、b、c，则该函数的函数体内前三条语句为：PARAM a、PARAM b 和 PARAM c。CALL 和 ARG 语句负责进行函数调用。在调用一个函数之前，我们先使用 ARG 语句传入所有实参，随后使用 CALL 语句调用该函数并存储返回值。仍以函数 func 为例，如果我们需要依次传入三个实参 x、y、z，并将返回值保存到临时变量 t1 中，则可分别表述为：ARG z、ARG y、ARG x 和 t1 := CALL func。注意 ARG 传入参数的顺序和 PARAM 声明参数的顺序正好相反。ARG 语句的参数可以是变量、以#开头的常数或以&开头的某个变量的地址。注意：当函数参数是结构体或数组时，ARG 语句的参数为结构体或数组的地址（即以传引用的方式实现函数参数传递）。

11)输入输出语句 READ 和 WRITE 用于和控制台进行交互。READ 语句可以从控制台读入一个整型变量，而 WRITE 语句可将一个整型变量的值写到控制台上。

除以上说明外，注意关键字及变量名都是大小写敏感的，也就是说“abc”和“AbC”会被作为两个不同的变量对待，上述所有关键字（例如 CALL、IF、DEC 等）都必须大写，否则虚拟机小程序会将其看作一个变量名。

在此任务中，你可能需要在语义分析的程序中做如下更改：在符号表中预先添加 read 和 write 这两个预定义的函数。其中 read 函数没有任何参数，返回值为 int 型（即读入的整数值），write 函数包含一个 int 类型的参数（即要输出的整数值），返回值也为 int 型（固定返回 0）。添加这两个函数的目的是让 C--源程序拥有可以与控制台进行交互的接口。在中间代码翻译的过程中，read 函数可直接对应 READ 操作，write 函数可直接对应 WRITE 操作。

在 **B 任务**中，你的程序应具有通用性，即能接受各种不同的算术表达式等语法成分。对于语法正确的算术表达式，能生成所选形式的相应序列，并输出结果；对不正确的表达式，能检测出错误。

4.3.2 输入格式

任务 1：你的程序的输入是一个包含源代码的文本文件，程序需要能够接收一个输入文件名作为参数。

任务 2：对于 **A 任务**，你的程序的输入是一个包含 C--源代码的文本文件，你的程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、输

出文件名为 out1.ir，程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 out1.ir 即可将输出结果写入当前目录下名为 out1.ir 的文件中。

对于 **B 任务**，你的程序输入是包含各种算术表达式的文本文件。

4.3.3 输出格式

任务 1：要求通过标准输出打印程序的运行结果。对于那些没有语义错误的输入文件，你的程序不需要输出任何内容。对于那些存在语义错误的输入文件，你的程序应当输出相应的错误信息，这些信息包括错误类型、出错的行号以及说明文字，其格式为：

Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类，否则将影响你的实验评分。

输入文件中可能包含一个或者多个错误（但每行最多只有一个错误），你的程序需要将它们全部检查出来。当然，有些时候输入文件中的一个错误会产生连锁反应，导致别的地方出现多个错误（例如，一个未定义的变量在使用时由于无法确定其类型，会使所有包含该变量的表达式产生类型错误），我们只会去考察你的程序是否报告了较本质那个的错误（如果难以确定哪个错误更本质一些，建议你报告所有发现的错误）。但是，如果源程序里有错而你的程序没有报错或报告的错误类型不对，又或者源程序里没有错但你的程序却报错，都会影响你的实验评分。

任务 2：对于 **A 任务**，要求你的程序将运行结果输出到文件。输出文件要求每行一条中间代码，每条中间代码的含义如前文所述。如果输入文件包含多个函数定义，则需要通过 FUNCTION 语句将这些函数隔开。FUNCTION 语句和 LABEL 语句的格式类似，具体例子见后面的样例。

对每个特定的输入，并不存在唯一正确的输出。我们将使用虚拟机小程序对你的中间代码的正确性进行测试。任何能被虚拟机小程序顺利执行并得到正确结果的输出都将被接受。此外，虚拟机小程序还会统计你的中间代码所执行过的各种操作的次数，以此来估计你的程序生成的中间代码的效率。

对于 **B 任务**，要求你的程序能输出生成的逆波兰、三元组或四元组序列。

4.3.4 测试环境

如果你选择的系统是 Ubuntu，你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

4) GNU Bison version 2.5.

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实验的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

4.3.5 提交要求

1) 可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

4.3.6 样例

任务 1：语义分析

请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

样例 1：

输入：

```
1    int main()
2    {
3        int i = 0;
4        j = i + 1;
5    }
```

输出：

样例输入中变量“j”未定义，因此你的程序可以输出如下的错误提示信息：

Error type 1 at Line 4: Undefined variable "j".

样例 2：

输入：

```
1    int main()
2    {
3        int i = 0;
4        inc(i);
5    }
```

输出：

样例输入中函数“inc”未定义，因此你的程序可以输出如下的错误提示信息：

Error type 2 at Line 4: Undefined function "inc".

样例 3：

输入：

```
1    int main()
2    {
3        int i, j;
4        int i;
5    }
```

输出：

样例输入中变量“i”被重复定义，因此你的程序可以输出如下的错误提示信息：

Error type 3 at Line 4: Redefined variable "i".

样例 4：

输入：

```
1   int func(int i)
2   {
3       return i;
4   }
5
6   int func()
7   {
8       return 0;
9   }
10
11   int main()
12   {
13   }
```

输出：

样例输入中函数“func”被重复定义，因此你的程序可以输出如下的错误提示信息：

Error type 4 at Line 6: Redefined function "func".

样例 5：

输入：

```
1   int main()
2   {
3       int i;
4       i = 3.7;
5   }
```

输出：

样例输入中错将一个浮点常数赋值给一个整型变量，因此你的程序可以输出如下的错误提示信息：

Error type 5 at Line 4: Type mismatched for assignment.

样例 6：

输入：

```
1   int main()
2   {
3       float j;
4       10 + j;
5   }
```

输出：

样例输入中表达式“10 + j”的两个操作数的类型不匹配，因此你的程序可以输出如下的错误提示信

息：

Error type 6 at Line 4: Type mismatched for operands.

样例 7：

输入：

```
1   int main()
2   {
3       float j = 1.7;
4       return j;
5   }
```

输出：

样例输入中“main”函数返回值的类型不正确，因此你的程序可以输出如下的错误提示信息：

Error type 7 at Line 4: Type mismatched for return.

样例 8：

输入：

```
1   int func(int i)
2   {
3       return i;
4   }
5
6   int main()
7   {
8       func(1, 2);
9   }
```

输出：

样例输入中调用函数“func”时实参数目不正确，因此你的程序可以输出如下的错误提示信息：

Error type 8 at Line 8: Function "func(int)" is not applicable for arguments "(int, int)".

样例 9：

输入：

```
1   int main()
2   {
3       int i;
4       i[0];
5   }
```

输出：

样例输入中变量“i”非数组型变量，因此你的程序可以输出如下的错误提示信息：

Error type 9 at Line 4: "i" is not an array.

样例 10：

输入：

```
1   int main()
```

```
2  {
3      int i;
4      i(10);
5  }
```

输出：

样例输入中变量“i”不是函数，因此你的程序可以输出如下的错误提示信息：

Error type 10 at Line 4: "i" is not a function.

样例 11：

输入：

```
1  int main()
2  {
3      int i[10];
4      i[1.5] = 10;
5  }
```

输出：

样例输入中数组访问符中出现了非整型常数“1.5”，因此你的程序可以输出如下的错误提示信息：

Error type 11 at Line 4: "1.5" is not an integer.

任务 2：中间代码生成

A 任务：

样例 1：

输入：

```
1  int main()
2  {
3    int n;
4    n = read();
5    if (n > 0) write(1);
6    else if (n < 0) write (-1);
7    else write(0);
8    return 0;
9  }
```

输出：

这段程序读入一个整数 n，然后计算并输出符号函数 sgn(n)。它所对应的中间代码可以是这样的：

```
1  FUNCTION main :
2  READ t1
3  v1 := t1
4  t2 := #0
5  IF v1 > t2 GOTO label1
6  GOTO label2
7  LABEL label1 :
```

```
 8  t3 := #1
 9  WRITE t3
10  GOTO label3
11  LABEL label2 :
12  t4 := #0
13  IF v1 < t4 GOTO label4
14  GOTO label5
15  LABEL label4 :
16  t5 := #1
17  t6 := #0 - t5
18  WRITE t6
19  GOTO label6
20  LABEL label5 :
21  t7 := #0
22  WRITE t7
23  LABEL label6 :
24  LABEL label3 :
25  t8 := #0
26  RETURN t8
```

需要注意的是，虽然样例输出中使用的变量遵循着字母后跟一个数字（如 t1、v1 等）的方式，标号也遵循着 label 后跟一个数字的方式，但这并不是强制要求的。也就是说，你的程序输出完全可以使用其它符合变量名定义的方式而不会影响虚拟机小程序的运行。

可以发现，这段中间代码中存在很多可以优化的地方。首先，0 这个常数我们将其赋给了 t2、t4、t7、t8 这四个临时变量，实际上赋值一次就可以了。其次，对于 t6 的赋值我们可以直接写成 t6 := # -1 而不必多进行一次减法运算。另外，程序中的标号也有些冗余。如果你的程序足够“聪明”，可能会将上述中间代码优化成这样：

```
 1  FUNCTION main :
 2  READ t1
 3  v1 := t1
 4  t2 := #0
 5  IF v1 > t2 GOTO label1
 6  IF v1 < t2 GOTO label2
 7  WRITE t2
 8  GOTO label3
 9  LABEL label1 :
10  t3 := #1
11  WRITE t3
12  GOTO label3
13  LABEL label2 :
```

```
14  t6 := #-1
15  WRITE t6
16  LABEL label3 :
17  RETURN t2
```

样例 2：

输入：

```
 1  int fact(int n)
 2  {
 3      if (n == 1)
 4      return n;
 5      else
 6      return (n * fact(n - 1));
 7  }
 8  int main()
 9  {
10      int m, result;
11      m = read();
12      if (m > 1)
13      result = fact(m);
14      else
15      result = 1;
16      write(result);
17      return 0;
18  }
```

