编译程序的设计与实现

刘磊　　金英

张晶　张荷花　单郸

编 著

吉林大学计算机科学与技术学院

2004 年 2 月

简　介

编译程序是计算机系统不可缺少的部分，是程序设计者的必备工具。学习并掌握编译程序的构造原理和实现技术，能够增强对程序设计语言的理解，提高程序设计、尤其是大型软件的设计能力。

本教材以一个简单的具有嵌套过程定义的过程式语言SNL作为教学语言，详细介绍了该语言编译程序的设计和实现方法，并且对已经实现的编译程序的源代码分阶段进行了详细的分析，尤其是对编译器程序的组成、实现算法、所用数据结构以及各功能部分所采用的编译技术都作了详细的介绍，并配有相应的框图说明。学生在学习《编译原理》课程的同时，可以配合本教材中编译实例的分析，进一步理解和掌握编译器程序的构造原理和实现方法。此外，通过对本教材中所提供的编译程序源代码的阅读和改进，可以大大地提高程序设计能力。本教材是一本难得的编译程序实例分析和教学辅导教材。

目　录

前　　言

编译原理课程是计算机科学与技术专业最为重要的专业课之一，掌握编译方法和技术是每一个优秀计算机软件专业人员的必备素质。对于计算机专业的学生来说，也许将来只有少部分人从事专业开发编译程序、维护编译程序的工作，但学好这门课程是十分重要的，体现在：

1．学好编译课程可以加深对程序设计语言的理解，因为设计一个编译程序，需要准确认识程序语言的语法和语义，了解目标机及目标代码的结构，这些知识对于学习新的程序设计语言是非常有帮助的。

2．因为编译程序本身是一个十分庞大而复杂的系统软件，涉及到许多复杂的数据结构和实现算法，若能系统全面的掌握编译技术，必将大大提高程序设计能力，特别是开发大型软件的能力。

3．编译技术可以应用于许多实际的软件开发工作中，如软件开发平台、软件自动生成、模式匹配等许多方面。

4．编译课程的学习可以培养学生的抽象思维能力，掌握形式化描述技术，这种思想和方法可能对今后从事的软件开发工作产生深远的影响。

5．编译程序是一种元级程序，即它处理的对象就是程序，因此学习编译原理和实现技术，对于我们掌握元级程序设计方法十分有帮助。

许多学过编译原理课程的同学都会感到有些困惑：首先觉得编译课程难度较大，不易掌握；其次虽然掌握了编译的各个阶段的技术，但缺乏对编译程序的总体理解，各部分之间难以衔接。针对这种情况，我们编写了本教材。

本教材设计了简单的具有嵌套过程的程序设计语言 SNL(Small Nested Language)。该语言具有标准数据类型和结构数据类型，可以嵌套定义过程，过程的参数可以分为值参和变参两种形式，控制语句和 PASCAL 语言基本相同，因此除指针类型外，SNL 语言具备了过程式语言的基本特征，以它作实例语言，构造其编译程序，使得绝大多数编译技术可以在该编译程序中得以体现。

本教材介绍了 SNL 语言的编译程序的实现方法，并且对 SNL 编译程序的源代码分阶段进行了详细的分析，尤其是对编译程序的组成、数据结构、所用算法以及各功能部分所采用的编译技术都作了详细介绍，并配有框图说明。学生在学习《编译原理》课程的同时，可以通过本教材中编译实例的分析，进一步理解和掌握编译器程序的构造原理和实现方法，同时，通过对本教材中提供的编译程序源代码的阅读和改进，可以大大地提高程序设计能力。

本书分为十二章，具体每章的主要内容如下：

第一章：简单介绍一下编译程序的构造原理；

第二章：介绍本教材使用的类 PASCAL 高级程序设计语言 SNL；

第三章：介绍 SNL 编译程序的总体结构和程序的组成；

第四章～第九章：分别介绍了 SNL 编译程序的词法分析、语法分析、语义分析、中间代码生成、中间代码优化和目标代码生成部分的实现原理、具体的程序说明以及源程序清单；

第十章：介绍了 SNL 编译程序的虚拟目标代码的解释程序的实现方法；

第十一章：总结了前面介绍的 SNL 编译程序，并指出了可以改进的方面和一些其它的应用实例；

第十二章：介绍了 SNL 编译器系统 SNLC(Small Nested Language's Compiler)的简明使用手册，以方便读者更好地使用 SNL 的编译器系统。

本书是作者根据多年教学实践经验总结而成，已在计算机科学与技术专业本科生教学中使用多次，效果良好。北京工业大学计算机学院蒋宗礼教授认真审阅了本书的全部初稿，并提出了很多中肯的修改意见，吉林大学计算机学院金成植教授在本书的成书过程中也给予了多方面的指导。此外，吉林大学计算机学院软件自动化实验室的教师、博士生和研究生的帮助和建议也对本书的出版起了积极的作用。在此，作者向所有对本书编写工作给予支持和帮助的人表示衷心的感谢。由于编者水平有限，时间匆忙，肯定会有不少缺点和不足，敬请读者多提宝贵意见，以便在适当的时间再作修订补充。

第一章 编译原理概述

1．1 高级程序设计语言的实现

我们都知道，计算机硬件系统只能执行机器指令程序，而通常的应用程序都是用高级程序设计语言编写的。因此，要想使高级程序设计语言编写的程序能够被计算机识别并运行，必须有这样一种程序，它能够把用汇编语言或高级程序设计语言写成的程序转换成等价的机器语言程序，我们把这种转换程序统称为翻译程序（Translator）。其中完成从高级程序设计语言编写的程序到等价的机器语言程序的转换任务的翻译程序称为编译程序(Compiler)，简称为编译器。

高级程序设计语言的实现通常有三种方式：

1．编译方式：编译程序的输入是高级程序设计语言程序，称为源程序(Source Program)，输出是低级程序设计语言程序，称为目标程序(Target Program)，其功能如图 1.1（a）所示：

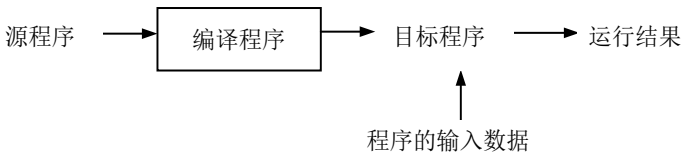


图 1.1(a) 高级程序设计语言的编译实现方式

2．解释方式：解释器的输入是源程序和输入程序的数据，其方法是边翻译边执行，当翻译结束时，计算结果也会随之计算出来，其功能如图 1.1(b)所示：

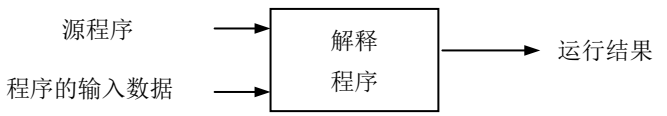


图 1.1(b) 高级程序设计语言的解释实现方式

3．转换方式：假如我们要实现 L 语言，现在有 L′ 语言的编译器，那么可以把 L 语言程序转换成 L′ 语言的程序，再利用 L′ 语言的编译器实现 L 语言，其功能如图 1.1(c)所示：

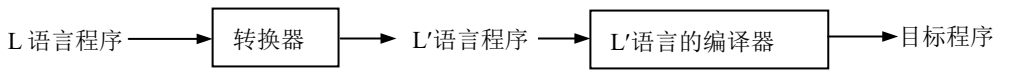


图 1.1(c) 高级程序设计语言的转换实现方式

在这三种实现方式中，编译方式和解释方式最为常见。在编译方式下，源程序

的执行是分阶段进行的。一般地，首先进行“翻译”，把用高级语言或汇编语言编写的程序翻译成与之等价的机器语言程序，然后再运行机器语言程序，最终得到运行结果。前一阶段的翻译工作由翻译程序（编译程序或汇编程序）来完成，后一阶段的运行计算需要有运行程序来配合完成。所谓运行程序是指运行目标代码程序时必须配置的各种子程序的全体，通常以库子程序的形式存在，如一些连接装配程序以及一些链接库等。在解释方式下，源程序的执行只有一个阶段－－解释执行阶段。具体讲，完成解释工作的解释程序将按源程序中语句的动态顺序逐句地进行分析解释，并立即予以执行。解释方式和编译方式的最根本区别在于：在解释方式下，并不生成目标代码，而是直接执行源程序本身。

1．2 编译程序的组成

不同的编译器都有各自不同的组织结构和实现方式，需要根据源语言和目标语言的特点以及要求来确定编译器的设计实现方案。因此，并没有一种固定的编译器的程序结构，但功能结构几乎都是一致的。这里说的功能结构是指编译器内部都做哪些工作，以及它们彼此之间的关系。图 1.2 说明了一般的编译程序的功能结构。

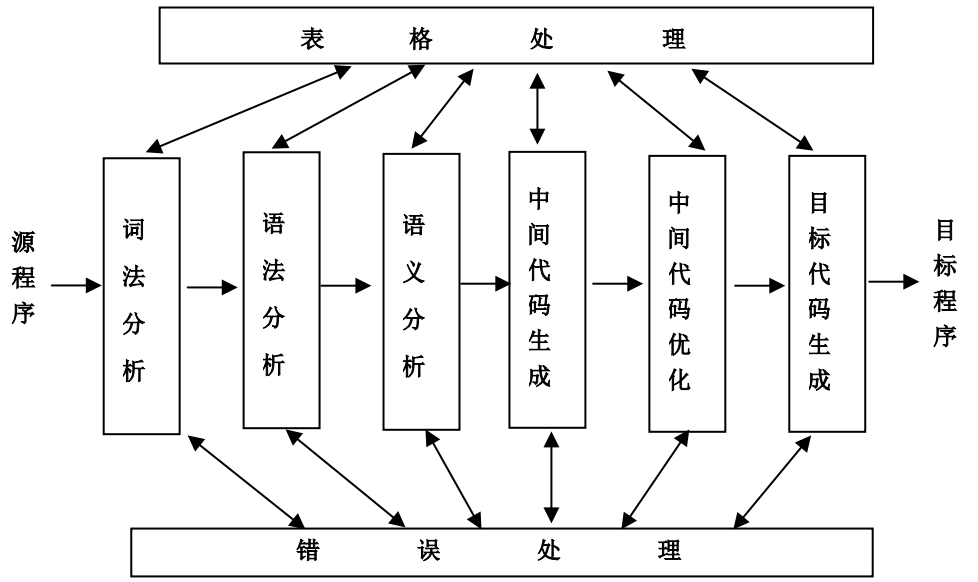


图 1.2 编译器的功能结构图

其中各部分完成的主要任务如下：

（1）词法分析：根据源语言的词法规则，扫描源程序的字母（ASCII 码）序列，并识别出一个一个具有独立意义的最小语法单位，即“单词”，同时确定该单词的种类（如标识符，或界限符，或常数，等等），并把每个单词的 ASCII 码序列替换为统一的标准形式－－所谓的机内表示 TOKEN 形式（这种形式既刻画了单词本身，又刻画了它所具有的属性），同时词法分析还要完成词法错误的检查

以及去掉注释等。词法分析阶段不依赖于语言的语法定义。

（2）语法分析：根据源语言的语法规则，逐一地扫描源程序的 ASCII 码序列或者是词法分析后的 TOKEN 序列（前者情形，词法分析程序将作为语法分析程序的子程序），以确定源程序的具体组成结构（语法结构）。分析时如发现有不符合语法规则的地方，则打印出错位置和错误类型，以便程序员进行修改；如果未发现语法错误，则将源程序转换成能够表示程序结构的语法树的形式。很多编译程序在进行语法分析的同时还要完成其他的工作，但要注意到如果语法有错误那么其他工作就没有意义了。

（3）语义分析：根据源语言的语义规则对语法分析得到的语法树进行语义检查（确定类型，类型和运算的合法性检查等），并建立标识符的符号表等各种信息表。只有在没有语义错误的情况下才继续下面的编译过程，否则将不进行下面的编译工作。语义分为静态语义和动态语义，编译阶段一般只检查静态语义，而静态语义检查中重点是类型检查。

（4）中间代码生成：扫描对象通常是语义分析后的结果，这一部分把源程序的 TOKEN 序列转换成更接近于目标代码的中间代码（如三元式或四元式的序列）。如果不搞优化，那么这部分的工作可以不要。

（5）中间代码优化：扫描对象是中间代码，任务是把原中间代码转换成可产生高质量目标代码的中间代码，其中传统的优化工作包括常表达式优化、公共子表达式优化、不变表达式外提和削减运算强度等。

（6）目标代码生成：扫描对象是中间代码，任务是从中间代码产生目标代码。这一部分的工作与目标机紧密相连，其它部分的工作几乎与目标机无关。

（7）错误处理：错误包括词法错误、语法错误、静态语义错误、动态语义错误。其中动态语义错误只能在运行目标程序时才能发现，在编译程序的各个阶段都要有错误处理部分，词法错误和语法错误都集中一次完成检查，而语义检查则分散在以后的各个阶段在完成别的工作时顺便完成。

（8）表格管理：较大的编译程序用到很多表格，甚至可以达几十种表。这些表将会占用大量存储区。因此，合理设计和使用表格是编译程序的一个重要问题。表格的分类、表的结构、表格的处理都是所要考虑的基本问题。有些表格以后可能无用，这时应及时删除他们以扩大空区。如果处理不好表格的管理工作，就可能出现因表区溢出而不得不停止编译的现象，但这时可能还有空区。为了合理的管理表格（构造、查找、更新）和表区，不少编译程序都设立一些专门子程序（称为表格管理程序），它们专门负责管理表格。

1．3 编译程序的实现

编译程序是一个相当复杂的系统程序，通常有上万甚至几万条指令。随着编译技术的发展，编译程序的开发周期也在逐渐缩短，但仍然需要很多人年，而且工作很艰巨，正确性也不易保证。

要实现一个编译程序，通常需要做到：

- 对源语言的语法和语义要有准确无误的理解，否则难以保证编译程序的正确性。
- 对目标语言和编译技术也要有很好的了解，否则会生成质量不高的目标代码。
- 确定对编译程序的要求，如搞不搞优化，搞优化搞到哪一级等等。
- 根据编译程序的规模，确定编译程序的扫描次数、每次扫描的具体任务和所要采用的技术。
- 设计各遍扫描程序的算法并加以实现。

一般开发编译程序有如下几种可能途径：

1．转换法（预处理法）：

假如我们要实现L语言的编译器，现在有L’语言的编译器，那么可以把L语言程序转换成L’语言的程序，再利用L’语言的编译器实现L语言，这种方法通常用于语言的扩充。如对于C++语言，可以把C++程序转换成C程序，再应用C语言的编译器进行编译，而不用重新设计和实现C++编译器。常见的宏定义和宏扩展都属于这种情形。

2．移植法：

假设在A机器上已有L语言的编译程序，想在B机器上开发一个L语言的编译程序。这里有两种实现方法：

（1） 实现方法一：最直接的办法就是将A机的代码直接转换成B机代码。

（2） 实现方法二：假设A机和B机上都有高级程序设计语言W的编译程序，并且A机上的L语言编译程序是用W语言写的，我们可以修改L编译程序的后端，即把从中间代码生成A机目标代码部分改为生成B机的目标代码。这种在A机上产生B机目标代码的编译程序称为交叉编译程序（Cross Compiler）。

3．自展法：

实现思想：先用目标机的汇编语言或机器语言书写源语言的一个子集的编译程序，然后再用这个子集作为书写语言，实现源语言的编译程序。通常这个过程会分成若干步，像滚雪球一样直到生成预计源语言的编译程序为止。我们把这样的实现方式称为自展技术。

4．工具法：

70年代随着诸多种类的高级程序设计语言的出现和软件开发自动化技术的提高，编译程序的构造工具陆续诞生，如70年代Bell试验室推出的LEX,YACC至今还在广泛使用。其中LEX是词法分析器的自动生成工具，YACC是语法分析器的自动生成工具。然而，这些工具大都是用于编译器的前端，即与目标机有关的代码生成和代码优化部分由于对语义和目标机形式化描述方面还存在困难，虽有不少生成工具

被研制，但还没有广泛应用。

5．自动生成法：

如果能根据对编译程序的描述，由计算机自动生成编译程序，是最理想的方法，但需要对语言的语法、语义有较好的形式化描述工具，才能自动生成高质量的编译程序。目前，语法分析的自动生成工具比较成熟，如前面提到的YACC等，但是整个编译程序的自动生成技术还不是很成熟，虽然有基于属性文法的编译程序自动生成器和基于指称语义的编译程序自动生成器，但产生目标程序的效率很低，离实用尚有一段距离，因此，要想真正的实现自动化，必须建立形式化描述理论。

1．4 其他相关程序

当今的编译程序一般都不产生目标机的机器代码，而是产生汇编语言的目标代码。因此，要得到可执行的目标代码，还需要汇编程序的支持。汇编程序产生的是可重定位的机器代码，因此只有进行装配和连接之后方可执行。

综上所述，一个高级程序设计语言的程序转换成可执行的机器代码一般包括以下过程：如图1.3所示。

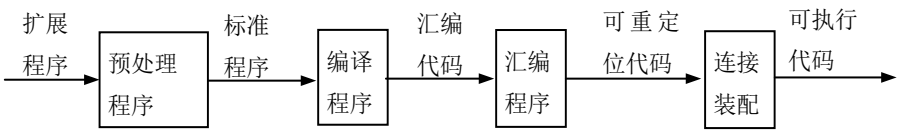


图 1.3 从扩展程序到可执行代码的过程

1．预处理程序：把某个高级程序设计语言 L 的扩充转换成 L 的标准版本。预处理程序一般作为一个单独的程序，由编译程序在翻译之前调用。

预处理程序又可以有以下几种类型：

（1） 宏处理器：宏可以看作是长结构的缩写。

（2） 文件包含：把头文件加入程序中。

（3） 语言扩展：通过内置的宏，给语言增加新的功能。

2．汇编程序：一个翻译程序，把汇编语言代码翻译成可重定位的机器代码。它的实现相对简单，大部分工作是一一对应地把符号化指令转换成相应的二进制码，把代表存储单元的每个标识符转换成相应的相对地址（偏移量）。

3．连接程序：把若干个可重定位的机器代码文件（或编译后的目标代码/汇编后/系统提供的库函数）构成一个可执行的目标代码文件。连接过程完全依赖于操作系统和处理器的细节。

4．装配程序：把可重定位的目标代码中需要重定位的地址进行修改（通过与一个给定的基地址/始地址相加）。装配程序或在操作系统中或与连接程序共同合在一起，并不单独使用。

第二章 SNL 语言介绍

2．1 SNL 语言的特点

SNL(Small Nested Language)语言是我们自行定义的教学模型语言，它是一种类 PASCAL 的“高级”程序设计语言。SNL 语言的数据结构比较丰富，除了整型、字符型等简单数据类型外，还有数组、记录等结构数据类型，过程允许嵌套定义，允许递归调用。SNL 语言基本上包含了高级程序设计语言的所有常用的成分，具备了高级程序设计语言的基本特征，实现 SNL 的编译器，可以涉及到绝大多数编译技术。通过对 SNL 语言编译程序的学习，我们可以更加深入更加全面的掌握编译程序的构造原理。但为了教学方便起见，略去了高级程序设计语言的一些复杂成分，如文件、集合、指针的操作等。

2．2 SNL 语言的词法

2．2．1 语言的字符表

程序是由字符组成的，每一种语言都对应一个字符表。SNL 语言的字符表定义如下：

<字符表> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y |z |
A| B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
W | X| Y | Z|
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+ | - | * | / | < | = | (|) | [|] | . | ; |EOF | 空白字符 |{ | }| ′ | :

注：在程序中，英文字母区分大小写；保留字只能由小写字母组成。

2．2．2 单词的巴科斯范式

SNL 编译系统的单词符号分类如下:

- 标识符 (ID)
- 保留字 (它是标识符的子集, if,repeat,read,write，…)
- 无符号整数 (INTC)
- 单字符分界符 (+ , - , * , / , < ,= ,(,) , [,] , . , ; , EOF ,空白字符)
- 双字符分界符 (:=)
- 注释头符 ({)

- 注释结束符　　(})
- 字符起始和结束符　(')
- 数组下标界限符　　(..)

上述各类符号的巴科斯范式如下：

```
< 标识符 >              ::=   字母 { 字母 | 数字 }
< 无符号整数 >          ::=   数字 { 数字 }
< 单字符分界符 >        ::=   + | - | * | / | ( | ) | [ | ] | ; | . | < | = | EOF | 空白字符
< 双字符分界符 >        ::=   : =
< 注释头符号 >          ::=   {
< 注释结束符号 >        ::=   }
< 字符标示符 >          ::=   '
< 数组下标界限符 >      ::=   ..
<字母>                  ::=   a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t |
                              u | v | w | x | y |z | A| B | C | D | E | F | G | H | I | J | K | L |
                              M | N | O | P | Q | R | S | T | U |V | W | X| Y | Z
<数字>                  ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2．3　SNL 语言的语法

2．3．1　语法的非形式说明

一个 SNL 程序是由程序头、声明部分和程序体组成的。声明部分包括类型声明、变量声明和过程声明。SNL 语言的语法规定可以声明整型（integer）、字符类型（char）、数组类型以及记录类型的类型标识符和变量。过程声明包括过程头、过程内部声明和过程体部分，过程声明内部还可以嵌套声明内层过程。程序体由语句序列构成，语句包括空语句、赋值语句、条件语句、循环语句、输入输出语句、过程调用语句和返回语句。表达式分为简单算术表达式和关系表达式。

1．程序头的形式是：关键字 **program**　后面跟着程序名标识符；

2．类型定义的形式是：类型名标识符=类型定义，其中类型定义可以是类型名或者是结构类型定义，类型名可以是基本类型，或者是前面已经定义的一个类型标识符；

3．变量声明的形式是：类型名后面跟着用逗号隔开的变量标识符序列；

4．过程声明的形式是：关键字 **procedure** 跟着过程名标识符，以及参数声明、类型定义、变量说明、内层过程声明和程序体；

5．程序体的形式是：以关键字 **begin** 开头，关键字 **end** 结尾，中间是用分号隔开的语句序列（注意最后一条语句后不加分号），最后用“**.**”标志程序体的结束。

2．3．2 语法的形式定义

下面将用上下文无关文法给出 SNL 语言的语法。实际上应该把它看成是 Token 化的语法，因为其中已省略了标识符和整型常数的产生式部分，而且关键字的 Token 是用相应的大写表示的。

SNL 语言的上下文无关文法

总程序：

1) Program ::= ProgramHead DeclarePart ProgramBody .

程序头：

2) ProgramHead ::= **PROGRAM** ProgramName
3) ProgramName ::= **ID**

程序声明：

4) DeclarePart ::= TypeDecpart VarDecpart ProcDecpart

类型声明：

5) TypeDecpart ::= ε
6) | TypeDec
7) TypeDec ::= **TYPE** TypeDecList
8) TypeDecList ::= TypeId = TypeDef ; TypeDecMore
9) TypeDecMore ::= ε
10) | TypeDecList
11) TypeId ::= **ID**

类型：

12) TypeDef ::= BaseType
13) | StructureType
14) | **ID**

15) BaseType ::= **INTEGER**
16) | **CHAR**

17) StructureType ::= ArrayType
18) | RecType

19) ArrayType ::= **ARRAY [**low**..**top **] OF** BaseType
20) Low ::= **INTC**
21) Top ::= **INTC**

22) RecType ::= **RECORD** FieldDecList **END**
23) FieldDecList ::= BaseType IdList **;** FieldDecMore
24) | ArrayType IdList **;** FieldDecMore

25) FieldDecMore ::= ε
26) | FieldDecList
27) IdList ::= **ID** IdMore
28) IdMore ::= ε
29) | **,** IdList

变量声明：

30) VarDecpart ::= ε
31) | VarDec
32) VarDec ::= **VAR** VarDecList
33) VarDecList ::= TypeDef VarIdList **;** VarDecMore

34) VarDecMore ::= ε
35） | VarDecList
36）VarIdList ::= **ID** VarIdMore
37）VarIdMore ::= ε
38） | **,** VarIdList

过程声明：

39) ProcDecpart ::= ε
40) | ProcDec
41) ProcDec ::= **PROCEDURE**
ProcName **(** ParamList **)** **;**
ProcDecPart
ProcBody
ProcDecMore

42) ProcDecMore ::= ε
43) | ProcDeclaration
44) ProcName ::= **ID**

参数声明：

45) ParamList ::= ε
46) | ParamDecList
47) ParamDecList ::= Param ParamMore

48) ParamMore ::= ε
49) | **;** ParamDecList
50) Param ::= TypeDef FormList
51) | **VAR** TypeDef FormList
52) FormList ::= **ID** FidMore
53) FidMore ::= ε
54) | **,** FormList

过程中的声明部分：

55) ProcDecPart ::= DeclarePart

过程体：

56) ProcBody ::= ProgramBody

主程序体：

57) ProgramBody ::= **BEGIN** StmList **END**

语句序列：

58) StmList ::= Stm StmMore
59) StmMore ::= ε
60) | **;** StmList

语句：

61) Stm ::= ConditionalStm
62) | LoopStm
63) | InputStm
64) | OutputStm
65) | ReturnStm
66) | ID AssCall

注：因为赋值语句和函数调用语句的开始部分都是标识符，所以将赋值语句和调用语句写在一起。

67) AssCall ::= AssignmentRest
68) | CallStmRest

赋值语句：

69) AssignmentRest ::= VariMore **:=** Exp

条件语句：

70) ConditionalStm ::= **IF** RelExp **THEN** StmList **ELSE** StmList **FI**

循环语句：

71) LoopStm ::= **WHILE** RelExp **DO** StmList **ENDWH**

输入语句：

72) InputStm ::= **READ (** Invar**)**
73) Invar ::= **ID**

输出语句：

74) OutputStm ::= **WRITE(** Exp **)**

返回语句：

75) ReturnStm ::= **RETURN**

过程调用语句：

76) CallStmRest ::= **(** ActParamList **)**
77) ActParamList ::= ε

78) | Exp ActParamMore
79) ActParamMore ::= ε
80) | , ActParamList

条件表达式：

81) RelExp ::= Exp OtherRelE
82) OtherRelE ::= CmpOp Exp

算术表达式：

83) Exp ::= Term OtherTerm
84) OtherTerm ::= ε
85) | AddOp Exp

项：

86) Term ::= Factor OtherFactor
87) OtherFactor ::= ε
88) | MultOp Term

因子：

89) Factor ::= **(** Exp **)**
90) | **INTC**
91) | Variable
92) Variable ::= **ID** VariMore
93) VariMore ::= ε
94) | **[** Exp **]**
95) | **.** FieldVar
96) FieldVar ::= **ID** FieldVarMore
97) FieldVarMore ::= ε
98) | **[** Exp **]**

99) CmpOp ::= **<**
100) | **=**

101)AddOp ::= +
102) | **-**

103) MultOp ::= ***
104） | /

例： 图 2.1 给出一个 SNL 程序例子，该程序实现了冒泡排序算法：

```
program   bubble
var   integer   i,j,num;
      array [1..20] of integer   a;
procedure   q(integer num);
var   integer i,j,k;
      integer t;
begin
  i:=1;
    while i < num do
      j:=num-i+1;
      k:=1;
      while k<j   do
          if a[k+1] < a[k]
           then
                  t:=a[k];
                  a[k]:=a[k+1];
                  a[k+1]:=t
           else   temp:=0
           fi;
      k:=k+1
      endwh;
  i:=i+1
  endwh
end

begin
    read(num);
    i:=1;
    while i<(num+1)   do
      read(j);
      a[i]:=j;
      i:=i+1
    endwh;
    q(num);
    i:=1;
    while   i<(num+1) do
        write(a[i]);
        i:=i+1
    endwh
end.
```

图 2.1 实现冒泡排序算法的 SNL 程序

2．4 SNL 语言的语义

SNL 是一个简化的类 PASCAL 语言，它保留了 PASCAL 语言的很多特性，因此其语义信息也非常丰富。

在 SNL 语言中，标识符可以表示变量名，类型名，记录的域名，过程名等；SNL 语言的程序包含显式的声明部分，所有标识符必须先声明再使用；在同一层过程里声明的标识符不允许重名，即标识符不允许重复声明；标识符的作用域和 PASCAL 语言的嵌套作用域规定相同。上述是与标识符有关的语义约定，要求 SNL 语言的程序员在编程的时候加以遵守，对于违反这些约定的程序，编译器会在语义分析时，通过构造标识符的符号表和相关信息表进行语义检查，标识出现的错误，以便程序员对程序进行修改。

SNL 语言规定，过程声明可以嵌套，即一个过程里还可以声明其他的过程；过程的参数有两类：一类是值参数，即值引用型；一类是变量参数，即地址引用型参数；过程允许递归调用；一个过程中出现的变量只能有三种形式，一是形参，二是局部量，三是全程量（即在该过程的外层声明的静态变量）；过程调用时，参数的个数和类型必须相符；这些规定与 PASCAL 语言完全相同，同样要在语义分析时进行检查。

SNL 语言的基类型只有整型和字符型，但是用户可以自己定义复杂类型，如数组类型和记录类型等，对于这些构造的类型也有一定的限制：构造的数组类型的基类型只能是整型和字符型；记录类型的域标识符不能声明为记录类型，即记录类型不允许嵌套定义。另外，SNL 语言提供了标准的算术运算加减乘除，算术表达式的运算分量和运算结果只能是整型；SNL 语言没有提供布尔类型，当然也没有提供布尔运算，只能通过整型值的关系运算表示条件，而且这种关系运算只能出现在条件语句和循环语句的条件表达式中。上面几条是与类型相关的语义信息，编译器也将在语义分析的时候，通过查找符号表，对标识符的类型和表达式的类型信息进行语义检查，标记出现的类型错误。

第三章 SNL 语言的编译程序简介

3．1 SNL 编译程序功能结构

SNL 编译系统把 SNL 源程序编译成虚拟目标机上的目标程序（这里我们采用的是由 Kenneth C.Louden 提出的 TM 虚拟计算机，该机上的指令系统是 TM 机指令），然后再由解释程序对该目标程序进行解释执行,才能得到最终的运行结果。

为了便于学习、理解和掌握编译技术，SNL 编译程序采用的是多遍扫描。

一个编译程序可以在语义检查后直接生成目标代码，也可以产生中间代码进行优化，然后再生成目标代码。在 SNL 编译器中，我们针对这两种不同的生成目标代码方法进行了不同的实现方法：

其中前三遍扫描完成的工作是一样的：

第一遍，词法分析，以源程序为输入，生成单词的内部表示 Token 序列。

第二遍，语法分析，以 Token 序列为输入进行语法分析并生成整个源程序的语法分析树。在 SNL 的编译器中，为了让读者掌握更多的语法分析技术，采用了两种语法分析方法实现：①LL(1)和 ②递归下降法。两种语法分析方法的输入和输出结果是一样的。编译过程中，学生可以选择其中一种进行语法分析。

第三遍，语义分析，以语法树为输入生成标识符的属性符号表以及相关的各类信息表，如数组信息表等，并进行相关的语义检查。

此后，

第一种方式输入的是语法树，输出的是 TM 机上的目标代码。

第二种方式第四遍扫描，以语法树为输入，生成四元式中间代码。

第五遍扫描，实现了常表达式节省。

第六遍扫描，实现了公共表达式的节省。

第七遍扫描，实现了循环不变式的外提。

第八遍扫描，输入的是优化后的中间代码，输出的是 TM 机上的目标代码。

在实际编译中，上述几种优化以及中间代码生成可以一遍扫描完成，但这里为了学习方便，分成了多遍扫描来实现。

此外，在每遍扫描的处理过程中，如遇到语法语义错误，则要转到相应的错误处理程序处理。编译的全过程都和各类表格有着密切联系，经常需要查找、添加等操作，一般也应有专门程序来管理。

1．SNL 编译器的功能结构框图:

（分支 1）

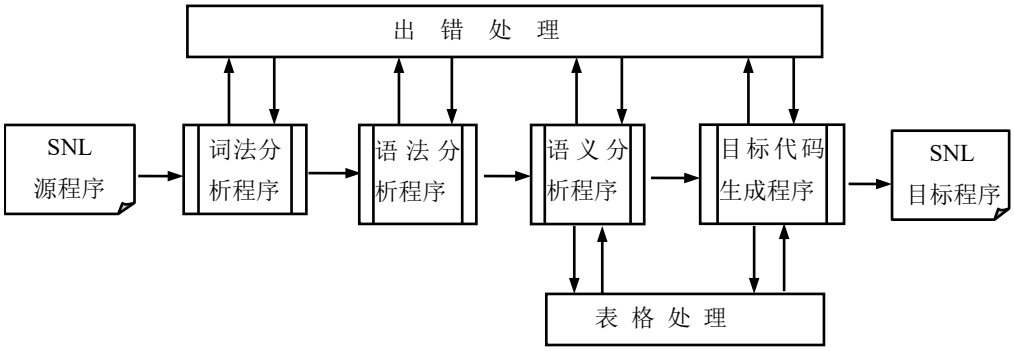


图 3.1 不带优化的 SNL 编译程序功能结构图

（分支 2）

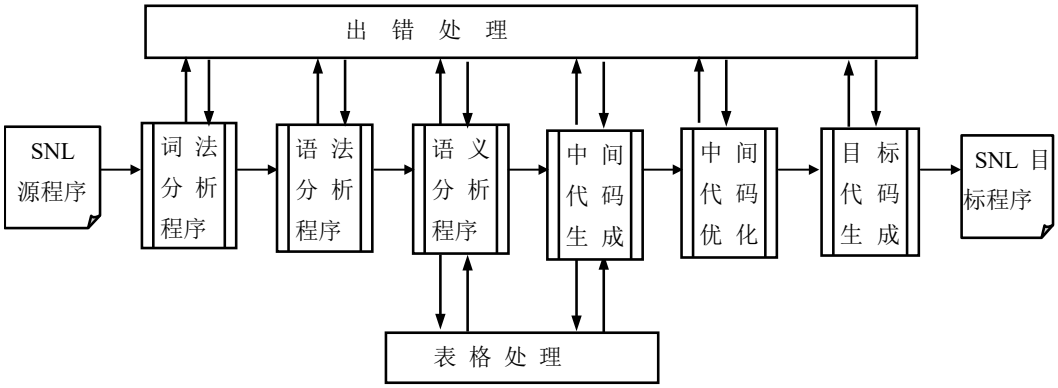


图 3.2 带优化的 SNL 编译程序功能结构图

2．SNL 目标语言程序解释执行示意图：

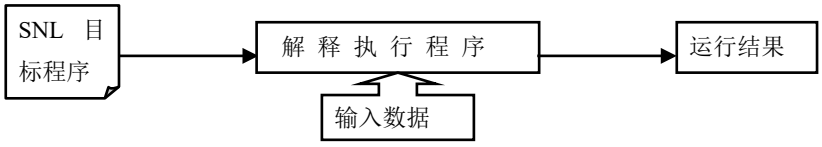


图 3.3 SNL 目标语言程序解释执行示意图

3．2 SNL 编译器的开发环境

任何软件都是在一定的硬件和软件的环境支持下开发的，SNL 编译器也不例外。SNL 编译器的开发环境如下：

硬件配置：

CPU：PIII1.0G

内存：256M

软件配置：

操作系统：中文 WINDOWS2000

编程语言：Visual C++6.0

为了保证编译器的正常运行，建议使用者最好使用与开发环境相兼容的硬件配置和软件环境，特别是操作系统。

3．3 SNL 编译器程序包

SNL 编译程序是用 C 语言编写的,整个编译器程序有 16 个文件(包括主程序文件),具体文件列表与所声明函数、变量说明如下表：

阶段	文件	函数	函数功能	参数	参数含义
词法分析阶段	scanner	getNextChar	取得下一非空字符	空	
		ungetNextChar	回退字符	空	
		reservedLookup	查找保留字	字符串 s	要检查的标识符
		getTokenlist	取得所有单词	空	
	util	printTokenlist	显示词法分析结果	空	
		printToken	显示一个单词	token	给出要打印的 token
		ChainToFile	将链表中的 token 存入文件	Chainhead	Token 链表头指针
		syntaxError	处理语法错误	message	要显示的错误提示信息
		Match	处理终极符匹配	expected	给出期待的单词符号
		Program	处理总程序	空	
		ProgramHead	处理程序头	空	
		DeclarePart	处理声明部分	空	
		TypeDec	处理类型声明	空	
		TypeDeclaration	处理类型声明	空	
		TypeDecList	处理类型声明	空	
		TypeDecMore	处理类型声明	空	

递归下降法语法分析阶段	parse	TypeId	处理类型标识符	t	要生成的当前语法树节点指针
		TypeName	处理类型名	t	要生成的当前语法树节点指针
		BaseType	处理基本类型	t	要生成的当前语法树节点指针
		StructureType	处理结构类型	t	要生成的当前语法树节点指针
		ArrayType	处理数组类型	t	要生成的当前语法树节点指针
		RecType	处理记录类型	t	要生成的当前语法树节点指针
		FieldDecList	处理域	空	
		FieldDecMore	处理域	空	
		IdList	处理标识符	t	要生成的当前语法树节点指针
		IdMore	处理标识符	t	要生成的当前语法树节点指针
		VarDec	处理变量声明	空	
		VarDeclaration	处理变量声明	空	
		VarDecList	处理变量声明	空	
		VarDecMore	处理变量声明	空	
		VarIdList	处理标识符	t	要生成的当前语法树节点指针
		VarIdMore	处理标识符	t	要生成的当前语法树节点指针
		ProcDec	处理函数声明	空	
		ProcDeclaration	处理函数声明	空	
		ParamList	处理参数声明	t	要生成的当前语法树节点指针
		ParamDecList	处理参数声明	空	
		ParamMore	处理参数声明	空	
		Param	处理参数声明	空	
		FormList	处理参数声明	t	要生成的当前语法树节点指针
		FidMore	处理参数声明	t	要生成的当前语法树节点指针
		ProcDecPart	处理函数中的声明	空	
		ProcBody	处理函数体	空	
		ProgramBody	处理程序体	空	
		StmList	处理语句序列	空	
		StmMore	处理语句序列	空	
		Stm	处理语句	空	
		AssCall	处理赋值和调用	空	
		AssignmentRest	处理赋值	空	
		ConditionalStm	处理条件语句	空	
		LoopStm	处理循环语句	空	
		InputStm	处理输入语句	空	

		OutputStm	处理输出语句	空	
		ReturnStm	处理返回语句	空	
		CallStmRest	处理调用语句	空	
		ActParamList	处理实参	空	
		ActParamMore	处理实参	空	
		Exp	处理表达式	空	
		Simple_exp	处理简单表达式	空	
		Term	处理项	空	
		Factor	处理因子	空	
		Variable	处理变量	空	
		VariMore	处理变量	t	要生成的当前语法树节点指针
		FieldVar	处理域变量	空	
		FieldVarMore	处理域变量	t	要生成的当前语法树节点指针
		Parse	语法分析主函数	空	
LL1法语法分析阶段	zparse	CreatLL1Table	创建 LL1 分析表	空	
		Gettoken	从 token 序列文件取单词	p	指向取得的单词的指针
		syntaxError	语法错误处理	message	要显示的错误提示信息
		Process1——process104	处理 1——104 个产生式	空	
		Predict	根据产生式编号选择处理过程	num	查 LL(1)分析表得到的编号
	Util(用于LL1语法分析)	Push	符号栈压栈	整型量 i	标志是终极符还是非终极符
				整型量 j	当前终极符或者非终极符
		Pop	符号栈弹栈	空	
		PushPA	语法树栈压栈	t	指向树节点的指针的地址
		PopPA	语法树栈弹栈	空	
		readStackflag	读符号栈栈顶标志	空	
		readstackN	读符号栈顶非终极符	空	
		readstackT	读符号栈顶终极符	空	
		PushOp	操作符栈压栈	t	指向操作符节点的指针
		PopOp	操作符栈弹栈	空	
		ReadOpStack	读操作符栈顶元素	空	
		PushNum	操作数栈压栈	t	指向操作数节点的指针
		PopNum	操作数栈弹栈	空	
		ReadNextToken	从文件中读出下一个 token	空	
		newRootNode	创建语法树根结点	空	
		newPheadNode	创建程序头类型语法树节点	空	
		newDecANode	创建声明标志类型语法树节点	kind	给出具体是哪种声明节点
		newTypeNode	类型声明标志类型语法树节点创建	空	
		newVarNode	变量声明标志类型语法树节点创建	空	
		newDecNode	创建声明类型语法树节点	空	

		newProcNode	创建过程头类型语法树节点	空	
		newStmlNode	创建语句标志类型语法树节点	空	
		newStmtNode	创建语句类型语法树节点	kind	给出具体是哪种语句节点
		newExpNode	创建表达式类型结点	kind	给出具体是哪种表达式节点
		printTree	打印语法树	tree	要打印的语法树的根节点指针
语义分析阶段	symbTable	NewTable	创建当前空符号表	空	
		CreatTable	创建新的局部化单位的空符号表	空	
		DestroyTable	撤销当前符号表	空	
		Enter	登记标识符和属性	id	要登记的标识符名字
				attirbP	标识符的属性
				entry	返回标识符在表中的地址
		FindEntry	寻找表项地址	id	要查找的标识符名字
				entry	返回标识符在表中的地址
		FindAttr	属性查询	entry	表中某项的位置指针
		Compat	判断类型是否相容	tp1	指向类型内部表示的指针
				tp2	指向类型内部表示的指针
		NewTy	创建当前空类型内部表示	kind	给出具体类型
		NewBody	创建当前空记录类型中	空	
		NewParam	创建当前空形参链表	空	
		ErrorPrompt	错误提示	line	出错行号
				name	出错的单词名字
				message	错误提示信息
		printTab	打印空格	tabnum	打印的空格数目
		printSymbTable	输出符号表	空	
		PrintOneLayer	打印符号表的一层	level	给出层数
	Analyze	initialize	初始化基本类型	空	
		TypeProcess	处理类型分析	t	指向当前处理的声明节点的指针
				deckind	给出声明节点的具体类别
		nameType	处理自定义类型名	t	指向当前处理的声明节点的指针
		arrayType	处理数组类型的内部表示	t	指向当前处理的声明节点的指针
		recordType	处理记录类型的内部表示	t	指向当前处理的声明节点的指针
		TypeDecPart	处理类型声明部分	t	指向当前类型声明节点的指针
		VarDecPart	处理变量声明部分	t	指向当前变量声明节点的指针
		VarDecList	处理变量声明部分	t	指向当前变量声明节点的指针
		ProcDecPart	处理过程声明部分	t	指向当前过程声明标志节点的指针

		HeadProcess	处理过程头的语义分析	t	指向当前过程声明标志节点的指针
		ParaDecList	处理参数声明的语义分析	t	指向当前过程声明标志节点的指针
		Body	处理执行体部分的语义分析	t	指向过程体（语句序列）的指针
		statement	处理语句状态	t	指向当前语句节点的指针
		Expr	处理表达式	t	指向当前变量表达式节点的指针
				Ekind	给出是值参还是变参
		arrayVar	处理下标变量	t	指向当前数组变量表达式节点的指针
		recordVar	处理域变量	t	指向当前域变量表达式节点的指针
		Assignstatement	处理赋值语句	t	指向当前赋值语句节点的指针
		callstatement	处理调用语句	t	指向当前过程调用语句节点的指针
		ifstatment	处理条件语句	t	指向当前条件语句节点的指针
		Whilestatement	处理循环语句	t	指向当前循环语句节点的指针
		readstatement	处理读语句	t	指向当前读语句节点的指针
		Writestatement	处理写语句	t	指向当前写语句节点的指针
		Returnstatement	处理返回语句	t	指向当前返回语句节点的指针
		Analyze	语义分析主程序	t	指向语法树根节点的指针
中间代码生成阶段	midcode	GenMidCode	中间代码生成主函数	t	指向程序的语法树的根节点
				StoreNoff	主程序 AR 的 display 表的偏移
		GenProcDec	过程声明的中间代码生成	t	指向过程声明标志节点的指针
		GenBody	语句序列中间代码生成	t	指向语句序列节点的指针
		GenStatement	语句的中间代码生成	t	指向当前语句节点的指针
		GenAssignS	赋值语句的中间代码生成	t	指向赋值语句节点的指针
		GenVar	处理变量	t	指向赋值语句节点的指针
		GenArray	处理数组成员变量	V1arg	数组名变量的 Arg 结构指针
				t	当前变量语法树节点指针
				low	数组的下界
				size	数组的大小
		GenField	处理域变量	V1arg	域名变量的 Arg 结构指针
				t	当前变量语法树节点的指针
				head	指向当前纪录的域表的头指针

		GenExpr	处理表达式的中间代码生成	t	指向表达式节点的指针
		GenCalls	过程调用的中间代码生成	t	指向语句节点的指针
		GenReadS	输入语句的中间代码生成	t	指向语句节点的指针
		GenWriteS	输出语句的中间代码生成	t	指向语句节点的指针
		GenIfS	条件语句的中间代码生成	t	指向语句节点的指针
		GenWhileS	循环语句的中间代码生成	t	指向语句节点的指针
	Util	NewTemp	创建一个新的临时变量	access	给出是直接变量还是间接变量
		NewLabel	创建一个新的标号	空	
		ARGAddr	地址的 ARG 结构的创建	id	变量的名字
				level	层数
				off	偏移
				access	访问方式：直接/间接
		ARGLabel	标号的 ARG 结构的创建	label	标号值
		ARGValue	常量值的 ARG 结构的创建	value	常量值
		GenCode	四元式中间代码的创建	codekind	中间代码类别
				Arg1	指向操作分量 1 的 Arg 结构
				Arg2	指向操作分量 2 的 Arg 结构
				Arg3	指向操作结果的 Arg 结构
		PrintMidCode	打印四元式中间代码序列	irstCode	指向中间代码序列的头指针
	ConstOpti(常量表达式优化)	DivBaseBlock	划分基本块函数	firstCode	指向中间代码序列的头指针
		ConstOptimize	常表达式优化主函数	firstCode	指向中间代码序列的头指针
		OptiBlock	基本块内常表达式优化	i	给出基本块号
		ArithC	算术和比较操作处理	code	指向当前处理的中间代码
		SubstiArg	对 ARG 结构进行值替换	code	指向当前处理的中间代码
				i	指出对中间代码的哪个成员操作
		FindConstT	常量定值表查找	arg	指向当前变量的 Arg 结构的指针
				Entry	返回值：变量在常量定值表中的位置的指针
		AppendTable	常量定值表添加	arg	指向要添加的变量的 Arg 结构的指针
				result	常量值
		DelConst	删除一个常量定值	arg	指向要删除的变量的 Arg 结构的指针
		PrintConstTable	常量定值表打印	i	基本块号
		PrintBaseBlock	打印基本块标志	blocknum	给出基本块的个数
		DivBaseBlock	划分基本块函数	firstCode	指向中间代码序列的头指针
		NewVN	创建一个新的值编码函数	空	

优化阶段	ECCsaving(公共表达式优化)	ECCsave	公共表达式优化主函数	firstCode	指向中间代码序列的头指针
		SaveInBlock	基本块内公共表达式优化	i	基本块号
		EquaSubsti	中间代码的等价替换函数	code	指向当前处理的中间代码
		Process	操作分量的值编码处理函数	code	指向当前处理的中间代码
				i	指出要对哪个分量进行操作
		FindTempEqua	临时变量等价表的查找函数	arg	指向临时变量的 Arg 结构的指针
		SearchValuNum	值编码表的查找函数	arg	指向要查找的变量的 Arg 结构的指针
		IsEqual	ARG 结构是否相同的判断	arg1	指向 Arg 结构的指针
				arg2	指向 Arg 结构的指针
		AppendValuNum	值编码表添加函数	arg	指向变量的 Arg 结构的指针
				Vcode	变量的编码
		FindECC	可用表达式代码表查找	codekind	代码的类别
				op1Code	操作分量 1 的值编码
				op2Code	操作分量 2 的值编码
		AppendTempEqua	临时变量等价表的添加函数	arg1	指向被替换的临时变量的 Arg 结构
				arg2	指向用来作替换的临时变量的 Arg 结构
		GenMirror	中间代码的映象码构造函数	op1	分量 1 对应的值编码
				op2	分量 2 对应的值编码
				result	结果对应的值编码
		AppendUsExpr	可用表达式代码表添加函数	code	指向可用的中间代码的指针
				mirror	指向此中间代码对应的映象码的指针
		SubstiVcode	值编码的替换函数	arg	指向变量的 Arg 结构的指针
				Vcode	用来替换的编码
		DelUsExpr	可用表达式代码表项的删除函数	arg	指向要删除的变量的 Arg 结构指针
		PrintValuNum	打印值编码表	空	
		PrintUsbleExpr	打印可用表达式编码表	空	
		PrintTempEqua	打印临时变量等价表	空	
		PrintBaseBlock	打印基本块标志	blocknum	给出基本块的个数
		LoopOpti	循环不变式优化主函数	firstCode	指向中间代码序列的头指针
		WhileEntry	循环入口处理函数	code	指向当前循环入口中间代码

	LoopOpti(循环不变式优化)	Call	过程调用的处理函数	code	指向当前过程调用中间代码
		WhileEnd	过程出口的处理函数	code	指向当前循环出口中间代码
		LoopOutside	循环外提的处理函数	entry	指向循环入口中间代码的指针
		SearchTable	循环变量定值表的查找函数	arg	指向要查找的变量的 Arg 结构
				head	本层循环的变量定值在表中的起始位置
		DelItem	变量定值表的删除函数	arg	指向要删除的变量的 Arg 结构
				head	本层循环的变量定值在表中的起始位置
		AddTable	变量定值表的添加函数	arg	指向要添加的变量的 Arg 结构
		PushLoop	循环信息栈压栈函数	t	指向一个循环信息表的指针
		PopLoop	循环信息栈弹栈函数	空	
	Code	emitComment	注释生成函数	字符串 c	注释内容
		EmitRO	寄存器地址模式指令生成函数	op	操作码
				r	目标寄存器
				s	第一源寄存器
				t	第二源寄存器
				c	为将写入代码文件 code 的注释内容
		EmitRM	变址地址模式指令生成函数	op	操作码
				r	目标寄存器
				d	偏移值
				s	基地址寄存器
				c	将写入代码文件 code 的注释内容
		EmitSkip	空过生成函数	howmany	给出空过代码写入位置的数量
		EmitBackup	地址回退函数	loc	给出要退到的代码写入位置
		emitRestore	地址恢复函数	空	
		emitRM_Abs	地址转换函数	op	操作码
				r	目标寄存器
				a	存储器绝对地址
				c	将写入代码文件 code 的注释
	Cgen(语法树直接生	genProc	生成过程节点的代码函数	t	指向过程声明的节点的指针
		GenStmt	生成一个语句节点的代码函数	t	指向语句节点的指针

目标代码生成阶段	成目标代码文件)	GenExp	表达式类型语法树节点代码生成函数	t	指向表达式节点的指针
		CGen	语法树代码生成函数	tree	指向语法树的根节点
		CodeGen	产生目标代码文件主函数	syntaxTree	指向语法树的根节点
				codefile	生成的目标代码文件的名字
	CodeGen(中间代码生成目标代码文件)	CodeGen	目标代码生成主函数	midcode	指向中间代码序列的指针
				destcode	生成的目标代码文件名
		arithGen	生成算术运算的目标代码函数	midcode	指向当前处理的中间代码的指针
		operandGen	生成操作数的目标代码函数	arg	指向当前操作数的 Arg 结构的指针
		aaddGen	生成地址加操作的目标代码函数	midcode	指向当前处理的中间代码的指针
		readGen	生成读操作的目标代码函数	midcode	指向读中间代码的指针
		writeGen	生成写操作的目标代码函数	midcode	指向写中间代码的指针
		returnGen	生成返回语句的目标代码函数	midcode	指向返回中间代码的指针
		assigGen	生成赋值语句的目标代码函数	midcode	指向赋值中间代码的指针
		labelGen	生成带有标号的目标代码函数	midcode	指向当前处理的中间代码的指针
		jumpGen	生成跳转的目标代码函数	midcode	指向当前处理的中间代码的指针
				i	决定从中间代码中哪个分量取标号值
		jump0Gen	生成跳转的目标代码函数	midcode	指向当前处理的中间代码的指针
		valactGen	形参为值参时的形实参结合代码生成函数	midcode	指向当前处理的中间代码的指针
		varactGen	形参为变参时的代码生成函数	midcode	指向当前处理的中间代码的指针
		callGen	过程调用处的处理工作	midcode	指向过程调用中间代码的指针
		pentryGen	过程体入口处的处理	midcode	指向当前处理的过程入口中间代码的指针
		endprocGen	过程出口处的处理	midcode	指向过程出口中间代码的指针
		mentryGen	主程序入口的处理部分	midcode	指向主程序入口中间代码的指针
				savedLoc	要回退到的目标代码位置
		FindAddr	计算变量的绝对地址	arg	指向变量的 Arg 结构的指针
		FindSp	找到该变量所在 AR 的 sp, 存入 ac 中	varlevel	变量所在的层数

释放存储空间阶段	Util	freeTree	释放语法树指针空间函数	t	指向语法树的根节点指针
		freeDec	释放声明类型指针空间函数	p	指向声明的语法树节点的指针
		freeStm	释放语句类型指针空间函数	p	指向语句的语法树节点的指针
		freeExp	释放表达式类型指针空间函数	t	指向表达式的语法树节点的指针
		freeTable	释放符号表空间函数	空	
		freeMidCode	释放中间代码空间	空	

表 3.1 SNL 编译器中的文件列表和内含的函数说明

为了对编译结果进行检测，我们还编写了几个有代表性的 SNL 语言程序实例，具体名字和功能见表 3.2：（限于篇幅，源程序见随书附带光盘：SNL 语言例子）

SNL 程序名	程序功能
exp.txt	测试表达式运算的小程序
const.txt	测试常量表达式优化的程序
eccexam1.txt ; eccexam2.txt	测试公共表达式优化的程序
loop1.txt ; loop2.txt	测试循环不变式优化的程序
sort.txt	测试目标代码的排序小程序
Cs1.txt	测试目标代码的 n ，r 组合小程序

表 3.2 SNL 程序实例列表

3．4 SNL 编译器的主程序说明

编译器主程序使用条件编译处理生成可执行文件。程序中使用条件编译，主要是为了适合不同的用户需求。用户可以通过设置条件编译的各项标志，设置生成的编译可执行程序的功能，这些可选的条件为：不进行语法分析及以后功能；不进行语义分析及以后功能；不进行中间代码生成及以后的功能；不进行中间代码优化或者只做其中几种优化；不生成目标代码；完全编译。主程序要求用户输入源程序文件目录名作为主程序调用参数，并在相同目录下自动创建一个与源文件名相同，但扩展名为. tm 的目标代码文件，将生成的目标代码写入其中。在编译过程中，主程序还根据追踪标志将编译的相关信息输出到屏幕上。

下面分章节介绍 SNL 编译程序的各个部分。

主程序文件设置：

为了实现主程序可执行文件的条件编译,SNL 编译程序在主程序文件 main.cpp 中定义了如下条件编译标志,以满足不同的编译功能需求,用户可根据自身要求修改。

NO_PARSE 语法分析条件编译标志,初始为 FALSE。如为 TRUE,则不联入头文件 parse.h,analyze.h,midcode.h,constOpti.h,ECCsaving.h, loopOpti.h,codeGen.h,生成一个只有词法扫描功能的编译器可执行文件。

NO_LL1 设置 LL1 语法分析条件编译标志，初始为 FALSE。在 NO_PARSE 为 FALSE 的前提下，如果为 TRUE，则进行递归下降的语法分析；否则进行 LL1 的语法分析。

NO_ANALYZ 语义分析条件编译标志,初始为 FALSE。如果为 TRUE,则不联入头文件 analyze.h,midcode.h,constOpti.h,ECCsaving.h, loop-Opti.h,codeGen.h,生成一个只有词法和语法分析功能的编译器可执行文件。

NO_CODE 设置代码生成标志，初始为 FALSE。如果为 TRUE，则不生成代码；否则，生成代码。

NO_MIDCODE 中间代码生成条件编译标志,初始为 TRUE。如果为 TRUE，则联入头文件 cgen.h,code.h，从语法树直接生成目标代码；否则，联入 midcode.h，从语法树生成中间代码。

NO_CONSOPTI 常量表达式优化条件编译标志,初始为 FALSE,如果为 TRUE,则不联入头文件 constOpti.h,生成一个不包括常量表达式优化的编译器可执行文件。

NO_ECCOPTI 公共表达式优化条件编译标志,初始为 FALSE,如果为 TRUE,则不联入头文件 ECCsaving.h,生成一个不包括公共表达式优化的编译器可执行文件。

NO_LOOPOPTI 循环不变式优化条件编译标志,初始为 FALSE,如果为 TRUE,则不联入头文件 loopOpti.h,生成一个不包括循环不变式优化的编译器可执行文件。

NO_DESTCODE 目标代码生成条件编译标志,初始为 FALSE,如果为 TRUE,则不联入头文件 codeGen.h,生成一个不包括目标代码生成的编译器可执行文件。

NO_EXECUTE 解释程序条件编译标志，初始为 FALSE，如果为 TRUE,则不联入头文件 tm.h，不对生成的目标程序解释执行。

为了确定编译各个功能处理部分相关信息是否输出,主程序文件还设置了各追踪标志初始值，用户可根据自身要求修改设置。这些跟踪标志为：

EchoSource：源程序代码跟踪标志，初始为 FALSE。为 TRUE，将源代码信息输出到指定的输出设备中；为 FALSE，不输出。

TraceScan： 词法分析追踪标志，初始为 FALSE。为 TRUE，将词法分析信息输出到指定的输出设备中；为 FALSE，不输出。

TraceParse： 语法分析追踪标志，初始为 TRUE。为 TRUE，将语法分析信息输出到指定的输出设备中；为 FALSE，不输出。

TraceTable： 符号表追踪标志，初始为 TRUE。为 TRUE，将语义分析时产生的符号表信息输出到指定的输出设备中；为 FALSE，不输出。

TraceCode： 目标代码追踪标志，初始为 TRUE。为 TRUE，将目标代码的注释信息输出到指定的输出设备中；为 FALSE，不输出。

Error： 错误追踪标志，防止错误的进一步传递，初始为 FALSE。为 TRUE，表明之前的分析中程序已经出现错误。

主程序文件中定义的主函数 main( )是典型的带参数的 C 语言函数。函数详细说明和算法框图如下：

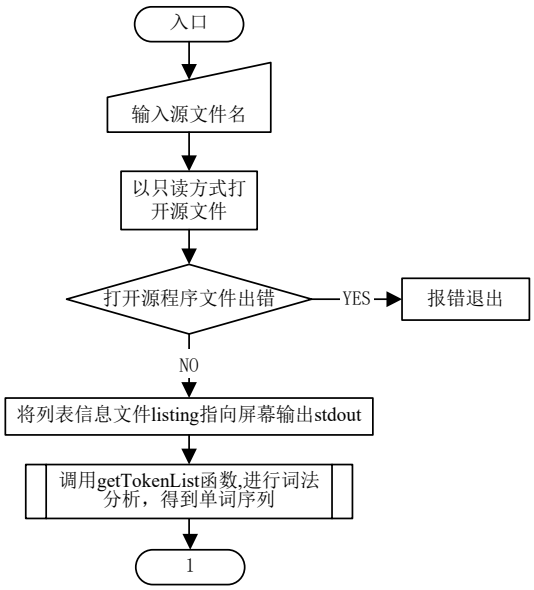
函数声明： main( )

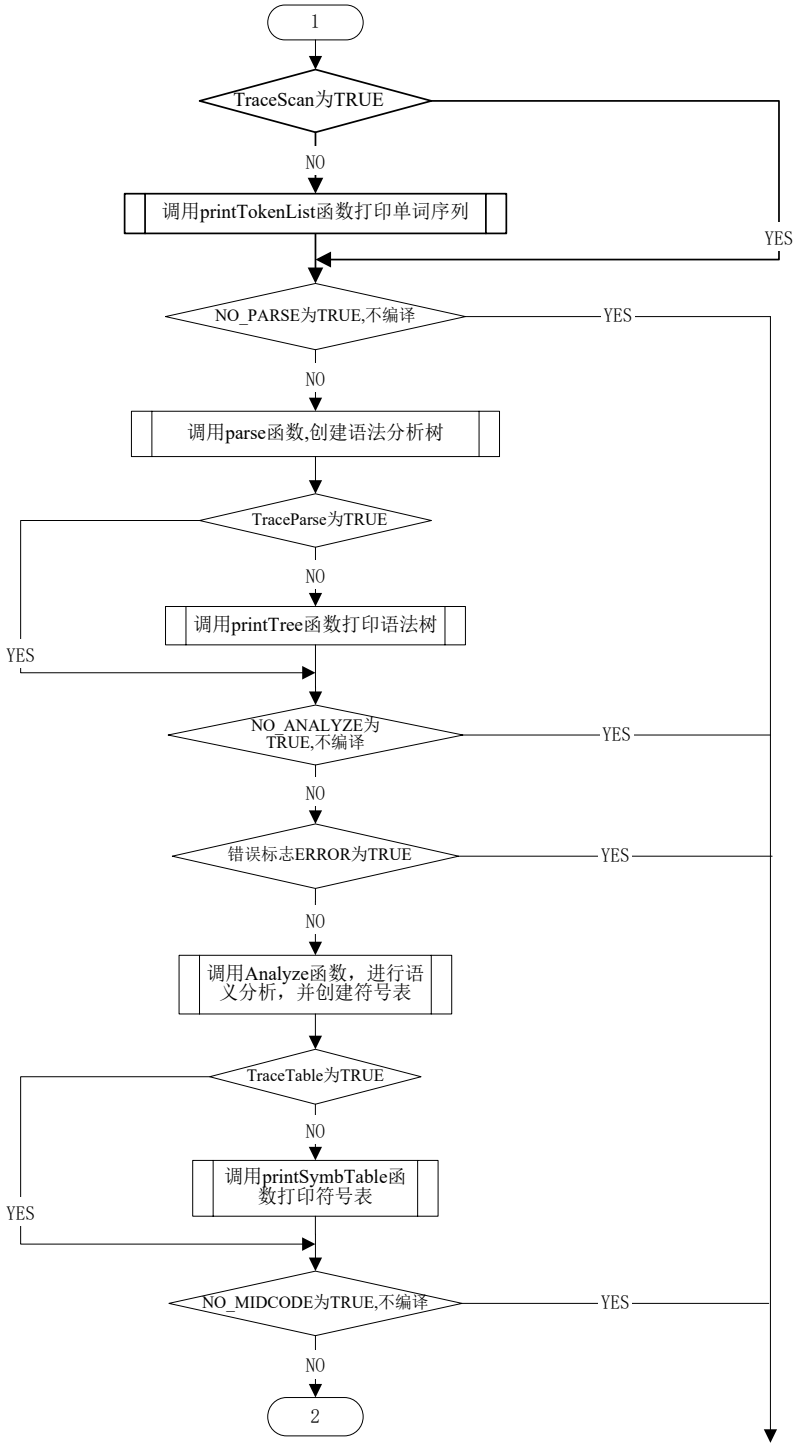
算法说明： 用户应输入要编译的源程序文件目录名。函数对源程序文件进行编译并将目标代码输入到与源文件主文件名同名，但扩展文件名为 tm 的文件中。函数采用条件编译生成可执行的编译器执行文件，根据条件编译标志选择编译，满足用户的不同需要。

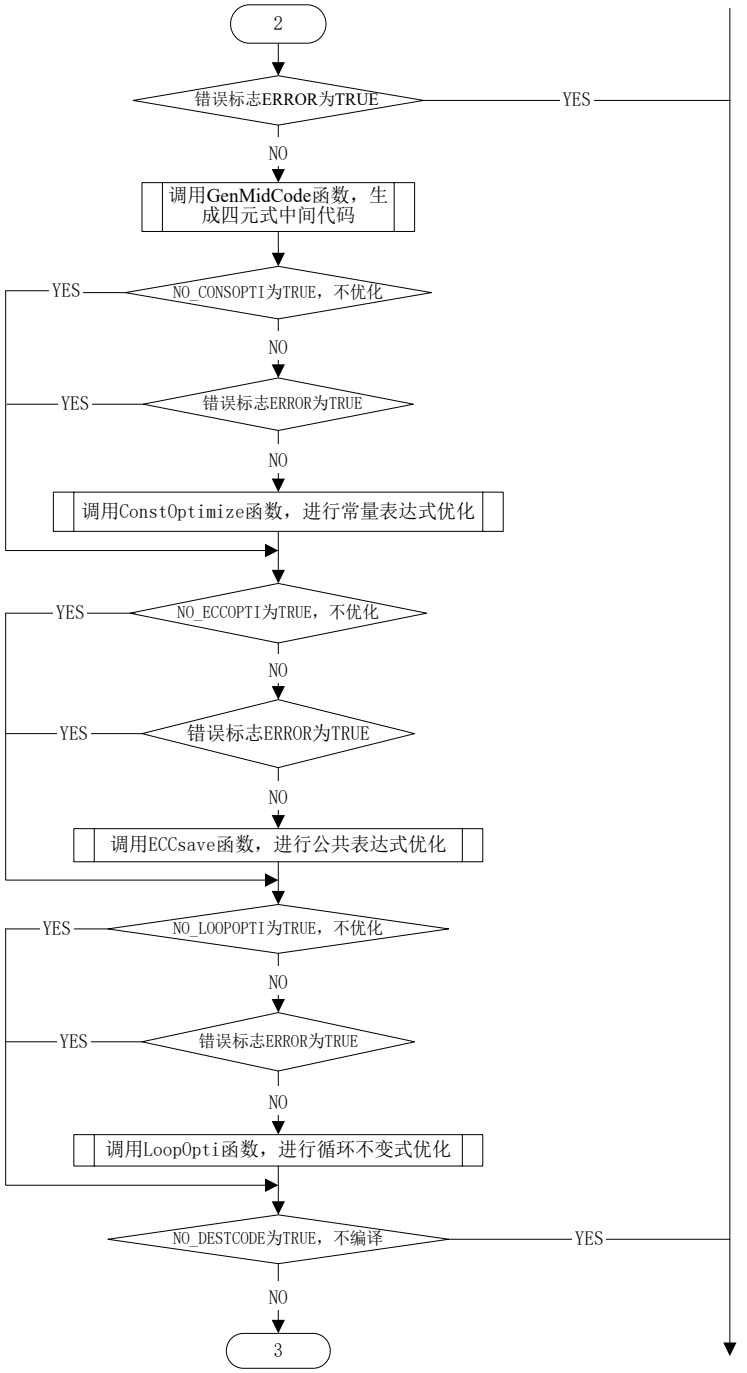
全局变量：

lineno	源程序文件当前代码行号，整数类型，初始化为 0。
source	源程序文件指针，文件指针类型。
listing	中间列表文件指针，文件指针类型。
Fp	Token 序列文件指针，文件指针类型。
Tokennum	Token 序列中单词（Token）的总数，整数类型，初始化为 0。
baseBlock	指向中间代码的基本块的指针，结构指针数组类型。
StoreNoff	存储主程序的 display 表的偏移。
Midcode	中间代码文件指针，结构指针类型。
code	目标代码文件指针，文件指针类型。

算法框图: 见图 3.4。




1
TraceScan为TRUE
NO
YES
调用printTokenList函数打印单词序列
NO_PARSE为TRUE,不编译
YES
NO
调用parse函数,创建语法分析树
TraceParse为TRUE
NO
调用printTree函数打印语法树
YES
NO_ANALYZE为TRUE,不编译
YES
NO
错误标志ERROR为TRUE
YES
NO
调用Analyze函数，进行语义分析，并创建符号表
TraceTable为TRUE
NO
调用printSymbTable函数打印符号表
YES
NO_MIDCODE为TRUE,不编译
YES
NO
2


2
错误标志ERROR为TRUE
YES
NO
调用GenMidCode函数，生成四元式中间代码
YES
NO_CONSOPTI为TRUE，不优化
NO
YES
错误标志ERROR为TRUE
NO
调用ConstOptimize函数，进行常量表达式优化
YES
NO_ECCOPTI为TRUE，不优化
NO
YES
错误标志ERROR为TRUE
NO
调用ECCsave函数，进行公共表达式优化
YES
NO_LOOPOPTI为TRUE，不优化
NO
YES
错误标志ERROR为TRUE
NO
调用LoopOpti函数，进行循环不变式优化
NO_DESTCODE为TRUE，不编译
YES
NO
3

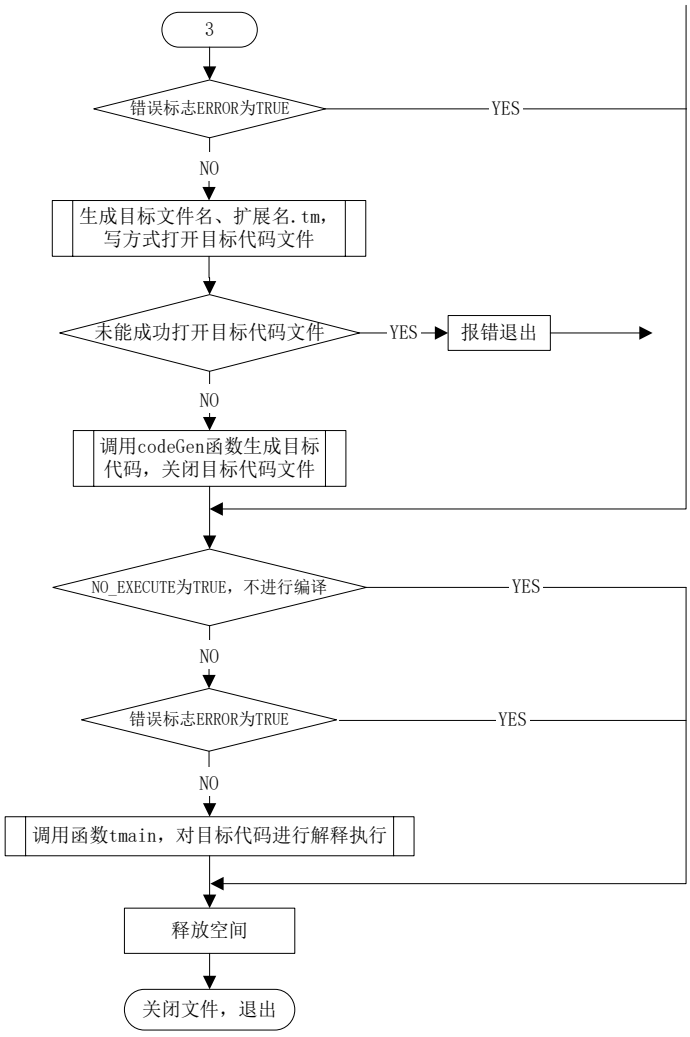

3
错误标志ERROR为TRUE
YES
NO
生成目标文件名、扩展名.tm，
写方式打开目标代码文件
未能成功打开目标代码文件
YES
报错退出
NO
调用codeGen函数生成目标
代码，关闭目标代码文件
NO_EXECUTE为TRUE，不进行编译
YES
NO
错误标志ERROR为TRUE
YES
NO
调用函数tmain，对目标代码进行解释执行
释放空间
关闭文件，退出

图3.4 主程序文件框图

第四章 SNL 语言的词法分析

4．1 词法分析简介

源程序一般表现为字符串（机器语言称其为 ASCII 码）序列的形式，而编译程序的翻译工作应该在单词一级上进行，这与我们自然语言的翻译理解的过程是类似的。因此要进行编译工作，首先需要把源程序的字符序列翻译成单词序列。

词法分析是编译过程的第一阶段。它的任务就是对输入的字符串形式的源程序按顺序进行扫描，根据源程序的词法规则识别具有独立意义的单词（符号），并输出与其等价的 Token 序列。Token 是单词（符号）的内部表示。完成词法分析任务的程序称为词法分析程序，通常也称为词法分析器或扫描器（Scanner）。词法分析程序的功能如图 4.1 所示：

图 4.1 词法分析程序的功能

4．1．1 单词的分类

单词是语言中具有独立意义的最小语法单位。可以看出单词的构成有两个要素：具有独立意义和是最小语法单位。如在我们的 SNL 语言中，表达式“A+12”虽然具有独立意义，但却不是最小的语法单位，还可以拆分成更小的单词“A”，“＋”和“12”。同样，程序中定义的变量 A12，本身已经具有了独立意义——表示变量的标识符，就不能再进行拆分了。究竟哪些符号串是语言的合法单词呢？这与具体语言的词法规则有关。一般常用程序设计语言的单词可以分为下面几类：

1．保留字（也称关键字，基本字等）：保留字一般是由语言系统自身定义的，通常是由字母组成的字符串。如 C 语言中的 main，break，switch，char 等。
2．标识符：用来标识程序中各个对象的名称。如变量名，常量名，过程名，数组名等。
3．常数：用来表示各类常数。如整型常数，实型常数，布尔常数，字符常数等。
4．运算符：表示程序中算术运算、逻辑运算、字符运算的确定的字符或字符串。如各类语言通用的＋，－，×，/，<，>等。还有一些是某种语言特有的运算符，如 C 语言中的++，？：，&等。
5．界限符：如逗号，分号，括号，单引号等。

在词法分析时一般是根据单词表示的含义对单词进行分类，因此不同类的单词，其构成规则可能是相同的，如保留字和标识符就都是由字母打头后跟若干字母和数字构成的字符串，对这样的单词，在拼写完成之后，还要进一步的进行类别的区分。因为语言中保留字的个数是有限的，标识符的个数是无限的，所以可以建立一张保留字表，每当遇到的单词可能是标识符也可能是保留字时，首先查保留字表，若找到该单词，则为保留字，否则为标识符。

例如在下面一小段 SNL 程序中，1 属于常数；**program**、**var**、**integer**、**begin**、**end** 属于保留字；example，i 属于标识符；“（”,“）”,“；”,“.”属于界限符。

