第 2 章：程序语言基础知识　　作者：希赛教育软考学院　来源：希赛网　2014年01月26日

机器语言与汇编语言

2.1.1 机器语言与汇编语言

每一种特定型号的计算机系统都有自己特定的机器指令集合，集合中每条指令都代表一项具体的操作，例如从内存取数据到寄存器。这个机器指令集合就是机器语言，由机器语言编写的程序就称为机器程序。机器指令本质上是一个特定长度的二进制串，特定的位表示操作码，而另外的位表示操作数。

由于机器程序都是由二进制的机器指令组成的，在编写机器程序的时候，不仅要记住特定操作码的二进制表示，还需要记下各个数据的地址的二进制表示。这是十分不方便的，而且容易出错，程序也很难读懂。于是人们就开始使用助记符（汇编指令）代表机器指令的操作码，并且使用伪指令（即不对应任何机器指令，只用于助记）和标号帮助确定数据或代码的位置，这就是汇编语言了。由于汇编指令和机器指令是相对应的，所以每种特定型号的计算机系统都有自己的汇编指令集合。

由汇编指令编写的程序就是汇编程序，计算机是不能直接执行汇编程序的，而必须由一个特殊程序根据伪指令的控制把汇编程序转化为对应的机器语言程序。这个特殊的程序就是汇编程序。

上一节　本书简介　下一节

第 2 章：程序语言基础知识　　作者：希赛教育软考学院　来源：希赛网　2014年01月26日

汇编程序

2.1.2 汇编程序

如前面所述，汇编程序的基本工作包括：

将每一条可执行汇编指令转换成对应的机器指令；

处理源程序中出现的伪指令。

这一工作通常需要对汇编程序进行超过一次的扫描。

前面的分析已经指出，形成操作数地址的各个部分有可能出现符号，而符号是稍后语句的标号：

```
SUB   1,C48
…
C48   DC   48
```

为了计算各汇编语句中标号的地址，我们在汇编程序中设立单元地址计数器LC,其初值一般为0.以后每处理完一条可执行的汇编语句和与存储分配有关的伪指令（如定义常数语句、定义存储语句），LC的值就增加相应长度，这样LC的值始终是下一个存储单元的相对地址。当处理一条汇编语

句标号时，就将LC当时的值定义为标号值。由于符号使用可能出现在符号定义前，整个汇编程序工作要通过对源程序进行二次扫描才能完成。

第一次扫描主要工作是定义符号的值。除了设置单元计数器LC外，我们设立机器指令表MOT1.由于本次扫描并不具体生成机器指令，MOT1的每一元素只需两个域：机器指令记忆码和机器指令长度。在扫描过程中，我们将符号及其值记录在符号表ST中。此外，在第一次扫描中，还需要对与定义符号值有关的伪指令进行处理。为了叙述方便，不妨设立伪指令表POT1,POT1表的每一个元素只有两个域：伪指令记忆码和相应处理子程序入口。下面是对第一次扫描的描述。

①单元计数器LC置初值0.

②打开源程序文件。

③反复执行：

从源程序文件读下一条语句；

如果该语句有标号，则将标号和LC当时值送符号表ST;

根据语句操作码，执行：

如果是可执行汇编语句，K是查MOT1表所得机器指令长度，则LC:=LC+K;

如果是伪指令记忆码，则调用POT1表相应元素所规定的子程序；

如果是非法记忆码，则调用出错子程序。

直至语句操作码是END为止。

④关闭源程序文件。

第二次扫描的目的是产生目标程序。除了前一次扫描所生成的符号表ST外，我们需要建立机器指令表MOT2,该元素包含下面区域：机器指令记忆码，机器指令的二进制操作码（binary_code），格式指示（type）和长度（length）。我们还设立第二次扫描的伪指令表POT2,它的每一元素仍是两个区域：伪指令记忆码和相应处理子程序入口。所不同的是，在第二次扫描中，伪指令有着完全不同的处理。

在第二次扫描中，可执行汇编语句应被翻译成对应的二进制代码机器指令。这一工作涉及两个方面：把机器指令记忆码转换成二进制机器指令操作码，以及求出操作数区各操作数的值（用二进制数表示）。在此基础上，可以装配出二进制代码的机器指令。对于第一部分工作，只要根据机器指令记忆码查机器指令表MOT2,就可以获得相应二进制数表示的机器指令操作码。从求值的角度来说，第二部分工作并不复杂。由于形成内存操作数地址的各个部分都以表达式的形式出现，我们统一定义一个过程eval-expr（index,value）。调用时，只要将表达式在汇编语句缓冲区S开始位置通过index传递给此过程，该过程就通过value返回此表达式的值。例如，虚拟计算机COMET的机器指令可归属于"X"型指令，其汇编语句为：

OP R1,N2,X2

OP R1,N2

我们可以写出下面处理"X"型指令的程序段（假定index已指向操作数在缓冲区S的首址）：

```
eval_expr（index,R1）；
index:=index+1;
eval_expr（index,N2）；
if S [index]=','then
begin
```

index:=index+1;

eval_expr（index,X2）

end

else

X2:=0;

其他类型指令的处理操作数的程序段都可以类似地写出。设当前可执行汇编语句的操作记忆码在MOT2表的索引值为i,则整个可执行汇编语句的处理可以描述如下：

OP:=MOT2 [i].binary_code;

TYPE:=MOT2 [i].type;

case TYPE of

'X':求X型指令操作数各个部分值，然后按规定字节形成指令；

…

end;

将形成指令送往输出区；

在第二次扫描中，DS伪指令的主要目的是保留存储空间。我们不妨设立一个工作单元k,用于累计以字节为单位的存储空间大小，k初值为0.从DS伪指令的操作数区求出k的大小后，就向输出区送k个空格以达到保留所规定存储单元的目的。DC伪指令处理和DS伪指令类似，只不过向输出区送的是所转换得到的常量。最后，START伪指令工作可能是输出目标程序开始的标准信息，而END伪指令则可能是输出目标程序结束的标准信息，这些信息都是为装配程序提供的。

装配程序

2.1.3 装配程序

装配程序也称为连接程序，该程序主要完成以下两个任务。

1.装入

装入是指读入可重定位的机器代码，修改重定位的地址，把修改后的指令和数据放在内存的适当位置或者形成可执行文件。

2.连接

连接是把几个可重定位的机器代码文件连接成一个可执行程序，这些文件可以是分别汇编得到的，也可以是系统提供的程序库机器代码。

这种装配我们称为相对装配。装配程序从操作系统得到整个用户程序的装入起始地址，汇编程序第一次扫描结束时，本程序段长度已经求出。因此，我们可以求出每个程序段的起始装入地址，并进一步求出由PUBLIC伪指令定义的每一符号的实际存储地址。另一方面，汇编程序虽然无法知道外名（如SUM）的实际地址，但它却可以知道本程序哪一个存储单元（如ASUM）需要修改，并通过伪指令EXTERN说明哪些外名的值是本程序段修改地址常数时需要的。

为了能使相对装配程序获得必要信息，汇编程序对每一程序段输出下面次序目标块。

段名定义块：该块的信息包括程序段段名和程序段长度。

整体符号定义块：程序块中定义的入口名及它在程序段中的相对地址。

正文块：其信息包括正文第一个字节装入的相对地址、正文字节数和正文本身。

地址常数块：地址常数所依赖的外名，地址常数在程序段中的相对地址。

结束块：其信息为启动地址或0.

装配程序的工作通过对各程序段的目标块进行二次扫描来完成。在第一次扫描中，它只处理段名定义块和整体符号定义块，为段名和各段的入口名分配地址，建立整体符号表。在第二次扫描中，当正文全部复写到所分配的内存后，就可以根据地址常数块的信息对地址常数所分配单元中的值进行修正。最后根据结束块中的启动地址，执行装配好的目标程序。

上一节　本书简介　下一节

宏指令

2.1.4 宏指令

用汇编语言进行程序设计时，用户经常会书写完全相同或类似的语句，为了程序设计方便，汇编程序往往向用户提供宏指令技术。按照这一技术，程序员可以相当自由地将一组汇编语句定义成一条新指令--宏指令。宏指令一经定义，用户就可以在程序段其他地方书写这条指令，而将宏指令替换成原来指令序列的工作留给宏指令处理程序去完成。

在使用宏指令技术时，用户应该先进行宏定义，将宏指令和一串指令序列联系起来。各种汇编语言在宏指令定义开始和结束的规定上存在着微小差异。IBM公司PC机上宏指令定义开始语句是：

宏指令名 MACRO 哑元表

而宏指令定义结束语句是：

ENDM

其间是一个指令序列。宏指令定义开始语句的哑元表可以默认。宏指令定以后，用户就可以直接在程序中书写宏指令，称为宏调用。如果宏指令定义带有哑元表，则用户在宏调用时必须提供替换相应哑元的变量信息。宏指令处理程序对宏指令定义开始语句和结束语句之间的语句一一进行必要的变量信息替换，然后依次插入宏指令调用处，这一过程称为宏指令展开。

宏指令处理程序的实现并不复杂，可建立宏定义表MDT,用来专门保存宏指令定义开始和结束之间的语句序列。另外，再设立宏定义名表MNT,用来建立宏指令名和它在MDT表第一条语句的对应关系。在宏指令展开时，只要根据宏指令名查宏定义名表MNT,就可以得到第一条语句的位置，并从这一条语句开始，对宏定义中所有语句依次进行变量替换，然后送到相应位置上。

上一节　本书简介　下一节

编译系统基本原理

2.2 编译系统基本原理

本节将介绍编译系统基本原理。

上一节　本书简介　下一节

编译概述

2.2.1 编译概述

编译程序的职能是把使用某程序设计语言书写的程序翻译为等价的机器语言程序，所谓等价是指目标程序执行源程序的预定任务。一般来说，编译程序分为以下几个部分：词法分析，语法分析和语义分析，代码优化，代码生成和符号表管理。各部分之间的关系如图2-2所示。

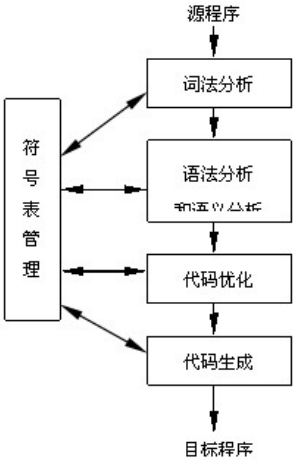


图 2-2　编译程序结构框图

词法分析程序是编译程序的第一个部分，它的输入是源程序中由字符组成的符号。编译程序需要把程序的这种外部形式转换成适合后续程序处理的形式，其功能如下。

识别出源程序中意义独立的最小词法单位--单词，并且确定其类型（例如是标识符、关键字、操作符还是数字等）。

删除无用的空格、回车和其他与输入介质有关的无用符号，以及程序注释。

报告分析时的错误。

经过词法分析程序处理后，源程序就转化为单词串。每个单词都是一个意义独立的单位，其所包含的信息量个数固定。语法分析程序根据特定程序设计语言的文法规则，检查单词串是否符合这些规则。一旦语法分析程序分解出其中一个文法结构，该结果的语义分析程序就进行相应的语义检查，在有需要的时候输出相应的中间代码。这里的中间代码可以理解为假想的虚拟机的指令，其执行次序反映了源程序的原始定义。语法和语义分析程序是编译程序中的关键部分。

中间代码作为代码生成程序的输入，由代码生成程序生成特定的计算机系统下的机器代码。为

了提高目标代码的运行效率和减小目标代码大小，也可以在语法语义分析程序与代码生成程序之间插入代码优化程序。代码优化程序在不改变代码所完成的工作的前提下对中间代码进行改动，使其变成一种更有效的形式。

编译程序在完成其任务的过程中，还需要进行符号表的管理和出错处理。在符号表中登记了源程序中出现的每一个标识符及其属性。在整个编译过程中，各部分程序都可以访问某标识符的属性，包括标识符被说明的类型、数组维数、所需存储单元数，所分配的内存单元地址等。错误管理程序是在分析程序发现源程序有错误而无法继续工作时进行其工作的。其任务是记录并向用户报告错误及其类型和位置，或者尝试进行某种恢复工作。

形式语言基本知识

2.2.2 形式语言基本知识

首先介绍关于字母表和符号串的定义。

无论是自然语言还是形式语言，均是由特定的符号，如字母、数字等组合而成的，符号的非空有限集合被称为字母表。由某一字母表中的符号组成的有限符号序列称为该字母表的符号串。符号串α的长度是指α中出现的符号个数，记为|α|.空串的长度为0,用ε表示。