输出：

这是一个读入 m 并输出 m 的阶乘的小程序，其对应的中间代码可以是：

```
 1  FUNCTION fact :
 2  PARAM v1
 3  IF v1 == #1 GOTO label1
 4  GOTO label2
 5  LABEL label1 :
 6  RETURN v1
 7  LABEL label2 :
 8  t1 := v1 - #1
 9  ARG t1
10  t2 := CALL fact
11  t3 := v1 * t2
12  RETURN t3
13
```

```
14  FUNCTION main :
15  READ t4
16  v2 := t4
17  IF v2 > #1 GOTO label3
18  GOTO label4
19  LABEL label3 :
20  ARG v2
21  t5 := CALL fact
22  v3 := t5
23  GOTO label5
24  LABEL label4 :
25  v3 := #1
26  LABEL label5 :
27  WRITE v3
28  RETURN #0
```

这个样例主要展示如何处理包含多个函数以及函数调用的输入文件。

4.4 实验指导

4.4.1 语义分析

除了词法和语法分析之外，编译器前端所要进行的另一项工作就是对输入程序进行语义分析。进行语义分析的原因很简单：一段语法上正确的源代码仍可能包含严重的逻辑错误，这些逻辑错误可能会对编译器后面阶段的工作产生影响。首先，我们在语法分析阶段所借助的理论工具是上下文无关文法，从名字上就可以看出上下文无关文法没有办法处理一些与输入程序上下文相关的内容（例如变量在使用之前是否已经被定义过，一个函数内部定义的变量在另一个函数中是否允许使用等）。这些与上下文相关的内容都会在语义分析阶段得到处理，因此也有人将这一阶段叫做上下文相关分析（Context-sensitive Analysis）。其次，现代程序设计语言一般都会引入类型系统，很多语言甚至是强类型的。引入类型系统可以为程序设计语言带来很多好处，例如它可以提高代码在运行时刻的安全性，增强语言的表达力，还可以使编译器为其生成更高效的目标代码。对于一个具有类型系统的语言来说，编译器必须要有能力检查输入程序中的各种行为是否都是类型安全的，因为类型不安全的代码出现逻辑错误的可能性很高。最后，为了使之后的阶段能够顺利进行，编译器在面对一段输入程序时不得不从语法之外的角度进行理解。比如，假设输入程序中有一个变量或函数 x，那么编译器必须要提前确定：

1) 如果 x 是一个变量，那么变量 x 中存储的是什么内容？是一个整数值、浮点数值，还是一组整数值或其它自定义结构的值？

2) 如果 x 是一个变量，那么变量 x 在内存中需要占用多少字节的空间？

3) 如果 x 是一个变量，那么变量 x 的值在程序的运行过程中会保留多长时间？什么时候应当创建

x，而什么时候它又应该消亡？

4) 如果 x 是一个变量，那么谁该负责为 x 分配存储空间？是用户显式地进行空间分配，还是由编译器生成专门的代码来隐式地完成这件事？

5) 如果 x 是一个函数，那么这个函数要返回什么类型的值？它需要接受多少个参数，这些参数又都是什么类型？

以上这些与变量或函数 x 有关的信息中，几乎所有都无法在词法或语法分析过程中获得，即输入程序能为编译器提供的信息要远超过词法和语法分析能从中挖掘出的信息。

从编程实现的角度看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的语法分析模块或者并入后面的中间代码生成模块。不过，由于其牵扯到的内容较多而且较为繁杂，我们还是将语义分析单独作为一块内容。我们下面先对语义分析所要用到的属性文法做简要介绍，然后对 C--语言编译中的符号表和类型表示这两大重点内容进行讨论，最后提出帮助顺利完成语义分析任务的一些建议。

4.4.1.1 属性文法

在词法分析过程中，我们借助了正则文法；在语法分析过程中，我们借助了上下文无关文法；现在到了语义分析部分，为什么我们不能在文法体系中更上一层，采用比上下文无关文法表达力更强的上下文相关文法呢？

之所以不继续采用更强的文法，原因有两个：其一，识别一个输入是否符合某一上下文相关文法，这个问题本身是 P-Space Complete 的，也就是说，如果使用上下文相关文法那么编译器的复杂度会很高；其二，编译器需要获取的很多信息很难使用上下文相关文法进行编码，这就迫使我们需要为语义分析寻找其它更实用的理论工具。

目前被广泛使用的用于语义分析的理论工具叫做**属性文法（Attribute Grammar）**，它是由 Knuth 在 50 年代所提出。属性文法的核心思想是，为上下文无关文法中的每一个终结符或非终结符赋予一个或多个属性值。对于产生式 A→ X1 … Xn 来说，在自底向上分析中 X1…Xn 的属性值是已知的，这样语义动作只会为 A 计算属性值；而在自顶向下分析中，A 的属性值是已知的，在该产生式被应用之后才能知道 X1…Xn 的属性值。终结符号的属性值通过词法分析可以得到，非终结符号的属性值通过产生式对应的语义动作来计算。

属性值可以分成不相交的两类：**综合属性（Synthesized Attribute）**和**继承属性（Inherited Attribute）**。在语法树中，一个结点的综合属性值是从其子结点的属性值计算而来的，而一个结点的继承属性值则是由该结点的父结点和兄弟结点的属性值计算而来的。如果对一个文法 P，$\forall A \to X_1 ... X_n \in P$ 都有与之相关联的若干个属性定义规则，则称 P 为**属性文法**。如果属性文法 P 只包含综合属性而没有继承属性，则称 P 为 **S 属性文法**。如果每个属性定义规则中的每个属性要么是一个综合属性，要么是 X_j 的一个继承属性，并且该继承属性只依赖于 $X_1 \ldots X_{j-1}$ 的属性和 A 的继承属性，则

称 P 为 **L 属性文法**。

以属性文法为基础可衍生出一种非常强大的翻译模式，我们称之为**语法制导翻译（Syntax-Directed Translation 或 SDT）**。在 SDT 中，人们把属性文法中的属性定义规则用计算属性值的语义动作来表示，并用花括号“{”和“}”括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交错的表示法。事实上，我们在之前使用 Bison 时已经用到了属性文法和 SDT。

4.4.1.2 符号表

符号表对于编译器至关重要。在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括：上述名字的种类、具体类型、维数、参数个数、数值及目标地址（存储单元地址）等。

符号表上的操作包括填表和查表两种。当分析到程序中的说明或定义语句时, 应将说明或定义的名字，以及与之有关的特性信息填入符号表中，这便是填表操作。查表操作则使用得更为广泛，需要使用查表操作的情况有：填表前查表，包括检查在输入程序的同一作用域内名字是否被重复定义，检查名字的种类是否与说明一致，对于那些类型要求更强的语言，则要检查表达式中各变量的类型是否一致等；此外生成目标指令时，也需要查表以取得所需要的地址或者寄存器编号等。符号表的组织方式也有多种，可以将程序中出现的所有符号组织成一张表，也可以将不同种类的符号组织成不同的表（例如，所有变量名组织成一张表，所有函数名组织成一张表，所有临时变量组织成一张表，等等）。你可以针对每个语句块都新建一张表，也可以将所有语句块中出现的符号全部插入到同一张表中。符号表可以仅支持插入操作而不支持删除操作（此时如果要实现作用域则需要将符号表组织成层次结构），也可以组织一张既可以插入又可以删除的、支持动态更新的表。不同的组织方式各有利弊，你可仔细思考并为**语义分析任务**做出决定。

至于在符号表里应该填些什么，这与不同程序设计语言的特性相关，更取决于编译器的设计者本身。只要觉得方便，可以向符号表里填任何内容！毕竟符号表就是为了支持编写编译器而设置的。就**语义分析任务**而言，对于变量至少要记录变量名及其类型，对于函数至少要记录其返回类型、参数个数以及参数类型。

至于符号表应该采用何种数据结构实现，这个问题同样没有统一的答案。不同的数据结构有不同的时间复杂度、空间复杂度以及编程难度，我们下面讨论几种最常见的选择。

线性链表：

符号表里所有的符号（假设有 n 个，下同）都用一条链表串起来，插入一个新的符号只需将该符号放在链表的表头，其时间复杂度是 O(1)。在链表中查找一个符号需要对其进行遍历，时间复杂度是 O(n)。删除一个符号只需要将该符号从链表里摘下来，不过在摘之前由于我们必须要执行一次查找操作以找到待删除的结点，因此时间复杂度也是 O(n)。

链表的最大问题是它的查找和删除效率太低，一旦符号表中的符号数量较大，查表操作将变得十分耗时。不过，使用链表的好处也是显而易见：它的结构简单，编程容易，可以被快速实现。如果你事先能够确定表中的符号数目较少（例如，在面向对象语言的一些短方法中），链表是一个非常不错的选择。

平衡二叉树：

相对于只能执行线性查找的链表而言，在平衡二叉树上进行查找天生就是二分查找。在一个典型的平衡二叉树实现（例如 AVL 树、红黑树或伸展树等）上查找一个符号的时间复杂度是 O(logn)。插入一个符号相当于进行一次失败的查找而找到待插入的位置，时间复杂度也是 O(logn)。删除一个符号可能需要做更多的维护操作，但其时间复杂度仍然维持在 O(logn)的级别。

平衡二叉树相对于其它数据结构而言具有很多优势，例如较高的搜索效率（在绝大多数应用中 O(logn)的搜索效率已经完全可以接受）以及较好的空间效率（它所占用的空间随树中结点的增多而增长，不像散列表那样每张表都需要大量的空间）。平衡二叉树的缺点是编程难度高，成功写完并调试出一个能用的红黑树所需要的时间不亚于你完成语义分析所需的时间。不过如果你真的想要使用类似于红黑树的数据结构，也可以从其它地方（例如 Linux 内核代码中）寻找别人写好的红黑树源代码。

散列表：

散列表是一种可以达到搜索效率极致的数据结构。一个好的散列表实现可以让插入、查找和删除的平均时间复杂度都达到 O(1)。同时，与红黑树等操作复杂的数据结构不同，散列表在代码实现上也很简单：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。对于符号表来说，一个最简单的散列函数（即 hash 函数）可以把符号名中的所有字符相加，然后对符号表的大小取模。你可以寻找更好的 hash 函数，这里我们提供一个不错的选择，由 P.J. Weinberger 所提出：

```
1   unsignedinthash_pjw(char* name)
2   {
3     unsignedint val = 0, i;
4     for (; *name; ++name)
5    {
6     val = (val << 2) + *name;
7     if (i = val & ~0x3fff) val = (val ^ (i >> 12)) & 0x3fff;
8    }
9    return val;
10  }
```