```
program    example
var  i   integer
begin
        i =1;
        write ( i )
end .
```

4．1．2 单词的 Token 表示

Token 是单词在编译程序处理过程中的一种内部表示，也是词法分析程序对程序中各类单词进行处理之后的输出形式。对于一种语言而言，如何对它的单词进行分类，每一类单词的 Token 数据结构的形式如何，都没有固定的模式，可以随编译器的不同而不同。通常 Token 的结构可以分成两部分，单词的词法信息和语义信息。其中词法信息记录的是这个单词的种类，语义信息则记录着这个单词的具体信息，如果是标识符，则记录该标识符的名字 id；如果是常数，则记录该常数的数值 val；如果是保留字，且在每个保留字具有独立的分类编码时，则可省略不填，等等。这样，就能为以后的语法分析和语义分析处理单词做好准备。

SNL 的 Token 结构如下图：

	域名(成分)	类 型	含 义
Token	Lineshow	int	记录该单词在源程序中的行数
	Lex	LexType	记录该单词的词法信息，其中 LexType 为单词的类型枚举
	Sem	Char*	记录该单词的语义信息

图 4.2 SNL 的 Token 结构内部表示

4．1．3 词法分析程序和语法分析程序的接口

词法分析器可有两种，一种是它作为语法分析的一个子程序，另一种是它作为编译器的独立一遍任务。在前一种形式时，词法分析器将不断地被语法分析器所调

用，每调用一次词法分析器将从源程序的字符序列拼出一个单词，并将其 TOKEN 值返回给语法分析器。后者则不被别的部分调用，而是完成编译器的独立一遍任务，将整个源程序的字符序列转换成 TOKEN 序列，并将该 TOKEN 序列交给语法分析器。

在这里，SNL 语言编译器的词法分析器将采用第二种形式，将词法分析和语法分析分别进行。主程序将首先调用词法分析程序（函数 getTokenlist），通过扫描源程序字符序列，按语言的词法规则将一个一个的字符组合成各类单词符号，并填写 TOKEN 信息，然后将结果通过链表存入文件 tokenlist 中，以便语法分析时从中顺序读取具有独立意义的单词符号。采用此种编译器结构作为教学用词法分析器，能够清晰易懂。见图 4.3：

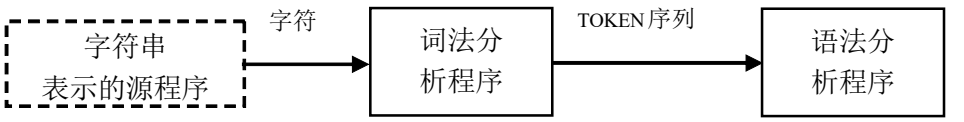


图 4.3 词法分析程序与语法分析程序的接口

4．2 DFA 的构造和实现

4．2．1 状态转换图

有限自动机是描述程序设计语言单词构成的工具，而状态转换图是有限自动机的比较直观的描述方法，而且可以方便地手工构造词法分析程序。遵循程序设计语言的词法规则构造的状态转换图，可识别出源程序的单词。这里我们使用确定的有限状态自动机，简记为 DFA。

例 1：表示整数的 DFA M1 的状态转换图如图 4.4 所示：其中 D 表示数字,other 表示 D 以外的字符。

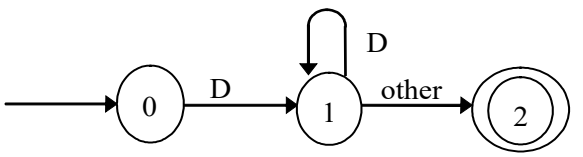


图 4.4 DFA M1 的状态转换图表示

例 2：表示标识符的 DFA M2 的状态转换图如图 4.5 所示：其中 L 表示字母，D 表示数字，other 表示 L，D 以外的字符。

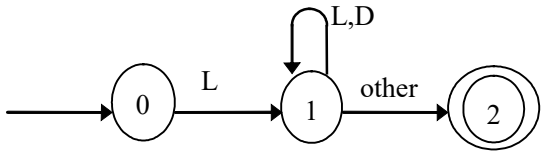


图 4.5　DFA　M2 的状态转换图表示

构造识别单词的状态转换图的方法与步骤：

1. 根据构成规则对程序语言的单词按类构造出相应的状态转换图。
2. 对各类单词的状态转换图合并，构成一个能识别语言所有单词的状态转换图。合并方法为：
   (1) 将各类单词的状态转换图的初始状态合并为一个唯一的初态；
   (2) 化简调整状态冲突和对冲突状态重新编号；
   (3) 如有必要，增加出错状态。

图 4.4 和图 4.5 给出的两类单词的状态转换图，经合并和调整后，得到识别标识符和正整数的 DFA 的状态转换图 4.6，为了统一起见，这里 other 一致表示 L 和 D 以外的一切字符。

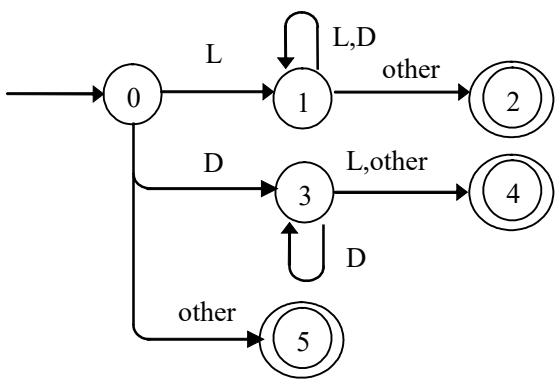


图 4.6　DFA M1 和 DFA M2 合并后的状态转换图表示

下面给出能够识别 SNL 中各类单词的 DFA，如图 4.7 所示。

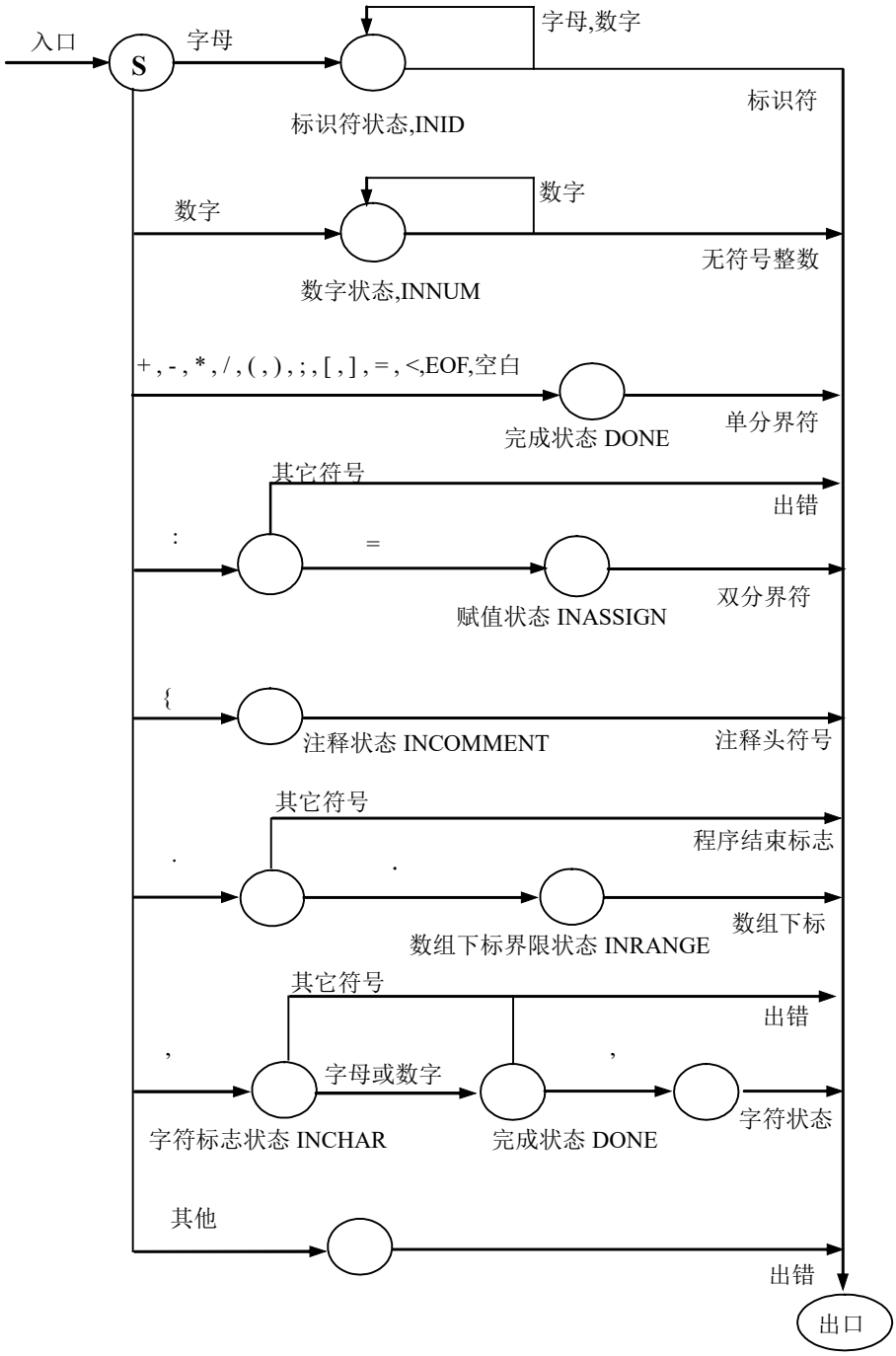


图 4.7　识别 SNL 单词的 DFA 表示

4．2．2 状态转换图的实现

根据语言的词法规则构造出识别其单词的确定有限自动机 DFA, 仅仅是词法分析程序的一个形式模型，距离词法分析程序的真正实现还有一定的距离。状态转换图的实现通常有两种方法，一种是用状态转换表 T；另一种是直接转向法。

- ✧ 状态转换表法又称数据中心法，是把状态转换图看作一种数据结构（状态转换表），由控制程序控制字符在其上运行，从而完成词法分析。用转换表的优点是程序短，但占存储空间多，直接转向法的优缺点正好与此相反。
- ✧ 直接转向法又称程序中心法，是把状态转换图看成一个流程图，从状态转换图的初态开始，对它的每一个状态结点都编一段相应的程序。

为了更好的说明这两种方法，下面分别给出用状态转换表法和直接转向法实现的状态转换图图 4.6。

1．状态转换表法：

(1) 表 4.1 的状态转换表构造如下：

输入字符 / 状态编号	L	D	other
0^0	1	3	5
1	1	1	2
2^*			
3	4	3	4
4^*			
5^*			

表 4.1 状态转换图的实现——状态转换表的构造

(2) 假设输入流以‘.’结束。给出用 C 语言实现的状态转换图如下：

```
enum state(s0,s1,s2,s3,s4,s5);                  /*DFA 的所有状态*/
enum letter(‘a’..’z’);                          /*字母的定义*/
enum digit(‘0’..’9’);                           /*数字的定义*/
enum outkind(id,num,error,start);               /*输出单词的类别*/
int T[6][3] = { ‘1’,‘3’,‘5’,‘1’,‘1’,‘2’,‘start’,‘start’,‘start’,‘4’,‘3’,‘4’,
                ‘start’,‘start’,‘start’,‘start’,‘start’,‘start’}
                                                /*状态转换表*/
char ch;                                        /*用于读入字符*/
state = s0;                                     /*状态初始为 s0*/
ch = getChar();                                 /*读取当前字符*/
while （（T[state][ch] != ‘start’）&&（ch != ‘.’））
{                                               /*根据状态转换表识别单词*/
    state = T[state][ch];
```

```
    ch = getChar();
}
if (state == s2) outkind = id;              /*当前单词是标识符*/
else if (state == s4) outkind = num;        /*当前单词是常数*/
else outkind = error;                       /*出错*/
```

2．直接转向法：

```
enum state(s0,s1,s2,s3,s4,s5);              /*DFA 的所有状态*/
enum letter('a'.. 'z');                     /*字母的定义*/
enum digit('0'.. '9');                      /*数字的定义*/
enum outkind(id,num,error);                 /*输入单词的类别*/
char ch;                                    /*用于读入字符*/
state = s0;                                 /*状态初始为 s0*/
ch = getChar();                             /*读取当前字符*/
switch (state)                              /*根据当前状态进行状态转换*/
{
    case s0:  switch (ch)
                  {
                      case 'a'.. 'z'   : state = s1;break;
                      case '0'.. '9'   : state = s3;break;
                      default    : state = s5;break;
                  }break;                           /*状态 s0 的分情况转向程序*/
    case s1:  while ((ch == letter)&&(ch == digit))
                      ch = getChar();
              state =s2;
              break;                                /*状态 s1 的分情况转向程序*/
    case s2:  ungetChar();
              outkind = id;
              break;                                /*当前单词是标识符*/
    case s3:  while (ch == digit)
                      ch = getChar();
              state =s4;
              break;                                /*状态 s3 的分情况转向程序*/
    case s4:  ungetChar();
              outkind = num;
              break;                                /*当前单词是常数*/
    case s5:  outkind = error;
              break;                                /*出错*/
}
return outkind;                                 /*返回识别出的单词的种类*/
```

从上面对同一个状态转换图的不同实现方法的比较，可以看出直接转向法尽管程序较长，但是比较清晰直观，因此我们对识别 SNL 的各类单词的状态转换图也采用直接转向法来实现。

4．3　词法分析程序的实现

4．3．1　词法分析程序的输入输出

1．输入

词法分析程序一般是从源程序中依次读入字符进行分析和处理。如果可以将源程序一次输入到内存的一个源程序区，则可以大大节省源程序的输入时间，从而提高编译程序的效率。但对于有限的内存空间而言，要满足不同规模源程序的一次输入是非常困难的，同时还会造成较大的系统开销。为了提高源程序的读取速度和方便词法分析处理，一般采用缓冲输入方案，即在内存中开辟一段大小适当的缓冲区，将源程序从磁盘上分批读入缓冲区，词法分析程序从这个缓冲区中依次读取字符进行处理。

SNL 词法分析器的输入是 SNL 源程序。SNL 编译系统在内存开辟了一个 256 个字节 (由常量 BUFLEN 定义)的输入缓冲区 lineBuf。输入时,先将源程序记录到磁盘上,然后分批读入缓冲区。词法分析程序就从这个缓冲区中读取字符。当缓冲区中的字符全部读完后,再从外存上读入一批,如此重复下去,直到源程序字符全部读入为止。此功能由 scan.c 文件中的函数 getNextChar()完成。

2．输出

词法分析程序读入字符串形式的源程序，识别出具有独立意义的最小语法单位——单词，并将单词序列变换成表明单词性质的 Token 序列。

SNL 词法分析对于每类单词的分析结果的 Token 结构为二元组（词法信息，语义信息），如表 4.2 所示：

单词类型	Token	
	LEX	SEM
标识符	ID（a String）	单词名
保留字	如 IF、BEGIN 等	单词名
无符号整数	INTC（23）	字符串
单字符分界符	如 EQ（＝）、SEMI（；）等	空
双字符分界符	ASSIGN（:＝）	空
注释头符	注释部分不用转换成 TOKEN 形式（{）	空
字符标识符	该标识符后的字符 CHARC（’b’）	字符（b）
数组下标界限符	UNDERANGE（..）	空
错误	ERROR	空

表 4.2 SNL 词法分析得到的 TOKEN 结构

SNL 词法分析程序的输出是用单词的 Token 结构链表结构将单词符号以二元组形式输出。

表 4.3 是 SNL 语言程序词法分析结果：（其中，每个 TOKEN 的第一项是该单词

在源程序中所在的行数，第二项是单词的词法信息，第三项是单词的语义信息。）

源程序	词法分析后的 TOKEN 序列		
	行数	词法信息	语义信息

program p
type t = integer ;
var t v1;
char v2;
begin
read(v1);
v1:=v1+10;
write(v1)
end.
	1	PROGRAM	保留字，无语义信息

	1	ID	p
	2	TYPE	保留字，无语义信息
	2	ID	t
	2	ASSIGN	分界符，无语义信息
	2	INTEGER	保留字，无语义信息
	2	SEMI	分界符，无语义信息
	3	VAR	保留字，无语义信息
	3	ID	t
	3	ID	v1
	3	SEMI	分界符，无语义信息
	4	CHAR	保留字，无语义信息
	4	ID	v2
	4	SEMI	分界符，无语义信息
	5	BEGIN	保留字，无语义信息
	6	READ	保留字，无语义信息
	6	LMIDPAREN	分界符，无语义信息
	6	ID	v1
	6	RMIDPAREN	分界符，无语义信息
	6	SEMI	分界符，无语义信息
	7	ID	v1
	7	ASSIGN	分界符，无语义信息
	7	ID	v1
	7	PLUS	分界符，无语义信息
	7	INTC	10
	7	SEMI	分界符，无语义信息
	8	WRITE	保留字，无语义信息
	8	LMIDPAREN	分界符，无语义信息
	8	ID	v1
	8	RMIDPAREN	分界符，无语义信息
	9	END	保留字，无语义信息
	9	DOT	程序结束符，无语义信息
	11	EOF	文件结束符，无语义信息

表 4.3 SNL 语言程序的词法分析结果例

4．3．2 实现词法分析器的注意事项

词法分析是编译程序的第一阶段，其实现原理比较简单，但在实际处理过程可

能会遇到各种各样的问题，对于 SNL 语言来讲，主要有以下几个方面：

1．保留字和标识符名字的区分

由于保留字的词法规则符合标识符的词法规则，故在处理时应加以区分。词法分析器在扫描过程中只要遇到符合标识符词法规则的单词，如果没有出错，则统一标记 ID。当扫描完一个单词时，调用函数 reservedLookup（char * s），查保留字表，以确定是否是保留字，如果是则将 LEX 部分标记为该单词在保留字表中对应的项。注意，所有保留字均需要大写。

2．复合单词的处理

在程序设计语言中，有一类单词是由两个或两个以上符号组成，这类单词的前部分也可以是一个独立的单词，如:=等，在处理到“:”时，我们还不能断定这个单词是“:”还是“:=”，这取决于“:”的下一个字符，如果下一个字符是“=”，则为“:=”，否则该单词为“:”。在处理这类单词时要加以注意。

3．向前搜索及回退

在词法分析过程中，为了准确的判定一个单词的最右端符号，经常需要向前多读一个有时甚至是两个字符。超前读的字符如不匹配的话，还需要回退。为此在 SNL 词法分析程序中提供了函数 ungetNextChar 用以超前读一字符后不匹配时，后退一个字符。例如数字 12，在读完字符 2 之后，仍需继续读下一个字符，如果不是数字，则回退，并将当前单词 LEX 说明为 INTC。

4．数字的转换

词法分析程序应该把字符串转换成数，如‘123’应转换成 123。在 SNL 词法分析中，数的转换是在语法分析中表达式的处理时进行的。

5．输入时边界的处理

SNL 词法分析器的输入采用的是缓冲输入方法。但是不管输入缓冲区有多大,都不能保证单词符号不会被它的边界所打断。所以必须防止超前读字符进行分析时将当前单词符号中未处理的字符冲掉。SNL 编译系统的词法分析函数 getTokenList()将当前单词中未处理的字符保存在变量 tokenString 中,直到一个完整的单词符号录入完成后，将变量 tokenString 作为单词的语义信息提供给语法分析程序，存入 TOKEN 结构中的 SEM 部分。这样就解决了单词的边界丢失问题。

6．注释的处理

在 SNL 源程序语言中，符号“{”用作注释的开始符号，注释以符号 “}”的第一次出现作为结束。下面是一个注释的例子：{ this is a single {comment }。在第一个“{”和第一个“}”之间的部分均为注释部分，词法分析程序将注释中的内容略过不输出。

4．3．3 词法分析程序的实现框图

SNL 编译程序在文件 scanner.h/scanner.c 中实现了词法分析函数 getTokenlist, 函数从源文件字符串序列中获取下一个单词符号,使用确定性有限自动机 DFA 的直接

转向处理方法。每次超前读一个字符,对保留字采用查表方式识别。产生词法错误的时候,仅仅略过产生错误的字符,不加改正。函数最终返回所识别到的单词, 并将单词的语义信息送入变量 tokenString，留给语法分析函数使用。

根据图 4.7 给出的 DFA，就可以构造相应的词法分析程序。DFA 的状态对照表如下：

START	INASSIGN	INCOMMENT	INNUM	INID	INCHAR	INRANGE	DONE
开始状态	赋值状态	注释状态	数字状态	标识符状态	字符标志状态	数组下标界限状态	完成状态

表 4.4 DFA 的状态对照表

1．在开始状态（程序中使用 START 标记）,首先要读进一个字符(调用函数 getNextChar)。如读入一个空白字符,则跳过它,继续读字符,直到读入一个非空字符为止。接下去根据所读进的非空字符跳转至相应的程序进行处理。

2．在标识符状态（程序中使用 INID 标记）,识别并组合成标识符以后,调用保留字查表函数 reservedlookup,用以确定是保留字还是用户自定义的标识符。按情况输出相应单词:标识符输出单词 ID,保留字输出相应的单词标志码。

3．在整数状态（INNUM）,识别并组合数字。输出单词 NUM。

4．在单分界符状态（DONE）,只需输出其内部的单词标志码。

5．在双分界符状态（INASSIGN）,如果读入的下一个字符不是“=”,则识别为冒号，即输出为 COLON。否则,输出单词 ASSIGN。

6．在注释状态（INCOMMENT）,略过注释内容。直到遇到注释结束标志“}”或者是文件结束标志 EOF，并不生成单词。

7．在数组下标界限状态（INRANGE）,如果读入的下一个字符不是“.”，则为程序结束标志，输出单词 DOT；否则，为数组下标界限标志，输出单词 UNDERANGE。

8．在字符标志状态（INCHAR）,如果读入的下一个字符是字母或者数字，则继续读入下一个字符，如果下一个字符是“ ’ ”,则为字符标志，输出单词 CHARC。其他情况均为出错。

9．在错误状态,表示词法分析程序从源程序读入了一个不合法的字符,打印错误信息,输出单词 ERROR,略过产生错误的字符。转开始状态（START）继续识别和组合下一个单词符号。

在词法分析过程中,为了判断是否已经读到了单词的右端符号,有时需要向前多读一个字符。因此词法分析程序提供了函数 ungetNextChar 用以超前读一字符后不匹

配时，后退一个字符。

下面列出了 SNL 词法分析程序中主要的函数的算法框图。

1. 词法分析程序 getTokenlist( )函数主框图：（图 4.8）

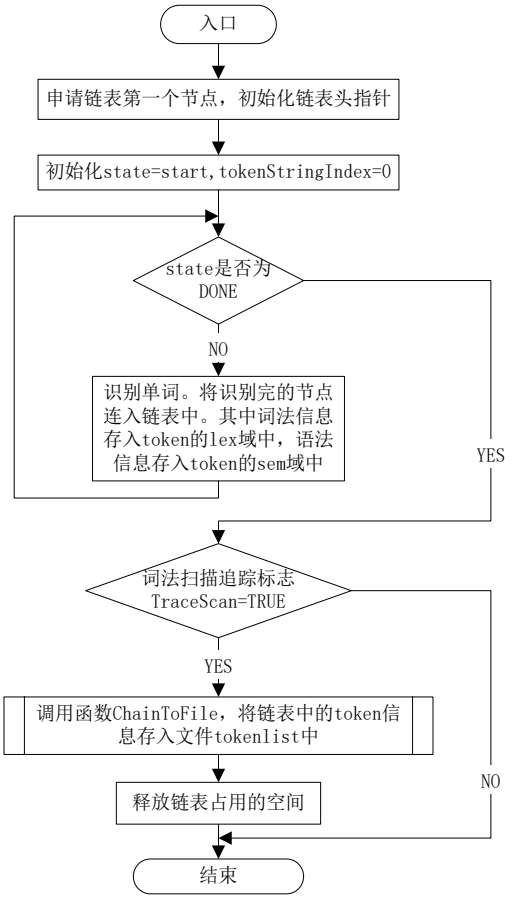


图 4.8 词法分析程序getTokenlist()函数主框图

2. 词法分析程序中循环处理部分程序框图。（见图 4.9）

3. 分类处理部分算法框图。（略）

4. 各函数之间调用关系图。（见图 4.10）

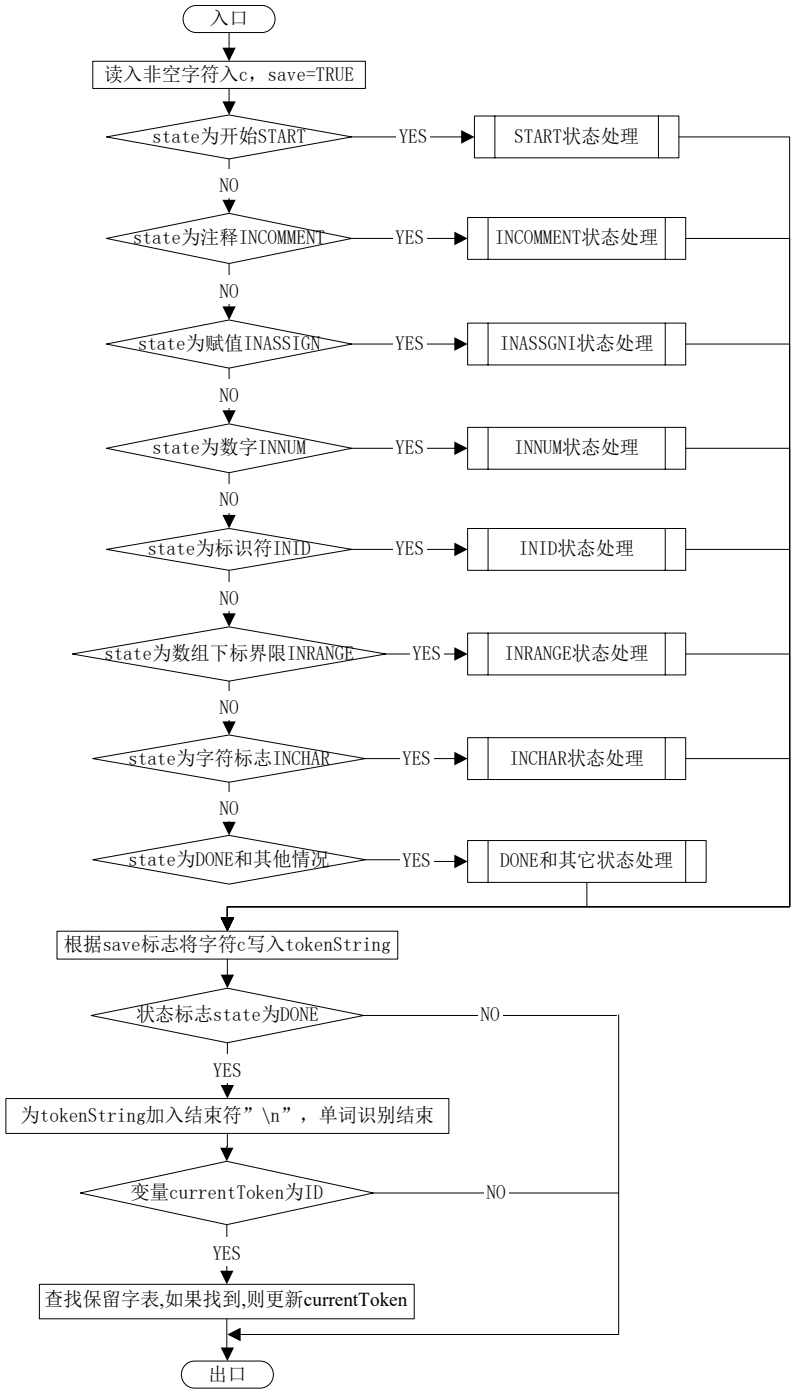


图 4.9 循环处理部分程序框图

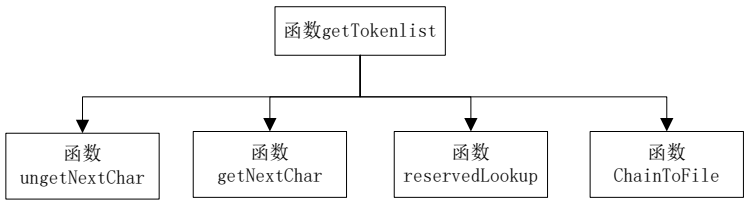


图4.10 各个函数之间的调用关系

4．4 词法分析程序的自动生成器

4．4．1 LEX/FLEX 简介

LEX 是词法分析程序的生成器（Lexical Analyzer Generator），是在 1972 年由贝尔实验室在 UNIX 上首先设计实现的，它是 UNIX 标准应用程序。1984 年的 GNU 工程推出 FLEX（Fast Lexical Analyzer Generator），它是对 LEX 的扩充，同时也与 LEX 兼容。目前，LEX/FLEX 已经可以在 UNIX、LINUX、MS-DOS 等环境运行，不仅高效率地为多种程序设计语言实现了词法分析器，在一些系统软件的开发过程中也得到了广泛的应用。由于 LEX/FLEX 是兼容的，后面将二者统称为 LEX。

我们知道以状态转换图为工具能够实现对单词的识别，而状态转换图是有限自动机的等价表示形式，并且为正则语言，其证明了正则式所表示的语言与有限自动机所识别的语言是完全等价的。一般程序语言单词的词法规则可以表示成正则文法，可以用正则式对其进行描述。由此，为词法分析器的自动生成奠定了理论基础。

词法分析程序生成器接收用正则式表示的定义在某语言字母表∑上的单词，然后构造出与此正则式等价的非确定有限自动机 NFA M′，再对 M′进行确定化和化简，得到确定有限自动机 DFA M，则 M 就是所求的扫描器。LEX/FLEX 即是根据这一原理实现的。

LEX 系统包括 LEX 语言和 LEX 编译器两部分。LEX 语言用来描述单词的正则式，LEX 源程序一般用“l”作为文件名的后缀，例如 A.l 表示语言 A 的 LEX 源程序文件。A.l 文件经 LEX 编译器编译后，输出词法分析程序 Lex.A.out，该程序可以对 A 源程序实现词法分析。

4．4．2 LEX 运行与应用过程

下面以 FLEX 为例，说明 LEX 的运行与应用过程。

首先，按照 Flex 语言的格式要求，编写 Flex 源程序。编写完的 Flex 源程序需由 Flex 系统的翻译程序进行处理。作为处理的结果，将输出一个名为 lexyy.c 的 C 语言程序文件。此文件含有两部分内容，一是根据正则式构造的状态转移表，二是

用来进行词法分析的驱动程序 yylex()，同时，Flex 源程序中所列的语义动作也被直接拷贝到文件 lexyy.c 之中。最后，再用 C 编译程序对 lexyy.c 进行编译，并将支持扫描器运行的有关库函数（它们都包含在文件 flexlib.lib 之中）连入目标码程序，便得到了所要生成的词法分析程序。而后，在编译程序运行时，每从主程序调用 yylex()一次，就从被编译的输入字符流中识别一个单词。显然，对不同的 Flex 源程序而言，Flex 系统所生成的扫描器的驱动程序 yylex()都是相同的，仅状态转移表相异。

Flex 工作过程的图示如下：

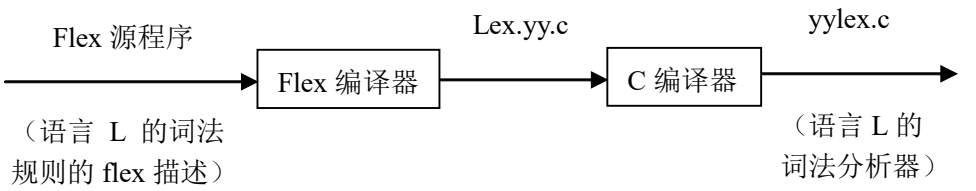


图 4.11 Flex 的工作过程

4．4．3 LEX 源程序结构

LEX 语言作为词法分析器自动构造的专用语言，其程序结构由%%分隔的三个部分组成，其书写格式为：

定义部分
%%
识别规则部分
%%
辅助函数部分

其中，定义部分和辅助函数部分是任选的，识别规则部分是必备的。如果辅助函数部分缺省，则第二个分隔号%%可以省去；但由于第一个分隔号%%用来指示识别规则部分的开始，故即使没有定义部分，也不能将其省去。

1．Flex 的源程序的定义部分

定义部分的作用是对规则部分要引用的文件和变量进行说明，通常可包含头文件表、常数定义、全局变量定义以及宏定义等。除宏定义外，它们与 C 语言程序的书写格式十分类似。

每一个宏定义由分隔符（适当个数的空格或制表符）连接的宏名字和宏内容组成。如语言中的字母可以定义为：

Letter　[a-zA-Z]

数字可以定义为：

Number [0-9]

需要注意的是，凡对已定义的宏名字的引用，都需用花括号将它们括起来，以免发生混乱。

除宏定义外，定义部分的其余代码需用符号%{ 和 %}括起来。另外，Flex 源程序所使用的 C 语言库文件和外部变量，也应分别用#include 和 extern 予以说明，并置于上述括号之内。

在 Flex 源程序中，起标识作用的符号%%，%{以及%}都必须处于所在行的最左字符位置。另外，在其中也可随意添加 C 语言形式的注释。

2．Flex 源程序的识别规则部分

识别规则部分由一组识别规则组成，其书写格式为：

模式 R1 动作 A1

模式 R2 动作 A2

……

模式 Rn 动作 An

其中模式 Ri 是正则式，用来描述单词的词型；动作 Ai 是 C 代码，与匹配的模式对应，用来指明从输入字符串中识别出词型为 Ri 的单词时，扫描器应执行的操作。每个模式都必须从所在行的最左字符位置开始书写，并用分隔符（适当个数的 space 或 tab 字符）与其后的动作分开。每个动作 Ai 可引用已定义的符号常量、全局变量和外部变量，并能调用辅助函数部分所定义的函数，必要时也可在动作中定义自己的局部变量。

通常使用的 LEX 模式定义如表 4.5 所示。

模式	说明	示例
x	匹配单个字符 x	a
[abc]	匹配 a 或 b 或 c	[a-zA-Z]
[^abc]	匹配除去 a,b,c 之外的任意字符	[^ab0-9]表示匹配除小写字母 ab、数字 0-9 以外的任意字符
\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。	
·	匹配除去换行符之外的任意字符	
r*	r 是正则式，r*匹配 0 个或多个 r	

r^+	r 同上，r^+匹配 1 个或多个 r	
r?	r 同上，r?匹配 0 个或 1 个 r	
r{2,5}	r 同上，匹配 2 到 5 之间次数的 r	
r{2,}	r 同上，匹配 2 次或更多次 r	
r{2}	r 同上，匹配 2 次 r	
{name}	name 是在定义部分出现的宏名	
“hello”	匹配字符串“hello”	
r|s	匹配正则式 r 或 s	
rs	匹配正则式 r 和 s 的连接	
……		

表 4.5 常用 LEX 模式定义

3．辅助函数部分

这部分包含了识别规则动作代码段中所调用的各个局部函数，这些函数由用户编写，它们将由 Flex 系统直接拷贝到输出文件 lexyy.c 之中。

4．4．4 应用 LEX 构造词法分析程序

Flex 可以用两种方式来使用，一种是将 Flex 作为一个单独的工具，用以生成所需的识别程序，而这些识别程序通常都出现在一些非开发编译器的应用领域中，诸如编辑器设计、命令行解释、模式识别、信息检索以及开关系统等。第二种方式是将 Flex 和语法分析器自动生成工具（诸如 YACC 和 OCCS 等）结合起来使用，以生成一个编译程序的扫描器和语法分析器。

作为 FLEX 的应用实例，下面给出用 FLEX 生成 SNL 的词法分析程序的全过程：

1．编写 FLEX 的输入文件（命名为 lexsnl.l）：

```
%{
// 库文件：
#include "string.h"
#include "stdio.h"

// Token 种类：
typedef enum
{
    /* 簿记单词符号 */
    ENDFILE,    ERROR,
    /* 保留字 */
    PROGRAM,  PROCEDURE,    TYPE,       VAR,        IF,
    THEN,     ELSE,         FI,         WHILE,      DO,
    ENDWH,    BEGIN1,       END1,       READ,       WRITE,
```

```
    ARRAY,      OF,                 RECORD,     RETURN1,
    //类型
    INTEGER,    CHAR1,
    /* 多字符单词符号 */
    ID,         INTC,           CHARC,
    /*特殊符号 */
    ASSIGN, EQ,     LT,         PLUS,           MINUS,
    TIMES,  OVER,   LPAREN,     RPAREN,         DOT,
    COLON, SEMI,    COMMA,      LMIDPAREN,      RMIDPAREN,
    UNDERANGE
} LexType;

/* 定义保留字数量常量 MAXRESERVED 为 21 */
#define MAXRESERVED 21
//保留字查找表：
static struct
    { char*     str;
      LexType tok;
    } reservedWords[MAXRESERVED]
    ={  {"program",PROGRAM},{"type",TYPE},{"var",VAR},
        {"procedure",PROCEDURE},{"begin",BEGIN1},{"end",END1},
        {"array",ARRAY},{"of",OF},{"record",RECORD},{"if",IF},{"then",THEN},
        {"else",ELSE},{"fi",FI},{"while",WHILE},{"do",DO},{"endwh",ENDWH},
        {"read",READ},{"write",WRITE},{"return",RETURN1},
        {"integer",INTEGER},{"char",CHAR1}   };

//声明保留字查找函数
LexType   reservedLookup (char * s);
//全局变量，存储 token 的内容
char yylval[20];
//全局变量，为了处理注释：flag=1 表示是注释内容，不处理
int flag=0;
%}
%%
[0-9]+          if (flag == 0)
                    {   //整型常量
                        yylval[0] = '\0';
                        strcpy(yylval, yytext);
                        return INTC;
                    }

[a-z][a-z0-9]*  if (flag == 0)
                    {   //标志符
                        yylval[0] = '\0';
                        strcpy(yylval, yytext);
                        /*调用保留字查找函数，若是保留字，则返回保留字对应的
                          Token，否则返回 ID*/
```

```
                        return (reservedLookup(yytext));
                    }
'[a-z,0-9]'     if (flag == 0)
                    {   //字符常量
                        yylval[0] = '\0';
                        strcpy(yylval,yytext);
                        return CHARC;
                    }
"+"                 if (flag == 0) return PLUS;      /*这部分为各运算符的处理*/
"-"                 if (flag == 0) return MINUS;
"*"                 if (flag == 0) return TIMES;
"/"                 if (flag == 0) return OVER;
"("                 if (flag == 0) return LPAREN;
")"                 if (flag == 0) return RPAREN;
"."                 if (flag == 0) return DOT;
"["                 if (flag == 0) return LMIDPAREN;
"]"                 if (flag == 0) return RMIDPAREN;
";"                 if (flag == 0) return SEMI;
":"                 if (flag == 0) return COLON;
","                 if (flag == 0) return COMMA;
"<"                 if (flag == 0) return LT;
"="                 if (flag == 0) return EQ;
":="                if (flag == 0) return ASSIGN;
".."                if (flag == 0) return UNDERANGE;
"EOF"               if (flag == 0) return ENDFILE;
"{"                 flag = 1;
"}"                 flag = 0;
[ \t\n]             ;  /*表示把所有空格，制表符，换行符去掉*/
.                   if (flag == 0) {   //处理没有规则可以匹配的情形：
                        yylval[0] = '\0';
                        strcpy(yylval, yytext);
                        return ERROR;
                    }
%%

LexType   reservedLookup (char * s)
{
    /*保留字查找函数: 查看一个标识符是否是保留字, 标识符如果在保留字表中则返回相应单词,否则返回单词 ID*/
}

void printToken(int token)
{
    //输出函数：将各类单词按种类输出
}
//  主函数：
void main()
```

```
{
  int n=1;
  while(n)
 {
     n=yylex();                  // 调用 yylex 函数，取得一个单词
     printToken(n);              // 将当前单词的信息打印出来
 }
}
```

2．运行 FLEX：

在命令行形式下，执行命令：flex247 lexsnl.l 得到 c 语言程序 yylex.c；这里我们使用 FLEX 的可执行文件：flex247.exe。

3．对 lexyy.c 进行编译和连接

注：需要自己添加 FLEX 的库文件 flexlib.lib 到 VC 中。

例：对于 SNL 源文件

```
program _p
type    t = integer;
var    t    v1 ;
        char    v2;
begin
     read(v1);
     v1:=v1*10;
     v1 = "d";
     v2:= 'a';
     write(v1)
end.
```

运行 yylex.c 生成的词法分析结果见下表：

reserved word: program
ERROR: _
ID, name= p
reserved word: type
ID, name= t
=
reserved word: integer
;
reserved word: var
ID, name= t
ID, name= v1
;
reserved word: char

ID, name= v2
;
reserved word: begin
reserved word: read
(
ID, name= v1
）
;
ID, name= v1
:=
ID, name= v1
*
NUM, val= 10
;
ID, name= v1
=
ERROR: "
ID, name= d
ERROR: "
;
ID, name= v2
:=
CHAR, char=a
;
reserved word: write
(
ID, name= v1
)
reserved word: end
.
EOF

表 4.6 运行 yylex.c 后生成的词法分析结果

第五章　SNL 语言的语法分析

5．1　语法分析概述

语法分析（Syntax Analysis）是编译程序的第二阶段，也是编译程序的核心部分。语法分析的任务是，根据语言的语法规则，对源程序进行语法检查，并识别出相应的语法成分。按照 SNL 编译程序的模型，语法分析的输入是从词法分析器输出的源程序的 Token 序列形式，然后根据语言的文法规则进行分析处理，语法分析的输出是无语法错误的语法成分，表示成语法树的形式。

语言是由具有独立意义的单词根据一定的语法规则组成的句子的集合，句子的结构由语法规则给出，句子的含义由语义规则给出，而对语言的语法分析就是对语言的句子结构的分析。对于程序设计语言而言，它的句子就是程序，程序设计语言定义的是符合其语法规则的程序的集合，因此程序设计语言的语法分析的关键是识别程序（句子）的语法结构。

完成语法分析任务的程序称为语法分析程序，也称为语法分析器或简称分析器。图 5.1 给出了语法分析器的功能示意图。

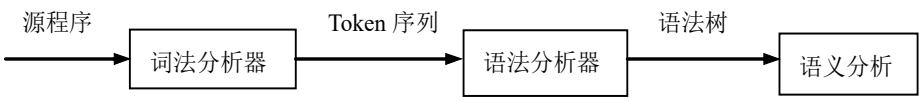


图 5.1 语法分析器的功能

5．1．1　上下文无关文法

程序设计语言的描述主要包括两部分，其一是文法描述部分，其二是语义描述部分。其中文法描述部分定义程序的形式结构，而语义描述部分则描述程序的含义。文法部分的作用有两个：一是定义什么样的字符串是合法的程序，二是指出每个语法成份是由哪些子成份组成。

程序设计语言的绝大多数语法成份文无关文法（2 型文法）来定义的，一个 2 型文法可定义成如下形式：

定义 1 一个上下文无关文法（CFG）：

$G = (V_T, V_N, S, P)$

其中，

V_T　　有限的终极符集

V_N　　有限的非终极符集

S　　开始符，　$S \in V_N$

P　　产生式的有限集，其中产生式具有下面形式：

$A \rightarrow X_1X_2\ldots\ldots X_n$，其中 A 属于 V_N ，$X_i \in （V_N \cup V_T \cup \varepsilon）$。

例 1.标识符的上下文无关文法 $G_i=(V_T, V_N,S,P)$,其中 V_T = {0,1,…,9,A,…,Z,a,…,z},V_N = {<Iden>,<D>,<L>},S = <Iden>,P 是下面的产生式：

<Iden>　→　<L> | <Iden> <L> | <Iden><D>

<L>　→　A | B | … | Z | a | b | … | z

<D>　→　0 | 1 | … | 9

上面文法定义标识符是由字母打头，后跟若干字符和数字的字符串。也就是凡是满足这样条件的字符串都可称之为标识符。所有满足文法约束条件的字符串组成的集合就称为文法定义的语言。语法分析的任务就是检查给定的字符串是否满足文法的定义。

定义 2 推导：我们称αAβ直接推出αγβ，即

$$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

仅当 A→γ是一个产生式，且α、β∈（$V_N \cup V_T$）*。如果$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$,则我们称这个序列是从$\alpha_1$至$\alpha_n$的一个推导。

若存在一个从α_1至α_n的推导，则称α_1可推导出α_n。我们用$\alpha_1 \Rightarrow^+ \alpha_n$表示从$\alpha_1$出发，经一步或若干步，可推导出$\alpha_n$。而用$\alpha_1 \Rightarrow^* \alpha_n$表示从$\alpha_1$出发，经 0 步或若干步，可推导出$\alpha_n$。若 S 是文法的开始符,如果 $S \Rightarrow^* \alpha$，则称α为文法的句型。

例 2．对文法 G_i,可以有：

<Iden>　⇒　<Iden> <D> ⇒ <L> <D>

<Iden>　⇒　<Iden> <L> ⇒ <L> <L>　⇒ <L> a

我们可以将上面的推导分别记做<Iden>　⇒* <L><D>，<Iden>　⇒* <L>a。可以看出推导给出了由文法的开始符出发，如何应用产生式，得到文法定义的句型。

定义 3 句子：如果有 $S \Rightarrow^* y$ ，S 是文法的开始符，且 y 中不包含非终极符（$y \in V_T^*$），称 y 为文法的句子。

例 3．对文法 Gi,因为有：

<Iden>　⇒　<Iden> <D> ⇒ <L> <D> ⇒ a <D> ⇒ a1

<Iden>　⇒　<Iden> <L> ⇒ <L> <L> ⇒ <L> a ⇒ ca

即有<Iden> ⇒* a1，<Iden> ⇒* ca，因为 a1 和 ca 都是终极符串，所以 a1 和 ca 都是 Gi 的句子。

定义 4 语言：文法 G 所产生的句子的全体是一个语言，将它记为 L（G）。

$$L（G）= \{\alpha \mid S \Rightarrow^+ \alpha \ \&\ \alpha \in V_T^*\}$$

在形式语言中，若推导每次替换的都是最右的非终极符，则称为最右推导，最右推导常被称为规范推导。由规范推导所得的句型称为规范句型。

定义 5 规范归约：规范推导的逆过程，称为规范归约，又称最左归约。

如果从给定的字符串能够归约出文法的开始符，则说明该字符串是文法的一个句子。如果一个程序是文法的一个句子，那么这个程序就没有语法错误。因此，程序设计语言作为一般语言的特例，语法分析的关键问题就是句子的识别问题：

设给定文法 G 和字符串（句子）α（$\alpha \in V_T^*$），检查 $\alpha \in L$（G）是否成立。即检查、判定 α 是否是文法 G 所能产生的合法的句子，同时检查和处理语法错误。

5．1．2 语法分析方法的分类

通常语法分析方法可以分为两大类：

1．一类方法是从开始符出发进行推导，如能把程序推导出来，则证明该程序为文法的句子，且没有语法错误。这类方法称为自顶向下的语法分析方法。推导的实现方式不同就产生不同的语法分析方法，如递归下降法，LL(1)方法等；

2．第二类方法则是上述推导过程的逆过程。从源程序出发，进行逐步规约，如能把源程序规约成为文法的开始符，则其逆过程一定成立，从而验证了程序为文法的句子，没有语法错误，否则程序一定有语法错误，这类方法称为自底向上的语法分析方法。规约方式的不同就产生了不同的语法分析方法，如简单优先方法，LR 方法等。

5．1．3 三个重要集合

在语法分析过程中，为了简化分析过程，提高分析效率，一般都对 2 型文法做一些限制，即要求文法满足一定的条件，不同的语法分析方法对语法的限制是不同的，受限后的语法是 2 型文法的子集，但它们一般都能描述程序设计语言的文法，这对于处理程序设计语言来说是足够用的。

由于对文法做了适当的限制，从而提高了语法分析的效率（如避免了回溯问题），这样在语法分析中就可以利用一些特殊的条件进行特殊的处理，通常在语法分析中会用到以下三个集合：First 集，Follow 集，Predict 集。

具体定义如下：（其中 $\beta \in (V_N \cup V_T)^*$，$A \in V_N$，S 表示文法开始符，ε 表示空串，# 表示输入的结束符）：

定义 6 $First(\beta) = \{a \in V_T \mid \beta \Rightarrow^* a \ldots\ldots\} \cup (\text{if } \beta \Rightarrow^* \varepsilon \text{ then}\{\varepsilon\} \text{ else } \varnothing)$

定义 7 $Follow(A) = \{a \in V_T \mid S \Rightarrow^+ \ldots\ldots Aa\ldots\ldots\} \cup (\text{if } S \Rightarrow^* \ldots\ldots A \text{ then}\{\#\} \text{ else } \varnothing)$

定义 8 Predict（$A \rightarrow \beta$）

$= First(\beta)$ ，当 first(β)不含 ε

$= First(\beta) - \{\varepsilon\} \cup Follow(A)$ ，当 First(β)含 ε

在有些书中 Predict 集也称为 Select 集。First(β)表示 β 串所能推导的所有可能的终极符串的头终极符集，如果能推导出空串，则令 First(β)包含 ε。Follow(A) 表示所有那些终极符的集合，这些终极符在某个句型（句型是指由文法开始符推导出来的符号串，可以包含终极符和非终极符）中出现在 A 的紧后面，这种集合主要用于求 Predict（A →β）集合。Predict 集合主要用于自顶向下的语法分析。

5．1．4 SNL 语言的 Predict 集

进行递归下降法和 LL(1)方法的语法分析，都需要用到 Predict 集，按照 SNL 语言的文法（见 2.3.2）我们求得 Predict 集如表 5.1。读者也可以针对 SNL 语言文法，自己求各产生式对应的 Predict 集，作为练习。

产生式编号	Predict 集合
1	{ **PROGRAM** }
2	{ **PROGRAM** }
3	{ **ID** }
4	{ **TYPE, VAR, PROCEDURE, BEGIN** }
5	{ **VAR, PROCEDURE, BEGIN** }
6	{ **TYPE** }
7	{ **TYPE** }
8	{ **ID** }
9	{ **VAR,PROCEDURE,BEGIN** }
10	{ **ID** }
11	{ **ID** }
12	{ **INTEGER, CHAR** }
13	{ **ARRAY, RECORD**}
14	{ **ID**}
15	{ **INTEGER**}
16	{ **CHAR** }
17	{ **ARRAY** }
18	{ **RECORD** }
19	{ **ARRAY** }
20	{ **INTC** }
21	{ **INTC** }
22	{ **RECORD** }
23	{ **INTEGER, CHAR** }
24	{ **ARRAY** }
25	{ **END** }
26	{ **INTEGER，CHAR， ARRAY**}
27	{ **ID** }
28	{ **;** }
29	{ **COMMA** }
30	{ **PROCEDURE, BEGIN**}
31	{ **VAR** }

32	{ **VAR** }
33	{ **INTEGER,CHAR,ARRAY, RECORD, ID**}
34	{ **PROCEDURE ,BEGIN** }
35	{ **INTEGER,CHAR,ARRAY, RECORD,ID** }
36	{ **ID** }
37	{ **;** }
38	{ **COMMA** }
39	{ **BEGIN**}
40	{ **PROCEDURE**}
41	{ **PROCEDURE**}
42	{ **BEGIN** }
43	{ **PROCEDURE** }
44	{ **ID**}
45	{ **)** }
46	{ **INTEGER,CHAR,ARRAY, RECORD,ID ,VAR** }
47	{ **INTEGER,CHAR,ARRAY, RECORD, ID, VAR** }
48	{ **(** }
49	{ **;** }
50	{ **INTEGER,CHAR,ARRAY, RECORD,ID** }
51	{ **VAR** }
52	{ **ID**}
53	{ **; ,)** }
54	{ **COMMA** }
55	{ **TYPE, VAR, PROCEDURE, BEGIN** }
56	{ **BEGIN** }
57	{ **BEGIN** }
58	{ **ID, IF, WHILE, RETURN, READ,WRITE** }
59	{ **ELSE , FI, END,ENDWH** }
60	{ **;**}
61	{ **IF** }
62	{ **WHILE** }
63	{ **READ** }
64	{ **WRITE** }
65	{ **RETURN** }
66	{ **ID** }
67	{ **:=** }
68	{ **(** }
69	{ **[, DOT ,:=** }
70	{ **IF** }
71	{ **WHILE** }
72	{ **READ** }
73	{ **ID** }
74	{ **WRITE** }
75	{ **RETURN** }
76	{ **(** }
77	{ **)** }

78	{ **(, INTC, ID** }
79	{ **) }**
80	{**COMMA** }
81	{ **(, INTC, ID** }
82	{ **< , =** }
83	{ **(, INTC, ID** }
84	{ **< , =,] , THEN, ELSE, FI, DO, ENDWH ,) , END, ; ,COMMA** }
85	{ **+ , -** }
86	{ **(, INTC, ID** }
87	{ **+, - ,< ,= ,] , THEN , ELSE , FI, DO, ENDWH,), END, ; ,COMMA** }
88	{ *** , /** }
89	{ **(** }
90	{ **INTC** }
91	{ **ID** }
92	{ **ID** }
93	{ **:=, * , / , + , - , <, =,THEN, ELSE, FI, DO, ENDWH,),END, ; ,COMMA** }
94	{ **[** }
95	{ **DOT**}
96	{ **ID** }
97	{ **:=, * , / , + , - , <, =,THEN, ELSE, FI, DO, ENDWH,),END, ; ,COMMA** }
98	{ **[** }
99	{ **<** }
100	{ **=** }
101	{ **+** }
102	{ **−** }
103	{ ***** }
104	{ **/** }

表 5.1 SNL 文法的 Predict 集

5．2 语法分析程序的实现

为了让读者接触到更多的语法分析方法，我们对 SNL 语言的语法分析采用了两种不同的自顶向下的语法分析方法：递归下降法和 LL(1)方法。

5．2．1 语法分析程序的输入与输出

1．输入

利用我们在第三章讲到的词法分析时产生的 Token 序列，调用函数 gettoken，每

次从 Token 序列中取出一个 Token，作为程序正在分析的当前单词，其中存有其词法信息 Token 码和语义信息 Token 符号名，作为语法分析的输入。

2．输出

SNL 语法分析程序输出与源程序结构相对应的语法分析树,为了清晰起见，额外加入了类型声明、变量声明、函数声明、以及语句序列等标志节点，SNL 语法树的定义在文件 globals.h 中实现。语法树格式如图 5.2 所示：

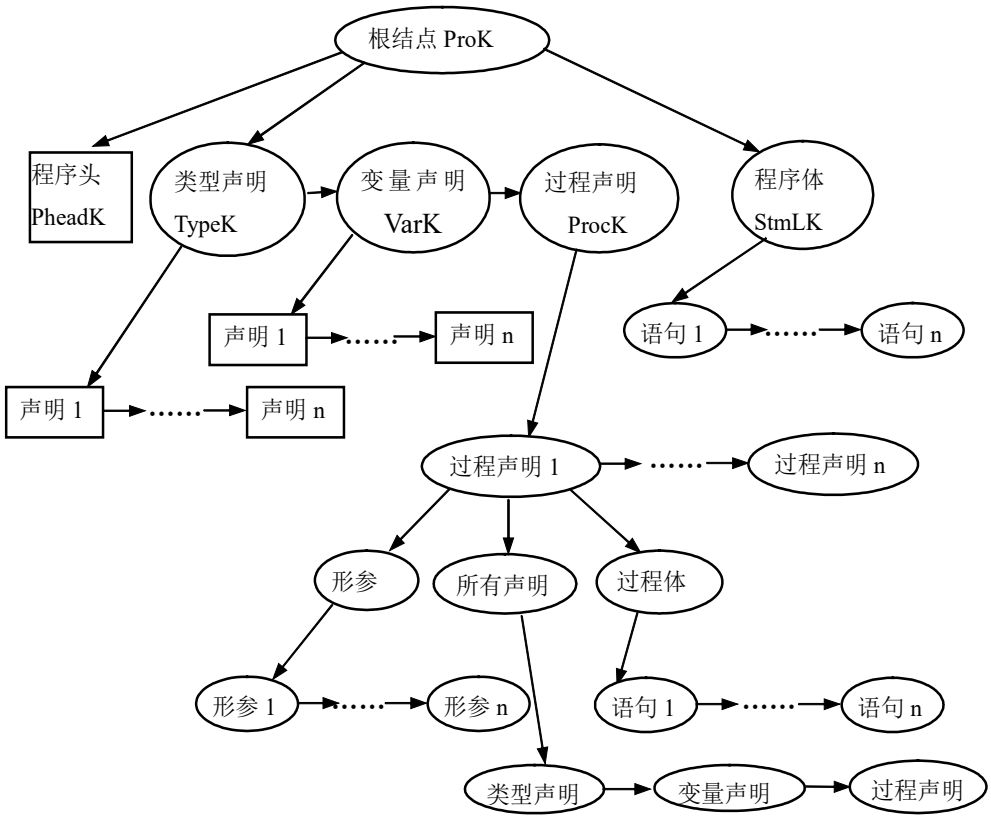


图 5.2 SNL 的语法树格式

5．2．2　语法树节点的数据结构

1．语法树节点类型

SNL 语法树节点的类型 NodeKind 有两类：标志节点和具体节点。

- 标志节点可分为：根标志节点 ProK，程序头标志节点 PheadK,类型声明标志节点 TypeK，变量声明标志节点 VarK，函数声明标志节点 ProcDecK 和语句序列标志节点 StmLK.之所以称为标志节点是因为这种节点只表示节点的类型，

没有具体内容。

- 具体节点可分为：声明节点 DecK,语句节点 StmtK 和表达式节点 ExpK。

由于类型标识符定义，变量的定义，以及函数的形参部分，需要保存的内容很相似，所以把它们用一种节点表示，称为声明节点 DecK。

（1） 声明节点根据类型部分的信息，可以分为：组类型 ArrayK、字符类型 CharK、整数类型 IntegerK、记录类型 RecordK 以及以类型标识符作为类型的 IdK。
（2） 语句节点由 StmtKind 定义，又可细分为：判断语句类型 IfK，循环语句类型 WhileK，赋值语句类型 AssignK，读语句类型 ReadK，写语句类型 WriteK，函数调用语句类型 CallK，返回语句类型 ReturnK 七种。
（3） 表达式类型由 ExpKind 定义可以细分为：操作符类型 OpK，常整数类型 ConstK，标识符类型 IdEK 三种。

2．语法树节点的数据结构

语法分析器的输出将作为语义分析的输入，因此语法分析树中不仅应该包含源程序的结构信息，还应该为语义分析提供必要的信息。SNL 语法树的节点（TreeNode）结构设计如下所示:

child			Sibling	Lineno	nodekind	kind			idnum	name	table	attr	type
0	1	2				dec	stmt	exp				……	

3．语法树节点结构说明

成员 child[i]	指向子语法树节点指针，为语法树节点指针类型。
成员 sibling	指向兄弟语法树节点指针，为语法树节点指针类型。
成员 lineno	记录源程序行号，为整数类型。
成员 nodekind	记录语法树节点类型，取值 ProK, PheadK, TypeK, VarK, ProcDecK, StmLK, DecK, StmtK, ExpK,为语法树节点类型。
成员 kind	记录语法树节点的具体类型，为共用体结构。
kind 成员 dec	记录语法树节点的声明类型，当 nodekind = DecK 时有效，取值 ArrayK,CharK,IntegerK,RecordK,IdK，为语法树节点声明类型。
kind 成员 stmt	记录语法树节点的语句类型，当 nodekind = StmtK 时有效，取值 IfK,WhileK,AssignK,ReadK,WriteK,CallK,ReturnK，为语法树节点语句类型。
kind 成员 exp	记录语法树节点的表达式类型，当 nodekind=ExpK 时有效，取值 OpK,ConstK,IdK，为语法树节点表达式类型。

成员 idnum　　记录一个节点中的标志符的个数。
成员 name　　字符串数组，数组成员是节点中的标志符的名字。
成员 table　　指针数组，数组成员是节点中的各个标志符在符号表中的入口。
成员 type_name　　记录类型名，当节点为声明类型，且类型是由类型标志符表示时有效。

成员 attr　　记录语法树节点其他属性,为结构体类型。

attr 成员 ArrayAttr　　记录数组类型的属性。
ArrayAttr 成员 low　　整数类型变量，记录数组的下界。
ArrayAttr 成员 up　　整数类型变量，记录数组的上界。
ArrayAttr 成员 childType　　记录数组的成员类型。

attr 成员 procAttr　　记录过程的属性。
procAttr 成员 paramt　　记录过程的参数类型，值为枚举类型 valparamtype 或者 varparamtype，表示过程的参数是值参还是变参。

attr 成员 ExpAttr　　记录表达式的属性。
ExpAttr 成员 op　　记录语法树节点的运算符单词，为单词类型。当语法树节点为“关系运算表达式”对应节点时，取值 LT,EQ；当语法树节点为“加法运算简单表达式”对应节点时，取值 PLUS,MINUS；当语法树节点为“乘法运算项”对应节点时，取值 TIMES,OVER；其它情况下无效。
ExpAttr 成员 val　　记录语法树节点的数值,当语法树节点为“数字因子”对应的语法树节点时有效,为整数类型。
ExpAttr 成员 varkind　　记录变量的类别；值为枚举变量 IdV,ArrayMembV 或 FieldMembV；分别表示变量是标志符变量，数组成员变量还是域成员变量；
ExpAttr 成员 type　　记录语法树节点的检查类型，取值 Void, Integer,Boolean,为类型检查 ExpType 类型。

在进行语法分析时，语法分析程序将根据 SNL 的文法产生式，为相应的非终极符创建一个语法树节点，并为之赋值，得到的是与程序结构相似的改进的语法树。

4. 举例

表 5.2 分别列出了源程序、词法分析后得到的对应的 TOKEN 序列以及语法分析后得到的对应的语法树各节点的主要数据项。其中并没有完全表示出语法树节点的所有域，只是列举了相关的类型信息。

源程序	词法分析后 的 TOKEN 序列	语法分析后的语法树

program p
type t1 = integer;
var integer v1,v2;
procedure
q(integer i);
var integer a;
begin
a:=i;
write(a)
end
begin
read(v1);
if v1<10
then v1:=v1+10
else v1:=v1-10
fi;
q(v1)
end.
	1: reserved word:
	program
	1: ID, name= p
	2: reserved word:
	type
	2: ID, name= t1
	2: =
	2: reserved word:
	integer
	2: ;
	3: reserved word:
	var
	3: reserved word:
	integer
	3: ID, name= v1
	3: ,
	3: ID, name= v2
	3: ;
	4: reserved word:
	procedure
	4: ID, name= q
	4: (
	4: reserved word:
	integer
	4: ID, name= i
	4:)
	4: ;
	5: reserved word:
	var
	5: reserved word:
	integer
	5: ID, name= a
	5: ;
	6: reserved word:
	begin
	7: ID, name= a
	7: :=
	7: ID, name= i
	7: ;
	8: reserved word:
	write
	8: (
	8: ID, name= a
	8:)
	9: reserved word:
	end
		ProK
		PheadK p
		TypeK
		DecK IntegerK t1
		VarK
		DecK IntegerK v1 v2
		ProcDecK q
		DecK value param:
		IntegerK i
		VarK
		DecK IntegerK a
		StmLk
		StmtK Assign
		ExpK a IdV
		ExpK i IdV
		StmtK Write
		ExpK a IdV
		StmLk
		StmtK Read v1
		StmtK If
		ExpK Op <
		ExpK v1 IdV
		ExpK Const 10
		StmtK Assign
		ExpK v1 IdV
		ExpK Op +
		ExpK v1 IdV
		ExpK Const 10
		StmtK Assign
		ExpK v1 IdV
		ExpK Op -
		ExpK v1 IdV
		ExpK Const 10
		StmtK Call
		ExpK q IdV
		ExpK v1 IdV

	10: reserved word:
	begin
	11: reserved word:
	read
	11: (
	11: ID, name= v1
	11:)
	11: ;
	12: reserved word:
	if
	12: ID, name= v1
	12: <
	12: INTC, val= 10
	12: reserved word:
	then
	12: ID, name= v1
	12: :=
	12: ID, name= v1
	12: +
	12: INTC, val= 10
	12: reserved word:
	else
	12: ID, name= v1
	12: :=
	12: ID, name= v1
	12: -
	12: INTC, val= 10
	12: reserved word:
	fi
	12: ;
	13: ID, name= q
	13: (
	13: ID, name= v1
	13:)
	14: reserved word:
	end
	15: .
	16: EOF	

表 5.2 SNL 编译器的语法分析结果实例

5．3 递归下降法的实现

5．3．1 递归下降法基本原理

递归下降法是语法分析中最易懂的一种方法。它的主要原理是，对每个非终极符按其产生式结构构造相应语法分析子程序，其中终极符产生匹配命令，而非终极符则产生过程调用命令。因为文法递归相应子程序也递归，所以称这种方法为递归子程序下降法或递归下降法。其中子程序的结构与产生式结构几乎是一致的，例如产生式

ConditionalStm ::= **IF** RelExp **THEN** StmList **ELSE** StmList **FI**

则对应该产生式的递归下降法语法分析程序如下：

```
ConditionalStm ( ) { Match(IF); RelExp(); Match(THEN); StmList();
                     Match(ELSE); StmList(); Match(FI); }
```

其中 Match 函数表示检查当前输入符是否是产生式所要求的单词，是则指向下一个单词，否则出错。RelExp( )，StmList( )分别是对应非终极符 RelExp 和 StmList 的语法分析子程序。

5．3．2 递归下降法应满足的条件

在采用递归下降法实现语法分析之前，需要考虑这样两个问题：

a) 当一个非终极符有多个产生式时，如何来选择产生式？如何保证被选择产生式是唯一的选择？也就是说若产生式的选择不唯一，会导致推导过程的回溯，降低分析效率。
b) 遇到空产生式如何处理？

为解决上述两个问题，需要用到前面提到的三个集合。这三个集合的作用，我们通过一个例子来说明。

假设有产生式组：$A \rightarrow \alpha_1 | \alpha_2 | \cdots\cdots | \alpha_n$，设当前输入符为 a，若 $a \in First(\alpha_i)$，则应该选择规则 $A \rightarrow \alpha_i$，则说明当前状态下，应用该产生式进行推导，可以保证产生的符号串的第一个符号就是当前输入符，因此该产生式可以用来进行推导。

如果有空产生式存在，或者 α_i 可以导出 ε，而且当前输入符 $a \in Follow(A)$，即说明应用此空产生式后推导得到的符号串的第一个符号也是当前符，因为它是 A 的后继符，而 A 推导为 ε。

综合上面两种情况，选择一个产生式的前提条件是当前输入流的头符应该属于相应产生式的 Predict 集。

因此递归下降法要满足的条件：

假设 A 的全部产生式为 $A \rightarrow \alpha_1|\alpha_2|\ldots\ldots|\alpha_n$ ，如果

$$predict(A \rightarrow \alpha_i) \cap predict(A \rightarrow \alpha_j) = \Phi, \text{当 } i \neq j.$$

这样才能保证可以唯一的选择合适的产生式。若将上述产生式的分析过程写成一个通式，

设 $\alpha_i := X_{i1}X_{i2}\ldots\ldots X_{imi}$， 则对应的语法分析子程序应该为

```
Procedure A( );
  case token of
       Predict(A—>α1):Begin match(x11);match(x12);……match(x1m1);
       ……
       Predict(A—>αi):Begin match(xi1);match(xi2);……match(ximi);
       ……
       Predict(A—>αn):Begin match(xn1);match(xn2);……match(xnmn);
  End;
```

其中，match(x)定义为：

$$match(x) = \begin{cases} \text{无定义} & x \in V_n \\ \text{检查 token=x？若是，则移位，否则，报错并停机} & x \in V_t \end{cases}$$

语法分析主程序可写为：