符号串α的前缀是指α的末尾删除零个或多个符号后得到的符号串，如pro是program的一个前缀。符号串α的后缀是指α的开头删除0个或多个符号后得到的符号串，如gram是符号串program的一个后缀。符号串α的子串是删除了α的前缀和后缀后得到的符号串，如og是program的子串，α的前缀和后缀都是它的子串。对于任意符号串α，其自身和ε都是α的前缀、后缀，也是α的子串。符号串α的真前缀、真后缀和真子串是指除空串ε和α自身外，α的前缀、后缀和子串。

符号串α的子序列是从α删除0个或多个符号（这些符号不要求是连续的）而得到的符号串。

下面介绍符号串之间的运算。

符号串α、β的连接αβ是指把β写在α的后面得到的符号串，从空串的定义可以推出εα=αε=α。符号串α的方幂αn定义为αα…α（n个），由αε，α1=α。

术语"语言"表示某个确定的字母表上符号串的任何集合。空集合{ }和只包含空串的集合{ε}也是符合定义的语言。在字符串运算的基础上，我们可以定义语言的运算：

①语言L和M的合并，$\mathrm{L}\cup\mathrm{M}=\{s\mid s\in \mathrm{L}\text{ 或 }s\in \mathrm{M}\}$

②语言L和M的连接，$\mathrm{LM}=\{st\mid s\in \mathrm{L},t\in \mathrm{M}\}$

③语言L的Kleene闭包，$\mathrm{L}^{*}=\bigcup_{i=0}^{\infty}\mathrm{L}^{i}=\mathrm{L}^{0}\cup\mathrm{L}^{1}\cup\mathrm{L}^{2}\cdots$

④语言L的正闭包，$\mathrm{L}^{+}=\bigcup_{i=1}^{\infty}\mathrm{L}^{i}=\mathrm{L}^{1}\cup\mathrm{L}^{2}\cup\mathrm{L}^{3}\cdots$

上面对语言的定义是非形式化的，下面要介绍形式化的语言定义，这里首先引入文法的概念。

所谓文法G是一个四元组，$G=\{\mathrm{V}_{T},\mathrm{V}_{N},\mathrm{S},\mathrm{P}\}$.其中$\mathrm{V}_{\mathrm{T}}$是一个非空有限的符号集合，它的每个元素成为终结符号。V_{N}也是一个非空有限的符号集合，它的每个元素称为非终结符号，并且有

$V_T \cap V_N = \Phi$。$S \in V_N$，称为文法G的开始符号。P是一个非空有限集合，它的元素称为产生式。所谓产生式，其形式为α→β，α称为产生式的左部，β称为产生式的右部，符号"→"表示"定义为",并且α、$\beta \in (V_T \cup V_N)^*$,α≠ε，即α、β是由终结符和非终结符组成的符号串。开始符S必须至少在某一产生式的左部出现一次。另外可以对形如α→β，α→γ的产生式缩写为α→β|γ，以方便书写。

1956年，著名的语言学家Noam Chomsky首先对形式语言进行了描述，把文法定义为四元组，并且根据对产生式所施加的限制的不同，把文法分成了4类，并定义了相应的4类形式语言。表2-1描述了4类文法，及其产生的语言。

表2-1 文法的类型

文 法 类 型	产生式的限制	文法产生的语言
0型文法	α→β 其中α、$\beta \in (V_T \cup V_N)^*$，$\|\alpha\| \neq 0$	0型语言
1型文法	α→β 其中α、$\beta \in (V_N \cup V_T)^*$，但需$\|\alpha\| \leqslant \|\beta\|$	1型语言，即上下文有关语言
2型文法	A→β 其中$A \in V_N$，$\beta \in (V_N \cup V_T)^*$	2型语言，即上下文无关语言
3型文法	A→α|αB（右线性）或A→α|Bα（左线性） 其中，$A, B \in V_N$，$\alpha \in V_T \cup \{\varepsilon\}$	3型语言，即正规语言，又分为左线性语言和右线性语言

对于文法G[S],我们称αAβ直接推导出αγβ（也可以说αγβ是αAβ的直接推导），仅当A→γ是文法G的一个产生式，且α，$\beta \in (V_T \cup V_N)^*$，记做αAβαγβ。如果存在直接推导序列：$\alpha \Rightarrow 1 \Rightarrow \alpha \Rightarrow 2 \ldots \alpha n$,我们称该序列为α0到αn的长度为n的推导，也称为α0可以推导出αn,记做$\alpha 0 \overset{+}{\Rightarrow} \alpha n$.如果n=0,即α0=αn或α0αn,则记为$\alpha 0 \overset{*}{\Rightarrow} \alpha n$.如果在每一步的直接推导中，都对最左边的非终结符应用相应的产生式的右部来代替，则称这种推导为最左推导。类似地，如果在每一步的直接推导中，都对最右边的非终结符应用相应的产生式的右部来代替，则称这种推导为最右推导。

在文法G[S]中，如果存在$S \overset{*}{\Rightarrow} \alpha$，则称α是文法G的一个句型，仅含终结符号的句型是文法G的一个句子。语言L（G）是由文法G产生的所有句子组成的集合，其形式定义为：$L(G) = \{\alpha \mid S \overset{+}{\Rightarrow} \alpha$且$\alpha \in V_T^*\}$。我们称文法G1和文法G2是等价的，如果有L（G1）=L（G2）。即有可能不同的文法产生相同的语言。

对于文法G,如果有$S \overset{*}{\Rightarrow} \alpha A \delta A \overset{+}{\Rightarrow}$且β，则称β是一个关于非终结符号A的、句型αβδ的短语。如果$A \Rightarrow \beta$，则称为β是直接短语。一个句型的最左直接短语称为该句型的句柄。

要检查由符号串x是否是文法G的一个句型或者句子，就要检查是否存在一个由S到α的x的推导。推导树的每一个结点和终结符或者非终结符相关联。和终结符关联的结点是叶结点，而与非终结符相关联的结点可以是叶结点，也可以是非叶结点，树的根结点为文法的开始符号S.已知符号串x在文法G中的一个推导，就可以构造相应的推导树。将x中的每一步产生式的应用表达从所替代的非终结符号生长出新的树杈，且子结点自左向右逐个和产生式的右部符号相关联。因此，每棵推导树的终端结点自左至右所构成的字符串应该是文法G的一个句型，如果所有的终端结点都是与终结符关联的，则该字符串是文法G的一个句子，此时该推导树是完全推导树。考查文法G=（{a, b}, {S, A}, S, P），其中：

S→aAS|a

A→SbA|SS|ba

句型aabAa相对应的推导树构造的全过程如图2-3所示。

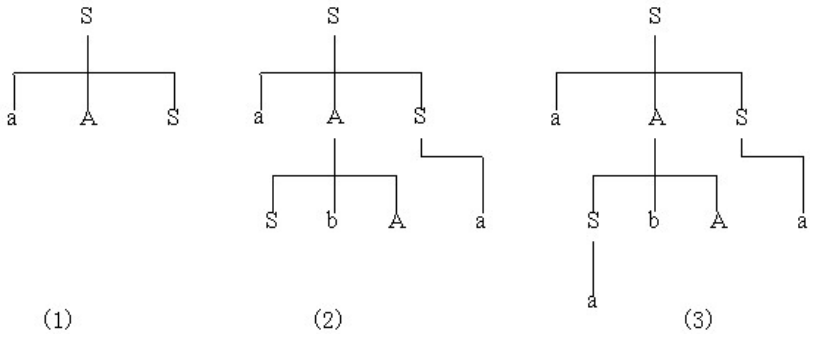


图2-3 推导树

这里我们再引入子树的概念。分析树的子树是树中一个特有的结点连同它的全部后裔和连接这些后裔的边，因此子树的根结点可能不是开始符号。子树和短语的关系十分密切。一棵树的所有叶子自左至右所构成的字符串就是相对于子树根的短语，一个句型的分析树中最左那棵只有父子两代的子树的所有叶子由左至右所构成的字符串就是该句型的句柄。

如果一文法的句子存在两棵不同的分析树，我们称该句子是二义性的，如果一文法包含二义性的句子，则称该文法为二义性的，否则该文法是无二义性的。需要注意的是，文法的二义性和语言的二义性是不同的。可能出现的情况是有两个文法G和G',且G有二义性而G'无二义性，但L（G）=L（G'），即文法G与文法G'产生相同的语言。因此，有时我们可以在不改变一个二义性文法的句子集合的情况下改变该文法，得到一个无二义性的文法。但是，也有一些语言，它们不存在无二义性的文法，这样的语言我们称为先天二义性的语言。

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年03月13日

词法分析

2.2.3 词法分析

词法分析是整个分析过程的一个子任务，它把构成源程序的字符串转换成语义上关联的单词符号（包括关键字、标识符、常数、运算符和分界符等）的序列。词法分析可以借助于有限自动机的理论与方法进行有效的处理。

1.有限自动机

有限状态自动机是具有离散输入和输出的系统的一种数学模型。系统可以处于内部状态的任何一个之中，系统当前状态概括了有关过去输入的信息，这些信息对在后来的输入上确定系统的行为是必需的。有限状态自动机与词法分析程序的设计有着密切的关系。下面是确定的有限状态自动机的形式定义：

一个确定的有限状态自动机M（记做DFA M）是一个五元组：

$$M = (\Sigma, Q, q_0, F, \delta)$$

其中：

Q是一个有限状态集合；

Σ是一个字母表，其中的每个元素称为一个输入符号；

$q_0 \in Q$，称为初始状态；

$F \subseteq Q$,称为终结状态集合；

δ是一个从Q×Σ（Q与Σ的笛卡儿乘积）到Q的单值映射：

δ（q, a）= q'　（q, q'∈Q,　a∈Σ）

表示当前状态为q,输入符号为a时，自动机将转换到下一个状态q',q'称为q的一个后继。

若 $Q=\{q_1,q_2,\cdots,q_n\}$，$\Sigma=\{a_1,a_2,\cdots,a_m\}$,则（δ（qi,aj））n×m是一个n行m列矩阵，称为DFA M的状态转换矩阵，或称转换表。

有限状态自动机可以形象地用状态转换图表示，设有限状态自动机：

DFA M =（{S, A, B, C, f}, {1, 0}, S, {f}, δ），

其中：

δ（S, 0）= B, δ（S, 1）= A, δ（A, 0）= f, δ（A, 1）= C, δ（B, 0）= C, δ（B, 1）= f,

δ（C, 0）= f, δ（C, 1）= f

其对应的状态转换图如图2-4所示。

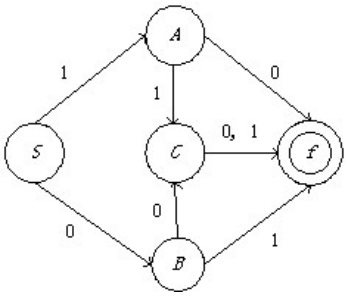


图2-4　状态转换图

图2-4中的圈表示状态结点，其中双圈表示终结状态结点。而边表示状态的转换，代表映射。边上的符号表示此转换需要输入的符号，代表映射的输入。

对于Σ上的任何字符串 $w \in \Sigma^*$，若存在一条从初态结点到终态结点的路径，在这条路径上的所有边的符号连接成的符号串恰好是w,则w被DFA M所识别（或接受、读出）。DFA M所能识别的符号串的全体记为L（M），称为DFA M所识别的语言。如果对所有 $w \in \Sigma^*$，以下述的递归方式扩张δ的定义：

δ（q,ε）=q

δ（q,wa）=δ（δ（q,w），a），对任何a∈Σ，q∈Q

我们则可以把DFA M所识别的语言形式定义为：

$$L(M)=\{w \mid w \in \Sigma^*,\ 若存在 q \in F,\ 使\ \delta(q_0,w)=q\}$$

前面介绍的是确定的有限自动机，即一个状态对于特定的输入字符有一个确定的后继状态。而当一个状态对于特定的输入字符有一个以上的后继状态时，我们称该有限自动机为非确定有限自动机（记做NFA M），其形式定义如下。

一个非确定的有限自动机M是一个五元组：

$$M=(\Sigma,Q,q_0,F,\delta)$$

其中Σ，Q,q0,F的意义和DFA的定义一样，而δ是一个从 $Q\times\Sigma$ 到Q的子集的映射，即δ：$Q\times\Sigma \to 2^Q$,其中 2^Q 是Q的幂集，即Q的所有子集组成的集合。

与确定的有限自动机一样，非确定有限自动机同样可以用状态转换图表示，所不同的是，在图中一个状态结点可能有一条以上的边到达其他状态结点。同样，对于任何字符串，若存在一条从初态结点到终态结点的路径，在这条路径上的所有边的符号连接成的符号串恰好是w,则称w为NFA M