需要注意的是，代码第 7 行的常数（0x3fff）确定了符号表的大小（即 16384），用户可根据实际

需要调整此常数以获得大小合适的符号表。如果散列表出现冲突，则可以通过在相应数组元素下面挂一个链表的方式（称为 open hashing 或 close addressing 方法，推荐使用），或再次计算散列函数的值而为当前符号寻找另一个槽的方式（称为 open addressing 或者 rehashing 方法）来解决。如果你还知道一些更酷的技术，如 multiplicative hash function 以及 universal hash function，那将会使你的散列表的元素分布更加平均一些。由于散列表无论在搜索效率和编程难度上的优异表现，它已经成为符号表的实现中最常被采用的数据结构。

Multiset Discrimination：

虽然散列表的平均搜索效率很高，但在最坏情况下它会退化为 O(n)的线性查找，而且几乎任何确定的散列函数都存在某种最坏的输入。另外，散列表所要申请的内存空间往往比输入程序中出现的所有符号的数量还要多，较为浪费。如果我们能只为输入程序中出现的每个符号单独分配一个编号和空间，那岂不是既省空间又不会有冲突吗？Multiset discrimination 就是基于这种想法。在词法分析部分，我们先统计输入程序中出现的所有符号（包括变量名、函数名等），然后把这些符号按照名字进行排序，最后申请一张与符号总数量一样大的符号表，查表功能可通过基于符号名的二分查找实现。

4.4.1.3 类型表示

“类型”包含两个要素：一组值，以及在这组值上的一系列操作。当我们在某组值上尝试去执行其不支持的操作时，类型错误就产生了。一个典型程序设计语言的类型系统应该包含如下四个部分：

1) 一组基本类型。在 C--语言中，基本类型包括 int 和 float 两种。
2) 从一组类型构造新类型的规则。在 C--语言中，可以通过定义数组和结构体来构造新的类型。
3) 判断两个类型是否等价的机制。在 C--语言中，默认要求实现名等价。
4) 从变量的类型推断表达式类型的规则。

目前程序设计语言的类型系统分为两种：强类型系统（Strongly Typed System）和弱类型系统（Weakly Typed System）。前者在任何时候都不允许出现任何类型错误，而后者可以允许某些类型错误出现在运行时刻。强类型系统的语言包括 Java、Python、LISP、Haskell 等，而弱类型系统的语言最典型的代表就是 C 和 C++语言。

编译器尝试去发现输入程序中的类型错误的过程被称为是**类型检查**。根据进行检查的时刻的不同，类型检查可被划分为两类，即**静态类型检查（Static Type Checking）**和**动态类型检查（Dynamic Type Checking）**。前者仅在编译时刻进行类型检查，不会生成与类型检查有关的任何目标代码，而后者则需要生成额外的代码在运行时刻检查每次操作的合法性。静态类型检查的好处是生成的目标代码效率高，缺点是粒度比较粗，某些运行时刻的类型错误可能检查不出来。动态类型检查的好处是更加精确与全面，但由于在运行时执行了过多的检查和维护工作，故目标代码的运行效率往往比不上静态类型检查。

关于什么样的类型系统更好，人们进行了长期、激烈而又没有结果的争论。动态类型检查语言更

适合快速开发和构建程序原型（因为这类语言往往不需要指定变量的类型），而使用静态类型检查语言写出来的程序通常拥有更少的错误（因为这类语言往往不允许多态）。强类型系统语言更加健壮，而弱类型系统语言更加高效。总之，不同的类型系统特点不一，目前还没有哪种选择在所有情况下都比其它选择来得更好。

介绍完基本概念后，我们来考察实现上的问题。如果整个语言中只有基本类型，那么类型的表示将会极其简单：我们只需用不同的常数代表不同的类型即可。但是，在引入了数组（尤其是多维数组）以及结构体之后，类型的表示就不那么简单了。想像一下多维数组该如何去表示呢？最简单的表示方法还是链表。多维数组的每一维都可以作为一个链表结点，每个链表结点存两个内容：数组元素的类型，以及数组的大小。例如，int a[10][3]可以表示为图 4-1 所示的形式。

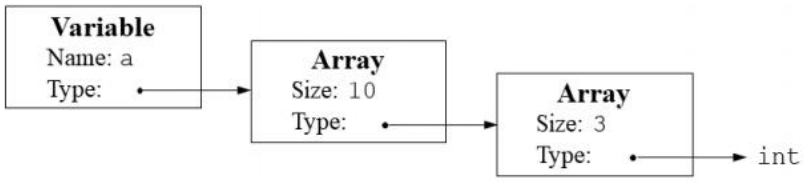


图 4-1 多维数组的链表表示示例

同作用域一样，类型系统也是语义分析的一个重要的组成部分。C--语言属于强类型系统，并且进行静态类型检查。当我们尝试着向 C--语言中添加更多的性质，例如引入指针、面向对象机制、显式/隐式类型转换、类型推断等时，你会发现实现编译器的复杂程度会陡然上升。一个严谨的类型检查机制需要通过将类型规则转化为形式系统，并在这个形式系统上进行逻辑推理。为了控制实验的难度我们可以无需这样费事，但应该清楚实用的编译器内部类型检查要复杂的多。

4.4.1.4 语义分析提示

语义分析需要在词法和语法分析的基础上完成，特别是需要在语法分析时所构建的语法树上完成。语义分析仍然需要对语法树进行遍历以进行符号表的相关操作以及类型的构造与检查。你可以模仿 SDT 在 Bison 代码中插入语义分析的代码，但我们更推荐的做法是，Bison 代码只用于构造语法树，而把和语义分析相关的代码都放到一个单独的文件中去。如果采用前一种做法，所有语法结点的属性值请尽量使用综合属性；如果采用后一种做法，就没有这些限制。

每当遇到语法单元 ExtDef 或者 Def，就说明该结点的子结点们包含了变量或者函数的定义信息，这时候应当将这些信息通过对子结点们的遍历提炼出来并插入到符号表里。每当遇到语法单元 Exp，说明该结点及其子结点们会对变量或者函数进行使用，这个时候应当查符号表以确认这些变量或者函数是否存在以及它们的类型是什么。具体如何进行插入与查表，取决于你的符号表和类型系统的实现。语义分析要求检查的错误类型较多，因此你的代码需要处理的内容也较复杂，请仔细完成。还有一点值得注意，在发现一个语义错误之后不要立即退出程序，因为实验要求中有说明需要你的程序有能力查出输入程序中的多个错误。

实验要求的必做内容共有 11 种语义错误需要检查，它们只涉及到查表与类型操作，其中类型不考

虑比较复杂的结构体，即默认输入的源程序里不包含结构体，以此来降低实验的难度。

4.4.2 中间代码生成

编译器里最核心的数据结构之一就是中间代码（Intermediate Representation 或 IR）。中间代码应包含哪些信息，这些信息又应有怎样的内部表示？这些问题会极大地影响编译器代码的复杂程度、编译器的运行效率、以及编译生成的目标代码的运行效率。

广义地说，编译器中根据输入程序所构造出来的绝大多数数据结构都被称为中间代码（或可更精确地译为“中间表示”）。例如，我们之前所构造的词法流、语法树、带属性的语法树等，都可视为中间代码。使用中间代码的主要原因是为了方便编写编译器程序的各种操作。如果我们在需要有关输入程序的任何信息时都只能去重新读入并处理输入程序源代码的话，编译器的编写将会变得非常麻烦，同时也会大大降低其运行效率。

狭义地说，中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式（这时它常被称作 Intermediate Code）。你可能会有疑问：为什么编译器不能把输入程序直接翻译成目标代码，而是要额外引入中间代码呢？实际上，引入中间代码有两个主要的好处。一方面，中间代码将编译器自然地分为前端和后端两个部分。当我们需要改变编译器的源语言或目标语言时，如果采用了中间代码，我们只需要替换原编译器的前端或后端，而不需要重写整个编译器。另一方面，即使源语言和目标语言是固定的，采用中间代码也有利于编译器的模块化。人们将编译器设计中那些复杂但相关性不大的任务分别放在前端和后端的各个模块中，这既简化了模块内部的处理，又使我们能单独对每个模块进行调试与修改而不影响其它模块。下文中，如果不特别说明，“中间代码”都是指狭义的中间代码。

4.4.2.1 中间代码的分类

中间代码的设计可以说更多的是一门艺术而不是技术。不同编译器所使用的中间代码可能是千差万别的，即使是同一编译器内部也可以使用多种不同的中间代码：有的中间代码与源语言更接近，有的中间代码与目标语言更接近。编译器需要在不同的中间代码之间进行转换，有时为了处理的方便，甚至会在将中间代码 1 转换为中间代码 2 之后，对中间代码 2 进行优化然后又转换回中间代码 1。这些不同的中间代码虽然对应了同一输入程序，但它们却体现了输入程序不同层次上的细节信息。举个实际的例子：GCC 内部首先会将输入程序转换成一棵抽象语法树，然后将该树转换为另一种被称为 GIMPLE 的树形结构。在 GIMPLE 之上它建立静态单赋值式的中间代码之后，又会将其转换为一种非常底层的 RTL（Register Transfer Language）代码，最后才把 RTL 转换为汇编代码。

我们可以从不同的角度对现存的这些花样繁多的中间代码进行分类。从中间代码所体现出的细节上，我们可以将中间代码分为如下三类：

1) **高层次中间代码（High-level IR 或 HIR）**：这种中间代码体现了较高层次的细节信息，因此往往和高级语言类似，保留了不少包括数组、循环在内的源语言的特征。高层次中间代码常在编译器的前端部分使用，并在之后被转换为更低层次的中间代码。高层次中间代码常被用于进行相关性分析

（ Dependence Analysis ）和解释执行。我们所熟悉的 Java bytecode、Python .pyc bytecode 以及目前使用得非常广泛的 LLVM IR 都属于高层次 IR。