```
Program p
   Token：类型；
   Begin
      读入一个 token;
      if token∈First( S)
      then S;
      else 报错并停机 ;
   End;
```

其中，S 为文法的开始符。

按照上述方法很容易构造一个文法的语法分析程序。

5．3．3 递归下降法的语法分析程序框图

SNL 采用了递归子程序方法和树型推导进行语法分析，对文法中的每个非终极符号按其产生式结构产生相应的语法分析子程序，完成相应的识别任务。其中终结符产生匹配命令，非终结符则产生调用命令。每次进入子程序之前都预先读入一个单词。因为使用了递归下降方法，所以程序结构和层次清晰明了，易于手工实现，且时空效率较高。实际的语法分析工作，从调用总程序的分析子程序开始，根据产生式进行递

归调用其他 55 个分析子程序。可将这些语法分析子程序按照功能分成：

总程序	一个程序的开始，由 parse 函数直接调用；
程序头	包括程序的名字；
程序声明	包括程序的整个声明部分，又可细分为：
类型声明	用于定义类型，其中类型又分为基本类型（整型，实型）、结构类型（数组类型，记录类型）和自定义类型（类型名称为标识符，实际类型为上述两种类型）；
变量声明	用于声明变量，定义变量类型；
函数声明	用于声明函数，其中包括参数（变参，值参）、函数的返回类型、函数中的声明和函数体；
程序体	由语句序列组成。
语句序列	在进入语句序列以后就是对语句部分处理。
语句	语句包括：赋值语句、条件语句、循环语句、输入语句、输出语句、返回语句、函数调用语句。对语句部分的分析处理时，在进入语句分析后，只要根据当前读入的单词符号（应是标识符，或者其它相应保留字）就可以立即判定它属于哪一种语句，选择相应的分析程序对其进行分析处理。处理过程中根据文法产生式创建相应的语法树节点，并对其赋值。在处理语句的时候，将要用到对表达式部分处理。
表达式	表达式的处理相对简单，只需根据文法产生式选择相应的分析程序分析处理，生成语法树节点。

语法分析程序最终输出与源程序结构对应的语法分析树。

从 SNL 的语法定义 (见第一章)可以看出,为了对 SNL 程序进行语法分析,各分析子程序之间必存在相互调用的关系。调用关系可用图 5.3 表明：

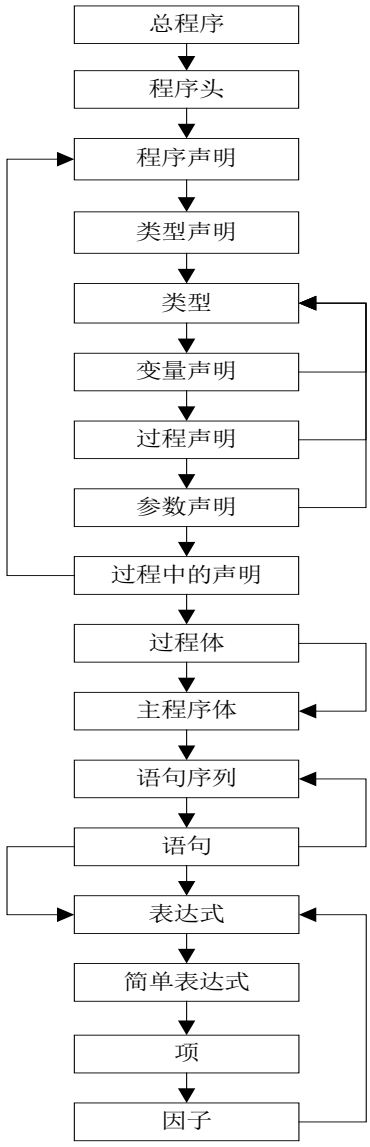


图5.3 SNL递归下降法语法分析中各语法成分之间依赖图

具体各函数说明和算法框图如下：（其中，每个以非终极符名命名的函数说明之前都列出了该非终极符对应的产生式，并且当有多个产生式时，在每条产生式右侧以{}括出本条产生式的 Predict 集合）

1. 函数声明：TreeNode * parse(void)
   算法说明：该函数调用总程序处理函数，创建语法分析树，同时处理文件的提前结束错误。函数处理成功，则返回所生成的语句类型语法树节点。否则，函数返回 NULL。
   算法框图: 见图 5.4。　　各函数调用关系：见图 5.5。

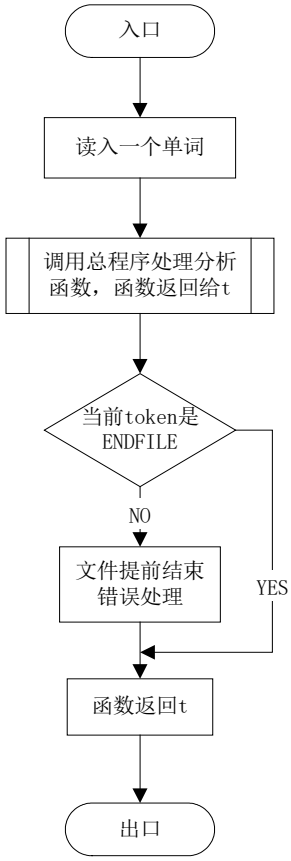


图5.4 语法分析函数parse()的算法框图

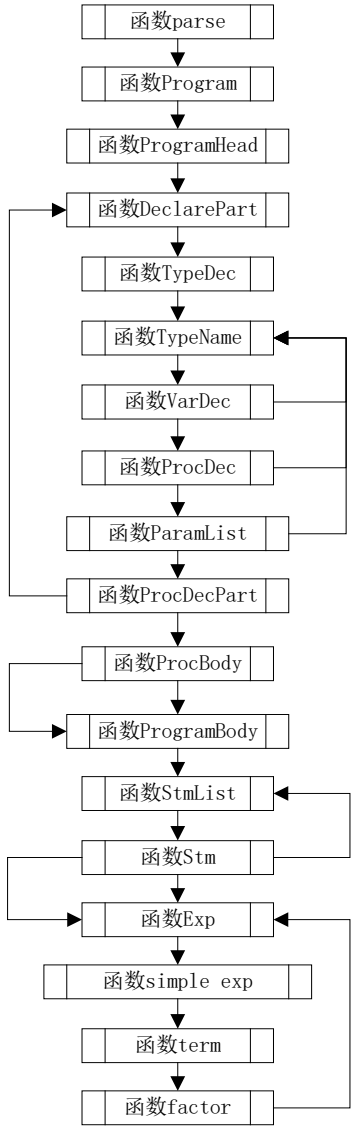


图5.5 语法分析中各函数调用关系

2． 总程序处理分析程序

产生式： < Program > **::=** ProgramHead DeclarePart ProgramBody **.**

函数声明： TreeNode * Program (void)

算法说明： 该函数根据产生式生成语法树的根节点 root，调用程序头部分分析函数、声明部分分析函数、程序体部分分析函数，分别为语法树的三个儿子节点。匹配程序结束标志“.”（DOT） 。如果处理成功,则函数返回程序根节点语法树节点 root。否则,函数返回 NULL。

算法框图： 见图 5.6。

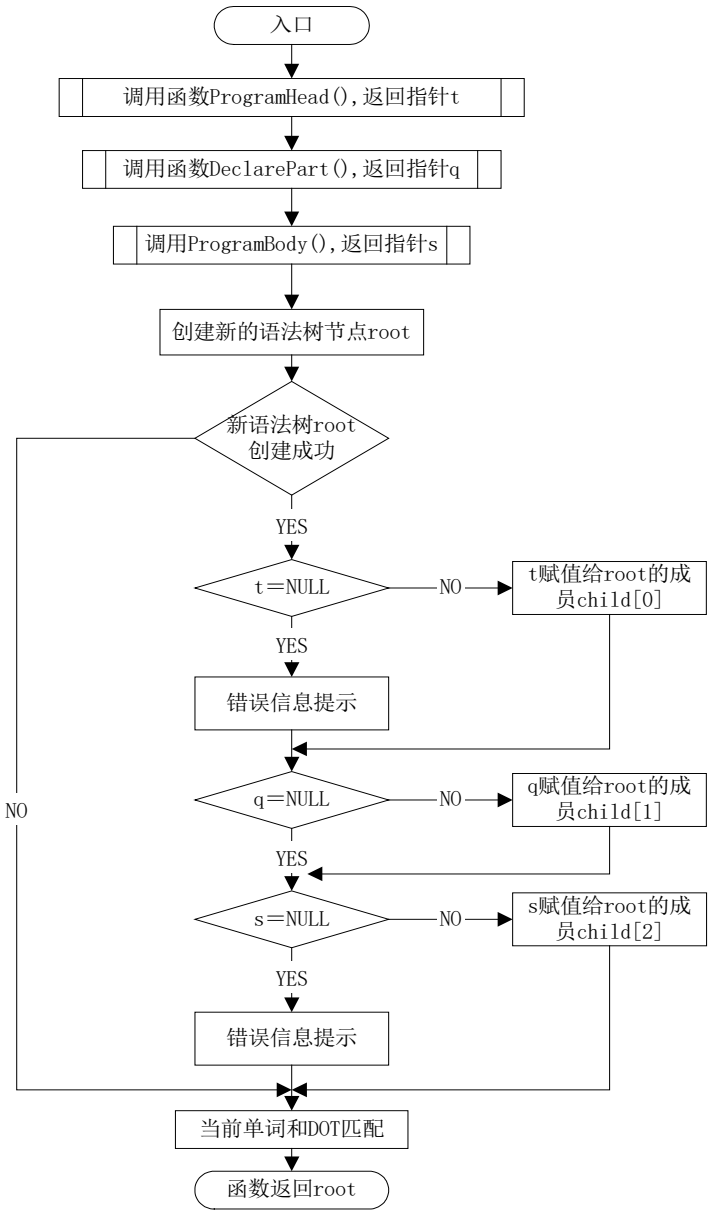


图5.6 总程序处理分析程序program()算法框图

3．程序头部分处理分析程序

产生式： < ProgramHead > ::= **PROGRAM** ProgramName

函数声明： TreeNode * ProgramHead(void)

算法说明： 该函数根据读入的单词，匹配保留字 PROGRAM，然后记录程序名于程序头节点，匹配 ID。如果处理成功，函数返回生成的程序头节点类型语法树节点。否则，函数返回 NULL。

算法框图： 见图 5.7。

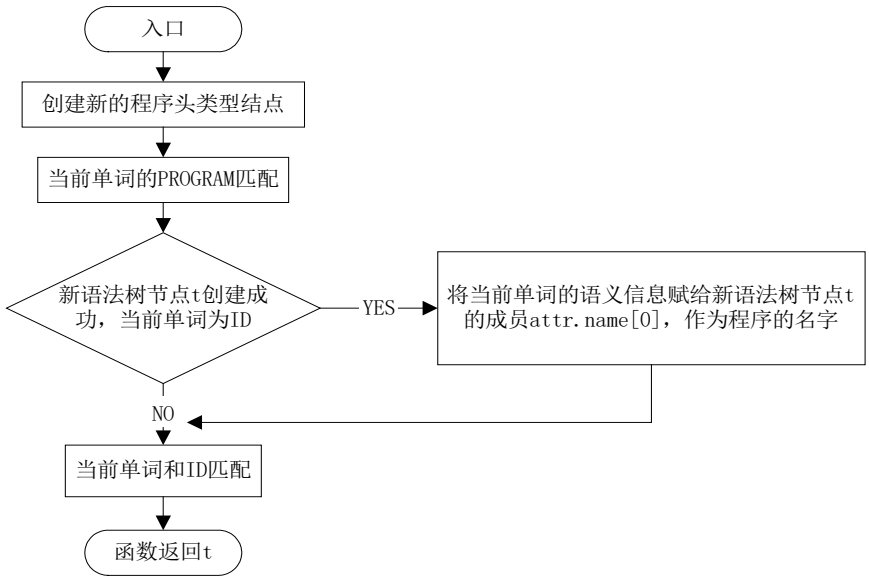


图5.7 程序头部分处理函数ProgramHead()的算法框图

4．程序声明部分处理分析程序

产生式： < DeclarePart > ::= TypeDec VarDec ProccDec

函数声明： TreeNode * DeclarePart(void)

算法说明： 该函数根据文法产生式，创建新的类型声明标志节点、变量声明标志节点。调用类型声明部分处理分析函数 TypeDec()（其返回指针 tp1 为类型声明标志节点的第一个子节点）、变量声明部分处理分析函数 VarDec()（其返回指针 tp2 为变量声明标志节点的第一个子节点）、函数声明部分处理分析函数 ProcDec()。使变量声明标志节点为类型声明标志节点的兄弟节点，函数声明节点为变量声明标志节点的兄弟节点。如果处理成功，则函数返回指向类型声明标志节点（如果没有类型声明，则指向变量声明标志节点；如果没有变量声明，则指向函数声明节点）。否则，函数返回 NULL。

算法框图： 见图 5.8。

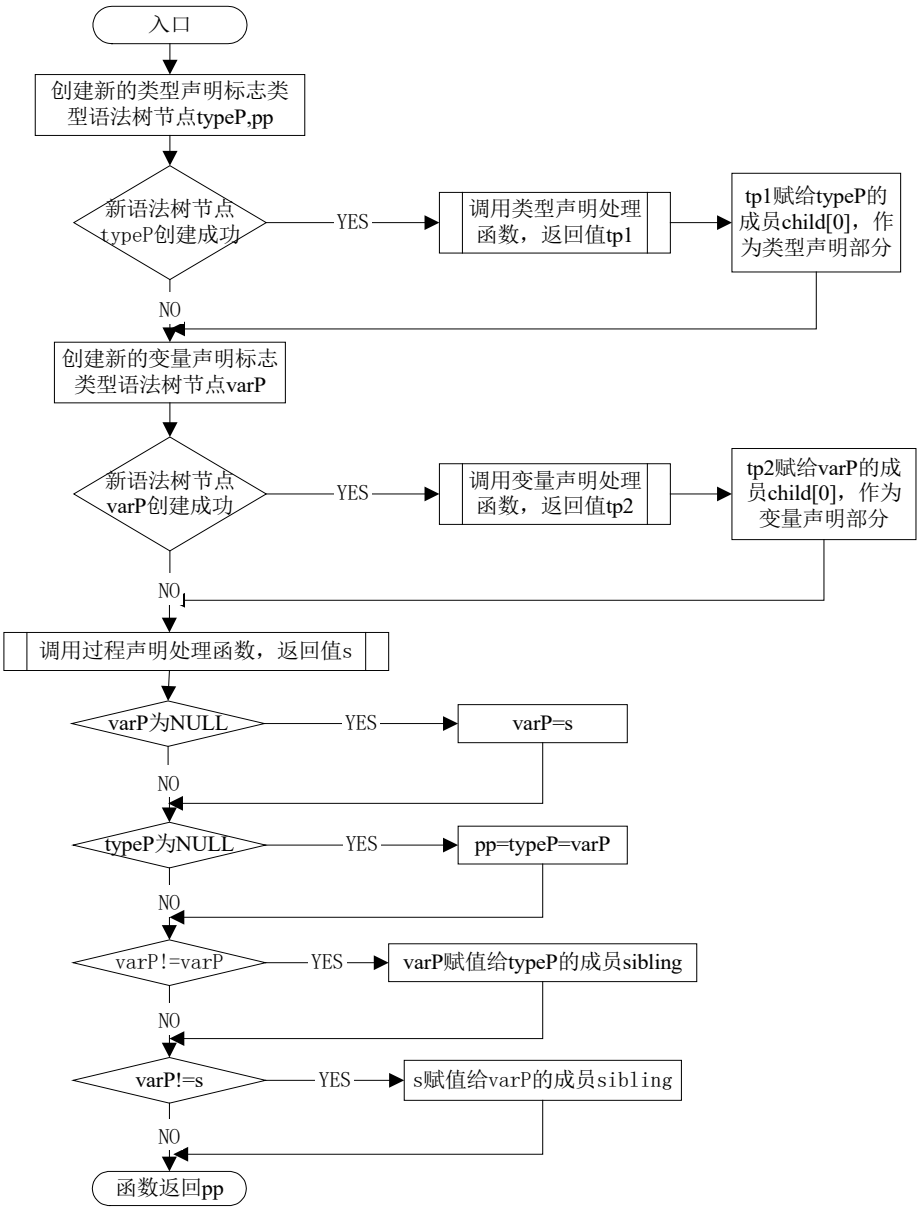


图5.8 程序声明部分处理函数DeclarePart()的算法框图

5．类型声明处理分析程序

产生式： < TypeDec > **::=** **ε** **{VAR,PROCEDURE,BEGIN}**

| typeDeclaration **{TYPE}**

函数声明： TreeNode * TypeDec(void)

算法说明： 该函数根据读入的下个单词，选择调用 t=typeDeclaration()，则函数返回 t；或者不做动作，或者跳过此错误单词，则返回 NULL。

算法框图： 见图 5.9。

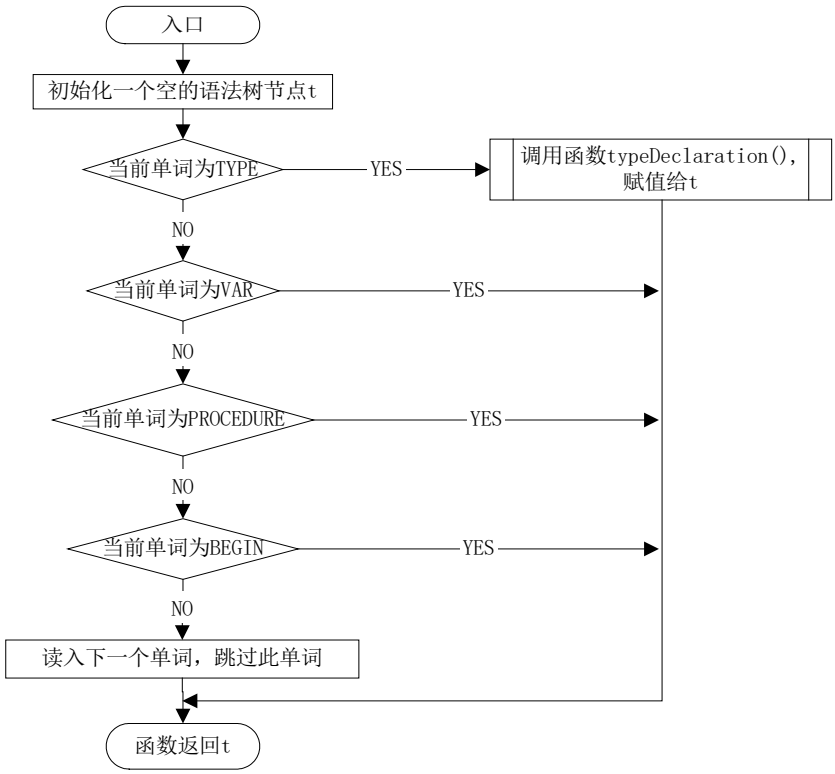


图5.9 可为空的类型声明部分处理函数typeDec()的算法框图

6．类型声明中的其他函数

产生式： < TypeDeclaration > **::=** **TYPE** TypeDecList

函数声明： TreeNode * TypeDeclaration(void)

算法说明： 该函数根据产生式，匹配保留字，调用函数 TypeDecList()，赋值为 t。如果 t 为 NULL，则显示提示信息。函数返回 t。

算法框图： 见图 5.10。

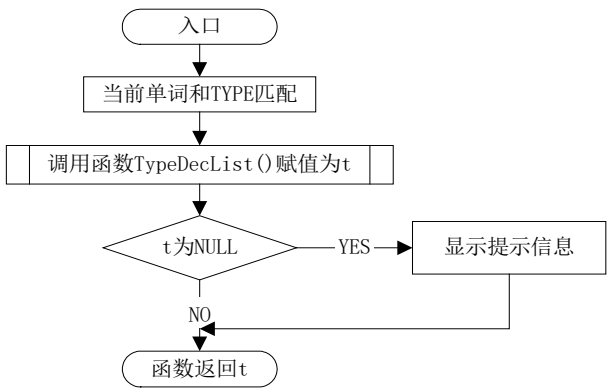


图5.10 类型声明部分处理函数typeDeclaration()的算法框图

7．类型声明中的其他函数

产生式： < TypeDecList > ::= TypeId = TypeDef ； TypeDecMore

函数声明： TreeNode * TypeDecList(void)

算法说明： 该函数根据产生式,创建新的声明类型节点 t。如果申请成功,调用函数 TypeId()，匹配保留字 EQ，调用函数 TypeDef()，匹配保留字 SEMI，调用函数 TypeDecMore()，返回值赋值给 t 的成员 sibling。函数返回 t。

算法框图： 见图 5.11。

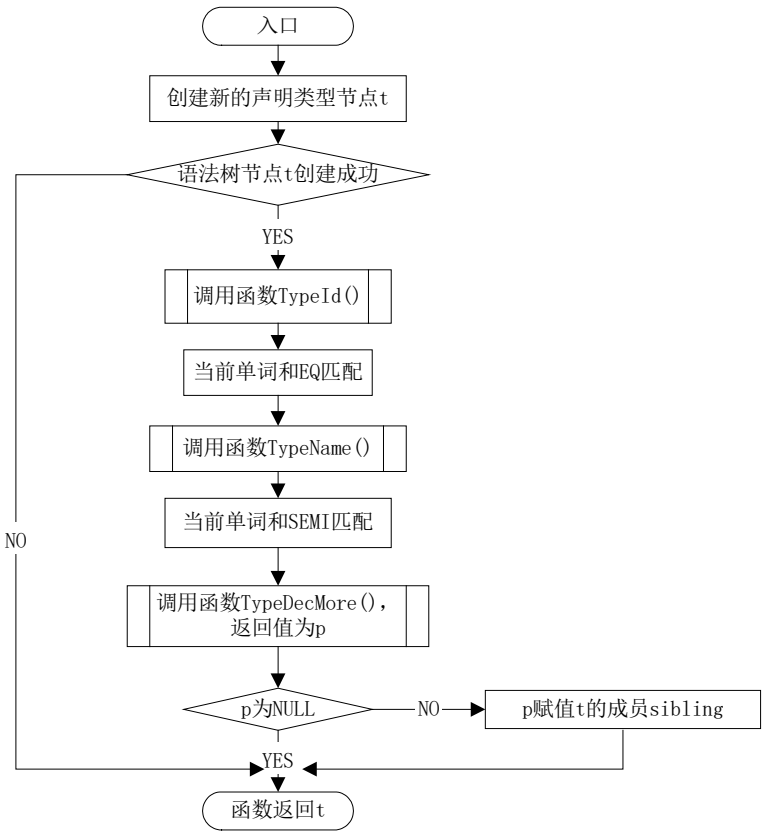


图5.11 类型声明中具体声明部分处理函数TypeDecList()的算法框图

8．类型声明中的其他函数

产生式： < TypeDecMore > ::= ε **{VAR,PROCEDURE,BEGIN}**

| TypeDecList **{ID}**

函数说明： TreeNode * TypeDecMore(void)

算法说明： 该函数根据读入的单词，或者调用函数 TypeDecList（）并赋值 t，返回 t；或者什么都不做，或者跳过此错误单词，读入下一个单词，返回 NULL.

算法框图： 见图 5.12。

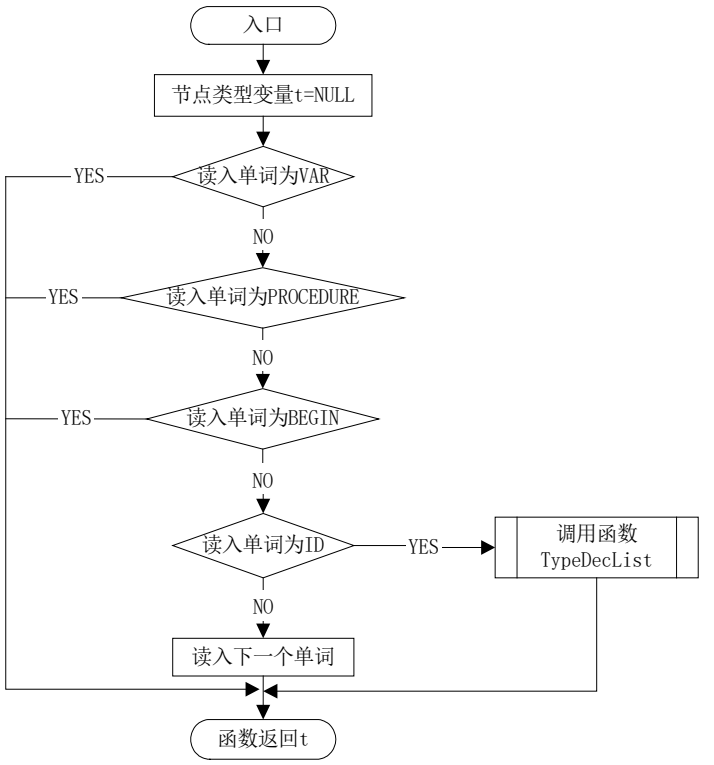


图5.12 类型声明中带有空选择的声明序列部分处理函数TypeDecMore()的算法框图

9．类型声明中新声明的类型名称处理分析程序

产生式： < TypeId > ::= id

函数声明： void TypeId (TreeNode * t)

算法说明： 参数 t，无返回值。该函数根据读入的单词，判断其是否 token.Lex = ID，如果为真，则将 token.Sem 字符串拷贝到参数 t 的成员 attr.name[tnum] 中，其中，tnum 是用来记录 name 个数的临时变量。然后再将 tnum 加 1，送回参数 t 的成员 attr.idnum 中。匹配单词 ID。

算法框图： 见图 5.13。

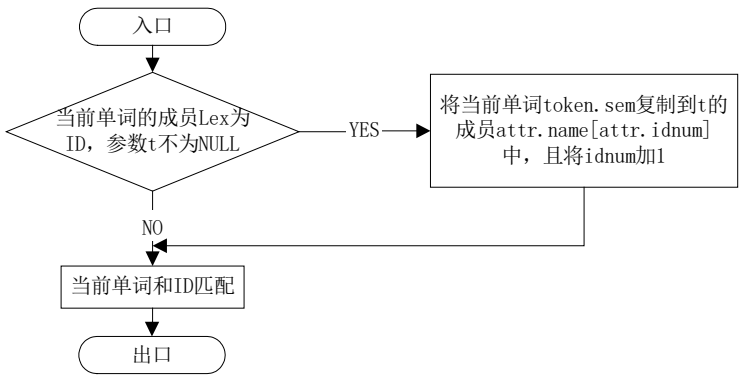


图5.13 类型声明中类型标识符处理函数TypeId()的算法框图

10．类型处理分析程序

产生式： < TypeDef > ::= BaseType {INTEGER,CHAR}
| StructureType {ARRAY,RECORD}
| id {ID}

函数声明： void TypeDef (TreeNode * t)

算法说明： 参数 t，无返回值。该函数根据读入的单词，或者调用 BaseType，或者调用 StructureType,或者复制标识符名称、匹配标识符，或者跳过错误单词、读入下一个单词。注：即为光盘所附程序源代码中的 typeName 函数。

算法框图： 见图 5.14。

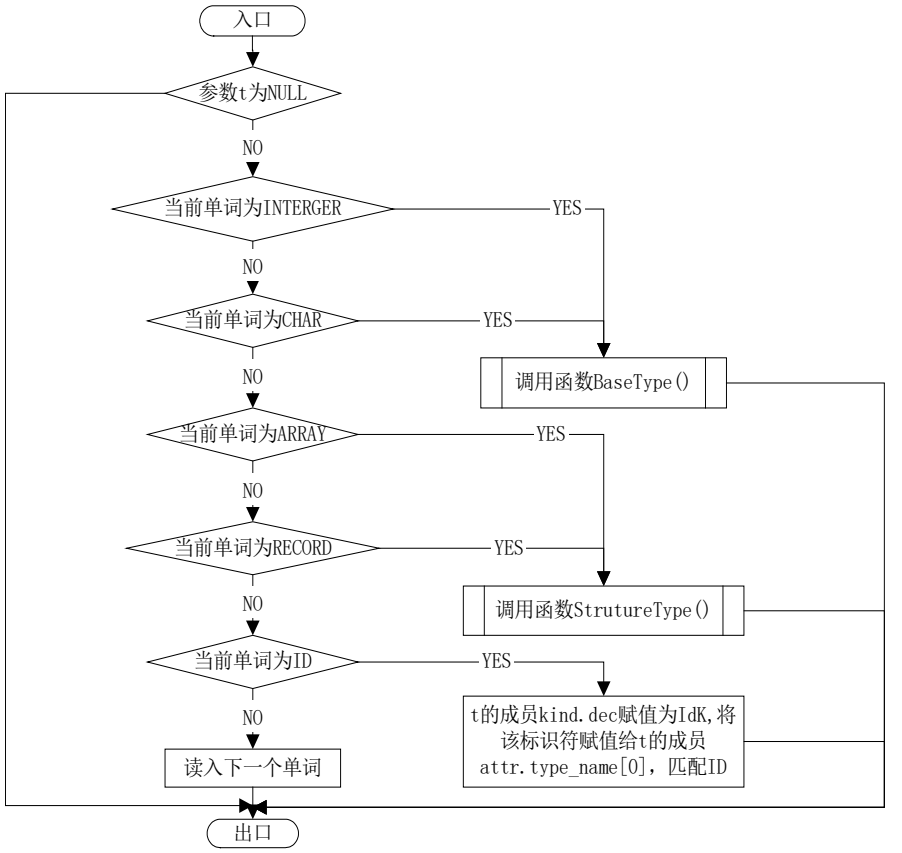


图5.14 具体类型处理函数TypeDef()的算法框图

11．基本类型处理分析程序

产生式： < BaseType > ::= INTEGER {INTEGER}
| CHAR {CHAR}

函数声明： void BaseType (TreeNode * t)

算法说明： 参数 t，无返回值。该函数根据读入的单词判断执行哪个分支。

算法框图： 见图 5.15。

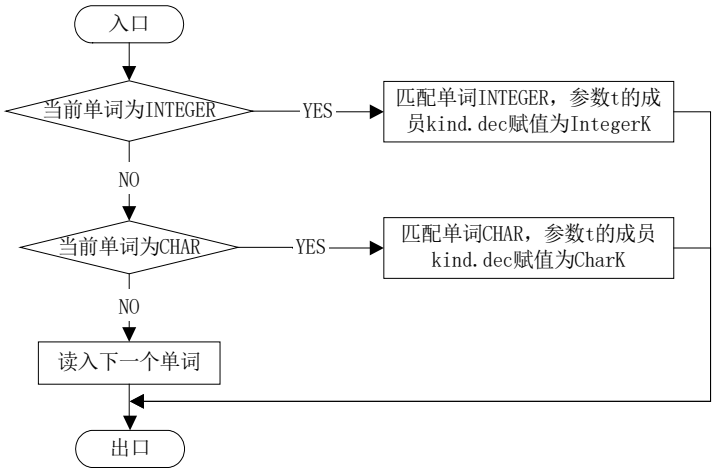


图5.15 基本类型处理函数BaseType()的算法框图

12．结构类型处理分析程序

产生式： < StructureType > ::= ArrayType **{ARRAY}**
| RecType **{RECORD}**

函数声明： void StructureType (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据读入的单词判断选择调用函数 ArrayType()，或者调用函数 RecType()。

算法框图： 见图 5.16。

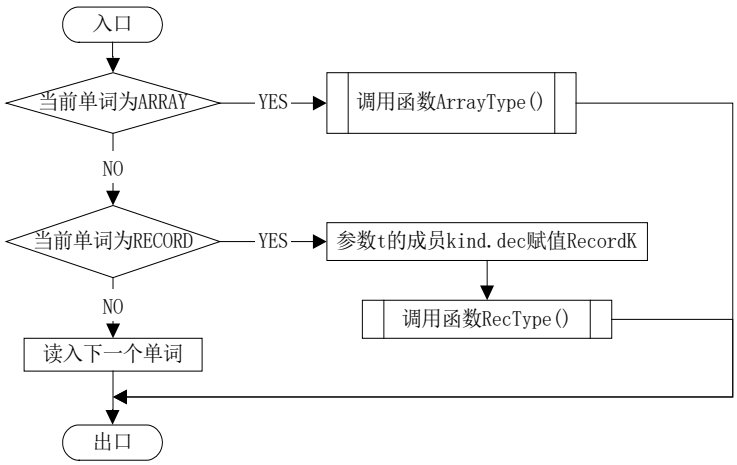


图5.16 结构类型处理函数StructureType()的算法框图

13．数组类型的处理分析程序

产生式： **< ArrayType > ::= ARRAY [low..top] OF** BaseType

函数声明： void ArrayType (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数对读入的单词进行匹配，并记录数组的上

界和下界数值，再匹配保留字，最后调用基本类型函数 BaseType（），记录数组的子类型。

算法框图： 见图 5.17。

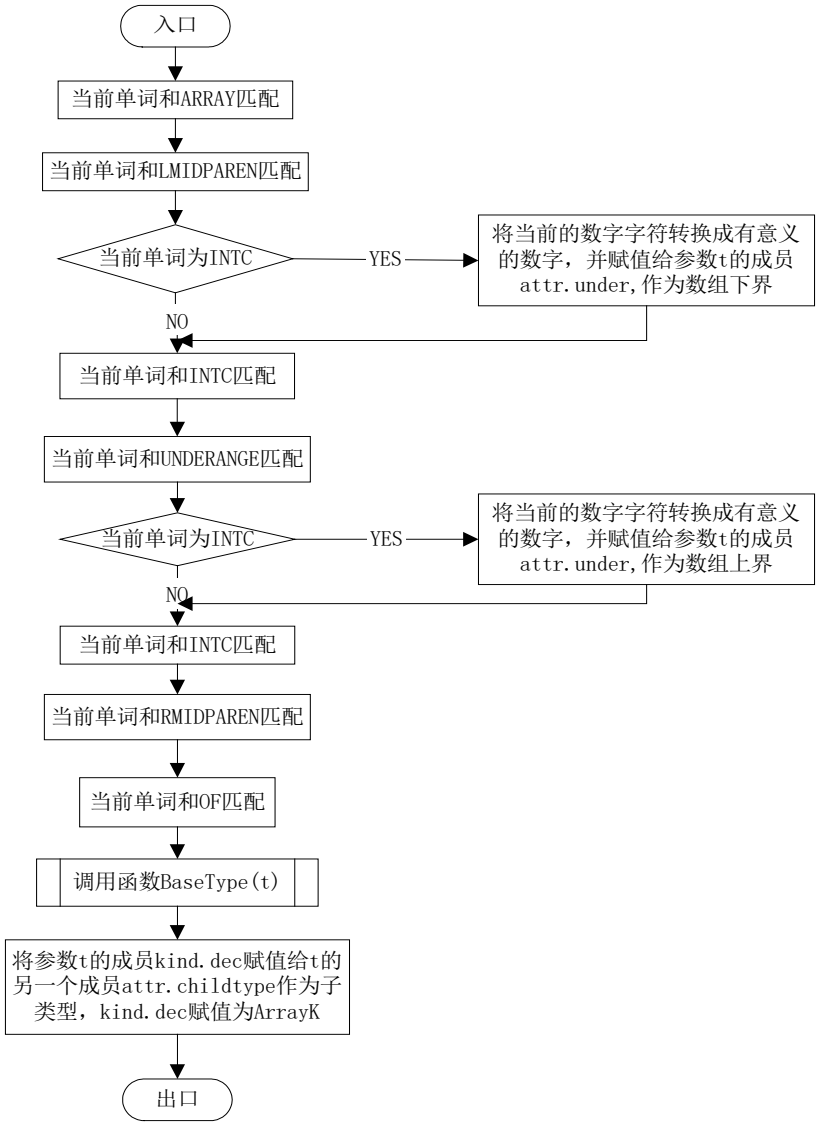


图5.17 数组类型的处理函数ArrayType()的算法框图

14．记录类型的处理分析程序

产生式： < RecType > **::=** **RECORD** FieldDecList **END**

函数声明： void RecType (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数对读入的单词进行匹配，调用记录中的域函数 FieldDecList()，最后再对读入的单词匹配。

算法框图： 见图 5.18。

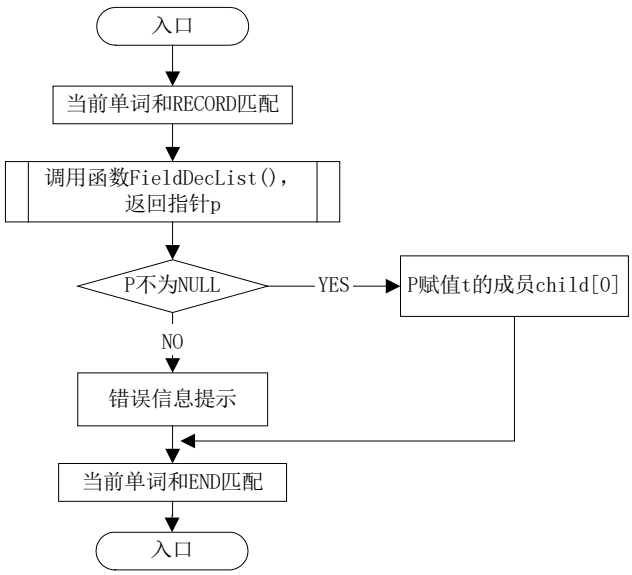


图5.18 记录类型的处理函数RecType()的算法框图

15．记录类型中的域声明处理分析函数

产生式： < FieldDecList > **::=** BaseType IdList **;** FieldDecMore **{INTEGER,CHAR}**

| ArrayType IdList **;** FieldDecMore **{ARRAY}**

函数声明： TreeNode * FieldDecList (void)

算法说明： 该函数根据读入的单词判断记录域中的变量属于哪种类型，根据产生式的 select 集合判断执行哪个分支程序。其中，相同类型的变量 id 用“,”分隔开，其名称记录在语法树的同一个节点中；不同类型的变量的节点互为兄弟节点。

算法框图： 见图 5.19。

16．记录类型中的其他域声明处理分析函数

产生式： < FieldDecMore > **::=** **ε** **{END}**

| FieldDecList **{INTEGER,CHAR,ARRAY}**

函数声明： TreeNode * FieldDecMore (void)

算法说明： 该函数根据文法产生式判断读入单词，若为 END，则不作任何处理；若为 INTEGER，CHAR 或 ARRAY，则调用 FieldDecList()递归处理函数；否则读入下一个 token。函数返回 t

算法框图： 见图 5.20。

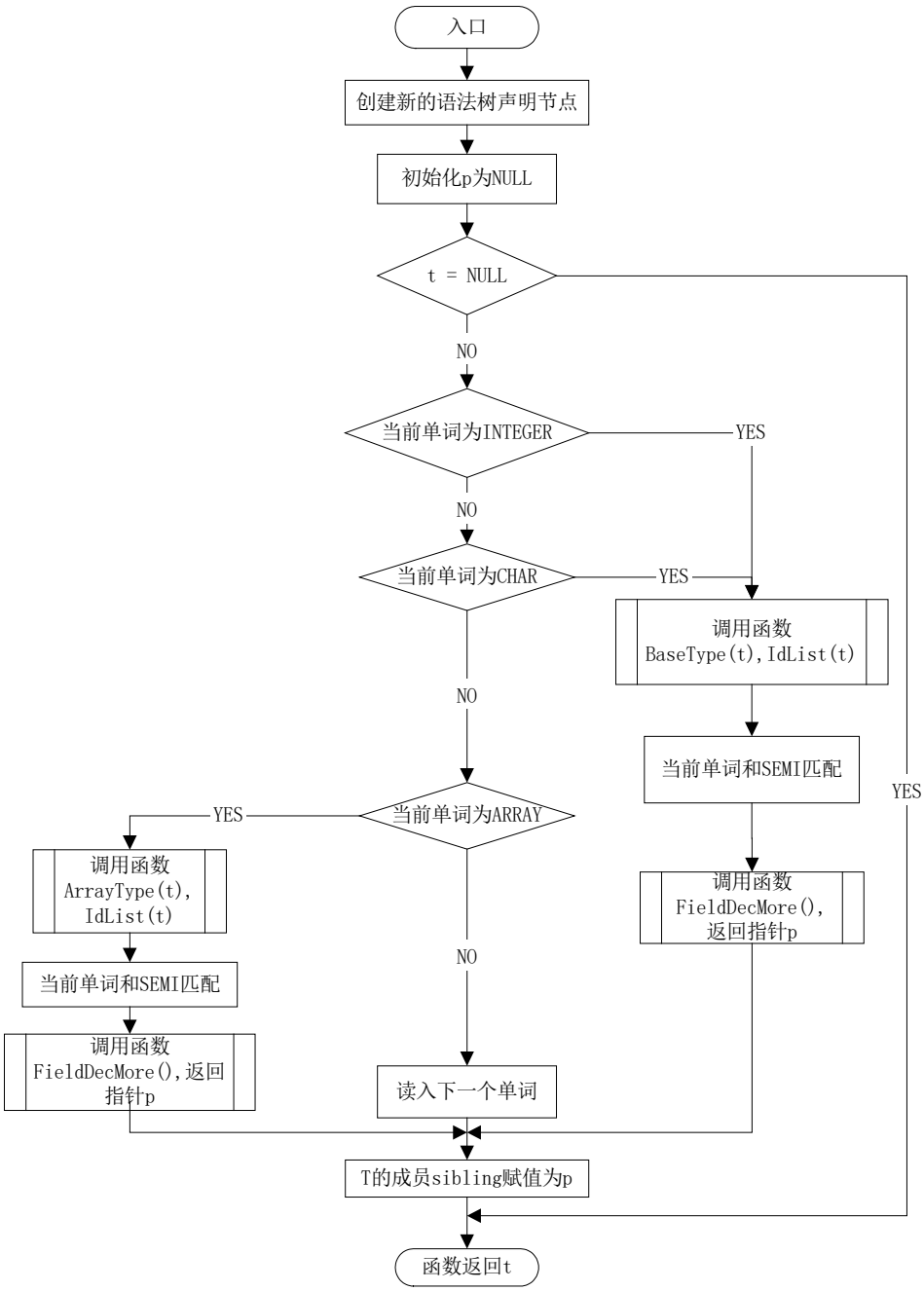


图5.19 记录类型中域的声明部分的处理函数FieldDecList()的算法框图

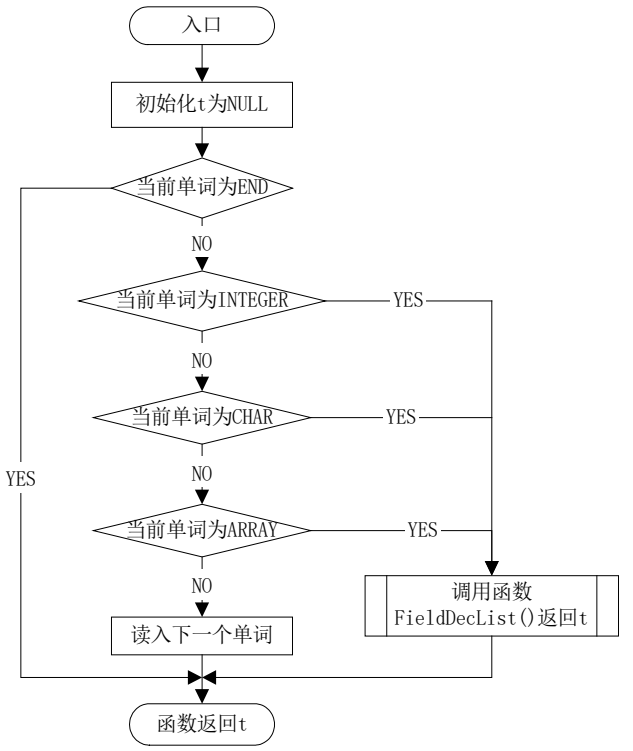


图5.20 记录类型中带空的域声明序列部分的处理函数FieldDecMore()的算法框图

17．记录类型域中标识符名处理分析程序

产生式： **< IdList > ::=** id IdMore

函数声明： void IdList (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据文法产生式，匹配标识符，记录标识符名称，调用递归处理函数 IdMore()。

算法框图： 见图 5.21。

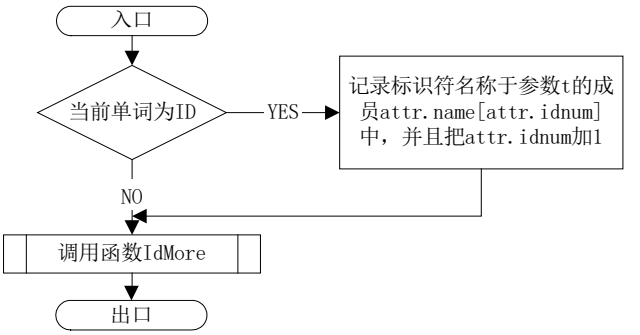


图5.21 记录类型域中标识符名处理函数IdList()的算法框图

18．记录类型域中其他标识符名处理分析程序

产生式： **< IdMore > ::=** **ε** **{SEMI}**

| **,** IdList **{COMMA}**

函数声明： void IdMore (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据文法产生式，判断读入单词，处理调用相应的函数。

算法框图： 见图 5.22。

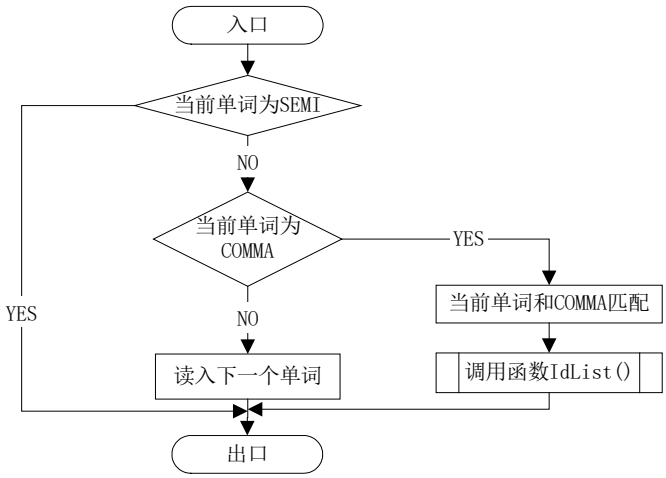


图5.22 记录类型域的带有空的标识符序列部分的处理函数IdMore()的算法框图

19．变量声明处理分析程序

产生式： **< VarDec > ::= ε** **{PROCEDURE,BEGIN}**

| VarDeclaration **{VAR}**

函数声明： TreeNode * VarDec (void)

算法说明： 该函数根据文法产生式，判断读入单词，处理调用相应的函数。处理成功，则返回 t，否则返回 NULL。

算法框图： 见图 5.23。

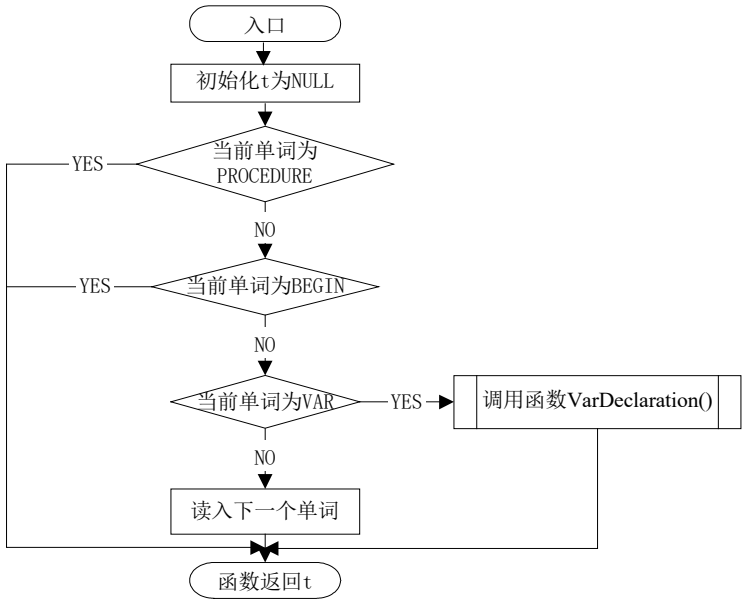


图5.23 带有空的变量声明处理函数VarDec()的算法框图

20．变量声明部分处理程序

产生式：　< VarDeclaration > **::=**　**VAR**　VarDecList

函数声明：　TreeNode * VarDeclaration (void)

算法说明：　根据文法产生式，匹配保留字，调用函数 VarDecList()。如果处理成功，则返回 t，否则返回 NULL。

算法框图：　见图 5.24。

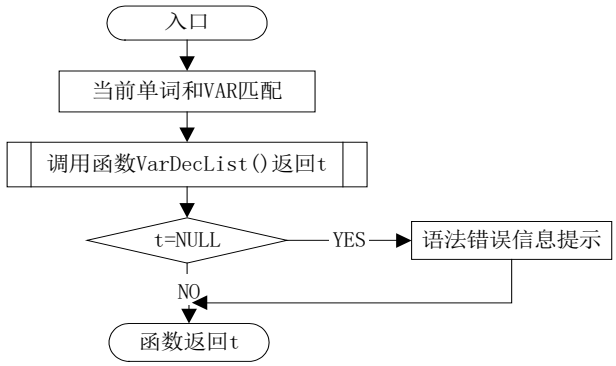


图5.24 变量声明部分处理函数VarDeclaration()的算法框图

21．变量声明部分处理程序

产生式：　< VarDecList > **::=**　TypeDef　VarIdList　;　VarDecMore

函数声明：　TreeNode　*　VarDecList(void)

算法说明：　该函数根据文法产生式调用相应的变量声明部分的处理函数 TypeDef(),VarIdList(),VarDecMore()。如果处理成功，则返回 t，否则返回 NULL。

算法框图：　见图 5.25。

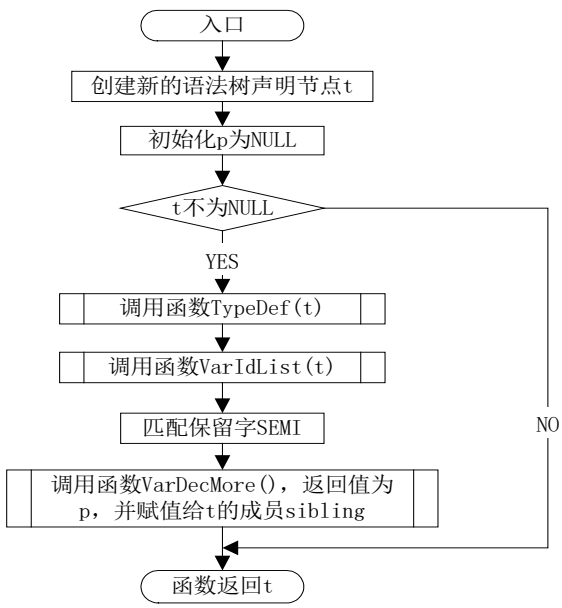


图5.25 变量声明序列部分处理函数VarDecList()的算法框图

22．变量声明部分处理程序

产生式：　< VarDecMore > ::= ε　　{PROCEDURE,BEGIN}
　　| VarDecList
　　{INTEGER,CHAR,RECORD,ARRAY,ID}

函数声明：　TreeNode * VarDecMore(void)

算法说明：　该函数根据文法产生式，由读入单词的不同判断执行哪个分支，最后返回 t。

算法框图：　见图 5.26。

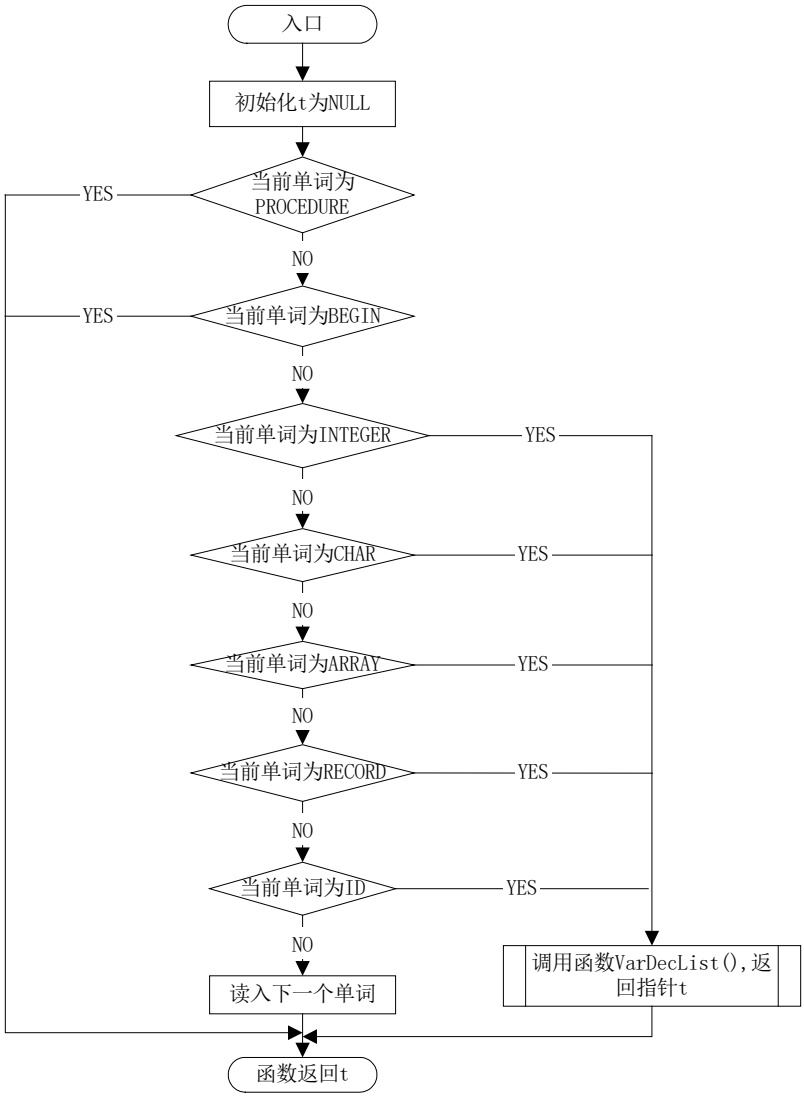


图5.26 带有空的变量声明序列部分处理函数VarDecMore()的算法框图

23．变量声明部分变量名部分处理程序

产生式： < VarIdList > **::=** id VarIdMore
函数声明： void VarIdList(TreeNode * t)
算法说明： 参数为 t，无返回值。该函数根据文法产生式，匹配标识符名 ID，记录标识符名称，调用函数 VarIdMore()。
算法框图： 见图 5.27。

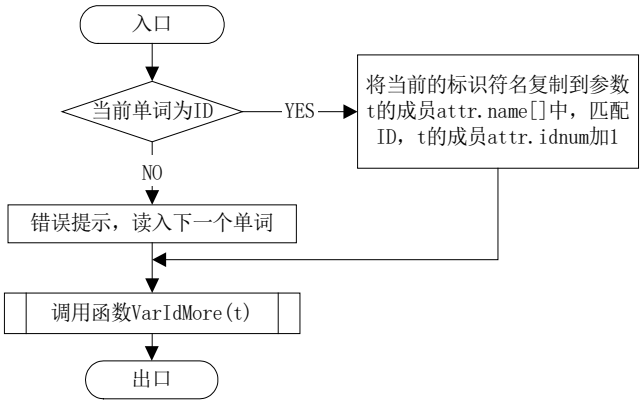


图5.27 变量声明中变量名序列的处理函数VarIdList()

24．变量声明部分变量名部分处理程序
产生式： < VarIdMore > **::=** **ε** **{SEMI}**
| **，** VarIdList **{COMMA}**
函数声明： void varIdMore (TreeNode * t)
算法说明： 参数 t，无返回值。该函数根据文法产生式，由读入的单词判断执行哪个分支。（判断是否还有其他变量）
算法框图： 见图 5.28。

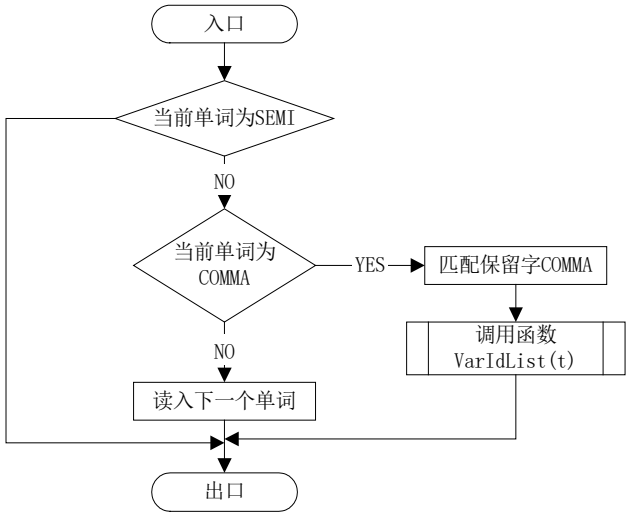


图5.28 变量声明中带有空的变量名处理函数VarIdMore()的算法框图

25．过程声明部分总的处理分析程序

产生式： < ProcDec > ::= ε {BEGIN}

| ProcDeclaration {PROCEDURE}

函数声明： TreeNode * ProcDec(void)

算法说明： 该函数根据文法产生式，对当前单词进行判断，若为 BEGIN，则不作任何动作；若为 PROCEDURE，则调用过程声明函数 ProcDeclaration(t)；否则，读入下一个单词。函数返回 t。

算法框图： 见图 5.29。

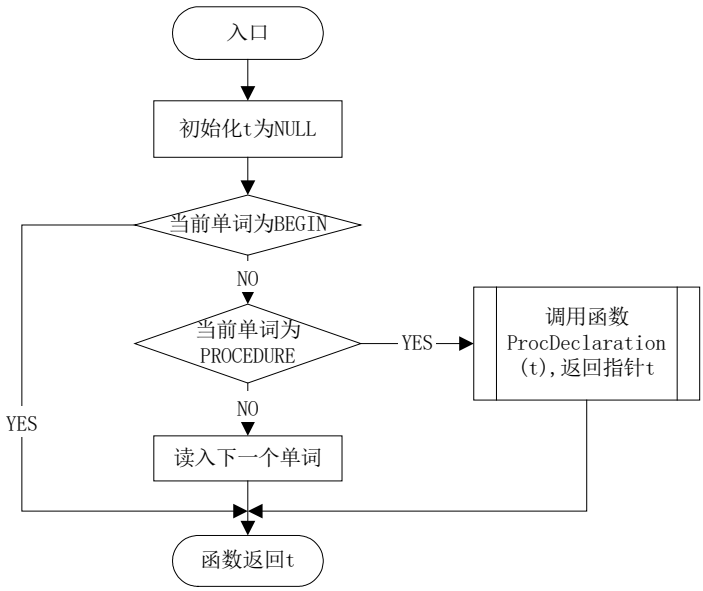


图5.29 带有空的过程声明部分处理函数ProcDec()的算法框图

26．过程声明部分具体的处理分析程序

产生式： < ProcDeclaration > ::= **PROCEDURE**
ProcName **(**ParamList **)**
ProcDecPart
ProcBody

函数声明： TreeNode * ProcDeclaration (void)

算法说明： 该函数根据文法产生式，匹配保留字，调用过程名函数 ProcName()，参数函数 ParamList()，过程体内部声明部分函数 ProcDecPart()，过程体函数 ProcBody()。如果处理成功，返回 t；否则返回 NULL。

算法框图： 见图 5.30。

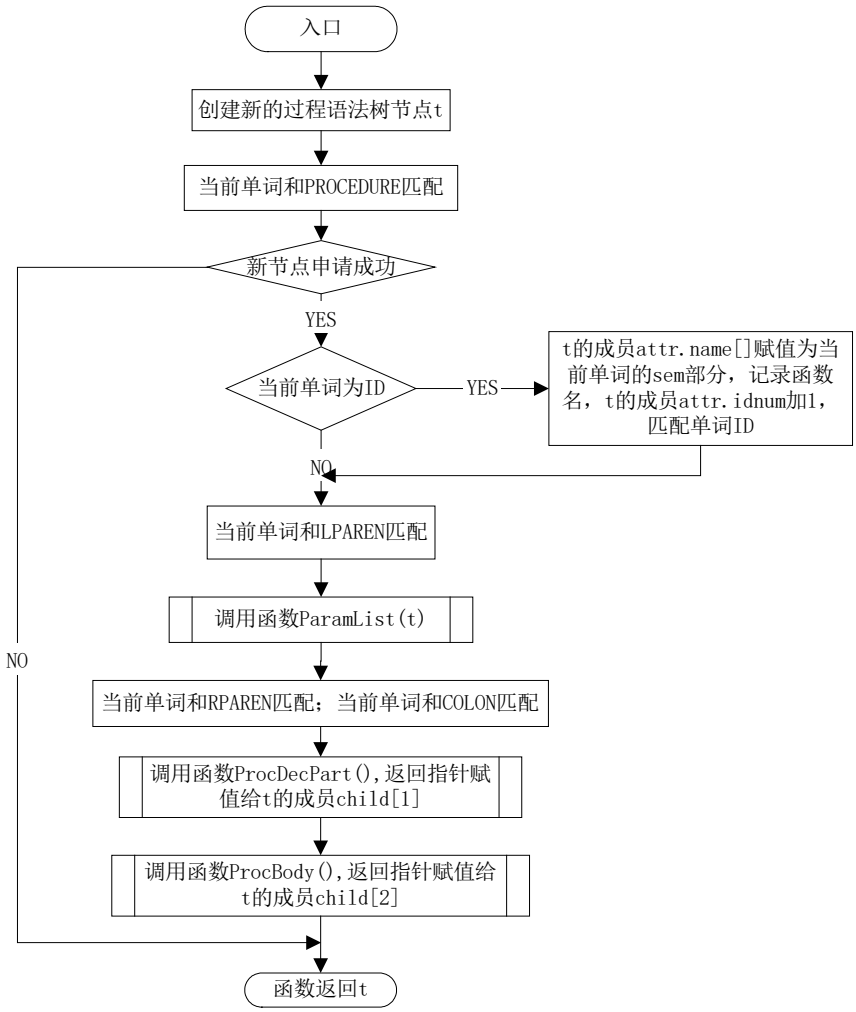


图5.30 过程声明部分的处理函数ProcDeclaration()的算法框图

27. 过程声明中的参数声明部分的处理分析程序

产生式： **< ParamList > ::= ε** **{RPAREN}**

| ParamDecList

{INTEGER,CHAR,ARRAY,RECORD,ID,VAR}

函数声明： void ParamList(TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据文法产生式，由读入的单词判断，若为 RPAREN，则不作任何动作；若为 INTEGER,CHAR,ARRAY, RECORD,ID,VAR，则调用参数声明序列处理函数 ParamDecList(t)；否则，读入下一个 token。

算法框图： 见图 5.31。

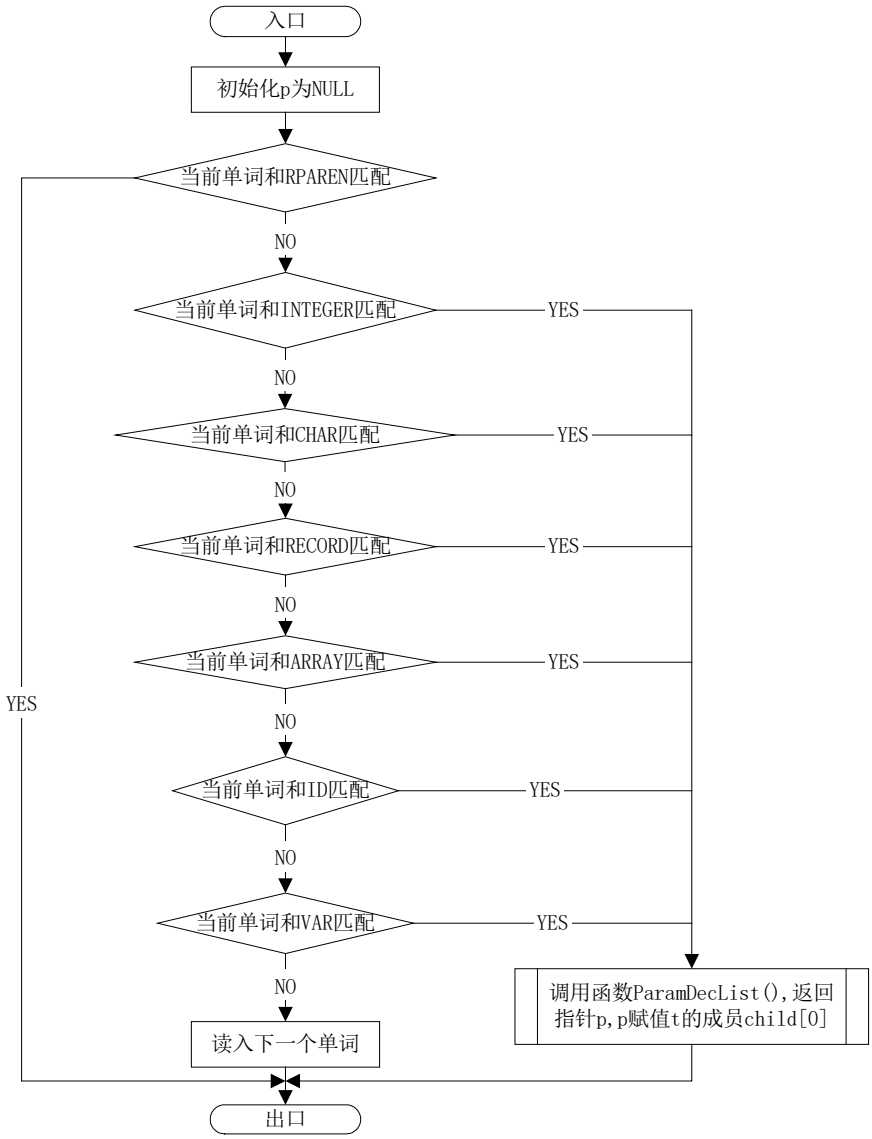


图5.31 过程声明中带有空的参数声明序列的处理函数ParamList

28．过程声明中的参数声明其他部分的处理分析程序

产生式： < ParamDecList > **::=** Param ParamMore

函数声明： TreeNode * ParamDecList(void)

算法说明： 该函数根据文法产生式，调用函数 Param()，返回指针 t；调用函数 ParamMore(),返回指针 p。如果 p 不为 NULL，p 赋值 t 的成员变量 sibling。如果处理成功，则返回指针 t，否则返回 NULL。

算法框图： 见图 5.32。

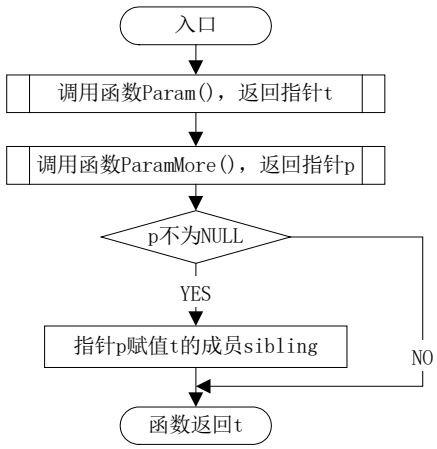


图5.32 过程声明中的参数声明序列的处理函数ParamDecList()的算法框图

29．过程声明中的参数声明其他部分的处理分析程序

产生式： < ParamMore > **::=** **ε** **{RPAREN}**

| **;** ParamDecList **{SEMI}**

函数声明： TreeNode * ParamMore (void)

算法说明： 该函数根据文法产生式，由读入的单词判断执行哪个分支。（看是否还有其他的参数声明）

算法框图： 见图 5.33。

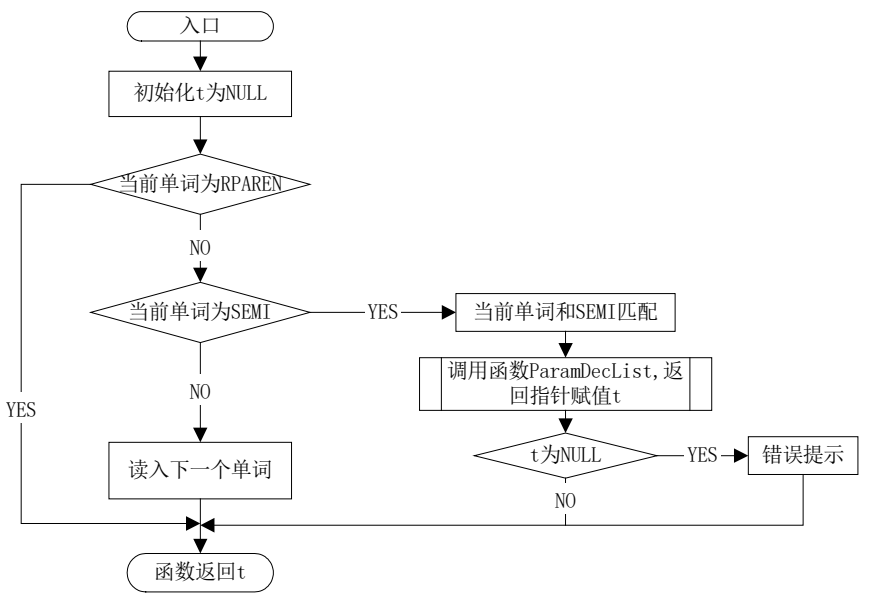


图5.33 过程声明中的参数声明其它部分的处理函数ParamMore()的算法框图

30．过程声明中的参数声明中参数部分的处理分析程序

产生式： < Param > **::=** TypeDef FormList

{INTEGER,CHAR,RECORD,ARRAY,ID}

| **VAR** TypeDef FormList **{VAR}**

函数声明： TreeNode * Param (void)

算法说明： 该函数根据文法产生式，由读入的单词判断执行哪个分支程序，即判断值参还是变参。如果处理成功，则返回 t，否则返回 NULL。

算发框图： 见图 5.34。

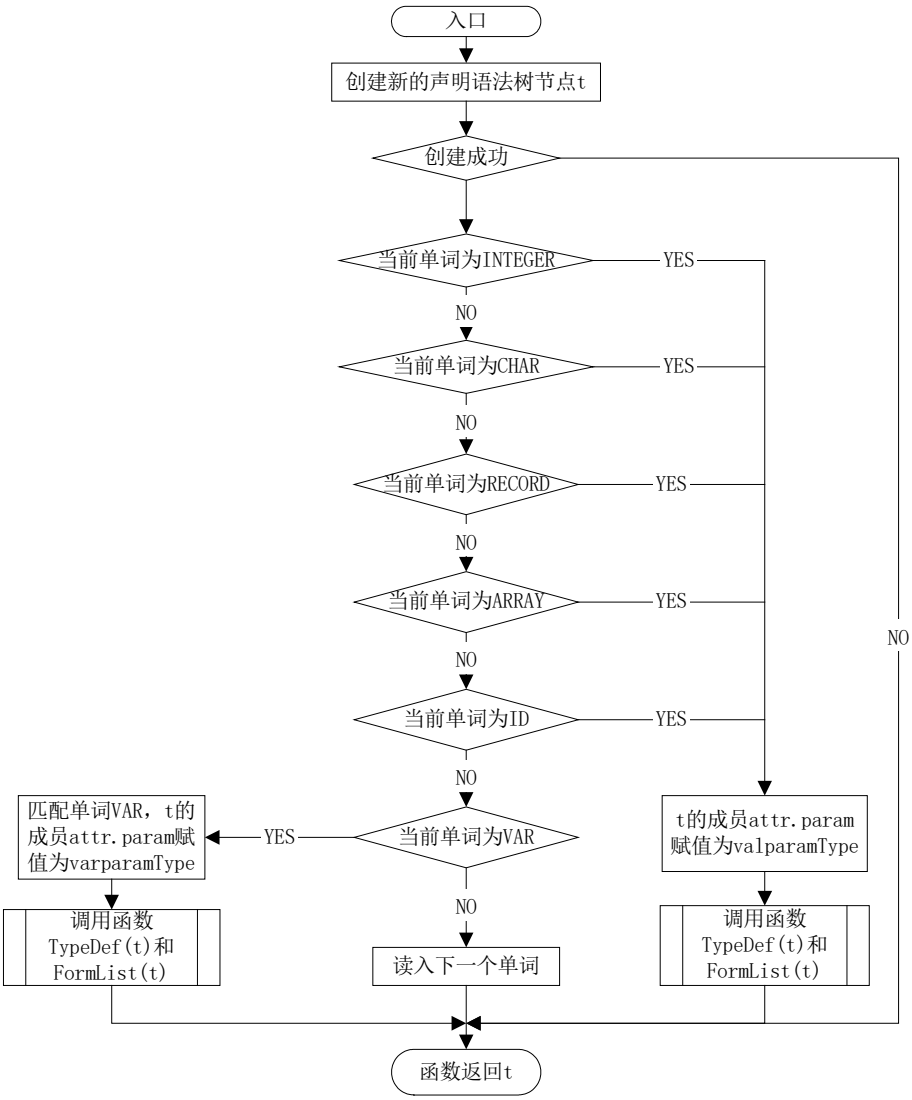


图5.34 过程声明中的参数声明中具体参数的处理函数Param()的算法框图

31．过程声明中的参数声明中参数名部分的处理分析程序

产生式： < FormList > **::=** id FidMore

函数声明： void FormList (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据文法产生式，记录参数声明中的参数名称，匹配保留字，调用 FidMore()函数。

算法框图： 见图 5.35。

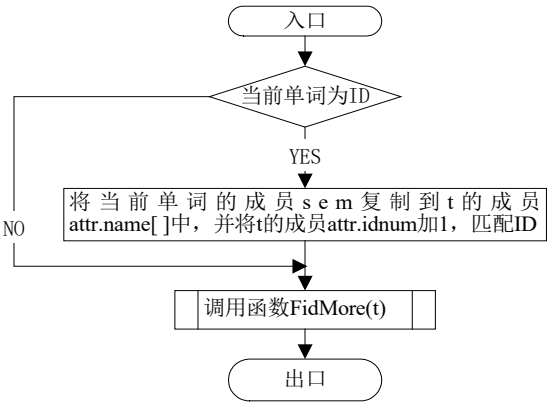


图5.35 过程声明中的参数名序列部分的处理函数算法

32．过程声明中的参数声明中参数名部分的处理分析程序

产生式： < FidMore > ::= ε **{SEMI,RPAREN}**

| , FormList **{COMMA}**

函数声明： void FidMore (TreeNode * t)

算法说明： 参数为 t，无返回值。该函数根据文法产生式，由读入的单词判断执行哪个分支程序。

算法框图： 见图 5.36。

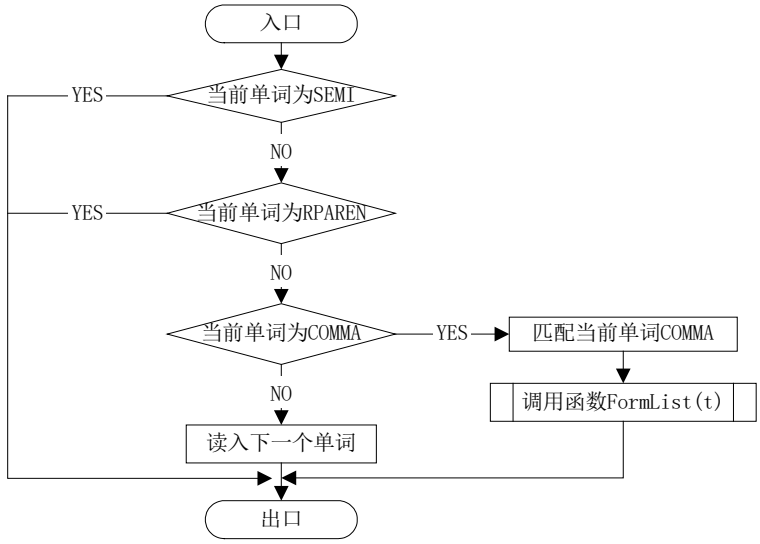


图5.36 过程声明中带空的参数名部分的处理函数FidMore的算法框图

33．过程中声明部分的分析处理程序

产生式：　< ProcDecPart > **::=**　DeclarePart

函数声明：　TreeNode * ProcDecPart(void)

算法说明：　该函数根据文法产生式，调用声明部分函数 DeclarePart()，如果处理成功，则返回 t，否则返回 NULL。

算法框图：　见图 5.37。

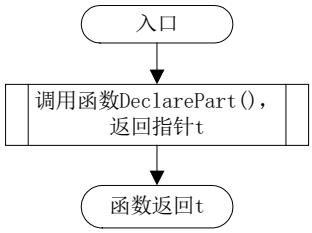


图5.37 过程中声明部分的处理函数ProcDecPart()的算法框图

34．过程体部分处理分析程序

产生式：　< ProcBody > **::=**　ProgramBody

函数声明：　TreeNode * procBody (void)

算法说明：　该函数根据文法产生式，调用程序体部分函数 ProgramBody()。如果处理成功，则返回 t，否则返回 NULL。

算法框图：　见图 5.38。

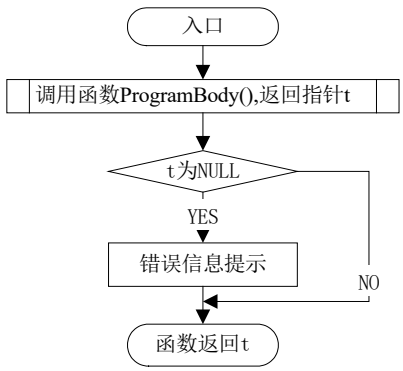


图5.38 过程体部分处理函数ProcBody()的算法框图

35．主程序体部分处理分析程序

产生式：　< ProgramBody > **::=**　**BEGIN**　StmList　**END**

函数声明：　TreeNode * ProgramBody (void)

算法说明：　该函数根据文法产生式，创建新的语句标志类型语法树节点，并匹配保留字，调用语句序列函数 StmList()。如果成功处理，则返回 t，否则返回 NULL。

算法框图：　见图 5.39。

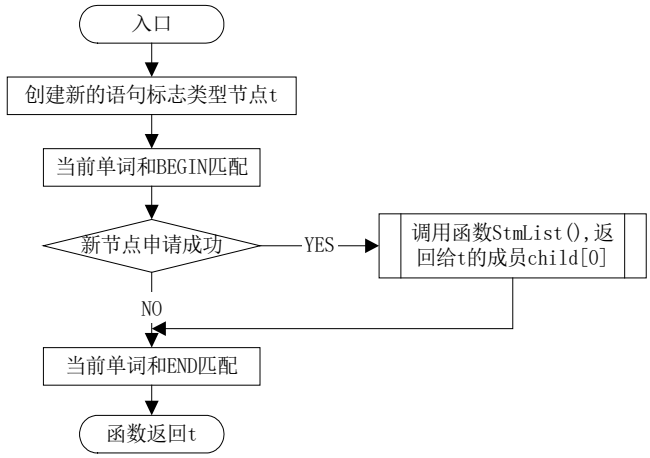


图5.39 主程序体部分的处理函数ProgramPart()的算法框图

36．语句序列部分处理分析程序

产生式： < StmList > **::=** Stm StmMore

函数声明： TreeNode * StmList (void)

算法说明： 该函数根据文法产生式，调用语句处理函数 Stm()，返回指针 t；调用更多语句处理函数 StmMore()，返回指针 p，并使 p 成为 t 的兄弟节点。如果处理成功，返回指针 t，否则返回 NULL。

算法框图： 见图 5.40。

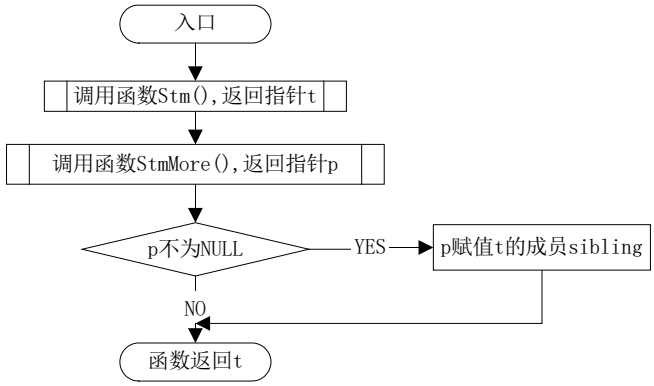


图5.40 语句序列部分的处理函数StmList()的算法框图

37．更多语句部分处理分析程序

产生式： < StmMore > **::=** **ε** **{END,ENDWH}**

| **;** StmList **{SEMI}**

函数声明： TreeNode * StmMore (void)

算法说明： 该函数根据文法产生式，由读入的单词判断执行哪个分支结构，即是否有更多的语句。如果处理成功，则返回 t，否则返回 NULL。

算法框图： 见图 5.41。

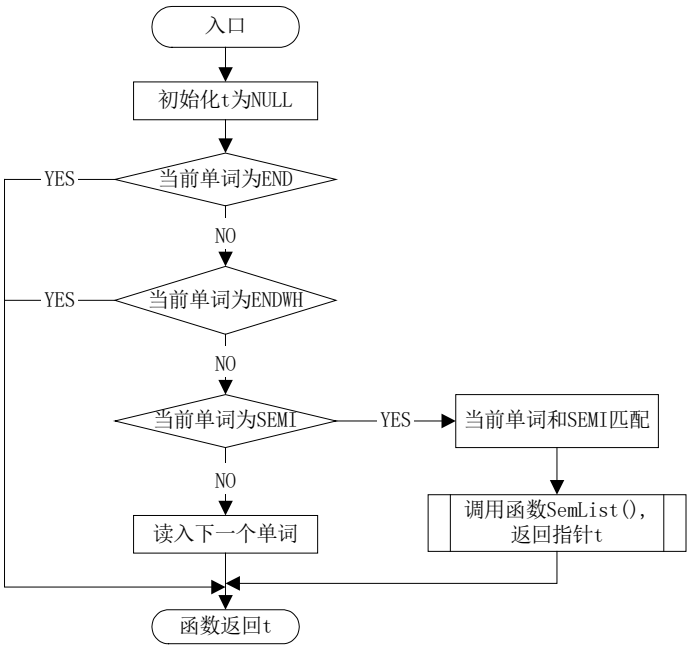


图5.41 带有空的语句序列部分的处理函数StmMore()的算法框图

38．语句递归处理分析程序

产生式： < Stm > **::=** ConditionalStm **{IF}**
| LoopStm **{WHILE}**
| InputStm **{READ}**
| OutputStm **{WRITE}**
| ReturnStm **{RETURN}**
| id AssCall **{id}**

函数声明： TreeNode * Stm(void)

算法说明： 该函数根据读入单词和每条产生式所对应的 predict 集合元素，选择调用相应的语句处理函数，例如，如果当前单词为 IF，则调用条件语句处理函数 ConditionalStm()。其中赋值或调用语句比较特殊，如果当前单词为 ID，则应先匹配 ID，再调用函数 AssCall()。如果处理成功，则函数返回生成的语句类型语法树节点 t，否则返回 NULL。

算法框图： 见图 5.42。

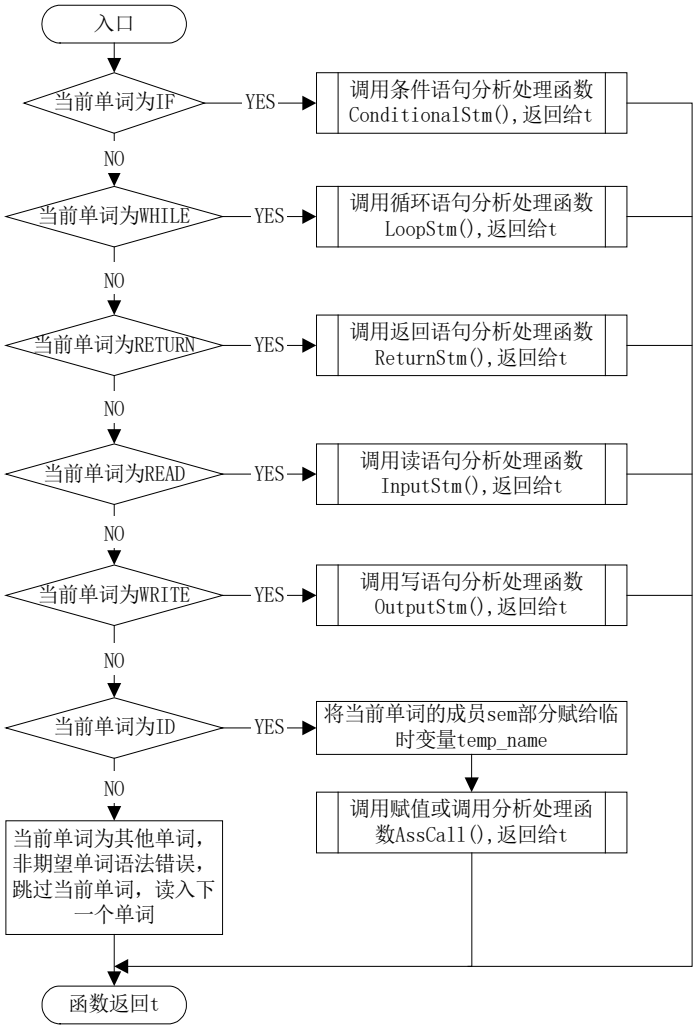


图5.42 语句部分的递归处理函数Stm()的算法框图

39．赋值语句和函数调用语句部分的处理分析程序

产生式： < AssCall > **::=** AssignmentRest **{ASSING}**

| CallStmRest **{RPAREN}**

函数声明： TreeNode * AssCall(void)

算法说明： 由于赋值语句和函数调用语句的开始部分都是标识符，所以该函数根据读入的单词选择调用相应的处理程序（赋值语句处理程序和函数调用语句处理程序）。如果处理成功，则函数返回生成的语句类型语法树节点 t，否则返回 NULL。

算法框图： 见图 5.43。

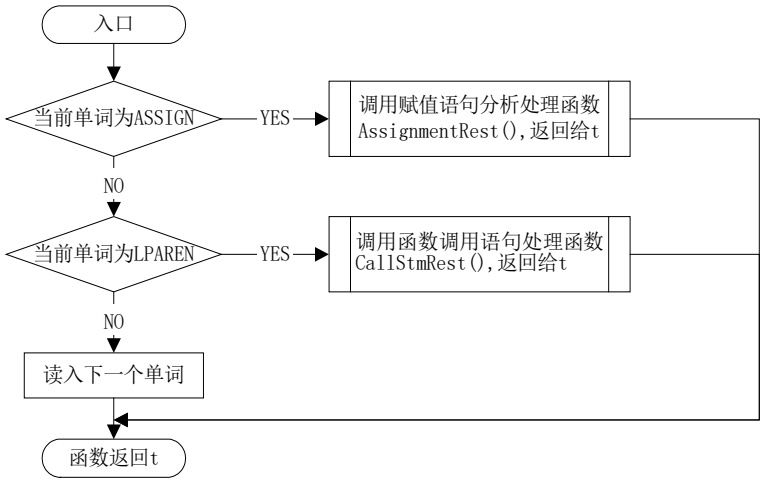


图5.43 赋值语句和函数调用语句部分的处理函数AssCall()的算法框图

40．赋值语句处理分析函数

产生式： < AssignmentRest > **::=** **: =** Exp

函数声明： TreeNode * AssignmentRest(void)

算法说明： 该函数根据文法产生式，创建新的赋值语句类型语法树节点。匹配保留字 EQ，调用表达式函数 Exp()。如果成功处理，则函数返回生成的赋值语句类型语法树节点 t，否则返回 NULL。

算法框图： 见图 5.44。

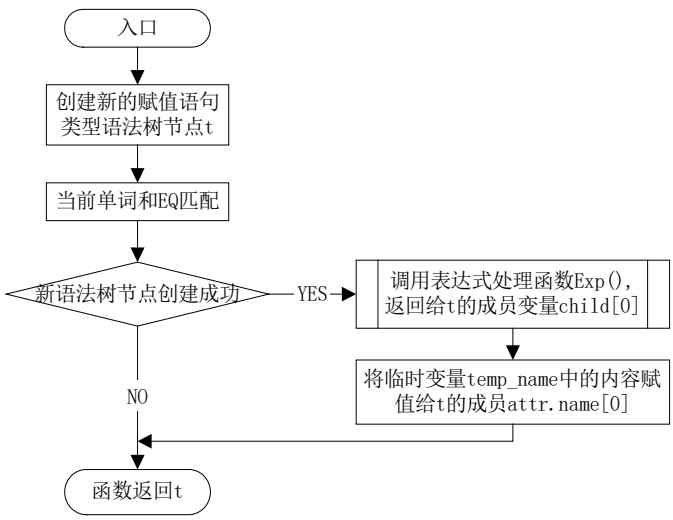


图5.44 赋值语句的处理函数AssignmentRest()的算法框图

41．条件语句处理分析程序

产生式： < ConditionalStm > **::= IF** Exp **THEN** StmL **ELSE** StmL **FI**

函数声明： TreeNode * ConditionalStm(void)

算法说明： 该函数根据文法产生式，匹配保留字 IF，调用表达式函数 Exp()；匹配保留字 THEN，调用语句序列函数 StmL（此处的语句序列函数不同于前处的 StmList()，如果只有一条语句，则语句后不用加分号；否则，如果有两条或者多于两条语句，则要在语句序列开始前加上 BEGIN，在结束后加上 END。其中的最后一条语句末尾亦不用加分号）。如果成功处理，则函数返回生成的条件语句类型语法树节点 t，否则返回 NULL。

算法框图： 见图 5.45。

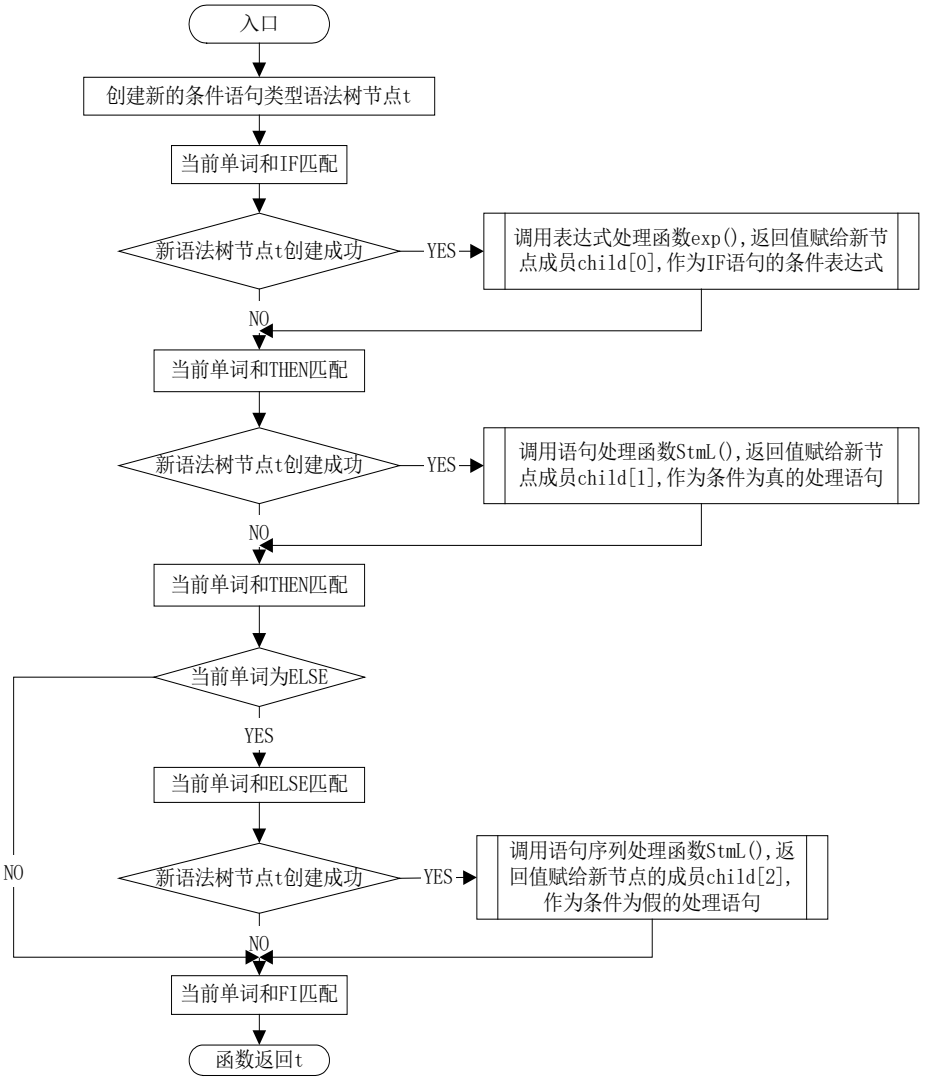


图5.45 条件语句的处理函数ConditionalStm()的算法框图

42．循环语句处理分析程序

产生式： **< LoopStm > ::= WHILE** Exp **DO** StmList **ENDWH**
函数声明： TreeNode * LoopStm(void)
算法说明： 该函数根据产生式，匹配保留字 WHILE,调用表达式处理函数 Exp()，再匹配保留字 DO，调用语句序列处理函数（不同于条件语句中的语句序列函数部分），最后匹配保留字 ENDWH。如果处理成功，则函数返回新生成的循环语句类型的语法树节点 t，否则返回 NULL。
算法框图： 见图 5.46。

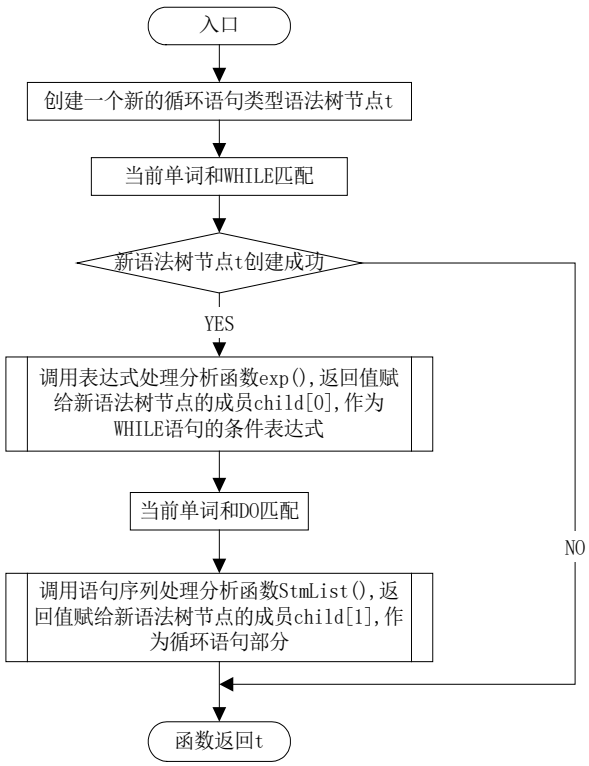


图5.46 循环语句的处理函数LoopStm()的算法框图

43．输入语句的处理分析程序
产生式： **< InputStm > ::= READ(id)**
函数声明： TreeNode * InputStm(void)
算法说明： 该函数根据文法产生式，创建新的输入语句类型语法树节点 t，匹配保留字 READ，LPAREN，记录标识符名称，匹配 ID,RPAREN。如果处理成功，则函数返回新建的语法树节点指针 t，否则返回 NULL。
算法框图： 见图 5.47。

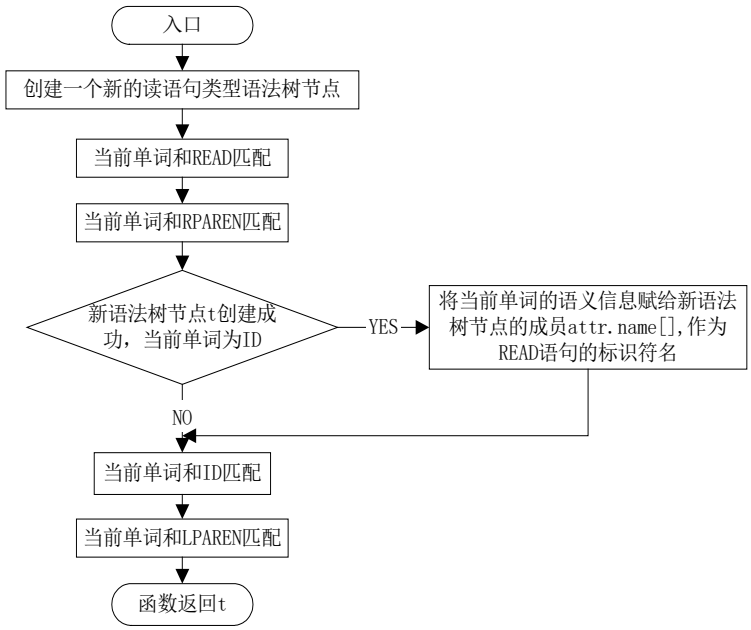


图5.47 输入语句的处理函数InputStm()的算法框图

44．输出语句处理分析程序

产生式： **< OutputStm > ::= WRITE(** Exp **)**

函数声明： TreeNode * OutputStm (void)

算法说明： 该函数根据文法产生式，创建新的输出语句类型语法树节点 t，匹配保留字 WRITE,LPAREN，调用函数 Exp()，匹配保留字 RPAREN。如果处理成功，则函数返回新创建的输出类型节点 t，否则返回 NULL。

算法框图： 见图 5.48。

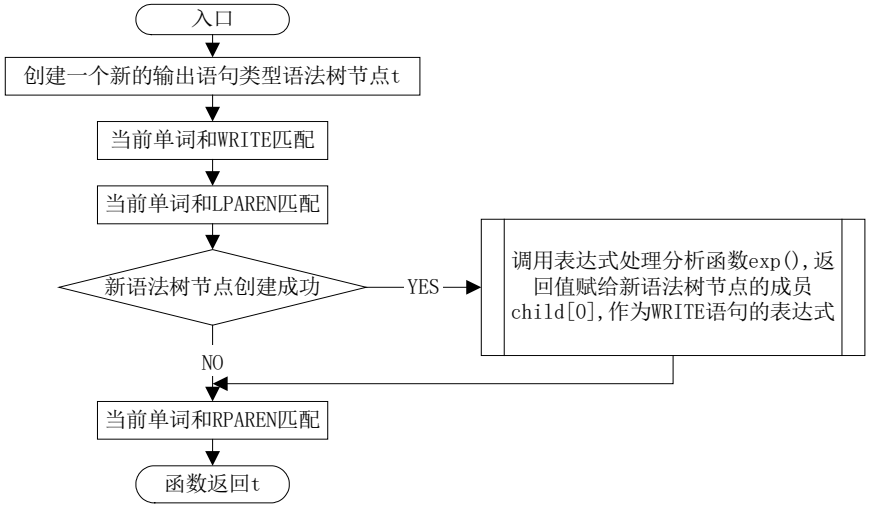


图5.48 输出语句处理函数OutputStm()的算法框图

45．过程返回语句处理分析程序

产生式：　< ReturnStm > **::=　RETURN**

函数声明：　TreeNode * ReturnStm(void)

算法说明：　该函数根据文法产生式，创建新的函数返回类型的语法树节点 t，匹配保留字 RETURN。如果处理成功，则函数返回新的函数返回类型的节点 t，否则返回 NULL。

算法框图：　见图 5.49。

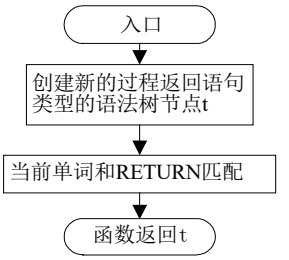


图5.49 过程返回语句处理函数ReturnStm()的算法框图

46．过程调用语句处理分析程序

产生式：　< CallStmRest > **::=　(ActParamList)**

函数声明：　TreeNode * CallStmRest(void)

算法说明：　该函数根据文法产生式，创建新的函数调用类型语法树节点 t，匹配保留字 LPAREN，调用实参处理分析函数 ActParamList()，匹配 RPAREN。如果处理成功，则函数返回新的函数调用类型节点 t，否则返回 NULL。

算法框图：　见图 5.50。

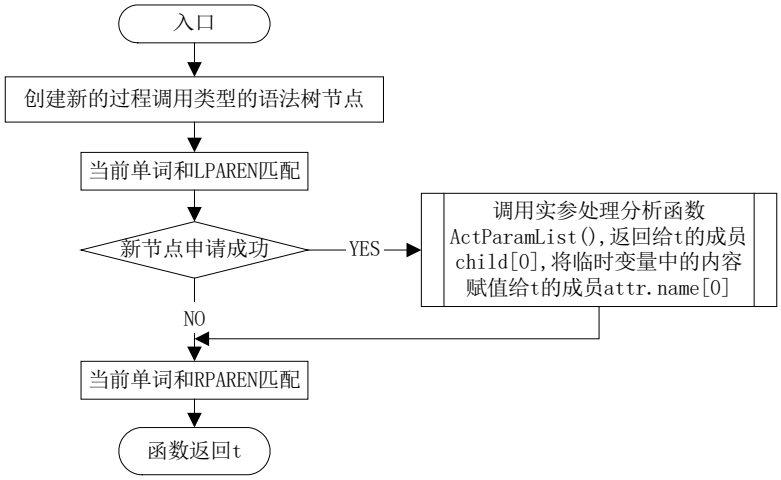


图5.50 过程调用语句的处理函数CallStmRest()的算法框图

47．实参部分处理分析程序

产生式：　< ActParamList > **::=　ε**　　**{RPAREN}**

　　|　Exp　ActParamMore　　**{ID,INTC}**

函数声明：　TreeNode * ActParamList(void)

算法说明：　该函数根据读入单词选择执行哪段程序（有参数或者无参数）。如果处

理成功，则函数返回 t，否则返回 NULL。

算法框图： 见图 5.51。

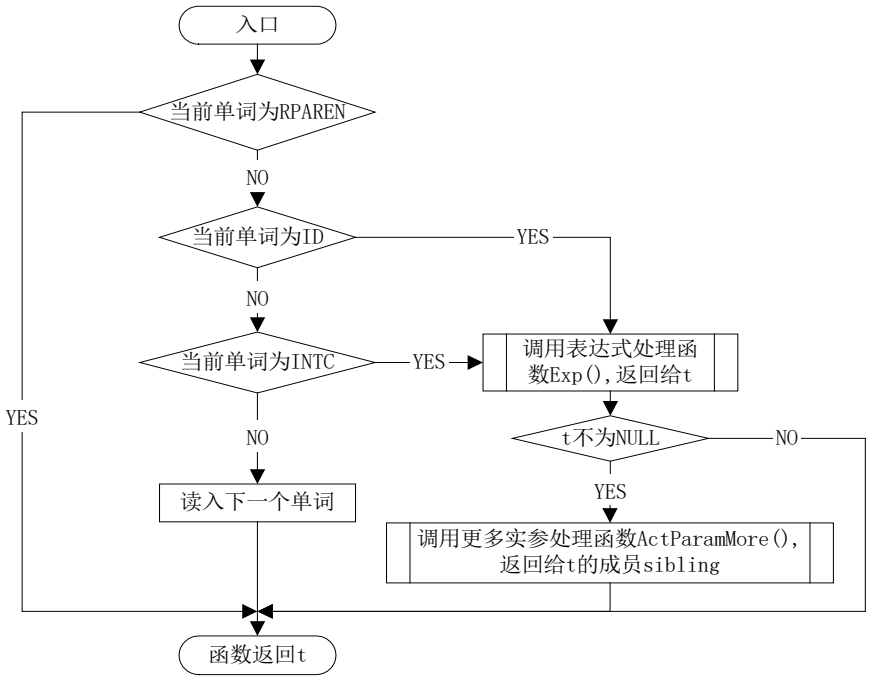


图5.51 实参部分处理函数ActParamList()的算法框图

48．更多实参部分处理分析程序

产生式： < ActParamMore > **::=** **ε** **{RPAREN}**

| , ActParamList **{COMMA}**

函数声明： TreeNode * ActParamMore(void)

算法说明： 该函数根据读入的单词选择执行哪段分支程序。如果处理成功，则函数返回 t,否则返回 NULL。

算法框图： 见图 5.52。

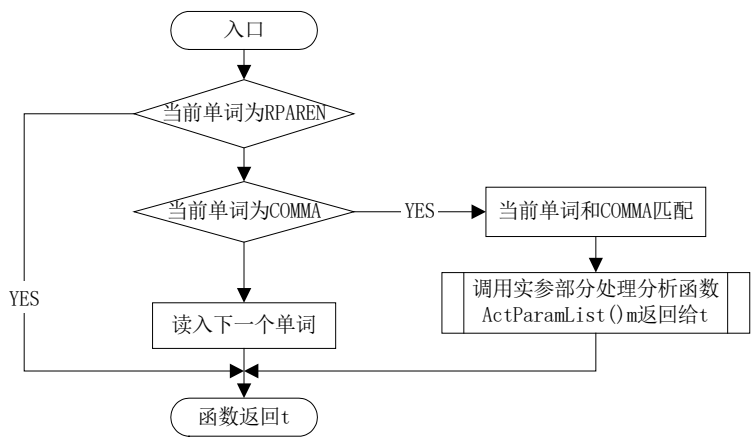


图5.52 带有空的实参序列部分处理函数ActParamMore()的算法框图

49．表达式递归处理分析程序

产生式： < Exp > ::= simple_exp | RelOp simple_exp

函数声明： TreeNode * Exp(void)

算法说明： 该函数根据文法产生式，调用相应递归处理函数。如果处理成功,则函数返回所生成的表达式类型语法树节点；否则，函数返回 NULL。

算法框图： 见图 5.53。

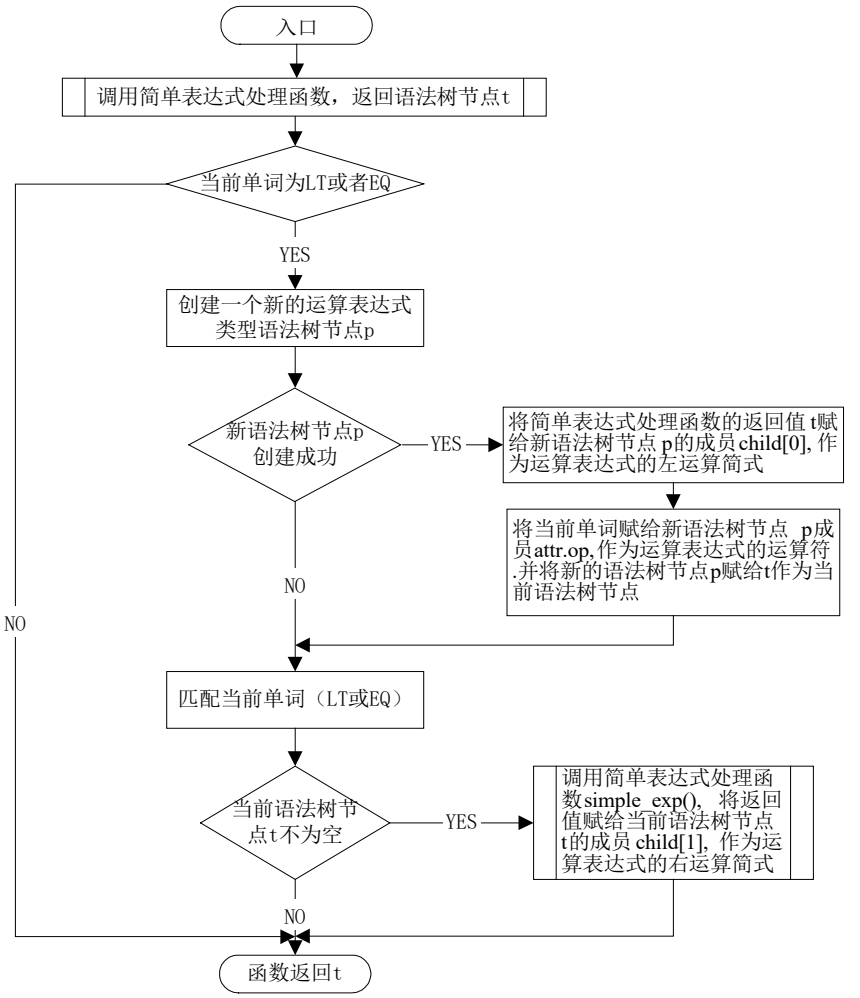


图5.53 表达式递归处理函数Exp()的算法框图

50．简单表达式递归处理分析程序

产生式： < simple_exp > ::= term | PlusOp term

函数声明： TreeNode * Simple_exp(void)

算法说明： 该函数根据文法产生式，调用相应递归处理函数。如果处理成功，则函数返回生成的表达式类型语法树节点；否则，函数返回 NULL。

算法框图： 见图 5.54。

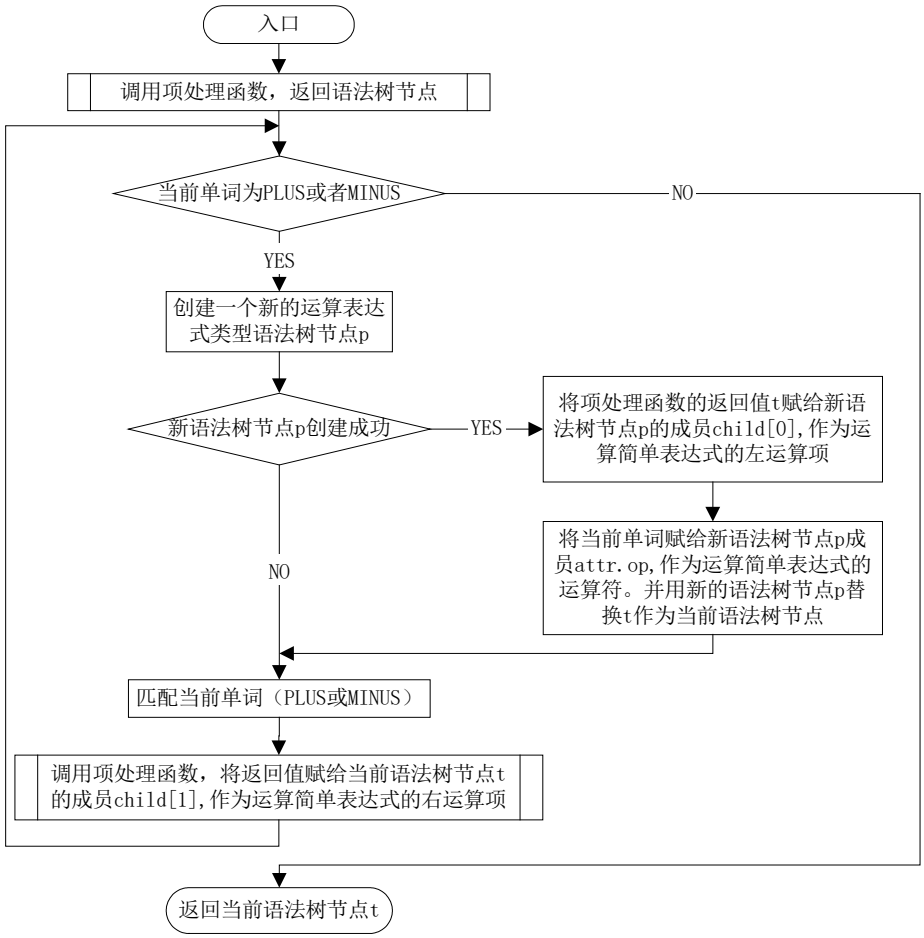


图5.54 简单表达式递归处理函数simple_exp()的算法框图

51．项递归处理分析程序。

产生式： < term > ::= factor | TimesOp factor

函数声明： TreeNode * term(void)

算法说明： 该函数根据文法产生式，调用相应递归处理函数。如果处理成功，函数返回生成的表达式类型语法树节点；否则，函数返回 NULL。

算法框图： 见图 5.55。

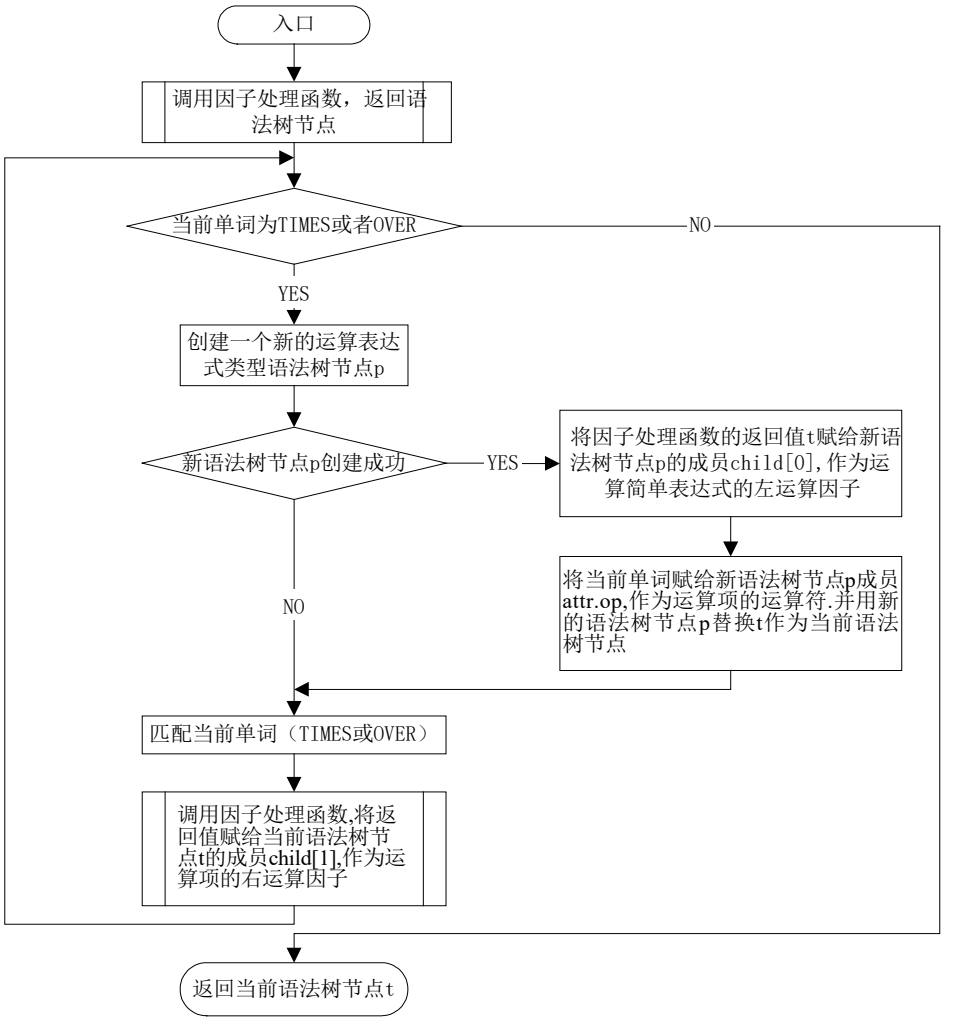


图5.55 项递归处理函数term()的算法框图

52．因子递归处理分析程序

产生式：　< factor > **::=** **(** Exp **)** **{RPAREN}**

| **INTC** **{INTC}**

| variable **{ID}**

函数声明：　TreeNode * factor(void)

算法说明：　该函数根据文法产生式，调用相应的递归处理函数。如果处理成功，函数返回生成的表达式类型语法树节点；否则函数返回 NULL。

算法框图：　见图 5.56。

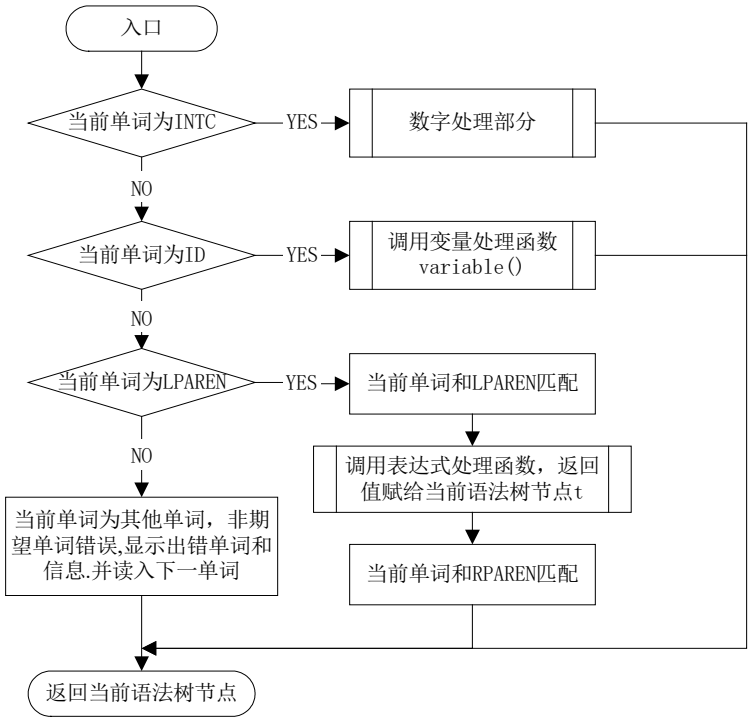


图5.56 因子递归处理函数factor()的算法框图

53．变量处理程序

产生式： **<variable>** **::=** id variMore

函数声明： TreeNode * variable(void)

算法说明： 该函数根据产生式，生成一新的表达式节点 t，如果创建成功且当前单词为 ID，则记录 ID 并匹配，调用处理其余变量部分处理函数 variMore(t)，返回 t。

算法框图： 见图 5.57。

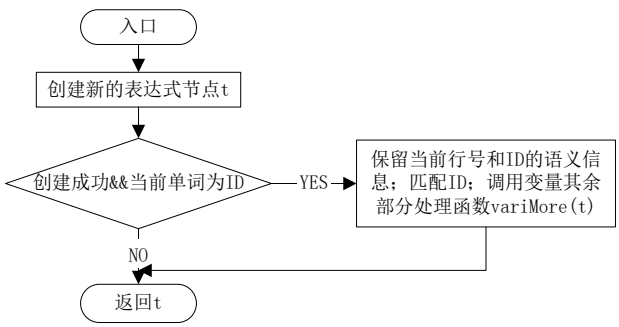


图5.57 变量处理函数variable的算法框图

54．变量其余部分处理程序

产生式： **<variMore>** **::=** **ε** **{ASSIGN,TIMES,EQ,LT,PLUS,MINUS,**

OVER,RPAREN,RMIDPAREN,SEMI,
COMMA,THEN,ELSE,FI,DO,ENDWH,END}
| **[**exp**]** **{LMIDPAREN}**
| **.** fieldvar **{DOT}**

函数声明： void variMore(TreeNode * t)

算法说明： 该函数根据文法产生式，调用相应的递归处理函数。如果当前单词为 ASSIGN 等，则不作任何处理；如果当前单词为 LMIDPAREN，则调用 exp 处理表达式；如果当前单词为 DOT，则匹配 DOT 处理域变量函数 fieldvar；否则，读下一个 TOKEN。

算法框图： 见图 5.58。

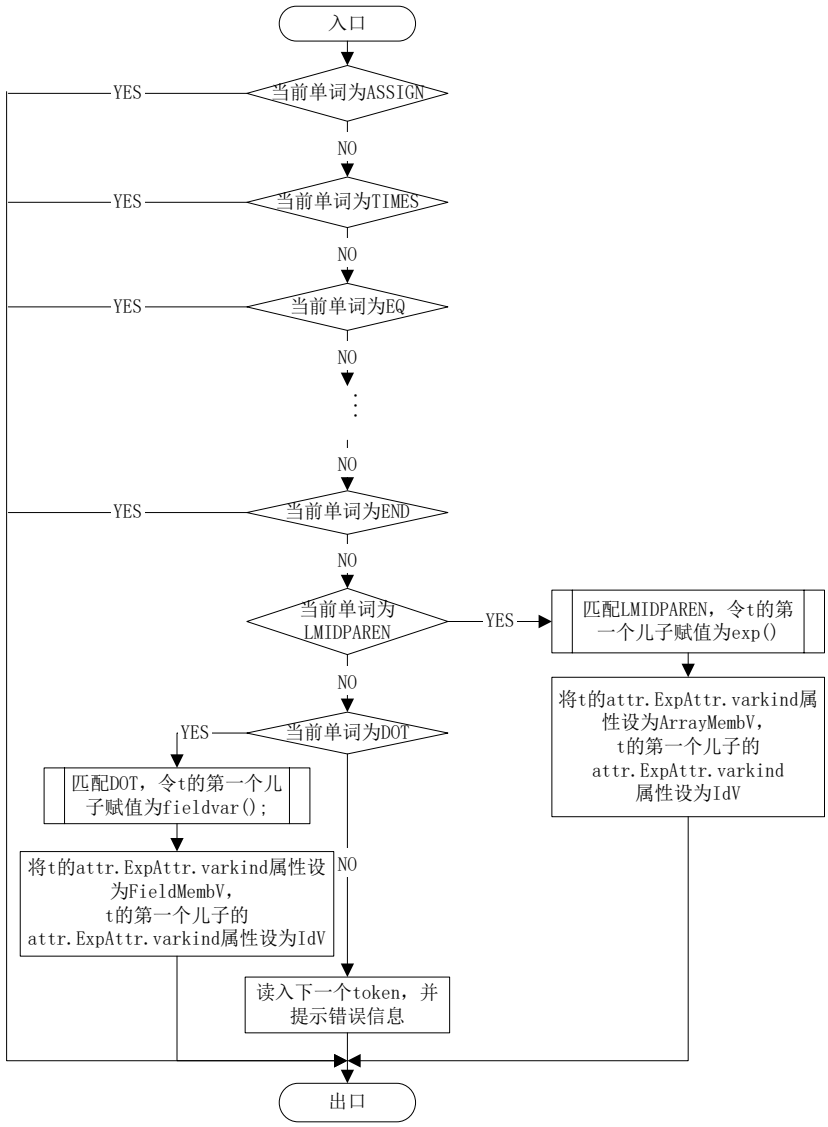


图5.58 变量其余部分处理函数variMore()的算法框图

55．域变量部分处理过程

产生式：　<fieldvar>　**::=**　id　fieldvarMore

函数声明：　TreeNode * fieldvar(void)

算法说明：　该函数根据产生式，生成一新的表达式节点 t，如果创建成功且当前单词为 ID，则记录 ID 并匹配，调用处理其余域变量部分处理函数 fieldvarMore(t)，返回 t。

算法框图：　见图 5.59。

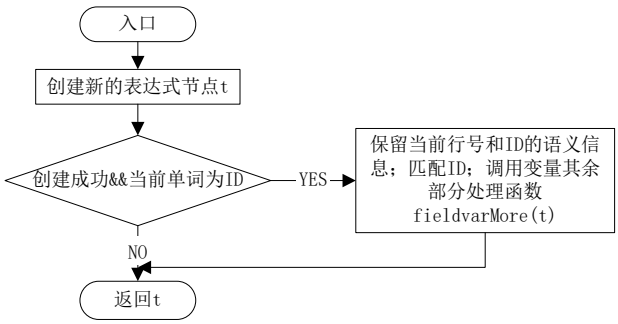


图5.59 域变量处理函数fieldvar()的算法框图

56．域变量其它部分处理过程

产生式：　<fieldvarMore> **::= ε**　　**{ASSIGN,TIMES,EQ,LT,PLUS,MINUS, OVER,RPAREN,SEMI,COMMA, THEN,ELSE,FI,DO,ENDWH,END}**

　　|　[exp]　　**{MIDLPAREN}**

函数声明：　void fieldvarMore (TreeNode * t);

算法说明：　该函数根据文法产生式，调用相应的递归处理函数。如果当前单词为 ASSIGN 等，则不作任何处理；如果当前单词为 LMIDPAREN，则调用 exp 处理表达式；否则，读下一个 TOKEN。

算法框图：　见图 5.60。

57．终极符匹配处理程序

函数声明：　void match(LexType expected)

算法说明：　该函数将当前单词与函数参数给定单词相比较，如果一致，则取下一单词；否则，函数退出语法分析程序。

算法框图：　见图 5.61。

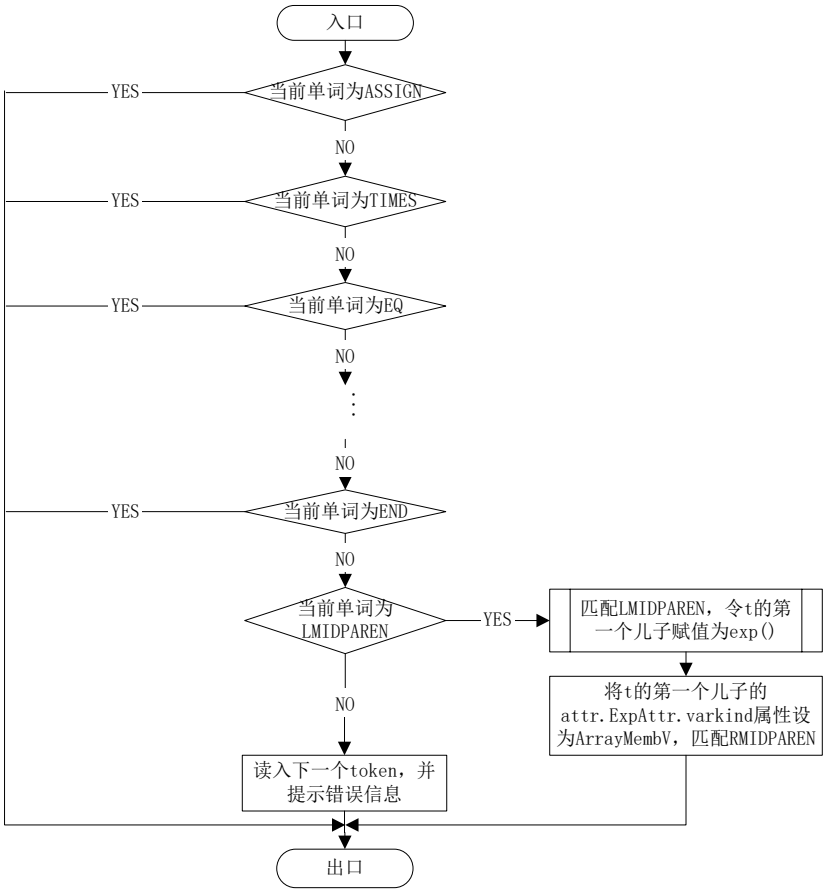


图5.60 域变量其余部分处理函数fieldvarMore()的算法框图

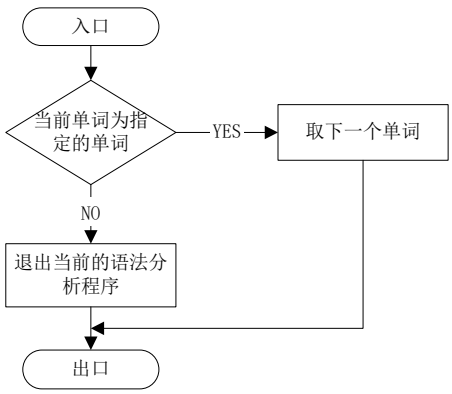


图5.61 终极符匹配处理函数match()的算法框图

5．4 LL(1)语法分析方法的实现

5．4．1 LL(1)语法分析方法的基本原理

LL(1)语法分析方法是一种自顶向下的语法分析方法，它是 LL(k)分析方法的特例，其中的 k 表示向前看 k 个符号的意思。和递归下降法有相同的问题，即在推导过程中如何唯一选择产生式的问题，因此，LL(1)语法分析方法与递归下降法一样对文法做了相同的限制，即：

假设 A 的全部产生式为 $A\rightarrow\alpha_1|\alpha_2|\ldots\ldots|\alpha_n$

则有：

$predict(A\rightarrow\alpha_i)\cap predict(A\rightarrow\alpha_j)=\Phi$,当 $i\neq j$.

有了上面的限制条件，可以保证选择产生式的唯一性，满足上面条件的文法称为LL(1)文法。

实际上，LL(1)分析方法和递归下降法的原理是一样的，只是递归下降法隐式地采用了栈（过程调用实际上就是采用栈的方法实现的），而 LL(1)方法则是显式地采用栈来保存当前分析的结果。

LL(1)语法分析程序由两部分组成的：第一部分是语法分析表，也称为 LL(1)分析矩阵；第二部分是语法分析驱动程序。

LL(1)矩阵的作用是对于当前非终极符和当前输入符确定应该选择的语法规则，它的行对应非终极符，列对应终极符，矩阵的值有两种：一种是产生式编号，另外一种是错误编号，其一般形式可定义为：LL(A,a)=k，其中 $A\in V_N$,$a\in V_T\cup\{\#\}$,k 为产生式编号[P]或错误编号。

设有产生式 $A\rightarrow\alpha$ [k]，其中 k 为产生式 $A\rightarrow\alpha$ 的编号。若 $a\in Predict(A\rightarrow\alpha)$，那么有 LL(A,a)=k；若 a 不属于任何一个以 A 为左部的产生式的 Predict 集，则 LL(A,a)=错误编号。

LL(1)分析程序工作过程是首先初始化，即把开始符压入栈中，以后的每步分析必为下面的四种情况之一：

1．分析栈的栈顶元素是终极符，则看其是否与输入流的头符相匹配，如果匹配成功，去掉栈顶元素,并读下一个单词；若匹配不成功，则报错。

2．栈顶是非终极符，则用栈顶和输入流的当前单词去查当前矩阵，如果查得的值是产生式编号，则把对应的产生式右部逆序压入栈中；如果查得的值为错误信息，则报错。

3．栈已空，输入流不空，这时输入流报错。

4．若栈已空，输入流也空，则语法分析成功。

由上可以看出，LL(1)语法分析驱动程序对于任何文法都是一样的，所不同的是不同的文法 LL(1)矩阵是不相同的。所以构造 LL(1)语法分析程序关键是如何构造文

法的 LL(1)矩阵。LL(1)的构造可以是手工操作的，也可以是自动生成的。

5．4．2 SNL 语言的 LL(1)语法分析概述

1．LL(1)分析表的构造：

SNL 语法程序的实现采用手工操作构造 LL(1)分析表。LL(1)分析表用一个二维矩阵表示，其中每个非终极符对应一行，每个终级符对应一列，一个非终极符和一个终极符可以确定矩阵中的一个元素，元素的值表示该非终极符和该终极符对应的产生式。每个矩阵元素都是一个整数，所有元素初始化为 0；构造 LL(1)表时，根据 SNL 语言的文法（见 2.3.2）和各个产生式的 Predict 集（见 5.1.4），填写 LL(1)分析表的内容。由于各产生式编号从 1 开始，所以用编号 0 表示 Error。在根据 LL(1)分析表选择产生式进行推导时，若查到的产生式编号为 0 表示无相应的产生式可选，不匹配错误；否则根据编号得到相应的产生式进行推导。请参考 5.4.3 节源程序中的创建 LL(1)分析表函数 CreatLL1Table。

2．SNL 语法分析的数据结构：

SNL 语言的 LL(1)语法分析程序共用到四个栈，分别称为：**符号栈**，**语法树栈**，**操作符栈**和**操作数栈**。其中，符号栈用于进行 SNL 的 LL(1)语法分析；其它的栈是为了在语法分析的过程中同时生成与源程序结构对应的语法树而设。语法树栈用于生成声明部分和语句部分的语法树；操作符栈和操作数栈用于生成表达式部分的语法树。

3．语法分析树的生成：

用语法制导的方式生成语法分析树时比较复杂，需要用到语法树栈、操作符栈和操作数栈。

（1）处理声明部分和语句部分的语法树生成时，设置一个**语法树栈**，存放语法树节点中指向儿子或者兄弟节点的指针的地址。在生成当前语法树节点时，如果以后需要对其儿子节点或者兄弟节点赋值，则按照处理顺序的逆序将这些儿子节点或者兄弟节点的指针的地址压入语法树栈，后面生成它的儿子节点或者兄弟节点时，只需弹栈，并对相应的指针进行赋值，就可以完成所需的语法树节点的链接。

（2）处理表达式时，需要另外设置两个栈，**操作数栈**和**操作符栈**，遇到操作数压入操作数栈，遇到操作符，则进行判断，如果当前操作符的优先级高于操作符栈的栈顶操作符，直接压入操作符栈；否则，弹出栈顶操作符，并弹出操作数栈顶的两个操作数，生成相应的子树，并对新生成的子树的父节点（操作符节点）进行循环判断。

5．4．3 LL(1)语法分析程序框图

1．主要函数：

（1） 函数 parse( )

函数声明： TreeNode * parseLL ( )

算法说明： LL(1)语法分析主函数。初始化符号栈，调用创建 LL(1)符号表函数，生成语法树的根节点，分别将指向程序体的指针 child[2]的地址，指向声明部分的指针 child[1]的地址，指向程序头的儿子指针 child[0]的地址压入语法树栈。利用 LL(1)分析表和符号栈进行语法分析，并处理标识符不匹配错误和文件提前结束错误，函数处理完毕时，整个语法分析结束，函数返回生成的语法分析树。

算法框图： 见图 5.62。

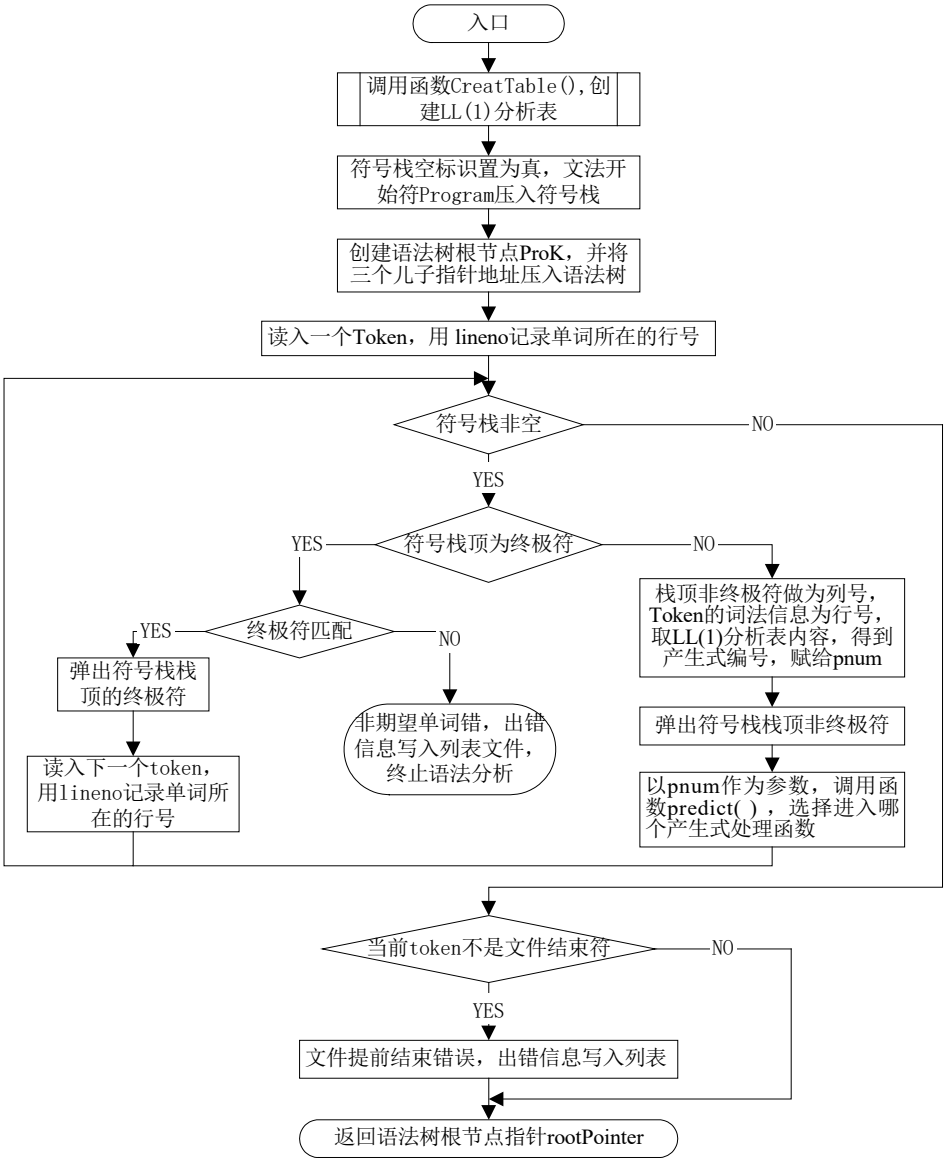


图5.62 LL(1)语法分析主函数parse()的算法框图

（2）函数 CreatLL1Table( )

函数声明： void CreatLL1Table( )

算法说明： 创建 LL(1)分析表。用二维数组表示 LL(1)分析表， 初始化二维数组所有元素为 0， 据给定的 LL(1)文法，对产生式从 1 编号，共 104 个产生式。对于每个产生式，产生式左部的非终极符作为行号，此产生式的 Predict 集中每个元素分别作为列号，数组中由此行号，列号对应的元素赋值为这个产生式的编号。

举例： 对于第 4 个产生式： <DeclarePart> ::= TypeDec VarDec ProcDec
它的 predict 集为：{ **TYPE, VAR**, **FUNCTION**, **BEGIN** }，创建 LL(1)分析表时，这个产生式部分对应的语句为：