所识别（或接受或读出）。若q0∈F,这时q0既是初始状态，也是终结状态，因而有一条从初态结点到终态结点的ε–路径，此时空符号串可以被NFA M接受。NFA M所能识别的符号串的全体记为L（M），称为NFA M所识别的语言。

对任何一个NFA M,都存在一个DFA M'使L（M'）=L（M），这时我们称M'与M等价。构造与M等价的M'的基本方法是让M'的状态对应于M的状态集合。即如果有δ $(q,a)=\{q_1,q_2,\cdots,q_n\}$，则把 $\{q_1,q_2,\cdots,q_n\}$ 看做M'的一个状态，即M'中的状态集合Q'的一个元素。

对于一个非确定有限自动机，如果我们把δ扩展为从 $Q\times\Sigma\cup\{\varepsilon\}$ {ε}到 2^Q 的映射，则我们称该自动机为带ε–转移的非确定有限自动机。同样，对于带ε–转移的非确定有限自动机，我们也可以构造与之等价的不带ε–转移的非确定有限自动机。

2.正规表达式

正规表达式是一个十分有用的概念，它紧凑地表达有限自动机所接受的语言。对正规表达式的递归定义为：一个正规表达式是按照一组定义规则由一些较简单的正规表达式所组成的。在字母表Σ上的正规表达式可以使用以下规则定义。

ε和Φ是Σ上的正规表达式，它们所表示的语言分别为{ε}和Φ。

如果a是Σ内的一个符号，则a是一个正规表达式，所表示的语言为{a},即包含符号串a的集合。

如果r和s分别是表示语言L（r）和L（s）的正规表达式，那么：

（r）|（s）是一个表示 $L(r)\cup L(s)$ 的正规表达式；

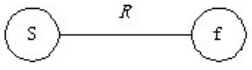
（r）（s）是一个表示 $L(r)L(s)$ 的正规表达式；

$(r)^*$ 是一个表示 $(L(r))^*$ 的正规表达式；

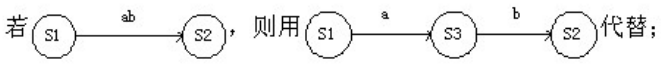
（r）是一个表示L（r）的正规表达式。

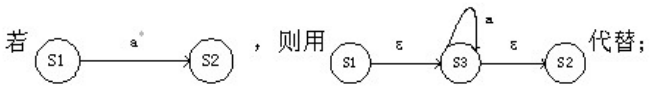
通常在正规表达式中，一元运算符"*"具有最高的优先级，连接运算具有次优先级，运算符"|"具有最低优先级，这三个运算都是左结合的。每一个正规表达式R都对应一个有限自动机M,使M所接受的语言就是正规表达式的值。经过以下步骤可以从一个正规表达式R构造出相应的有限自动机M.

首先定义初始状态S和终止状态f,并且组成有向图：



然后反复应用以下规则：





直到所有的边都以Σ中的字母或ε标记为止。由此产生了一个带ε–转移的非确定有限自动机，然后可以通过上面介绍的方法，把该自动机转换成确定有限状态自动机。

下面举一个例子说明自动机理论在词法分析程序中的应用。C语言中对标识符的规定为由"_"或以字母开头的由"_"、字母和数字组成的字符串，该标识符的定义可以表示为下面的正则表达式：

$$(\_\mid a)(\_\mid a\mid d)^*$$

式中的α代表字母字符{A,…，Z,a,…，z},d代表数字字符{0,1,…，9}.利用前面的方法构造出如图2-5所示的有限自动机。

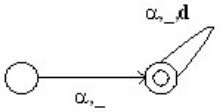


图2-5 有限自动机图例

该自动机所接受的语言就是C语言中的标识符。

在有限自动机的状态转换过程中，需要执行相关的语义动作。例如当识别到一个标识符时，需要在符号表中添加该标识符，并且向语法分析程序输送表示该标识符的单词。

第 2 章：程序语言基础知识 作者：希赛教育软考学院 来源：希赛网 2014年03月13日

语法分析

2.2.4 语法分析

1.下推自动机

为了帮助理解语法分析程序，在这里先介绍下推自动机的概念。下推自动机（PDA）是自动机理论中定义的一种抽象的计算模型。下推自动机比有限状态自动机复杂，除了有限状态组成部分外，还包括一个长度不受限制的栈；下推自动机的状态迁移不但要参考有限状态部分，也要参照栈当前的状态；状态迁移不但包括有限状态的变迁，还包括一个栈的出栈或入栈过程。下推自动机可以形象地理解为，把有限状态自动机扩展使之可以存取一个栈。

下推自动机存在确定与非确定两种形式，两者并不等价。

每一个下推自动机都接受一种形式语言，确定下推自动机接受的语言是上下文无关语言。如果我们把下推自动机扩展，允许一个有限状态自动机存取两个栈，我们得到一个能力更强的自动机，这个自动机与图灵机等价。下面是下推自动机的形式定义：

下推自动机M是如下的一个七元组（Q, Σ，Γ，δ，q_0, Z_0, F）

其中：

Q是一个有限状态集合；

Σ是一个字母表，称为输入字母表；

Γ是一个字母表，称为栈字母表；

q_0属于Q,是初始状态；

Z_0属于Γ，是一个特殊的栈符号，称为栈起始符号；

F包含于Q,是终结状态集合；

δ：Q×（Σ∪{ε}）×Γ →Q×Γ^* 是M的动作函数。

我们可以把下推自动机想像为由一个状态寄存器、一个下推栈及一个读入头组成的自动机模型，每次从读入头读入下一个字符，并根据状态寄存器的状态及下推栈的栈顶符号进行动作，改变状态寄存器的状态值，弹出栈顶符号并下推入相应的符号。当字符串读入完毕，并且栈顶符号为Z0时表示接受该符号串。

2.自顶向下语法分析

该分析方法是以文法开始符号为根，试图寻找以输入符号串为端点的推导树。每次都以最左边的那个非终结符为根，选择适当的产生式，往下扩展子树。

考虑上下文无关文法G=（{a, b, c}, {S, B}, S, P），其中：

S→aBS

S→b

B→cBS

B→d

设输入字符串为w=acdbb,如果w是G的一个句子，则可以从S出发构造一棵以w为端点的推导树。我们首先构造分析该文法的下推自动机，对于输入串acbb,则下推栈与输入的过程如表2-2所示。

表2-2 下推自动机生成表

下 推 栈	输 入	文法产生式
Z₀S	acdbb$	S→aBS
Z₀aBS	cdbb$	B→cBS
Z₀aSBS	dbb$	B→d
Z₀aSS	bb$	S→b
Z₀aS	b$	S→b
Z₀	------	------

这里的关键是，每个产生式都有一个由相应终结符组成的选择集合。只要产生式左部相同的选择集合两两不相交，则下推自动机就能确定性地工作。为此引入两个集合FIRST（α）、FOLLOW（A）来求选择集合。

（1）$\mathrm{FIRST}(\alpha)=\{a \mid \alpha \overset{*}{\Rightarrow} a\cdots, a\in V_T, \alpha\in V^*\}$

特别是，若$\alpha \overset{*}{\Rightarrow} \varepsilon$，则规定ε∈FIRST（α）。

求法：

①对每一文法符号X∈V计算FIRST（X）。

（a）若X∈VT,则X加入FIRST（X）。

（b）若X∈VN,考查它的产生式：

若X→a…，a∈VT,则a加入FIRST（X）；

若X→ε，则ε加入FIRST（X）；

若X→R1R2,…，Rk,且R1∈VN,则先将FIRST（R1）/ε（所有非ε元素）加入FIRST（X），然后在R1R2…Ri–1 $\overset{*}{\Rightarrow}$ε（意味着：对每个j:1≤j≤i–1,ε∈FIRST（Rj））时，将FIRST（Ri）/ε加入FIRST（X）。特别是，若ε∈FIRST（Rj），j=1,2,…，k,将ε加入FIRST（X）。

（c）反复执行（b）步，直到每个符号的FIRST集合不再增大为止。

②对α∈V*,计算FIRST（α），设α=X1 X2,…，Xn,Xi∈V,1≤i≤n.

（a）FIRST（α）=FIRST（X1）/ε。

（b）若ε∈FIRST（Xj），j=1,2,…，i–1,2≤i≤n,则将FIRST（Xi）/ε加入到FIRST（α）中；特别是，若ε∈FIRST（Xj），j=1,2,…，n,则ε加入到FIRST（α）中。

（2）$\mathrm{FOLLOW}(A)=\{a \mid S \overset{*}{\Rightarrow} \cdots Aa\cdots, a\in V_T, \alpha\in V_N\}$

其中，S是文法开始符。特别是S $\overset{*}{\Rightarrow}$ …A,则规定#∈FOLLOW（A）。

求法：对每一个A∈VN,逐个考查产生式：

①将#加入FOLLOW（S），S为文法开始符。

②若A→αBβ是一条产生式，B∈VN,则把FIRST（β）/ε加入FOLLOW（B）。

③若A→αB是一条产生式或A→αBβ，且β $\overset{*}{\Rightarrow}$ε（即（ε∈FIRST（β））），则将FOLLOW（A）加入FOLLOW（B）。

④反复使用2）、3）直到每个非终结符的FOLLOW集不再增大为止。

现在我们根据FIRST集合和Follow集合求SELECT集合，公式如下：

SELECT（A→α）=FIRST（α）∪FOLLOW（A）

即当α不能推导出ε时，该产生式的选择集合就是α的头符号集合，否则，该产生式的选择集合是α的头符号集合和A的后继集合之和。对于上面的文法G[S],其各个产生式的选择集合为：

SELECT（S→aBS）= {a}

SELECT（S→b）= {b}

SELECT（B→cBS）= {c}

SELECT（B→d）= {d}

设G是一个上下文无关文法，如果该文法中左部相同的产生式的选择集合两两不相交，则称G为LL（1）文法。

除使用下推自动机外，另一个自顶向下的语法分析方法是采用递归子程序技术。其基本思想是对每一个非终结符构造一个过程，由这个过程识别相应的非终结符所能生成的终结符串。过程执行时假定输入符号总是语法结构A所能产生的终结符号串的头符号。过程结束时，当前输入符号是非终结符A的后继符号。由于一个过程在识别时会调用其他分析过程（包括它自己）来识别其各自所能识别的符号串，实际上，递归程序法是使用允许递归过程调用的语言在运行时刻的调用栈来模拟下推自动机的下推栈。

3.自底向上语法分析

自底向上分析技术是从输入符号串出发，试图把它归约为识别符号。从语法树的角度看，这个技术首先以输入符号作为语法树的末端结点，然后向根结点方向构造语法树。

这里介绍的分析方法称为"移进–规约"方法。

首先介绍规范归约的定义。

假定α是文法G的一个句子。我们称右句型序列为 $a_n, a_{n-1}, \cdots, a_1, a_0$ 是α的一个规范归约，如果序列满足：

$$a_n = a, a_0 = S$$

对于任何i（0<i≤n），a_{i-1}是从 a_i 经过把句柄替换为相应产生式的左部符号而得到的。

可见，规范归约是关于α的一个最右推导的逆过程。因为规范归约也称为最左归约。在形式语言中，最右推导常被称为规范推导，由规范推导得到的句型成为规范句型。如果文法G是没有二义性的，则其规范推导的逆过程必定是规范归约。规范归约的中心问题是如何寻找或确定一个句型的句柄。

要进行移进–归约分析还是使用下推自动机，方法是尽可能从已输入的符号串或其已规约后的形式中寻找可以进行规约的句柄，只有当再找不到句柄时才读入下一个输入字符。据此，下推自动机的动作可以分为4类。

把当前读入的符号移入到下推栈顶部，这一动作成为移进动作。

弹出一个栈符号串（句柄），并把另一个栈符号串（通常是句柄产生式左部的非终结符）下推

入栈，这一动作称为归约动作。

当下推栈中只有最终归约符号S（在规范归约的情况下是文法G的开始符号），并且符号串被全部吸收，则接受符号串。

当发现语法错误时，调用出错处理程序进行校正。

移进–归约的分析过程就是在从左至右将输入符号移进栈内的过程中，一旦发现栈顶出现可归约串就立即进行归约，直至接受符号串或发生语法错误。由于规范归约的最左性，因此，在分析过程中，任何可归约串的出现都必定在栈顶。

正如规范归约的中心问题是如何确定一个句型的句柄一样，移进–归约分析的中心问题是如何精确定义可归约串。对可归约串的不同定义形成了不同的分析方法。

我们首先关注文法的符号之间的约束关系。设R和S是文法G的两个符号，且R、S∈VT∪VN,U、V和W是文法G的两个符号，且U、V、W∈VN,我们可以区分下面的4种情况。