2) **中层次中间代码（Medium-level IR 或 MIR）**：这个层次的中间代码在形式上介于源语言和目标语言之间，它既体现了许多高级语言的一般特性，又可以被方便地转换为低级语言的代码。正是由于 MIR 的这个特性，它是三种 IR 中最难设计的一种。在这个层次上，变量和临时变量可能已经有了区分，控制流也可能已经被简化为无条件跳转、有条件跳转、函数调用和函数返回四种操作。另外，对中层次中间代码可以进行绝大部分的优化处理，例如公共子表达式消除（Common-subexpression Elimination）、代码移动（Code Motion）、代数运算简化（Algebraic Simplification）等。

3) 低层次中间代码（Low-level IR 或 LIR）：低层次中间代码与目标语言非常接近，它在变量的基础上可能会加入寄存器的细节信息。事实上，LIR 中的大部分代码和目标语言中的指令往往存在着一一对应的关系，即使没有对应，二者之间的转换也属于一趟指令选择就能完成的任务。前面提到的 RTL 就属于一种非常典型的低层次 IR。

图 4-2 给出了一个完成相同功能的三种 IR 的例子（从左到右依次为 HIR、MIR 和 LIR）。

```
t1 = a[i][j+2]        t1 = j + 2            r1 = [fp - 4]
                      t2 = i * 20           r2 = r1 + 2
                      t3 = t1 + t2          r3 = [fp - 8]
                      t4 = 4 * t3           r4 = r3 * 20
                      t5 = addr a           r5 = r4 + r2
                      t6 = t5 + t4          r6 = 4 * r5
                      t7 = *t6              r7 = fp - 216
                                            f1 = [r7 + r6]
```

图 4-2 三种不同层次的中间代码示例

从表示方式来看，我们又可以将中间代码分成如下三类：

1) **图形中间代码（Graphical IR）**：这种类型的中间代码将输入程序的信息嵌入到一张图中，以结点和边等元素来组织代码信息。由于要表示和处理一般的图代价会很大，人们经常会使用特殊的图，例如树或有向无环图（DAG）。一个典型的树形中间代码的例子就是抽象语法树（Abstract Syntax Tree 或 AST）。抽象语法树中省去了语法树里不必要的结点，将输入程序的语法信息以一种更加简洁的形式呈现出来。其它树形中间代码的例子有 GCC 中所使用的 GIMPLE。这类中间代码将各种操作都组织在一棵树中。

2) **线形中间代码（Linear IR）**：线形结构的代码我们见得非常多，例如我们经常使用的 C 语言、Java 语言和汇编语言中语句和语句之间就是线性关系。你可以将这种中间代码看成是某种抽象计算机的一个简单的指令集。这种结构最大的优点是表示简单、处理高效，而缺点就是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得很复杂。

3) 混合型中间代码（Hybrid IR）：顾名思义，混合型中间代码主要混合了图形和线形两种中间代码，期望结合这两种代码的优点并避免二者的缺点。例如，我们可以将中间代码组织成许多基本块，块内部采用线形表示，块与块之间采用图表示，这样既可以简化块内部的数据流分析，又可以简化块与块之间的控制流分析。

在中间代码生成 A 任务中，你需要按照格式输出中间代码。虽然实验要求中规定的中间代码格式类似于线形的中层次中间代码，但这只是输出格式，而你的程序内部可以采用任何形式的中间代码，而这些中间代码中又体现了多少细节信息，则完全取决于你自己的设计。

4.4.2.2 中间代码的表示（线形）

在 A 任务中，你可能会一边对语法树进行处理一边把要输出的代码内容打印出来。这种做法其实并不好，因为当代码内容被打印出来的那一刻起，我们就已经失去了对这些代码进行调整和优化的机会。更加合理的做法是将所生成的中间代码先保存到内存中，等全部翻译完毕，优化也都做完后再使用一个专门的打印函数把在内存中的中间代码打印出来。既然生成好的中间代码会被放到内存中，那么如何保存这些代码以及为其设计怎样的数据结构就是值得考虑的问题了。我们下面对一种典型的线形 IR 的实现细节进行介绍，关于树形 IR 的

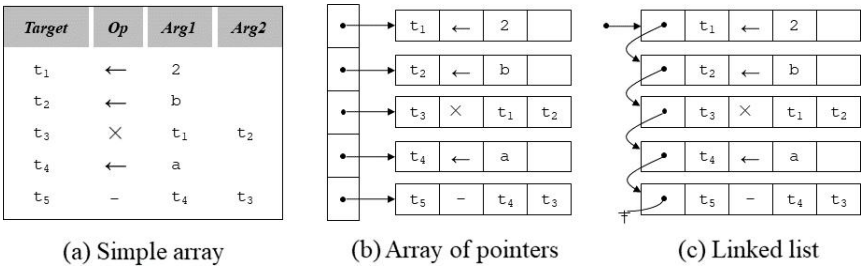


图 4-3 表示线性 IR 的三种基本数据结构

实现细节我们将放到下一节介绍。

相对而言，线形 IR 是实现起来最简单，而且打印起来最方便的中间代码形式。由于代码本身是线形的，我们可以使用几种最基本的线形数据结构来表示它们，如图 4-3 所示。

其中图 4-3(a)为一个大的静态数组，数组中的每个元素（图中的一行）就是一条中间代码。使用静态数组的好处是写起来编程方便，缺点是灵活性不足。中间代码的最大行数受限，而且代码的插入、删除以及调换位置的代价较大。图 4-3(b)同样为一个大数组，但数组中的每个元素并不是一条中间代码，而是一个指向中间代码指针。虽然采用这种实现时代码行数也会受限，不过它和图 3(a)的实现相比则大大减少了调换代码位置的开销。图 4-3(c)是一个纯链表的实现，图中画出来的链表是单向的，但我们更建议使用双向循环链表。链表以增加实现的复杂性为代价换得了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且几乎不存在代码最大行数的限制。

假设单条中间代码的数据结构定义为：

```
1  typedef struct Operand_* Operand;
2  struct Operand_ {
3    enum { VARIABLE, CONSTANT, ADDRESS, … } kind;
4    union {
5      int var_no;
6      int value;
```

```
7       …
8     } u;
9   };
10
11   struct InterCode
12   {
13     enum { ASSIGN, ADD, SUB, MUL, … } kind;
14     union {
15       struct { Operand right, left; } assign;
16       struct { Operand result, op1, op2; } binop;
17       …
18     } u;
19   }
```

那么，图 4-3(a)中的实现可以写成：

```
InterCode codes[MAX_LINE];
```

图 4-3(b)中的实现可以写成：

```
InterCode* codes[MAX_LINE];
```

图 4-3(c)中的（双向链表）实现可以写成：

```
struct InterCodes { InterCode code; struct InterCodes *prev, *next; };
```

要想打印出线形 IR 非常简单，只需从第一行代码开始逐行访问，根据每行代码 kind 域的不同值按照不同的格式打印即可。对于数组，逐行访问其实就意味着一个 for 循环；对于链表，逐行访问则意味着以一个 while 循环顺着链表的 next 域进行迭代。

4.4.2.3 中间代码的表示（树形）

树形 IR 看上去可能让人感到复杂，但仔细想想就会发现其实它与线形 IR 一样直观。树形结构天然具有层次的概念，在靠近树根的高层部分的中间代码其抽象层次较高，而靠近树叶的低层部分的中间代码则更加具体。例如，中间代码 t1 := v2 + #3 可用树形结构表示为如图 4-4 所示。

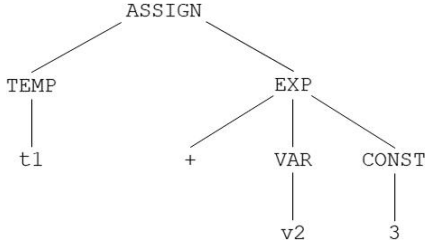


图 4-4 中间代码 t1 := v2 + #3 的树形结构表示

我们也可以从另一个角度来理解树形 IR。在上次实验中，我们曾为输入程序构造过语法树，这棵

语法树所对应的程序设计语言是编译器的源语言；而在这里，树形结构同样可以看作是一棵语法树，它所对应的程序设计语言则是我们的中间代码。源语言的语法相当复杂，但此次实验要求我们输出的中间代码的语法规则却是简单的，因此树形结构的中间代码的设计与实现也会比语法树更简单。

之前我们已经做过有关语法树的实验，你对于树形结构该如何实现应当非常熟悉。正如前面所述，树形 IR 可以看作是一种基于中间代码的语法树（或抽象语法树），因此其数据结构以及实现细节与语法树非常类似。有了写语法树的经验，写树形 IR 只需在原有基础上稍加修改即可，不会带来太多困难。

树形 IR 的打印要比线形 IR 复杂一些，该任务类似于给定一棵输入程序的语法树需要将该输入程序打印出来。你需要对树形 IR 进行（深度优先）遍历，根据当前结点的类型递归地对其各个子结点进行打印。从另外一个角度看，从之前实验中构造的语法树到实验三要求输出的中间代码之间总要经历一个由树形到线形的转换，使用线形 IR 其实就是将这步转换提前到构造 IR 时，而使用树形 IR 则是将这步转换推后到输出时才进行。

4.4.2.4 运行时环境初探

在一个程序员眼里，程序设计语言中可以有很多机制，包括类型（基本类型、数组和结构体），类和对象，异常处理，动态类型，作用域，函数调用，名空间等等，而且每个程序似乎都有使用不完的内存空间。但很显然，程序运行所基于的底层硬件不能支持这么多机制。一般来说，硬件只对 32bits 或 64bits 位整数、IEEE 浮点数、简单的算术运算、数值拷贝，以及简单的跳转提供直接的支持，并且其存储器的大小也是有限的、结构也是分层的。程序设计语言中 的其它机制则需要编译器、汇编/链接器、操作系统等共同努力，从而让程序员们产生一种幻觉，认为他们眼中所看到的所有机制都是被底层硬件直接支持的。

使用程序设计语言所书写出来的变量、类、函数等都是些抽象层次比较高的概念，为了能使用硬件直接支持的底层操作来表示这些高抽象层次的概念，仅靠编译时刻字面上的代码翻译是远远不够的，我们需要能够生成额外的目标代码，使程序在运行时刻可以维护起一系列的结构以支撑起程序设计语言中的各种高级特性。这些程序员一般不可见、但又确实存在于运行时刻的结构就被称为运行时环境（Runtime Environment）。运行时环境与源语言、目标语言和目标机器都紧密相关，其中包含很多细节，此次实验中我们以介绍原理为主，附带介绍一些简单结构（例如数组和结构体）的实现方式。

