第1章 引 论

编译器的原理与技术具有十分普遍的意义，在计算机科学中也是比较成熟的学科，其发展历史虽然不长，但其内容却十分丰富，用途也十分广泛。它是计算机专业，特别是软件专业的主要基础课。编译器的编写涉及到程序设计语言、计算机体系结构、语言理论、算法和软件工程等学科。

本章主要内容包括：编译程序的组成、编译的各个阶段主要内容、与编译器相关的其他程序等，最后通过对一个简单编译器的介绍，让大众对编译器的全貌有一个初步的印象。

1.1 什么是编译程序

编译器是一个程序，是一个将一种语言翻译为另一种语言的计算机程序。人们要用计算机来解决问题，必须要让计算机明白要解决的问题以及如何解决这个问题。但是，人们所习惯的语言和计算机所能理解的机器指令有很大的差别，而用机器指令来描述人们想要解决的问题又十分的不便，于是计算机科学家设计了一种人们比较习惯的语言来描述要解决的问题。一般我们把这些称之为高级程序语言，而能被计算机直接理解与执行的语言即机器语言，我们称为低级机器语言。有时，也把高级程序语言称为源语言，把机器语言称为目标语言。

这一过程可以用图 1-1 表示：

图 1-1 编译器

目前，世界上存在着数千种源语言，既有 Fortran 和 Pascal 这样的传统程序设计语言，也有各计算机应用领域中出现的专用语言。目标语言也同样广泛，可以是微处理机到超级计算机的任何计算机的机器语言。不同语言需要不同的编译器，根据编译器的构造方法或者它们要实现的功能，一般可分为一遍编译器、多遍编译器、调试编译器和优化编译器等等。虽然编译器的种类很多，但它们要完成的基本任务都是相同的，只不过有不同的侧重点。

编译器是一种相当复杂的程序，其代码的长度可从 1 万行到 100 万行不等。编写甚至读懂这样的一个程序决非易事，大多数的计算机科学家和专业人员也从来没有编写过一个完整的编译器。但是，几乎所有形式的计算均要用到编译器，而且任何一个与计算机打交道的专业人员都应掌握编译器的基本结构和操作。除此之外，计算机应用程序中经常遇到的一个任务

就是命令解释程序和界面程序的开发，这比编译器要小，但使用的却是相同的技术。因此，掌握这一技术具有非常大的实际意义。

第一个编译器的问世可以追溯到 20 世纪 50 年代，IBM 的 John Backus 带领的一个研究小组对 FORTRAN 语言及其编译器进行了开发，由于当时处理中所涉及到的大多数程序设计语言的翻译并不为人所掌握，所以花费了 18 人年才得以实现。随后，Chomsky 对文法的研究，使得人们可以根据语言文法的难易程度以及识别它们所需的算法来为语言分类，其中的上下文无关文法被认为是程序设计语言中最有用的一种语言文法。语法分析的研究是在 20 世纪 60 年代和 70 年代，这一时期的研究比较完善地解决了对上下文无关文法所定义的语言的有效识别算法，现在这一部分已经是编译理论的一个标准部分。人们接着又深入研究了如何使目标代码更加有效的方法，通常称为优化技术。从第一个编译器出现至今，我们所掌握的有关编译器的知识已经得到了长足的发展。目前，我们已经系统地掌握了处理编译期间发生的许多重要技术。良好的实现语言、程序设计环境和软件工具也已经被成功开发出来。尽管近年来对此进行了大量的研究，但是基本的编译器设计在近 20 年中都没有太大的改变，它们正迅速成为计算机科学课程中的中心环节。

1.2 编译器的基本阶段

每个编译器虽然内部结构和组织方式各种各样，但主要由两部分组成：分析和综合。分析部分是对源程序进行分析，将源程序切分成一些基本块并形成源程序的中间语言表示；综合部分是在分析正确无误之后，把源程序的中间语言表示转换为所需的目标程序，综合出可以执行的机器语言程序，执行的结果应正确无误，同源程序应达到的目的完全一致。有时也把分析和综合称为前端和后端。为不同的机器编写相同源语言的编译器时，人们经常会采取这种方法：为所有的机器编写相同的编译器前端或者采用已有的编译器前端，然后为每种具体机器编写编译器的后端。同样，也可以将不同的源语言编译成同一种中间语言，对不同的前端使用相同的后端。这在很大程度上能提高编译器的代码重用和代码移植。但由于程序设计语言和机器构造的快速发展以及一些根本性的变化，这方面的工作是很难做到的，这方面的研究也只取得了有限的成果。

编译器内部包括了许多步骤或称为阶段，它们执行不同的逻辑操作。每个阶段将源程序从一种表示转换成另一种表示。编译器的一个典型阶段划分如图 1-2 所示。将这些阶段设想为编译器中一个个单独的片断是很有用的，尽管在应用中它们是经常组合在一起的，但它们确实是作为单独的代码操作过程来编写的。图 1-2 中的前三个阶段构成了编译器的分析部分，图中的符号表管理和错误处理是编译器的所有阶段都要涉及的两项活动。