（1）U→…RS…

（2）U→…RV…，且$V \Rightarrow S\cdots$

（3）U→…VS…，且$V \Rightarrow \cdots R$

（4）U→…VW…，且$V \Rightarrow \cdots R, W \Rightarrow S\cdots$

要区分上面的情况，下推自动机需要每次比较栈顶符号R和当前输入符号S.在（1）中，R和S都在句柄中，此时相对于归约动作，R和S具有相同的优先级，必须同时被归约。在（2）中，S是当前句柄中的一个符号，而R不在该句柄中，这时S必须先参加归约，只有当S归约成V时，R才有可能参加归约，我们说R的优先级低于S的优先级。在（3）和（4）中，R属于某一句柄，而S不在该句柄中，我们称R的优先级高于S.只有当R是某产生式右部的尾符号时，才可能出现这种情况。在（3）和（4）中，S一定是终结符。对于上面的各种情况，我们为文法引入弱优先关系。对符合（1）和（2）的文法符号R和S,规定R<·S,对符合（3）和（4）的文法符号R和S,规定R·>S.根据前面的分析，当R·>S时，R一定是句柄的尾符号，否则，R一定不是句柄的尾符号。

引入弱优先关系后，我们就可以定义弱优先文法。

设G是一个文法，如果G满足：

文法G的任意两个符号之间至多只有一个弱优先关系成立；

G的任何两个产生式右部各不相同；

若U1→xSy,U2→y是G的两个产生式，则S和U2之间不存在任何弱优先关系，

则称G为弱优先文法。

为了使语法分析有开始和终结，必须对下推栈的栈底符号Z0和输入字符串的终止符号"$"做特殊处理：

$$Z_0 <\cdot R,\ R \in \{S_i \mid S_i \in V_T \cup V_N,\ 且 S \overset{*}{\Rightarrow} S_i \ldots\}$$

$$R \cdot > \$,\ R \in \{S_i \mid S_i \in V_T \cup V_N,\ 且 S \overset{*}{\Rightarrow} \ldots S_i\}$$

这样，就可以根据文法符号及栈底符号、终止符号之间的弱优先关系构造弱优先矩阵。无论哪种语法分析方法，其工作实质均相同，就是在弱优先矩阵中标记"<·"的元素对应的动作是移进动作；而标记"·>"的元素所对应的动作是归约，即检查相关产生式的右部是否和栈顶符号串匹配，如匹配则用产生式的左部非终结符置换栈顶的符号串。当涉及的产生式超过一条时，应遵循"最长产生式优先"的原则。对于在矩阵中开始符号S行和符号串终止符号$列中标记"·>"的元素来说，对应的动作还需要检查栈中是否是Z0S,即检查是否应接受符号串。

这里再介绍算符优先分析法。这是一种特别有利于分析表达式的方法，适用于下面形式定义的

称为算符文法的文法类。

设G是一个文法，如果G中不存在形如A→ε及A→αBCβ的产生式（其中A,B,C为文法非终结符，α，β为文法符号串），即G中没有右部为ε或右部具有相邻非终结符的产生式，则称G为算符文法。

例如表达式文法：

E→E+E | E–E | E*E | E/E | E^E | （ E ） | –E | id

就是算符文法。

仿照弱优先关系的定义，我们为算符文法引入算符优先关系"="、"·>"和"<·":如果存在产生式U→...RS...或者U→...RVS...，则称R=S;如果存在产生式U→...RW...，且 $W \overset{+}{\Rightarrow} S\cdots$ 或者 $W \overset{+}{\Rightarrow} VS\cdots$,则R<·S;如果存在产生式U→...WS...，且 $W \overset{+}{\Rightarrow} \cdots R$ 或者 $W \overset{+}{\Rightarrow} \cdots R$ ，则R·>S.对于一个算符文法，如果任何两个终结符之间至多只存在一个优先关系，则称该文法为算法优先文法。

终结符之间的优先关系"="可以直接从文法的产生式得出。优先关系"<·"可以通过下面的方法求出。

检查文法的各产生式，若U→S...或U→VS...，其中S是终结符，则S是U的最左终结符。

若U→V...，则V的最左终结符是U的最左终结符。

反复执行（2），直至得到文法的每个非终结符的最左终结符集合。

检查文法中的所有形如U→...RV...的产生式，对于V的最左终结符集合中的每个元素，可得出R<·S.

使用类似的方法，我们可以求出关系"·>".

运算符优先文法的语法分析可以表达成下面的算法形式：

```
k := 1; S[k] := 'Z0';
repeat
next_token;
if S[k] ∈VT then j := k else j := k – 1;
while S[j] ·> token do
begin
repeat
q := S[j];
if S[j – 1] ∈VT then j := j – 1 else j := j – 2;
until S[j] <·q
if N→S[j + 1]...S[k] then
begin
k := j + 1;
S[k] := N;
end;
end;
if S[j] <· token or S[j] = token then
begin
k := k + 1;
S[k] := token;
```

end

else error

until token='$'

语法翻译

2.2.5 语法翻译

1.中间代码

虽然编译程序可以直接把一个源程序翻译成目标程序，但是在许多编译系统的设计中，仍采用独立于机器的中间代码作为过渡。其优点是便于编译系统的建立和移植，并且便于进行独立于机器的代码优化。常见的中间代码表示有语法树、后缀式和三地址代码。这里介绍后缀式和三地址代码表示。

后缀表示又称为逆波兰表示，最初是用于表示表达式的计算次序。在后缀表示中，运算符紧跟在相应的运算对象后。例如，表达式（A+B）*C,使用后缀表示为AB+C*.在后缀表示中，运算符既表示了对运算对象所执行的运算，也表示了这一运算之前的中间结果。因此，在后缀表示中不需要使用括号。

通过使用栈，计算后缀表达式的算法如下。

如果P的下一项是运算对象，则将它压入栈。

如果P的下一项是二元操作符（即需要参数个数为2的运算符），则对栈顶两个对象实施运算，并且将运算的结果代替这两个运算对象而进栈。

如果P的下一项是一元操作符（即需要参数个数为2的运算符），则对栈顶对象实施运算，并且将运算的结果代替这个对象而进栈。

如果P的下一项是n元操作符（即需要参数个数为n的运算符），那么它的参数就是栈顶的n项，把该运算符作用于这n项，得到的结果作为操作数替代栈顶的n项。

最后的结果留在栈顶。

例如，AB+C*的计算过程为：

①A进入堆栈；

②B进入堆栈；

③遇到二元运算符"+",则AB出堆栈，并将A+B的结果X送入堆栈；

④C进入堆栈；

⑤遇到二元运算符"*",则CX出堆栈，并将C*X的结果送入堆栈；

⑥现在堆栈顶部存放的是整个表达式的值。

把运算符扩展到n元后，后缀表达式可以表示为：$O_1O_2...O_n\theta$，其中θ是n运算符，运算对象的个数由运算符决定。

程序语言中的赋值语句通常形式为<V> := <E>,其中<V>是变量，而<E>是表达式。使用后缀形式表达时，赋值语句可以表示为<V><E>:=.因此，表达式n:=（A+B）*C的后缀形式为nAB+C*:=.在扫描到赋值符号":="时，我们需要把栈顶的值送到栈中第二个元素所指向的存储单元，然后把这两个元素从栈中弹出。

对于无条件跳转语句GOTO L的后缀表示是L BR.其中，BR是一元运算符，表示跳转到L指出的位置上继续执行。在这个基础上，我们可以设立条件跳转运算，其后缀形式为：<E> L BZ,其中BZ是一个二元运算符，表示当表达式<E>的值为真时跳到L指出的位置继续执行，否则向下顺序执行。有了这两个运算符，我们可以把if b1<b2 then a:=b+c else a:=c,使用后缀表示为b1 b2 < L1 BZ a c := L2 BR （L1） a b c + := （L2）。其中带括号的L1和L2表示L1和L2标号出现的位置。在实际应用中，通常L会以标明位置的偏移表示。

三地址代码，也称为带有结果的四元组表示，是另外一种较常见的中间代码。三地址代码与汇编语言代码是类似的。语句可以带有用符号命名的标号，而且存在各种控制流的语句。下面列出了三地址代码的表示方法。

赋值语句：对于二元算术运算符或逻辑运算符op,其形式为x := y op z,或（op, y, z, x），表示将op作用在y与z上的结果赋予x,对于一元算术运算符，其形式为x := op y或（op, y, , x），表示op作用于y上的结果赋予x,对于简单的复制语句，其形式为x := y或（ : =, y, , x），表示y的值赋予x.

无条件转移语句：其形式为GOTO L,或（BR, , ,L），表示无条件跳转到L指出的语句继续执行。

条件转移语句：其形式为if x GOTO L或（BZ, x, , L），表示当x为逻辑真时跳转到L指出的语句继续执行。

前面使用后缀表达式表示的条件赋值语句，可以用三地址代码表示，如表2-3所示。

表2-3 三地址代码表

op	arg1	arg2	result
<	b1	b2	t1
BZ	t1		L1
:=	c		a
BR			L2
(L1) +	b	c	a
(L2)			

oparg1arg2result

<b1b2t1

BZt1L1

:=ca

BRL2

（L1） +bca

（L2）

其中t1是编译程序生成的用于存放结果的临时变量。

2.表达式的翻译

在进行适当的改造后，下推自动机就可以进行语义分析工作。下面以自底向上进行语法分析的下推自动机为例进行说明。

考查文法G（E）：

E→E+T

E→T

T→T*P

T→P

P→（E）

P→i

设输入的表达式为i+i,为方便说明起见，我们把两个i分别称为i1和i2.自底向上语法分析是规范推导的逆过程，而该表达式句子的规范推导为：

$$E \Rightarrow E+T \Rightarrow E+P \Rightarrow E+i2 \Rightarrow T+i2 \Rightarrow P+i2 \Rightarrow i1+i2$$

在进行分析时，当把i2规约成P,我们可以输出i2,当把i1规约成P,我们可以输出i1,当把E+T规约为E,我们最后则输出+,完成分析。这时，就需要对文法进行改动，使其能表达使用产生式时的语义动作。这些语义动作通常附在相关的产生式下面。改动后的文法如下。

G（E）：

E→E+T

Generate（+）

E→T

T→T*P

Generate（*）

T→P

P→（E）

P→i

Generate（i.Value）

其中，Generate是输出子程序，其作用是把参数放到中间代码中，i.Value是变量的值属性。在分析中，如果一个非终结符出现在句柄产生式中，那么与该非终结符有关的语义工作已经完成。

下面把翻译为三地址代码的讨论扩展到中缀表达式。为了能存放中间结果，我们定义能生成临时变量的过程newtemp.其作用为申请合适大小的存储区，将临时变量标记T和序号填入相应的单元，然后返回该存储区的首地址。此外，表达式中各参数的类型可能不同，编译程序需要根据程序语言的类型系统做出相应的处理。为此，我们还需要引入量的类型属性，通过E.Type进行标记。为方便起见，我们现在只讨论两种类型的情况，一种是整型，一种是实型。考虑两个变量相加，只有当两个变量类型都为整型时，得到的结果的类型才为整型，当两个变量类型都为实型时，结果类型也为实型，当一个变量类型为整型，另一个为实型时，编译程序首先需要把整型变量转换为实型，以避免数据丢失，然后进行加法运算，得到的结果的类型为实型。为此，我们需要引入数据类型转换的一元运算：

（FLOAT I, R）

它将整型量I转换成实型量，并存放在R中。另外，原来的加法运算也需要分为实型加法和整型加法，分别记为REAL+和INTEGER+.对于上面文法G（E）的产生式（1）E→E'+T的语义动作可以表示成：

E.Value := newtemp;

if E'.Type = INTEGER and T.Type=INTEGER then

begin

E.Type := INTEGER

Generate（INTEGER+, E'.Value, T.Value, E.Value）；

end;

else if E'.Type=INTEGER and T.Type=REAL then

begin

E.Type := REAL;

T1 := newtemp;

Generate（FLOAT, E'.Value, ,T1）；

Generate（REAL+, T1, T.Value, E.Value）；

end;

else if E'.Type=REAL and T.Type=INTEGER then

begin

E.Type := REAL;

T2 := newtemp;

Generate（FLOAT, T.Value, ,T2）；

Generate（REAL+, E'.Value, T2, E.Value）；

end;

else

begin

E.Type := REAL;

Generate（REAL+, E'.Value, T.Value）；

end;