```
LL1Table[DeclarePart][TYPE]=        4;
LL1Table[DeclarePart][VAR]=         4;
LL1Table[DeclarePart][FUNCTION]=    4;
LL1Table[DeclarePart][BEGIN]=       4;
```

其他产生式与此类似。

算法框图： 略。

（3）函数 predict ( )

函数声明： void predict(int num)

算法说明： 根据产生式编号选择一个要执行的函数。函数参数为已选择的产生式编号 pnum，每个产生式需要做不同的动作，所以每个产生式对应一个处理函数，函数 process1( )到 process97( ) 分别对应 97 个产生式。若产生式编号为 m，则选择函数 process m ( )。

算法框图： 见图 5.63。

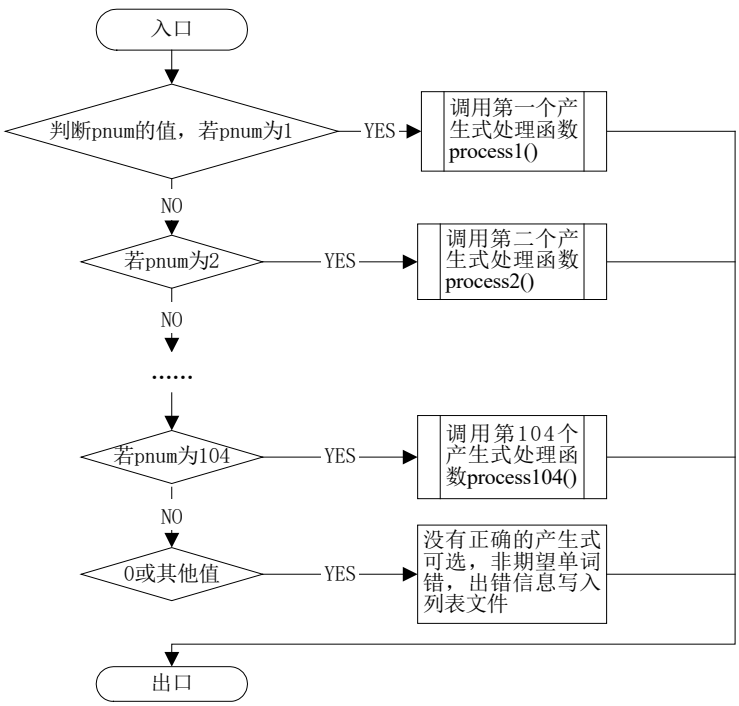


图5.63 选择处理哪个产生式的函数predict()的算法框图

（4）函数 Priosity

函数声明： int Priosity(LexType op)

算法说明： 参数为操作符，类型为单词的词法类型。对于给定的操作符，此函数返回操作符的优先级，返回类型是整数，规定返回值越大所给操作符的优先级越高。

优先级由高到低排序为：

乘法运算符 > 加法运算符 > 关系运算符 > 栈底标识 END

算法框图： 见图 5.64。

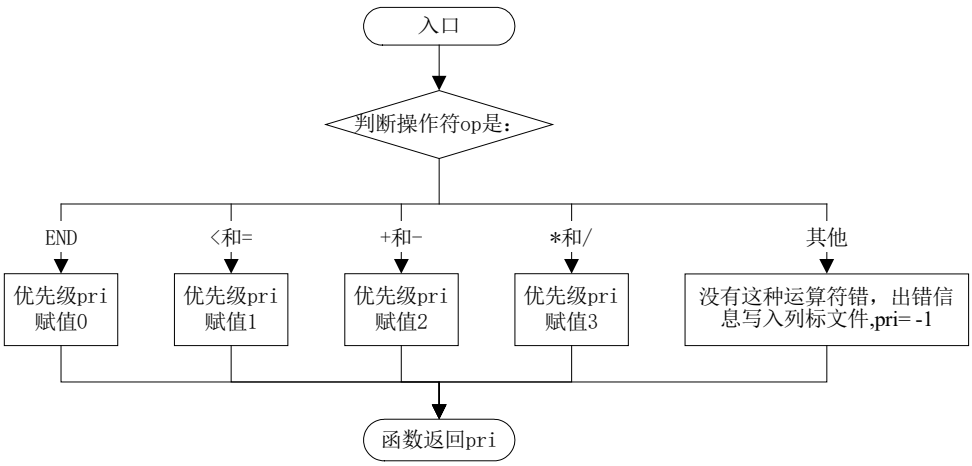


图5.64 确定操作符优先级的函数Priosity()的算法框图

2．各个产生式处理函数：

（1） 函数 process1( )

产生式： <Program> **::=** ProgramHead DeclarePart ProgramBody **.**

函数声明： void process1()

算法说明： 右部非终极符从右至左依次入符号栈，语法树无动作。

算法框图： 略。

（2） 函数 process2( )

产生式： <ProgramHead> **::=** **PROGRAM** ProgramName

函数声明： void process2( )

算法说明： 当栈顶为 ProgramHead ，当前 Token 的词法信息为 PROGRAM 时，选择这个产生式，处理程序头：将产生式右部压入符号栈；生成程序头节点，弹语法树栈，并对弹出元素进行赋值，使得语法树根节点的儿子节点指向程序头节点。

算法框图： 见图 5.65。

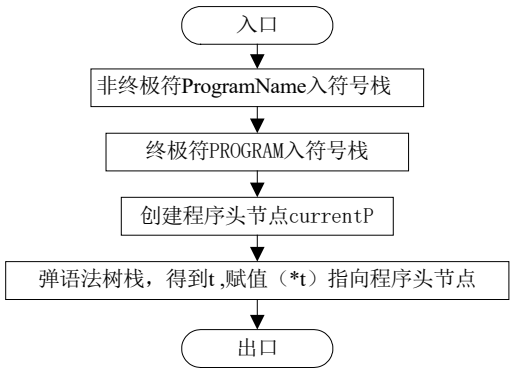


图5.65 产生式2的处理函数process2()的算法框图

（3） 函数 process3( )
产生式： <ProgramName> **::=** **ID**;
函数声明： void process3( )
算法说明： 处理程序名：ID 压入符号栈；并将当前标识符的语义信息（标识符名）写入程序头节点，标识符个数加 1。
算法框图： 略。

（4） 函数 process4( )
产生式： <DeclarePart> **::=** TypeDec VarDec FuncDec
函数声明： void process4( )
算法说明： 处理程序声明：产生式右部非终极符从右至左依次进符号栈，语法树部分没有动作。
算法框图： 略。

（5） 函数 process5( )
产生式： <TypeDec> **::=** **ε**
函数声明： void process5( )
算法说明： 空函数。
算法框图： 略。

（6） 函数 process6( )
产生式： <TypeDec> **::=** TypeDeclaration
函数声明： void process6( )
算法说明： 右部入符号栈。
算法框图： 略。

（7） 函数 process7( )
产生式： <TypeDeclaration> **::=** **TYPE** TypeDecList
函数声明： void process7( )
算法说明： 处理类型声明，产生式右部压入符号栈；语法树部分，生成类型声明标志节点，弹语法树栈，得到指针的地址，令指针指向此声明节点，

使得此节点作为根节点的儿子节点出现。当前类型声明节点的兄弟节点应该指向变量声明标识节点，函数声明节点或语句序列节点，而子节点则应指向具体的声明节点，故将当前节点的兄弟节点和第一个儿子节点压入语法树栈，以待以后处理。

算法框图： 见图 5.66。

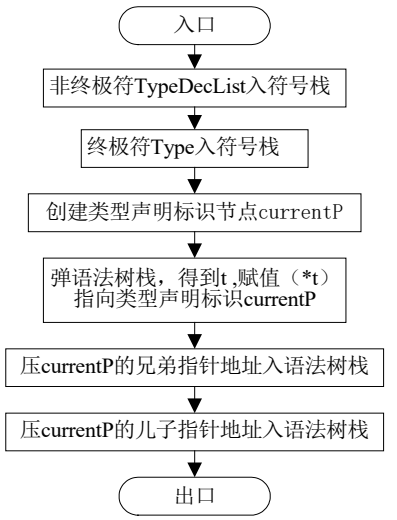


图5.66 产生式7的处理函数process7()的算法框图

（8） 函数 process8( )

产生式： <TypeDecList> ::= TypeId = TypeDef ; TypeDecMore

函数声明： void process8( )

算法说明： 进入具体的类型声明。语法树处理部分，生成一个声明类型节点，不在此添加任何其他信息；弹语法树栈，得到指针的地址，令指针指向此声明类型节点，若是第一个声明节点，则是 Type 类型的子节点指向当前节点，否则，作为上一个类型声明的兄弟节点出现。并将此节点的兄弟节点压入语法树栈，以便处理下一个类型声明。

算法框图： 见图 5.67。

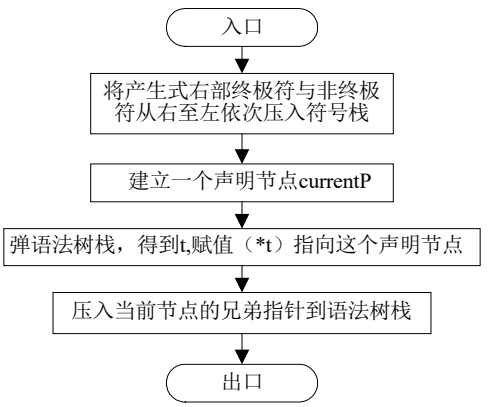


图5.67 产生式8的处理函数process8()的算法框图

（9） 函数 process9 ( )
产生式： <TypeDecMore> ::= ε
函数声明： void process9( )
算法说明： 没有其它的类型声明，这时语法树栈顶存放的是最后一个类型声明节点的兄弟节点，弹出，完成类型部分的语法树节点生成。
算法框图： 略。

（10） 函数 process10( )
产生式： <TypeDecMore> ::= TypeDecList
函数声明： void process10( )
算法说明： 右部入符号栈。
算法框图： 略。

（11） 函数 process11( )
产生式： <TypeId > ::= **ID**
函数声明： void process11( )
算法说明： 处理类型声明节点的类型标识符，把标识符名存入节点中，标识符个数加 1。
算法框图： 见图 5.68。

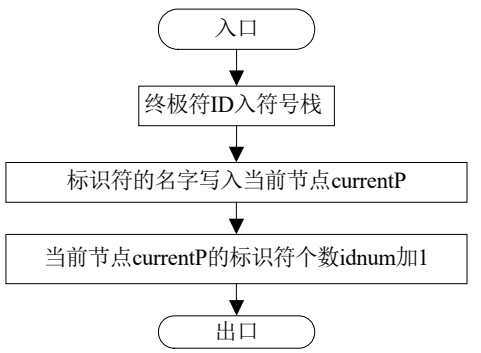


图5.68 产生式11的处理函数process11()的算法框图

（12） 函数 process12()
产生式： <TypeDef> ::= BaseType
函数声明： void process12 ( )
算法说明： 处理声明节点的标识符的类型部分。基本类型可以是整型和字符型，这里用变量 temp 记录节点上填写标识符类型信息的部分的地址，在下面的产生式处理对 temp 里的内容进行赋值，就完成了类型部分的填写。
算法框图： 见图 5.69。

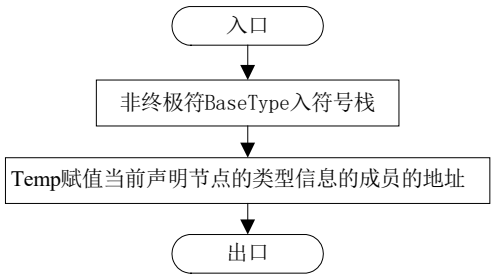


图5.69 产生式12的处理函数process12()的算法框图

（13） 函数 process13( )
产生式： <TypeDef > ::= StructureType
函数声明： void process13( )
算法说明： 右部入符号栈。
算法框图： 略。

（14） 函数 process14( )
产生式： <TypeDef> ::= **ID**
函数声明： void process14( )
算法说明： 声明的类型标识符的类型是用已声明过的类型标识符给出的，在当前节点存储此标识符的名字，节点上标识符的个数加 1。
算法框图： 见图 5.70。

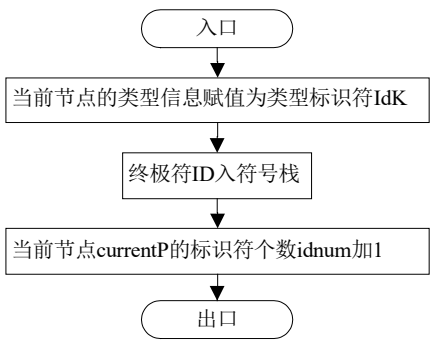


图5.70 产生式14的处理函数process14()的算法框图

（15） 函数 process15( )
产生式： <BaseType> ::= **INTEGER**
函数声明： void process15( )
算法说明： 声明的类型是整型，对 temp 这个地址的内容赋值，将整型信息存入声明节点。
算法框图： 见图 5.71。

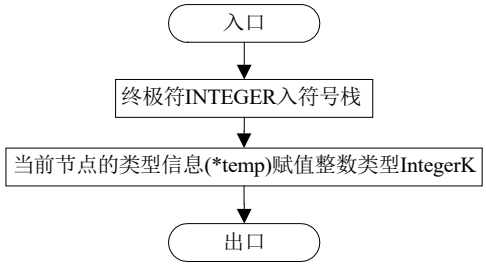


图5.71 产生式15的处理函数process15()的算法框图

（16） 函数 process16( )
产生式： <BaseType> **::=** **CHAR**
函数声明： void process16( )
算法说明： 声明的类型是字符型，对 temp 这个地址的内容赋值，将字符型信息存入声明节点。
算法框图： 见图 5.72。

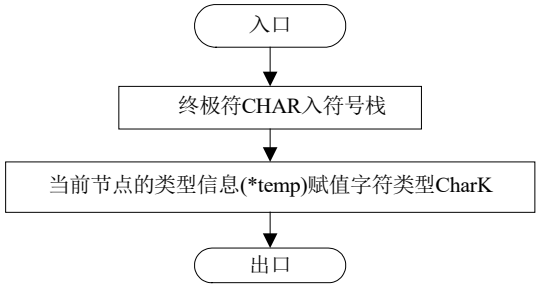


图5.72 产生式15的处理函数process15()的算法框图

（17） 函数 process17( )
产生式： <StructureType> **::=** ArrayType
函数声明： void process17( )
算法说明： 右部入符号栈。
算法框图： 略。

（18） 函数 process18( )
产生式： <StructureType> **::=** RecType
函数声明： void process18( )
算法说明： 右部入符号栈。
算法框图： 略。

（19） 函数 process19( )
产生式： <ArrayType> **::=** **ARRAY [**low..top **] OF** BaseType
函数声明： void process19( )
算法说明： 声明的类型信息赋值为数组类型 ArrayK，并用 temp 记录填写数组的

基类型部分的地址，以后对 temp 地址的内容赋值，就完成了对数组基类型信息的填写。

算法框图： 见图 5.73。

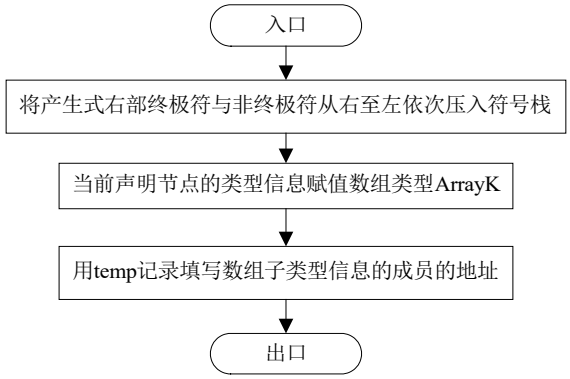


图5.73 产生式16的处理函数process16()的算法框图

（20） 函数 process20( )

产生式： <Low> ::= **INTC**

函数声明： void process20( )

算法说明： 右部终极符入符号栈，并将整数的值写入数组类型声明节点的下界。

算法框图： 略。

（21） 函数 process21( )

产生式： <Top> ::= **INTC**

函数声明： void process21( )

算法说明： 右部终极符入符号栈，并将整数的值写入数组类型声明节点的上界。

算法框图： 略。

（22） 函数 process22( )

产生式： <RecType > ::= **RECORD** FieldDecList **END**

函数声明： void process22( )

算法说明： 声明的类型部分赋值为记录类型 RecordK，用变量 saveP 保存当前指向记录类型声明节点的指针，以便处理完记录的各个域以后能够回到记录类型节点处理没有完成的信息，并压入指向记录的第一个域的指针进语法树栈，在后面对指针赋值。

算法框图： 见图 5.74。

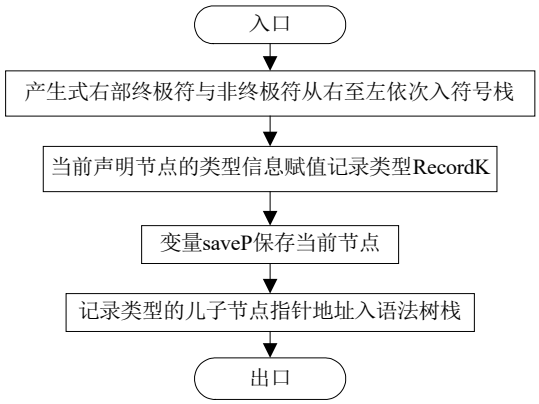


图5.74 产生式22的处理函数process22()的算法框图

（23） 函数 process23( )

产生式： <FieldDecList > ::= BaseType IdList ; FieldDecMore

函数声明： void process23( )

算法说明： 生成记录类型的一个域，节点为声明类型的节点，不添加任何信息；类型属于基类型，用 temp 记录填写类型信息的成员地址，以待以后填写是整数类型还是字符类型。弹语法树栈，令指针指向这个节点。若是第一个，则是 record 类型的子节点指向当前节点；否则，是上一个记录域声明的兄弟节点。最后，压入指向记录类型下一个域的指针，以处理多个域。

算法框图： 见图 5.75。

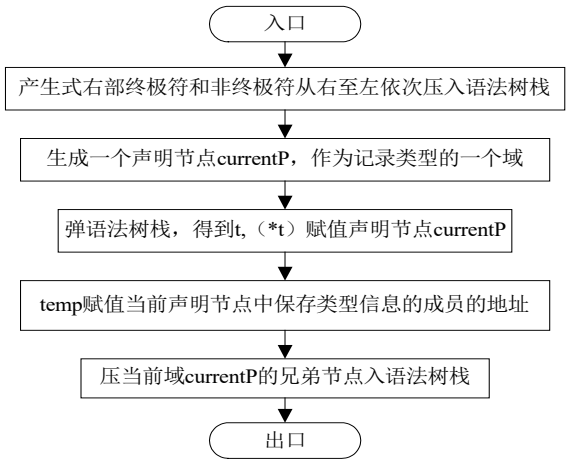


图5.75 产生式23的处理函数process23()的算法框图

（24） 函数 process24( )

产生式： <FieldDecList> ::= ArrayType IdList ; FieldDecMore

函数声明： void process24( )

算法说明： 生成记录类型的一个域，节点为声明节点，类型是数组类型，不添加

任何信息，弹语法树栈，令指针指向这个节点，若是第一个，则是 record 类型的子结点指向当前结点，否则，是上一个记录域声明的兄弟结点，最后，压入指向记录类型下一个域的指针，以处理多个域。

算法框图： 见图 5.76。

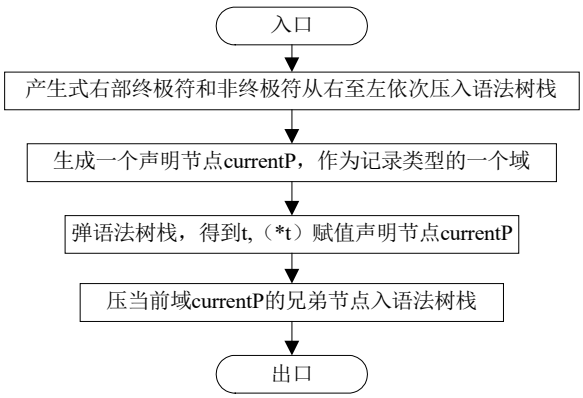


图5.76 产生式24的处理函数process24()的算法框图

（25） 函数 process25( )

产生式： <FieldDecMore > ::= ε

函数声明： void process25( )

算法说明： 没有记录类型的下一个域了，弹出栈顶保存的最后一个域的兄弟节点，表示记录的域全部处理完；并利用 saveP 恢复当前记录类型节点的指针。

算法框图： 见图 5.77。

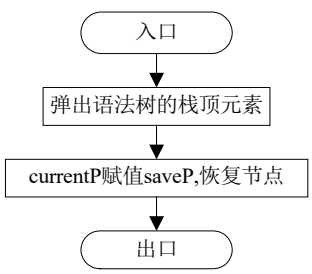


图5.77 产生式25的处理函数process25()的算法框图

（26） 函数 process26( )

产生式： <FieldDecMore> ::= FieldDecList

函数声明： void process26( )

算法说明： 右部非终极符入符号栈，语法树没有任何动作。

算法框图： 略。

（27） 函数 process27( )

产生式： <IdList > ::= id IdMore

函数声明： void process27( )
算法说明： 右部入符号栈，并将当前标识符的名字，存入节点中，标识符个数加1。
算法框图： 略。

（28） 函数 process28( )
产生式： **<IdMore > ::=ε**
函数声明： void process28( )
算法说明： 空函数
算法框图： 略。

（29） 函数 process29( )
产生式： **<IdMore > ::=** , IdList
函数声明： void process29( )
算法说明： 右部入符号栈。
算法框图： 略。

（30） 函数 process30( )
产生式： **<VarDec> ::= ε**
函数声明： void process30( )
算法说明： 空函数，不做任何动作。选择这个产生式，表示程序没有变量声明。
算法框图： 略。

（31） 函数 process31( )
产生式： **<VarDec > ::=** VarDeclaration
函数声明： void process31( )
算法说明： 右部入符号栈。
算法框图： 略。

（32） 函数 process32( )
产生式： **<VarDeclaration> ::= VAR** VarDecList
函数声明： void process32( )
算法说明： 处理变量声明，产生式右部压入符号栈；语法树部分，生成变量声明标志节点，弹语法树栈，得到指针的地址，令指针指向此声明节点，使得此节点作为根节点的儿子节点或者类型标识节点的兄弟节点出现。当前变量声明节点的兄弟节点应该指向函数声明节点或语句序列节点，而子节点则应指向具体的声明节点，故将当前节点的兄弟节点和第一个儿子节点压入语法树栈，以待以后处理。
算法框图： 见图 5.78。

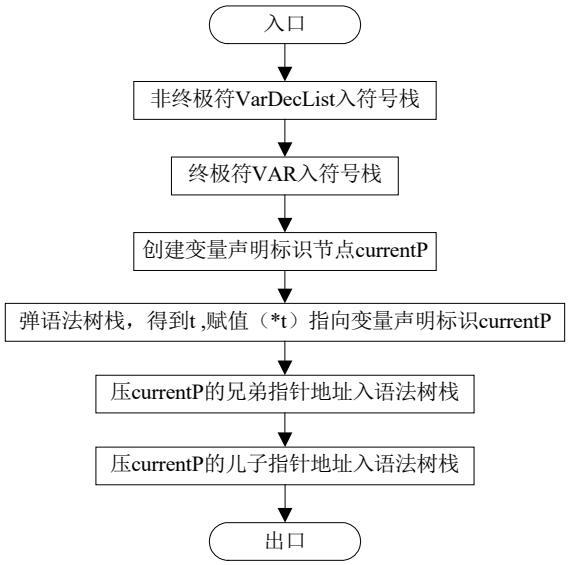


图5.78 产生式32的处理函数process32()的算法框图

（33） 函数 process33( )

产生式： <VarDecList >::= TypeDef VarIdList ; VarDecMore

函数声明： void process33( )

算法说明： 进入具体的变量声明。语法树处理部分，生成一个声明类型节点，不在此添加任何其他信息；弹语法树栈，得到指针的地址，令指针指向此声明类型节点，若是第一个声明节点，则是 VarK 类型的子节点指向当前节点，否则，作为上一个变量声明的兄弟节点出现。并将此节点的兄弟节点压入语法树栈，以便处理下一个变量声明。

算法框图： 见图 5.79。

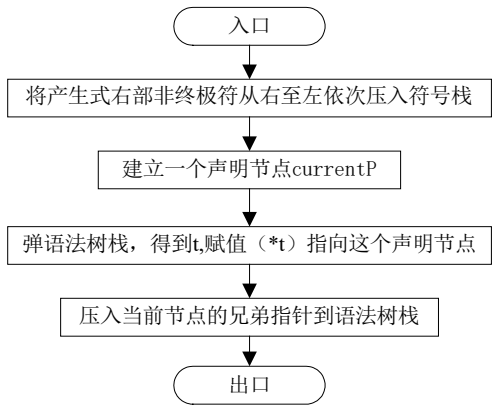


图5.79 产生式33的处理函数process33()的算法框图

（34） 函数 process34( )

产生式： <VarDecMore > ::= ε

函数声明： void process34( )
算法说明： 后面没有变量声明了，弹出栈中指向下一个变量声明的指针，变量声明部分处理结束。
算法框图： 略。

（35） 函数 process35( )
产生式： <VarDecMore > **::=** VarDecList
函数声明： void process35( )
算法说明： 右部入符号栈。
算法框图： 略。

（36） 函数 process36( )
产生式： <VarIdList > **::= ID** VarIdMore
函数声明： void process36( )
算法说明： 处理变量声明的若干个标识符，将当前标识符的名字，存入节点中，标识符个数加 1。
算法框图： 见图 5.80。

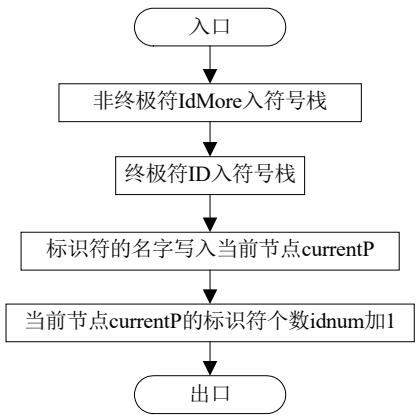


图5.80 产生式36的处理函数process36()的算法框图

（37） 函数 process37( )
产生式： <VarIdMore > **::= ε**
函数声明： void process37( )
算法说明： 空函数。
算法框图： 略。

（38） 函数 process38( )
产生式： <VarIdMore > **::= ,** VarIdList
函数声明： void process38( )
算法说明： 非终极符 VarIdList 和终极符逗号入符号栈。
算法框图： 略。

（39） 函数 process39( )
产生式： <ProcDec> ::= ε
函数声明： void process39( )
算法说明： 空函数。
算法框图： 略。

（40） 函数 process40( )
产生式： <ProcDec> ::= ProcDeclaration
函数声明： void process40( )
算法说明： 右部非终极符入符号栈。
算法框图： 略。

（41） 函数 process41( )
产生式： <ProcDeclaration> ::= **PROCEDURE**
ProcName **(** ParamList **) :** BaseType **;**
ProcDecPart
ProcBody ProcDecMore
函数声明： void process41( )
算法说明： 处理过程声明，产生式右部压入符号栈；语法树部分，生成过程头结点 ProcK，弹语法树栈，得到指针的地址，令指针指向此声明节点，使得此节点作为根节点的儿子节点或者类型标识节点，或者变量标识节点的兄弟节点出现。当前过程声明节点的兄弟节点指向下一个过程声明节点或语句序列节点，第三个儿子节点指向过程体部分；第二个子节点指向过程中的声明部分，若没有声明，这个指针为空；第一个子节点指向函数的形参部分。故将当前节点的兄弟节点和三个儿子节点压入语法树栈，以待以后处理。
算法框图： 见图 5.81。

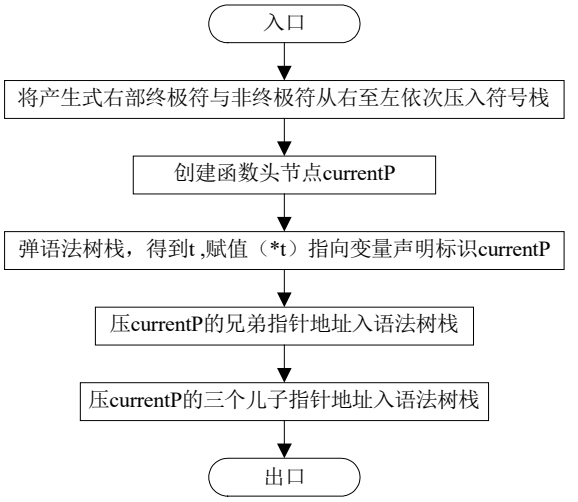


图5. 81 产生式41的处理函数process41( )的算法框图

（42） 函数 process42( )

产生式： <ProcDecMore> ::= ε
函数声明： void process42( )
算法说明： 空函数。
算法框图： 略。

（43） 函数 process43( )
产生式： <ProcDecMore> ::= ProcDeclaration
函数声明： void process43( )
算法说明： 过程声明，产生式右部进符号栈，语法树无动作。
算法框图： 略。

（44） 函数 process44( )
产生式： <ProcName> ::= id
函数声明： void process44( )
算法说明： 右部终极符入符号栈，并将函数名字写入节点中，标识符个数加 1。
算法框图： 略。

（45） 函数 process45( )
产生式： <ParamList> ::= ε
函数声明： void process45( )
算法说明： 函数形参部分为空，从语法树栈中弹出指向形参部分的指针地址。
算法框图： 略。

（46） 函数 process46 ( )
产生式： <ParamList> ::= ParamDecList
函数声明： void process46( )
算法说明： 右部非终极符入符号栈，语法树无动作。
算法框图： 略。

（47） 函数 process47( )
产生式： <ParamDecList> ::= Param ParamMore
函数声明： void process47( )
算法说明： 右部非终极符入符号栈，语法树无动作。
算法框图： 略。

（48） 函数 process48( )
产生式： <ParamMore>::= ε
函数声明： void process48( )
算法说明： 后面没有别的参数，说明函数的形参部分处理完毕，弹出最后一个形参声明的兄弟节点；并利用 saveFuncP 恢复当前节点为函数声明节点，并把 temp 赋值为记录函数返回类型信息的成员的地址，以后对（*temp）赋值，就把函数返回类型信息写入了函数节点。
算法框图： 见图 5.82。

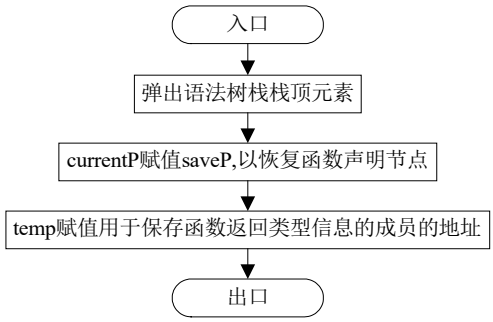


图5.82 产生式48的处理函数process48()的算法框图

（49） 函数 process49( )
产生式： <ParamMore> ::= ; ParamDecList
函数声明： void process49( )
算法说明： 右部终极符与非终极符入符号栈。
算法框图： 略。

（50） 函数 process50( )
产生式： <Param> ::= TypeDef FormList
函数声明： void process50( )
算法说明： 进入形参声明。该产生式说明参数为值参。语法树处理部分，生成一个声明类型节点，在节点的参数类型部分写上形参，弹语法树栈，得到指针的地址，令指针指向此声明类型节点，若是第一个声明节点，则是 ProcK 类型的第一个子结点指向当前结点；否则，作为上一个形参声明的兄弟结点出现，并将此节点的兄弟节点压入语法树栈，以便处理下一个形参声明。
算法框图： 见图 5.83。

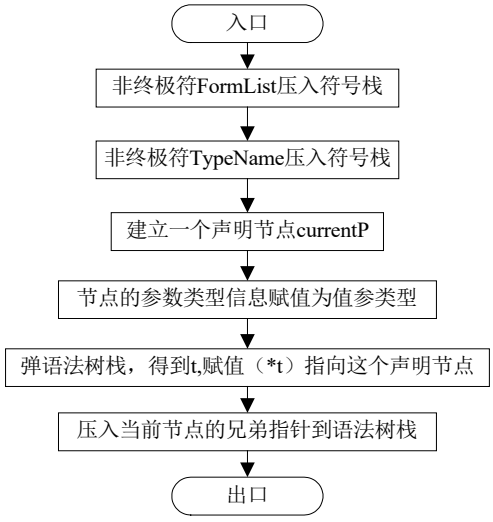


图5.83 产生式50的处理函数process50()的算法框图

（51） 函数 process51( )
产生式： <Param> **::=** **VAR** TypeDef FormList
函数声明： void process51( )
算法说明： 进入形参声明。这个产生式说明参数为变参。语法树处理部分，生成一个声明类型节点，在节点的参数类型部分写上变参，弹语法树栈，得到指针的地址，令指针指向此声明类型节点，若是第一个声明节点，则是 FuncK 类型的第一个子结点指向当前结点，否则，作为上一个形参声明的兄弟结点出现。并将此节点的兄弟节点压入语法树栈，以便处理下一个形参声明。
算法框图： 见图 5.84。

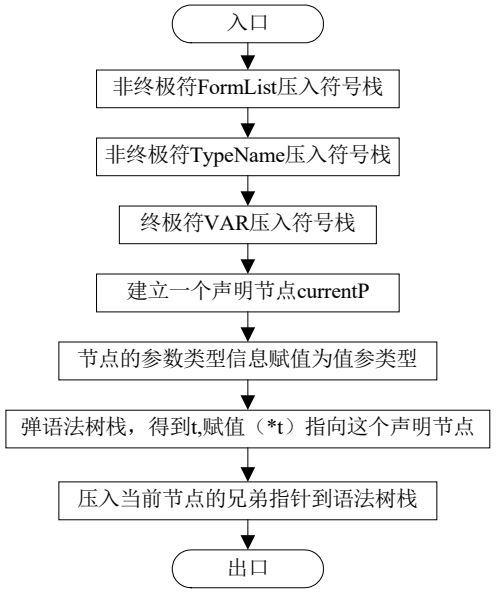


图5.84 产生式51的处理函数process51()的算法框图

（52） 函数 process52( )
产生式： <FormList> **::=** **ID** FidMore
函数声明： void process52( )
算法说明： 右部进符号栈，并将标识符的名字写入当前语法树节点中，节点中标识符个数 idnum 加 1。
算法框图： 略。

（53） 函数 process53( )
产生式： <FidMore> **::=** **ε**
函数声明： void process53( )
算法说明： 空函数。
算法框图： 略。

（54） 函数 process54( )
产生式： <FidMore> **::=** **,** FormList

函数声明： void process54( )
算法说明： 右部终极符与非终极符入符号栈。
算法框图： 略。

（55） 函数 process55( )
产生式： <ProcDecPart> **::=** DeclarePart
函数声明： void process55( )
算法说明： 右部非终极符入符号栈。
算法框图： 略。

（56） 函数 process56( )
产生式： <ProcBody> **::=** ProgramBody
函数声明： void process56( )
算法说明： 右部非终极符入符号栈。
算法框图： 略。

（57） 函数 process57( )
产生式： <ProgramBody> **::=** **BEGIN** StmList **END**
函数声明： void process57( )
算法说明： 处理语句序列，产生式右部压入符号栈；语法树部分，先弹语法树栈，删除栈顶保存的多余指针，以保证一定是 child[2]指向语句序列节点；生成语句序列标志节点，此节点下的所有节点都是语句节点。弹语法树栈，令指针指向此标志节点，使得节点作为根节点或函数头节点的最后一个儿子节点出现。将当前节点的第一个儿子节点压入语法树栈，以待以后处理。
算法框图： 见图 5.85。

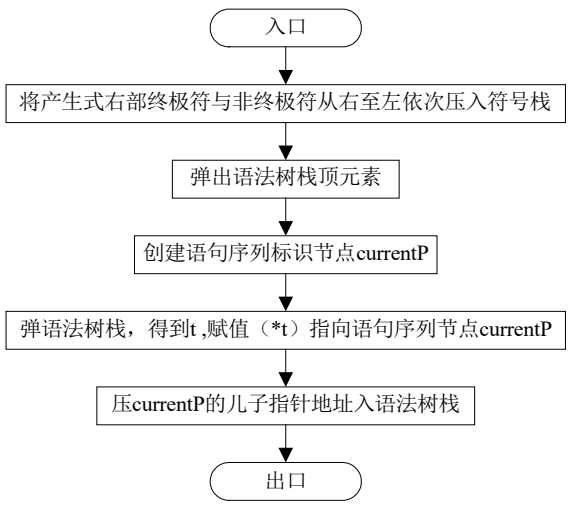


图5.85 产生式52的处理函数process52()的算法框图

（58） 函数 process58( )
产生式： <StmList> ::= Stm StmMore
函数声明： void process58( )
算法说明： 右部非终极符从右至左依次入符号栈，语法树无动作。
算法框图： 略。

（59） 函数 process59( )
产生式： <StmMore> ::= ε
函数声明： void process59( )
算法说明： 后面没有其他语句了，最后一条语句没有兄弟节点，故将栈顶存放的最后一条语句的兄弟节点指针的地址从语法树栈中弹出来，表示语句序列处理完毕。
算法框图： 略。

（60） 函数 process60( )
产生式： <StmMore> ::= ; StmList
函数声明： void process60( )
算法说明： 右部终极符与非终极符从右至左入符号栈，语法树无动作。
算法框图： 略。

（61） 函数 process61( )
产生式： <Stm> ::= ConditionalStm
函数声明： void process61( )
算法说明： 创建一个语句类型节点，具体类型值设为条件语句 IfK, 从语法树栈中弹出指针的地址，令相应指针指向此条件语句节点，并压入 if 语句的兄弟节点指针地址以处理下一条语句。
算法框图： 见图 5.86。

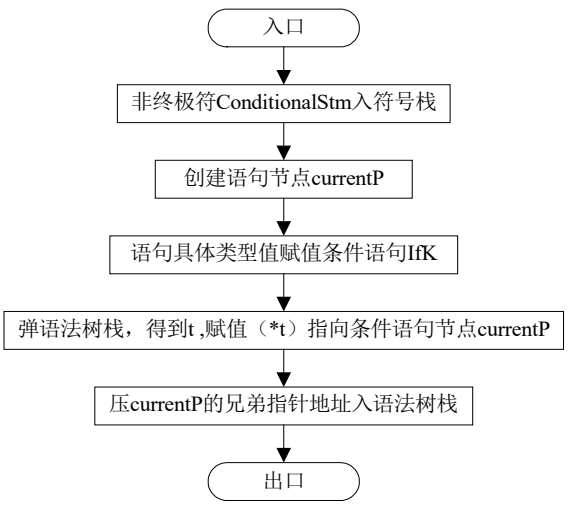


图5.86 产生式53的处理函数process53()的算法框图

（62） 函数 process62( )

产生式： <Stm> ::= LoopStm

函数声明： void process62( )

算法说明： 创建一个语句类型节点，具体类型值设为循环语句 WhileK, 从语法树栈中弹出指针的地址，令相应指针指向此循环语句节点，并压入 while 语句的兄弟节点指针地址以处理下一条语句。

算法框图： 见图 5.87。

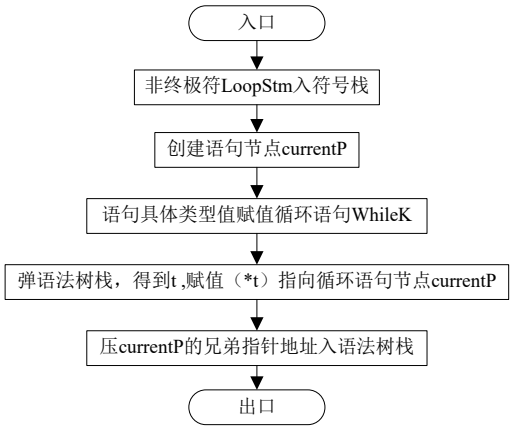


图5.87 产生式62的处理函数process62()的算法框图

（63） 函数 process63( )

产生式： <Stm> ::= InputStm

函数声明： void process63( )

算法说明： 创建一个语句类型节点，具体类型值设为输入语句 ReadK, 从语法树栈中弹出指针的地址，令相应指针指向此输入语句节点, 并压入 read 语句的兄弟节点指针地址以处理下一条语句。

算法框图： 见图 5.88。

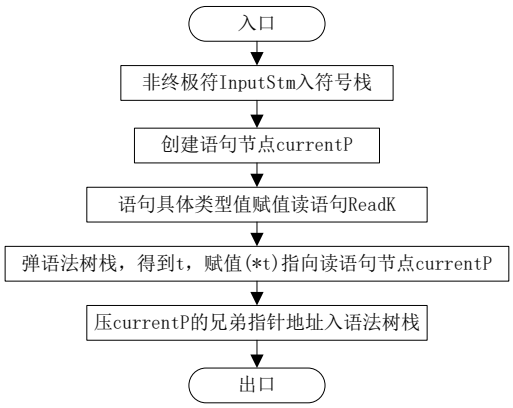


图5.88 产生式63的处理函数process63( )的算法框图

（64） 函数 process64( )

产生式： <Stm> ::= OutputStm
函数声明： void process64( )
算法说明： 创建一个语句类型节点，具体类型值设为输出语句 WriteK, 从语法树栈中弹出指针的地址，令相应指针指向此输出语句节点, 并压入 write 语句的兄弟节点指针地址以处理下一条语句。
算法框图： 见图 5.89。

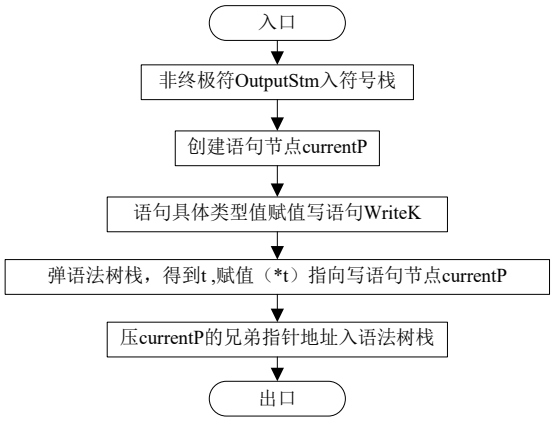


图5.89 产生式64的处理函数process64()的算法框图

（65） 函数 process65( )
产生式： <Stm> ::= ReturnStm
函数声明： void process64( )
算法说明： 创建一个语句类型节点，具体类型值设为返回语句 ReturnK, 从语法树栈中弹出指针的地址，令相应指针指向此返回语句节点, 并压入 return 语句的兄弟节点指针地址以处理下一条语句。
算法框图： 见图 5.90。

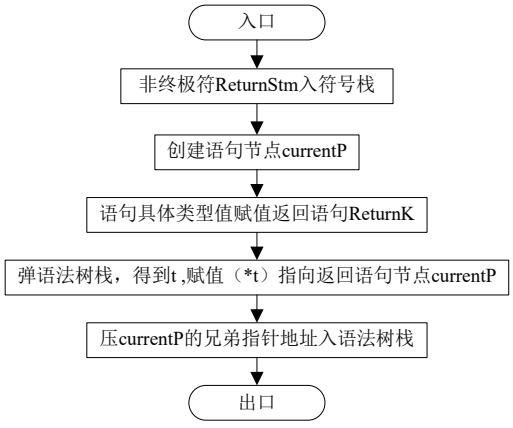


图5.90 产生式65的处理函数process65()的算法框图

（66） 函数 process66( )
产生式： <Stm> ::= id AssCall
函数声明： void process66( )

算法说明： 创建一个语句类型节点，由于赋值语句和过程调用语句都以标识符开始，故现在不能确定此语句的具体类型；创建一个变量表达式节点，记录赋值左部，并使其成为此语句节点的第一个儿子节点； 从语法树栈中弹出指针的地址，令相应指针指向此返回语句节点, 并压入此语句的兄弟节点指针地址以处理下一条语句。

算法框图： 见图 5.91。

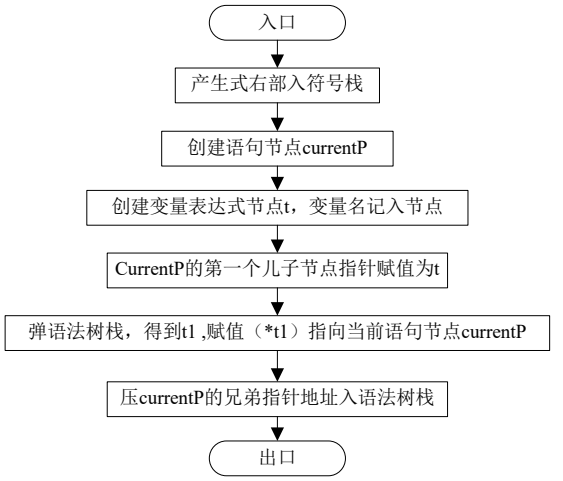


图5.91 产生式66的处理函数process66()的算法框图

（67） 函数 process67( )

产生式： <AssCall> **::=** AssignmentRest

函数声明： void process67( )

算法说明： 产生式右部非终极符入符号栈；语句的具体类型值确定为赋值语句，将 AssignK 写入当前处理节点中。

算法框图： 略。

（68） 函数 process68( )

产生式： <AssCall> **::=** CallStmRest

函数声明： void process68( )

算法说明： 产生式右部非终极符入符号栈；变量的具体类型确定为标志符变量；语句的具体类型值确定为过程调用语句，将 CallK 写入当前处理节点中。

算法框图： 略。

（69） 函数 process69( )

产生式： <AssignmentRest> **::=** VariMore **:=** Exp

函数声明： void process69( )

算法说明： 赋值语句节点的赋值右部子节点指针压栈，改变当前节点指针为赋值右部子节点。压入特殊栈底标志。

算法框图： 见图 5.92。

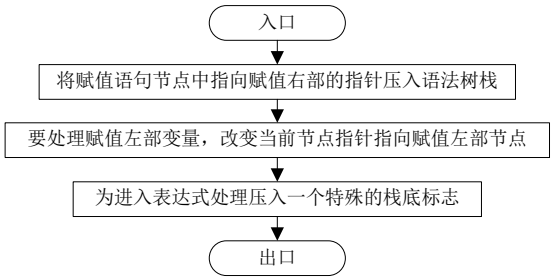


图5.92 产生式69的处理函数process69()的算法框图

（70） 函数 process70( )

产生式： <ConditionalStm> **::= IF** RelExp **THEN** StmList **ELSE** StmList **FI**

函数声明： void process70( )

算法说明： if 语句有三个部分，条件表达式部分，then 部分 和 else 部分，分别用三个儿子节点指向相应的部分。故将三个儿子节点按照逆序入语法树栈，以待以后赋值。

算法框图： 见图 5.93。

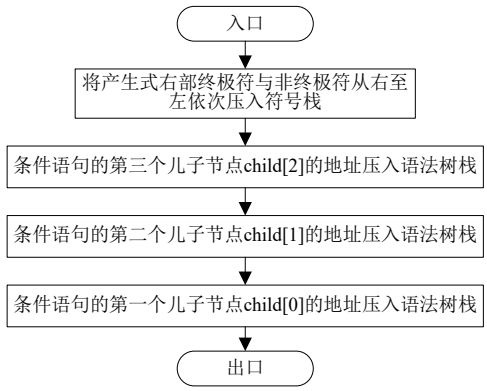


图5.93 产生式70的处理函数process70()的算法框图

（71） 函数 process71( )

产生式： <LoopStm> **::= WHILE** RelExp **DO** StmList **ENDWH**

函数声明： void process71( )

算法说明： while 语句有两部分，条件表达式和循环体。分别用两个儿子节点指针指向。将两个儿子节点指针地址按照处理次序的逆序压入语法树栈中。

算法框图： 见图 5.94。

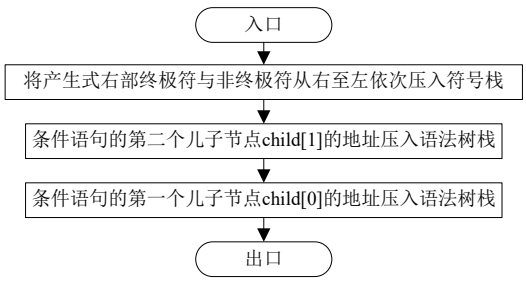


图5.94 产生式71的处理函数process71()的算法框图

（72） 函数 process72( )
产生式： <InputStm> ::= **READ (** InVar **)**
函数声明： void process72( )
算法说明： 右部终极符与非终极符从右至左入符号栈，语法树无动作。
算法框图： 略。

（73） 函数 process73( )
产生式： <InVar> ::= **ID**
函数声明： void process73( )
算法说明： 将标识符名字存入输入语句节点中，作为输入变量名，此节点中的标识符个数加 1。
算法框图： 略。

（74） 函数 process74( )
产生式： <OutputStm> ::= **WRITE(** Exp **)**
函数声明： void process74( )
算法说明： 输出语句节点的第一个儿子节点应指向表达式子树的根节点。故先将输出语句的第一个儿子节点指针地址压语法树栈。另外，要进入表达式处理，应该先初始化表达式处理需要的操作符栈，压入一个特殊的操作符，令它的优先级最低。操作符栈和操作数栈中存放的都是指向树节点指针，以完成表达式子树节点的联接。这里先生成一个以“END”作为内容的操作符节点，再将其指针压入操作符栈，实现初始化功能。
算法框图： 见图 5.95。

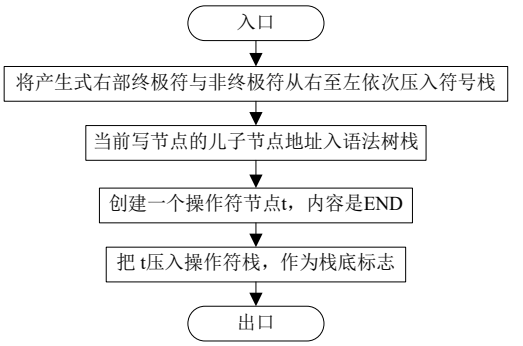


图5.95 产生式74的处理函数process74()的算法框图

（75） 函数 process75( )
产生式： <ReturnStm> ::= **RETURN**
函数声明： void process75( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（76） 函数 process76( )

产生式： <CallStmRest> ::= (ActParamList)
函数声明： void process76( )
算法说明： 处理函数调用语句，首先压入函数调用语句的第一个儿子节点。
算法框图： 略。

（77） 函数 process77( )
产生式： <ActParamList> ::= ε
函数声明： void process77( )
算法说明： 函数调用的实参部分为空，即不带参数的函数调用。将指向实参部分的指针地址从语法树栈中弹出来。
算法框图： 略。

（78） 函数 process78( )
产生式： <ActParamList> ::= Exp ActParamMore
函数声明： void process78( )
算法说明： 函数调用语句的实参是表达式。将要进入表达式处理，先初始化操作符栈，放入一个特殊的优先级最低的操作符。
算法框图： 见图 5.96。

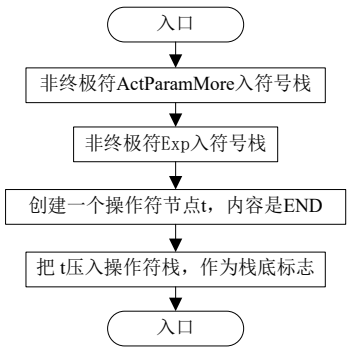


图5.96 产生式78的处理函数process78()的算法框图

（79） 函数 process79( )
产生式： <ActParamMore> ::= ε
函数声明： void process79( )
算法说明： 没有其他的实参，函数调用语句处理结束。空函数。
算法框图： 略。

（80） 函数 process80( )
产生式： <ActParamMore> ::= , ActParamList
函数声明： void process80( )
算法说明： 说明还有别的实参，产生式右部入符号栈；压当前实参节点的兄弟节点入语法树栈。
算法框图： 略。

（81） 函数 process81( )

产生式： <RelExp> ::= Exp OtherRelE

函数声明： void process81( )

算法说明： 处理条件表达式，在 if 和 while 语句中将用到条件表达式，在这里对表达式的操作符栈进行初始化，压入一个优先级最低的操作符类型节点的指针。另外，将变量 getExpResult 赋值为假，一般表达式，在遇到 process84()时，表达式处理结束，弹出表达式的操作数栈和操作符栈，得到表达式部分的树结构，并连入语句中；而对于关系表达式，关系运算符左部的表达式处理结束，整个表达式并未处理完，不弹栈，处理完右部的表达式时，才结束。

算法框图： 见图 5.97。

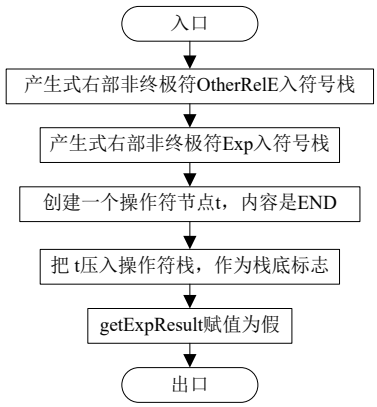


图5.97 产生式81的处理函数process81()的算法框图

（82） 函数 process82( )

产生式： <OtherRelE> ::= CmpOp Exp

函数声明： void process82( )

算法说明： 建立一个操作符类型表达式节点，记录这个关系运算符的内容。比较栈顶操作符和这个操作符的优先级，若栈顶操作符的优先级高于或者等于当前运算符，从操作符栈弹出一个操作符的指针 t，从操作数栈弹出两个操作数节点指针，分别作为 t 的两个儿子节点，再将这个操作符 t 的节点指针压入操作数栈，作为下一个操作符的运算分量，将当前操作符指针压入操作符栈。否则，直接将当前指针压入操作符栈。另外，选择了这个产生式，说明作为关系运算符左分量的表达式 Exp，已经处理完，要进入右分量的处理，作为右分量的 Exp 结束后，整个关系表达式也结束，故设置在函数 process84()中结束，得到表达式结果标识 getExpResult 为真。

算法框图： 见图 5.98。

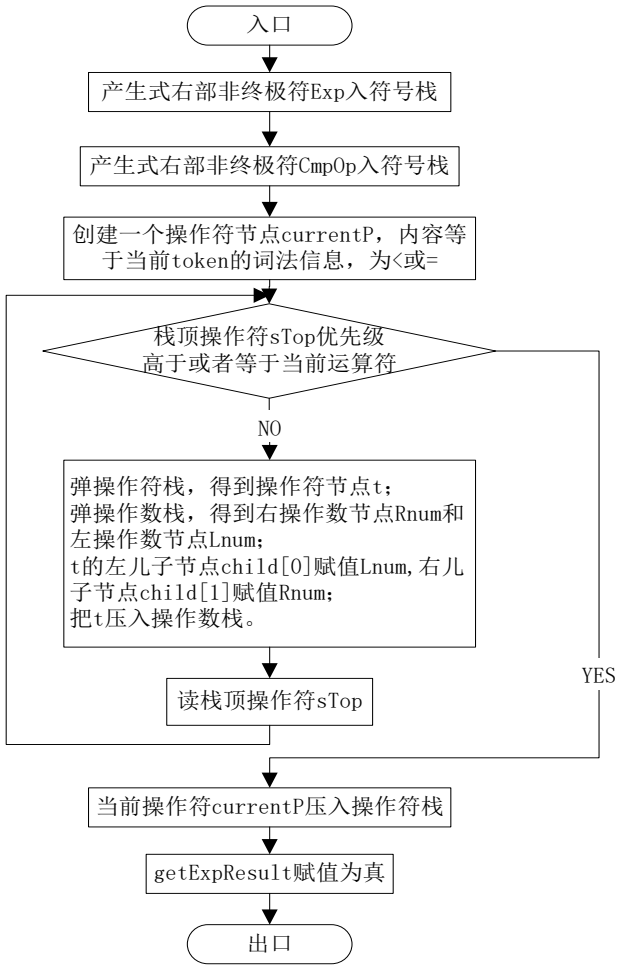


图5.98 产生式82的处理函数process82()的算法框图

（83） 函数 process83( )
产生式： <Exp> ::= Term OtherTerm
函数声明： void process83( )
算法说明： 右部非终极符入符号栈。
算法框图： 略。

（84） 函数 process84( )
产生式： <OtherTerm> ::= ε
函数声明： void process84( )
算法说明： 由于 write, return 等语句以右括号）作为参数表达式的结束，而且表达式的因子又可以是（表达式），所以遇到右括号，需要分情况处理：

如果当前所读单词为右括号 ），而且 expflag 不为 0，说明前面有与之配对的左括号（ ，这时的右括号是表达式的一部分，循环做：弹出一个操作符栈内容，弹出两个操作数栈内容，对操作符节点的两个

儿子节点赋值操作数节点指针，将他们作为左右运算分量，并把这个操作符指针压入操作数栈，作为下一个操作符的运算分量，如此循环，直到操作符栈顶为（，弹出（ ,并将 expflag 减 1，说明处理完表达式中的一对括号，结束。

否则，当前所读单词标识了 exp 的处理结束，判断：如果 getExpResult 为真或者 getExpResult2 为真，则代表当前表达式处理结束，循环做：弹出一个操作符栈内容，弹出两个操作数栈内容，连接操作符的左右分量，并把结果指针压入操作数栈。直到遇到操作符栈底标识 END，弹出栈底标识，表达式子树生成完毕，根节点在操作数栈中，弹出来，并从语法树栈中弹出应指向表达式子树根节点的指针地址，赋值指针指向表达式根节点。

其中，getExpResult2 专用于数组成员变量，当数组成员表达式处理结束时，需要将其与数组变量节点相连，数组成员表达式的根结点作为数组成员变量节点的儿子节点。

算法框图： 见图 5.99。

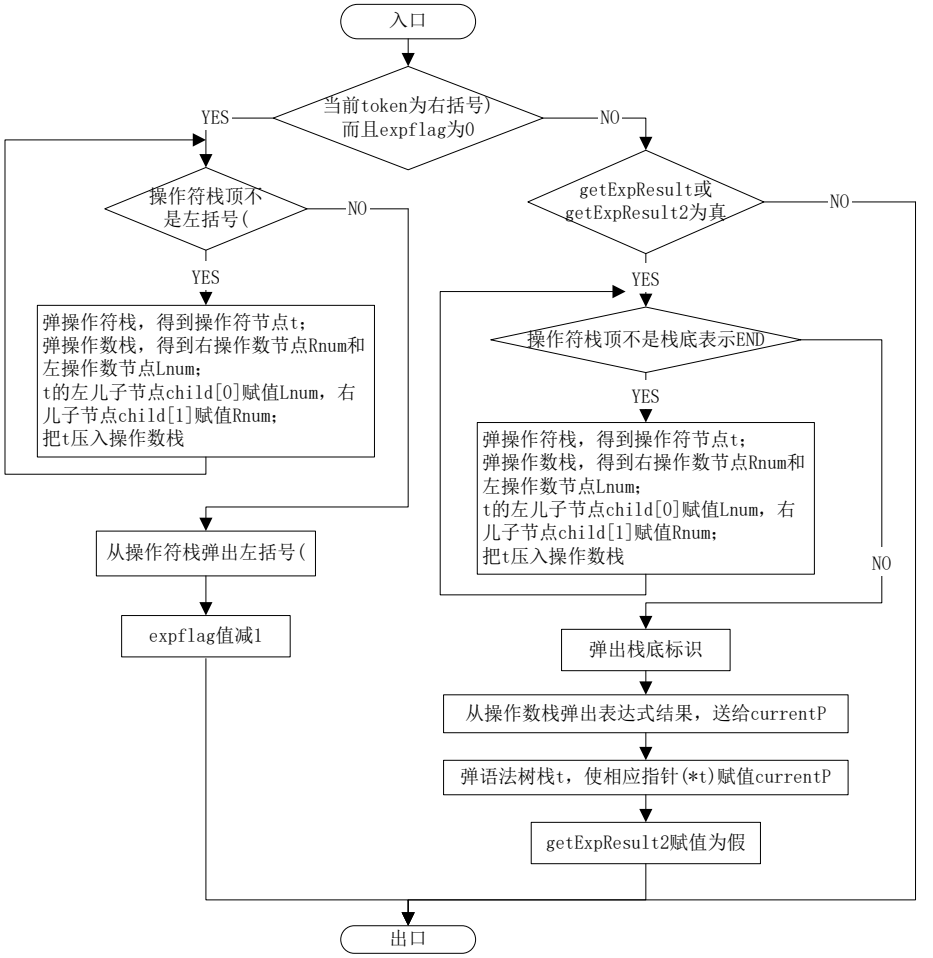


图5.99 产生式84的处理函数process84()的算法框图

（85） 函数 process85( )

产生式： <OtherTerm> ::= AddOp Exp

函数声明： void process85( )

算法说明： 遇到加法运算符，建立一个表达式节点，具体类型是操作符类型，记录这个加法运算符的内容。看当前操作符栈顶内容，若栈顶操作符的优先级高于或等于当前运算符，从操作符栈弹出一个操作符的指针 t，从操作数栈弹出两个操作数节点指针，分别作为 t 的两个儿子节点，再将这个操作符 t 的节点指针压入操作数栈，作为下一个操作符的运算分量，将当前操作符指针压入操作符栈。否则，直接将当前指针压入操作符栈。

算法框图： 见图 5.100。

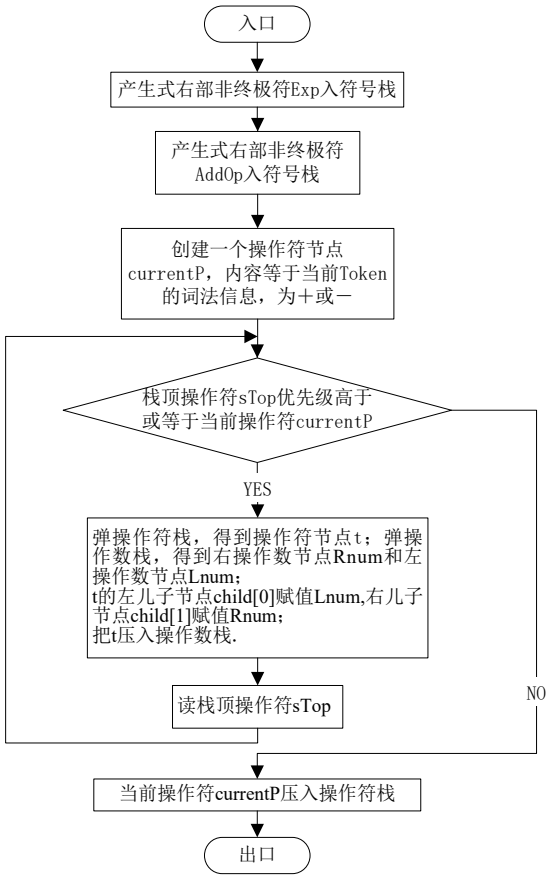


图5.100 产生式85的处理函数process85()的算法框图

（86） 函数 process86( )

产生式： <Term> ::= Factor OtherFactor

函数声明： void process86( )

算法说明： 右部非终极符入符号栈。

算法框图： 略。

（87） 函数 process87( )
产生式： <OtherFactor> ::= ε
函数声明： void process87( )
算法说明： 空函数。
算法框图： 略。

（88） 函数 process88( )
产生式： <OtherFactor> ::= MultOp Term
函数声明： void process88( )
算法说明： 遇到乘法运算符，建立一个表达式节点，具体类型是操作符类型，记录这个乘法运算符的内容。看当前操作符栈顶内容，若是*, / 说明栈顶操作符的优先级等于当前运算符，从操作符栈弹出一个操作符的指针 t，从操作数栈弹出两个操作数节点指针，分别作为 t 的两个儿子节点，再将这个操作符 t 的节点指针压入操作数栈，作为下一个操作符的运算分量，将当前操作符指针压入操作符栈。保证表达式的计算属于左结合。否则，直接将当前指针压入操作符栈。
算法框图： 见图 5.101。

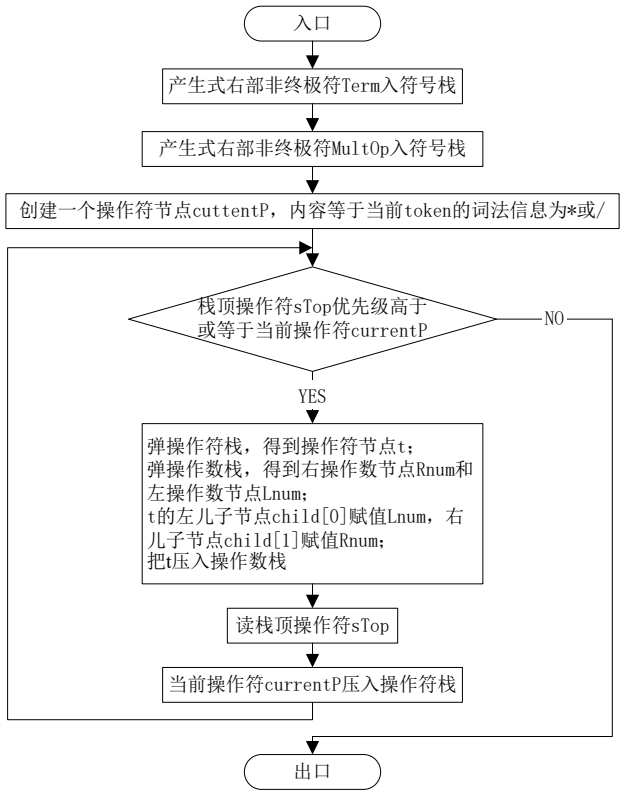


图5.101 产生式88的处理函数process88()的算法框图

（89） 函数 process89( )

产生式： <Factor> **::=** **(** Exp **)**
函数声明： void process89( )
算法说明： 表达式的因子为带括号的表达式，当前单词为左括号，左括号也要进操作符栈。所以，建立一个操作符表达式节点，内容赋值为左括号 (，指针压入操作符栈中，并将 expflag 加 1，代表还有 expflag 这么多的（ 未配对。
算法框图： 见图 5.102。

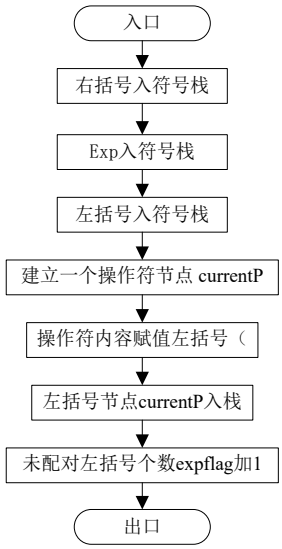


图5.102 产生式89的处理函数process89()的算法框图

（90） 函数 process90( )
产生式： <Factor> **::=** **INTC**
函数声明： void process90( )
算法说明： 创建一个操作数表达式节点，具体类型为常数表达式，将整数的值写入节点中，将这个常操作数节点指针压入操作数栈。
算法框图： 见图 5.103。

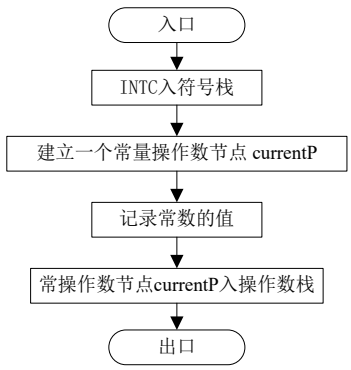


图5.103 产生式90的处理函数process90()的算法框图

（91） 函数 process91( )
产生式： <Factor> **::=** Variable
函数声明： void process91( )
算法说明： 产生式右部非终极符 Variable 入符号栈。
算法框图： 略。

（92） 函数 process92( )
产生式： <Variable> **::=** **ID** VariMore
函数声明： void process92( )
算法说明： 创建一个变量的表达式节点，将标志符名字记入节点中；并将此节点压入操作数栈。
算法框图： 略。

（93） 函数 process93( )
产生式： <VariMore> **::=** **ε**
函数声明： void process93( )
算法说明： 变量表达式的具体类型是标志符变量；将变量类别信息填入当前变量表达式节点。
算法框图： 略。

（94） 函数 process94( )
产生式： <VariMore> **::=** **[** Exp **]**
函数声明： void process94( )
算法说明： 产生式右部入符号栈，变量的具体类型设置为数组成员变量并压入指向数组成员表达式的儿子节点指针，压入具有最低优先级的特殊栈底标志。
算法框图： 见图 5.104。

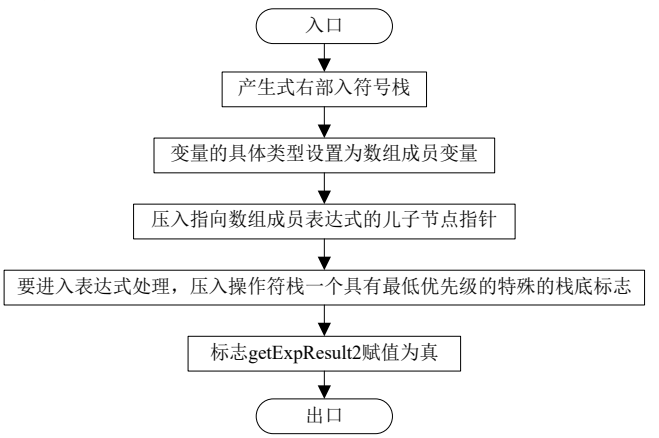


图5.104 产生式94的处理函数process94()的算法框图

（95） 函数 process95( )
产生式： <VariMore> **::=** **.** FieldVar

函数声明： void process95( )
算法说明： 产生式右部入符号栈；当前变量节点的具体类型设置为域成员类型变量；压入域成员变量节点的儿子节点，以处理域成员表达式。
算法框图： 略。

（96） 函数 process96( )
产生式： <FieldVar> **::=** **ID** FieldVarMore
函数声明： void process96( )
算法说明： 创建一个变量的表达式节点；将标志符名记入节点中，弹语法树栈，得到域变量的儿子节点指针，指向此域成员节点。
算法框图： 略。

（97） 函数 process97( )
产生式： <FieldVarMore> **::=** **ε**
函数声明： void process97( )
算法说明： 说明变量表达式的具体类型是标志符变量；将变量类别信息填入当前变量表达式节点。
算法框图： 略。

（98） 函数 process98( )
产生式： <FieldVarMore> **::=** **[** Exp **]**
函数声明： void process98( )
算法说明： 同函数 process94 的处理。

（99） 函数 process99( )
产生式： <CmpOp> **::=** **LT**
函数声明： void process99( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（100） 函数 process100( )
产生式： <CmpOp> **::=** **EQ**
函数声明： void process100( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（101） 函数 process101( )
产生式： <AddOp> **::=** **PLUS**
函数声明： void process101( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（102） 函数 process102( )
产生式： <AddOp> **::=** **MINUS**

函数声明： void process102( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（103） 函数 process103( )
产生式： <MultOp> ::= **TIMES**
函数声明： void process103( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

（104） 函数 process104( )
产生式： <MultOp> ::= **OVER**
函数声明： void process104( )
算法说明： 右部终极符入符号栈。
算法框图： 略。

5．5 语法分析程序的自动生成器

5．5．1 YACC/Bison

1. YACC/Bison 简介:

YACC（Yet Another Compiler-Compiler）是一个 LALR(1)分析器的自动生成器。YACC 与 LEX 一样，是贝尔实验室在 UNIX 上首先实现的，而且与 LEX 有直接的接口，它是 UNIX 的标准应用程序。GNU 工程推出 Bison，是对 YACC 的扩充，同时也与 YACC 兼容。目前，YACC/Bison 与 LEX/FLEX 一样，可以在 UNIX、LINUX、MS-DOS 等环境运行，鉴于 YACC/Bison 的兼容，后面仅针对 Bison 进行介绍。

Bison 是一个通用的语法分析生成器，它读入一个 LALR(1)上下文无关文法，生成一个用来分析这个文法的语法分析器，通常生成的语法分析器是一个 C 程序。只要熟练掌握 Bison 这个强有力的工具，就可以很容易开发出大多数语法分析器，如简单一点的计算器和复杂一点的程序设计语言。

自底向上语法分析的关键是构造出识别文法活前缀的状态机。LALR(1)文法的状态机的构造有两种途径，一种是先构造文法的 LR(1)状态机，然后通过同心状态合并的方法得到 LALR(1)状态机，YACC 采用的就是这种方法；另一种是先构造 LR(0)的状态机，再通过展望符传播的方式得到 LALR(1)状态机，Bison 采用的是这种方法。LALR(1)状态机的自动构造是语法分析生成器自动构造的理论基础。

2 .YACC/Bison 的应用过程:

使用 Bison 自动构造语法分析器的过程如图所示，

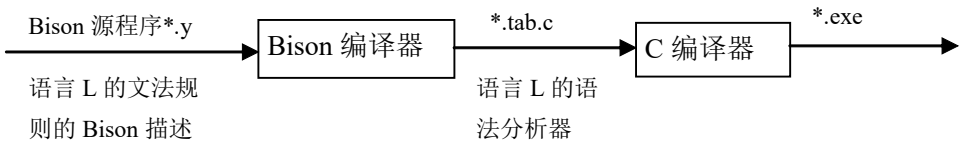


图 5.105 Bison 的应用过程

Bison 编译器接收 Bison 源程序*.y，产生一个*.tab.c 的 C 程序，该程序就是生成的语法分析器，经过 C 编译器编译之后，再连接目标文件，生成可执行程序。

在使用 Bison 之前，需要建立 Bison 语法文件，建立过程分为如下几步：
（1）用 Bison 语法书写 LALR(1)语法规则
（2）编写词法分析器
（3）编写 main()函数
（4）编写错误处理例程

建立了 Bison 语法文件以后，将其生成可执行代码，分为如下几步：
（1）用 Bison 程序将 Bison 语法文件生成语法分析程序
（2）编译 Bison 语法分析程序，以及其他附属程序
（3）连接目标文件，生成可执行程序

3 .Bison 语法:

Bison 语言和源程序是对语言的语法规则的描述，用来输入文法规则。Bison 源程序由三部分组成：声明部分、语法规则部分、辅助 C 代码部分，各部分之间用符号％％隔开，其结构如下：