编译器的若干阶段通常以一遍（pass）来实现，每遍读一次输入文件，产生一个输出文件，遍可以和阶段相对应，也可以和阶段无关，遍中通常含有若干个阶段。例如，词法分析、语法分析、语义分析以及中间代码的生成可以被组合成一遍，然后余下的部分组合成一遍。语法分析器根据它读到的标记识别语法结构，当它需要下一个标记时，它通过调用词法分析器获得所需的标记，当某一个语法结构识别后，语法分析器就调用中间代码生成器完成语义分析并生成中间代码的一部分。实际上，根据语言的不同，编译器可以是一遍通过，即所有的阶段一遍完成，但可能完成代码的质量不太高。大多数带有优化的编译器都需要超过一遍。

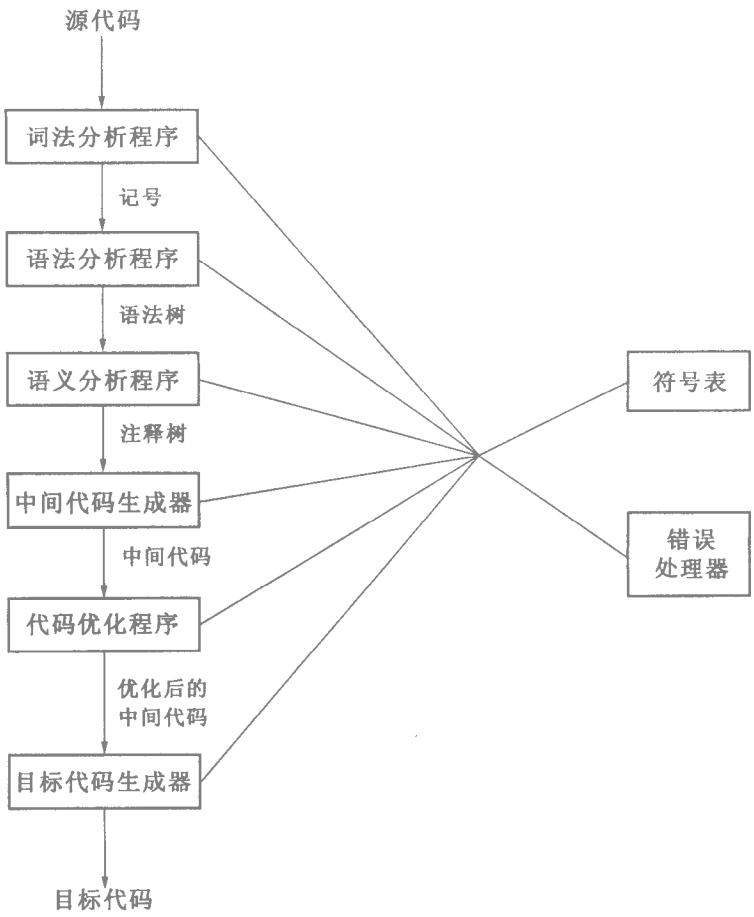

图 1-2 编译器的组成阶段

1.2.1 词法分析程序

在这个阶段编译器实际阅读源程序（通常以字符流的形式表示）。词法分析程序从左到右读入字符，按照源语言规定的词法规则，识别出一个个的单词（token），单词(token)又可以称为标记，标记同自然语言的英语单词相似。因此可以认为词法分析程序执行与拼写相似的任务。例如标识符、关键字、运算符、分隔符都是单词，编译程序把单词当作源语言的最小单位。每个标记表示一个逻辑上相关的字符序列，如关键字是一类标记，具体这类标记可能包括很多，如 if,while,for 等等，形成一个标记的字符序列称为该记号的词素（lexeme)或标记值,如 if 属于关键字这一类标记，它的词素或标记值是 if。

在 C 语言的表达式代码 `x[index] = initial + 2 * 4` 中包括了 20 个非空字符，但只有 10 个词素：

`x`	标识符
`[`	左括号
`index`	标识符
`]`	右括号

=	赋值
initial	标识符
+	加号
2	数字
*	乘号
4	数字

词法识别程序还可完成与识别标记一起执行的其他操作。例如，如果该标记值在符号表中不存在，还要在符号表中为它建立一个记录，会将该单词输入到符号表中。

1.2.2 语法分析程序

语法分析程序从词法识别程序中获取记号形式的源代码，并完成定义程序结构的语法分析(syntax analysis 或者 parsing)，这与自然语言中句子的语法分析类似。语法分析定义了程序的结构元素及其关系。通常将语法分析的结果表示为分析树（ parse tree)或语法树(syntax tree)。

例如,还是那行 C 代码，它表示一个称为表达式的结构元素，该表达式是一个由左边为下标表达式、右边为算术表达式的赋值表达式组成。这个结构可按下面的形式表示为一个分析树，如图 1-3 所示。