当产生式（1）被用于规约时，应将E连同它的属性E.Value和E.Type一起下推入栈。产生式（3）的语义动作可以类似写出。产生式（2）、（4）、（5）和（6）的动作比较简单，以产生式（4）为例，其语义动作为：

T.Type := P.Type;

T.Value := P.Value;

附值语句S→V:=E也可以类似推得，需要注意的是赋值符号左边的变量的类型是主类型，如果右边的变量类型不符，则必须根据程序语言的类型系统进行类型转换，我们这里假设类型系统只允许整型到实型的转换而不允许实型到整型的转换。上面的产生式的语义动作如下：

if V.Type=E.Type then

begin

V.Type:=E.Type

V.Value:=E.Value

end;

else if V.Type=REAL and E.Type=INTEGER then

begin

T1 := newtemp;

Generate（FLOAT, E, , T1）；

V.Type := T1.Type;

V.Value := T1.Value;

else

begin

error;

end;

代码生成

2.2.6 代码生成

代码生成时编译程序的最后一个阶段，它将程序的中间代码表示作为输入，并产生等价的目标代码作为输出。

类似于中间代码，目标代码也有若干种形式：绝对机器代码、可再定位机器语言、汇编语言等。为了讨论方便，我们假定代码生成程序产生用汇编语言书写的目标程序。

为此，我们必须对目标机器及其指令系统做出定义。这里，我们将采用具有多个通用寄存器的机器作为目标机器。我们的目标机器是一个按字节编址的机器，以4个字节为一个字，有n个通用寄存器R0,R1,Rn–1,并有形如op src, dest的两地址指令。其中op为操作码，src和dest称为源和目标，是数据域。此目标机器有如下的基本指令：

MOV（将源移到目标中）

ADD（将源加到目标中）

SUB（在目标中减去源）

源和目标域并不足以用来存放存储地址，因此一条指令的源和目标是通过将寄存器与带有地址方式的存储单元结合起来确定的。下面的contents（a），表示由a所代表的存储单元或寄存器的内容。如表2-4所示是地址方式及它们的汇编语言形式。

表2-4 各地址方式及在汇编语言中的形式

地 址 方 式	汇 编 形 式	地　址
直接地址方式	M	M
寄存器方式	R	R
间接寄存器方式	*R	contents(R)
索引方式	c(R)	c+contents(R)
间接索引方式	*c(R)	contents(c+contents(R))
字面常数	#C	C

寄存器分配是代码生成中的一个重要问题。寄存器是有限的资源，并且用途广泛。这些寄存器是否有效，直接涉及目标代码质量的好坏。

首先介绍基本块的概念。一个基本块是这样一个连续语句序列：其中控制流从第一条语句（称

为入口语句）进入，从最后一条语句离开，没有中途停止或分支。下面是划分三地址程序基本块的方法。

①首先确定基本块的入口。

代码序列的第一条语句是一条入口语句。

任何一个条件或无条件转移语句转移到的那条语句是一条入口语句。

任何紧接在一条条件或无条件转移语句后面的语句是一条入口语句。

②对于上述求出的每一个入口语句，其基本块由该语句到下一条入口语句（不包括这一条入口语句），或到一条转移语句（包括转移语句），或到一个停止语句（包括停止语句）之间的语句序列组成。

为了有效利用寄存器，我们首先设立一个寄存器工作表RVALUE,表中的每个单元对应一个寄存器，用以记录运行时刻寄存器中值的情况。开始时，各寄存器的值为空。当我们向一个寄存器中存放变量时，我们只要将变量挂在对应的RVALUE单元上。这样，RVALUE中的单元值就反映了运行时寄存器的值。每当需要将某变量的值取至寄存器时，代码生成程序首先检查寄存器中是否有该值，如果有，则不必生成相应的取数指令。

除了关心寄存器的值，我们还要关心寄存器的状态，一般状态会分为以下几种情况。

寄存器不含有任何值，即该寄存器处于空闲状态。

寄存器中的值是程序中某变量的值，但与正在处理的中间代码基本块中的后续语句无关。

寄存器中的值是与正在处理的中间代码无关的中间变量的值。

寄存器中的值是正在处理的中间代码基本块中后续语句需要引用的。

寄存器中的值是当前要处理的中间代码的某操作数的值。

上面的状态对寄存器的分配有至关重要的意义，为此，我们设立一张寄存器状态表RSTATE,表中每个单元对应一个寄存器，它的值反映寄存器的状态。代码生成程序应该先选择状态为1的寄存器进行分配，其次是状态为2的寄存器，再次是状态为3的寄存器，依次类推。这样可以避免生成不必要的存取指令。

至此，所谓的分配寄存器，就是往某一约定工作单元j中送一个该寄存器的号码，并将RVALUE中对应的单元内容送到工作单元jVALUE.代码生成程序在处理每一条三地址代码前，必须根据代码各分量及寄存器当时的值之间的关系修改RSTATE.在处理完后，必须及时修正RSTATE和RVALUE的值，以正确反映寄存器的值和状态。

由于机器指令的位移区的位数是有限的，因此与内存有关的指令只能访问有限范围内的存储单元。要存取超过此范围的存储单元，我们需要使用前面介绍的地址方式，把特定的寄存器定义为变址寄存器。通过设定变址寄存器的值，并运用各种地址方式，则可以灵活地访问各存储单元。

访问变量的过程为，（1）按照程序语言的作用域规则，沿活动记录的访问链定位该变量所在的活动记录；（2）把变址寄存器定位为该活动记录的相应存储工作区的首地址；（3）使用该变量的偏移和变址寄存器的值访问该存储单元。这里的关键是要准确定位变量所在的块及其首地址。

最后，介绍一下简单的代码生成的算法。代码生成程序的工作过程为逐个处理三地址代码，在每次处理一条三地址代码前，将该三地址代码放在固定的地方，然后根据三地址代码的操作码转入相应的处理程序，直至所有的三地址代码全部处理完毕。这样，代码生成程序可以简单表达为输入三地址代码和处理三地址代码。

根据上面的预备，我们可以将变量赋值a:=b的代码生成算法写出如下：

检查RVALUE表，b的值是否在寄存器中；

如果b的值不在寄存器中，则：

即为其分配一个寄存器R1;

定位变量b的地址，设置变址寄存器Rd,使其指向b所在块的首地址A,输出语句MOV Rd, A;

根据b的偏移D,把b挂在R1对应的RVALUE单元中，输出MOV R1, *D（Rd）；

如果b的值在寄存器中，我们为描述方便也假定其为R1;

定位变量a的地址，设置变址寄存器Rd,使其指向a所在块的首地址A,输出语句MOV Rd, A;

根据a的偏移D,把b挂在R1对应的RVALUE单元中，输出MOV *D（Rd）， R1.

其他的语句也可以使用类似的过程生成目标代码。

解释系统基本原理

2.3 解释系统基本原理

解释程序是一种语言处理程序，它实际是一台虚拟的机器，直接理解执行源程序或源程序的内部形式（中间代码）。因此，解释程序并不产生目标程序，这是它和编译程序的主要区别。图2-6显示了解释程序实现的3种可能情况。

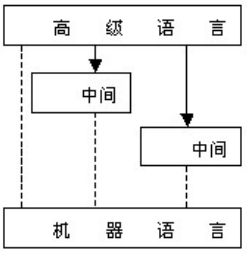


图 2-6　解释程序实现的三种可能情况

在类型A的实现方案中，解释程序直接对源程序进行解释执行。这种解释程序对源程序进行逐字符的检查，然后执行程序指令表示的动作。例如，当解释程序扫描到字符串序列：

GOTO Label

解释程序意识到GOTO命令代表无条件跳转至Label所标识的位置，于是就开始搜索源程序中标号Label后面紧跟冒号":"的出现位置，然后跳转至该位置继续执行。这类系统在实现时需要反复扫描源程序，因此按A方案实现的解释程序效率很低，早期的解释性BASIC语言就是采取该方案实现的。

在类型B的实现方案中，翻译程序先将源程序翻译成较贴近高级语言的高级中间代码，然后再扫描高级中间代码，对高级中间代码进行解释执行。所谓较贴近高级语言的高级中间代码，是指中间代码与高级语言的语句形式相像，两者存在着一一对应的关系。APL和SNOBOL的实现很多都采用这种方法。

类型C又是一种解释程序的实现方案。类型C的解释程序和类型B的解释程序不同点在于，类型C的解释程序首先将源程序转化成和机器代码十分接近的低级中间代码，然后再解释执行这种低级中间代码。一般说来，在这种实现方案下，高级语言的语句和低级中间代码之间存在着1–n对应关系。

例如，微软的C#语言，首先被编译成一种形式上较类似汇编语言的中间语言IL表示的代码，然后在通用语言运行时（Common Language Runtime）解释执行IL程序。这类系统具有良好的可移植性。

下面对解释系统的结构加以扼要的描述。这类系统通常可以分成两部分，第一部分包括通常的词法分析程序及句法和语义分析程序，它的作用仍是把源程序翻译成中间代码。第二部分是解释部分，用来对第一部分所产生的中间代码进行解释。由于真正的解释工作在解释部分完成，下面的介绍仅涉及第二部分。

用数组MEM来模拟计算机内存，（源程序的）中间代码程序和解释部分的各个子程序都存放在数组MEM中。全局变量PC是一个程序计数器，它记录了当前正在执行的中间代码位置。这种解释部分的常见的总体结构可以由下面两部分组成：

I1.PC=PC+1

I2.执行位于opcode_table[MEM[PC]]的子程序（解释子程序执行后返回前面I1）

用一个简单例子来说明其工作情况。设有两个实型变量A和B,A与B相加的中间代码是：

Start:Ipush

A

Ipush

B

Iaddreal

其中，中间代码Ipush,Iaddreal实际上都是opcode_table表的索引值（即位移），而该表单元中存放的值：

opcode_table[Ipush]= push

opcode_table[Iaddreal]= addreal

就是对应的解释子程序的起始地址，A和B都是MEM中的索引值，解释部分开始执行时，PC的值为Start-1.解释部分可表示如下：

interpreter_loop:

PC=PC+1;

goto opcode_table[MEM [PC]];

Push:

PC=PC+1

stackreal （ MEM [MEM [PC]] ）；

Addreal:

stackreal （ popreal （ ） + popreal （ ） ）；

goto interpreter_loop;

……

其中，stackreal表示把相应值压入下推栈，而popreal （ ）表示把下推栈的栈顶元素取出，然后弹出栈顶元素。

作者：希赛教育软考学院 来源：希赛网 2014年01月26日

程序语言的数据类型

2.4 程序语言的数据类型

数据类型是一组数据对象及创建和操纵它们的操作集合所组成的类。每种程序设计语言都有自己特定的数据类型系统，通常都会为程序员提供基本的数据类型。而现代语言大多都会提供用户自定义新的数据类型的机制，用于表示结构化或抽象的数据类型，例如C++中为用户提供了自定义结构（struct）、联合（union）和类（class）的机制。

一个数据类型中所有元素构成了这个数据类型的定义域，即该数据类型的对象的取值范围。如果一个数据类型的定义域仅由常量值组成，则该数据类型就是标量类型，例如整型、实型、枚举类型都是标量类型。结构化数据类型和抽象数据类型的定义域中的元素都有自己的域，这些域又有自己的数据类型。

上一节 本书简介 下一节

作者：希赛教育软考学院 来源：希赛网 2014年03月13日

基本数据类型

2.4.1 基本数据类型

基本数据对象只有单一的数据值，这种对象及其上的操作的集合组成的类称为基本数据类型。基本数据类型包含有内建数据类型（Built-In），枚举类型和复合数据类型。

所谓内建数据类型，是指由程序设计语言提供的、无须用户另外定义的数据类型。各程序设计语言提供的内建数据类型都会有所不同，但通常都会提供整型、实型、字符型、布尔型4种内建数据类型。下面是一些程序语言提供的内建数据类型：

Pascal:integer,real,boolean和char;

C++:short,int,long,float,double,bool和char.