高级语言中的 char、short 和 int 等类型一般会直接对应到底层机器上的一个、两个或四个字节，而 float 和 double 类型则会对应到底层机器上的四个和八个字节，这些类型都可以由硬件直接提供支持。底层硬件中没有指针类型，但指针可以用四个字节（32bits 位机器）或者八个字节（64bits 位机器）整数表示，其内容即为指针所指向的内存地址。

以上是一些比较基本的类型，下面我们来考察一维数组的表示。最熟悉的表示数组的方式是 C 风格的（如图 4-5 所示），即数组中的元素一个挨着一个并占用一段连续的内存空间。

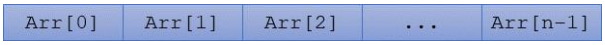


图 4-5　C 语言中一维数组的内存表示方式

图 4-6 Java 语言中一维数组的内存表示方式

当然这不是唯一的表示方法。Java 在编译 bytecode 时就会采取另外一种布局，将数组长度放在起始位置（Pascal 中的 string 类型数据也是这样保存的），如图 4-6 所示。

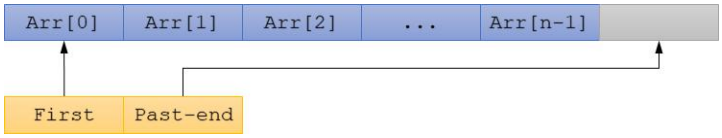


图 4-7 D 语言中一维数组的内存表示方式

图 4-8　C 语言中多维数组的内存表示方式

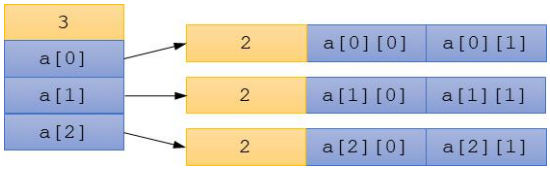


图 4-9　Java 语言中多维数组的内存表示方式

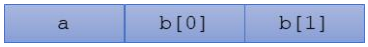


图 4-10　结构体的内存表示示例

另外还有一种表示方式为 D 语言所采用：数组变量本身仅由两个指针组成，一个指向数组的开头，另一个指向数组的末尾之后，数组的所有信息存在于另外一段内存之中，如图 4-7 所示。可以看出，无论是哪一种表示方式，数组元素在内存中总是连续存储的，这当然是为了使数组的访问能够更快（只需计算基地址+偏移量，然后取值即可）。多维数组可以看作一维数组的数组，C 风格的表示方法仍然是使用一段连续的内存空间，如图 4-8 所示。

而 Java 中每个一维数组是一个独立的对象，因此多维数组中的各维一般不会聚在一起，如图 4-9 所示。

结构体的表示与数组类似，最常见的办法是将各个域按定义的顺序连续地存放在一起。比如 struct { int a; float b[2]; }在内存中的表示如图 4-10 所示。

图 4-11　C—语言中结构体的内存表示的错误示例

图 4-12 C—语言中结构体的内存表示的正确示例

我们此次实验中只包含 int 和 float 两种类型，而这两种类型的宽度都是四字节，这省去了我们许多的麻烦。如果 C—语言允许其它宽度的类型存在又会如何呢？例如，struct { int a; char b; double c; } 这个结构体在内存中会排成如图 4-11 所示的表示方式吗？

答案是不会。如果没有特别指定，GCC 总会将结构体中的域对齐到字边界。因此在 char b 和 double c 之间会有 3 字节的空间被浪费掉，如图 4-12 所示。

x86 平台允许变量在存储时不对齐，但 MIPS 平台则要求对齐，这是我们需要注意的。最后我们简单讨论一下函数的表示。众所周知，在调用函数时我们需要进行一系列的配套工作，包括找到函数的入口地址、参数传递、为局部变量申请空间、保存寄存器现场、返回值传递和控制流跳转等等。在这一过程中，我们需要用到的各种信息都必须有地方能够保存下来，而保存这些信息的结构就称为活动记录（Activation Record）。因为活动记录经常被保存在栈上，故它往往也被称为栈帧（Stack Frame）。活动记录的建立是维护运行时刻环境的重点，编译器一般也会为它生成大段的额外代码，这意味着函数调用的开销一般会很大。所以对于功能简单的函数来说，内联展开往往是一种有效的优化方法。在本任务中我们并不需要关心活动记录是如何建立的，只需要压入相应的参数然后调用 CALL 语句即可。但要注意的是， 若数组或结构体作为参数传递，需谨记数组和结构体都要求采用引用调用（Call by Refer- ence）的参数传递方式，而非普通变量的值调用（Call by Value）。

4.4.2.5 翻译模式（基本表达式）

中间代码生成的任务比较简单，你只需根据语法树产生出中间代码，然后将中间代码按照输出格式打印出来即可。中间代码如何表示以及如何打印我们都已经讨论过了，现在需要解决的问题是：如何将语法树变成中间代码？

最简单也是最常用的方式仍是遍历语法树中的每一个结点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码。和语义分析一样，中间代码的生成需要借助于实验二中我们已经提到的工具：语法制导翻译（SDT）。具体到代码上，我们可以为每个主要的语法单元“X”都设计相应的翻译函数“translate_X”，对语法树的遍历过程也就是这些函数之间互相调用的过程。每种特定的语法结构都对应了固定模式的翻译“模板”，下面我们针对一些典型的语法树结构的翻译“模板”进行说明。这些内容你也可以在课本上找到，课本上介绍的翻译模式与下面我们介绍的可能略有不同，但核心思想是一致的[1]。

我们先从语言最基本的结构表达式开始。

表 4-2 列出了与表达式相关的一些结构的翻译模式。假设我们有函数 translate_Exp()，它接受三个参数：语法树的结点 Exp、符号表 sym_table 以及一个变量名 place，并返回一段语法树当前结点及其子孙结点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。根据语法单元 Exp 所采用的产生式的不同，我们将生成不同的中间代码：

1)如果 Exp 产生了一个整数 INT，那么我们只需要为传入的 place 变量赋值成前面加上一个“#”的相应数值即可。

2) 如果 Exp 产生了一个标识符 ID，那么我们只需要为传入的 place 变量赋值成 ID 对应的变量名（或该变量对应的中间代码中的名字）即可。

3) 如果 Exp 产生了赋值表达式 Exp1 ASSIGNOP Exp2，由于之前提到过作为左值的 Exp1 只能是三种情况之一（单个变量访问、数组元素访问或结构体特定域的访问），而对于数组和结构体的翻译

模式我们将在后面讨论，故这里仅列出当 Exp1 → ID 时应该如何进行翻译。我们需要通过查表找到 ID 对应的变量，然后对 Exp2 进行翻译（运算结果储存在临时变量 t1 中），再将 t1 中的值赋于 ID 所对应的变量并将结果再存回 place，最后把刚翻译好的这两段代码合并随后返回即可。

4) 如果 Exp 产生了算术运算表达式 Exp1 PLUS Exp2，则先对 Exp1 进行翻译（运算结果储存在临时变量 t1 中），再对 Exp2 进行翻译（运算结果储存在临时变量 t2 中），最后生成一句中间代码 place := t1 + t2，并将刚翻译好的这三段代码合并后返回即可。使用类似的翻译模式我们也可以对减法、乘法和除法表达式进行翻译。

5) 如果 Exp 产生了取负表达式 MINUS Exp1，则先对 Exp1 进行翻译（运算结果储存在临时变量 t1 中），再生成一句中间代码 place := #0 − t1 从而实现对 t1 取负，最后将翻译好的这两段代码合并后返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。

6) 如果 Exp 产生了条件表达式（包括与、或、非运算以及比较运算的表达式），我们则会调用 translate_Cond 函数进行（短路）翻译。如果条件表达式为真，那么为 place 赋值 1；否则，为其赋值 0。

表 4-2 基本表达式的翻译模式

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value][2]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[3] ($Exp_1 \rightarrow$ ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] +[4] [place := variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp_1 RELOP Exp_2	label1 = new_label() label2 = new_label() code0 = [place := #0] code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]
NOT Exp_1	
Exp_1 AND Exp_2	
Exp_1 OR Exp_2	

[1] 使用模板并不是生成中间代码的唯一方法，也存在着其他方法（例如构造控制流图）可以完成翻译任务，只不过使用模板是最简单和被介绍得最为广泛的方法。

[2] 用方括号括起来的内容表示新建一条具体的中间代码。

[3] 这里Exp的下标只是用来区分产生式Exp → Exp ASSIGNOP Exp中多次重复出现的Exp。

[4] 这里的加号相当于连接运算，表示将两段代码连接成一段。

由于条件表达式的翻译可能与跳转语句有关，表中并没有明确说明 translate_Cond 该如何实现，这一点我们在后面介绍。

表 4-3 语句的翻译模式

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_CompSt(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1 + code2
IF LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP $Stmt_1$ ELSE $Stmt_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) code3 = translate_Stmt($Stmt_2$, sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

表 4-4 条件表达式的翻译模式

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp_1	return translate_Cond(Exp_1, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label1, label_false, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label_true, label1, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

4.4.2.6 翻译模式（语句）

C--的语句包括表达式语句、复合语句、返回语句、跳转语句和循环语句，它们的翻译模式如表4-3所示。

你可能注意到，无论是if语句还是while语句，表4-4中列出的翻译模式都不包含条件跳转。其实我们是在翻译条件表达式的同时生成这些条件跳转语句，translate_Cond函数负责对条件表达式进行翻译，其翻译模式如表4-4所示。

对于条件表达式的翻译，课本上已经花了较大的篇幅进行介绍，尤其是与回填有关的内容更是重点。不过，表4-4中没有与回填相关的任何内容。原因很简单：我们将跳转的两个目标label_true和label_false作为继承属性（函数参数）进行处理，在这种情况下每当我们在条件表达式内部需要跳转到外面时，跳转目标都已经从父结点那里通过参数得到了，直接填上即可。所谓回填，只用于将label_true和label_false作为综合属性处理的情况，注意这两种处理方式的区别。