```
    %{
    C 语言声明部分
    %}                    } 声明部分
    Bison 声明部分
    %%
语法规则部分
    %%
辅助 C 语言代码部分
```

其中声明部分又包含两部分内容：C 语言声明部分，该部分用符号%{和%}括起来；文法符号（一般为终极符）和文法规则的说明以及对文法规则说明的一些限定规则和条件的声明部分（如运算符的结合性和优先级等），该部分的每一项均以％开头。

语法规则部分是 Bison 源程序的主体部分，是对文法的全部规则及每一规则相关的语义动作的描述。如文法中某一文法规则

<左部文法符号>→<候选式 1>|<候选式 2>|…|<候选式 n>

用 Bison 描述的一般形式为：

```
<左部文法符号>：  <候选式 1>  {语义动作 1}
                |<候选式 2>  {语义动作 2}
                |…
                |<候选式 n>  {语义动作 n}
```

其中左部文法符号即非终极符要用小写字符串来表示，文法规则描述的候选式中，对文法的形如+，－，*，/的单字符终极符要用单引号括起来，其它多字符终极符要用％token 来声明，并且用大写字符串表示。如'+'，'-'，INTEGER，IDENTIFIER，IF 和 RETURN 等都是终极符；expr，stmt 和 declaration 等都是非终极符。

语义动作是完成语义处理的 C 语言程序。一旦有规则匹配，这个规则所包含的语义动作也得到执行。通常，语义动作用来计算整条规则的语义值。例如，表达式规则可以写成如下形式：

```
expr:     expr '+' expr   { $$ = $1 + $3; }
          ;
```

规则后面用大括号括起来的部分定义的就是这条规则的语义动作。$j 表示规则右边（':'右边）第 j 个终极符或非终极符的语义值，$$表示整个规则的语义值。这个规则的语义动作定义了加法表达式的值为加号左右两个表达式的值之和。

当产生语法错误时，很多编译器和解释器在报告错误类型的同时会报告出错误的产生位置（如哪行哪列等）。使用 Bison 可以很容易做到这一点。

符号不仅有类型值和语义值，而且还有位置值。位置值的计算同语义值一样，都是在语义动作中完成的（默认情况下，Bison 会自动生成位置值的计算代码）。同语义值的计算方式类似，每个符号都有位置值。用@j 表示规则右边第 j 个终极符或非终极符的位置值，用@$表示整个规则的位置值。

Bison 程序的第三部分，即辅助子程序部分，是由若干 C 语言函数构成的，如词法分析程序 yylex 及错误诊断程序等。

4. Bison 应用——代数计算器（中缀表达式计算器）：calc

处理算术表达式（中缀表达式）需要解决的问题有二：操作符优先级和括号嵌套。

calc 的声明部分：

```
%{
    #define YYSTYPE double
    #include <math.h>
%}
```

```
/* Bison 声明 */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG              /* 相反数 */
%right '^'             /* 幂运算 */
```

C 声明部分中预定义了 YYSTYPE 宏，用于指定终极符和非终极符的类型。Bison 声明部分中定义了终极符类型。在本例中，所有的操作符都用字符形式表示，所以终极符类型就只有一个常数类型：NUM。

在 Bison 声明部分中还引入了两个重要的特性：

- %left 用于声明终极符和左结合性质。相应的%right 用于声明终极符和右结合性质。而%token 仅仅声明了一个终极符，但没有指明其结合性。另外，操作符的优先级也和终极符的声明顺序相关。声明越靠前（上）的终极符优先级越低，声明越靠后（下）的终极符优先级越高。因此上述的语法中'+'和'-'具有相同的优先级，且都最低，而'^'具有最高的优先级；
- 另一个重要的特性是语法定义部分中的%prec。用来指明规则'| '-'exp'具有同 NEG 一样的优先级。

calc 的语法规则部分：

```
%%
input:    /* 空串 */
        | input line
;
line:     '\n'
        | exp '\n'                  { printf ("\t%.10g\n", $1); }
;
exp: NUM                            { $$ = $1; }
        | exp '+' exp               { $$ = $1 + $3; }
        | exp '-' exp               { $$ = $1 - $3; }
        | exp '*' exp               { $$ = $1 * $3; }
        | exp '/' exp               { $$ = $1 / $3; }
        | '-' exp %prec NEG         { $$ = -$2; }
        | exp '^' exp               { $$ = pow($1, $3); }
        | '(' exp ')'               { $$ = $2; }
;
```

上述文法中包含 3 个非终极符：input，line 和 exp。其中 input 为起始符。每个非终极符都有好几条规则，不同规则之间用'或者'（'|'）分隔。有些规则还包含了语义动作。先来看看 input，input 包含两条规则，可以解释成"输入可以是空串，也可以是自己后面跟一个 line"。input 的第一条规则是空串原因是':'与'|'之间为空。定义这条规则是为了使 calc 在运行之后可以什么也不做就退出；input 的第二条规则是左递归定义的，因为 input 永远是输入序列中最左边的符号。定义这条规则是为了使 calc 尽可能多的读取输入行。

再看看 line 规则，第一条规则使 calc 可以接受空行；第二条规则使 calc 在接受

了完整的 exp 并且换行时，输出 exp 的语义值。注意，这条规则的语义动作并没有给整条规则（‘$$’）赋值，原因是整条规则的值在将来不会用到。

最后看看 exp 规则，每条规则中的语义动作都对这条规则的语义值进行了赋值。

rpcalc 的词法分析部分

语法分析器的输入是词法分析器输出的符号串。由于语法分析器只关心符号的类型，所以词法分析器输出给语法分析器的实际上是符号的类型串。例如，假设有整数类型 INTEGER，浮点类型 FLOAT，则当词法分析器遇到 40 时，就返回 40 的类型 INTEGER 给语法分析器；而当词法分析器遇到 4.0 时，就返回 4.0 的类型 FLOAT 给语法分析器。

这些类型都是以宏的形式表示，Bison 把每一个终极符类型都定义成一个宏，并将每个宏与一个不同的整数（大于 255）联系。对于用字符形式表示的终极符，Bison 就直接用它的 ASCII 码表示其类型。用于这些宏是在程序开头定义的，所以在词法分析器 yylex()中可以直接使用这些宏。

尽管语法分析器只关心符号的类型，而且词法分析器也只需将符号的类型返回给语法分析器，但是符号的语义值对程序仍然非常重要，所以在词法分析阶段，要由词法分析器 yylex()生成每个终极符的语义值（非终极符的语义值是在语法分析阶段的语义动作中完成的）。全局变量 yylval 用于设定当前终极符的语义值，它的类型由 YYSTYPE 宏指定，默认情况下是 int 类型。当遇到文件结束符时，词法分析器应当返回 0，表示输入结束。

下面是 calc 的词法分析器 yylex()：

```
#include <ctype.h>
int   yylex (void)
{
      int c;
      /*  跳过空白符  */
      while ((c = getchar ()) == ‘ ’ || c == ‘\t’)
            ;
      /*  处理数  */
      if (c == ‘.’ || isdigit  （c）)
      {
            ungetc (c, stdin);
            scanf (“%lf”, &yylval);
            return NUM;
      }
      /*  返回输入结束  */
      if (c == EOF)
      return 0;
      /*  返回单个字符  */
      return c;
}
```

调用语法分析程序 yyparse()

通常在 main()函数中调用语法分析程序 yyparse()：

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

错误报告程序 yyerror()

当遇到语法错误的时候，yyparse()会调用错误报告程序 yyerror()来输出错误信息：

```
#include <stdio.h>
void yyerror (const char *s)
{
      printf (“%s\n”, s);
}
```

5．5．2 ACCENT

1. ACCENT 简介

ACCENT（A Compiler Compiler for the Entire Class of Context-Free Languages）是在 UNIX(LINUX)上开发的共享软件，也是一个语法分析器的自动生成器。同 YACC/Bison 一样，它可以在 UNIX、LINUX、MS-DOS 等环境下运行。

同 Bison 相比，ACCENT 是一个更加通用的语法分析器的自动生成器。

（1）ACCENT 对输入的文法没有任何限制，能处理所有的上下文无关文法，而 Bison 只能处理 LALR(1)文法；

（2）ACCENT 在处理文法时不会产生任何冲突，而 Bison 受所处理文法类的限制，可能会产生移入/归约或归约/归约冲突；

（3）ACCENT 产生的语法分析器分析效率略低，由 YACC 生成的分析器分析效率很高。

2. ACCENT 的语法

ACCENT 语言和源程序是对语言的语法规则的描述，用来输入文法规则。ACCENT 的源程序也是由三部分组成：

全局定义部分　　token 声明部分　　语法规则部分

（1）全局定义部分用来定义一些函数、全局变量以及类型，这部分可空。

格式为：%prelude {c-code}

用花括号括起来的部分为任意的 C 代码，它会原封不动的复制到 ACCENT 产生的语

法分析器文件的首部。

（2）token 声明部分用来声明文法中出现的所有终极符（称为 tokens），这部分可空。
格式为：%token tk1，tk2，…… ，tkn ；
ACCENT 中的 tokens 分为两种：

- 多字符终极符，这类 token 必须用“%token”关键字声明，
  如：%token NUMBER，IF，THEN，ELSE，WHILE，DO，
- 单字符终极符，称为“literal token”，这类 token 不必声明，用单引号引起来即可，可直接出现在产生式中，
  如：statement：variable ‘=’ exp
  | IF exp THEN statement ELSE statement
  | ……
  ；

（3）规则部分用来说明文法的产生式，至少应有一条规则。
格式为：rule1 rule2 …… rulen
其中每条规则的格式为：
<左部文法符号>→<候选式 1> | <候选式 2> | … | <候选式 n> ；
文法中的每个非终极符都用一条规则来定义，第一条规则的左部为整个文法的开始符。

ACCENT 的规则部分，还增加了新的描述手段：

- 局部选择（Local Alternatives）
  格式：(alt_1 | …… | alt_n)，可以避免引入新的非终极符
  如：signed_number：sign NUMBER；
  sign ：‘+’ | ‘-’；
  利用局部选择可以写成：signed_number：(‘+’ | ‘-’) NUMBER；
- 可选符(Optional Elements)
  格式：(M_1 … M_n)?
  意义：括号中的 M_1 … M_n 可出现在输入中，也可以不出现
  例如：integer：(sign)? NUMBER；
  则 123 和 +123 都是正确的输入
- 星闭包(Repetitive Elements)
  格式：(M_1 … M_n)*
  意义：表示括号中项目的任意次重复(包括 0 次)
  例如：number_list：NUMBER (‘,’ NUMBER) * ；
  则“12”、“12 , 24”、“12 ， 32 , 4”都是合法的输入

ACCENT 在进行语法分析的同时，也可以进行语义处理。ACCENT 的语义动作可以嵌套在文法的任意位置，当指定的分支匹配时，对应的语义动作也被执行。语义动作为用花括号括起来的任意 C 代码，这些 C 代码将会被原文拷贝到语法分析程序的适当位置。
例如有如下文法：
N: {printf(“1\n”);} ‘a’ {printf(“2\n”);} ‘b’ {printf(“3\n”);}

| {printf(“x\n”);} ‘c’ {printf(“y\n”);}

则对于输入 “ab”，语法分析程序会产生如下输出：

1
2
3

ACCENT 输出的语法分析程序，对应每个非终极符都会有一个函数，所以与程序设计语言中的函数类似，非终极符也可以有参数，这些参数可以在语义动作内访问。参数有两种模式：

① in 模式：in 类型参数是将信息从上下文传递到非终极符中，即继承属性。

② out 模式：将信息从非终极符传递到上下文中，称作综合属性。

下面是产生式左、右部参数的格式和用法：

- 若规则左部的非终极符带有参数，则其格式如下：

  left_hand<parameter_spec_list>
  left_hand<%in parameter_spec_list>
  left_hand<%out parameter_spec_list>
  left_hand<%in parameter_spec_list , %out parameter_spec_list>

  parameter_spec_list: 类型名 参数名，类型名 参数名，……

  其中“类型名”可选，如果类型名不写，则默认为 YYSTYPE 类型，YYSTYPE 是一个宏，代表 long 类型，用户可对其重定义。如果一个参数既不加%in 修饰，也不加%out，则默认为是综合属性。

  例： N<%in int context , %out int result>：……；

- 产生式右部的非终极符如果有参数，则将其用尖括号“<”、“>”括起来跟在该非终极符的后面。

  例：N<actual_context , actual_result>

注意：

（1）参数能在语义动作内访问，输入参数的值必须在语义动作内定义或者是其它文法符号的输出参数。

（2）若非终极符有综合属性，则其每个分支都要定义输出参数并在语义动作内给其赋值。

（3）如果在语义动作内使用左部非终极符的输出参数，需用“*”操作符访问。

（4）参数变量均不需定义。

例如：

```
demo:
    {actual_context=10;}
     N<actual_context，actual_result>
     {printf(“%d\n”，actual_result);}
N<%in int context，%out int result>：{*result=context+1;};
```

经 ACCENT 产生的语法分析程序如下：

```
demo()
{
        int actual_context;      /* 参数变量不需定义*/
        int actual_result;
        switch(yyselect()){
        case 1:
        {
```

```
                actual_context=10;   /* 输入参数必须有值   */
                N(actual_context, &actual_result); /*综合属性实际上是地址引用*/
                Printf("%d\n",actual_result);
            }
            break;
        }
    }

    N(context,result)
        int context;
        int * result;
    {
        switch(yyselect()){
        case 2:
        {
            *result=context+1;   /* 输出参数必须赋值 */
        }
        break;
        }
    }
```

用%token 语句声明的 token 都有一个 YYSTYPE 类型的输出参数，若要在规则的右部使用 token，则可以为其指定一个实参来访问其语义值。

例如： Value： NUMBER<n>{printf("%d\n"，n);}

这里 n 就表示 NUMBER 的数值(语义值)，可以在语义动作内使用。

一个 token 的语义值是在该 token 的词法规则的语义动作内被计算的，其值必须赋给 YYSTYPE 类型的系统变量 yylval。

规则中使用的属性变量都不需定义。如果想在语义动作内声明其它变量即局部变量，则分下面两种情况：

（1）该变量只对某一分支可见

如 demo：{int i=0;} alt_1 | alt_2 ；

（2）对规则内所有分支可见(但对其它规则不可见)

demo：

```
    %prelude {int i=0;}
      alt_1 |  alt_2
```

3. ACCENT 应用——代数计算器（中缀表达式计算器）：calculator

下面是按照 ACCENT 的语法一个计算器的文法，经 ACCENT 产生的分析程序能分析计算含简单加减乘除运算的算术表达式。

```
%token NUMBER;
expression :
        term<n> { printf("%d\n", n); };
term<n> :
        term<x> '+' factor<y> { *n = x+y; }
```