1.内建数据类型

1）数字数据类型

整型和实型都是数字数据类型，它们通常都是由硬件支持的。为整数类型定义的整数值的集合是一个有界的数学中研究的无限整数集的子集。类似地，为实数类型定义的实数值的集合是一个有界的数学中研究的无限实数集的子集。整型值和实型值在不同语言和不同的计算机系统中的取值范围是不同的。

数字数据对象上的操作通常包括：

赋值操作

算术操作

二元算术操作的形式是number x number → number.二元操作符可以是加、减、乘、除和求余（取模）等类似操作。要注意的是，两个不同类型的数字数据对象进行二元算术操作时，通常会

将有效取值范围较小的数据对象转化为有效取值范围较大的数据类型，然后再进行算术操作。例如，在C++中，如果一个short类型数据对象与一个double类型数据对象进行加法运算，则会先把short类型数据对象转换为double类型，然后再进行运算，得到的结果为double类型的数据对象。

一元算术操作的形式是number→number.一元操作符通常有"−"和"+"等，还有以函数库中的子程序形式提供的其他算术操作。

（1）关系操作

关系操作的形式是number x number → boolean.关系操作符可以是大于、小于、等于、大于等于、小于等于和不等于。关系操作比较输入操作数的大小，然后返回布尔型数据对象作为结果。需要注意的是，因为近似问题，很少有两个实型数的比较会是完全相等的。在进行实型数的比较时，通常是定义一个足够小的误差范围，例如ε=0.0001,若两个实型数的差的绝对值小于ε，则认为两个实型数相等。

（2）位操作

在某些程序语言中，如C/C++,在整型数据对象上还定义了位操作，其形式为integer x integer → integer.C/C++中的位操作包括与、或和移位等。

2）布尔类型

布尔型数据对象的定义域只有两个对象，true和false,表示逻辑值的真和假。在布尔型数据对象上的基本操作有赋值、相等判定、与、或和非，它们的形式分别是boolean x boolean → boolean（其中x是相等判定操作）；boolean x boolean → boolean（其中x是与操作）；boolean x boolean → boolean（其中x是或操作）；x boolean → boolean（其中x是非操作）。

3）字符类型

字符数据类型是以单个字符作为其值的数据对象。字符类型的定义域通常是由语言定义的、和标准字符集有关的集合，该集合通常得到计算机硬件和操作系统的支持，如ASCII字符集。在字符集中的字符排序成为字符集的对照序列。对照序列为通过关系操作决定字符的顺序提供了依据。对字符类型的数据对象的操作有赋值、关系操作和检查字符是否属于某个字符集合（如可打印字符）的操作。

2.枚举类型

我们经常会让一个变量只取少量的几个值，如一周有7天，代表星期的变量就只会取7个值中的一个。虽然用整型数据对象可以表达该概念（例如用整数1~7表示星期），但是程序员没法保证不对这些变量进行无意义的操作（如Monday与Monday相乘是毫无意义的）。枚举类型使程序员可以更好地定义和操作这些变量。

枚举数据类型的取值范围是在列表中给出的，而在枚举数据对象上的运算为赋值和相等判定。

如周天的声明可以如下：enum DayOfWeek {Monday, TuesDay, Thursday, Friday, Saturday, Sunday};

3.复合数据类型

1）指针类型

指针是一个数据对象在内存中的地址。指针变量实际上是用来存放某个数据对象的地址的变量。指针加上动态的数据对象的生成和对指针值的间接引用的机制，为程序员提供了处理可变大小的数据结构的机制。

指针类型的单个数据对象可以有两种处理方式：

针对仅能引用一种类型的数据对象。指针数据对象的值被限制为指向同一类型的数据对象。在C/C++,Pascal和Ada中都采用了这一方式。例如，在C++中声明了指针变量：int* p,则p仅能指向int类型的对象，而无法指向char等其他类型的对象。

针对可以引用任何类型的对象。这种方式允许指针数据对象在程序运行时指向多种类型的数据对象。SmallTalk就是这种方式的典型代表。

在某些语言中（如C或C++），指针是可以被程序操纵的数据对象，指针可以进行相减操作，指针也可以加上某个整数，以指向当前数据对象后面（或前面）的某个数据对象。而在另外一些语言中（如Java或C#），指针是由语言实现管理的隐藏数据结构的一部分，通常被称为引用（reference）。

在指针上的操作有生成操作和选区操作。

生成操作为固定大小的数据对象分配空间，同时生成一个指向新生成对象的指针，该指针存放在指针类型数据对象中。在C语言中，系统函数malloc提供了该功能，如需C语言中动态生成一个int类型的数据对象，可以使用以下语句：int* p = malloc（sizeof（int））；

而C++、Java和C#等语言简化了malloc,提供了new函数进行分配，上述语句在C++中可以改写为：int* p = new int;.

选取操作允许沿着指针来访问指定的数据。由于指针本身也是普通的数据对象，所以指针对象也可以使用普通的选取机制进行访问。在C/C++中，使用指针访问其指向的数据对象需要使用操作符*。例如，为上面生成的int型数据对象进行赋值，可以写为*i = 0;.

2）字符串类型

字符串是由一个字符序列组成的数据对象。字符串数据类型至少有3种不同的处理：

固定长度说明。字符串具有固定长度，并且在程序中加以声明。

指定上界的可变长度。字符串对象有一个最大长度，这个最大值可以在程序中定义，而数据对象中的实际值的长度可能较短，甚至可以为零。

长度无限制。字符串数据对象可以是任意长度的字符串，字符串长度在运行过程中可以动态地变化。

C语言的情况比较特别，它把字符串看做是字符类型的线性数组，而没有专用的字符串声明。C语言的惯例是使用null字符（"\0"）作为字符串的结束符，表示字符串到此结束。

前两种字符串的处理方法可以在编译时确定，而第三种处理方法则需要在运行时动态地分配字符串存储。

对于字符串数据对象的操作一般有连接，关系操作，使用定位下标选取子串，格式化，模式匹配和动态字符串。

上一节　本书简介　下一节

结构化数据类型

2.4.2 结构化数据类型

一个数据结构是一个包含其他数据对象作为其元素或成员的数据对象。比较重要的结构化数据类型有数组和记录。

1.数组

数组是编程语言中普遍的数据结构。一个数组是包含固定数目的相同数据类型成员的数据结构。我们用数组名、数组元素的类型、维数及下标来刻画一个数组的特征。对于数组，成员数量常由下标范围序列隐式给出，数组中的每个成员都是单一数据类型，而数组的下标集通常由连续整数组成。典型的一维数组声明是Pascal的声明：n：array[–10…10] of integer.该声明定义了一个有21个成员、各成员类型为integer的Pascal数组，各成员使用下标访问，下标的范围为–10, –9,…，0,…，10.而在C语言中，数组的下标规定从0开始，如下面C语言中的数组声明：double a[10];,声明了10个成员的double类型数组，下标为0到9.

在给定下标值范围的语言中，下标范围不必从1开始，也不必是整数的范围，可能是枚举类型或枚举类型的子序列，例如：

type color = {Red, Green, Blue};

var RGB:array[color] of integer;

对数组的操作通常是使用下标进行成员的访问，这个操作成为下标标定，下标标定返回左值或者该成员代表的数据对象的位置。如果给出了数组成员的地址，取得数组成员的值（右值）就变成了一个简单的操作。

C语言为用户提供了初始化数组的手段，对数组的初始化可以写为：

float fa[]={1.0, 2.1, 3.1};

这里声明了一个有三个成员的float类型数组，其中fa[0]=1.0,fa[1]=2.1,fa[2]=3.1.另外，在C语言中，数组和指针的关系非常密切，常可以互换使用。如前面的声明的数组fa,可以被看做一个float类型的常数指针（即其指针值不可变），使用指针操作可以达到与下标操作同样的效果，如表2-5所示，左右两边的操作是两两等价的。

使用下标操作	使用指针操作
fa[1] = 1.5;	*(fa + 1) = 1.5;
float f = fa[2];	float f = *(fa + 2);
float* pf = &fa[1];	float* pf = fa + 1;

表2-5 指针操作和下标操作

2.记录

一个由固定数目的不同类型元素组成的数据结构称为记录。记录和数组都是有固定长度的线性数据结构的数据类型。记录与数组的不同之处在于：

记录的元素可以是异构的，即可以由不同类型的数据元素组成；

记录的元素使用标识符命名，而不是通过下标制定的。

记录的属性有：元素的个数；每个元素的数据类型和每个元素的选择符。记录的元素通常称为字段，元素的名称称为字段名。请看下面的C语言结构的声明（结构是C语言中的一种记录）：

```
struct student{
int id;
float score;
int class_id;
```

};

该声明定义了一个date类型的记录，包含三个元素id、score、class_id.

记录的一个基本操作是元素选择，例如student.id.记录元素的选择是需要明确给出所选择元素的名称的。

C语言所提供的初始化手段也可以用于对结构的初始化，例如：

struct student stu = { 1, 80.5, 205 };

就声明了一个类型为student的结构对象stu,并且初始化为id=3,score=80.5,class_id=205.

带有变体的记录类型使得该记录类型会发生变化。该记录类型的定义域就由所有这些变化可能产生的值的集合组成。C语言提供的联合类型（union）正是该种数据类型，联合可以作为一个单独的类型进行处理，在不同的情况下它可以存储不同类型的数据。例如，以下是在C语言中的声明：

union {

int i;

float f;

}u;

联合u既可以存储int类型数据，也可以存储float类型数据。我们可以把C中的联合看成一个特殊的结构，其中所有的成员相对于结构开始处的偏移量都为0,且结构的存储量大得足够存储最大的成员。在任一时刻，我们只能访问联合中的一个成员。

3.其他结构化数据类型

1）列表

列表是由一连串有序的数据结构构成的数据结构，通常是不定长和异构的。在JL、Lisp和Prolog等语言中，列表是基本数据对象。列表的典型变形有堆栈（stack）、队列（queue）、树（tree）、有向图和属性列表。

2）集合

集合是一种包含无序的不同值的数据对象，集合中的值是不能重复的。

上一节　本书简介　下一节

第 2 章：程序语言基础知识　作者：希赛教育软考学院　来源：希赛网　2014年01月26日

抽象的数据类型

2.4.3 抽象的数据类型

抽象的数据类型的概念为：

数据对象的集合一般使用一个或多个类型定义；

在所定义的数据对象之上的抽象操作的集合；

以上两个集合以如下原则封装起来，新的数据类型的使用者不能直接操作这种类型的数据对象，即该类型的使用者仅需要知道该类型的名字和可进行的操作的语义。

信息隐藏表示程序员定义的抽象设计中的原则：每一个这样的程序组件对于该组件的用户而言应该隐藏尽可能多的信息。当信息被封装在一个抽象中的时候，即意味着该抽象的用户无须知道所

隐藏的信息即可使用该对象，且不允许直接使用或操作隐藏的信息。一个成功的抽象就是让用户无须了解该抽象数据类型定义的数据对象的具体表示和相应操作所使用的算法。

在C++中提供了类（class）这个新的定义抽象的数据类型的机制。C++中类的成员包括成员数据和成员函数，其中成员数据相当于抽象数据类型中的数据对象的集合，而成员函数相当于抽象操作的集合。C++通过访问存取控制关键字对成员的访问权限进行限制，它们分别是public、protected和private.为实现信息隐藏，通常会把成员数据的访问权限设置为protected或private,意味着只有该类对象和派生类对象可以访问，或只有该类对象可以访问，也就是限制外部对象访问这些成员，而另一方面，把操作这些数据成员的函数的访问权限设置为public,允许外界通过这些函数对数据成员进行操作。下面是一个典型C++类的声明：

```
class stack{
private:
int a[100]={0};
int h=0;
public:
int pop ( ) ;
void push ( int item ) ;
};
```

这个声明定义了一个堆栈类，使用数组作为数据的存储方式，h是栈顶的指针，通过pop（）和push（）进行出栈和压栈的操作。但是该类的用户完全无须了解类内部是如何实现的，只需要知道pop（）和push（）的语义就可以使用该类型的数据对象了。

上一节　本书简介　下一节

类型和错误检查

2.4.4 类型和错误检查

前面讨论的类型检查都是围绕单个数据对象的类型及在该类型的操作上展开的。在程序设计中，最常用的数据对象是变量。程序语言需要确定变量是否应该具有固定的类型。类型是对变量的一个约束，如果对变量类型的约束发生在程序运行前，即变量在程序中一旦声明为一种类型，即不能改变其类型的约束，我们称类型的约束为静态的，也称为早约束。如果对变量类型的约束发生在程序运行时，即变量在程序中的类型是可以改变的，我们称类型的约束为动态的，也称为迟约束。Pascal、C语言都是静态类型约束语言，Lisp和Smalltalk则是动态类型约束语言。

进一步，我们可以把类型从单个的数据对象扩展至由数据对象和操作符组成的表达式。表达式的类型应该在编译时就已知且为固定的。

一个语言的类型系统就是一组规则，这些规则为语言中的每个表达式关联一个类型。如果一个表达式无法与某个类型相关联，那么类型系统就拒绝这个表达式。类型系统的规则也规定了每个运算符的正确使用。