表4-5 函数调用的翻译模式

translate_Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	function = lookup(sym_table, ID) if (function.name == "read") return [READ place] return [place := CALL function.name]
ID LP Args RP	function = lookup(sym_table, ID) arg_list = NULL code1 = translate_Args(Args, sym_table, arg_list) if (function.name == "write") return code1 + [WRITE arg_list[1]] + [place := #0] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name]

表4-6 函数参数的翻译模式

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA $Args_1$	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args($Args_1$, sym_table, arg_list) return code1 + code2

4.4.2.7 翻译模式（函数调用）

函数调用是由语法单元Exp推导而来的，因此，为了翻译函数调用表达式我们需要继续完善translate_Exp，如表4-5所示。

由于实验要求中规定了两个需要特殊对待的函数read和write，故当我们从符号表中找到ID对应的函数名时不能直接生成函数调用代码，而是应该先判断函数名是否为read或write。对于那些非read和write的带参数的函数而言，我们还需要调用translate_Args函数将计算实参的代码翻译出来，并构造这些参数所对应的临时变量列表arg_list。translate_Args的实现如表4-6所示。

4.4.2.8 翻译模式（数组与结构体）

C--语言的数组实现采取的是最简单的 C 风格。数组和结构体不同于一般变量的一点在于，访问某个数组元素或结构体的某个域需要牵扯到其内存地址的运算。以三维数组为例，假设有数组 int array[7][8][9]，为了访问数组元素 array[3][4][5]，我们首先需要找到三维数组 array 的首地址（直接对变量 array 取地址即可），然后找到二维数组 array[3]的首地址（array 的地址加上 3 乘以二维数组的大小（8×9）再乘以 int 类型的宽度 4），然后找到一维数组 array[3][4]的首地址（array[3]的地址加上 4 乘以一维数组的大小（9）再乘以 int 类型的宽度 4），最后找到整数 array[3][4][5]的地址（array[3][4]的地址加上 5 乘以 int 类型的宽度 4）。整个运算过程可以表示为：

$$\mathrm{ADDR}(\mathrm{array}[i][j][k]) = \mathrm{ADDR}(\mathrm{array}) + \sum_{t=0}^{i-1} \mathrm{SIZEOF}(\mathrm{array}[t]) + \sum_{t=0}^{j-1} \mathrm{SIZEOF}(\mathrm{array}[i][t]) + \sum_{t=0}^{k-1} \mathrm{SIZEOF}(\mathrm{array}[i][j][])。$$

上式很容易推广到任意维数组的情况。

结构体的访问方式与数组非常类似。例如，假设要访问结构体 struct { int x[10]; int y, z; }，st 中的域 z，我们首先找到变量 st 的首地址，然后找到 st 中域 z 的首地址（st 的地址加上数组 x 的大小（4×10）再加上整数 y 的宽度 4）。我们可以把一个有 n 个域的结构体看成为一个有 n 个元素的“一维数组”，它与一般一维数组的不同点在于，一般一维数组的每个元素的大小都是相 同的，而结构体的每个域大小可能不一样。其地址运算的过程可以表示为：

$$\mathrm{ADDR}(\mathrm{st}.field_n) = \mathrm{ADDR}(\mathrm{st}) + \sum^{n-1} \mathrm{SIZEOF}(\mathrm{st}.field_t)。$$

将数组的地址运算和结构体的地址运算结合起来也并不是太难的事。假如我们有一个结构体，该结构体的某个元素是数组，为了访问这个数组中的某个元素，我们需要先根据该数组在结构体中的位置定位到这个数组的首地址，然后再根据数组的下标定位该元素。反之，如果我们有一个数组，该数组的每个元素都是结构体。为了访问某个数组元素的某个域，我们需要先根据数组的下标定位要访问的结构体，再根据域的位置寻找要访问的内容。这个过程中唯一需要关注的是，我们应记录并区分在访问过程中使用到的临时变量哪些代表地址，哪些代表内存中的数值。如果弄错，会导致代码的运行结果出错或者非法的内存访问。当然，上述访问方式需要经历多次地址计算，如果我们能通过其它手段将这多次地址计算合并成一次，那么得到的中间代码的效率就会得到一定的提高。

4.4.2.8 中间代码生成的提示

中间代码生成需要在语义分析的基础上完成。你可以在语义分析部分添加中间代码生成的内容，使编译器可以一边进行语义检查一边生成中间代码；也可以将关于中间代码生成的所有内容写到一个单独的文件中，等到语义检查全部完成并通过之后再生成中间代码。前者会让你的编译器效率高一些，后者会让你的编译器模块性更好一些。

确定了在哪里进行中间代码生成之后，下一步就要实现中间代码的数据结构（最好能写一系列可以直接生成一条中间代码的构造函数以简化后面的实现），然后按照输出格式的要求自己编写函数将

你的中间代码打印出来。完成之后建议先自行写一个测试程序，在这个测试程序中使用构造函数人工构造一段代码并将其打印出来，然后使用我们提供的虚拟机小程序简单地测试一下，以确保自己的数据结构和打印函数都能正常工作。准备工作完成之后，再继续做下面的内容。

接下来的任务是根据前面介绍的翻译模式完成一系列的translate函数。我们已经给出了Exp和Stmt的翻译模式，你还需要考虑包括数组、结构体、数组与结构体定义、变量初始化、语法单元CompSt、语法单元StmtList在内的翻译模式。需要你自己考虑的内容实际上并不太多，最关键的一点在于一定要读懂前面介绍的几个translate函数的意思。如果读懂了，那么无论是将它们实现到你的编译器中还是书写新的translate函数，或者是对这些translate函数进行改进，都将是比较容易的。如果没读懂，例如没想明白某些translate函数中出现的place参数究竟有什么用，那么建议你还是先别急着动手写代码。如果顺利完成了所有translate函数，并将其连同中间代码的打印函数添加到了你的编译器中，那么中间代码生成的要求就差不多完成了。建议在这里多写几份测试用例检查你的编译器，如果发现了错误需要及时纠正。

最后，虚拟机小程序将以总共执行过的中间代码条数为标准来衡量你的编译器所输出的中间代码的运行效率。因此如果要进行代码优化，重点应该放在精简代码逻辑以及消除代码冗余上。

附录 A　C--语言文法

在本附录中，我们给出 C--语言的文法定义和补充说明。

A.1 文法定义

A.1.1　Tokens

INT→ /* A sequence of digits without spaces[1] */

FLOAT →/* A real number consisting of digits and one decimal point. The decimal point must be surrounded by at least one digit[2] */

ID → /* A character string consisting of 52 upper- or lower-case alphabetic, 10 numeric and one underscore characters. Besides, an identifier must not start with a digit[3] */

SEMI → ;

COMMA → ,

ASSIGNOP→ =

RELOP → > | < | >= | <= | == | !=

PLUS → +

MINUS → -

STAR → *

DIV → /

AND → &&

OR → ||

DOT → .

NOT → !

TYPE →int | float

LP → (

RP →)

LB → [

RB →]

LC → {

RC → }

STRUCT → struct

1 你需要自行考虑如何用正则表达式表示整型数，你可以假设每个整型数不超过 32bits 位。

2 你需要自行考虑如何用正则表达式表示浮点数，你可以只考虑符合 C 语言规范的浮点常数。

3 你需要自行考虑如何用正则表达式表示标识符，你可以假设每个标识符长度不超过 32 个字符。

RETURN → return

IF → if

ELSE → else

WHILE→while

A.1.2 High-level Definitions

Program → ExtDefList

ExtDefList → ExtDef ExtDefList

| ε

ExtDef → Specifier ExtDecList SEMI

| Specifier SEMI

| Specifier FunDec CompSt

ExtDecList → VarDec

| VarDec COMMA ExtDecList

A.1.3 Specifiers

Specifier → TYPE

| StructSpecifier

StructSpecifier → STRUCT OptTag LC DefList RC

| STRUCT Tag

OptTag →ID

| ε

Tag → ID

A.1.4 Declarators

VarDec → ID

| VarDec LB INT RB

FunDec → ID LP VarList RP

| ID LP RP

VarList → ParamDec COMMA VarList

| ParamDec

ParamDec → Specifier VarDec

A.1.5 Statements

CompSt → LC DefList StmtList RC

StmtList →Stmt StmtList

| ε

Stmt → Exp SEMI

| CompSt

| RETURN Exp SEMI

| IF LP Exp RP Stmt

| IF LP Exp RP Stmt ELSE Stmt

| WHILE LP Exp RP Stmt

A.1.6 Local Definitions

DefList →Def DefList

| ε

Def → Specifier DecList SEMI

DecList→ Dec

| Dec COMMA DecList

Dec →VarDec

| VarDec ASSIGNOP Exp

A.1.7 Expressions

Exp → Exp ASSIGNOP Exp

| Exp AND Exp

| Exp OR Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp MINUS Exp

| Exp STAR Exp

| Exp DIV Exp

| LP Exp RP

| MINUS Exp

| NOT Exp

| ID LP Args RP

| ID LP RP

| Exp LB Exp RB

| Exp DOT ID

| ID

| INT

| FLOAT

Args→Exp COMMA Args

| Exp

A.2 补充说明

A.2.1 Tokens

这一部分的产生式主要与词法有关：

1) 词法单元 INT 表示的是所有（无符号）整型常数。一个十进制整数由 0~9 十个数字组成，数字与数字中间没有如空格之类的分隔符。除“0”之外，十进制整数的首位数字不为 0。例如，下面几个串都表示十进制整数：0、234、10000。为方便起见，你可以假设（或者只接受）输入的整数都在 32bits 位之内。

2) 整型常数还可以以八进制或十六进制的形式出现。八进制整数由 0~7 八个数字组成并以数字 0 开头，十六进制整数由 0~9、A~F（或 a~f）十六个数字组成并以 0x 或者 0X 开头。例如，0237（表示十进制的 159）、0xFF32（表示十进制的 65330）。

3) 词法单元 FLOAT 表示的是所有（无符号）浮点型常数。一个浮点数由一串数字与一个小数点组成，小数点的前后必须有数字出现。例如，下面几个串都是浮点数：0.7、12.43、9.00。为方便起见，你可以假设（或者只接受）输入的浮点数都符合 IEEE754 单精度标准（即都可以转换成 C 语言中的 float 类型）。