```
            | term<x> '-' factor<y> { *n = x-y; }
            | factor<n>;
factor<n> :
             factor<x> '*' primary<y> { *n = x*y; }
            | factor<x> '/' primary<y> { *n = x/y; }
            | primary<n>;
primary<n> :
             NUMBER<n>
            | '(' term<n> ')'
            | '-' primary<x> { *n = -x;};
```

下面是 ACCENT 产生的分析程序的一部分：

```
term (n)
    YYSTYPE *n;
{
    YYSTYPE x;
    YYSTYPE y;
    switch(yyselect()) {
    case 2: {
        term(&x);
        get_lexval();
        factor(&y);
#line 8 "example\calc.acc"
  *n = x+y;
# line 39 "yygrammar.c"
        } break;
    case 3: {
        term(&x);
        get_lexval();
        factor(&y);
#line 9 "example\calc.acc"
  *n = x-y;
# line 47 "yygrammar.c"
        } break;
    case 4: {
        factor(n);
        } break;
    }
}
```

4. ACCENT 的应用过程

首先是按照 ACCENT 的语法编写输入文件*.acc，“acc”是 ACCENT 输入文件的后缀名，然后是用 ACCENT 的编译器编译文法输入文件*.acc，产生输出文件 yygrammer.c 和 yygrammer.h，其中 yygrammer.h 含有所有 token 编码的定义和 YYSTYPE 类型的定义（在这之前如果用户没有重定义才有效），yygrammer.c 是生成的语法分析器，经过 C 编译器编译后，得到可执行文件，其中包含语法分析函数

yyparse()。

因为语法分析的输入是经词法分析后的 token 序列，而不是用户直接写的源程序，因此必须为ACCENT产生的语法分析器提供词法分析函数，函数名固定为yylex。yylex 可以自己编写，也可以用 Lex 自动生成（最好用 Lex 生成），其返回值为一个整数，表示所识别单词的类别。Lex 自动生成的 C 文件中有词法分析函数 yylex。

ACCENT 还包含一个文件 auxil.c，其中包含用户定义的一些辅助函数，主要有：

（1）主函数 main()，用来调用语法分析函数 yyparse()；

（2）错误处理函数 yyerror(char * msg)，当语法分析程序有错误时，将调用该函数，输出错误信息以及出错行号；

（3）yywrap()函数，词法分析程序分析完一个源程序后调用 yywrap，若其返回 1 则表示再没有输入文件了，若返回 0 则表示还有其它输入文件。通过 yywrap 函数可以一次分析多个源程序文件。

以上面提到的计算器 Calculator 为例，用 ACCENT 来生成语法分析器的过程可图示如下：

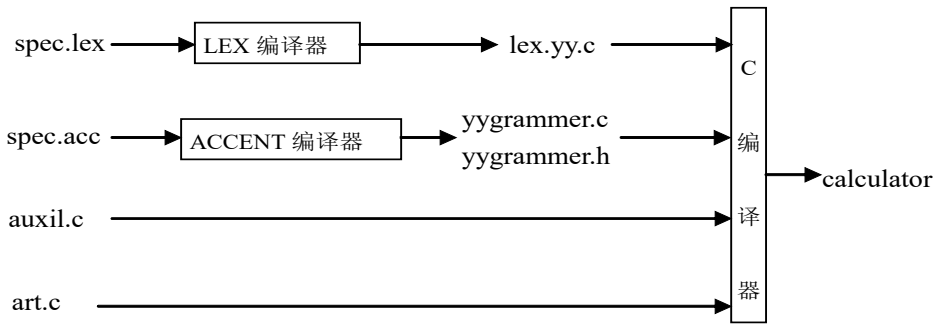


图 5.106 使用 ACCENT 生成语法分析

第六章 符号表管理与语义分析

6．1 语义分析概述

语义分析（Semantic Analysis）是任何编译程序必不可少的一个阶段，也是编译程序最实质性的工作。在整个编译过程中，词法分析和语法分析是对源程序形式上的识别和处理，而语义分析程序是对源程序的语义做相应的处理工作。

语义分析的主要任务是针对语法分析后得到的内部结构进行语义处理，既要做相应的语义检查工作，同时也要为后面的编译工作提供足够的信息，这些信息包括标识符的语义信息、类型信息、过程信息等等。

SNL 的语义分析主要完成以下两项工作：

1．构造符号表和信息表：
（1） 构造标识符的符号表；
（2） 构造类型信息表，包括数组信息表和记录信息表；
（3） 构造过程信息表；
（4） 构造形参信息表。

2．进行语义错误检查，语义错误通常包括下面几类：
（1） 标识符的重复定义；
（2） 无声明的标识符；
（3） 标识符为非期望的标识符类别（类型标识符，变量标识符，过程名标识符）；
（4） 数组类型下标越界错误；
（5） 数组成员变量和域变量的引用不合法；
（6） 赋值语句的左右两边类型不相容；
（7） 赋值语句左端不是变量标识符；
（8） 过程调用中 ，形实参类型不匹配；
（9） 过程调用中， 形实参个数不相同；
（10）过程调用语句中，标识符不是过程标识符；
（11）if 和 while 语句的条件部分不是 bool 类型；
（12）表达式中运算符的分量的类型不相容。

6．2 符号表管理

编译程序在执行的过程中，为了完成源程序到目标代码的翻译，需要不断收集、记录和使用源程序中的一些语法符号的类型、特征和属性等相关信息。一般的做法

是在编译程序的工作过程中，建立并保持一系列的表格，如常数表、变量名表、数组名表、过程名表以及标号表等，习惯上将它们统称为符号表或名字表。符号表的每一登记项，将填入名字标识符以及与该名字相关联的一些信息。这些信息，全面的反映各个符号的属性及它们在编译过程中的特征，诸如名字的种属（常数、变量、数组、标号等），名字的类型（整型、实型、逻辑型、字符型等），名字的特征（当前是定义性出现还是使用性出现等），给该名字分配的存储单元地址以及与该名字的语义有关的其他信息等等。符号表中的各类信息将在编译程序工作过程中的适当时候填入。对在语义分析阶段建造符号表的编译程序，当遇到标识符声明部分时，每当遇到一个名字声明，就以此名字查符号表；若表中无此登记项，则将该名字填入表中；若该表中已有此登记项，则说明该名字是重复声明，报告语义错误。至于与该名字相关的一些信息，可视工作的方便，分别在编译程序工作过程中的适当时候填入：种属，类型，特征等信息可在语义分析阶段完成；名字的存储地址等信息则要在代码生成阶段完成。几乎在编译程序工作的全部过程中，都需要对符号表进行频繁的访问，查表和填表等操作，是编译程序的一笔很大的开销。因此，合理地组织符号表，并相应地选择好查表和填表的方法，是提高编译程序工作效率的重要一环。

6．2．1 符号表的内容

一般而言，即使对于同一类符号表，例如变量名表，它的结构以及表中的每一登记项所包含的内容，由于程序设计语言和目标计算机的不同，都可能有较大差异。然而抽象地看，各类符号表一般都具有如表 6.1 所示的形式。由表 6.1 可以看出，符号表的每一记录项都由两个数据项组成：第一个数据项为名字，用来存放标识符；第二个数据项为信息，一般由若干子项（或域）组成，用来记录与该名字相对应的各种属性和特征。

名字项	信息项
Name1	Name1_info
Name2	Name2_info
……	……
Name*n*	Name*n*_info

表 6.1 符号表的结构

对于标识符的长度有限制或长度变化范围不大的语言而言，每一登记项名字栏的大小可以取标识符的最大允许长度。例如，SNL 语言规定每个标识符最多可包含 10 个字符，因此可用 10 个字符的空间作为名字栏的长度。

在源程序中，由于不同种属的标识符起着不同的作用，因而相应于各类标识符所需记录的信息也就可能有很大的差异。如果根据标识符的不同种属，在编译程序中分门别类地组织多种表格，如常数表、变量名表、数组名表、过程名表、标号表等等，对于表格的使用将非常方便。但是，如果能合理组织符号表信息项各个子项所存信息的内容（例如适当地增加标志位），则在编译程序中为各类标识符设置一张

共用的表格也是可行的。在 SNL 的编译系统中，符号表将表示变量名，类型名，过程名等对象的相关信息，采用统一的符号表结构，名字的种属用一个标记位 kind 来区分。下面给出 SNL 的符号表的结构：

类型标识符

TypePtr	**kind**
	typeKind

- TypePtr 指向类型的内部表示
- kind 指示标识符的类别，值为 typeKind 表示类型标识符

变量标识符

TypePtr	**kind**	**Access**	**Level**	**Off**
	varKind			

- TypePtr 指向类型的内部表示
- kind 指示标识符的类别，值为 varKind 表示变量标识符
- Access 表示是直接变量还是间接变量：access = （dir,indir）
  indir 表示是间接变量（变参属于间接变量），dir 表示直接变量
- Level 表示该变量声明所在主程序/过程的层数
- Off 表示该变量相对它所在主程序/过程的偏移量

过程标识符

TypePtr	**kind**	**Level**	**Parm**	**Code**	**Size**
NULL	procKind				

- 过程标识符无返回类型,故 TypePtr 的值为 NULL
- kind 指示标识符的类别,值为 procKind 表示过程标识符
- Level 表示过程层数
- Param 表示形参信息表地址
- Size 表示过程所需的空间大小
- Code 表示过程的目标代码地址（此部分在代码生成阶段填写）

类型内部表示

size	**kind**	**MORE**		
		ArrayAttr		body
		indexTy	elemTy	

- size 表示类型所占空间的大小
- kind 指示具体的类型,如值为 intTy 表示整数类型
- indexTy 当类型为数组类型时有效,指向数组下标类型
- elemTy 当类型为数组类型时有效,指向数组元素类型
- body 当类型为记录类型时有效,指向域成员的链表

在 SNL 编译系统中，记录类型域名标识符的属性登记在记录类型的内部表示中，

并不记录在符号表中，即在标识符种类里没有域名种类。

在标识符的内部表示中涉及到层数、偏移量、过程的存储大小和目标代码入口地址等内容。其中层数和偏移量可在处理声明部分时确定，而其它部分只能在目标代码的生成阶段确定。这些部分放在后面介绍。

标识符信息项的数据结构的 C 语言定义如下：

```
typedef struct
{
    struct typeIR       * idtype;          /*指向标识符的类型内部表示*/
    IdKind              kind;              /*标识符的类型*/
    union
    {
        struct
        {
            AccessKind   access;
            int          level;
            int          off;
        }VarAttr;                          /*变量标识符的属性*/
        struct
        {
            int          level;
            ParamTable   * param;          /*参数表*/
            int          code;
            int          size;
        }ProcAttr;                         /*过程名标识符的属性*/

    }More;                                 /*标识符的不同类型有不同的属性*/
}AttributeIR;
```

其中所有标识符都有类型部分，即第一个属性为类型信息是所有标识符的共同性质。其它信息部分则随标识符的种类不同而不同。由于一个标识符的类别只可能为 typeKind, varKind, procKind 其中之一，所以在结构体中采用了共同体（联合）的形式，能够在分配内存时减少无用的浪费。

SNL 的类型包括：整数类型，字符类型，数组类型，记录类型，布尔类型（其中布尔类型只在判断条件表达式的值时使用）。其中整型和字符类型是标准类型，其内部表示可以事先构造，数组和记录类型等构造类型则要在变量声明或类型声明时构造。

类型的种类采用枚举定义：

```
typedef    Typekind = { intTy, charTy , arrayTy, recordTy, boolTy }
```

1．标准类型整型、字符型的内部表示：

（1） 整数类型的内部表示 intPtr：

```
struct  {  size     //值为 1（我们假定整数占用一个单元）
           kind     //值为 intTy
        }
```

（2） 字符类型的内部表示 charPtr：

```
struct  {  size     //值为 1（我们假定字符占用一个单元）
           kind     //值为 charTy
        }
```

2．非标准类型的内部表示：

（1） 数组类型的内部表示：

```
struct  {  size        //其值需要计算才能确定
           kind        //值为 arrayTy
           indexTy     //指向数组下标类型的内部表示
           elemTy      //指向数组元素类型的内部表示，即指
                       //向整数或字符
        }
```

（2） 记录类型的内部表示：

```
struct  {  size        //值需要计算确定
           kind        //值为 recordTy
           body        //指向记录的域类型表（链）的地址
        }
```

其中，Body 指向记录的域类型表（链）地址。因此，此处还应设置域类型单元结构：

```
struct   fieldChain
         {   idname     // 变量名
             unitType   // 域中成员的类型
             offset     // 所在记录中的偏移
             next       // 链表指针
         }
```

类型内部结构中的第一个成分是类型大小的信息，第二个成分是具体类型。其他信息的内容随类型不同而不同。可用下列统一的数据结构描述：

```
struct  typeIR
{
    int              size;              /*类型所占空间大小*/
    TypeKind  kind;
    union
   {   struct
           {  struct typeIR   * indexTy;
              struct typeIR   * elemTy;
           }ArrayAttr;
       fieldChain * body;          /*记录类型中的域链*/
   } More;
```

```
}TypeIR;
```

至此，可以给出 SNL 的符号表 symbTable（id,AttributeIR）的数据结构的定义：

```
struct    symbtable
{
      char    idname[10];
      AttributeIR    attrIR;
      struct symbtable    *    next;
}SymbTable;
```

进行语义分析时，除了要用到符号表外，还可能会用到记录的域表，形参信息表等其他信息表。它们的结构如下：

记录域表：

idname	unitType	offset	next

- idname　　记录域中的标识符；
- unitType　　指向域类型的内部表示；
- offset　　表示当前标识符在记录中的偏移；
- next　　指向下一个域。

形参信息表：

entry	next

- entry　　指向形参标识符在符号表中的位置；
- next　　指向下一个形参。

6．2．2 符号表的组织

对于按照什么样的次序来安排符号表中的各登记项，与所选用的查表方式有关。当符号表的内容不多时，一般按照标识符在源程序中出现的先后顺序，将各名字及其信息依次填入表中。因此查表时，也是将待查的名字与表中的各登记项相比较，直至得到与待查名字匹配的登记项，或查完表中现有的各登记项为止。SNL 编译系统采用的就是顺序查表法。

许多高级程序设计语言都具有分程序结构或嵌套过程结构。在用这类语言所写的程序中，可以再包含嵌套的程序单元，而且相同的标识符可以在不同的程序单元重复定义和使用。如 Pascal 语言中的过程说明等。由于程序单元的嵌套导致名字作用域的嵌套的情况，也可以视为分程序结构。如 C 语言中，用花括号{ 和 }括起来的分程序和复合语句，也具有嵌套的特征。

对于分程序结构或嵌套过程结构的语言，应采用分层建立和处理符号表的方式。这样每当遇到一个标识符时，可以方便地找到所对应的符号表项。SNL 语言是分程序结构语言，每个可独立编译的程序单元可能是多模块或单模块，下面，就以 SNL 语言为例来说明构造这种符号表的方法。

在 SNL 程序中，标识符的作用域是包含声明（定义）该标识符的一个最小分程序。具体说，即：

（1） 如果一个标识符在某一分程序首部被声明，则不论此分程序是否含有内层分程序，也不论内层分程序再嵌套多少层，只要在内层分程序中没有对此标识符的再次声明，则该标识符的作用域就是整个分程序。对于声明该标识符的分程序而言，该标识符是局部量，但对该分程序所包含的内层分程序而言，则为全局量。

（2） 由于 SNL 程序中的过程可具有嵌套结构，因此，可将每一过程声明都看作是一个分程序。出现在过程体中的非形式参数，依其在相应的过程体中被说明（定义）与否，确定它们对过程而言是局部量还是非局部量，而形式参数则总是局限于相应的过程体的。

由此可见，SNL 程序中标识符的作用域，总是与声明（定义）这些标识符的分程序的层次相关联的。为了体现 SNL 程序中各个分程序的嵌套关系，我们将这些分程序按其开始符号在源程序中出现的先后顺序进行编号。这样，在从左至右扫描源程序时，可按分程序在源程序的自然顺序（静态层次），对出现在各个分程序中的标识符进行处理，其过程如下：

（1） 当在分程序的首部某声明中遇到一个标识符时，则查相应本层分程序的符号表，若符号表中已有该标识符，则说明该标识符被重复说明，应按语义错误处理。否则，就在符号表中新登记一项，将该标识符及其相关信息填入。

（2） 当在分程序的语句中遇到一个标识符时，首先在该层分程序的符号表中查找此标识符，如果找不到，则在其直接外层符号表中查找，直到在某一层符号表中找到了标识符，则从表中提取有用信息并作相应处理；如果查遍了所有外层符号表，都没有找到该标识符，则说明程序中使用了一个未声明的标识符，应按语义错误处理。

为实现上述符号表的处理思想，SNL 的符号表组织原则如下：

（1） 分层建立符号表，使各分程序的符号表项连续地排列在一起，不会被内层分程序的符号表所割裂；

（2） 建立一个分程序表，用来记录各层分程序符号表的起始位置。

SNL 编译器的局部符号表的一般结构如图 6.1 所示, 其中分程序表用 SymbTable 类型的数组 Scope 表示，Scope[i]指向第 i 层符号表。

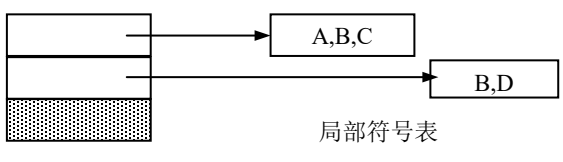


图 6.1 SNL 的局部符号表示意图

6．2．3 符号表的操作

1． 符号表的创建、删除与查找：

创建：当前层数 Level 加 1，表示创建了一个新的符号表，并不真正创建符号表；遇到新的符号表的第一个元素时，使 scope[Level] 指向它。

删除：当前层数 Level 减 1，表示删除当前符号表。

查找：符号表的查找使用简单的顺序查表法。

2． 符号表的局部化：

采用局部符号表的方法：每个局部化单位一个符号表，利用 scope 栈，栈的所有元素都指向一个局部符号表。使用 scope 栈实现局部化和嵌套规则的具体过程是：

- 进入新的局部化单位时，scope 栈的层数 Level 加 1，表示创建一个新的符号表，并压入 scope 栈中；
- 遇到定义性标识符时，将其属性登记到当前的符号表中；
- 遇到使用性标识符时，首先查栈顶所指的符号表，其次查次栈顶所指的符号表，一直进行到查到或再没有表可查为止；
- 结束一个局部化单位时，scope 栈的层数 Level 减 1，表示弹出 Scope 栈的栈顶元素。

6．2．4 符号表的实现

符号表相关函数的说明及算法框图如下：

1． 创建一个符号表

函数声明：void CreatTable（void）

功能说明：当进入一个新的局部化单位时，调用本子程序。功能是建立一个空符号表 table，层数加 1，偏移初始化为 0。

算法说明：Off 赋值初始值 initOff ；

其中 initOff 的值根据目标代码生成需要而定，取值过程活动记录中，从 sp 到形参区开始地址的偏移量；由于目标代码生成阶段，首先要保存：动态链指针，返回地址，过程层数，三个累加寄存器，sp 到 display 表的偏移量等 7 个信息，故这里 initOff 定为常数 7。

2． 撤销一个符号表

函数声明：void DestroyTable（ ）

算法说明：Level = Level –1

3． 登记标识符和属性到符号表

函数声明：bool Enter（char * Id, AttributeIR *AttribP ,SymbTable **Entry）

算法说明：将标识符名，标识符属性登记在符号表中，登记成功，返回值为真，Entry 指向登记的标识符在符号表中的位置；登记不成功，返回值为假，Entry 值为空。

算法框图：见图 6.2

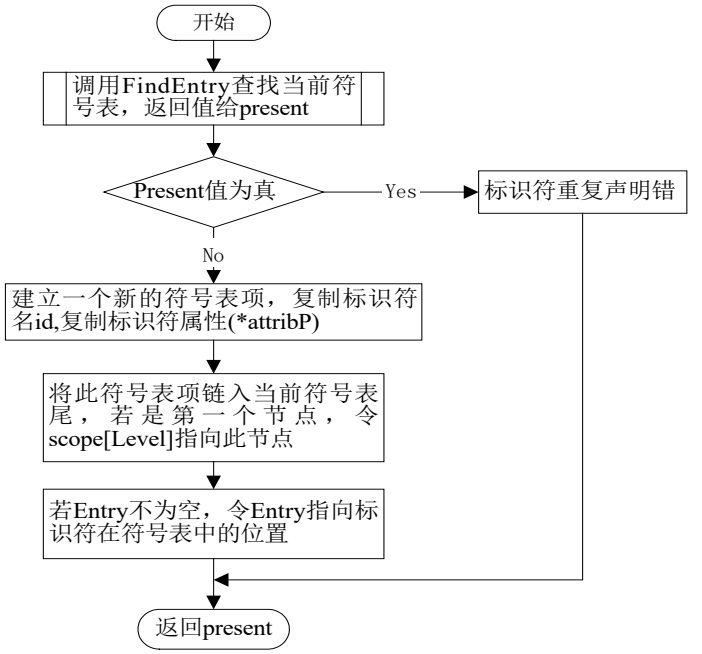


图6.2 登记标志府到符号表的函数Enter的算法框图

4. 符号表中查找标识符：

函数声明：bool *FindEntry（char *id , flag , SymbTable **Entry）

算法说明：根据 flag 的值为 one 还是 total，决定在当前符号表中查找标识符，还是在 scope 栈中的所有符号表中查找标识符。

算法框图：见图 6.3

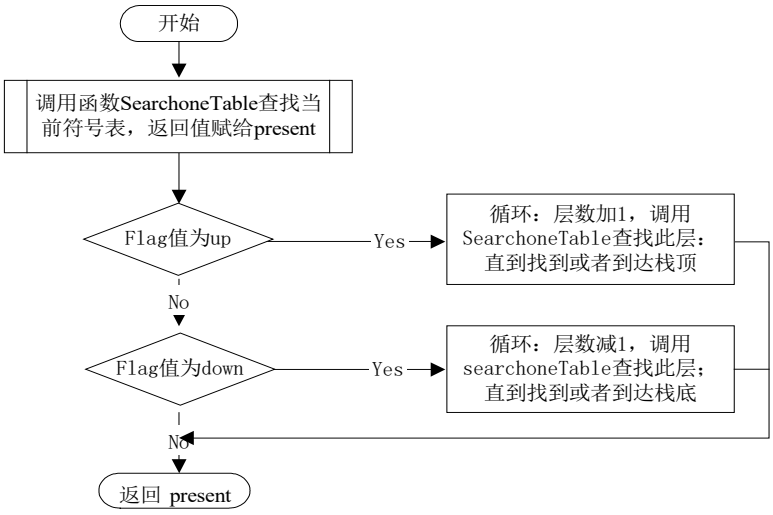


图6.3 标识符查找函数FindEntry的算法框图

函数 SearchoneTable （char * id , currentLevel ,SymbTable **Entry ）

算法说明：从表头开始，依次将节点中的标识符名字和 id 比较是否相同，直到找到此标识符或到达表尾，若找到，返回真值，Entry 为标识符在符号表中的位置，否则，返回值为假。

5. 在域表中查找域名

函数声明： bool FindField（char * Id , FieldChain *head,FieldChain **Entry ）

参数说明： head 指向域表头；id 为要查找的标识符；查找成功，Entry 记录标识符在域表中的位置，返回真值；否则，Entry 为空，返回假值。

算法说明： 从域表头开始 ，依次将节点中的标识符名字和 id 比较是否相同，直到找到此标识符或到达表尾，若找到，返回真值，Entry 为标识符在符号表中的位置，否则，返回值为假。

6. 打印符号表：

函数声明： void PrintSymbTable（ ）

算法说明： 从第一层开始，依次打印各层符号表的内容。

6．3 语义分析实现

语义分析主要完成两部分工作：建立符号表和进行语义错误检查。

在 SNL 编译器中，标识符可以是变量名，类型名和过程名，因此是把变量标识符，类型标识符和过程标识符添加到符号表。在生成符号表时，将以先根次序遍历语法分析树，即在遍历处理语法树节点的子节点之前，先处理该节点，进行标识符属性填表。在对语法树节点处理过程中，只对含有变量标识符，类型标识符和过程标识符声明的语法树节点进行处理。即：类型声明节点，变量声明节点，过程声明节点。处理的时候，先查符号表，如果标识符不在符号表中，则为标识符的第一次声明，填入抽象地址，名称，类型等信息。否则，为标识符的重复声明，输出语义错误信息。

在 SNL 编译器中，语义错误检查涉及标识符的重复声明错误，有使用无声明错误和类型一致性检查。前面提到了标识符的重复声明错误的检查，对于标识符有使用无声明错误，也是以先根次序遍历语法分析树节点，遍历时只对含有变量标识符，类型标识符和过程标识符使用的语法树节点进行处理。即：赋值语句类型节点，读语句类型节点，表达式类型节点和过程调用语句节点。处理的时候，先查符号表，如果标识符不在符号表中，则为标识符的有使用无声明错误，输出语义错误信息。否则，读取标识符的相关信息。在类型一致性检查过程中，按照后根次序遍历语法分析树，即在遍历处理语法分析树的子节点之后，再处理该节点，进行类型检查。在对语法树节点处理过程中，只针对运算类型表达式节点，常量表达式类型节点，标识符表达式类型节点，条件语句类型节点，赋值语句类型节点，写语句类型节点，循环语句类型节点，过程调用语句类型节点。

概括地讲，SNL 语义分析主要完成以下一些任务：

（1）构造符号表；

（2）检查相关的语义错误；

（3）输出语义错误信息；
（4）把过程标识符和语句中的变量在符号表中的地址信息回填到语法树中；
（5）在 TraceTable 值为真时，调用 printSymbTable 函数将生成的符号表输出到列表文件 listing。

6．3．1 输入输出

SNL 语义分析程序以语法分析程序所生成的与源程序结构相对应的语法分析树为输入。语义分析程序通过遍历语法分析树进行语义分析，在 TraceTable 值为真时，调用 printSymbTable 函数将生成的符号表输出到列表文件 listing，以及类型能够检查信息和语义错误信息。

6．3．2 算法框图

1． 语义分析主函数
函数声明：void Analyze（TreeNode *currentP）
算法说明：语义分析总程序，对语法树进行分析。
算法框图：见图 6.4。

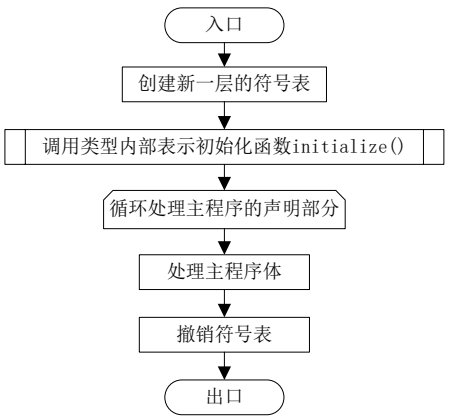


图6.4 语义分析主函数Analyze的算法框图

2． 初始化基本类型内部表示函数
函数声明：void initialize（void）
算法说明：初始化整数类型，字符类型，布尔类型的内部表示说明由于这三种类型均为基本类型，内部表示固定。
算法框图：略。

3． 类型分析处理函数
函数声明：TypeIR * TypeProcess（TreeNode * t,DecKind deckind）
算法说明：类型分析处理。处理语法树的当前节点类型，构造当前类型的内部表示，并将其地址返回给 Ptr 类型内部表示的地址。
算法框图：见图 6.5。

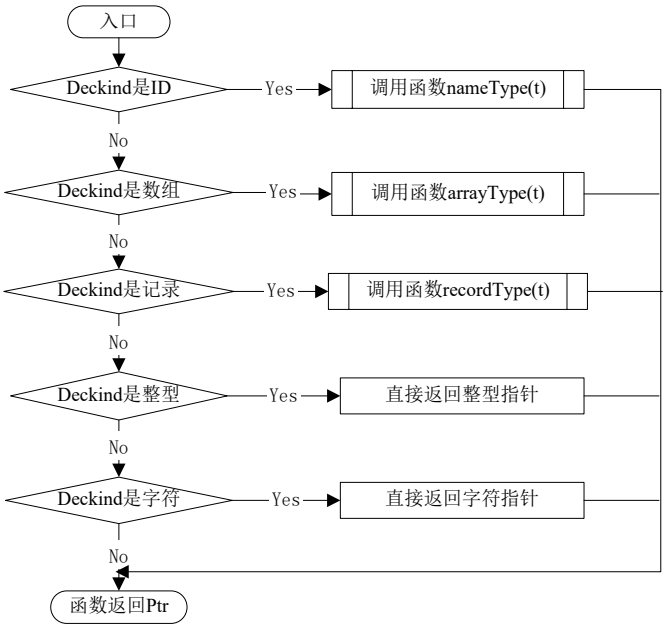


图6.5 类型分析处理函数TypeProcess的算法框图

4. 自定义类型内部结构分析函数

函数声明：TypeIR * nameType （TreeNode * t）

算法说明：符号表中寻找已定义类型名，调用寻找表项地址函数 FindEntry，返回找到的表项地址指针 entry。如果 present 为 0，则发生无声明错误。如果符号表中的该标识符属性信息不是类型，则非类型标识符错。

算法框图：见图 6.6。

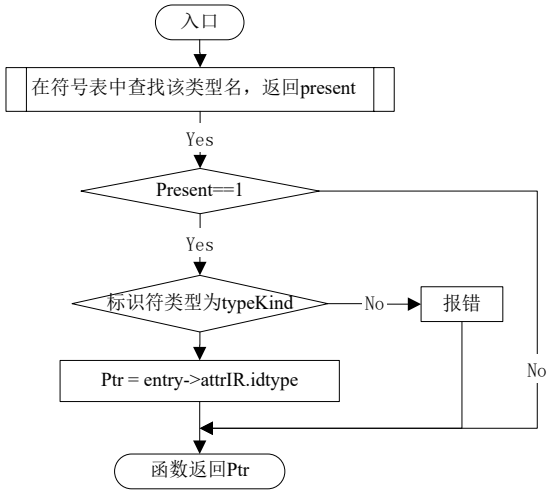


图6.6 自定义类型内部结构分析函数nameType的算法框图

5. 数组类型内部表示处理函数

函数声明：TypeIR * arrayType（TreeNode * t）

算法说明：创建数组类型的内部表示。此时需检查下标是否合法。

算法框图：见图 6.7。

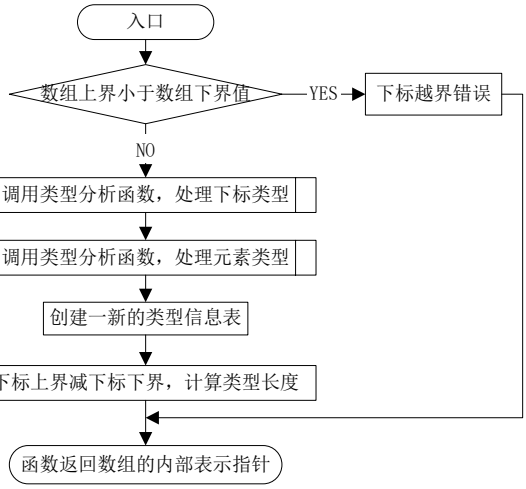


图6.7 数组类型内部表示处理函数arrayType的算法框图

6. 处理记录类型的内部表示函数

函数声明：TypeIR * recordType（TreeNode * t）

算法说明：类型为记录类型时，是由记录体组成的。其内部节点需要包括 3 个信息：一是空间大小 size；二是类型种类标志 recordTy；三是体部分的节点地址 body。记录类型中的域名都是标识符的定义性出现，因此需要记录其属性。

算法框图：见图 6.8。

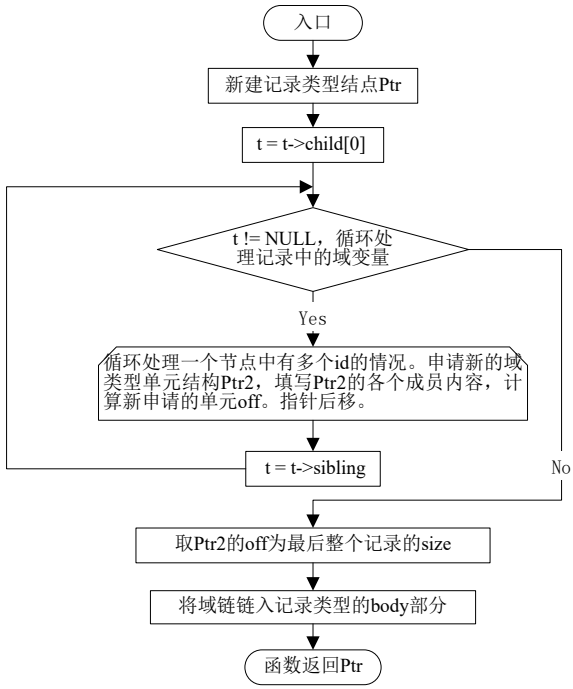


图6.8 记录类型的内部表示处理函数recordType的算法框图

7. 类型声明部分分析处理函数

函数声明： void TypeDecPart（TreeNode * t）

算法说明： 遇到类型 T 时，构造其内部节点 TPtr；对于"idname=T"构造符号表项；检查本层类型声明中是否有重复定义错误。

算法框图： 见图 6.9。

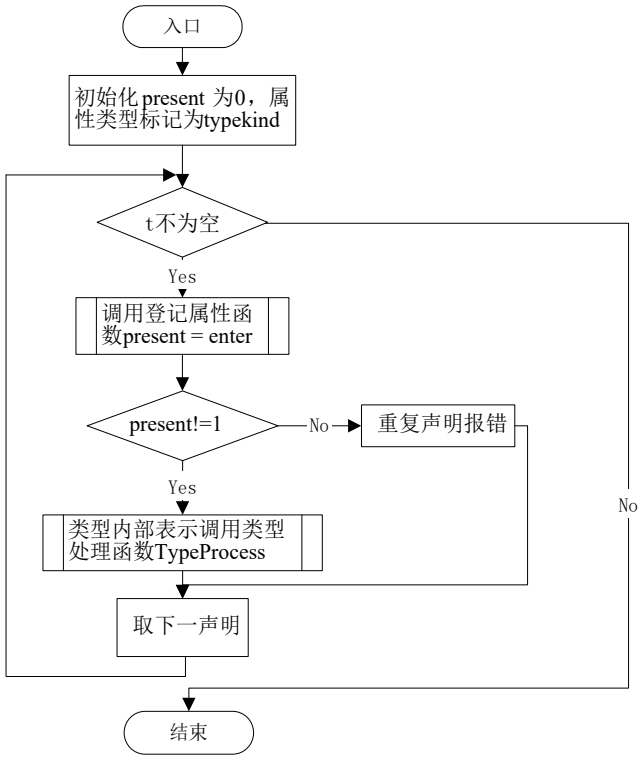


图6.9 类型声明部分分析处理函数TypeDecPart的算法框图

8. 变量声明部分分析处理函数

函数声明：void VarDecList（TreeNode * t）

算法说明：当遇到变量标识符 id 时，把 id 登记到符号表中；检查重复性定义；遇到类型时，构造其内部表示。

算法框图：见图 6.10。

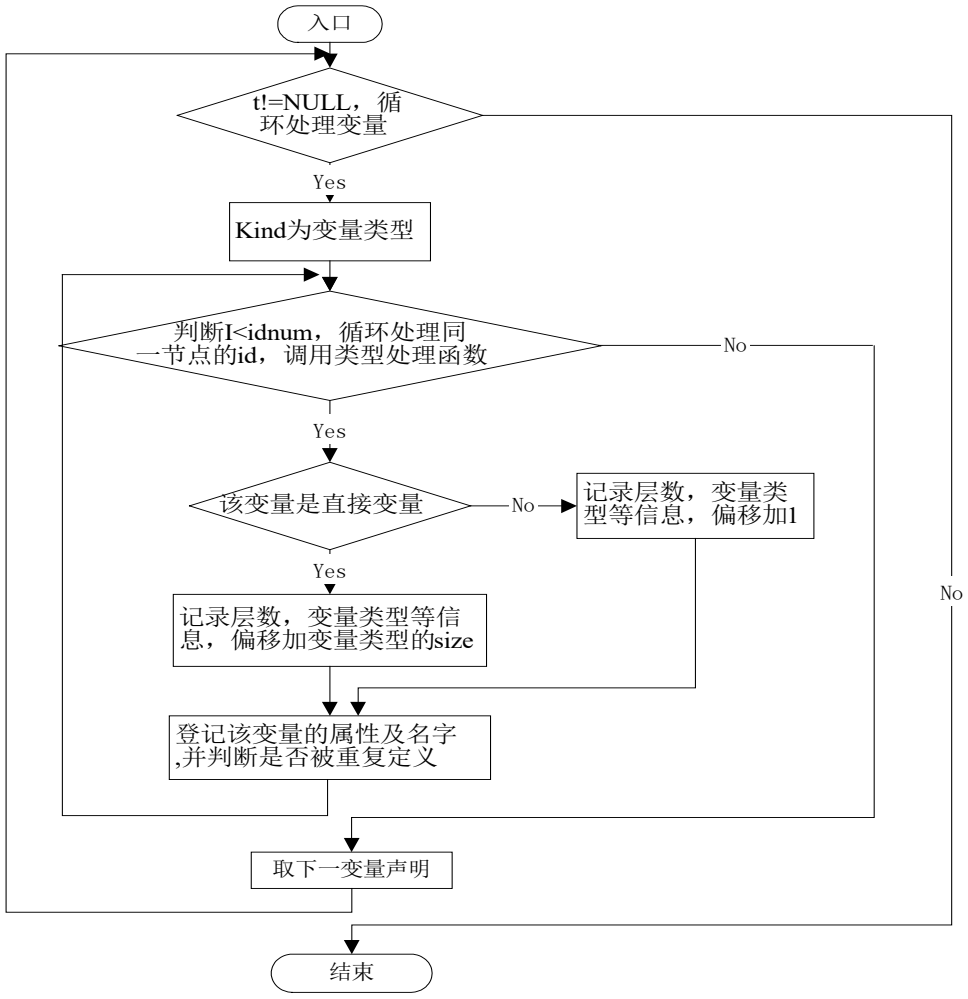


图6.10 变量声明部分分析处理函数VarDecList的算法框图

9. 过程声明部分分析处理函数

函数声明：void ProcDecPart（TreeNode * t）

算法说明：在当前层符号表中填写过程标识符的属性；在新层符号表中填写形参标识符的属性。

算法框图：见图 6.11。

10. 过程声明头分析函数

函数声明：SymbTable * HeadProcess（TreeNode * t）

算法说明：在当前层符号表中填写函数标识符的属性；在新层符号表中填写形参标识符的属性。其中过程的大小和代码需以后回填。

算法框图：见图 6.12。

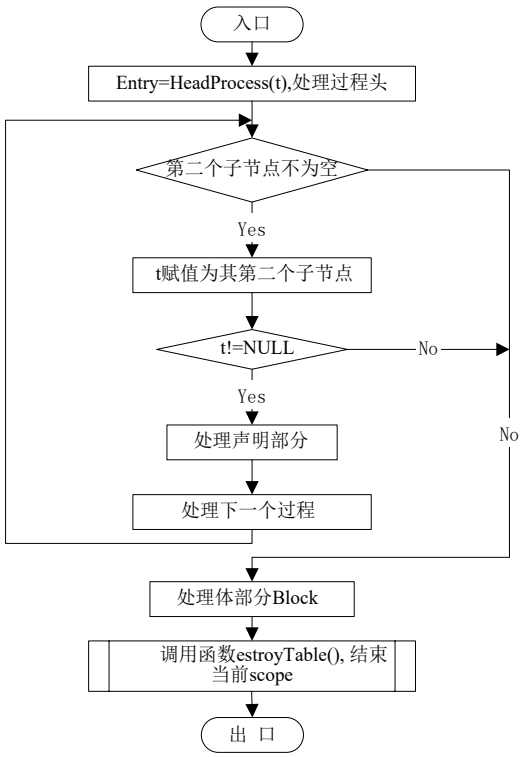


图6.11 过程声明部分分析处理函数ProcDecPart的算法框图

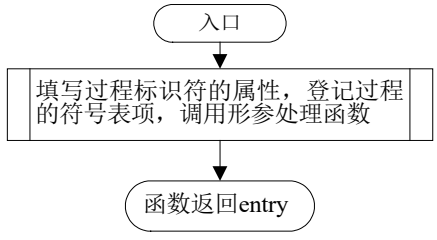


图6.12 过程声明头分析函数HeadProcess的算法框图

11．形参分析处理函数

函数声明：ParamTable * ParaDecList（TreeNode * t）

算法说明：在新的符号表中登记所有形参的表项，构造形参表项的地址表，并使 param 指向此地址表。

算法框图：见图 6.13。

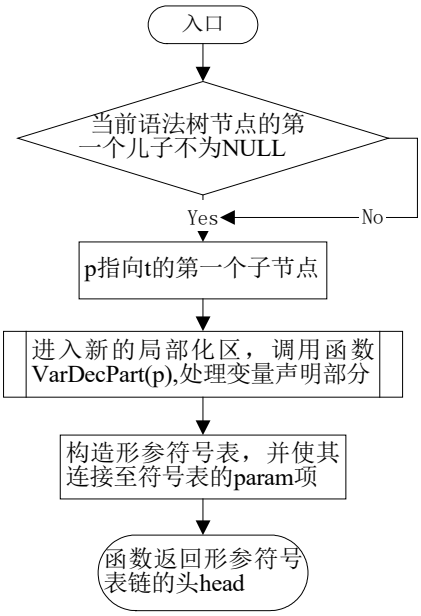


图6.13 形参分析处理函数ParaDecList的算法框图

12．执行体部分分析处理函数

函数声明：void Body（TreeNode * t）

算法说明：SNL 编译系统的执行体部分即为语句序列，故只需处理语句序列部分。

算法框图：见图 6.14。

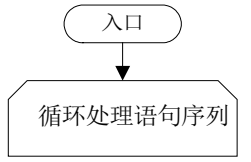


图6.14 执行体部分分析函数Body的算法框图

13．语句序列分析处理函数

函数声明：void statement （TreeNode * t）

算法说明：根据语法树节点中的 kind 项判断应该转向处理哪个语句类型函数。

算法框图：见图 6.15。

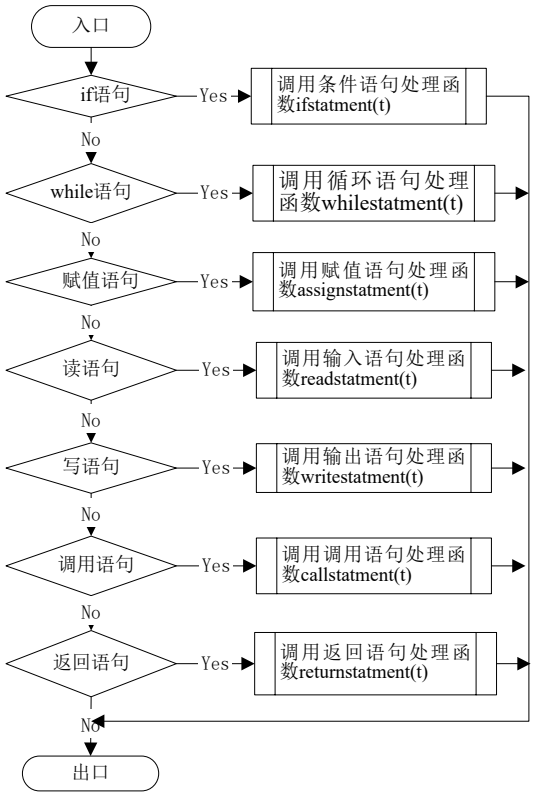


图6.15 语句序列分析处理函数statement的算法框图

14．表达式分析处理函数

函数声明：TypeIR * Expr（TreeNode * t,AccessKind　*　Ekind）

算法说明：表达式语义分析的重点是检查运算分量的类型相容性，求表达式的类型。其中参数 Ekind 用来表示实参是变参还是值参。

算法框图：见图 6.16。

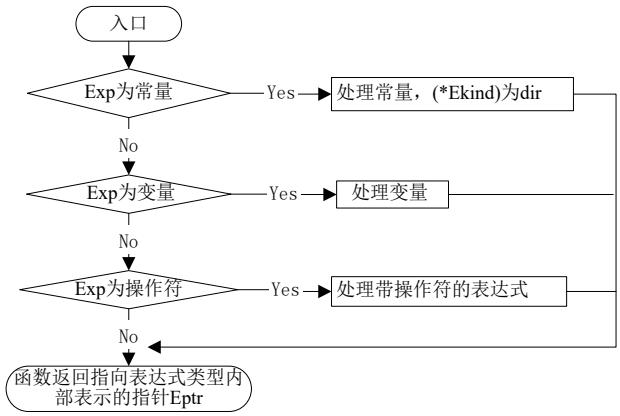


图6.16 表达式分析处理函数Expr的算法框图

其中，变量处理部分的算法如图 6.17 所示：

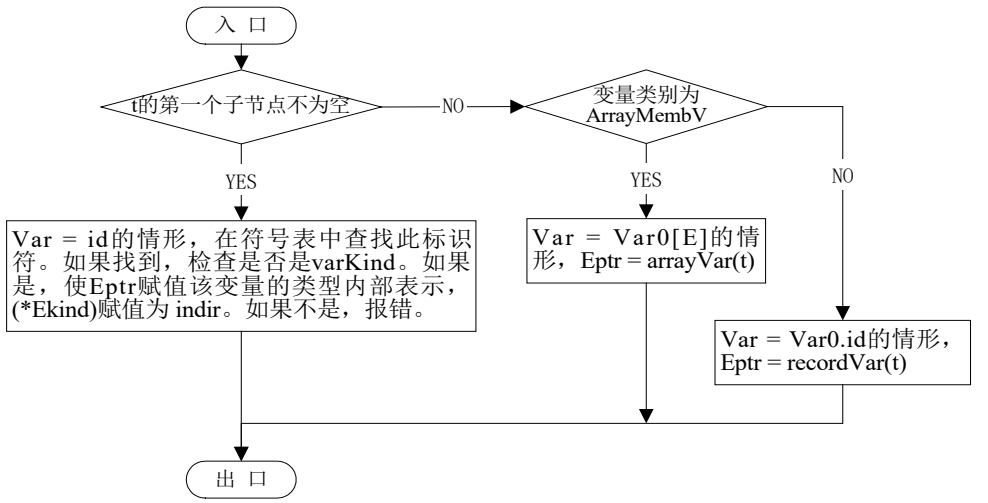


图6. 17 变量分析处理部分的算法框图

处理带操作符的表达式的算法如图 6.18 所示：

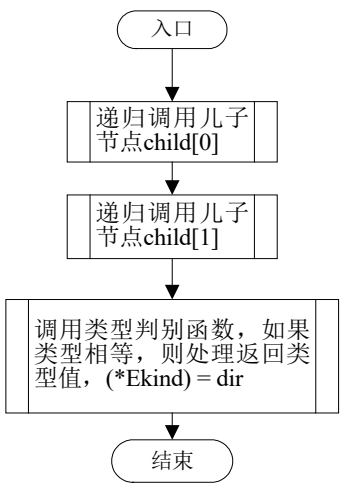


图6. 18 带操作符的表达式分析处理的算法框图

15．数组变量的处理分析函数

函数声明：TypeIR * arrayVar（TreeNode * t）

算法说明：检查 var := var0[E]中 var0 是不是数组类型变量，E 是不是和数组的下标变量类型匹配。

算法框图：见图 6.19。

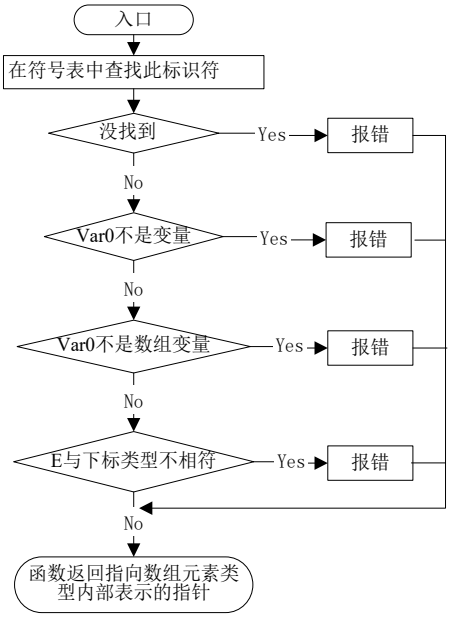


图6.19 数组变量分析处理函数arrayVar的算法框图

16．记录变量中域变量的分析处理函数

函数声明：TypeIR * recordVar（TreeNode * t）

算法说明：检查 var:=var0.id 中的 var0 是不是记录类型变量，id 是不是该记录类型中的域成员。

算法框图：见图 6.20。

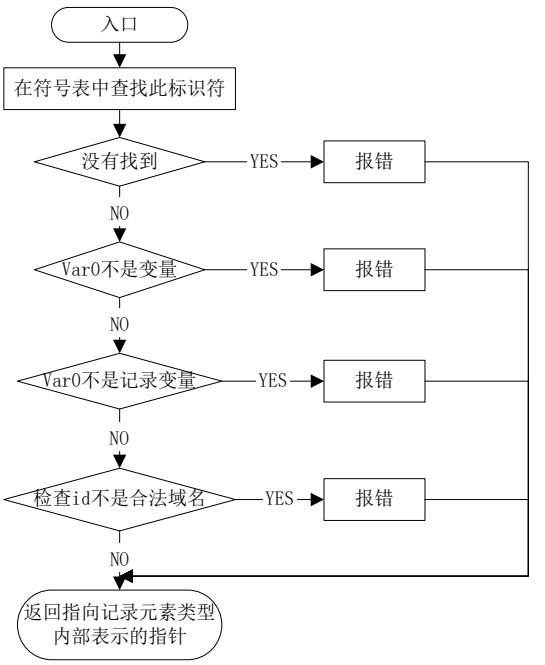


图6.20 记录变量分析处理函数recordVar的算法框图

17．赋值语句分析函数

函数声明：void assignstatement（TreeNode * t）

算法说明：赋值语句的语义分析的重点是检查赋值号两端分量的类型相容性。

算法框图：见图 6.21。

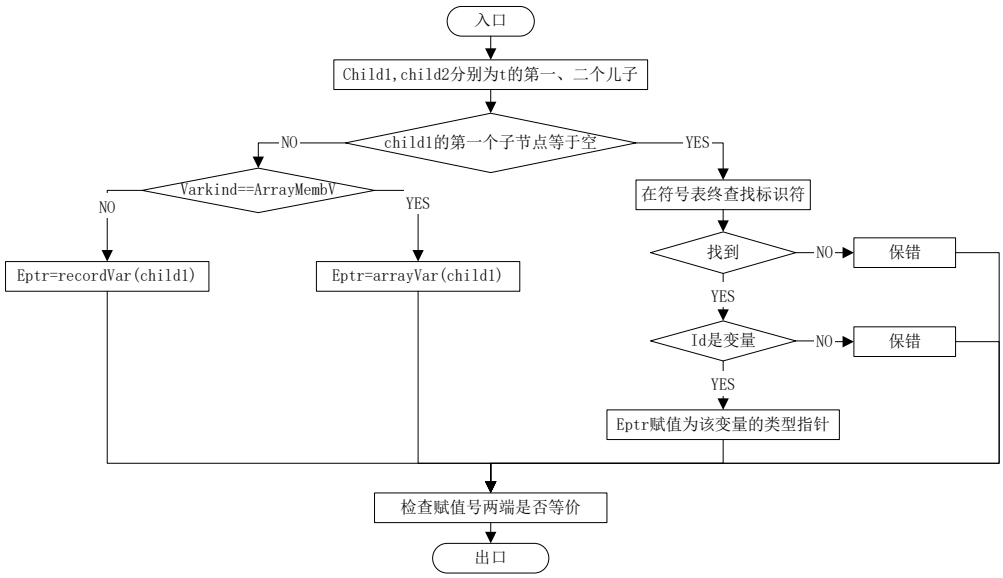


图6.21 赋值语句分析处理函数assignstatement的算法框图

18．过程调用语句分析处理函数

函数声明：void callstatement（TreeNode * t）

算法说明：函数调用语句的语义分析首先检查符号表求出其属性中的 Param 部分（形参符号表项地址表），并用它检查形参和实参之间的对应关系是否正确。

算法框图：见图 6.22。

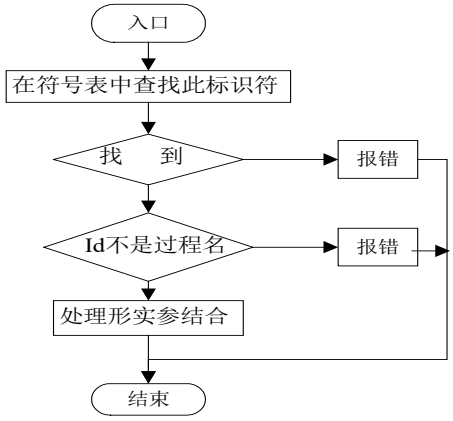


图6.22 过程调用语句分析处理函数callstatement的算法框图

19．条件语句分析处理函数

函数声明：void ifstatment（TreeNode * t ）
算法说明：检查条件表达式是否为 bool 类型，处理 then 语句序列部分和 else 语句序列部分。
算法框图：见图 6.23。

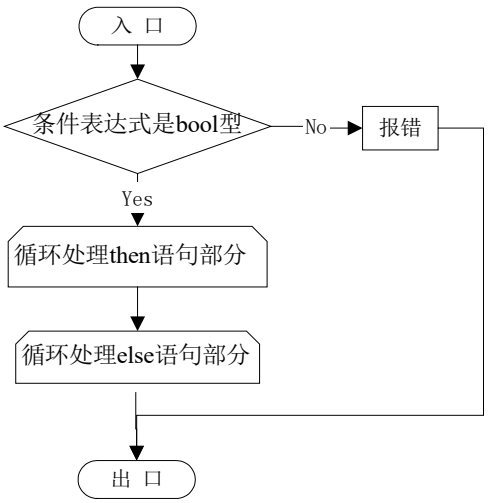


图6.23 条件语句分析处理函数ifstatement的算法框图

20．循环语句分析处理函数
函数声明：void whilestatement（TreeNode * t）
算法说明：检查条件表达式是否为 bool 类型，处理语句序列部分。
算法框图：见图 6.24。

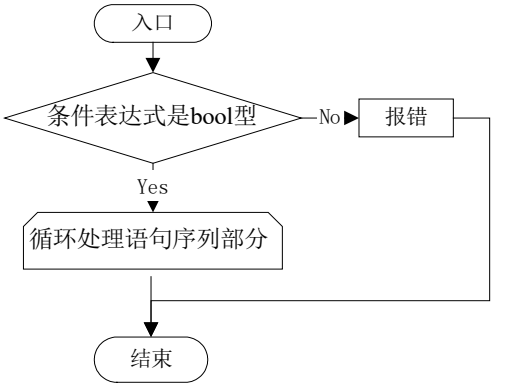


图6.24 循环语句分析处理函数whilestatement的算法框图

21．读语句分析处理函数
函数声明：void readstatement （TreeNode * t）
算法说明：检查要读入的变量有无声明和是否为变量。
算法框图：见图 6.25。

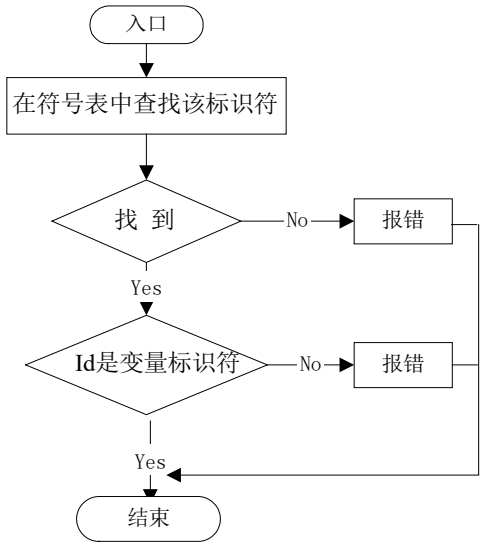


图6.25 读语句分析处理函数readstatement的算法框图

23．写语句分析处理函数

函数声明：void writestatement（TreeNode * t ）

算法说明：分析写语句中的表达式是否合法。

算法框图：见图 6.26。

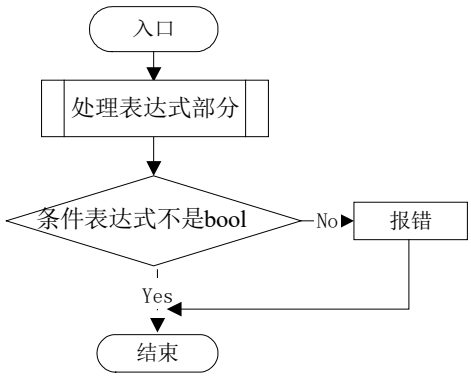


图6.26 写语句分析处理函数writestatement的算法框图

第七章 中间代码生成

7．1 中间代码简介

中间代码生成阶段不是编译器的必须阶段。我们知道编译器可分为单遍扫描和多遍扫描两种方式。其中单遍扫描编译器是从源代码直接生成目标代码，也就是在一遍扫描中，词法分析、语法分析、语义检查、目标代码生成全部完成，这时就不需要生成中间代码；而多遍扫描编译器是对源程序进行多次扫描,完成不同的分析和代码生成工作。不同的编译器扫描次数也不尽相同,一种方案是经过词法分析、语法分析、语义检查等工作后先生成中间代码，然后再从中间代码生成目标代码。通常把与目标机不相关的编译工作称之为编译器的前端，即从源代码的词法、语法、语义分析到生成中间代码的部分，而把与目标机相关的部分称之为编译器的后端，即从中间代码生成目标代码的部分。

在采用多遍扫描方式中，有的编译器并不生成中间代码，而是在语义分析之后直接生成目标代码，这样做的好处是：可避免重复性工作，从而减小编译器的体积。在大多数编译器中，都是采用生成中间代码的方式，即与目标机无关的编译工作完成的越多越好。主要原因有三点：

- 便于移植：从词法分析到中间代码生成部分都不依赖于目标机，因此对不同的目标机只需要修改编译器的目标代码生成部分，就可以生成不同该语言在不同机器上的编译器。例如我们有 A 机上的 L 语言编译器，想在 B 机上开发 L 语言的编译器，则可把 A 机的编译器前端移植过来，不需要做任何改变，只开发其编译器的后端即可。
- 便于优化：要对程序进行处理，首先要把程序转换成某种类型的数据，而中间代码本身是程序的一种数据结构的表示，因此给优化过程带来了很大方便。
- 便于修改：中间代码比目标代码的结构更接近于源程序的结构，有助于了解程序的意图，方便编译程序的修改和维护。

在 SNL 语言的编译器中，经过语法分析或语义分析得到的语法树实际上已经是一种中间代码表示形式，可以在它的基础上进行优化和目标代码生成操作。但是，从教学的角度出发，为了向读者介绍更多的中间代码表示形式、中间代码生成和优化技术（经典的优化技术大多是在四元式上进行的），我们还是将中间代码生成作为独立的一遍扫描，介绍给大家。出于同样的考虑，在后面的目标代码生成阶段，我们又分别介绍了由语法树直接生成目标代码和由四元式生成目标代码两种方法，读者在实际编写编译器时，可以自由选择其中的一种方法。

7．1．1 中间代码的表示形式

中间代码作为源程序的一种等价表示，其复杂性介于源程序和机器语言程序之间，由于各种原因，在编译器的历史上出现了各种形式的中间语言。常见的有以下几种：

1．后缀式

后缀式也叫逆波兰式。这种表示的特点是把运算量（操作数）写在前面，运算符（操作）写在后面，因此称为后缀式。例如表达式 X+Y 写成 XY+。后缀式的另一个特点是不需要括号，每当遇到操作符即可进行处理，操作符的静态顺序即是实际的运算顺序。后缀式主要适合于栈式目标机，在优化和代码生成方面并不具有优势。

2．三元式

三元式是一个三元组，其形式为：n（OP，A，B），其中 n 是三元式的编号，用于对三元式计算结果的引用，OP 为操作符，A，B 为操作数，既可以是一般的操作数也可以是三元式的编号。如表达式 X+Y，其三元式为：三元式编号（+，X，Y）。三元式的优点是表示形式简单，缺点是不同三元式之间是通过三元式编号联系在一起的，三元式的顺序就表明了计算的顺序，即对位置的依赖性太强，不便于代码优化。改进的办法是设立一个间接三元式表，当代码优化需要调整运算顺序时，只需重新安排间接三元式表而无需改动三元式。

3．四元式

四元式是一个四元组，其形式为：（OP，A，B，T），其中 OP 为操作符，A，B 为操作数，T 为运算结果。如表达式 X+Y，其四元式是：（+，X，Y，T）。操作数和运算结果可以是一般的变量，常量，标号，也可以是编译时引入的临时变量。同三元式一样，四元式的顺序同样表明了计算的顺序，所不同的是，四元式之间的联系是通过临时变量实现的，因此可以很方便地调整四元式的位置，便于代码优化处理。

4．树形表示

树也是一种比较常用的中间表示形式，其形式简单直观，简单变量和常数的树就是该变量或常数自身，二目运算对应二叉子树，多目运算对应多叉子树。如果表示 e1 和 e2 的树为 T1 和 T2，那么，e1+e2，e1*e2 和-e1 的树分别如图 7.1（a），（b），（c）所示。

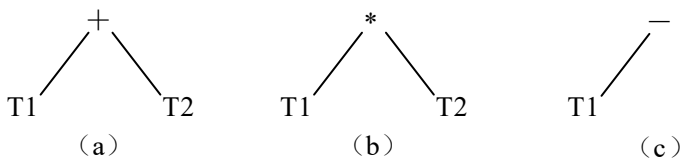


图 7.1 表达式的树形表示

为了便于安排存储空间，一棵多叉子树可以通过引入新节点的办法而表示成一棵二叉子树。树形表示实际上是三元式表示的翻版，因此树形表示也不利于优化。

这些种中间表示各有所长，难以判断哪一种方式比另外一种方式好，选择哪一种中间代码的形式取决于实际编译的需要。

7．1．2 中间代码的生成方法

中间代码常用的生成方法有以下几种：

■ 语法制导的翻译方法

语法制导方法是在语法分析的同时进行所需的语义分析，也就是把语义子程序插入在语法规则当中，当语法分析分析到该处时，则执行相应的语义子程序。中间代码的生成也可以采用这种方法来实现，即在语义子程序中加入中间代码生成处理。语法制导方法的特点是形式化好，简洁，可靠性高。语法制导方法的具体实现有很多形式，如动作文法、属性文法等。

■ 基于语法树的方法

基于语法树的方法是利用已经生成的语法树，通过对树的遍历，根据节点的种类进行代码生成处理。例如对于声明类节点，只有过程声明节点需要生成中间代码，因此当遍历到声明类节点时，如果节点不是过程声明节点，则不生成代码。这种方法的特点是原理简单，对于初学者非常适用。缺点是不够灵活，当文法稍作改变，则要做相当大的改动，易出错。但作为教学实践来讲，还是非常合适的。

■ 栈的数据结构

传统的方法可利用栈的数据结构来实现中间代码的翻译，这种方法的思想来源于表达式求值的处理模型。在表示式求值时，需要定义两个栈，操作数栈和操作符栈。求值过程中，遇到操作数，则送入操作数栈；遇到操作符，则和操作符栈的当前操作符比较优先级，优先级高则进栈，优先级低，则从操作数栈弹出两个元素，进行计算，并将计算结果送回操作数栈，重复上面的过程，直到处理完输入串。利用栈的数据结构来生成中间代码的过程与上述过程类似，扫描输入的 token 序列，根据扫描单词的种类，分别送入不同的栈中，遇到可以处理的情况，将各个栈中相应的单词弹出，生成对应的中间代码。这种方法处理起来比较复杂，一般只适用于简单的文法，如表达式文法等。

7．2 SNL 的中间语言

SNL 的源程序经过语法分析之后被转换为一棵语义等价的语法树，这其实已经是一种中间表示形式。出于教学原因，我们把语法树转换为简单而且常用的四元式形式的中间代码。此种中间代码形式包括四种信息，其一是运算符，其二和其三是左右运算分量，其四是运算结果，即：

中间代码形式：（ Op, Operand1, Operand2, Result）

其含义是：Result ： ＝ Operand1 Op Operand2

四元式中间代码的各分量和结果随着所表示的代码种类的不同，可以是标号，数值，或者地址，我们定义 ARG 结构来统一表示各操作分量和操作结果。而且，ARG 结构中还保存了操作项的一些属性信息，便于后面阶段的处理。

1．ARG 结构：

	Attr		
			Addr
form	value	Label	name dataLevel dataOff access

结构说明：

成员 form　根据 ARG 结构是数值类，标号类，还是地址类，分别取值 ValueForm ,LabelForm ,AddrForm ；

成员 Attr　记录 ARG 结构的具体内容

Attr 成员 value　在 form 取值 ValueForm 时有效，记录常数的值；

Attr 成员 label　在 form 取值 LabelForm 时有效，记录标号值；

Attr 成员 Addr　变量的 ARG 结构的内容；

Addr 成员 name　记录变量的名字，若是临时变量，则名字为空；

Addr 成员 dataLevel　记录变量标识符所在层数，若是临时变量，没有层数概念，取值 –1 ；

Addr 成员 dataOff　记录变量标识符的偏移量，若是临时变量，取值临时变量的编码；

Addr 成员 access　记录变量的访问方式，根据是直接访问还是间接访问，分别取值 dir 和 indir。

注：地址类（Addr）的 ARG 结构的 name 属性并不是必要的，加入 name 属性只是用于显示中间代码时，用变量名表示变量，增强可读性，便于读者理解。

ARG 结构的示例见表 7.1

中间代码	操作分量或运算结果	ARG 结构
ADD, a, 3, t1	变量 a	AddrForm, a, 1, 3 , dir
	值 3	ValueForm,　3
	临时变量 t1	AddrForm, -, -1,4, dir
JUMP　L1	标号 L1	LabelForm, 1

表 7.1　ARG 结构示意图

2．中间代码结构：

代码类别	操作分量 1	操作分量 2	结果
codekind	Arg1	Arg2	Arg3

3．中间代码序列：（用双向链表表示）

前一条代码　中间代码结构　下一条代码

former	onecode	next

4．所有四元式中间代码及解释：

种类	中间代码形式	中间代码解释
算术运算	（ADD, ARG1, ARG2, ARG3）	ARG3 = ARG1+ARG2
	（SUB , ARG1, ARG2, ARG3）	ARG3 = ARG1- ARG2
	（MULT ,ARG1, ARG2, ARG3）	ARG3 = ARG1* ARG2
	（DIV , ARG1, ARG2, ARG3）	ARG3 = ARG1 / ARG2
关系运算	（EQC, ARG1, ARG2, ARG3）	若 ARG1=ARG2 ,则 ARG3=1, 否则 ARG3 = 0
	（LTC, ARG1, ARG2, ARG3）	若 ARG1<ARG2，则 ARG3=1, 否则 ARG3=0
语句	（READC,ARG1, _ , _ ）	数据读入
	（WRITEC, ARG1, _ , _ ）	数据输出
	（RETURNC, _ , _ , _ ）	返回语句
	（ASSIG, ARG1 , ARG2 , _）	ARG1 = ARG2
	（AADD, ARG1, ARG2, ARG3）	ARG3=ARG1 的地址+ARG2
	（LABEL,ARG1 , _ , _ ）	定义标号 ARG1
	（JUMP, ARG1 , _ , _ ）	转向标号 ARG1
	（JUMP0,ARG1,ARG2, _ ）	若 ARG1=0 ,则转 ARG2
	（CALL,ARG1, _ , ARG3 ）	调用入口为 ARG1 的过程；过程的 display 表的偏移量记录在 ARG3 中
	（VARACT, ARG1 ,ARG2 , _）	对应变参的实参；ARG2 表示对应形参的偏移
	（VALACT, ARG1 , ARG2, _）	对应值参的实参；ARG2 表示对应形参的偏移
定位	（PENTRY,ARG1,ARG2, ARG3 ）	过程体入口声明代码： ARG1 为过程的入口标号； ARG2 记录过程活动记录的大小； ARG3 记录过程的层数
	（ENDPROC, _ , _ , _ ）	过程出口
	（MENTRY, _ ,ARG2 , ARG3）	主程序入口声明代码： ARG2 表示主程序活动记录的大小； ARG3 记录主程序 display 表的偏移量

注：(1) MENTRY 语句，PENTRY 语句，CALL 语句中的分量，涉及到活动记录的大小，display 表的偏移量等信息，这些信息都是生成目标代码时所需要的，而且在语义分析阶段就已经求出，并存放在语法树对应节点中。由于中间代码生成后，以后的处理都是对中间代码进行，不再使用语法树，所以要在相应的中间代码中保存这些信息。

(2) 临时变量的编号问题：临时变量并不是从 0 开始编号，而是初始化为此临时变量所在的活动记录中第一个临时变量的偏移。这也是目标代码生成阶段，确定临时变量所在的绝对地址所需。

(3) 关于(1)和(2)，如果难于理解，可以参考第九章目标代码生成时的讲解，或者留到学习目标代码生成阶段时一起理解。

7．3 SNL 的中间代码生成

在 SNL 的编译器中，经过了语义分析之后得到的是源程序的一棵等价语法树，为进行后面的代码优化，我们需要将语法树转换成等价的中间代码形式。

7．3．1 输入输出

SNL 中间代码生成程序以经过语义分析后的语法分析树作为输入，故中间代码生成阶段不再进行语义检查。通过遍历语法树，生成源程序的四元式中间代码，并在 TraceMidCode 为真时，将生成的中间代码序列输出到列表文件 listing 中。

例子：（见表 7.3）。

7．3．2 中间代码的构造方法

下面就程序设计语言中的一些典型语句，结合具体的 SNL 语言，讨论代码生成的基本方法。

声明类语句的中间代码

在一般的程序设计语言中，需要包含较多的声明语句，如常量声明、变量声明、类型说明以及过程或函数声明等等。大多数声明语句并不产生中间代码（或目标代码），一般只有过程或函数声明部分需要生成代码。这主要是因为过程声明实际上是声明了一段可以被反复调用的代码，过程声明中不仅包含了一些局部量的声明，还会包含过程体甚至是嵌套的过程声明，为过程声明生成的代码需要记录过程的一些重要信息和相关代码，表 7.2 中间代码及其解释示意图 ，需要生成过程入口的标记语句，需要为过程体生成中间代码；同样，为了标记过程代码的结束，还需要生成过程出口的标记语句。同时，因为中间代码阶段不再进行语义分析，必须把在语义分析阶段得到的有关过程的信息保留在中间代码中，以便留给目标代码生成阶段使

用。例如在语义分析阶段得到的过程的 display 表的偏移量，过程的层数信息等等。

源程序	语法树	四元式代码

program p
 	
type t1 = integer;
var integer v1,v2;
 	
procedure
q(integer i);
var integer a;
begin
a:=i;
write(a)
end
 	
begin
read(v1);
if v1<10
then
v1:=v1+10
else
v1:=v1-10
fi;
q(v1)
end.
	ProK
	PheadK p
	TypeK
	DecK IntegerK t1
	VarK
	DecK IntegerK v1 v2
	ProcDecK q
	DecK value param:IntegerK i
	VarK
	DecK IntegerK a
	StmLk
	StmtK Assign
	ExpK a IdV
	ExpK i IdV
	StmtK Write
	ExpK a IdV
	StmLk
	StmtK Read v1
	StmtK If
	ExpK Op <
	ExpK v1 IdV
	ExpK Const 10
	StmtK Assign
	ExpK v1 IdV
	ExpK Op +
	ExpK v1 IdV
	ExpK Const 10
	StmtK Assign
	ExpK v1 IdV
	ExpK Op -
	ExpK v1 IdV
	ExpK Const 10
	StmtK Call
	ExpK q IdV
	ExpK v1 IdV
		0: PENTRY 1 11
		1:ASSIG I a
		2: WRITE A
		3: ENDPROC
		4:MENTRY 13 9
		5: READ v1
		6:LT v1 10 temp10
		7:JUMP0 temp10 2
		8:ADD v1 10 temp11
		9:ASSIG temp11 v1
		10: JUMP 3
		11: LABEL 2
		12: SUB v1 10 temp12
		13: ASSIG temp12 V1
		14: LABEL 3
		15: VALACT v1 7
		16: CALL 1 9

表 7.3 语法树与对应的中间代码示例图

在 SNL 程序中，只有过程声明需要生成中间代码，而且过程声明允许嵌套，其处理思想如下：

- 为过程 Q 分配新的标号 LabelQ，并将它填入到 Q 的符号表项中；
- 从符号表中读出 Q 的 display 表的偏移量 display_off 和层数信息 levelQ；
- 如果有嵌套的过程声明，则为嵌套过程生成中间代码；
- 为过程生成入口中间代码（PENTRY，LabelQ，sizeQ，levelQ），其中 sizeQ 需要在过程体结束时得到过程活动记录大小之后回填；
- 为过程体生成中间代码；
- 临时变量的空间分配结束后，可以确定过程活动记录的大小 sizeQ，回填给

过程入口代码和过程 Q 的符号表；

- 为过程生成出口中间代码（ENDPROC，NULL，NULL，NULL）。

赋值语句与表达式的中间代码

赋值语句和各种类型的表达式可以说是程序设计语言的核心语句及成分，赋值语句和表达式的中间代码生成是其它语句代码生成的基础。

一般赋值语句的文法描述如下：

Assign ::= V := E

其中 Assign 表示赋值语句，V 表示赋值语句的左部变量，E 为算术表达式、逻辑表达式或其它类型的表达式。

赋值语句直观的语义是将赋值号右边表达式的值赋予左部变量，所生成的中间代码如下：

（ASSIGN，V_arg，E_arg，NULL）

其中 ASSIGN 表示赋值，V_arg 表示左部变量的 ARG 结构，E_arg 表示右部表达式的 ARG 结构。

可见赋值语句的代码生成比较简单，只需要分别得到左部变量和右部表达式的 ARG 结构。若语言中不包含复杂变量，则左部变量的 ARG 结构（地址）可以直接得到；反之，若语言中包含数组和记录类型，则左部变量的 ARG 结构也需要通过生成代码计算得到。以 SNL 语言为例，SNL 可以包含的复杂数据类型有数组类型和记录类型，我们分别对这两种类型的变量的代码生成加以说明。

下标变量 V ::= V_1［E］的中间代码

这里 V_1 必须是数组变量，若用 low 和 size 分别表示数组类型的下界和元素类型的存储大小，用 address(V)表示变量 V 的起始地址，则有：

address(V) = address(V_1) + (E － low) × size

其中 low 和 size 可以通过查 V_1 的符号表得到，根据上述计算公式，下标变量 V_1［E］的中间代码结构设计如下：

- V_1 的中间代码
- E 的中间代码
- (SUB，Earg，lowArg，temp1)
- (MULT，temp1，sizeArg，temp2)
- (AADD，V1arg，temp2， temp3)

其中 Earg 表示表达式计算结果值的 ARG 结构，lowArg 和 sizeArg 分别表示数组下届和元素类型存储大小的值的 ARG 结构，V1arg 表示数组的起始地址的 ARG 结构，temp1，temp2 和 temp3 分别是生成代码所需的临时变量。

域变量 V ::= V_1.id 的中间代码

这里 V_1 必须是记录变量，并且域标识符都存放在记录类型的内部表示域表中。因此 V_1.id 的偏移量可以通过查找域表直接得到并表示成 offArg，变量 V 的地址等于 V_1 的起始地址再加上域的偏移量 offArg。域变量 V_1.id 的中间代码结构设计如下：

- V1 的中间代码
- (AADD ，V1arg， offArg， temp1)

其中 V1arg 表示记录的起始地址的 ARG 结构，offArg 表示域名 id 的偏移量的 ARG 结构， temp1 是生成代码所需的临时变量。

表达式的中间代码

SNL 语言包含了简单的算术表达式和关系表达式，因为这两种表达式的计算是类似的，所以统一起来处理，其一般形式如下：

E ::= E1 op E2 | V | n

也就是表达式可以是常数，变量和复杂表达式。常数类型和简单类型的变量的表达式不需要生成中间代码，表达式的结果分别表现为常数值的 ARG 结构和变量地址的 ARG 结构；复杂变量，如下标变量和域变量需要生成中间代码，其生成方法上面已经做了介绍，这里不再赘述。表达式的代码表现为这两种变量的中间代码；复杂表达式的代码由左部表达式的代码和右部表达式的代码，再加上运算符为 op 的运算代码（op，larg，rarg，temp1）组成，其中 larg 和 rarg 分别表示左部表达式和右部表达式的计算结果。表达式 E ::= E1 op E2 的中间代码结构设计如下：

- E1 的中间代码
- E2 的中间代码
- （op，larg，rarg，temp1）

条件语句的中间代码

一般条件语句的文法表示为：IfS ::= **if** E **then** SL1 **else** SL2 **fi**

其中 SL1 和 SL2 分别表示当条件成立和不成立时执行的语句序列，SL2 可为空。条件语句的语义含义就是：首先计算表达式的值，如果表达式的值为真，则执行 SL1，而后转向整条语句的下一条语句（标号为 outL）；否则转向 SL2 的第一条语句（标号为 elseL）。由此可知，每个条件语句需要引进二个标号 elseL 和 outL，分别用于标记 else 分支的开始和条件语句的结束。根据条件语句的语义，设计条件语句的中间代码结构如下：

- E 的中间代码
- （JUMP0，Earg，ElseLarg，NULL）
- SL1 的中间代码
- （JUMP，OutLarg，NULL，NULL）
- （LABEL，ElseLarg，NULL，NULL）
- SL2 的中间代码
- （LABEL，OutLarg，NULL，NULL）

其中 ElseLarg 和 OutLarg 分别表示 else 分支的开始标号和条件语句出口标号的 ARG 结构，Earg 表示表达式值的 ARG 结构。

循环语句的中间代码

一般循环语句的文法表示为：WhileS ::= **while** E **do** S **endwh**

循环语句的语义含义是：首先计算表达式的值，若表达式的值为假，则转至循环语句的出口 outL；如果表达式的值为真，则执行 S，然后再回到循环的入口语句 startL，重复上述过程。

根据循环语句的语义，可以设计循环语句的中间代码结构如下：

- （LABEL，startLarg，NULL，NULL）
- E 的中间代码
- （JUMP0，Earg，OutLarg，NULL）
- S 的中间代码

- （JUMP，startLarg，NULL，NULL）
- （LABEL，outLarg，NULL，NULL）

其中 startLarg 和 outLarg 分是循环的入口和出口标号的 ARG 结构，Earg 是表达式值的 ARG 结构。

过程调用语句的中间代码

一般过程调用语句的文法表示为：CallS ::= id(E1，E2，……，En)

其中 id 为被调用的过程名，E1，E2，……，En 为实参列表，可为空。过程调用语句主要是完成形参和实参的结合，并转向被调用过程的入口。其中间代码结构如下：

- E1 的中间代码
- E2 的中间代码
- ……
- En 的中间代码
- （VALACT/VARACT,E1arg,Offset1,Size1）
- （VALACT/VARACT,E2arg,Offset2,Size2）
- ……
- （VALACT/VARACT,Enarg,Offsetn,Sizen）
- （CALL,labelArg,NULL,NULL）

其中根据形参是值参还是变参，分别采用 VALACT 和 VARACT 的传值命令；Offseti 是对应形参的偏移量，Sizei 表示第 i 个传参语句要传送的单元个数,在处理结构类型的实参时需要用到此信息。LabelArg 表示被调用过程的入口地址，可以从当前的语法树节点中得到。

读语句的中间代码

一般读语句的文法表示为：ReadS ::= **Read** (id)，读语句的中间代码只有一条代码：

（READC，Varg，NULL，NULL）

其中 Varg 是用标识符 id 的属性信息生成的 ARG 结构。

写语句的中间代码

一般写语句的文法表示为：WriteS ::= **Write**（E），有了表达式的代码生成作基础，写语句的代码生成相对非常简单，其中间代码结构为：

- E 的中间代码
- （WRITEC，Earg，NULL，NULL）

其中 Earg 是表达式 E 的结果 ARG 结构。

7．3．3 从语法树生成四元式

由 7.3.1 节可知，我们将从语法树生成四元式。本小节将依据 7.3.2 节给出的中间代码的生成方法，按照语法树的数据结构，给出如何由语法树得到相应的四元式，即通过对语法树的遍历，每种节点结构应该生成怎样一些四元式。这里仅选取赋值语句节点和 if 语句节点为例给出解释，其它节点略。在 7.3.5 节将通过框图的形式详细地给出每种节点的四元式生成的具体步骤。

1．赋值语句节点

⑴节点结构：

StmtK	AssignK	child[0]	child[1]	NULL

⑵四元式的生成过程：

- 以赋值语句节点的第一个儿子节点 child[0]为参数,调用变量的处理函数 GenVar,生成赋值左部变量的中间代码,并返回变量的 ARG 结构指针 Larg；
- 以赋值语句节点的第二个儿子节点 child[1]为参数,调用表达式的处理函数 GenExpr,生成赋值右部表达式的中间代码,并返回表达式的 ARG 结构指针 Rarg；
- 生成中间代码(ASSIG，Rarg，Larg，NULL)

2．If 语句节点

⑴节点结构：

StmtK	IfK	child[0]	child[1]	child[2]

⑵四元式的生成过程：

- 生成 Else 部分的入口标号的 ARG 结构，用指针 ElseLarg 指向；
- 生成 if 语句的出口标号的 ARGj 结构，用指针 OutLarg 指向；
- 以 if 语句节点的第一个儿子节点 child[0]为参数，调用表达式处理函数 GenExpr，生成表达式的中间代码,并返回表达式的结果 ARG 结构指针 Earg；
- 生成中间代码（JUMP0，Earg，ElseLarg，NULL）；
- 以 if 语句节点的第二个儿子节点 child[1]为参数，调用语句体处理函数 GenBody，生成 then 部分的中间代码；
- 生成中间代码（JUMP，OutLarg，NULL，NULL）
- 生成中间代码（LABEL，ElseLarg，NULL，NULL）
- 以 if 语句节点的第 3 个儿子节点 child[2]为参数，调用语句体处理函数 GenBody，生成 else 部分的中间代码；
- 生成中间代码（LABEL，OutLarg，NULL，NULL）

7．3．4 相关的应用函数

1． 新建一个临时变量

函数声明：ArgRecord *NewTemp (AccessKind access)

算法说明：临时变量的层数为-1，偏移为编号值，访问方式由参数确定。

算法框图：见图 7.2。

图7.2 建立临时变量函数NewTemp的算法框图

2. 新建数值类 ARG 结构

函数声明：ArgRecord *ARGValue(int value)

算法说明: 申请新的 ARG 结构空间,并填充相应的值。

算法框图：见图 7.3。

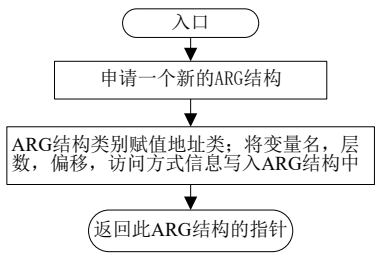


图7.3 建立数值类ARG结构的函数ARGValue的算法框图

3. 产生一个新的标号

函数声明：int NewLabel( )

算法说明：标号值 Label 加 1，并返回新的标号。

算法框图：略。

4. 新建标号类 ARG 结构

函数声明：ArgRecord *ARGLabel(int label)

算法说明：申请新的 ARG 结构空间,并填充相应的值。

算法框图：见图 7.4。

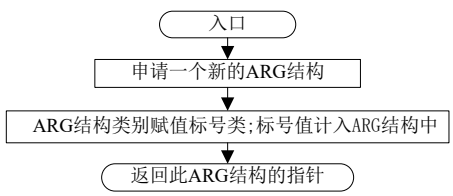


图7.4 新建标号类ARG结构的函数ARGLabel的算法框图

5. 创建地址类 ARG 结构

函数声明：ArgRecord *ARGAddr(char *id ,int level, int off, AccessKind access)

算法说明：申请新的 ARG 结构空间,并填充相应的值。

算法框图：见图 7.5。

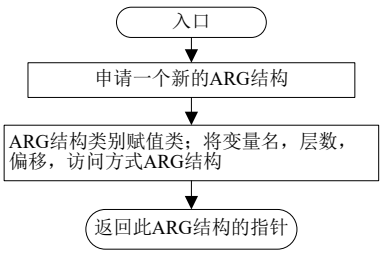


图7.5 创建地址类ARG结构的函数ARGAddr的算法框图

6. 输出中间代码
函数声明：void PrintMidCode(CodeFile *firstCode)
算法说明：从第一条代码开始，循环输出各条代码内容。
算法框图：略。

7. 生成中间代码
函数声明：CodeFile* GenCode (CodeKind codekind , ArgRecord *Arg1 ,ArgRecord *Arg2 ,ArgRecord *Arg3)
算法说明：申请一个中间代码节点空间，填充内容，并链入中间代码表中。
算法框图：见图 7.6。

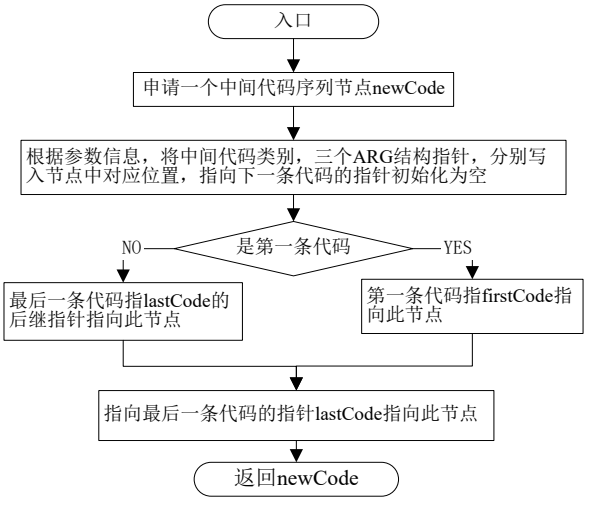


图7.6 生成一条中间代码的函数GenCode的算法框图

7．3．5 中间代码生成程序说明

1. 中间代码生成主函数
函数声明：CodeFile * GenMidCode(TreeNode *t)
算法说明：若有过程声明，调用过程声明的代码声明函数；调用程序体的代码生成函数。

算法框图：见图 7.7。

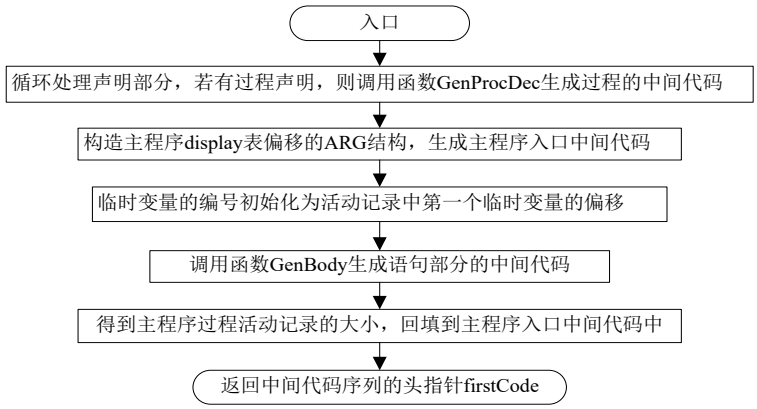


图7.7 中间代码生成主函数GenMidCode的算法框图

2. 过程声明中间代码生成函数：

函数声明：void GenProcDec(TreeNode *t)

算法说明：生成过程入口的中间代码，生成过程体的中间代码，生成过程出口的中间代码。

算法框图：见图 7.8。

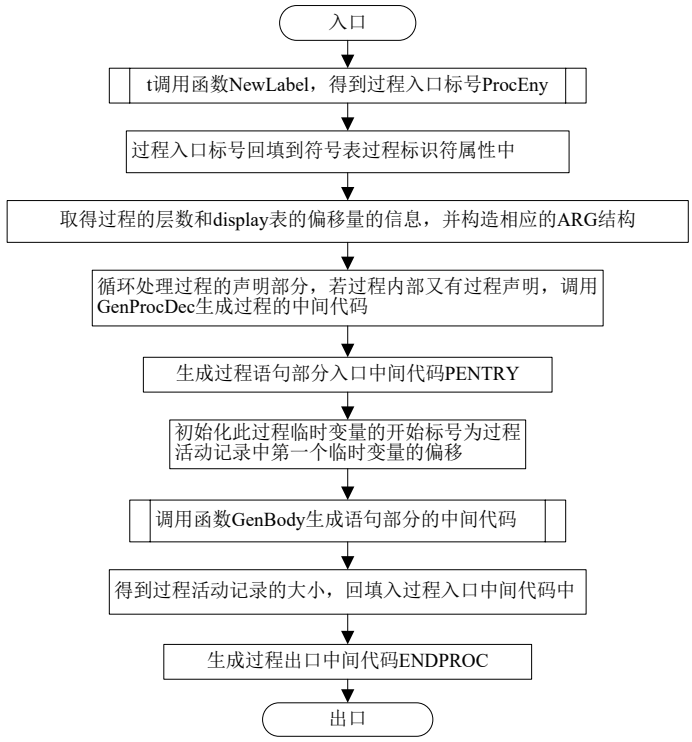


图7.8 过程声明代码生成函数GenProcDec的算法框图

3. 语句体中间代码生成函数
函数声明：void GenBody(TreeNode *t)
算法说明：循环调用语句中间代码生成函数 GenStatement，处理每条语句。
算法框图：略。

4. 语句的中间代码生成函数
函数声明：void GenStatement(TreeNode *t)
算法说明：根据语句的具体类型，分别调用相应的语句处理函数。
算法框图：见图 7.9。

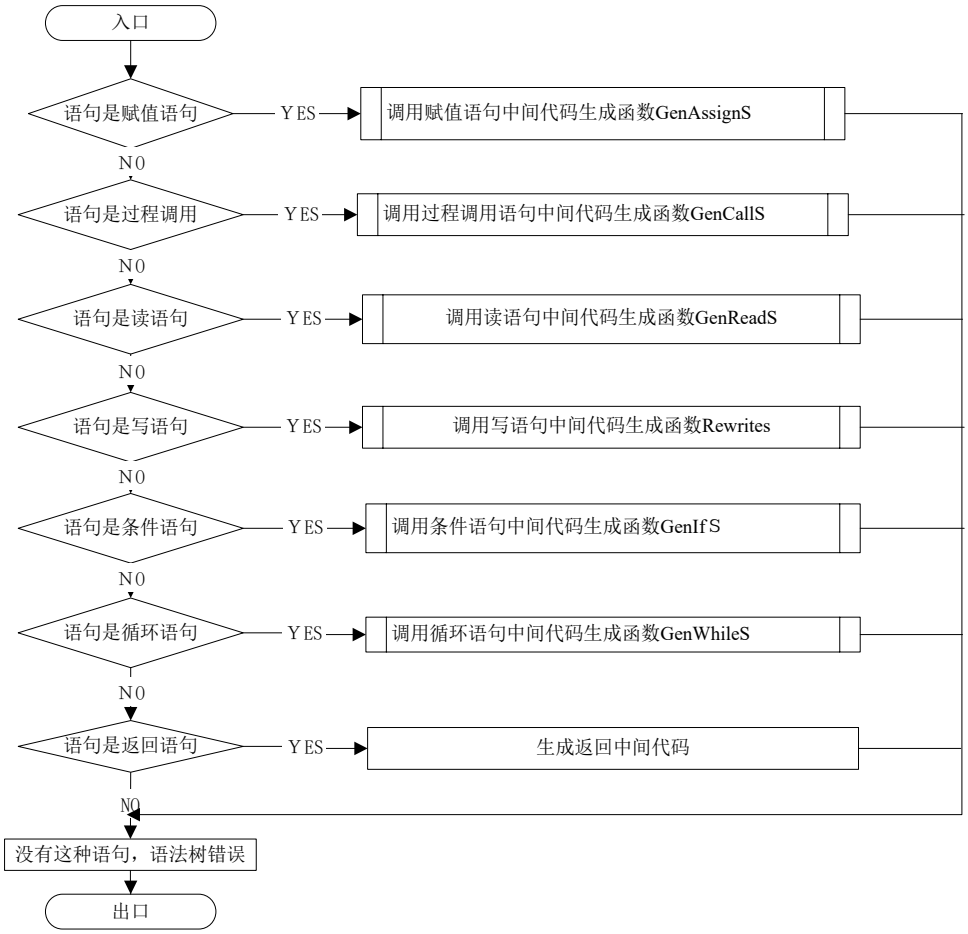


图7.9 语句的中间代码生成函数GenStatement的算法框图

5. 赋值语句中间代码生成函数

函数声明：void GenAssignS(TreeNode *t)

算法说明：处理左部变量，处理右部表达式，生成赋值语句中间代码。

算法框图：见图 7.10。

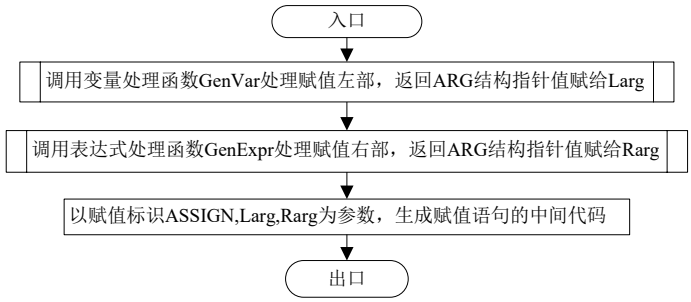


图7.10 赋值语句中间代码生成函数GenAssignS的算法框图

6. 变量中间代码生成函数

函数声明：ArgRecord *GenVar(TreeNode *t)

算法说明：根据变量是标识符，数组变量或者域变量分别处理。

算法框图：见图 7.11。

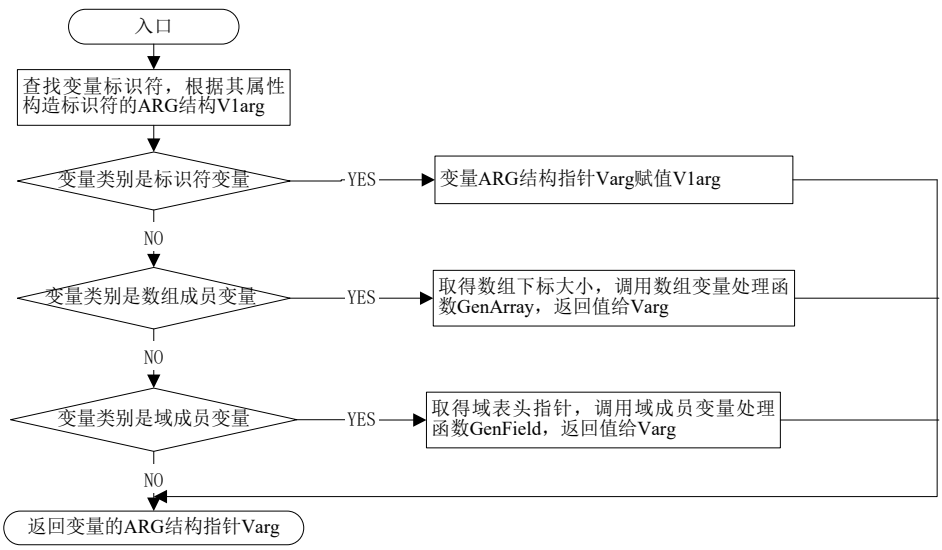


图7.11 变量中间代码生成函数GenVar的算法框图

7. 数组变量的中间代码生成函数

函数声明：ArgRecord *GenArray(ArgRecord *V1arg,TreeNode *t,int low ,int size)

算法说明：由变量的中间代码生成函数或记录域的中间代码生成函数调用。

算法框图：见图 7.12。

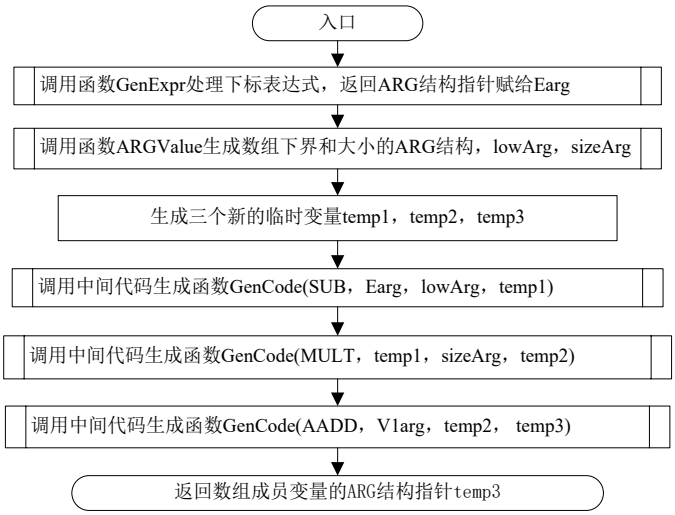


图7.12 数组变量的中间代码生成函数GenArray的算法框图

8. 域成员变量的中间代码生成

函数声明：ArgRecord*GenField(ArgRecord*V1arg,TreeNode*t ,FieldChain*head)

算法说明：由变量的中间代码生成函数调用。

算法框图：见图 7.13。

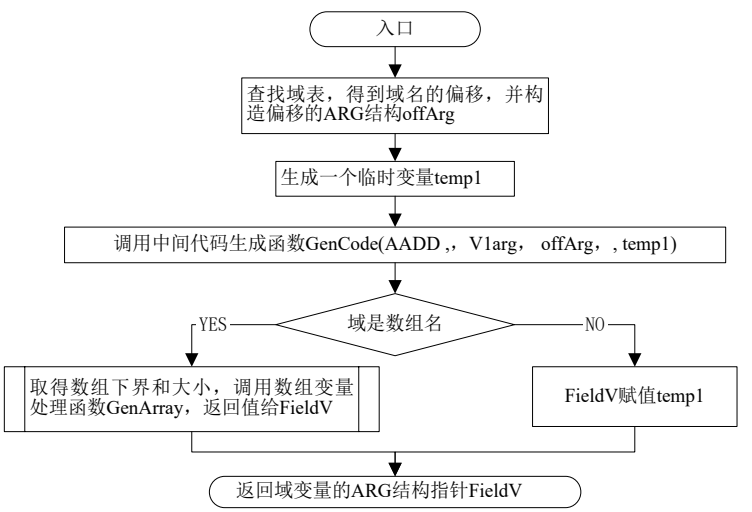


图7.13 域变量的中间代码生成函数GenField的算法框图

9. 表达式的中间代码生成函数

函数声明：ArgRecord *GenExpr(TreeNode *t)

算法说明：根据表达式是变量，常量还是运算表达式分别处理。

算法框图：见图 7.14。

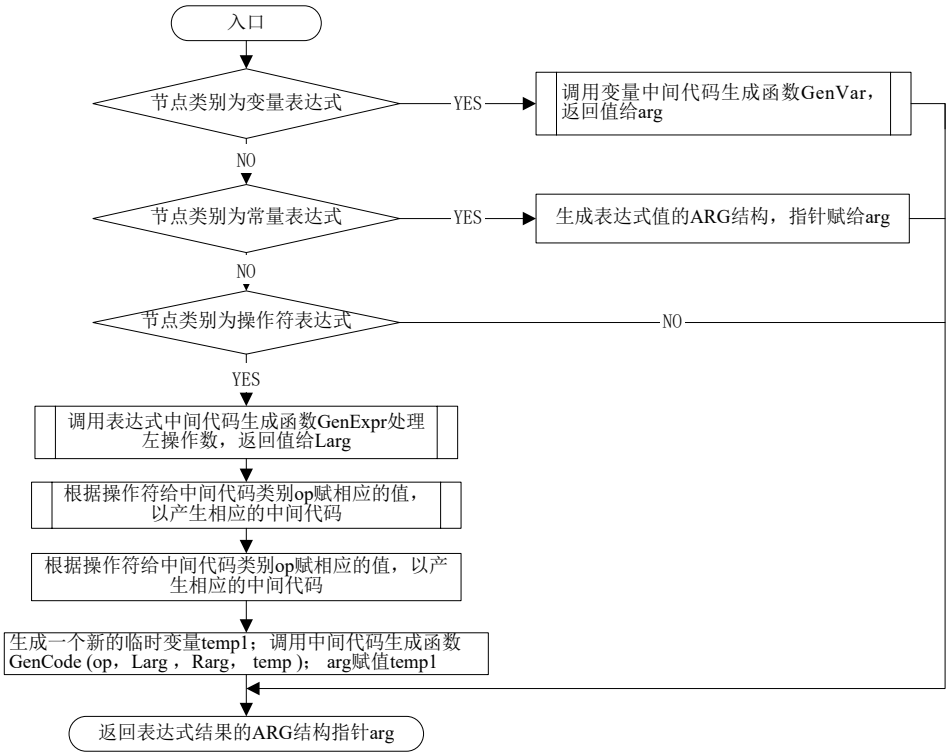


图7.14 表达式的中间代码生成函数GenExpr的算法框图

10．过程调用语句中间代码生成函数

函数声明：void　GenCallS(TreeNode *t)

算法说明：分别调用表达式处理函数处理各个实参，并生成相应的形实参结合中间代码；从符号表的过程标识符属性中，查到入口标号，产生过程调用中间代码。

算法框图：见图 7.15。

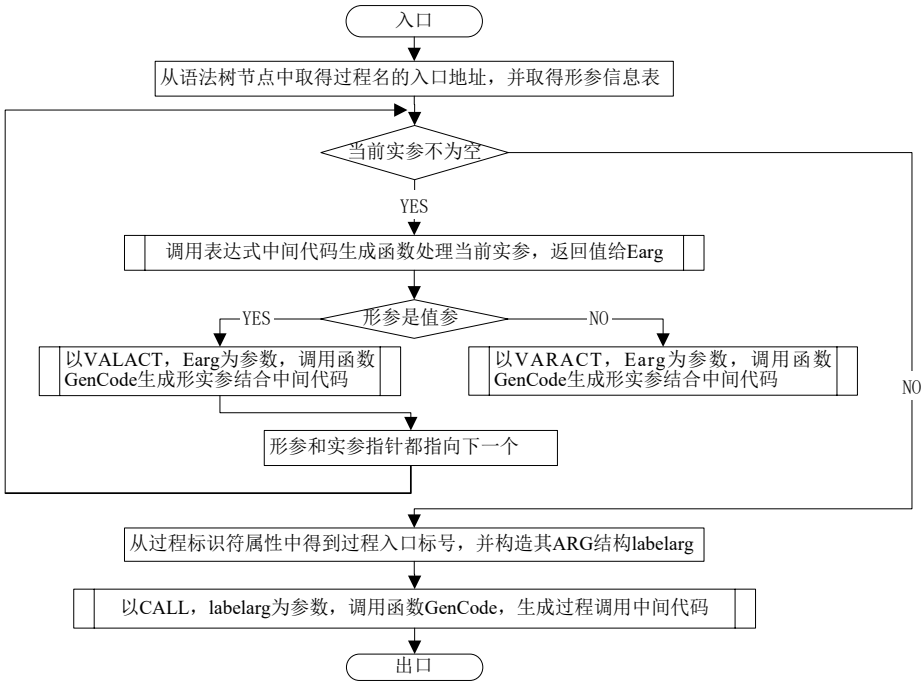


图7.15 过程调用语句中间代码生成函数GenCallS的算法框图

11．读语句中间代码生成函数

函数声明：void GenReadS(TreeNode *t)

算法说明：得到读入变量的 ARG 结构，生成读语句中间代码。

算法框图：见图 7.16。

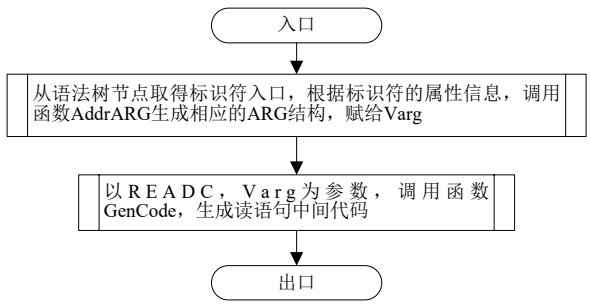


图7.16 读语句中间代码生成函数GenReadS的算法框图

12．条件语句中间代码生成函数

函数声明：void GenIfS(TreeNode *t)

算法说明：生成 else 的入口标号，if 的出口标号和中间代码部分。

算法框图：见图 7.17。

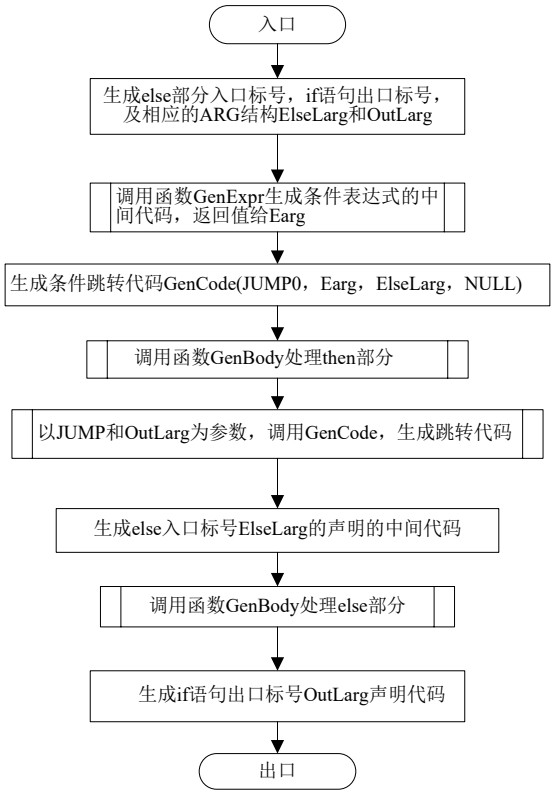


图7.17 条件语句中间代码生成函数GenIfS的算法框图

13．写语句中间代码生成函数

函数声明：void GenWriteS(TreeNode *t)

算法说明：调用表达式的中间代码生成函数，并产生写语句的中间代码。

算法框图：见图 7.18。

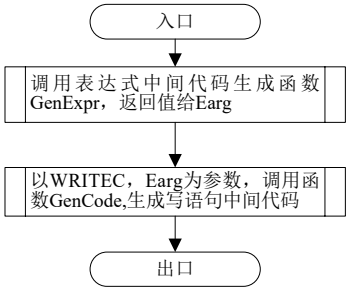


图7.18 写语句中间代码生成函数GenWriteS的算法框图

14．循环语句中间代码生成函数

函数声明：void GenWhileS(TreeNode *t)

算法说明：将循环入口和出口用不同的中间代码标志，是实现循环不变式外提的需要。

算法框图：见图 7.19。

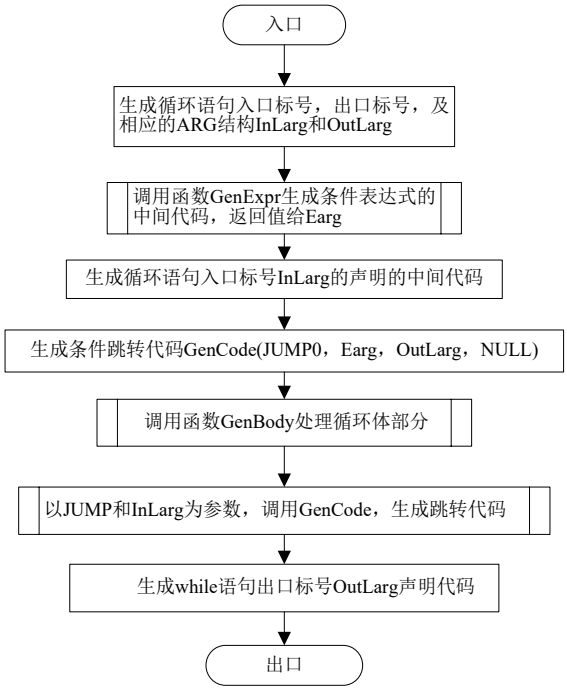


图7.19 循环语句中间代码生成函数GenWhileS的算法框图

第八章 中间代码优化

8．1 中间代码优化简介

所谓优化，是指在不改变程序运行结果的前提下，对被编译的程序进行重排、删除、合并等变换，使之生成更高效的目标代码。优化的目的是要使存储空间减小而又尽可能使运行速度提高，常常更关注后者。

一般来讲，对于同一个源程序，进行优化与不进行优化所产生的目标程序的效率可以相差很大，所以优化与优化的效果是编译器的一个重要性能指标。当然，在实施优化时还必须考虑实施优化的代价，否则同样会影响编译器的质量。

当然，为了改进、提高程序的运行效率，途径可以有很多，例如可以由用户通过改进算法、利用系统提供的程序库等对源程序进行等价变换，从而实现源程序级的优化；利用编译器的前端对中间代码进行优化；由编译器的后端对目标代码进行优化等等。但是编译阶段的优化是其它途径不可替代的。因为用户的水平参差不齐，无法苛求他们对源语言的熟练程度和编程技巧等。

中间代码的优化与具体的目标机无关，是独立于目标机的优化，因此更具有通用性和可移植性，是很重要的优化途径。中间代码优化的主要对象是深层循环和下标变量地址的计算。中间代码优化要注意的问题是，绝对要保证正确性，即对于相同的输入，要输出与源程序同样的结果。

8．1．1 优化种类介绍

从优化所涉及的源程序的范围而言，可以将优化分为全局优化和局部优化。如果优化是在整个程序上实施的，则称之为全局优化，否则称之为局部优化。局部优化又可以分为基本块上的优化和循环上的优化。基本块上的优化是在基本块上进行的优化，循环优化是在程序中隐式或显式的循环体范围内的优化。常见的中间代码优化种类如图 8.1 所示：

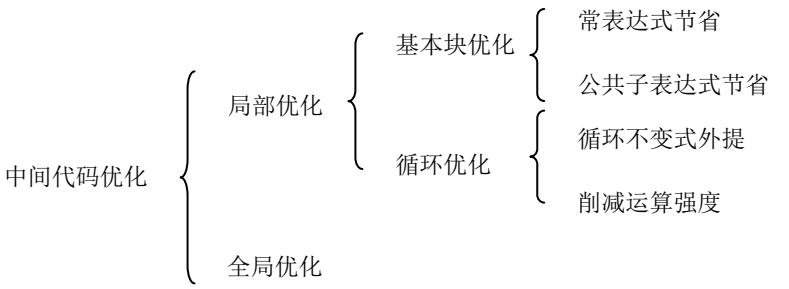


图 8.1 常见中间代码优化的种类

其中常量表达式节省优化是把程序中静态可计算的部分计算出来；公共子表达

式节省优化是消除代码中的重复运算；不变表达式的循环外提可以把循环体中不变的表达式提到循环体外，从而避免了重复计算；削减运算强度是把强度大的运算(如乘)等价的转换为强度小的运算(如加)。做全局优化通常需要进行控制流分析和数据流分析，由于从大范围求得信息，因此优化率很高，但是比较复杂，开销也很大，如对编译没有特殊要求，一般不做此种优化。

从优化相对于编译程序的逻辑功能实现的不同阶段及与目标机的关系而言，可以把优化分为中间代码级的优化和目标代码级的优化，或与机器无关的优化和与机器有关的优化。与机器有关的优化是在目标代码机上进行的优化，例如寄存器优化，窥孔优化，并行分支的优化等。与机器无关的优化可以在源程序级或中间代码级上进行，是在目标代码生成前进行的优化，其特点是不依赖于具体的目标机环境，这种优化更具普遍性，一般多在中间代码级上进行。这一章里，我们将给出基于中间代码的常量表达式、公共表达式和循环不变式的局部优化的实现程序和说明。

8．1．2 基本块的划分

由于常量表达式优化和公共表达式优化的基本单位是基本块，因此，先介绍一下基本块的概念。

基本块是指程序中一段顺序执行的语句序列，它只有一个入口和一个出口。也就是说，进入一个基本块，必须且只能通过入口语句(中间代码)进入；同样退出一个基本块，必须且只能通过出口语句(中间代码)退出。换句话说，一个基本块内的语句要么全执行，要么全不执行，不能只执行一部分。也就是说，基本块内的语句是顺序执行的，没有转进转出，分叉汇合等问题。

常用如下方法划分基本块：

- 遇到声明类代码 LABEL ,ENTRY ,WHILESTART 则从这条标号声明代码开始，进入一个新的基本块；
- 遇到跳转类代码 JUMP0,JUMP,RETURN,ENDPROC,ENDWHILE 则表示当前基本块的结束，从这条代码的下一条代码开始，进入下一个基本块；
- 遇到变参的形实参结合代码，由于过程中可能改变变参的值，为了保证正确性，将与此形实参结合对应的过程语句，作为当前基本块的结束，即从这条代码的下一条代码开始，进入下一个基本块。

注，变参赋值情况的解释示例如图 8.2 所示：