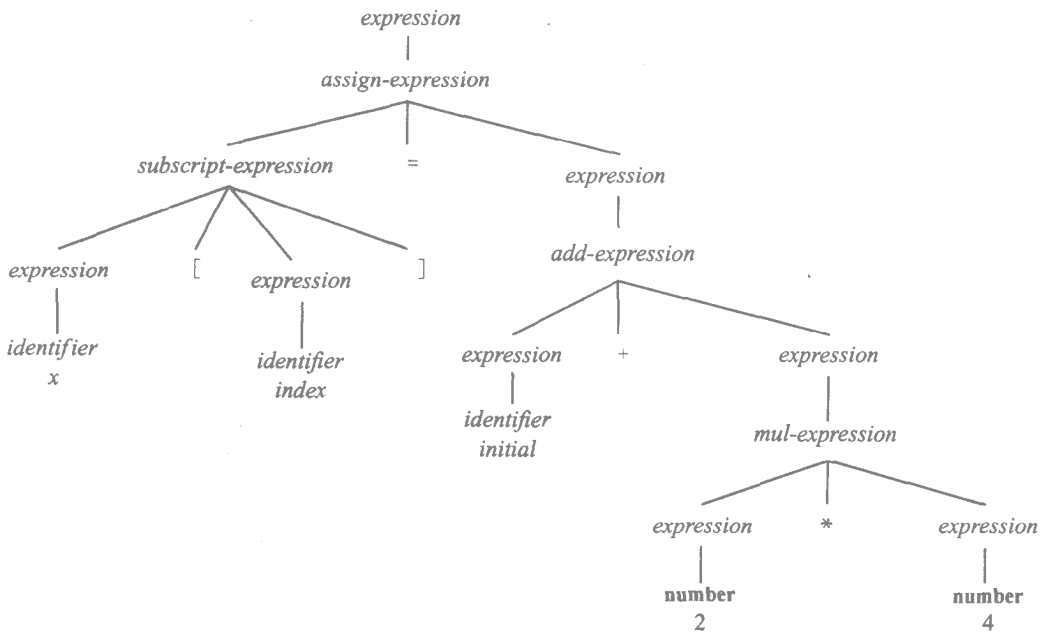

图 1-3 语法分析后生成的分析树

分析树的内部节点均由其表示的结构名标示出，而分析树的叶子则表示输入中的记号序列（结构名以不同字体表示以示与记号的区别）。图 1-4 给出了这种与该分析树相对应的更一般的内部表示语法树。语法树是分析树的压缩表示形式。在语法树中，许多节点（包括记号节点在内)已经消失。例如,如果知道表达式是一个下标运算,则不再需要用括号‘[’和‘]’来表示该操作是在原始输入中。

在语法分析程序中，如果源程序没有语法错误，就能正确地给出其分析树或语法树，否则，

要指出语法错误，并给出相应的诊断信息。

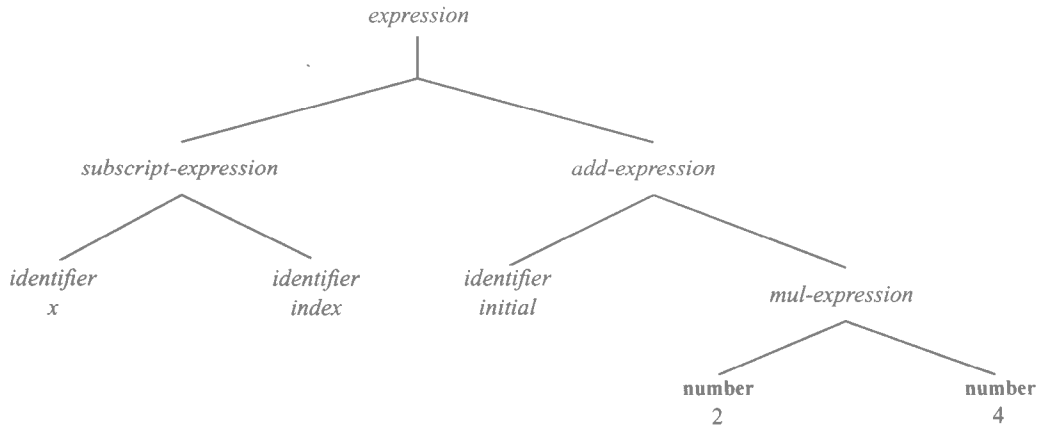

图 1-4 语法树

1.2.3 语义分析程序

程序的语义就是它的“意思”，它与语法或结构不同。程序的语义确定程序的运行，但是大多数的程序设计语言都具有在执行之前被确定而不由语法表示和由分析程序分析的特征。只有语法、语义正确的源程序才能被翻译成正确的目标代码。

语义分析的一个重要组成部分是类型检查。类型检查负责检查每个操作符的操作数是否满足源语言的说明。例如，很多程序设计语言都要求每当一个实数用于数组的索引时都要报错，也有一些程序设计语言允许一些操作数的强制类型转换。例如，一个二元算术操作数可以分别是一个整数和实数，那么，编译器就要把整数强制转换成实数。