1.类型检查的基本规则

类型系统中的规则基于函数的以下属性：从集合A到集合B的函数应用于集合A的元素，得到的结果是集合B的元素。

1）算数运算符

算数运算符是函数，对应每个运算符op都有一条规则规定如何由表达式E和F的类型确定表达式E op F的类型。

2）重载

运算符在不同的上下文中有不同的解释，可以接受不同类型的参数，并且可能根据参数的类型得到不同类型的结果，我们称为重载。我们常用的运算符，像+、−等都是重载的。例如运算符+,既可对整型进行加法运算，也可对实型进行加法运算。

3）隐式类型转换

当运算符需要的参数类型与实际参数类型不一致时，程序语言通常会在保证没有数据损失的前提下，自动地进行类型转换。例如，C语言表达式 2*3.14*R中，R被声明为double类型，2是int类型，3.14是浮点类型，程序语言在不进行隐式类型转换的情况下无法确定表达式的类型而必须否决此表达式。而在有隐式类型转换时，常数2和3.14都会被转换为double类型（因为double类型为此表达式中各数据对象的最高精度数据类型，这样的转换不会产生数据损失）。进行运算后，结果也为double类型。

4）多态

多态函数具有参数化的类型，参数化类型也称为类属性。多态类型允许只定义一次这类数据结构，以后可以应用到所需要的任何数据类型上。

在类似Pascal和C的命令式程序语言中，仅有的多态函数是内部数据类型的运算符。C++利用模板机制来支持参数化类型。

2.类型等价

类型等价是在程序语言的类型检查中必然会遇到的问题，类型等价一般可以分为两种。

1）结构等价

结构等价的含义是，一个变量与自己的结构等价；如果两个类型结构是对结构等价的类型应用相同类型构造符而形成的，则这两个结构等价；在type n=T（C语言中使用typedef n T）声明下，n与T等价。

2）按名等价

按名等价可分为纯名字等价，类型名与其自身等价，但结构化类型不与任何结构化类型等价；传递名字等价，类型名与其自身等价，且可以声明为与其他类型名等价；类型表达式等价，类型名仅与其自身等价，两个类型表达式等价，假设它们是从等价表达式应用相同的构造方式而形成的。

3.静态和动态类型检查

类型检查是为了防止错误，保证程序中的运算应用都是正确的。如果某个函数或操作符接收到类型错误的参数，就会出现类型错误。如果程序执行中不会出现类型错误，则我们说它是类型安全的。

所谓静态类型检查，就是在编译时，编译器通过类型系统规则，根据程序的源文本，推断每次表达式f（a）求值时，参数的类型是否正确。所谓动态类型检查，是把额外的检查代码加插到程序中，在程序运行时进行类型检查。因为需要加插代码，因此动态检查的效率不如静态检查的效率

高，并且潜藏在程序中的类型错误需要到运行时才会被检查出来。由于两者的差异，一般程序语言只检查在源程序中可以静态检查的属性，而对需要在运行时才能进行检查的属性则很少进行检查。

程序语言的类型系统还有"强"、"弱"的属性之分。这里的强、弱是一个相对概念，是指类型系统防止错误的有效性。一个强类型系统仅接受类型安全的表达式，不是强的系统就成为弱的。例如，相对来说，C++的类型系统就要比C强，也就是一部分C语言类型系统中可以接受的表达式，在C++类型系统中无法被接受。

上一节　本书简介　下一节

程序语言的控制结构

2.5 程序语言的控制结构

程序设计语言的控制结构提供了一个将操作和数据组合成程序和程序组的基本框架，考虑的是如何组织数据和操作，使其成为一个可执行的程序，这包括两个方面的内容，一是对操作执行次序的控制，我们称其为顺序控制；二是对程序中的过程间数据传递的控制，我们称其为数据控制。

控制结构可以简单地分为3类。

表达式是程序语句的基本组成，体现了程序控制和数据改变的方法。

用在语句间或一组语句中的结构，如条件语句和循环语句。

过程结构，如过程调用和协同程序。

上一节　本书简介　下一节

表达式

2.5.1 表达式

1.表达式的前缀、后缀、中缀表示法

1）前缀表示法

在前缀表示法中，是以从左到右的顺序先写操作符后写操作数，如果操作数本身是一个具有操作数的操作，则对其使用同样的规则。例如，公式，使用前缀表达式表示将变为* + a b – a b.函数调用可以看做一种前缀表达式，因为一般是把操作符函数名写在它的参数的左边，像f（a, b）这样。

以前缀表示的表达式可以在一次扫描后计算出值，前提是要明确知道每个操作符的参数的数目。用以下的算法通过使用一个执行栈，计算给定的前缀表达式P。

如果P的下一项是一个操作符，将它压入栈，并把参数计数器设置为该操作符所需要的参数的数

目n.

如果P的下一项是一个操作数，把它压入栈。

如果栈顶操作数的个数为n,则把操作符作用于这n个操作数，得出的结果替换该操作符和它所有的参数，作为操作数压入栈。

前缀表达式的计算方法意味着在每个操作数压入栈后都必须检查操作数的数目是否满足最近栈顶的操作符的要求，而后缀表达式就无须做这种检查。

2）后缀表示法

后缀表示法类似于前缀表示法，不同之处在于操作符跟在操作数之后。前面的公式使用后缀表示法时表示为a b + a b– * .由于这一点不同，在计算后缀表达式的时候，当扫描到操作符时，栈中已压入了它的操作数。因此计算后缀表达式的算法如下。

如果P的下一项是操作数，将它压入栈。

如果P的下一项是n元操作符（即需要参数个数为n的操作符），那么它的参数就是栈顶的n项，把该操作符作用于这n项，得到的结果作为操作数替代栈顶的n项。

由于后缀表达式的计算是直接的，并且易于实现，因此它是很多翻译器产生表达式代码的基础。

3）中缀表示法

中缀表示法是日常最通用的用法，但是在程序语言中使用中缀表示法会产生下面的问题。

由于中缀表示法仅适合于二元操作符，一种语言不能只使用中缀表示法，而必须结合中缀与前缀表示法。这种混合使用会使翻译过程相对复杂。

当一个以上的中缀操作符出现在表达式中时，如果不使用括号就有可能产生二义性。

考虑表达式10–2×5的值，我们会认为其值是0,而这仅仅是因为我们已经习惯把一个隐含规则应用于表达式的求值，这就是先乘除后加减。若我们把规则定义为先加减后乘除，则表达式值应为40.括号可以消除任何表达式的二义性，但在复杂的表达式中，将会导致深层次的括号嵌套而产生混乱。因此，程序语言通常都引入隐含的控制规则，使得大多数的括号的使用成为不必要，这就是操作符的优先级和结合性。

2.操作符的优先级和结合性

操作符的优先规则是指可以出现在表达式中的操作符的优先次序，操作符在该次序中的级别就是该操作符的优先级。在有处于一个以上优先级的操作符的表达式中，具有较高优先级的操作符先执行。

结合性规定了相同等级的多个操作符的操作次序。例如，在a–b–c中，应是第一个减法还是第二个减法先完成呢？通常的隐含规则是从左到右的结合。因此，a–b–c被看做（a–b）–c.如表2-6所示是C语言中的操作符的优先级和结合性。

表2-6 C语言中的操作符的优先级和结合性

优 先 级	操 作 符	操 作 符 名	结 合 性
17	Tokens, a[k], f()	文字，下标，函数调用	左
	., ->	选择	左
16	++, --	后缀增量/减量	左
15	++, --	前缀增量/减量	左
	~, -, sizeof	一元操作符，存储量	左
	!, &, *	逻辑非，取地址，指针	右
14	（类型名）	强制类型转换	左
13	*, /, %	乘，除，取余	左
12	+, -	加，减	左
11	<<, >>	移位	左
10	<, >, <=, >=	关系	左
9	==, !=	相等	左
8	&	按位与	左
7	^	按位异或	左
7	^	按位异或	左
6	|	按位或	左
5	&&	逻辑与	左
4	||	逻辑或	左
3	?:	条件	右
2	=, +=, -=, *=, /=, %=, ^=, <<=, >>=, &=, |=	赋值	右
1	,	顺序计算	左

由于程序员都知道表达式语义的基本数学模型，因此，通常的算法表达式都有良好的合理性。但是不少语言都以不同的方式扩充了操作符集合，优先性常常会被破坏。

上一节　本书简介　下一节

语句间的顺序控制

2.5.2 语句间的顺序控制

表达式的顺序控制是把操作数和操作符看做基本单位，研究操作符的计算顺序。而语句间的顺序控制是把程序语句作为基本单位，研究语句执行的顺序。程序语言对语句的执行都遵循一条隐含规则，这就是在没有其他顺序控制结构规定的情况下，按照语句在程序中的物理位置执行程序，也就是顺序执行。改变这种语句执行次序的方法是使用程序顺序控制结构，这些控制结构有跳转结构、选择结构和循环结构。

1.跳转结构

跳转结构就是令程序控制无条件地从当前语句转向给定的语句执行的控制结构，跳转语句的执行非常有效，它反映了计算机本身硬件的转移指令，如x86指令中的jmp指令。通常的跳转语句都有如下形式：goto <标号>,Fortran和C语言等都提供了goto语句。当程序控制遇到goto语句时，会转移到标号所指出的相应语句继续执行。

虽然goto语句的使用十分简单和高效，但是大量的使用会令程序控制逻辑混乱，程序变得难以理解和维护。人们已经证明可以使用顺序结构、选择结构和循环结构组成任何程序，而抛弃掉"有害的"goto语句。目前比较一致的观点是，程序员必须谨慎地使用goto语句，使用时必须考虑是否可以用更好的结构来代替。

2.选择结构

选择结构是对给定条件进行判断，然后根据结果执行不同的语句或语句块的结构。最典型的选

择结构的形式如下：

If <expr> then

<statements1>

Else

<statements2>

Endif

意味着如果expr条件为真，则执行statements1语句块，否则执行statements2语句块。在某些复杂的情况下，需要对多个条件进行判断，则if-then-else语句会进一步复杂，演化为if-then-elseif-then-else等。

在两个分支的选择结构基础上，多数语言也会提供多分支的选择结构，它在许多情况下可以改善程序的可读性。典型的多分支选择结构如下：

Switch （ <expr> ）

Case of result1:

<statements1>

Case of result2:

<statements2>

…

Default:

<default_statements>

Endswitch

虽然case控制结构的功能可以由if-then-else结构来模拟，但是case控制结构能提供更清晰的计算过程的反映。

C/C++的情况比较特别，在case结构中使用break语句表示跳出结构的控制，如果在其中一个case中没有使用break语句，则控制会顺序执行至下一个case中的语句。这个特点在为程序员带来方便的同时，也为程序员带来了麻烦。程序员疏忽漏掉的break语句会导致程序有意想不到的执行结构。因此，在C#中不允许这种case的"贯穿",而强制程序员使用goto语句跳转至相应的case标号，以保证程序员清楚地知道程序控制的行为。

3.循环结构

循环结构是根据条件重复执行指定语句的控制结构。循环结构是由循环头和循环体组成的。循环头就是循环的条件，用于控制循环的次数，循环体则是提供动作的语句。典型的循环头结构有以下几种。

1）计数器循环

这种结构需要说明一个循环计数器，并且在头部说明计数器的初值、终值和增量。典型的计数器循环的结构是Pascal的计数器循环：

For I:=0 to 30 step 2 Do <body>

该循环的头部说明了计数器为I,其初值为0,终值为30,增量为2,循环的执行次数为16次。

2）条件循环

条件循环是指在给出的条件表达式成立时，重复执行循环体的循环结构，它的头部说明了该条件表达式。这种循环期望在循环体执行的时候会改变条件测试表达式中的某个变量的值，否则循环

将永不终止。典型的条件循环的结构有两种，一种是：

While <expr> do <body>

另一种是：

Repeat <body> until <expr>

前者是先测试条件，然后执行循环体，循环体执行零次或零次以上。而后者是先执行循环体，再测试条件，循环体执行一次或以上。

3）基于数据的循环

基于数据的循环的循环次数是由数据格式决定的，例如，C#中的foreach结构：

foreach （object o in <collection>）{…}

对于每一次循环，变量o都会取得数据集中的下一个值，数据集元素的个数决定了循环的次数。

4）不定循环

如果循环结束条件过于复杂，不容易在头部表示，通常会使用在循环头部没有显示终止测试的无限循环，然后在循环体中通过条件判断退出循环。C/C++中有两种典型的不定循环结构，一是：for （；；）<statements>,二是：while（1）<statements>。

过程控制

2.5.3 过程控制

1.过程简介

在程序设计中，习惯把程序看做是层次结构，程序从主程序开始执行，然后进入各层次的过程执行，到最后返回主程序结束。过程为程序员提供了一种抽象手段，其实际上是一组输入到一组输出的映射。