```
program   a
var   integer   r , y ;
procedure   P ( var   x)
   begin
        x := 10
   end
begin
      y := 1 ;
      p ( y );
      r := y
end.
```

图 8.2　　一个含有变参赋值的程序例

在上面程序中，由于 y 作为过程 p 的变参，所以不能保证在过程 p 内部不改变 y 的值。如上例过程内改变了 y 的值，若仍作为一个基本块处理优化后，r 的取值错误,不能保证程序的正确性。

8．2 常量表达式优化

8．2．1 常量表达式优化的原理

常表达式节省有的书上也叫做常量和并与传播，是基本块上的一种优化。所谓常表达式是指在任何时候都取固定常数值的表达式。显然，对于这类表达式如果能够避免反复计算，则会大大提高所生成代码的效率。图 8.3 给出了优化前和经过常表达式节省优化后的代码(为清晰起见，采用源语言表示优化后的代码)，说明了常表达式节省的概念。

优化前代码	优化后代码

a := 1;
b := a + 1;
c := b + 5;
	a := 1;
	b := 2;
	c := 7;

图 8.3 常表达式优化示例

常表达式节省的基本思想是：如果一个四元式的两个运算分量都是取常数值，则由编译器将其值计算出来，以所求得的值替换原来的运算，并删除当前四元式。

为了实现常表达式节省，我们需要定义一个常数定值表，其元素是二元组(var,val),其中 var 是变量名，val 是变量名所对应的常数。如果在常量定值表中有(Y,5)，表示当前的 Y 一定取值 5，并且在 Y 未被重新赋值以前，后面出现的 Y 都可替换成 5(称为值替换或值传播)；如果一个四元式的两个分量都在常量定值表中，即为常数值，则计算当前四元式的结果值，将表示结果的临时变量和结果值填入常量定值表，并删除当前四元式；如果变量 Y 被赋值为非常数值，则从所构造的常量定值表中删除变量 Y 的登记项，以表示变量 Y 不取常数值。

概括来讲，关于常量定值表的操作分为以下几种：

- 填表：在下面两种情况下填表：
  第一种情况： 当前四元式是(:=, X, T, NULL)，若 T 为已知量，则将(X , T.VAL)填入表中；
  第二种情况：当前四元式是(OP,X1,X2,T)， 若 X1,X2 都为已为知，则将(T,X1.VAL OP X2.VAL) 填入表中。
- 删表：当前四元式是(:=, X , T , NULL)，若 T 为未知量，则将(X,-)从表中删除，即 X 的原定值已经改变。

- 使用：当前四元式是(OP ,X1,X2,T)，若 X1,X2 都在表中，则用 X1,X2 在表中的对应值替换 X1,X2.即变为(OP, X1.VAL , X2.VAL,T)。

常量表达式的局部优化算法如下：

- 调用函数 DivBaseBlock 划分基本块；
- 循环处理各个基本块：
  (1) 在基本块入口处置 ConsDef 表为空；
  (2) 读当前中间代码 tuple；
  (3) 对 tuple 中的分量进行值替换；
    a．对于运算代码和关系比较代码，替换两个运算分量；
    b．对于赋值语句、条件跳转语句、输出语句，替换第一个 ARG 结构；
    c．对于地址加运算 AADD，替换第二个 ARG 结构；
    d．其他代码，不做任何替换；
  (4) tuple 是运算代码和关系比较代码(w, A ,B , t)情形：若 A 和 B 都是常数，则计算 A w B 的值 v，并在 ConsDef 表中填入(t ,v)，同时删除本 tuple。其中判断运算分量是否是常数的方法为：若分量是常数 ARG 结构，则当前是常数；若分量是变量，且在常量定值表中有定值，则当前也为常数；
  (5) 若 tuple 是赋值代码(ASSIG ,A ,B)情形：若 A 是常数，则把(B,A)填入 ConsDef 表中(具体做法：若已有 B 项，则只需改值，否则，增加一项)；否则，从 ConsDef 删除 B 的登记项(具体做法：若没有 B 项，则什么也不做，否则，删掉一项)；
  (6) 转(2)，直到基本块结束；
  (7) 基本块结束，释放占用的常量定值表空间。

8．2．2 常量表达式节省的实现

1．输入输出

为了让读者更清楚各种中间代码的优化方法，我们将各种优化方法分别作为编译的一遍扫描过程。因此常量表达式节省方法输入的是四元式中间代码，输出的是经过常量表达式优化后的中间代码。

表 8.1 给出了一个源程序优化前和常量表达式优化后的代码的对比。

源程序	优化前的中间代码	常量表达式优化后的中间代码
program aa **type** t1=integer;	**0: PENTRY 1 13 1**	**0: PENTRY 1 13 1**
	1: ADD x 1 temp11	**1: ADD x 1 temp11**
	2: ASSIG temp11 x	**2: ASSIG temp11 x**

var
t1 v1,v2;
integer a,b,c;
char o1;
 	
procedure
p(integer x; **var** integer y);
begin
x:=x+1 ;
y:=y+1
end
 	
begin
v1:=10;
v2:=26;
a:=1;
b:=a+1;
a:=c;
c:=b+5;
p(a,b);
if v2< 3*(4+5)
then v2:= v2+1
else
write(o1)
fi
end.
	3: ADD y 1 temp12	**3: ADD y 1 temp12**

	4: ASSIG temp12 y	**4: ASSIG temp12 y**
	5: ENDPROC	**5: ENDPROC**
	6: MENTRY 20 13	**6: MENTRY 20 13**
	7: ASSIG 10 v1	**7: ASSIG 10 v1**
	8: ASSIG 26 v2	**8: ASSIG 26 v2**
	9: ASSIG 1 a	**9: ASSIG 1 a**
	10: ADD a 1 temp14	**10: ASSIG 2 b**
	11: ASSIG temp14 b	**11: ASSIG c a**
	12: ASSIG c a	**12: ASSIG 7 c**
	13: ADD b 5 temp15	**13: VALACT a 7**
	14: ASSIG temp15 c	**14: VARACT b 8**
	15: VALACT a 7	**15: CALL 1 9**
	16: VARACT b 8	**16: LT v2 27 temp18**
	17: CALL 1 9	**17: JUMP0 temp18 2**
	18: ADD 4 5 temp16	**18: ADD v2 1 temp19**
	19: MULT 3 temp16 temp17	**19: ASSIG temp19 v2**
	20: LT v2 temp17 temp18	**20: JUMP 3**
	21: JUMP0 temp18 2	**21: LABEL 2**
	22: ADD v2 1 temp19	**22: WRITE o1**
	23: ASSIG temp19 v2	**23: LABEL 3**
	24: JUMP 3	
	25: LABEL 2	
	26: WRITE o1	
	27: LABEL 3	

表 8.1 常量表达式优化结果示例表

2．数据结构

对源程序进行常量表达式节省的优化需要用到常量定值表 ConstDefT，常量定值表用于记录当前可用的常量定值。我们将常量定值表定义成一个双向链表，其结构图示如下：

former	variable	constValue	next

成员 former　　指向前一个常量定值节点；
成员 variable　　变量的 ARG 结构指针；
成员 constValue　记录变量当前定值；
成员 next　　指向下一个常量定值节点。

对常量定值表的操作包括添加、修改和删除，与普通的双向链表相同，这里就不再一一叙述了。

3．常量表达式优化程序说明

(1) 常表达式优化主函数
函数声明：CodeFile　*ConstOptimize( )
算法说明：循环对各个基本块进行常表达式优化。
算法框图：见图 8.4。

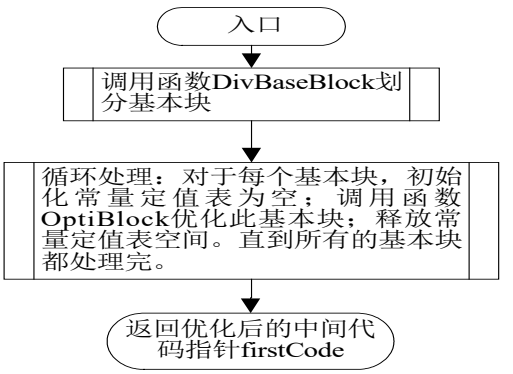


图8.4 常表达式优化主函数ConstOptimize的算法框图

(2) 基本块内的常表达式优化函数
函数声明：void OptiBlock(int　i)
算法说明：循环处理每条代码，直到当前基本块结束。
算法框图：见图 8.5。

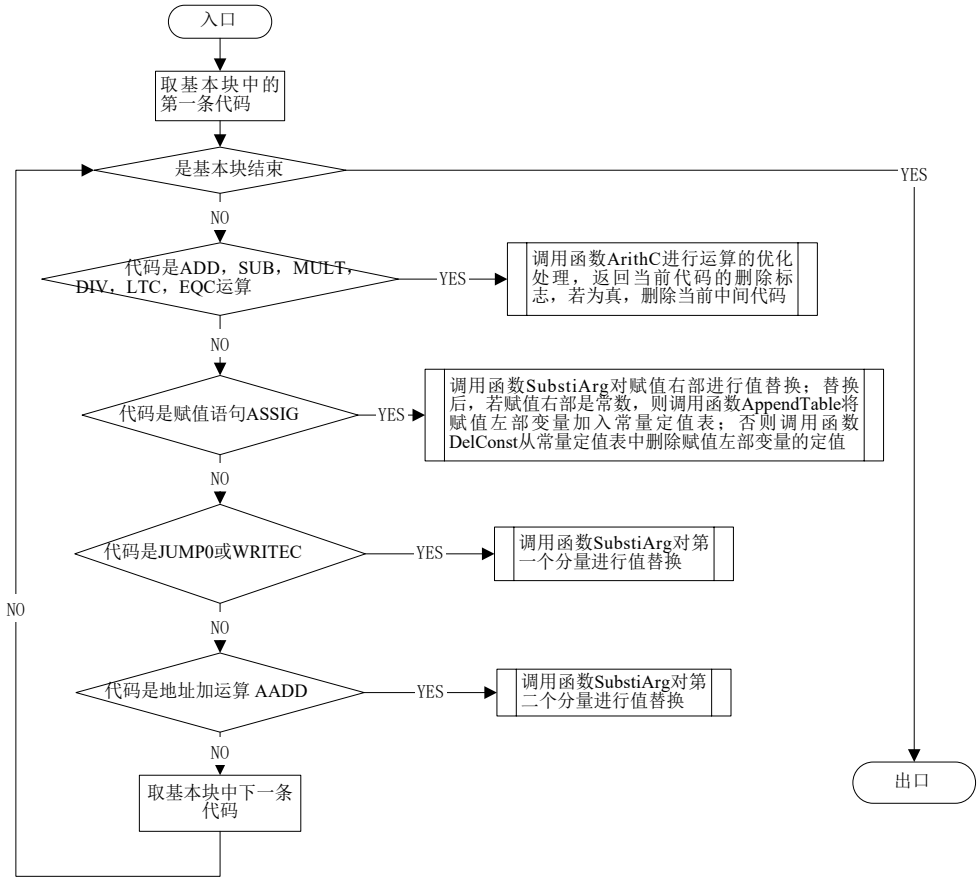


图8.5 基本块内的常表达式优化函数OptiBlock的算法框图

(3) 算术和比较运算的优化处理

函数声明：bool ArithC (CodeFile *code)

算法说明：对运算分量 1 和运算分量 2 进行值替换，若都是常数，将结果写入常量定值表，并置四元式删除标志为真。

算法框图：见图 8.6。

(4) 值替换函数：

函数声明：void SubstiArg(CodeFile *code,int i)

算法说明：参数 i 指出对中间代码的哪个 ARG 结构进行替换。

算法框图：见图 8.7。

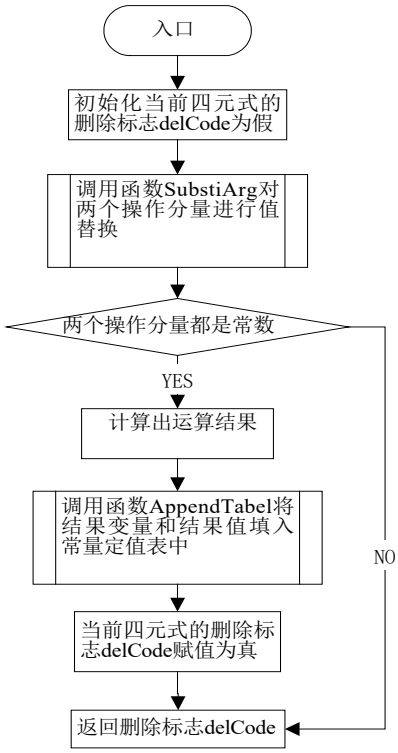


图8.6 算术和比较运算的优化函数ArithC的算法框图

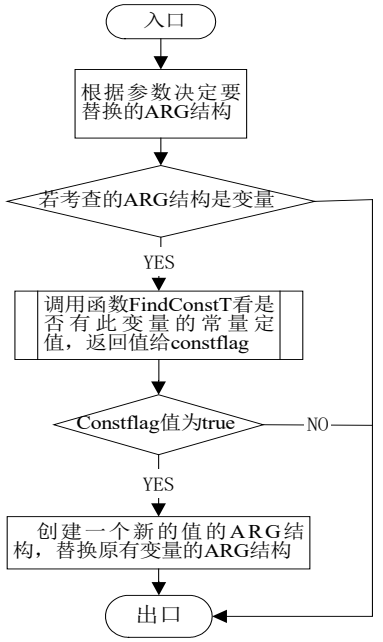


图8.7 值替换函数SubstiArg的算法框图

4．应用函数

(1) 变量定值表查找函数：
函数声明：bool FindConstT(ArgRecord *arg, ConstDefT **Entry)
算法说明：从表头开始查找，依次将要查找的变量和各项相比较，若层数和偏移都相等，则找到，返回值为真，并将入口赋值给参数 EnTry；否则，返回值为假，入口为空。
算法框图：略。

(2) 变量定值表添加函数：
函数声明：void AppendTable(ArgRecord *arg , int result)
算法说明：首先调用函数 FindConstT 查找常量定值表，若已经在表中，则只需改变此变量的定值；否则，新建常量定值表中一项，填写变量和定值，并把此项链入表尾。
算法框图：略。

(3) 删除常量定值表中的一项：
函数声明：void DelConst(ArgRecord *arg)
算法说明：首先调用函数 FindConstT 查找要删除的变量，若存在表中，则删除此项，否则，直接结束。
算法框图：略。

8．3 公共表达式节省方法

8．3．1 公共表达式优化原理

公共表达式节省是基本块上的另一种优化，它的目标是尽可能地消除等价表达式值的重复计算。若在基本块中有：

……

di：(op1，a，b，ti)

……

dk：(op2，a′，b′，tk)

……

如果在 di 和 dk 之间的中间代码都不改变 a 和 b 的值，则称 di 所定义的表达式 a op1 b 在 dk 处可用。如果有 op1 = op2，a = a′，b = b′，且 a op1 b 在 dk 处可用，则称 dk 所定义的表达式 a op1 b 为公共子表达式。这样的公共子表达式是可以节省的，记为 ECC。公共表达式的优化就是 ECC 的节省。

在进行公共表达式节省时，我们要注意以下两点：

1．形式相同的表达式不一定被优化；

例如：① D := A * B；

② A := T；

③ C := A * B

上面①中的表达式 A * B 和③中的表达式 A * B 形式上是相同的，但是因为在②中对 A 进行了重新赋值，表达式 A * B 不满足 ECC 的条件，所以③中的表达式 A*B 不能被节省。

2．形式不同的表达式也可能优化。

例如：① D := A * B；

② C := A；

③ D := C * B

①中的表达式 A * B 和③中的表达式 C * B 尽管从形式上是不同的，但是由于在②中 C 取得了与 A 相同的值，所以③中的表达式 C*B 也可以被节省。

我们对 SNL 源程序采用基于值编码的公共表达式局部优化。值编码方法的主要思想是： 对中间代码中出现的每个分量，确定一个编码(值编码)，使得具有相同编码的分量等价(反之不然)。具体操作步骤为：

a) 遇到变量、临时变量、常量的第一次出现时，给它们分一个编码。

b) 遇到 A:=B 时，令 A 取和 B 相同的值编码。

c) 遇到复杂变量，将产生间接临时变量，临时变量存的是一个存储单元的地址。计算过程中有时用到地址，也有时用到值，因此需分配两个编码，一个是地址码，一个是值码。

当两个四元式操作符相同，且操作分量的编码相同时，我们把这两个表达式称为公共表达式，可以进行优化。为了实现这个目标，我们要用到下面三张表：

i. 编码表 ValuNum：用于记录常量和变量的编码；

ii. 可用表达式代码表 UsableExpr：用来表示哪些中间代码是当前可用的；

iii. 临时变量的等价表 TempEqua：用来记录哪些临时变量是等价的。

此外，对于每条代码(ω，A，B，t)要产生相应的映像码，映像码的定义为：若用 u(X)表示 X 的值编码，则(ω，A，B，t)代码对应的映像码代码为(u(A)，u(B)，t)。

下面给出基于值编码的公共表达式节省算法：(newVN 表示新的值编码)

1．调用函数 DivBaseBlock 划分基本块；
2．循环处理各个基本块：
- 在基本块的入口处置 ValueNum 表，UsableExpr 表，TempEqua 表为空；
- 逐条扫描基本块的中间代码；
- 对于当前中间代码进行等价替换：即若是临时变量，则查找 TempEqua 表，

对分量进行替换；

- 设替换后的中间代码为 Tuple = dk：(w，A， B， t) 型，则
  - 若 A 是首次出现，则把(A，newVN)填入 ValuNum 表；
  - 若 B 是首次出现，则把(B，newVN)填入 ValuNum 表；
    - 判断是否首次出现的办法是：查找值编码表 ValuNum，若没有此变量或值的编码，则为首次出现。newVN 通过调用一个函数取得，这个函数每被调用一次，编码值就加 1；填表时对于间接变量，填写两个编码。
  - 若存在 di∈UsableExpr 使得 dk 是 di 的 ECC 则删除当前代码 Tuple，并将(t ，result i)填入 TempEqua 表中；否则(t，newVN)填入 ValuNum 表；构造 dk 的映像码。把 dk 加入到 UsableExpr 表中；
    - 判断 dk 是否是 di 的 ECC 的办法是：取得 UsableExpr 表中的一项，得到中间代码地址表，比较运算符是否相同，若相同，则取得中间代码的映像码，判断分量的编码是否相同，根据操作是 AADD 还是 ADD 等算术操作决定比较的是值码还是地址码，若相同，则等价，其他情况不等价。这里考虑了可交换运算。
- 设替换后的中间代码为 Tuple = (ASSIG，A, B , _) 型：则
  - 若 A 是首次出现，则把(A，newVN) 填入 ValuNum 表中；
  - 若 A 为间接临时变量，则令 B 的值编码等于 A 的地址码，否则，令 B 的值编码等于 A 的值编码，即把(B， A 的编码)填入 ValuNum 表中；
  - 不优化本代码，从 UsableExpr 删除所有用到 B 的值编码的中间代码。
    - 做法：查找 UsableExpr 表，对每一项，得到中间代码地址，若中间代码不是 AADD，且中间代码的每一个 ARG 中，有和 B 重名，则删除这一项。

8．3．2 公共表达式节省的实现

1．输入输出

公共表达式节省优化作为独立的一遍，其输入是经过常量表达式优化或者没有经过常量表达式优化的中间代码，输出是经过公共表达式优化后的中间代码。

表 8.2 给出了公共表达式节省优化前(进行了常表达式优化)和优化后的代码对比：

源程序	公共表达式优化前 (注：做了常表达式优化)	公共表达式优化后的中间代码
program pp	0: MENTRY 44 22	0: MENTRY 44 22
	1: ASSIG 5 i	1: ASSIG 5 i
	2: ASSIG 3 j	2: ASSIG 3 j

var
integer
v,x,i,j,y;
array[1..10]of
integer a;
begin
i:=5;
j:=3;
read(v);
a[i]:= v;
x:=a[i]+j;
write(x);
a[i]:=a[i]+j;
write(a[i]);
y:=a[i]+j;
write(y)
end.
	3: READ v	3: READ v

	4: AADD a 4 temp25	4: AADD a 4 temp25
	5: ASSIG v temp25	5: ASSIG v temp25
	6: AADD a 4 temp28	**6: ADD temp25 3 temp29**
	7: ADD temp28 3 temp29	7: ASSIG temp29 x
	8: ASSIG temp29 x	8: WRITE x
	9: WRITE x	**9: ASSIG temp29 temp25**
	10: AADD a 4 temp32	**10: WRITE temp25**
	11: AADD a 4 temp35	**11: ADD temp25 3 temp43**
	12: ADD temp35 3 temp36	12: ASSIG temp43 y
	13: ASSIG temp36 temp32	13: WRITE y
	14: AADD a 4 temp39	
	15: WRITE temp39	
	16: AADD a 4 temp42	
	17: ADD temp42 3 temp43	
	18: ASSIG temp43 y	
	19: WRITE y	

表 8.2　公共表达式优化结果示例表

2．数据结构

（1）编码表 ValuNum：记录常量和变量的编码；

<table>
<tr><td rowspan="3">Arg</td><td rowspan="3">access</td><td colspan="3">CodeInfo</td><td rowspan="3">next</td></tr>
<tr><td rowspan="2">valueCode</td><td colspan="2">twoCode</td></tr>
<tr><td>Valuecode</td><td>addrcode</td></tr>
</table>

成员 arg　　指向变量或者常量的 ARG 结构，以指明是哪个常量或变量；
成员 access　　记录变量的访问方式，直接访问时取值 dir，间接访问时取值 indir；
成员 codeInfo　　记录编码的信息；
　　codeInfo 成员 valueCode　　记录常量和直接访问方式的变量的值编码；
　　codeInfo 成员 twoCode　　记录间接访问变量的两种编码；
　　　　twoCode 成员 valuecode　　变量的值编码；
　　　　twoCode 成员 addrcode　　变量的地址码；
成员 next　　指向编码表中下一项。

(2) 可用表达式代码表 UsableExpr：用来表示哪些中间代码是当前可用的；

Code	mirrorC	Next

成员 code　　指向四元式中间代码；
成员 mirrorC　　指向此中间代码对应的映像码；映像码用来比较某个四元式是否可换；
成员 next　　指向可用表达式代码表中的下一项。

映像码结构：

arg1 编码	arg2 编码	arg3 编码
op1	op2	Result

(3) 临时变量的等价表 TempEqua：用来记录哪些临时变量是等价的。

arg1	arg2	Next

成员 arg1　指向被替换的临时变量的 ARG 结构；
成员 arg2　指向用于替换的临时变量的 ARG 结构；
成员 next　指向临时变量等价表中的下一项。

3．公共表达式优化程序说明

(1) 公共表达式优化主函数
函数声明：CodeFile　*ECCsave()
算法说明：循环对各个基本块进行公共表达式优化。
算法框图：见图 8.8。

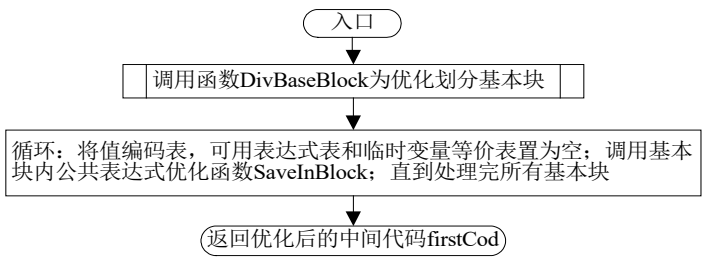


图8.8 公共表达式优化主函数ECCsave的算法框图

(2) 基本块内公共表达式优化函数
函数声明：void　SaveInBlock(int i)
算法说明：循环处理基本块中的各条语句，根据不同的类别进行相应的替换。
算法框图：见图 8.9。

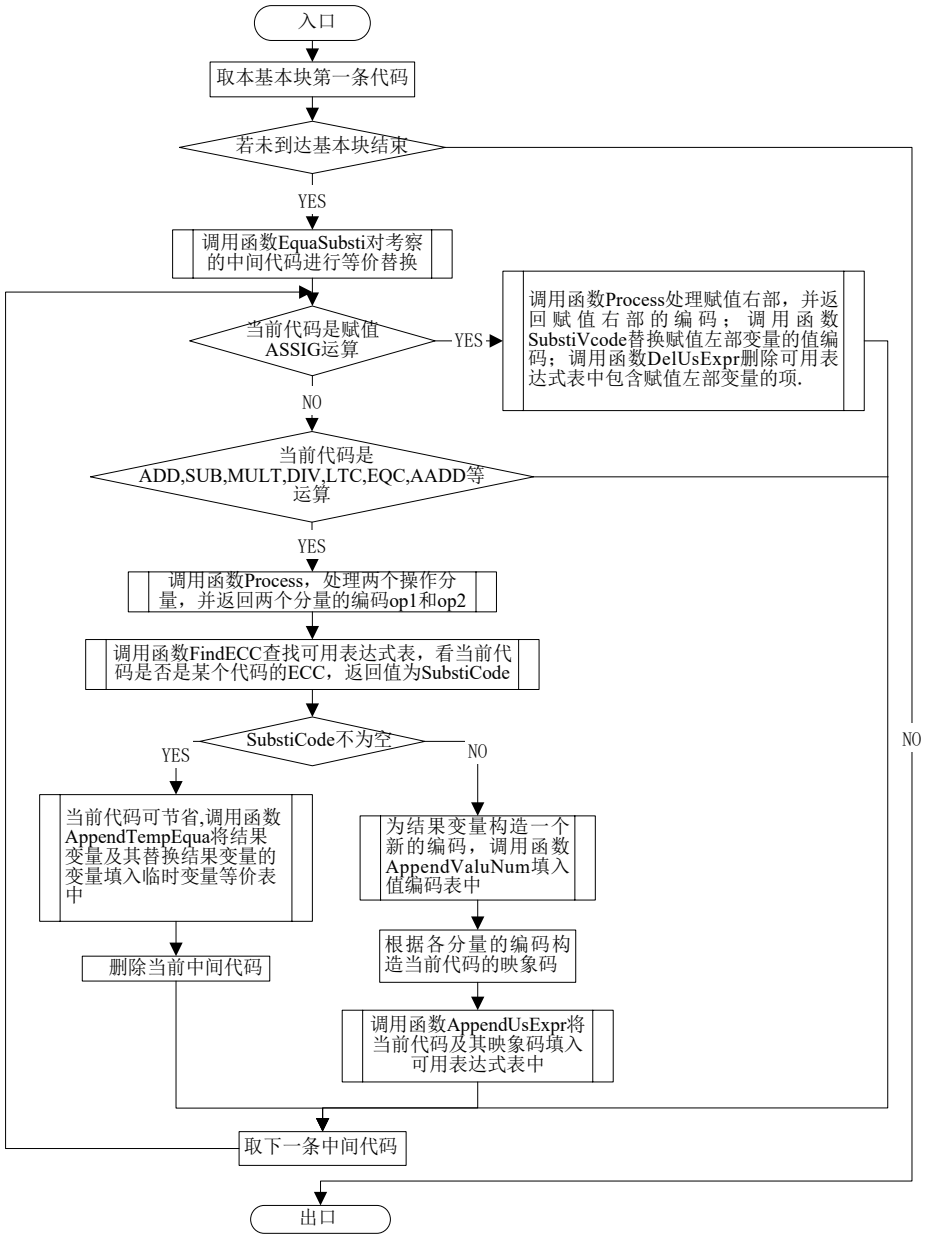


图8.9 基本块内的公共表达式优化函数SaveInBlock的算法框图

(3) 等价替换函数

函数声明：void EquaSubsti(CodeFile *code)

算法说明：若操作数是临时变量,且存在于临时变量等价表中，则替换。

算法框图：见图 8.10。

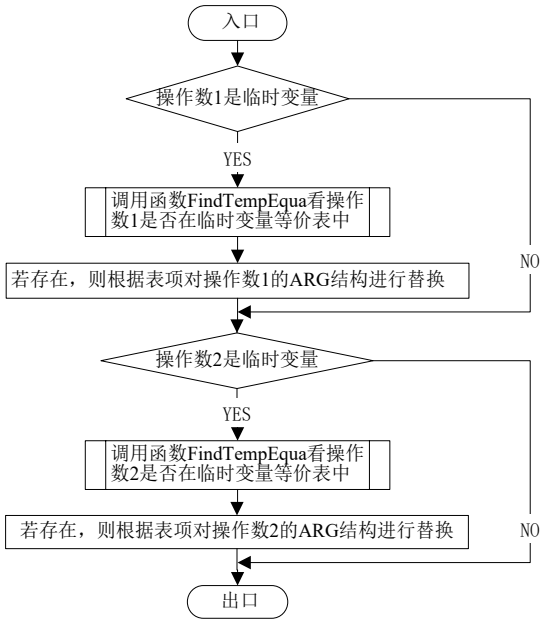


图8.10 等价替换函数EquaSubsti的算法框图

(4) 临时变量等价表查找函数

函数声明：TempEqua *FindTempEqua(ArgRecord *arg)

算法说明：从表头开始查找，依次将要查找的临时变量指针和表中各项要被替换的临时变量指针比较，若相等，则此临时变量可以被替换，返回此表项的指针；若都不相等返回指针值为空。

算法框图：略。

(5) 值编码表查找函数

函数声明：ValuNum *SearchValuNum(ArgRecord *arg)

算法说明：从表头开始查找，对于每一项，调用函数 IsEqual 判断两个变量的 ARG 结构是否相同；若相同，则返回此变量在值编码表的入口指针；若都不相同，则没找到，返回指针值为空。

算法框图：略。

(6) 运算操作分量的处理函数

函数声明：int Process(CodeFile *code , int i)

算法说明：若首次出现，则分配新编码，填入编码表中，返回值取这个新的编码；否则，根据是否间接变量，返回相应的值编码或地址码。

算法框图：见图 8.11。

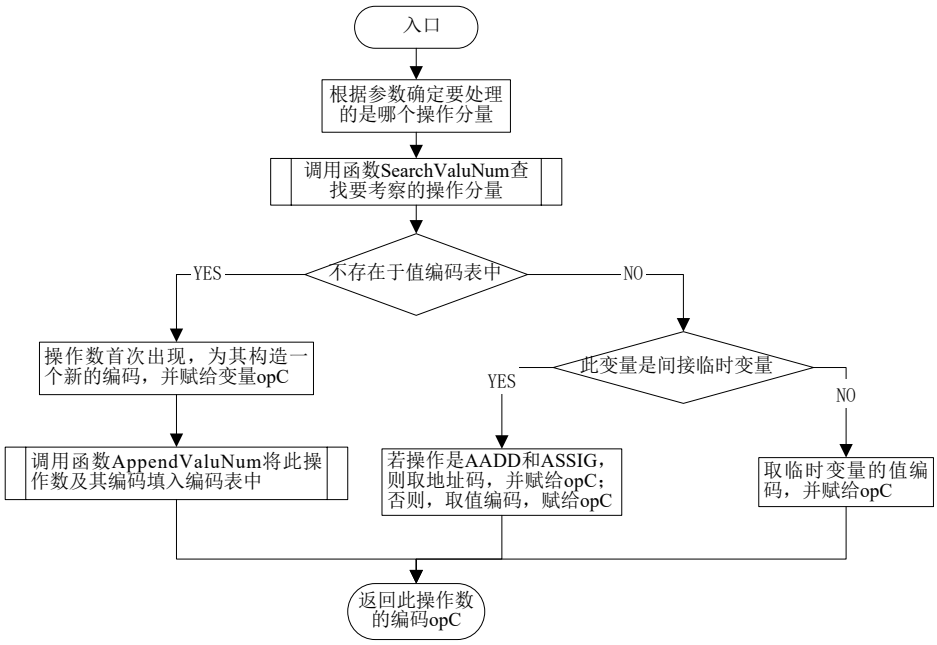


图8.11 运算操作分量的处理函数Process的算法框图

(7) 判断两个 ARG 结构是否等价的函数

函数声明：bool IsEqual(ArgRecord *arg1,ArgRecord *arg2)

算法说明：这里运算分量没有标号，故只考虑了常量类和地址类 ARG 结构。

算法框图：见图 8.12。

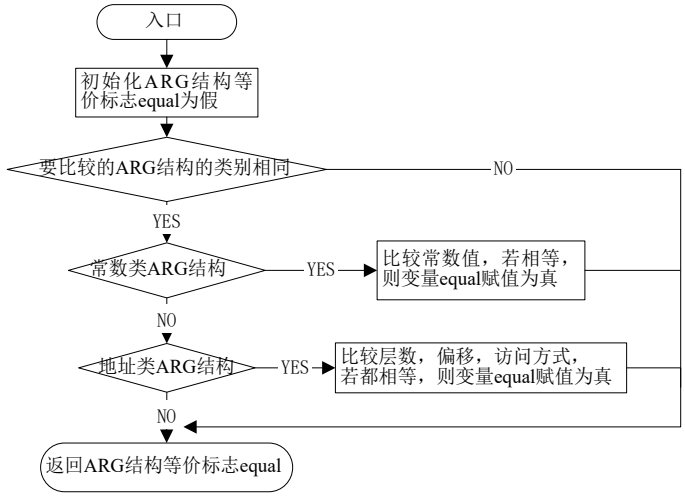


图8.12 判断两个ARG结构是否等价的函数IsEqual的算法框图

(8) 加入值编码表的函数

函数声明：void AppendValuNum(ArgRecord *arg , int Vcode)

算法说明：创建值编码表的一个新的节点。根据参数判断：若变量是间接临时

变量，则在节点中填入其值编码和地址码；否则只填入值编码。将此节点连入值编码表的表尾。

算法框图：略。

(9) 查找可用的表达式函数

函数声明：CodeFile* FindECC(CodeKind codekind,int op1Code, int op2Code)

算法说明：若有可用的表达式，返回用于替换的中间代码指针；否则返回为空。

算法框图：见图 8.13。

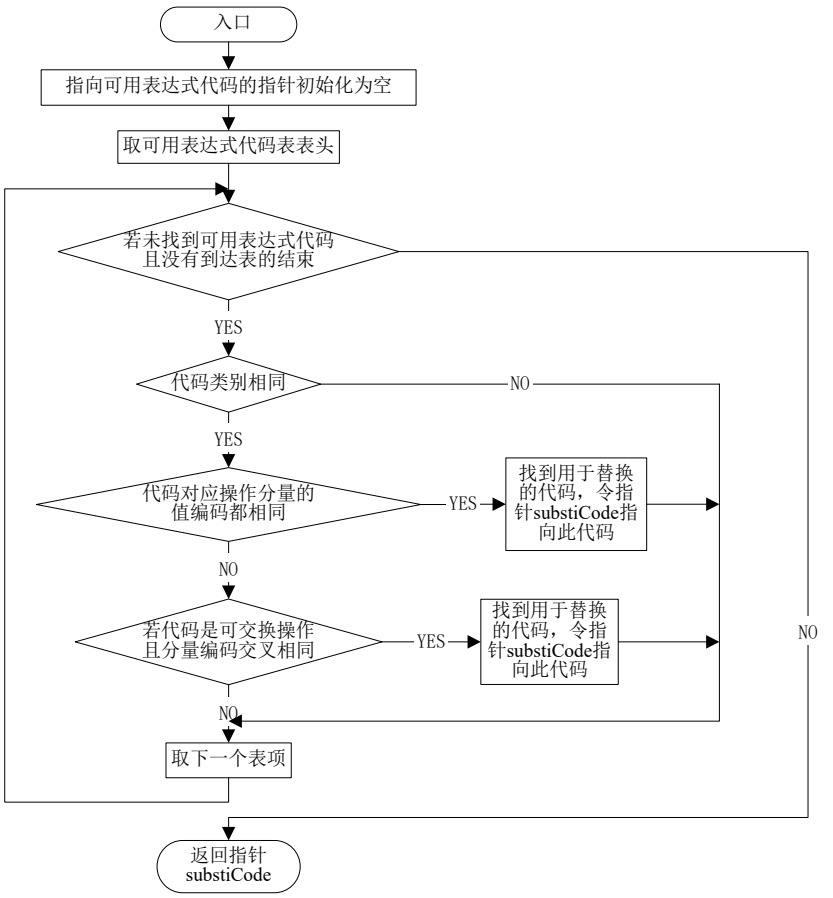


图8.13 查找可用的表达式函数FindECC的算法框图

(10) 加入临时变量等价表函数

函数声明：void AppendTempEqua(ArgRecord *arg1,ArgRecord *arg2)

算法说明：创建一个新的临时变量等价表的节点。根据参数，在节点中填入被替换变量的 ARG 结构指针 arg1 和用于替换的变量的 ARG 结构指针 arg2 ；将节点连入临时变量等价表的表尾。

算法框图：略。

(11) 构造映像码函数

函数声明：MirrorCode * GenMirror(int op1,int op2,int result)

算法说明：申请一个新的映像码结构空间，将分量 1，分量 2，和操作结果的编码 op1，op2，result 分别填入节点对应位置，返回此映像码结构的指针。

算法框图：略。

(12) 加入可用表达式代码表

函数声明：void AppendUsExpr(CodeFile *code,MirrorCode *mirror)

算法说明：申请一个新的可用表达式代码表节点，填入指向中间代码的指针 code，和此代码的映像码指针 mirror；将节点连入可用表达式代码表的表尾。

算法框图：略。

(13) 替换编码表中变量编码的函数

函数声明：void SubstiVcode(ArgRecord *arg,int Vcode)

算法说明：若变量首次出现，则添加一项；否则，将表中此变量的值编码替换为新的值编码。

算法框图：见图 8.14。

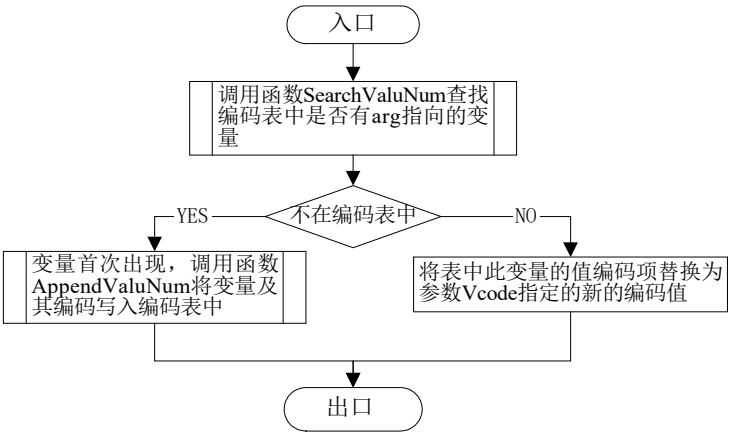


图8.14 替换编码表中变量编码的函数SubstiVcode的算法框图

(14) 删除可用代码表项函数

函数声明：void DelUsExpr(ArgRecord *arg)

算法说明：从头开始查找可用表达式代码表，对于每一项，若用到了 arg 指针指向的变量的值编码，则删除此可用表达式项。

算法框图：略。

8．4 循环不变式外提

循环不变式外提是循环优化的一种，所谓循环优化是指对循环中的代码进行优化。要进行循环优化，首先要通过对程序的控制流程的分析来确定程序中的循环，其次要通过对程序的数据流程分析来确定构成循环的语句序列中变量的定值和引用

关系。对循环中的代码，可以实行强度削减、不变式外提、变换循环控制条件等优化。我们仅就其中最常用的循环不变式外提优化加以介绍。

8．4．1 循环不变式外提的原理

所谓循环不变式是指与循环执行次数无关的运算或不受循环控制变量影响的运算。一个循环不变式的值只需要计算一次，但它出现在循环体中，就要被重复计算很多次，这样会降低程序的执行效率。例如：

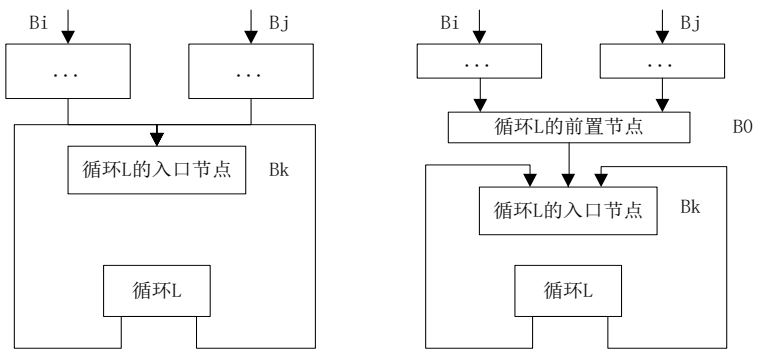
```
while i <= 10000 do
begin
a[i] := x * y;
i := i + 1;
endwh
```

其中 x*y 就是循环不变式，因为这个表达式出现在循环体中，则这个表达式就被多计算了 9999 次。如果把该表达式提到循环体的外面，即：