在上面的 C 表达式 `x[index] = initial + 2 * 4` 中，该行分析之前收集的典型类型信息可能是：`x` 是一个整型值的数组，`index` 则是一个整型变量。接着，语义分析程序将用所有的子表达式类型来标注语法树，并检查赋值是否使这些类型有意义了，如若没有，则声明一个类型匹配错误。在上例中，所有的类型均有意义，有关语法树的语义分析结果可用以下注释了的树(图 1-5)来表示：

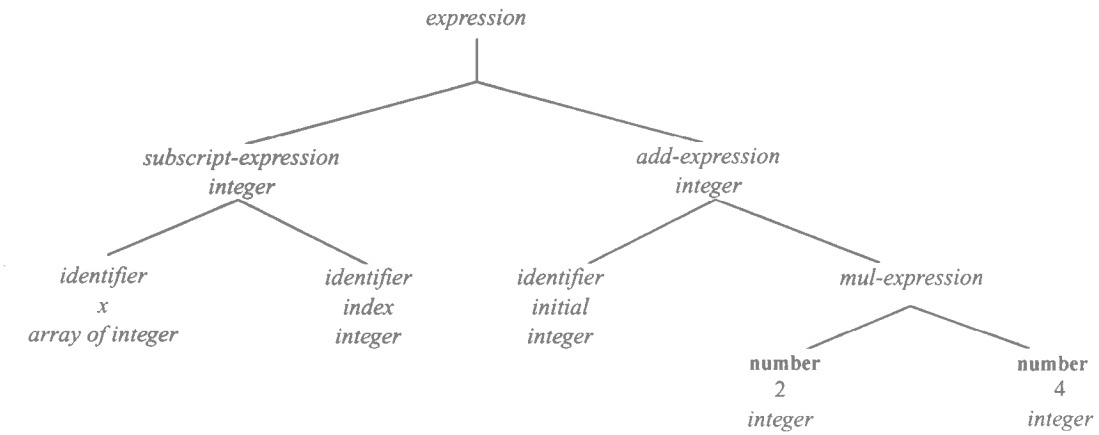

图 1-5 具有语义信息的语法树

1.2.4 中间代码生成器

编译器在完成语法分析和语义分析以后，把源程序转换为一个中间语言表示。一般来说，这种中间语言表示应有两个重要的性质：易于产生和易于译成目标程序。中间代码生成器可以看成是一个独立的编译阶段，也可以是语义分析阶段的一部分工作。

源程序的中间表示形式有很多。例如，像“三地址码”，是一种类似于某种机器的汇编语言，这种机器的每个存储单元的作用类似寄存器。三地址码由指令序列组成，每个指令最多有三个操作数。上面的 C 代码表达式可转换成下面的三地址代码：

```
temp1 := 8    (经过了源代码层次的优化)
temp2 := initial + temp1
x[index] = temp2
```

后面的有关章节中将具体讨论编译程序中使用的几种中间语言表示及有关中间语言代码的生成算法。

1.2.5 代码优化程序

代码优化阶段主要为了提高中间代码的质量，以便提高目标程序的运行速度。代码优化可以在语义分析之后完成，也可以在中间代码生成后完成，语义分析之后的代码优化可能只依赖于源代码。在 1.2.4 节中就包括了一个源代码层次的优化。每个编译器不论在已完成的优化种类方面还是在优化阶段的定位中都有很大的差异。进行了大量优化工作的编译程序通常称作“优化编译程序”，这种编译程序的大部分编译时间都花在这个阶段。在实用上仍有许多简单有效的优化算法，能够显著地提高目标程序的运行速度而不明显地降低编译速度。

上面的中间代码经过优化可以改为：

```
x[index] := initial + 8
```

1.2.6 目标代码生成器

目标代码生成器由得到的中间代码生成目标机器的代码。尽管大多数编译器直接生成目标代码，但是为了便于理解，本书用汇编语言来编写目标代码。正是在编译的这个阶段中，目标机器的特性成了主要因素。例如，整型数据类型的变量和浮点数据类型的变量在存储器中所占的字节数或字数也很重要。

代码生成器要对源程序中使用的变量分配寄存器，然后为每一条中间语言指令选择合适的机器指令，包括确定机器指令的操作码或编址模式。在上面的例子中，必须决定怎样存储整型数来为数组索引生成代码。例如，下面是所给表达式的一个可能的样本代码序列（在假设的汇编语言中）：

```
MOV R0, index     ; value of index → R0
MUL R0, 2         ; double value in R0
```

```
MOV R1, &x          ; address of x → R1
ADD R1, R0          ; add R0 to R1
MOV R2, initial     ; value of initial → R2
ADD R2, 8           ; add 8 to R2
MOV *R1,R2          ; value of R2 → address in R1
```

在以上代码中，为编址模式使用了一个类似 C 的协定，因此 &a 是 a 的地址（也就是数组的基地址），*R1 则意味着间接寄存器地址（因此最后一条指令将 R2 的值存放在 R1 包含的地址中）。这个代码还假设机器执行字节编址，并且整型数占据存储器的两个字节（所以在第二条指令中用 2 作为乘数）。