过程通常有四个要素，分别为过程名，过程体，形式参数列表，返回值类型。例如C语言中的函数（即C语言中的过程）如下：

int Function1（int x, int y）；

其中Function1为函数名，（int x, int y）为形式参数列表，int是返回值类型。

2.参数传递方式

当用户调用一个过程时，就会发生通过参数传递信息的过程之间的通信。形式参数就是过程定义中用于命名所传递的数据或其他信息的标识符，而实际参数是在调用点表示向被调用过程传递的数据或其他信息的表达式。在大多数的语言中，形式参数和实际参数之间的对应关系通常按位置来确定。程序语言传递参数的方式通常有传值调用、引用调用和传值–结果调用。

1）传值调用

在按传值调用时，过程的形式参数取得的是实际参数的值。在这种情况下，形式参数实际上是过程中的局部量，其值的改变不会导致调用点所传送的实际参数的值发生改变，也就是说数据的传送是单向的。在C语言中只有按值调用的过程参数传递方式。

2）引用调用

在按引用调用时，过程的形式参数取得的是实际参数所在单元的地址。在过程中，对该形式参数的引用相当于对实际参数所在的存储单元的地址引用。任何改变形式参数值的操作会反映在该存储单元中，也就是反映在实际参数中，因此，数据的传送是双向的。C++语言既支持按值调用，也支持按引用调用。

3）传值–结果调用

传值–结果调用也称为拷入/拷出，因为初始时实际参数被拷贝到形式参数，而在过程调用结束时再把形式参数拷贝回实际参数。

3.标识符的作用域规则与活动记录

在程序设计语言中，标识符可以用于标识任何东西。某个标识符的声明为其赋予了一个新的含义，而标识符的作用域规则决定了当程序中遇到x时，适用的是哪一个声明。

作用域规则分为词法作用域规则和动态作用域规则。在词法作用域规则下，标识符出现和声明之间的约束可以在编译时静态完成，对语言的所有程序都可以做好。词法作用域规则也被称为静态作用域规则。在动态作用域规则下，名字出现与声明的约束是动态建立的，在运行中确定。

与标识符作用域的概念紧密关联的是运行时刻存储管理问题。存储管理指的是目标程序在运行时对内存的使用和再使用的方法。每个过程的活动都需要具备变量的存储，与过程的每个活动相关联的是一块为过程中声明的变量而用的存储，这一存储块被称为活动记录。活动记录通常要包含以下几部分信息：

控制链

访问链

调用点机器状态，如返回地址和寄存器值

参数

返回值

局部变量区

1）活动及其局部数据

不同的活动需要对形式参数和局部变量有不同的存储，因此，不同的活动需要有不同的活动记录，在允许递归过程的语言中，每个活动都必须有自己的活动记录。过程中的声明促使为该过程的活动记录分配存储空间。一个过程的所有活动记录布局都相同。变量的布局由其类型决定，基本数据类型用单个的存储位置，而数组则需要一系列的存储位置。

2）控制与访问链

通过在活动记录之间维护两条链的方式对它们进行管理。

控制链，也称为动态链，指向运行时调用者的活动记录。

访问链，也称为静态链，用于实现采用词法作用域的语言，指向嵌套着该过程的外层过程的活动记录。

我们使用一个下推栈来存储活动记录的信息。每次过程调用时，就生成该过程的一个活动记录，然后下推入栈，让过程的每次调用与活动记录相对应。过程执行结束就把栈顶的活动记录弹出。为了使过程能在运行时访问本次活动记录中的数据，我们设立一个指针SP,始终指向当前正在执行的过程的活动记录的某一特定单元，那么过程访问本次执行的活动记录中的数据就可以通过SP+数据偏移量来进行。

过程执行结束后，程序流通过活动记录中的控制链寻找到调用者的活动记录，把SP值设置到相应活动记录的特定单元。为了程序能够在调用点正确执行下去，在过程执行结束时还必须根据活动记录中保留的内容，恢复调用点的机器状态。

当过程需要访问使用标识符表示的外部或全局变量时，可以通过访问链，逐层地回溯至相应的外层过程的活动记录，根据变量的偏移量找出需要访问的标识符所对应的变量。例如，在下面的Pascal程序中：

```
procedure p1;
integer a, p, c;
...
procedure p2;
integer d, e;
a := c + d;
...
end;
...
p2;
end;
```

设现在运行至过程p中的a:=c+d一句，则单看p1与p2的活动记录，p2活动记录的控制链指向p1的活动记录，访问链也是指向p1的活动记录。而p1活动记录的控制链指向调用p1的过程的活动记录，而访问链则指向全局记录。当p2需要访问变量c时，程序源访问链回溯至p1的活动记录，根据c的偏移找出局部变量，然后取得其值。

我们将变量在过程中的偏移地址和该变量在访问链中所属的层（称为静态层）保存在符号表的登记项中，以便在代码生成阶段正确形成地址。

第 2 章：程序语言基础知识　作者：希赛教育软考学院　来源：希赛网　2014年01月26日

程序语言的种类、特点及适用范围

2.6 程序语言的种类、特点及适用范围

1.按程序设计范型分类

按照程序设计范型的分类，程序设计语言基本上可以分为命令式程序设计语言、函数式程序设计语言、面向对象程序设计语言和逻辑程序设计语言。

命令式程序设计语言是基于动作的语言，计算在这里被看做是一个动作的序列。这些动作能够改变变量的值，最典型的动作就是赋值。命令式程序设计语言的代表有：Fortran,Pascal和C语言等。

函数式程序设计语言的代表有Lisp、ML等。

面向对象程序设计语言中最核心的东西是对象和类的概念。面向对象的三个核心概念是封装、

继承和多态。面向对象程序设计语言的代表有C++、SmallTalk、Java等。

逻辑程序设计语言的代表有Prolog.

2.几种语言的特点及适用范围

1）C++

C++是目前最流行的程序设计语言之一，其特点是既支持面向对象程序设计的概念，也支持原来在C语言中的过程式程序设计，因此，也有人称其为混合式的面向对象语言。C++支持的面向对象概念包括类、继承、多态、模板、多重继承等。C++在增加了这些面向对象的概念支持后，生成的目标程序与C语言生成的相同功能的目标程序的效率相差不超过10%,是一种极其高效的语言。由于这些特点，以及其在各种计算机系统中被广泛支持，C++语言大量应用于系统程序的设计，包括嵌入式、桌面式和服务器操作系统的设计，大型软件系统的核心模块的设计，以及各类桌面软件的设计。

2）Java

Java是一个纯面向对象的程序设计语言。Java与C++不同，不允许有独立于类存在的过程，所有的概念都必须使用类表达。Java为了提高代码质量和安全性，去掉了C++中的指针概念，而完全使用引用的概念。另外，为了提高程序可靠性，Java提供了内存收集机制，动态内存的管理完全由系统接管。Java的一个最大的特点是一种半解释语言。编译程序首先把原程序编译为中间代码，然后通过不同平台上的Java虚拟机（Java VM）解释执行这些中间代码。较新的方式是不同平台上的Java虚拟机把这些中间代码编译为本级代码（Native Code）再执行，以提高执行速度。因此，Java语言提供了强大的跨平台能力，尤其适用于互联网上的信息系统的开发。

3）Lisp

Lisp是表处理（List Processing）的截头缩写词，它是函数式程序设计语言。在Lisp中，所有的操作均通过表操作进行，变量的赋值也是通过表操作的副作用进行的。Lisp的初始设计是为了做符号处理。它被用于各种符号演算：微分和积分演算，电子电路理论，数理逻辑，游戏推演，以及人工智能的其他领域。

4）Prolog

Prolog程序是以特殊的逻辑推理形式回答用户的查询。Prolog程序具有逻辑的简洁性和表达能力。实际应用上多用于数据库和专家系统。

上一节　本书简介　下一节

例题分析

2.7 例题分析

例题1（2011年5月试题22）

若一种程序设计语言规定其程序中的数据必须具有类型，则有利于 （22）

①在翻译程序的过程中为数据合理分配存储单元

②对参与表达式计算的数据对象进行检查

③定义和应用动态数据结构

④规定数据对象的取值范围及能够进行的运算

⑤对数据进行强制类型转换

（22） A.①②③　　B.①②④　　C.②④⑤　　D.③④⑤

例题分析：

一种程序设计语言规定其程序中的数据必须具有类型，好处如下：

（1）有利于在翻译程序的过程中为数据合理分配存储单元，因为程序设计语言为不同的数据类型规定了其所占的存储空间，如果数据类型确定，其所占的存储空间也是确定的。

（2）有利于对参与表达式计算的数据对象进行检查，因为知道数据的数据类型，我们就可以根据类型来判断该数据是否可以参与某表达式计算，如自加、自减的操作数不允许是浮点数，这只要根据数据的类型就能判断某操作数，是否能进行自加、自减运算。

（3）有利于规定数据对象的取值范围及能够进行的运算，根据数据类型，我们可以数据的存储空间，也同时能知道数据的表示范围，如C语言中的整型数据，它占两个字节（16位），能表示的数据范围就是-216至216-1.

综上所述，可知本题的正确答案选B.

例题答案：B

例题2（2011年5月试题48）

以下关于高级程序设计语言翻译的叙述中，正确的是 （48） .

（48）A.可以先进行语法分析，再进行词法分析

B.在语法分析阶段可以发现程序中的所有错误

C.语义分析阶段的工作与目标机器的体系结构密切相关

D.目标代码生成阶段的工作与目标机器的体系结构密切相关

例题分析：

在对用高级程序设计语言编写的程序进行执行时，首先是将源代码翻译成目标代码，然后在连接成可执行的二进制代码。因此在翻译阶段，目标代码生成阶段的工作与目标机器的体系结构密切相关。

例题答案：D

例题3（2011年5月试题49）

图2-7所示为一个有限自动机（其中，A是初态、C是终态），该自动机可识别（49）。

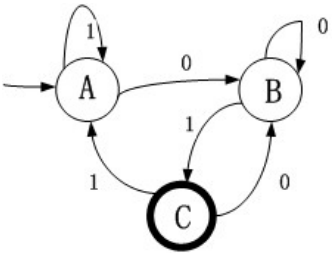


图2-7 有限自动机

（49）A.0000　　B.1111　　C.0101　　D.1010

例题分析：

本题主要考查有限自动机。

在本题中，A是初始状态，C是终止状态，通过选项中的字符串可以从初始状态到达终止状态，

则说明该字符串能被题目中的自动机识别。也可以理解为依次输入选项中的字符串，可以在该自动机中找到相应的路径。

对于选项A的字符串0000,在输入0后，从初始状态A转移到状态B,然后接着输入3个0,状态然后停留在B,而无法到达终态C,因此选项A不能被该自动机识别。

同样的道理，我们可以找到字符串0101能被该自动机识别，在输入0后，状态跳转到B,输入1则由B转至C,再输入0,又由C转至B,最后输入1,由B转至终态C.

例题答案：C

例题4（2011年5月试题50）

传值与传地址是函数调用时常采用的信息传递方式 （50） .

（50）A.在传值方式下，是将形参的值传给实参

B.在传值方式下，形参可以是任意形式的表达式

C.在传地址方式下，是将实参的地址传给形参

D.在传地址方式下，实参可以是任意形式的表达式

例题分析：

在函数调用时，系统为形参准备空间，并把实参的值赋值到形参空间中，在调用结束后，形参空间将被释放，而实参的值保持不变，这就是传值传递方式。传值传递方式中实参与形参之间的数据传递是单向的，只能由实参传递给形参，因而即使形参的值在函数执行过程中发生了变化，也不会影响到实参值。在C语言中，当参数类型是非指针类型和非数组类型时，均采用传值方式。

传地址方式把实参的地址赋值给形参，这样形参就可以根据地址值访问和更改实参的内容，从而实现双向传递。当参数类型是指针类型或数组类型时，均采用传地址方式。

区别于参数传值方式和返回值传递方式，传地址方式具有以下明显的优势。

（1）参数传值方式是主调函数与被调函数之间的单向数据传递方式，而参数的传地址方式则实现了二者之间的双向数据传递。

（2）函数的返回值每次只能把一个数据项从被调函数传递到主调函数，而参数的传地址方式却可一次性地传递多个数据项到主调函数。

例题答案：C

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年01月26日

操作系统的功能、类型和层次结构

第3章　操作系统基础知识

根据考试大纲，本章要求考生掌握以下知识点：

操作系统的内核（中断控制）、进程、线程概念；

处理机管理（状态转换、共享与互斥、分时轮转、抢占、死锁）；

存储管理（主存保护、动态连接分配、分段、分页、虚存）；

设备管理（I/O控制、假脱机）；