4) 浮点型常数还可以以指数形式（即科学记数法）表示。指数形式的浮点数必须包括基数、指数符号和指数三个部分，且三部分依次出现。基数部分由一串数字（0~9）和一个小数点组成，小数点可以出现在数字串的任何位置；指数符号为“E”或“e”；指数部分由可带“+”或“- ”（也可不带）的一串数字（0~9）组成，“+”或“- ”（如果有）必须出现在数字串之前。例如 01.23E12（表示 1.23×10^{12}）、43.e-4（表示 43.0×10^{-4}）、.5E03（表示 0.5×10^{3}）。

5) 词法单元 ID 表示的是除去保留字以外的所有标识符。标识符可以由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。为方便起见，你可以假设（或者只接受）标识符的长度小于 32 个字符。

6) 除了 INT、FLOAT 和 ID 这三个词法单元以外，其它产生式中箭头右边都表示具体的字符串。例如，产生式 TYPE → int | float 表示：输入文件中的字符串“int”和“float”都将被识别为词法单元 TYPE。

A.2.2 High-level Definitions

这一部分的产生式包含了 C--语言中所有的高层（全局变量以及函数定义）语法：

1) 语法单元 Program 是初始语法单元，表示整个程序。

2) 每个 Program 可以产生一个 ExtDefList，这里的 ExtDefList 表示零个或多个 ExtDef（像这种 xxList

表示零个或多个 xx 的定义下面还有不少，要习惯这种定义风格）。

3) 每个 ExtDef 表示一个全局变量、结构体或函数的定义。其中：

a) 产生式 ExtDef →Specifier ExtDecList SEMI 表示全局变量的定义，例如“int global1, global2;”。其中 Specifier 表示类型，ExtDecList 表示零个或多个对一个变量的定义 VarDec。

b) 产生式 ExtDef →Specifier SEMI 专门为结构体的定义而准备，例如“struct {…};”。这条产生式也会允许出现像“int;”这样没有意义的语句，但实际上在标准 C 语言中这样的语句也是合法的。所以这种情况不作为错误的语法（即不需要报错）。

c) 产生式 ExtDef →Specifier FunDec CompSt 表示函数的定义，其中 Specifier 是返回类型，FunDec 是函数头，CompSt 表示函数体。

A.2.3 Specifiers

这一部分的产生式主要与变量的类型有关：

1) Specifier 是类型描述符，它有两种取值，一种是 Specifier→ TYPE，直接变成基本类型 int 或 float，另一种是 Specifier → StructSpecifier，变成结构体类型。

2) 对于结构体类型来说：

a) 产生式 StructSpecifier → STRUCT OptTag LC DefList RC：这是定义结构体的基本格式，例如 struct Complex { int real, image; }。其中 OptTag 可有可无，因此也可以这样写：struct { int real, image; }。

b) 产生式 StructSpecifier → STRUCT Tag：如果之前已经定义过某个结构体，比如 struct Complex {…}，那么之后可以直接使用该结构体来定义变量，例如 struct Complex a, b;而不需要重新定义这个结构体。

A.2.4 Declarators

这一部分的产生式主要与变量和函数的定义有关：

1) VarDec 表示对一个变量的定义。该变量可以是一个标识符（例如 int a 中的 a），也可以是一个标识符后面跟着若干对方括号括起来的数字（例如 int a[10][2]中的 a[10][2]，这种情况下 a 是一个数组）。

2) FunDec 表示对一个函数头的定义。它包括一个表示函数名的标识符以及由一对圆括号括起来的一个形参列表，该列表由 VarList 表示（也可以为空）。VarList 包括一个或多个 ParamDec，其中每个 ParamDec 都是对一个形参的定义，该定义由类型描述符 Specifier 和变量定义 VarDec 组成。例如一个完整的函数头为：foo(int x, float y[10])。

A.2.5 Statements

这一部分的产生式主要与语句有关：

1) CompSt 表示一个由一对花括号括起来的语句块。该语句块内部先是一系列的变量定义 DefList，然后是一系列的语句 StmtList。可以发现，对 CompSt 这样的定义，是不允许在程序的任意位置定义变

量的，必须在每一个语句块的开头才可以定义。

2) StmtList 就是零个或多个 Stmt 的组合。每个 Stmt 都表示一条语句，该语句可以是一个在末尾添了分号的表达式（Exp SEMI），可以是另一个语句块（CompSt），可以是一条返回语句（RETURN Exp SEMI），可以是一条 if 语句（IF LP Exp RP Stmt），可以是一条 if-else 语句（IF LP Exp RP Stmt ELSE Stmt），也可以是一条 while 语句（WHILE LP Exp RP Stmt）。

A.2.6 Local Definitions

这一部分的产生式主要与局部变量的定义有关：

1) DefList 这个语法单元前面曾出现在 CompSt 以及 StructSpecifier 产生式的右边，它就是一串像 int a; float b, c; int d[10];这样的变量定义。一个 DefList 可以由零个或者多个 Def 组成。

2) 每个 Def 就是一条变量定义，它包括一个类型描述符 Specifier 以及一个 DecList，例如 int a, b, c;。由于 DecList 中的每个 Dec 又可以变成 VarDec ASSIGNOP Exp，这允许我们对局部变量在定义时进行初始化，例如 int a = 5;。

A.2.7 Expressions

这一部分的产生式主要与表达式有关：

1) 表达式可以演化出的形式多种多样，但总体上看不外乎下面几种：

   a) 包含二元运算符的表达式：赋值表达式（Exp ASSIGNOP Exp）、逻辑与（Exp AND Exp）、逻辑或（Exp OR Exp）、关系表达式（Exp RELOP Exp）以及四则运算表达式（Exp PLUS Exp 等）。

   b) 包含一元运算符的表达式：括号表达式（LP Exp RP）、取负（MINUS Exp）以及逻辑非（NOT Exp）。

   c) 不包含运算符但又比较特殊的表达式：函数调用表达式（带参数的 ID LP Args RP 以及不带参数的 ID LP RP）、数组访问表达式（Exp LB Exp RB）以及结构体访问表达式（Exp DOT ID）。

   d) 最基本的表达式：整型常数（INT）、浮点型常数（FLOAT）以及普通变量（ID）。

2) 语法单元 Args 表示实参列表，每个实参都可以变成一个表达式 Exp。

3) 由于表达式中可以包含各种各样的运算符，为了消除潜在的二义性问题，我们需要给出这些运算符的优先级（precedence）以及结合性（associativity），如表 A-1 所示。

A.2.8 Comments

C--源代码可以使用两种风格的注释：一种是使用双斜线“//”进行单行注释，在这种情况下，该行在“//”符号之后的所有字符都将作为注释内容而直接被词法分析程序丢弃掉；另一种是使用“/*”以及“*/”进行多行注释，在这种情况下，在“/*”与之后最先遇到的“*/”之间的所有字符都被视作注释内容。需要注意的是，“/*”与“*/”是不允许嵌套的：即在任意一对“/*”和“*/”之间不能再包含成对的“/*”和“*/”，否则编译器需要进行报错。

表 A-1 C--语言中运算符的优先级和结合性

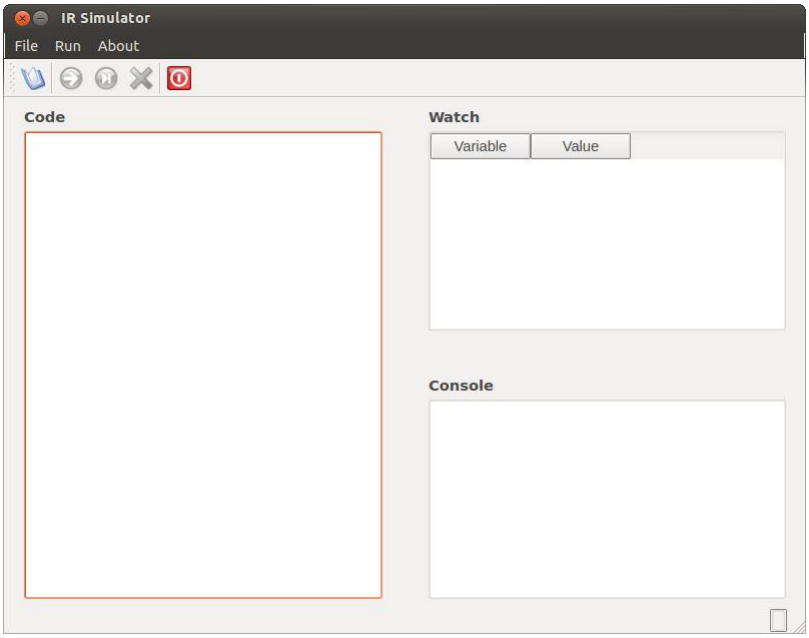
优先级[1]	运算符	结合性	描述
1	(,)	左结合	括号或函数调用
	[,]		数组访问
	.		结构体访问
2	-	右结合	取负
	!		逻辑非
3	*	左结合	乘
	/		除
4	+		加
	-		减
5[2]	<		小于
	<=		小于或等于
	>		大于
	>=		大于或等于
	==		等于
	!=		不等于
6	&&		逻辑与
7	||		逻辑或
8	=	右结合	赋值

1 数值越小表示代表优先级越高。

2 在标准 C 语言中，“>”、“<”、“>=”和“<=”四个关系运算符的优先级要比“==”和“!=”两个运算符的优先级高；在这里，我们为了方便处理而将它们的优先级统一了。