在这个阶段中，编译器还会有很多代码优化的工作可做，以改进由代码生成器生成的目标代码。这种改进包括选择编址模式以提高性能，将速度慢的指令更换成速度快的，以及删除多余的操作。在上面给出的目标代码中，还可以做许多更改。在第二条指令中，利用移位指令替代乘法（通常地，乘法很费时间），还可以使用更有效的编址模式（例如用索引地址来执行数组存储）。使用了这两种优化后，目标代码就变成：

```
MOV R0, index       ; value of index → R0
SHL R0              ; double value in R0
MOV R2, initial     ; value of initial → R2
ADD R2, 8           ; add 8 to R2
MOV &x[R0],R2       ; value of R2 → address x +  R0
```

编译过程的前面几个阶段，包括词法分析、语法分析、语义分析直至中间代码生成，依赖于被编译的源语言，因此，将这几个阶段归在一起，称为编译程序的前端。前端还包括与这几个部分有关的符号表操作，代码优化中与目标机无关的部分也属于前端。前端中各个阶段中的错误处理部分也属于前端。编译过程中同目标机有关的部分属于编译程序的后端，后端与源语言无关，只同中间语言有关。后端包括代码优化中涉及目标机的部分、目标代码生成以及相关的错误处理及符号表操作。

1.2.7 符号表管理

在编译过程中，源程序中的标识符及其各种属性都要记录在符号表中，这些属性提供标识符的存储分配信息、类型信息、作用域信息等等。对于过程标识符，还有参数信息，包括参数的个数及类型，参数结合的方式等等。

符号表是一个数据结构。每个标识符在符号表中都有一条记录，记录的每个域对应于该标识符的一个属性。这种数据结构允许我们快速地找到每个标识符的记录，并在该记录中快速地存储和检索信息。

当源程序中的一个标识符被词法分析器识别出来时，词法分析器将在符号表中为该标识符建立一条记录，而它的属性信息将由后面的各个阶段写入符号表，并在需要时使用。例如，当进行语义分析和中间代码生成时，需要知道标识符是哪种类型，以便知道源程序是否正确地使用了这些标识符。因此，会涉及到很多查询和填入属性的问题。

1.2.8 错误处理

编译器的一个最为重要的功能是对源程序中错误的反应。几乎在编译的每一个阶段中都可以诊察出错误来。这些静态或称为编译时（compile-time）的错误(static error)必须由编译器来报告，而编译器能够生成有意义的出错信息并在每一个错误之后恢复编译是非常重要的，也就是说，编译器不能因为有错误的出现而使编译工作停止。

在词法分析中可能发现字符拼写错误；在语法分析阶段会检查出单词串违反语言的结构规则的错误；在语义分析中，编译程序进一步查出语法上虽正确但含有无意义的操作的成分，如两个标识符相加，一个是数组名，一个是过程名，虽然语法上允许，但语义上不允许。我们会在相应的章节中介绍各阶段的错误处理方法。

1.3 与编译器有关的程序

1.3.1 解释程序

解释程序(interpreter）是如同编译器的一种语言翻译程序。它与编译器的不同之处在于：它执行程序但不作翻译，它并不把源程序翻成可执行的目标代码。

从原理上讲，可以直接通过解释程序执行同一源代码，也可以先对源代码进行编译，然后再执行生成的机器码。但是根据所使用的语言和翻译情况，很可能会选用解释程序而不用编译器。解释代码的运行速度比编译代码慢，因为解释程序在每次执行程序的语句时，都必须对各语句逐一进行解释，然后再执行所需的操作，而编译代码只执行操作，不解释语句。在解释程序中存取变量也比较慢，这是因为必须在运行时，而非编译时重复实现从标识符到存储位置的映射。

有一些语言,如 BASIC,LISP 等，既有解释性程序，又有编译性程序。前者主要用于程序开发和调试，而后者主要用于生产运行。一般会根据需要来确定是使用解释程序还是翻译程序。例如，我们经常解释 BASIC 语言而不是去编译它。类似地，诸如 LISP 这种函数语言也常常是被解释的。解释程序也经常用于教育和软件的开发，此时的程序很有可能被翻译若干次。而另一方面，当执行的速度是最为重要的因素时就使用编译器。事实上，编译与解释并非总有明显的分界线，解释程序具有许多与编译器共享的操作，而两者之间也有一些混合之处。

1.3.2 汇编程序

汇编语言的实质和机器语言是相同的，都是直接对硬件操作，只不过指令采用了英文缩写的标识符，更容易识别和记忆。它同样需要编程者将每一步具体的操作用命令的形式写出来。由于是直接面对低层的操作，所以，汇编语言到机器语言的翻译比较简单，该翻译程序我们特称为汇编程序(assembler)。有时，编译器会生成汇编语言以作为其目标语言，然后再由一个汇编程序将它翻译成目标代码。

1.3.3 链接和装入程序

一个程序要想在内存中运行，除了编译之外还要经过链接和装入(linker and loader)这两个步骤。随着 C语言这种支持分别编译的程序设计语言的流行，一个完整的程序往往被分割

为若干个独立的部分并行开发，而各个模块间通过函数接口或全局变量进行通讯。这就带来了一个问题，编译器只能在一个模块内部完成符号名到地址的转换工作。为了解决不同模块间的链接问题就使用链接和装入程序，用它主要是完成符号解析和重定位。

符号解析：就是当一个模块使用了在该模块中没有定义过的函数或全局变量时，编译器生成的符号表会标记出所有这样的函数或全局变量，而链接器的责任就是要到别的模块中去查找它们的定义，如果没有找到合适的定义或者找到的合适的定义不唯一，符号解析都无法正常完成。

重定位：编译器在编译生成目标文件时，通常都使用从零开始的相对地址。然而，在链接过程中，链接器将从一个指定的地址开始，根据输入的目标文件的顺序以段为单位将它们一个接一个地拼装起来。除了目标文件的拼装之外，在重定位的过程中还要完成两个任务：一是生成最终的符号表；二是对代码段中的某些位置进行修改，所有需要修改的位置都由编译器生成的重定位表指出。

1.3.4 预处理器

预处理器(preprocessor) 产生编译器的输入，它是在真正的翻译开始之前由编译器调用的独立程序。一般具有以下功能：删除注释，进行宏处理、包含文件处理等等。预处理器可由语言(如 C) 要求或以后作为提供额外功能（如 FORTRAN ）的附加软件。

1.3.5 调试程序

调试程序(debugger) 是可在被编译了的程序中判定执行错误的程序，它可以访问带以下命令的 DSP 硬件逻辑(start，stop，read/write，register，reset)， 此外,程序和数据存储器、寄存器和其他相关目标均可看到。调试程序也可完成算法分析及把数据下载到目标板上。

调试程序一般经常与编译器一起放在 IDE 中。运行一个带有调试程序的程序与直接执行不同，这是因为调试程序保存着所有的或大多数源代码信息（诸如行数、变量名和过程）。它还可以在预先指定的位置暂停执行，并提供有关已调用的函数以及变量当前值的信息。

1.4 一个简单的编译程序

1.4.1 语言概述

我们在这里举一个简单编译器的例子作为这一章的结束。该语言是一个表达式语言，设定语言为一个由整数，＋，－，＊，／组成的整数算术表达式。为了把该语言描述清楚，一般来说要包括以下几个方面。词法：即字符集中的符号如何组成单词，它是构成程序的最小单元。语法：程序由单词串组成，单词是以什么样的规则构成程序中的一个个语句，一般我们用上下文无关文法来表示。语义：是确定合法程序的含义，在定义语言的语义时，一般采用非形式化方法来描述。

1.4.1.1 单词的描述

语言的字符集是 ASCII 码，单词包括整数（由数字 0,1,2,3,…,9 组成）、操作符(＋,－,

*,/)及左右小括号((,))。

1.4.1.2 语法的定义

我们采用上下文无关文法来描述语言的语法，该文法将一直作为我们描述语法的一种方法贯穿本书。这里我们只是给出一个简单的例子，有关该文法的具体定义将在后面的章节中展开。

描述该表达式语言的文法如下：

```
1. expression → expression + item
2.            | expression - item
3.            | item
4. item → item * factor
5.      | item / factor
6.      | factor
7. factor → number
8.        | (expression)
```

这是用 Backus 范式描述表达式语法的语法公式。它是由一组公式组成，每个公式定义一个概念，被定义的概念的名字出现在箭头的左部，箭头右部则是被定义的内容。

1.4.1.3 语义的描述

该语言的语义就是用于整数计算的计算规则，因为该语言比较简单，所以可说明的语义不是很多。

1.4.2 词法分析程序

我们首先给出该表达式语言的单词或标记的集合。标记包括以下：

1. NUM(整数),它由“0”—“9”组成。
2. +,-,*,/,(,),每个是一类标记,标记值就是它自身。

程序 1.1 lex.h **单词的定义**

```
1  # define  EOI      0     /*  输入结束  */
2  # define  PLUS     1     /*     +      */
3  # define  MINUS    2     /*     -      */
4  # define  TIMES    3     /*     *      */
5  # define  DIVI     4     /*     /      */
6  # define  LP       5     /*     (      */
7  # define  RP       6     /*     )      */
8  # define  NUM      7     /*  十进制数  */
9
10 extern   char    * yytext;
```

```
11  extern   int     yyleng;
12  extern   int     yylineno
```

lex.h 中有另外三个外部变量，是词法分析程序用来传递信息给语法分析程序的，yytext 是一个字符指针，指向当前的词文，但它不以“\0”结尾；yyleng 是一个整形变量，它的值是识别出的标记中的字符个数；yylineno 也是一个整形变量，它的值是当前的输入的行号。这些全局变量用于词法分析程序的自动生成器 LEX 中。有关 LEX 我们在后面的章节中会有更详细的介绍。