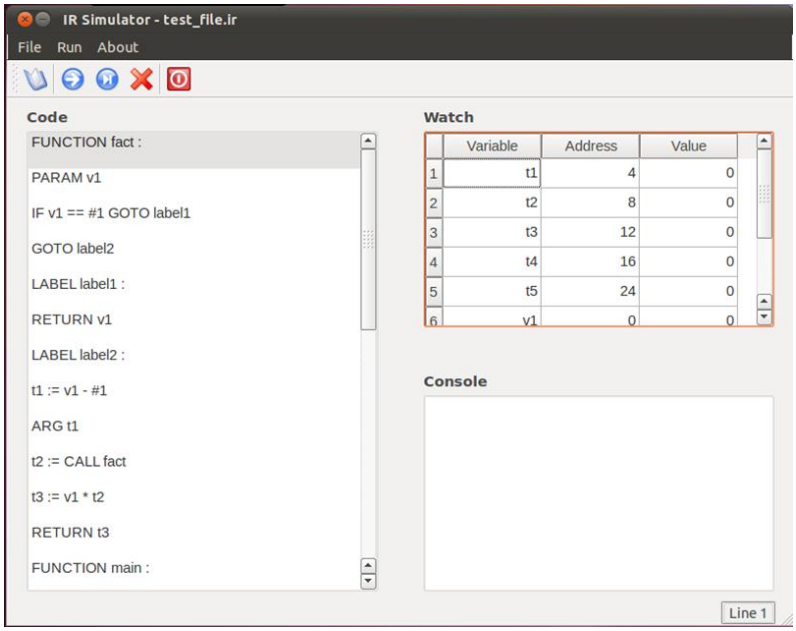
附录B 虚拟机小程序使用说明

我们提供的虚拟机小程序叫做IR Simulator，它本质上就是一个中间代码解释器。这个程序使用Python写成，图形界面部分则借助了跨平台的诺基亚Qt库。由于实验环境为Linux，因此该程序只在Linux下发布。注意，运行虚拟机小程序之前要确保本机上装有Qt运行环境。你可以在Linux终端下使用“sudo apt-get install python-qt4”命令以获取Qt运行环境（也可参考附录C中的离线安装方式），然后通过我们课程网站下载虚拟机小程序（请参考附录C中的资源下载和安装介绍）。



图B-1. 虚拟机小程序IR Simulator的界面

IR Simulator的运行界面如图B-1所示。整个界面分为三个部分：左侧的**代码区**、右上的**监视区**和右下方的**控制台区**。代码区显示已经载入的中间代码，监视区显示中间代码中当前执行函数的参数与局部变量的值，控制台区供WRITE函数进行输出用。我们可以点击上方工具栏中的第一个按钮或通过菜单选择File　Open来打开一个保存有中间代码的文本文件（注意该文件的后缀名必须是ir）。如果中间代码中包含语法错误，则IR Simulator会弹出对话框提示出错位置并拒绝继续载入代码；如果中间代码中没有错误，那么你将看到类似于图B-2所示的内容。此时中间代码已被完全载入到左侧的代码区，因为中间代码尚未执行，所以右侧监视区的显示为空。此时你可以单击工具栏上的第二个按钮或通过菜单选择Run　Run（快捷键为F5）直接运行该中间代码，或者单击第三个按钮或通过菜单选择Run　Step（快捷键为F8） 来对该中间代码进行单步执行。单击第四个按钮或通过菜单选择Run　Stop可以停止中间代码的运行并重新初始化。如果中间代码的运行出现错误，那么IR Simulator会弹出对话框以提示出错的行号以及原因（一般的出错都是因为访问了一个不存在的内存地址，或者在某个函数中缺少RETURN语句等）；如果一切正常，你将会看到如图B-3所示的对话框。这提示我们中间代码的运行已经正常结束，并且本次运行共执行了85条指令。由于实验四会考察优化中间代码的效果，如果你想得到更高的评价，就需要设法使Total instructions后面的数字尽可能小。



图B-2. IR Simulator载入中间代码成功的界面

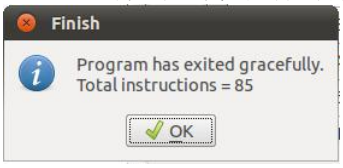


图 B-3. IR Simulator 运行中间代码成功的提示

另外，关于IR Simulator有几点需要注意：

1) 在运行之前请确保你的中间代码不会陷入死循环或无穷递归中。IR Simulator不会对这类情况进行判断，因此一旦当它执行了包含死循环或无穷递归的中间代码，你就需要将IR Simulator强制退出了。

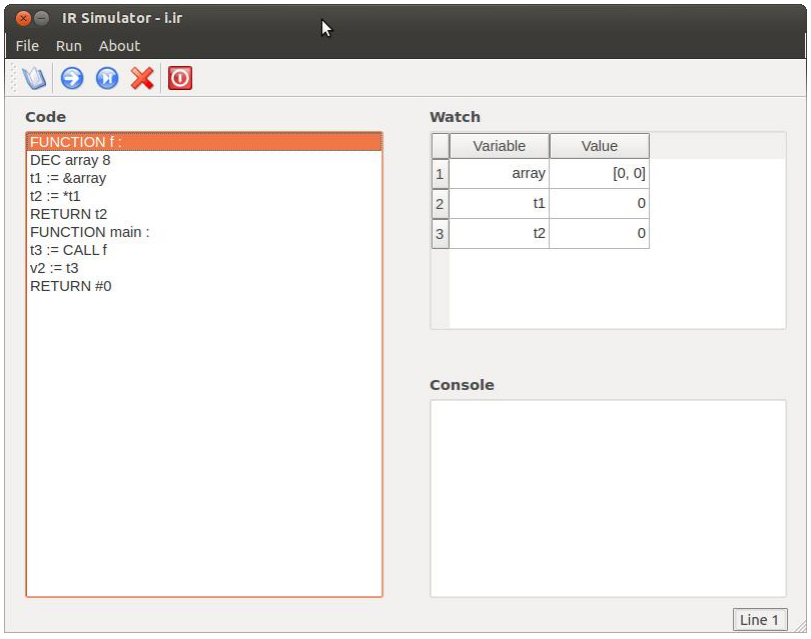
2) 互相对应的ARG语句和PARAM语句数量一定要相等，否则可能出现无法预知的错误。

3) 由于IR Simulator实现上的原因，单步执行时它并不会在GOTO、IF、RETURN等与跳转相关的语句上停留，而是会将控制直接转移到跳转目标那里，这是正常的情况。

4) 在中间代码的执行过程中，右侧监视区始终显示当前正在执行函数的中间代码中的变量及其值。如图B-4中所示，当执行到函数“f”的第一条代码时，右侧监视区中会显示此函数对应的中间代码中使用的变量，且变量的初始值默认为0。

5) 使用DEC语句定义的变量在监视区中的显示与普通的变量的区别如图B-4所示。注意监视区中变量array的值：与其它变量不同，array中的内容被一对中括号“[”和“]”包裹了起来。有时候，因为DEC出来的变量内容较多，Value一栏可能无法显示完全，你可以用鼠标调整Value栏的宽度使其所有内容都能被显示出来。

6) 所有函数的参数与局部变量都只在运行到该函数之后才会去为其分配存储空间并在监视区显示。存储空间采用由低地址到高地址的栈式分配方式，递归调用的局部变量将不会影响到上层函数的相应变量的值。如果想要修改上层函数中变量的值，则需要向被调用的函数传递该变量的地址作为参数，也就是我们常说的引用调用。不过，监视区中显示的变量总是当前这一层变量的值，只要不退回到上一层，我们就无法看到上层函数局部变量的值是多少。



图B-4. DEC定义的变量与普通变量之间的显示区别

附录 C 资源下载和安装介绍

在本附录中，我们介绍编译实验环境的搭设。有两种方式：一是在连接 Internet 的条件下进行在线软件安装；二是下载安装软件后再进行离线软件安装。

C.1 在线软件安装方式

在线软件安装的步骤如下:

1)在 Ubuntu 官方网站上下载 Ubuntu12.04 操作系统并进行安装，其下载地址为：http://old-releases.ubuntu.com/releases/12.04.2/，文件名为 ubuntu-12.04.2-desktop-i386.iso。

2)在 Ubuntu 终端使用命令“sudo apt-get install flex 以安装 flex 软件。安装完成后,可使用命令“flex --version”来查看 flex 的版本号(目前版本为“flex 2.5.35”)。

3)在 Ubuntu 终端使用命令“sudo apt-get install bison 以安装 bison 软件。安装完成后,可使用命令“ bison --version”来查看 bison 的版本号(目前版本为“bison 2.5”)

4)在 Ubuntu 终端使用命令“sudo apt-get install python-qt4”以安装 Qt 运行环境。

5)从我们的课程网站上下载我们提供的虚拟机小程序(用于中间代码的执行)，其下载地为 http://cs.nju.edu.cn/changxu/compiler/irsim.tar.gz。将下载得到的压缩包中的三个文件解压到同一用户目录下，并在该目录下执行“python irsim.pyc 即可启动虚拟机小程序。

6)从 SPIM Simulator 的官方网站上下载 QtSPIM 软件(用于执行目标代码)，其下载地址为 http://pages.cs.wisc.edu/~larus/spim.html，文件名为：qtspim_9.1.9_linux32.deb。双击该文件以进行安装，在安装过程中会弹出“The package is of bad quality”的提示框，这时选择“Ignore and install”可继续安装。

如果以上软件的版本号与说明的不完全一致，在大多数情况下并不会影响编译实验的进行。如果遇到特殊情况或需要完全一致，也可参照下面的离线软件安装。

C.2 离线软件安装方式

离线软件安装的步骤如下:

1）在“/资源/ubuntu 12.04.2/”目录下获得 ubuntu-12.04.2-desktop-i386.iso 文件，完成 Ubuntu12.04 操作系统的安装。

2）在“/资源/flex 2.5.35”首先双击 libfl-dev_2.5.35-10.1ubuntu2_i386.deb 文件以安装 flex 的运行库，然后双击 flex_2.5.35-10.1ubuntu2_i386.deb 文件以安装 flex 2.5.35 软件。

3）在“/资源/bison 2.5/ ”目录下，双击 bison_2.5.dfsg-2.1_i386.deb 文件以安装 bison 2.5 软件。

4）在“/资源/python-qt 4”目录下，双击 python-qt4_4.9.1-2ubuntu1_i386.deb 文件以安装 Qt 运

行环境。

5）在“/资源/irsim/”目录下，获得我们的虚拟机小程序压缩包 irsim.tar.gz，将其中的三个文件解压到同一用户目录下，并在该目录下执行“python irsim.pyc”即可启动虚拟机小程序。

6）在“/资源/qtSPIM 9.1.9”目录下，双击 qtspim_9.1.9_linux32.deb 文件以安装 QtSPIM 9.1.9 模拟器，在安装过程中会弹出“The package is of bad quality”的提示框，这时选择“Ignore and install”可继续安装。