程序 1.2 是有关这些标记的一个词法分析程序。它使用了一个简单的缓冲输入系统，一次从标准输入设备输入一行字符，然后从该行中识别出一个个单词，一次识别一个，当前行用完时输入下一个。使用缓冲输入系统主要有两个原因：一个是为了速度，另一个是在分析过程中便于字符的超前和回送，而在缓冲区是很容易做到这一点的，只要移动一下相应的指针就可以了。

下面对该程序作一个简单的分析。程序在第一次被调用时，在 14 行的 current 被初始化，指向一个空串，18 行的 while 循环有两个目的，首先置 current 指向缓冲的第一个位置，25 行的 if 语句把输入行读入缓冲区，若输入结束或输入错，在输入缓冲区置“\0”字符，并以 EOI 作为单词返回。否则，行号加 1，并跳过全部空行。该循环到 input_buffer 含一非空行时终止。从 37 行开始的 for 循环实际识别单词。当该循环终止时，yytext 指向该单词的第一个字符，yyleng 中存储着该单词的长度。第二次调用该程序时，14 行使 current 跳过前一词文，如果此时输入缓冲区还未用完，18 行的 while 测试失败，程序跳到 37 行的单词识别部分，如果到达缓冲的末端时，lex() 才读入下一行，此时，current 是“\0”。当 lex() 返回时，返回值是下一个单词，current 及 yytext 指向单词的第一个字符，yyleng 是该单词的长度，yylineno 是该单词所在的行号。

与 lex.c 一起运行的还有两个子程序，一个是 match(token)。match 如果是初次运行，调用 lex()读下一个单词，并把识别出的单词与参数 token 比较，一致时返回 true 非初次运行的话，只把上次读出的单词与参数 token 作比较。另一个是 advance()，该程序调用 lex()读下一单词，识别出的单词存储于静态整变量 lookahead。这两个程序配合使用，解决了超前查看一个单词的问题。

程序 1.2　lex.c　词法分析程序

```
 1  # include " lex.h "
 2  # include < stdio.h >
 3  # include < ctype.h >
 4
 5  char     * yytext =  " " ;          / * 词文（不以 \ 0 结束） * /
 6  char     yyleng =  0 ;              / * 词文长度 * /
 7  char     yylineno =  0 ;            / * 输入行号 * /
 8
 9  lex ( )
10  {
```

```
11      static char input_buffer [ 128 ] ;
12      char    * current ;
13
14      current =  yytext +  yyleng ;/ *  跳过当前词文  * /
15
16      while ( 1 )          / *  读下一个单词  * /
17            {
18            while ( ! * current )
19            {
20                    / *  读新行，跳过该行前面的空白字符  * /
21
22
23
24                  current =  input_buffer ;
25                  if ( ! gets ( input_buffer ) )
26                  {
27                  * current =  ' \ 0 ' ;
28                  return EOI ;
29                  }
30
31                  + + yylineno ;
32
33                  while ( isspace ( * current ) )
34                        + + current ;
35            }
36
37            for ( ; * current ; + + current )
38            {
39            / *  读下一个单词  * /
40
41                  yytext =  current ;
42                  yyleng =  1 ;
43
44                  switch ( * current )
45                  {
46                        case EOF : return EOI ;
47                        case ' ; ' : return SEMI ;
48                        case ' + ' : return PLUS ;
49                        case ' (' : return MINUS ;
50                        case ' * ' : return TIMES ;
```

```
51                      case ' / ' : return DIVI ;
52                      case ' ( ' : return LP ;
53                      case ' ) ' : return RP ;
54
55                      case ' \ n ' ;
56                      case ' \ t ' ;
57                      case ' ' : break ;
58
59                      default ;
60                      if ( ! isalnum ( * current ) )
61                            fprintf ( stderr , "Ignoring illegal input
                              < % c> \n ", * current);
62                      else
63                      {
64                            while ( isalnum ( * current ) )
65                                  + + current ;
66
67                            yyleng = current - yytext ;
68                            return NUM_OR_ID ;
69                      }
70
71                      break ;
72                }
73          }
74      }
75 }
76 static int Lookahead = - 1 ;     / *   向前看单词 * /
77
78 int match ( token )
79 int token ;
80 {
81     / * 如果 " token " 与当前的向前看单词匹配，则返回真 . * /
82
83     if (Lookahead = = - 1)
84           lookahead = lex( ) ;
85
86     return token = = Lookahead ;
87 }
88
89 void advance ( )
```

```
{
    /* 把下一个输入字符置入 Lookahead. */
    Lookahead = lex();
}
```

1.4.3 递归下降语法分析

递归下降分析的概念比较简单：将每个语法规则看作为识别该结构的一个过程的定义。语法规则的左部是一个非终结符号，代表一个语法单位，对应该非终结符号有一个子程序，可以用来识别该符号定义的结构或单词序列。语法规则的右部说明了这个子程序的代码结构：由终结符号（标记）或非终结符号的序列组成，终结符号必须能和词法分析程序识别出的标记相匹配，非终结符号则进一步调用对应于该符号的子程序。

我们所用的文法还是表达式文法，只是为了使该文法适合于递归下降的方法来分析，对该文法作了相应的修改。修改后的文法如下：

```
1. expression   → item expression'
2. expression'  → + item expression'
3.              | - item expression'
4.              | ε
5. item         → factor item'
6. item'        → * factor item'
7.              | / factor item'
8.              |ε
9. factor       → number
10.             | (expression)
```

上面语法中的‘|’表示‘或者’的意思，‘ε’表示空串的意思。

对于语法公式 *expression → item expression'* 有如下的对应子程序生成：

```
expression( )
{
    item();
    expression'();
}
```

程序 1.3 parser.c 递归下降分析器

```
1   /* 基本分析器不含代码生成 */
2
3   # include < stdio.h >
4   # include " lex.h"
```

```
5
6   expression ( )
7   {
8       / * expression   → item expression' * /
9
10      item ( )
11      expr _ prime ( )
12  }
13
14  expr _ prime ( )
15  {
16      / * expression'→ + item expression'
17      *            | - item expression'
18      *          | ε
19      * /
20
21      if ( match ( PLUS || MINUS ) )
22  {
23      advance ( ) ;
24      item ( ) ;
25      expr _ prime ( ) ;
26  }
27  }
28
29  item ( )
30  {
31      / * item  → factor item' * /
32
33      factor ( ) ;
34      item _ prime ( ) ;
35  }
36
37  item _ prime ( )
38  {
39      / * item' → * factor item'
40        *        | / factor  item'
41        *    | epsilon
42        * /
43
44  if ( match (TIMES  ||  DIVI) )
```

```
45    {
46        advance ( ) ;
47        factor ( ) ;
48          item _ prime ( ) ;
49          }
50    }
51
52    factor ( )
53    {
54        / * factor → number
55          *      | LP expression RP
56          * /
57
58        if ( match ( NUM ) )
59        advance ( ) ;
60        else  if ( match (LP ) )
61              {
62              advance ( ) ;
63              expression ( ) ;
64              if ( match ( RP ) ) ;
65                    advance ( ) ;
66              else
67                    fprintf ( stderr , " % d : Missmatched parenthesis \n " ,
                      yylineno ) ;
68              }
69              else
70                    fprintf ( stderr , " %  d Number   expected \ n " ,
                      yylineno ) ;
71    }
```

1.4.4 中间代码生成及优化

上一节中的递归下降分析器只是进行了语法分析，如果被分析的源程序没有错误，那就一直到语法分析器运行完毕，都不会有错误的信息出现。那么，这部分工作也只是进行了识别，但我们的目的还不仅仅如此，是要生成一个和源程序等价的可运行程序，所以，编译程序到这儿仅仅是完成了一部分的工作，还有一部分工作需要做，包括中间代码生成、代码优化和机器代码或汇编代码的生成。

1.4.4.1 中间语言

因为后面的章节中会对中间语言作进一步的论述，我们这儿只是为了这个表达式语言而

用，所以，仅简单介绍一下使用的一种中间语言。此处采用的是三地址码的汇编码，这种代码的形式是

OP A. B, C

其中 OP 是操作码（或伪操作码），A,B 是进行指定操作的操作数，C 是存放操作结果的存储单元。操作数可以是变量名或直接是一个数，对于某些操作 OP,A,B,C 三个中可根据需要少用或不用，如停机指令就是 HALT，不需要操作数。中间语言的输出格式可以是字符串。

在代码生成过程中，一般会需要临时分配的存储单元来保存计算过程中的中间结果。编译程序要对这些临时变量进行管理。一般在具体实现时，通常会使用寄存器或运行栈中专门分配一块区域供临时变量使用。在中间代码阶段不一定要确定临时变量的具体安排，我们只是假定它们都被适当地声明过了。约定临时变量的名字的形式是 t0,t1,t2,…。

例如，给定一个表达式：1+2*3

它的中间代码输出是：

```
:= , 1, ,t0
:= , 2, ,t1
:= ,3, ,t2
* , t1,t2,t1
+ , t0,t1,t0
```

每个临时变量都保存当前子表达式的计算结果。有时会同时使用几个临时变量，以便适应运算的优先与结合规则。有关临时变量，应该有一个比较好的管理方法，应考虑到回收问题，用过的临时变量名在不再使用时可另作分配。下面的程序 1.4 就是两个用于临时变量分配的管理子程序。

程序 1.4　alloc_name.c　临时变量分配子程序

```
1   char *name[ ] = {"t0","t1","t2","t3","t5","t6","t7"};
2   char **Namep = names;
3
4   char *newname ( )
5   {
6       if ( Namep >= &Names[ sizeof (Names) / sizeof ( *Names ) ] )
7       {
8           fprintf ( stderr , " % d : Expression too complex \n " ,
            yylineno);
9           exit ( 1 ) ;
10      }
11
12  return ( *Namep ++ ) ;
13  }
```