```
T := x * y
while i <= 10000 do
begin
a[i] := T;
i := i + 1;
endwh
```

则表达式 x*y 只被计算一次，程序效率明显提高。

因为循环体中可能包含很多基本块，所以循环不变式外提优化不能以基本块为单位，是在整个循环范围内进行的优化。我们规定要将循环不变式外提到循环的前置节点中。循环的前置节点是在循环的入口节点前建立的一个新的节点(基本块)，循环的入口节点是它唯一的后继，并且原中间代码中从循环外转向循环入口节点的代码修改为转向循环的前置节点。因为循环的入口节点是唯一的，所以前置节点也是唯一的。循环前置节点的建立如图 8.15 所示。



（a）建立前置节点B0前的循环L　　（b）建立前置节点B0前的循环L

图8.15 循环前置节点的建立

因此，要实现循环不变式外提，首先要识别出循环的入口部分、循环体部分和

出口部分。SNL 只有一种循环，就是 while 循环，故只要考虑 while 循环的不变式外提就可以了。while 的循环体部分紧接着 while 的入口部分，故只需识别出循环入口和循环出口部分。四元式中间代码中，入口标号和出口标号，分别用 WHILESTART 和 ENDWHILE 标志。

1．用到的表的说明

- 变量定值表：要实现循环不变式的外提，首先要判断出哪些表达式是循环不变式，判断不变式的标准就是表达式的变量是否在循环体中被改变，因此要记录变量在循环体中的定值情况；
- 循环信息表：用来记录一层循环的信息，其中包括循环的入口，出口，特征标志以及本层的变量地址表的位置；
- 循环信息栈：因为循环可能是嵌套的，所以要用到用来记录嵌套的各层循环的信息表指针。

2．要注意的几个问题

在循环不变式的外提中，下面几种情形要特别注意：

(1)外提是对循环不变表达式而言，不能对赋值语句进行外提，因为赋值语句外提可能导致程序执行的错误。例如：

```
while e1 do
begin
        if e2 then x:=1 else x:=2;
        y:=x+2;
endwh
```

x:=1 和 x:=2 是无法外提的。

如果要进行外提的话，可能要进行大量的分析工作，开销过大，一般都不做。

(2)可外提运算只包括加、减、乘和地址加，不包含除法运算，因为如果对出现在条件语句中的除法外提后，可能会有除零溢出的错误。例如：

```
while e1 do
if y=0 then x:=1
else x:=1/y;
endwh
```

1/y 是一个不变表达式，如果把它提到循环之外，当 y:=0 时则会产生错误,如果不进行外提则不会有这种错误。

(3)循环体中有过程调用和对变量参数赋值情形，则该循环视为不可外提循环。因为过程调用可能有副作用，变量参数赋值，我们也无法判断它们所对应的实在参数，如果不做大量分析工作的话，就无法判定哪些表达式是不变表达式，这时就不能做外提优化。

3．算法说明

下面给出循环不变式外提的算法说明：

循环不变式优化算法：

(1) 初始化变量定值表 VarDefSet 为空，取第一条中间代码；
(2) 若代码为空，则结束。否则，判断当前代码的种类：
- 若代码为算术运算(ADD,ARG1,ARG2,ARG3)或(SUB,ARG1,ARG2,ARG3)或(MULT,ARG1,ARG2,ARG3)或(DIV,ARG1,ARG2,ARG3)或地址运算(AADD, ARG1, ARG2, ARG3), 则将结果变量 ARG3 加入变量定值表 VarDefSet 中，转(3)；
- 若代码为赋值 (ASSIGN,ARG1,ARG2,_)，则将被赋值的变量 ARG2 加入变量定值表 VarDefSet 中；转(3)；
- 若代码为循环入口(WHILESTART, ARG1, _ , _)，将外提标志，此循环在变量定值表的入口，循环入口指针等信息压入循环信息栈；转(3)；
- 若代码为循环出口(ENDWHILE, ARG1 , _ , _)，且外提标志为真，则调用函数 LoopOutside 进行代码外提，弹循环信息栈，转(3)；
- 若代码为过程调用(CALL,ARG1, _ , ARG3), 则将所有循环信息栈中的所有外提标志均设置为不可外提状态 0；转(3)；

(3) 取下一条中间代码，转(2)。

代码外提算法：

(1) 取得循环入口位置和出口位置，设置层数 level ＝ 0 ；
(2) 取循环的第一条代码
(3) 若已是到达循环结尾，则结束，否则判断此代码：
- 若代码类别为 WHILESTART, 则层数 level 加 1；转(4)
- 若代码类别为 ENDWHILE，则层数 level 减 1；转(4)
- 若代码类别为 ADD,SUB, MULT, AADD,且 level＝0，则判断：
  若 ARG1 和 ARG2 都不在变量定值表中，则可以外提，将代码提到循环入口的前一条语句，并从变量定值表中删去结果变量 ARG3；
  否则，将 ARG3 加入变量定值表中；

(4) 取下一条代码，转(3)

这里需要注意一点：在进行外提时，应该跳过内层循环，因为它们已经被外提过。我们的实现办法如下：设置一个循环层数计数器 level，在外提开始时，令 level=0。以后每当遇到标志循环开始的代码 WHILESTART 时，计数器 Level 加 1，而每当遇到循环结束代码 ENDWHILE 时，将计数器 level 减 1。这样当扫描一个中间代码时，只需看 Level 的当前值，如果 Level>0，则表示当前代码是内层循环中的代码，已经被处理过了，可以跳过；否则再检查是否可以外提，如果可以外提，则进行外提处理。

8．4．2 循环外提的实现

1．输入输出

循环不变式优化作为独立的一遍，其输入可以是经过或没经过常量表达式优化和公共表达式优化的中间代码。其输出是经过循环不变式优化后的中间代码，而且为了清晰地体现代码对变量定值表的改变情况，我们还输出了经过每条代码改变后的变量定值表。

循环不变式优化前和优化后的代码对比见表 8.3：

源程序	循环不变式优化前	循环表达式优化后

program pp
 	
var
integer i,n,m,j,k;
array [1..10]
of integer a;
 	
begin
i:=0;
j:=0;
k:=0;
while k<3 do
while i<5 do
m:=10*(5+j);
{运算提两层}
m:=k+5;
{运算提一层}
n:=i;
write(n);
i:=i+1
endwh;
i:=0;
k:=k+1
endwh;
 	
write(m)
end.
	0: MENTRY 30 22	0: MENTRY 30 22

	1: ASSIG 0 i	1: ASSIG 0 i
	2: ASSIG 0 j	2: ASSIG 0 j
	3: ASSIG 0 k	3: ASSIG 0 k
	4: WHILESTART 1	(外提代码) **4: ADD 5 j temp25**
	5: LT k 3 temp23	(外提代码) **5: MULT 10 temp25 temp26**
	6: JUMP0 temp23 2	6: WHILESTART 1
	7: WHILESTART 3	7: LT k 3 temp23
	8: LT i 5 temp24	8: JUMP0 temp23 2
	9: JUMP0 temp24 4	(外提代码) **9: ADD k 5 temp27**
	(可以外提两层) **10: ADD 5 j temp25**	10: WHILESTART 3
	(可以外提两层) **11: MULT 10 temp25 temp26**	11: LT i 5 temp24
	12: ASSIG temp26 m	12: JUMP0 temp24 4
	(可以外提一层) **13: ADD k 5 temp27**	13: ASSIG temp26 m
	14: ASSIG temp27 m	14: ASSIG temp27 m
	15: ASSIG I n	15: ASSIG i n

	16: WRITE n	16: WRITE n
	17: ADD I 1 temp28	17: ADD i 1 temp28
	18: ASSIG temp28 i	18: ASSIG temp28 i
	19: JUMP 3	19: JUMP 3
	20: ENDWHILE 4	20: ENDWHILE 4
	21: ASSIG 0 i	21: ASSIG 0 i
	22: ADD k 1 temp29	22: ADD k 1 temp29
	23: ASSIG temp29 k	23: ASSIG temp29 k
	24: JUMP 1	24: JUMP 1
	25: ENDWHILE 2	25: ENDWHILE 2
	26: WRITE m	26: WRITE m

表 8.3 循环不变式优化结果示例表

2．用到的数据结构

(1) 变量定值表 VarDefSet：

因为外层循环的变量定义集一定包含内层循环的变量定义集，因此多层循环可以只用一个变量定义表，而每层循环信息中的变量定义集只需记录本层循环中的变量定值在变量定值表中的开始位置。变量定值表用一个指向变量 ARG 结构的指针数组表示，而开始位置即为数组的下标。

(2) 循环信息表 LoopInfo：每层循环一个。

state	whileEntry	varDef	whileEnd

成员 state　　整型变量，表示循环状态，取值 0 时为不可外提状态，取值 1 时为可外提状态；

成员 whileEntry　指针变量，指向循环入口标志中间代码；

成员 varDef　　整型变量，取值变量定值表的下标值，表示本层的变量定值部分在整个变量定值表中的开始位置；

成员 whileEnd　　指针变量，指向循环出口标志中间代码。

(3) 循环信息栈：

因为循环可以嵌套，需要用到一个栈，以保存未结束循环的信息。栈的元素就是循环的信息表的内容。每当进入新一层循环时，将新循环的信息压入栈；而每当退出循环时，将删除栈顶元素，以使外层循环成为当前循环。

3. 程序说明

(1) 循环不变式优化主函数

函数声明：CodeFile *LoopOpti( )

算法说明：从第一条代码开始，循环处理每条代码，直到中间代码结束。

算法框图：见图 8.16。

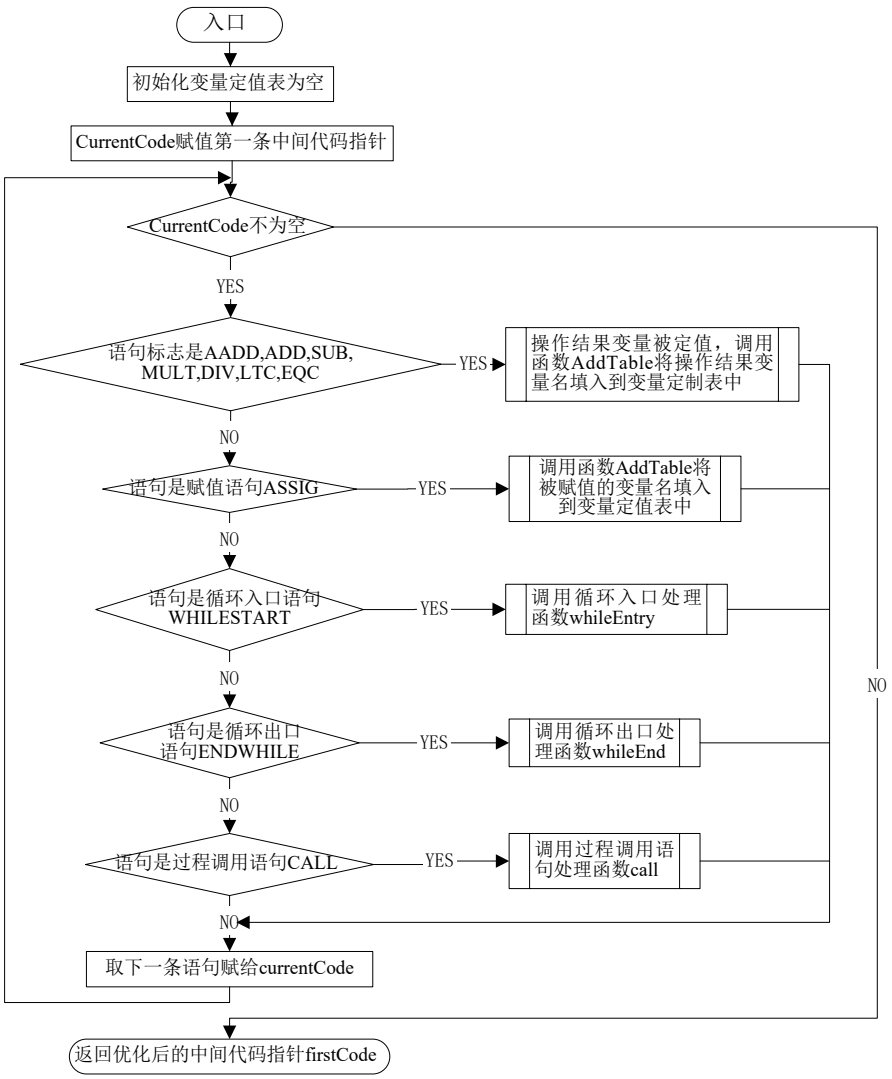


图8.16 循环不变式优化主函数Loop0pti的算法框图

(2) 循环入口的处理函数

函数声明：void whileEntry(CodeFile *code)

算法说明：记录当前循环的各种信息，并压入循环信息栈中。
算法框图：见图 8.17。

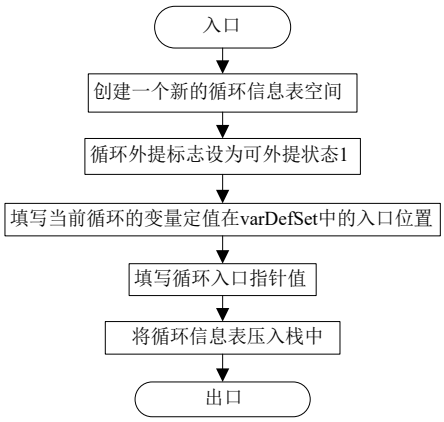


图8.17 循环入口处理函数whileEntry的算法框图

(3) 过程调用语句的处理函数
函数声明：void call(CodeFile *code)
算法说明：所有包含此调用语句的循环都不能做不变式外提。
算法框图：见图 8.18。

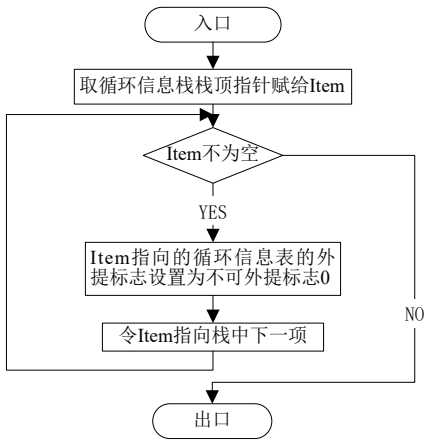


图8.18 过程调用语句的处理函数call的算法框图

(4) 循环出口处理函数
函数声明：void whileEnd(CodeFile *code)
算法说明：若可以外提，则执行代码外提，结束本层循环处理。
算法框图：见图 8.19。

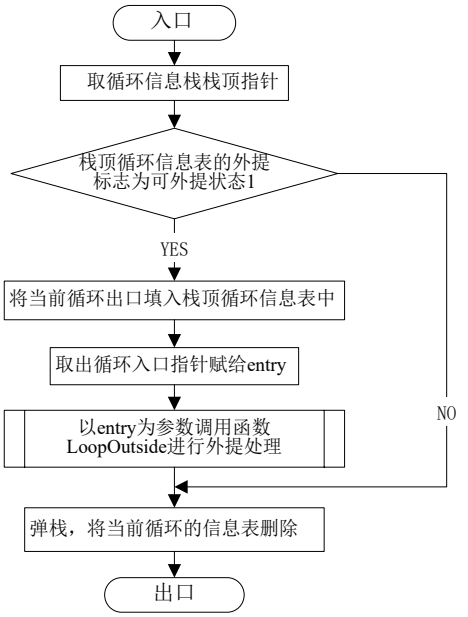


图8.19 循环出口处理函数whileEnd的算法框图

(5) 其它应用函数

a．变量定值表查找函数

函数声明：int SearchTable(ArgRecord *arg , int head)

算法说明：根据参数 head 确定表的开始位置，由变量的 ARG 结构，可以得到变量的层数和偏移，其中，若变量为临时变量，层数为-1，偏移表示编号。从表头开始查找，若层数和偏移都相等，则找到，返回变量所在的位置；否则，没找到，返回值为-1。

算法框图：略。

b．删除变量定值表中的一项

函数声明：void DelItem(ArgRecord *arg, int head)

算法说明：调用查找函数 SearchTable 查找要删除的标志符，若在表中，则删除此变量，变量定值总数减 1；否则直接结束。

算法框图：略。

c．添加变量定值表中的一项

函数声明：void AddTable(ArgRecord *arg)

算法说明：若要添加的变量在某层循环中，则取循环的变量定值表的开始位置赋给 head，否则，head 值赋为 varDefSet 的开始位置 0；以 head 为参数调用查找函数 SearchTable 查找要添加的标志符，若不在表中，则填加到表尾，变量定值总数加 1；否则，直接结束。

算法框图：略。

d．循环外提处理函数

函数声明：void LoopOutside(CodeFile *entry)

算法说明：从循环信息栈栈顶取得当前循环信息，循环检查每条代码是否可以

外提，直到本层循环结束。

算法框图：见图 8.20。

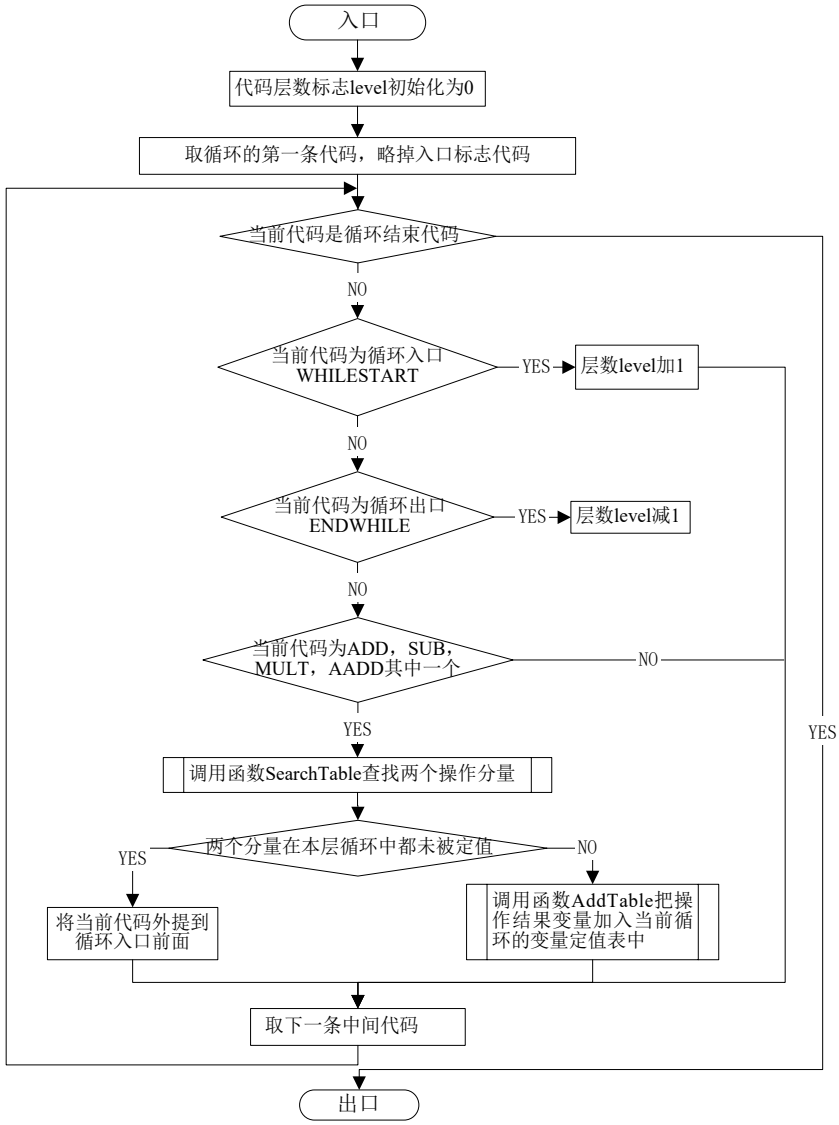


图8.20　循环外提处理函数Loop0utside的算法框图

第九章 SNL 语言的目标代码生成

9．1 虚拟目标机 TM

一个好的编译器的设计，应考虑到便于移植到新的机器上，考虑到在现存机器上生成新的版本。对于先生成中间代码后生成目标代码的编译器，如果想改变目标机，只需重新设计或修改目标代码的生成器部分，因此便于编译器的移植。

虚拟目标机 TM(Tiny Machine)，是 Kenneth C. Louden 在“Compiler Construction Priciples and Practice”一书中为 Tiny 语言设计的虚拟目标机。由于 TM 上的指令简洁易懂，且对于 SNL 语言来说功能上够用，因此我们采用虚拟目标机 TM 上的指令集作为 SNL 编译器的目标代码。这也使得 SNL 编译器具有很高的可移植性。

9.1.1 节和 9.1.2 节将对 TM 的结构和指令集给出完整详细的描述。虚拟目标机 TM 的解释执行程序及讲解见本书第十章。

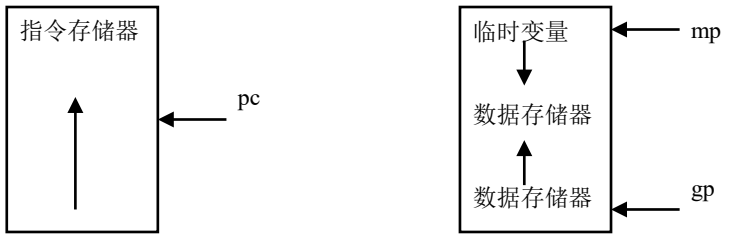
9．1．1 TM 的寄存器和存储器

TM 虚拟机有 8 个寄存器,两个存储器。

- 寄存器由整数类型数组 reg[8]构成,各寄存器功能如下:
  - reg[0] 第一累加器 ac,用于执行计算。
  - reg[1] 第二累加器 ac1,用于执行计算。
  - reg[2] 第三累加器 ac2，用于计算。
  - reg[3] 未用
  - reg[4] sp 到 display 表的偏移 Noff。
  - reg[5] 下一个 AR 指示器 top，记录下一个可用的地址。
  - reg[6] 当前 AR 指示器 sp，记录当前 AR 的始地址。
  - reg[7] 程序地址指示器 pc,记录下一条执行指令地址。
- 存储器有两个,指令存储器和数据存储器,可看作线性数组。
  - 指令存储器由指令结构数组 iMem[1024]构成,用于存放机器指令(只读)。
    数组单元指令结构如下:
    INSTRUCTION 指令结构类型:操作码,参数 1,参数 2,参数 3

Iop 操作码	iarg1 参数 1	iarg2 参数 2	iarg3 参数 3

  - 数据存储器由整数类型数组 dMem[1024]构成,用于存放程序数据。



（a）指令存储器　　（b）数据存储器

图 9.1 SNL 的存储器

9．1．2　TM 的地址模式和指令集

虚拟目标机 TM 有三种地址模式:立即模式,变址模式,寄存器模式。三种地址模式可以组合成的指令模式:

```
Op      r,  s,  t    (寄存器-寄存器)
Op      r,  d,  s    (存储器-寄存器)
Op      r,  d,  s    (寄存器-立即数)
```

TM 虚拟机提供了丰富的指令：

- 寄存器-寄存器指令,格式:op r,s,t;其中 r,s,t 都是寄存器。
  HALT 停止指令,忽略操作数
  IN　读指令,将外部值读入到寄存器 r,忽略 s,t 参数
  OUT　写指令,将寄存器 r 的值输出,忽略 s,t 参数
  ADD　加法指令,寄存器 r 中值赋为寄存器 s 值加上寄存器 t 中值
  SUB　减法指令,寄存器 r 中值赋为寄存器 s 值减去寄存器 t 中值
  MUL　乘法指令,寄存器 r 中值赋为寄存器 s 值乘上寄存器 t 中值
  DIV　除法指令,寄存器 r 中值赋为寄存器 s 值除以寄存器 t 中值

- 寄存器-内存指令,格式:op r,d,s;其中 r,s 为寄存器,d 为地址偏移。
  LD　载入指令,寄存器 r 的值赋为地址为 d+reg(s)的内存单元值
  ST　设置指令,将地址 d+reg(s)指定内存单元写成寄存器 r 的值

- 寄存器-立即数指令,格式:op r,d,s;其中 r,s 为寄存器,d 为立即数。
  LDA　地址载入指令,将寄存器 r 中值赋为立即数 d 与寄存器 s 的值的和
  LDC　数值载入指令,将寄存器 r 中值赋为立即数 d,参数 s 被忽略
  JLT　条件跳转指令,如果寄存器 r 的值小于 0,将指令指示寄存器 pc 的值赋为立即数 d 与寄存器 s 的值的和
  JLE　条件跳转指令,如果寄存器 r 的值小于等于 0,将指令指示寄存器 pc 的值赋为立即数 d 与寄存器 s 的值的和
  JGT　条件跳转指令,如果寄存器 r 的值大于 0,将指令指示寄存器 pc 的值赋为立

即数 d 与寄存器 s 的值的和

JGE 条件跳转指令,如果寄存器 r 的值大于等于 0,将指令指示寄存器 pc 的值赋为立即数 d 与寄存器 s 的值的和

JEQ 条件跳转指令,如果寄存器 r 的值等于 0,将指令指示寄存器 pc 的值赋为立即数 d 与寄存器 s 的值的和

JNE 条件跳转指令,如果寄存器 r 的值不等于 0,将指令指示寄存器 pc 的值赋为立即数 d 与寄存器 s 的值的和

9．2 编译程序中运行时存储空间管理

目标代码生成是编译程序的最后一个阶段，它的输入主要是源程序的某种中间表示，还有符号表的信息，其中符号表的信息主要用于决定程序中名字所代表的数据对象的运行地址，它的输出是等价的目标代码，将完成这种功能的程序称为目标代码生成器。如图 9.2 所示。

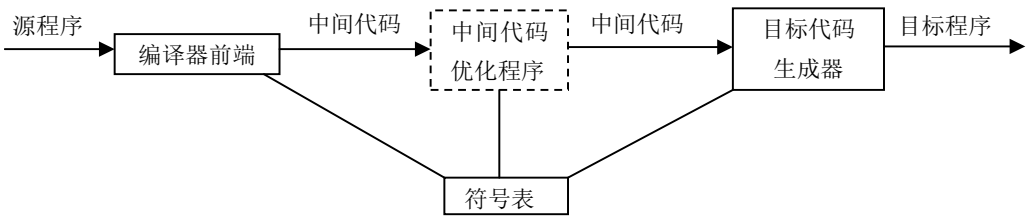


图 9.2 目标代码生成器的功能

如第七章所述，中间代码有多种表示形式，如逆波兰式、三元式、四元式以及语法树表示等。为了让读者更好地了解目标代码生成地过程，我们将介绍两种目标代码生成方法：由语法树生成目标代码和由四元式生成目标代码。

9．2．1 存储空间结构

存储管理实际是一个地址映射问题，即把源程序中用标识符表示的名字映射到运行时数据对象的地址。在语义分析和中间代码生成阶段已经为此做好了准备工作，如为名字建立符号表，根据声明语句中的类型确定名字所对应的存储空间大小，因此根据符号表中登记的相应名字信息可直接确定该名字在数据区的相对地址。目标代码生成阶段则要把这些相对地址转换成目标机的绝对地址。

编译程序必须分配目标程序运行的数据空间。程序语言关于名字的作用域和生存期的定义规则决定了分配目标程序数据空间的基本策略。

- 静态分配策略：如果一个程序语言不允许递归过程，不允许含有可变体积的数据项目或待定性质的名字，则可以在编译时完全确定其程序的每个数据项目存储空间的位置，这种策略叫做静态分配策略。

- 栈式动态分配策略：如果程序语言允许有递归过程或可变(体积的)数组，则其程序数据空间的分配需采用某种动态策略(在程序运行时动态地进行分配)。此时，目标程序可以用一个栈作为动态的数据空间。运行时，每当进入一个过程或分程序，它所需的数据空间就动态地分配于栈顶，一旦退出，它所占用的空间就予以释放，这种办法叫做栈式动态分配策略。
- 堆式动态分配策略：如果程序语言允许用户动态地申请和释放存储空间，而且申请和释放之间不一定遵守“先申请后释放”和“后申请先释放”的原则，此时就必须让运行程序拥有一个大的存储区(称为堆)，遇到申请指令，从堆中分配一块合适的空间，遇到释放指令，将分配的空间收回，这种办法叫做堆式动态分配策略。

不同的分配策略对应的存储空间结构也不同。在 SNL 语言中，因为允许过程递归，并且不存在动态申请和释放空间的指令，所以 SNL 的编译程序采用栈式的动态分配策略。

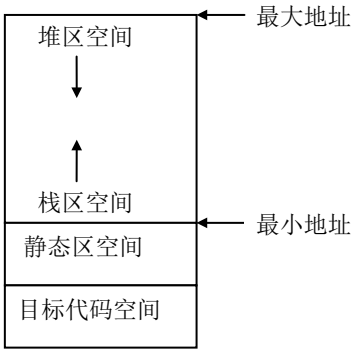


图 9.3 (a)运行时的一种存储结构

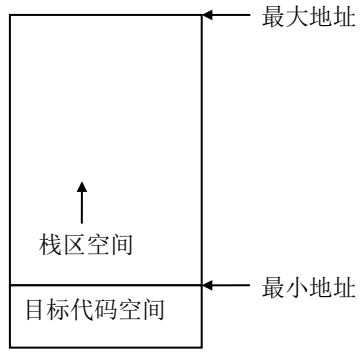


图 9.3(b) SNL 编译器运行时的存储结构

常用的一种运行时的存储结构如图 9.3(a)所示，SNL 目标程序的运行空间的结构如图 9.3(b)所示。首先是由编译器产生的目标代码空间，在 SNL 编译系统中，我们将采用指令存储器(由指令结构数组 iMem[1024]构成)来表示；然后是静态区空间，静态区用于分配那些可用绝对地址的数据和变量(如全程变量)，由于在 SNL 编译系统中没有涉及到静态变量和全局变量，所以这部分不予考虑；最后剩下的部分是栈和堆区，它们之间没有事先划好的界限。在目标代码运行时，栈区指针和堆区指针不断的变化，而且朝相对方向增长。当两个指针相互越界时，表示存储空间要溢出。由于在 SNL 编译系统中不存在动态申请空间分配的情况，故在此我们可以不考虑堆区，只考虑栈区——过程活动记录的管理情况。

9．2．2　过程活动记录

过程的活动是指过程的一次执行，即每次执行一个过程体，就产生该过程的一个活动。为了管理过程在一次执行中所需要的信息，需要使用一个连续的存储块，称其为过程活动记录(Action Record ,简记为 AR)。当调用一个过程时，产生该过

程的一个新的活动记录，将其压入运行栈的栈顶，当过程执行完最后一条语句时，又会从栈里将该过程的活动记录弹出。

一个过程活动记录 AR 中存放的信息大体有以下几个部分：

- 过程控制信息：包括返回地址，先行活动记录的动态链指针，层数等。
- 机器状态信息：包括寄存器状态等过程中断时的机器状态。
- 局部变量值：包括形参变量，局部变量和临时变量的值。

图 9.4(a)和图 9.4(b)分别给出了从语法树直接生成目标代码和从四元式中间代码生成目标代码所用到的过程活动记录。

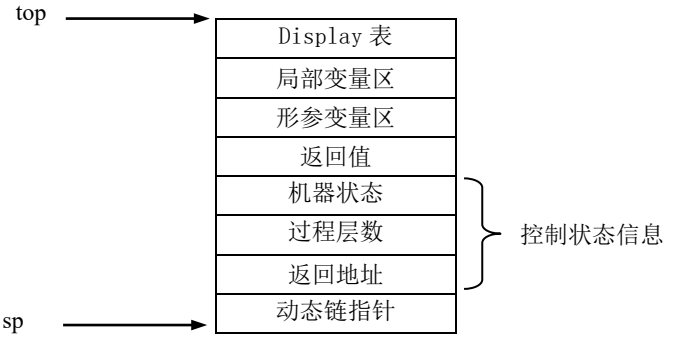


图 9.4(a) 由语法树直接生成目标代码时采用的 AR 结构

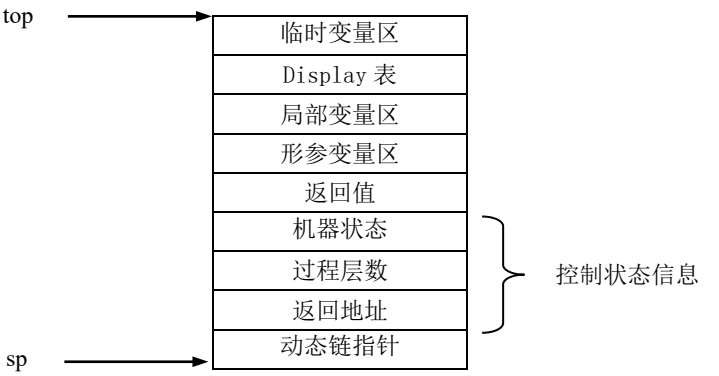


图 9.4(b) 由四元式生成目标代码时采用的 AR 结构

在图 9.4(a)和图 9.4(b)中，sp 表示栈指针，它始终指向当前 AR 的始地址；top 始终指向当前 AR 的末地址。它们的区别在于对临时变量的处理，由于语法树结构中不包含临时变量，只是在生成目标代码时需要用临时变量来暂存中间结果，因此临时变量并不存储在 AR 中，而是从整个存储区的最大地址开始向下分配。对于四元式到目标代码的生成，由于四元式结构中包含临时变量，故将临时变量存储在过程活动记录中。这里我们采用静态分配方法处理临时变量的存储。静态分配方法是：编译器事先计算出临时变量的空间大小，并在过程调用时和源变量一样申请到空间，即在调用时就将临时变量的空间安排在 AR 内。

此处应该设置形参开始偏移量 offset 值为 4 + 机器状态中寄存器的个数= 8。因为 sp、top、指令地址寄存器 reg[7]和临时变量 mp 不用保留，其余 4 个寄存器 Noff(用 display 表示)和三个累加器 R0、R1、R2 均须保留，故机器状态中寄存器的个数为 4。这 8 项从底向上依次为：old sp，return address，level，ac，ac1，ac2，displayOff，return value。

当过程调用被执行时，其目标代码将负责完成以下工作：

(1) 实参值送入 NewAR 的形参单元中(此处应该特殊处理，因为有多种情况)；

(2) 将控制信息填入 NewAR 的相应单元中；

(3) 将机器状态信息填入 NewAR 的相应单元中。

其中 NewAR 表示新 AR。调用过程意味着要开辟一个新的 AR，而且新 AR 将成为当前 AR，sp = top ,top = top + 当前 AR 的大小。当过程调用结束时，将释放当前活动记录，即 top = sp；把保存在 currentAR 中的 sp 值取出来赋给寄存器 sp。

9．2．3 动态链

当一个过程结束时需要释放当前 AR，即把寄存器 sp 的内容恢复成指向当前 AR 下面的 AR(当前过程是主程序时除外)，如果每个 AR 的长度是固定的，那么 sp 减去一个固定值(AR 的长度)就是新的 sp，但实际上 AR 的长度并不相同，因此在每个活动记录中要保存前一 AR 的始地址，于是栈上的 AR 就被指针连了起来，这种连起来的 AR 结构，称之为动态链。其逻辑示意图如图 9.5 所示：

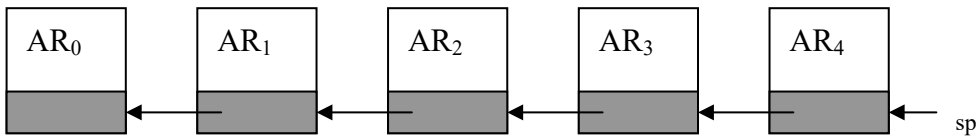


图 9.5 动态链的逻辑示意图

调用过程 P 时，产生新的 AR 为 NewAR，并将当前 sp 的内容填入到 NewAR 中，再修改 sp 的内容，使得 sp 指向 NewAR。操作过程如下：

- 参数传递；
- NewAR.DynaChainPointer = sp;
- NewAR.Return = ReturnAddr;
- NewAR.Level = Level(P);
- NewAR.Machine = (R0,R1,R2,R4);
- 填 display 表;
- sp = top ;
- top = sp + Moff(P);

其中过程的返回地址需要在处理完此过程入口处代码后回填；保存一些指定寄存器的内容：累加器 ac、ac1、ac2 和 display 表到该过程活动记录的距离 displayOff(Noff)。

退出过程 P 时，操作过程为：

- (R0,R1,R2,R4) = CurrentAR.Machine;
- top = sp;
- sp = CurrentAR.DynaChainPointer;
- 按照 CurrentAR.return 返回(将返回地址送入 reg[7]);

由于在 SNL 语言中不存在非正常出口情况，所以在此不考虑非正常出口情况。

9．3　语法树到目标代码的生成

9．3．1　原理

由语法树到目标代码的生成过程，是通过对语法树自顶向下深度优先遍历，根据遍历经过的语法树节点的类型分类处理，生成最终的目标代码。因为只有过程声明需要生成代码，所以对于声明类节点，给出过程声明类型语法树节点生成目标代码的过程。对于语句类节点，给出赋值语句类型节点，条件语句类型节点，循环语句类型节点，读语句类型节点，写语句类型节点，过程调用和返回语句类型节点，表达式节点的目标代码生成过程。

下面给出各类语法树节点生成目标代码的处理思想。

过程声明语法树节点的处理方法如表 9.1 所示：

对应节点处理	生成的目标代码
p1 是该过程的声明部分： p1 = t->child[1] p2 是该过程的语句部分： p2 = t->child[2]	**LDC　displayOff , 该过程活动记录的 displayOff , 0**
	循环处理过程中的声明部分： 调用 genProc(p1)处理过程； **LDC　pc , currentLoc , 0** 注：currentLoc 为回填指令地址
	调用 genStmt(p2)处理语句部分； **LD　ac2 , 1 , sp** **LDA　pc , 0 , ac2**

表 9.1 过程声明语法树节点的处理方法

表达式节点的处理方法如表 9.2 所示：

<table>
<tr><td>节点类型</td><td>对应节点具体分类</td><td colspan="3">生成的目标代码</td></tr>
<tr><td rowspan="6">表达式类型节点</td><td>ConstK 表达式类型</td><td colspan="3">LDC　ac ,　, 0</td></tr>
<tr><td rowspan="3">IdK 表达式类型</td><td colspan="3">FindAdd(t)，将变量的绝对偏移送入 ac，具体处理见表一，下同</td></tr>
<tr><td colspan="3">该变量是否是间接变量？</td></tr>
<tr><td colspan="2">是，则：
LD　ac1 , 0 , ac
LD　ac , 0 , ac1</td><td>不是：
LD　ac , 0 , ac</td></tr>
<tr><td rowspan="5">OpK 表达式类型</td><td rowspan="5">分别将 t 的左、右儿子节点赋值 p1，p2。
cGen(p1)生成左操作数目标代码，并在 ac 中压入左操作数：
ST ac,tmpOffset - -,mp
cGen(p2)为右操作数生成目标代码，并将表达式的值送入 ac1 中：
LD ac1,++tmpOffset,mp
再对 t 的成员运算符 attr.op 分类处理：</td><td>PLUS（加）</td><td>ADD　ac , ac1 , ac</td></tr>
<tr><td>MINUS（减）</td><td>SUB　ac , ac1 , ac</td></tr>
<tr><td>TIMES（乘）</td><td>MUL　ac , ac1 , ac</td></tr>
<tr><td>OVER（除）</td><td>DIV　ac , ac1 , ac</td></tr>
<tr><td>LT（小于）</td><td>SUB　ac , ac1 , ac
JLT　ac , 2 , pc
LDC　ac , 0 , ac
LDA　pc , 1 , pc
LDC　ac , 1 , ac</td></tr>
<tr><td>EQ（等于）</td><td>SUB　ac , ac1 , ac
JEQ　ac , 2 , pc
LDC　ac , 0 , ac
LDA　pc , 1 , pc
LDC　ac , 1 , ac</td></tr>
<tr><td colspan="4">注：tmpOffset 为临时变量偏移</td></tr>
</table>

表 9.2　表达式节点的处理方法

语句类型节点的处理如表 9.3 所示：

<table>
<tr><th>节点类型</th><th>具体语句类型</th><th>对应语法树节点的处理</th><th colspan="2">生成的目标代码</th></tr>
<tr><td rowspan="11">语句类型节点</td><td rowspan="2">If 语句</td><td rowspan="2">条件表达式部分：
p0 = t->child[0];
then 语句序列部分：
p1 = t->child[1];
else 语句序列部分：
p2 = t->child[2];</td><td colspan="2">cGen(p0);
写入跳转到 else 的指令：
JEQ ac , currentLoc
cGen(p1);
写入跳转到 end 的指令：
LDA pc , currentLoc
cGen(p2);</td></tr>
<tr><td rowspan="2">While 语句</td><td rowspan="2">条件表达式部分：
p0 = t->child[0];
语句序列部分：
p1 = t->child[1];</td><td colspan="2">cGen(p0);
跳至 while 语句结束：
JEQ ac , currentLoc
cGen(p1);
跳到条件表达式处，进入下一次循环：
LDC pc , currentLoc , 0</td></tr>
<tr><td rowspan="3">Assign 语句</td><td rowspan="3">赋值号左侧的部分：
p0 = t->child[0];
赋值号右侧的部分：
p1 = t->child[1];</td><td colspan="2">FindAdd(p0)
LDA ac2 , 0 , ac
cGen(p1);</td></tr>
<tr><td colspan="2">赋值号左侧变量是否为直接变量？</td></tr>
<tr><td>是：
ST ac , 0 , ac2</td><td>不是：
LD ac2 , 0 , ac2
ST ac , 0 , ac2</td></tr>
<tr><td rowspan="3">Read 语句</td><td colspan="3">IN ac , 0 , 0
LDA ac2 , 0 , ac
FindAdd(t);</td></tr>
<tr><td colspan="3">该变量是否为直接变量？</td></tr>
<tr><td>是：
ST ac2 , 0 , ac</td><td>不是：
LD ac1 , 0 , ac
ST ac2 , 0 , ac1</td></tr>
<tr><td>Write 语句</td><td>p0 = t->child[0]</td><td>cGen(p0);
OUT ac , 0 , 0</td></tr>
<tr><td>Return 语句</td><td colspan="3">空</td></tr>
<tr><td colspan="5">注：currentLoc 记录目标代码标号</td></tr>
</table>

<table>
<tr><td rowspan="11">语句类型结点</td><td rowspan="11">Call 语句</td><td rowspan="11">过程名：
p0 = t->child[0];
过程的实参：
p1 = t->child[1];
遍历实参表，循环执行右侧处理。</td><td rowspan="2">形参是间接变量</td><td colspan="2">实参是值参：FindAdd(p1)
ST ac , FormParam , top</td></tr>
<tr><td colspan="2">实参是变参：FindAdd(p1)
LD ac , 0 , ac
ST ac , FormParam , top</td></tr>
<tr><td rowspan="3">形参是直接变量</td><td colspan="2">实参是数值或表达式：genExp(p1)
ST ac , FormParam , top</td></tr>
<tr><td colspan="2">实参是变量：FindAdd(p1)</td></tr>
<tr><td>是间接变量：
LD ac2,0,ac
LD ac2,0,ac2
ST ac2,FormParam,top</td><td>是直接变量：
LD ac2,0,ac
ST ac2, FormParam,top</td></tr>
<tr><td colspan="3">ST sp , 0 , top
ST ac , 3 , top
ST ac1 , 4 , top
ST ac2 , 5 , top
ST displayOff , 6 , top
LDC displayOff , 新的 displayOff 的值 , 0
LDC ac1 , 该过程的层数 , 0
ST ac1 , 2 , top</td></tr>
<tr><td>I<过程层数,
做 for 循环,
I++</td><td>LD ac2 , 6 , top
LDA ac2 , ss , ac2
ADD ac2 , ac2 , sp
LD ac1 , 0 , ac2
LDA ac2 , ss , displayOff
ADD ac2 , ac2 , top
ST ac1 , 0 , ac2</td></tr>
<tr><td colspan="3">LDA ac2, 当前层数 , displayOff
ADD ac2 , top , ac2
ST top , 0 , ac2
LDA sp , 0 , top
LDA top,当前过程活动记录长度,top
LDC ac1 , currentLoc , 0
ST ac1 , 1 , top
LDC pc , 当前过程活动记录入口地址 , 0
LD ac , 3 , sp
LD ac1 , 4 , sp
LD ac2 , 5 , sp
LD displayOff , 6 , sp
LDA top , 0 , sp
LD sp , 0 , sp</td></tr>
<tr><td colspan="6">注：FormParam 表示该形参的偏移</td></tr>
</table>

表 9.3 语句类型节点的处理

取变量的绝对地址对应的目标代码如表 9.4 所示

变量种类	取绝对地址对应的目标代码
普通变量	**LDC ac , Loc , 0**
数组类型变量	**LDC ac1 , 数组下界 , 0** **SUB ac , ac , ac1** **LDA ac , Loc , ac**
记录类型变量	域变量为基本类型变量： **LDC ac , Loc , 0** **LDA ac , 域成员变量偏移 , ac**
	域变量是数组变量： **LDC ac1 , 数组下界 , 0** **SUB ac , ac , ac1** **LDA ac , 域成员变量偏移 , ac** **LDA ac , Loc , ac**
注：Loc 为变量在符号表中的地址	

表 9.4 取变量的绝对地址处理

说明：

t 是指向当前语法树节点的指针；

ac,ac1,ac2 三个累加寄存器，用于执行计算；

displayoff 表示记录 sp 到 display 表的偏移的寄存器；

top 寄存器是下一个 AR 指示器，记录下一个可用的地址；

sp 是当前 AR 指示器，记录当前 AR 的始地址；

pc 是程序地址指示器,记录下一条执行指令地址。

9．3．2 框图

为了完成代码生成的必要功能,SNL 编译系统在 code.h 和 code.c 文件，cgen.h 和 cgen.c 文件中提供了目标代码生成函数。

函数之间调用关系如图 9.6 所示:

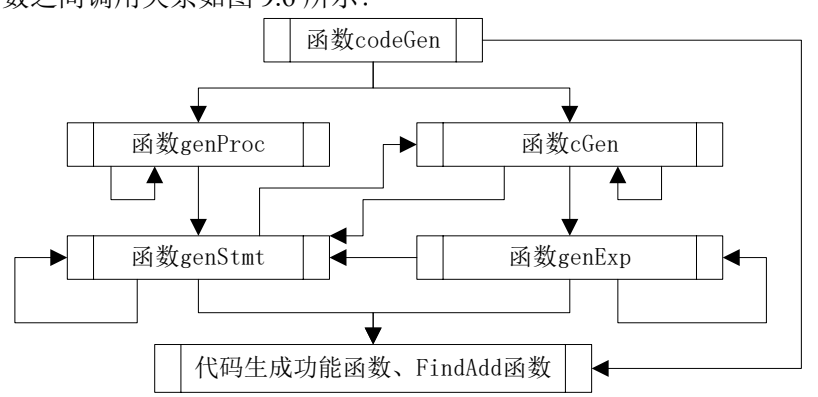


图9.6 目标代码生成函数调用关系图

各函数详细说明和算法框图如下：

1. 过程声明语法树节点代码生成函数 genProc

   函数声明： void genProc(TreeNode * t)

   功　　能： 该函数生成过程声明节点的代码，在过程信息表中填写过程入口地址,遇到过程声明节点再递归调用，最后调用语句类型语法树节点代码生成函数 genStmt，生成语句部分代码。结束时在 pc 中存入返回地址。

   所用变量： p1，p2 为语法树节点类型指针，分别记录当前语法树的二儿子节点和三儿子节点，即声明节点和过程体节点；savedLoc1，currentLoc 为指令回填地址，用作回填指令的跳转地址。

   算法框图： 见图 9.7。

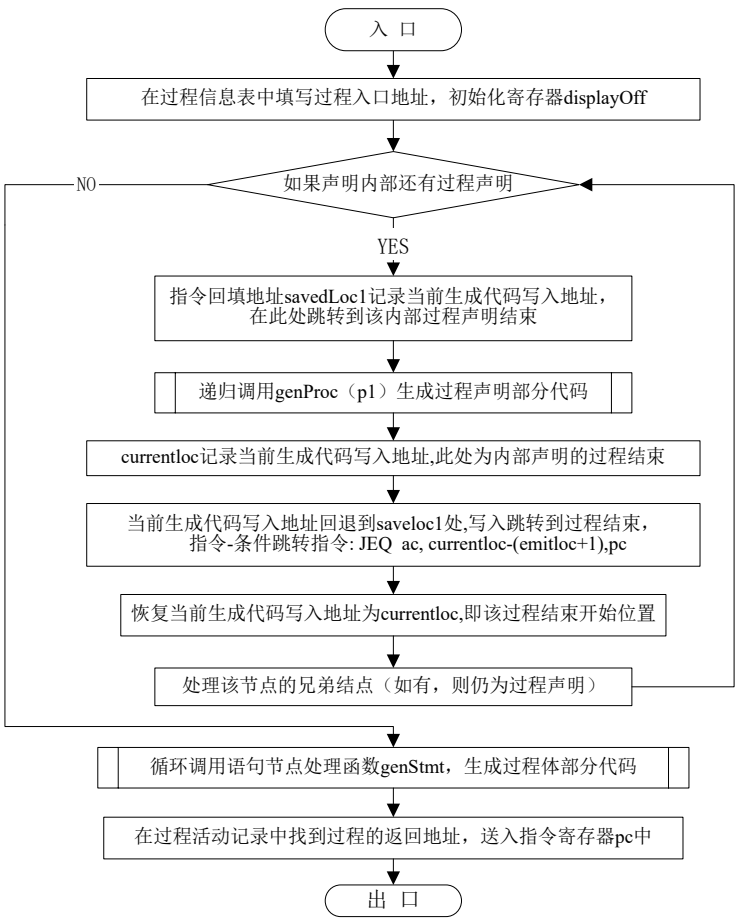


图9.7 过程声明语法树节点代码生成函数genProc的算法框图

2. 语句类型语法树节点代码生成函数 genStmt

函数声明: static void genStmt(TreeNode * tree)
功　　能: 函数通过对语句类型分类处理,调用相关代码生成函数。为语句类型语法树节点生成相应的目标代码。
所用变量: p1,p2,p3 为语法树节点类型指针,记录当前语法树节点的三个子节点。savedLoc1,savedLoc2,currentLoc 为指令回填地址,用作回填指令的跳转地址。
算法框图: 见图 9.8。

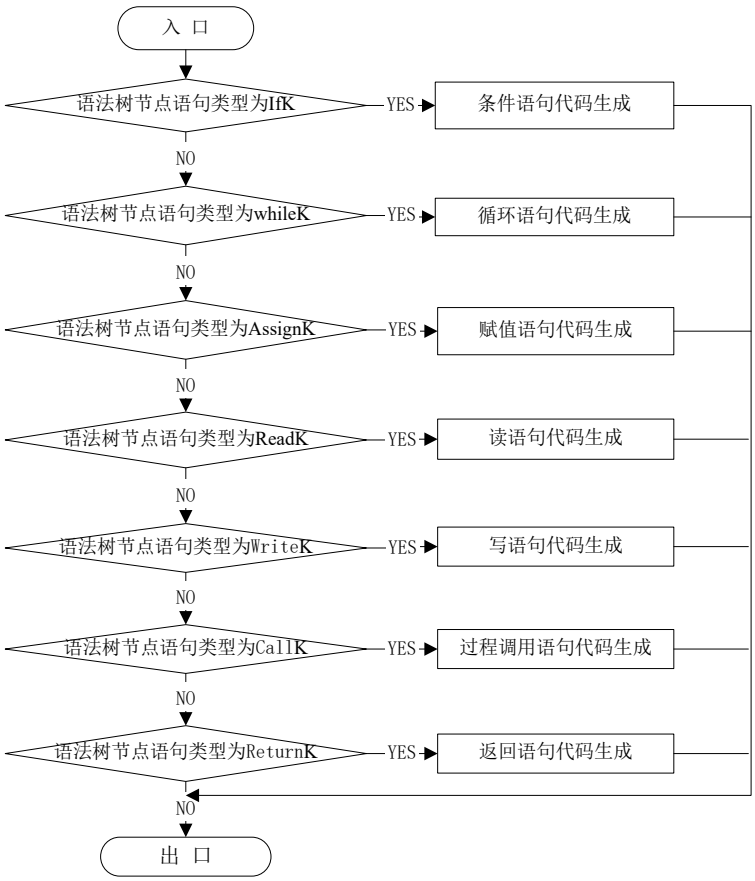


图9.8 语句类型语法树节点代码生成函数genStmt

(1) 条件语句代码生成处理

算法说明: 条件语句 S → if E then S1 else S2 首先计算 E 的值并判断,如果为 TRUE 则执行 S1,而后转向整条语句的下一条 S.OUT;否则转向 S2 的头 S2.START。没有无 else 部分的情况。
算法框图: 见图 9.9。

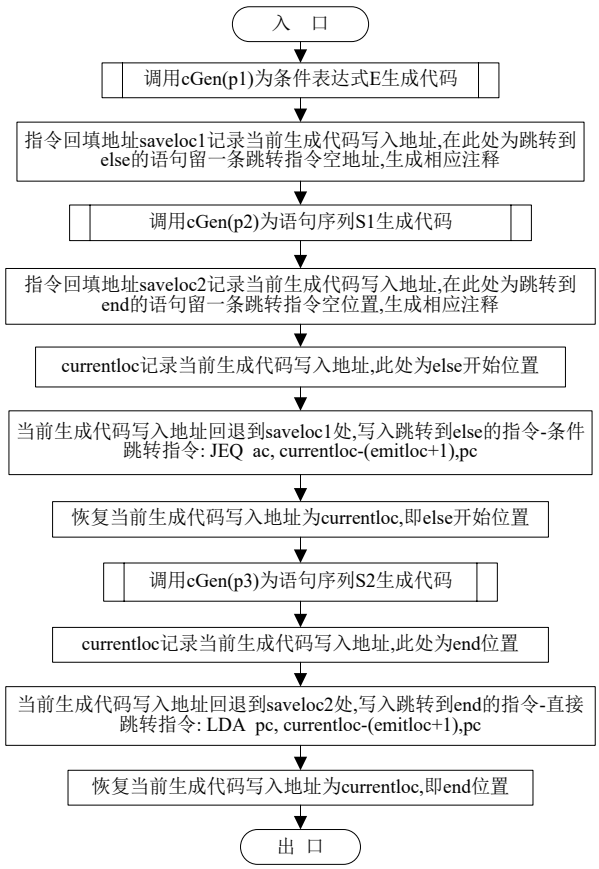


图9.9 条件语句代码生成的算法框图

(2) 循环语句代码生成的处理

语句形式：S → while E do S1

算法说明：首先计算条件表达式 E 的值，如果为 TRUE，则执行重复体语句部分 S1，然后再跳回计算条件表达式 E 的值；如果为 FALSE，则终止循环，执行该循环语句的下面的语句。

算法框图：见图 9.10。

(3) 赋值语句代码生成的处理

语句形式：S → S1 := E

算法说明：首先调用偏移查找函数 FindAddr，求出变量 S1 所对应的存储单元地址，然后再计算赋值表达式 E 的值,将值赋给变量标识符 S1。

算法框图：见图 9.11。

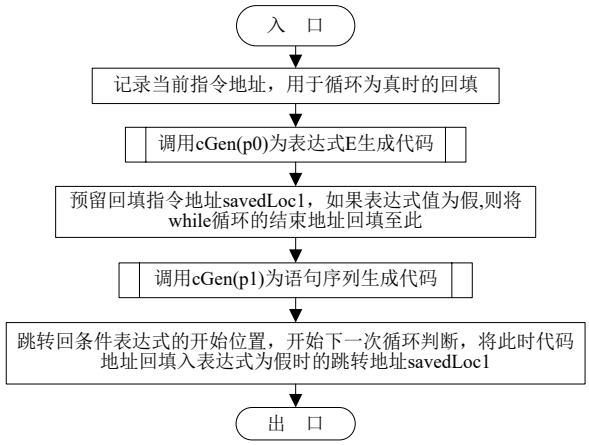


图9.10 循环语句代码生成的算法框图

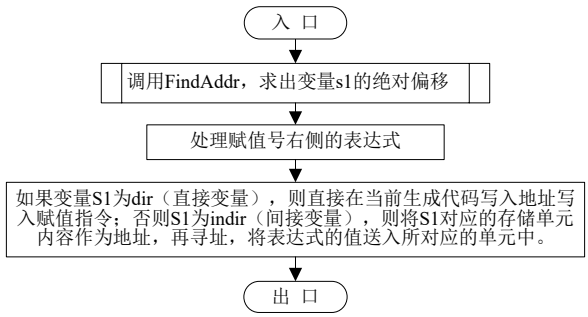


图9.11 赋值语句代码生成的算法框图

(4) 读语句代码生成的处理

语句形式：S → READ(S1)

算法说明：首先将输入的值读入到累加器中,然后再判断该变量是否为间接变量，分类处理。

算法框图：见图 9.12。

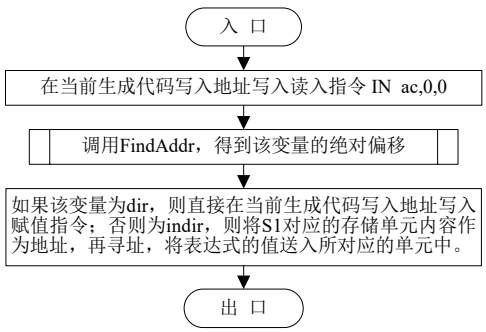


图9.12 读语句代码生成的算法框图

(5) 写语句代码生成的处理

语句形式：S → WRITE(E)

算法说明：首先计算表达式 E 的值,然后输出该值。

算法框图：见图 9.13。

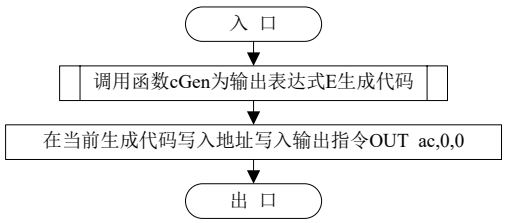


图9.13 写语句代码生成的算法框图

(6) 返回语句代码生成的处理

语句形式：S → RETURN

算法说明：将当前过程活动记录中的过程返回地址值送入当前指令寄存器 pc 中。

算法框图：见图 9.14。

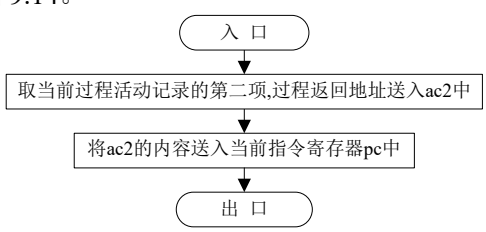


图9.14 返回语句的代码生成算法框图

(7）过程调用语句代码生成的处理

语句形式： S → CALL S1 (param1,param2,……paramN)

算法说明： 首先处理参数传递部分(分 7 种情况处理)，然后再处理过程入口部分，之后转向子过程处理部分，最后再处理过程出口部分。

算法框图： 见图 9.15，其中参数传递过程的 7 种情况框图略。

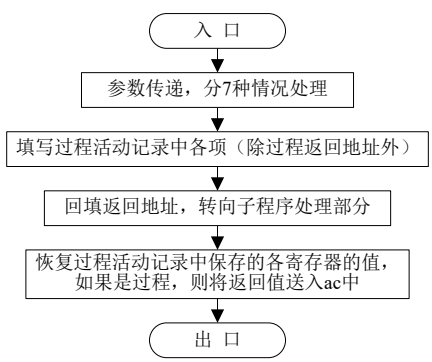


图9.15 过程调用语句代码生成的算法框图

3. 表达式语法树节点代码生成函数

函数声明：void genExp(TreeNode * t)

算法说明：对语法树节点的表达式类型进行分类处理，生成代码。

算法框图：见图 9.16。

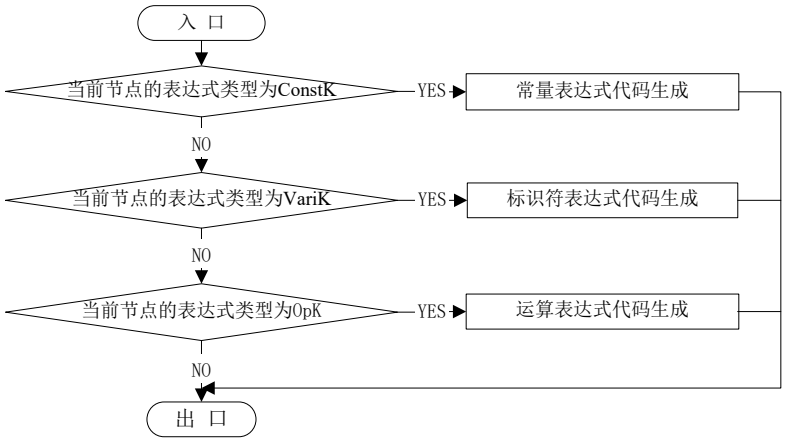


图9.16 表达式语法树节点代码生成主函数genExp的算法框图

（1）常量表达式类型处理

算法说明：调用函数 emitRM("LDC",ac,tree->attr.val,0,"load const");将常量表达式类型语法树节点成员常量整数值 tree->attr.val 载入到累加器 ac。

算法框图：略。

（2）标识符表达式类型处理

算法说明：先查找变量的在数据区的绝对偏移(调用函数 FindAddr),取得变量的目标代码地址。而后再判断该变量是 dir 还是 indir，分情况处理。如果是 dir，则直接使用该地址生成载入指令。否则，按照 FindAddr 找到的地址所取的内容再作为地址取内容送入 ac 中。

算法框图：略。

（3）运算表达式类型处理

算法说明：根据运算表达式中的运算符分类处理,所用临时变量栈式存储于数据存储区的高端。

算法框图：见图 9.17。

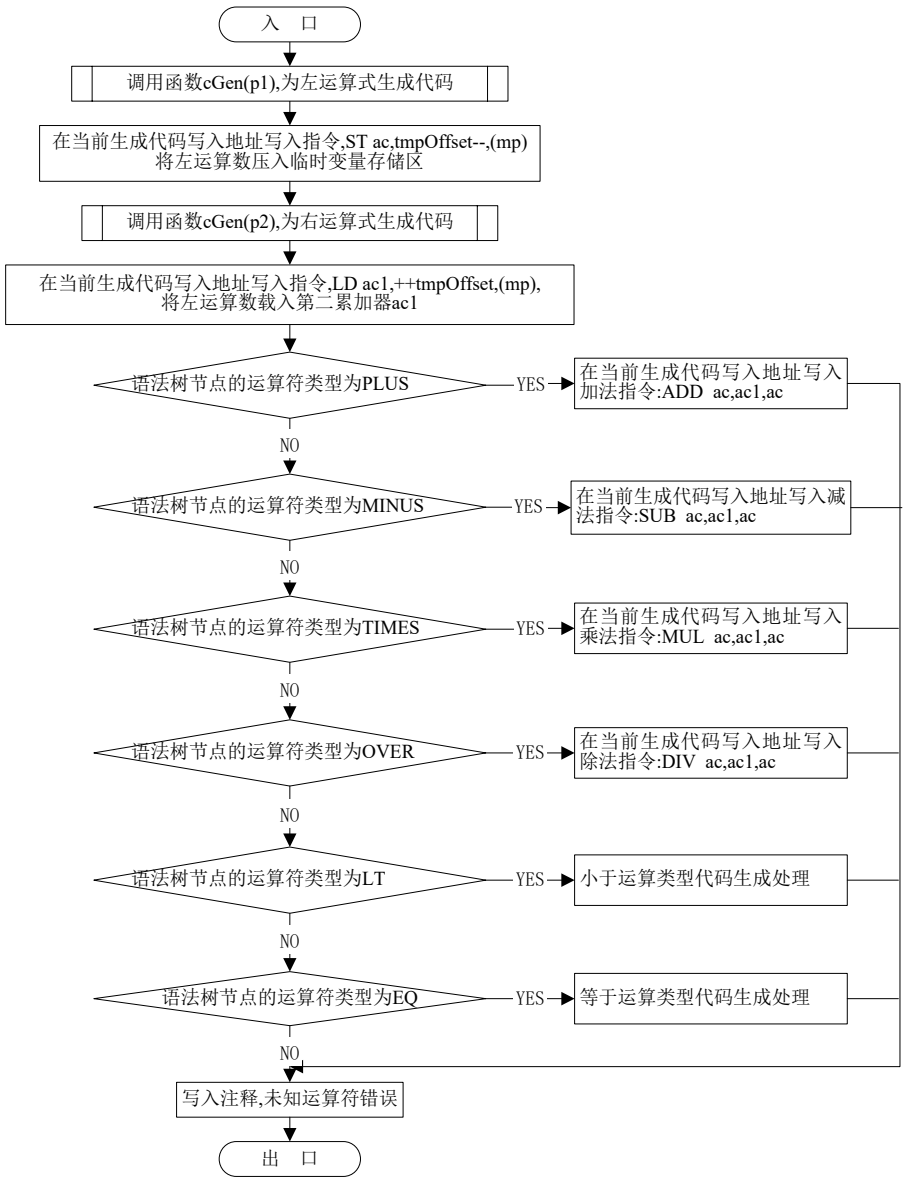


图9. 17 运算表达式类型处理的算法框图

注：运算表达式中的逻辑运算表达式的处理复杂一些,需要先进行比较两个操作数,再设置结果值。如果比较后为真，那么 ac 值为 1，否则 ac 中为 0。

a．小于运算表达式的处理的算法见框图 9.18。

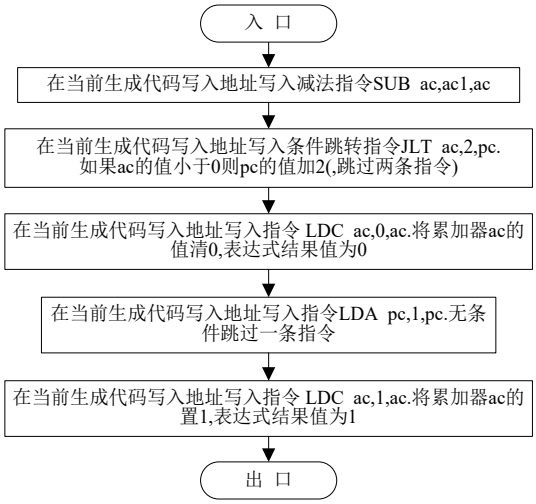


图9.18 小于运算表达式代码生成的算法框图

b．等于运算表达式类型处理算法见框图 9.19。

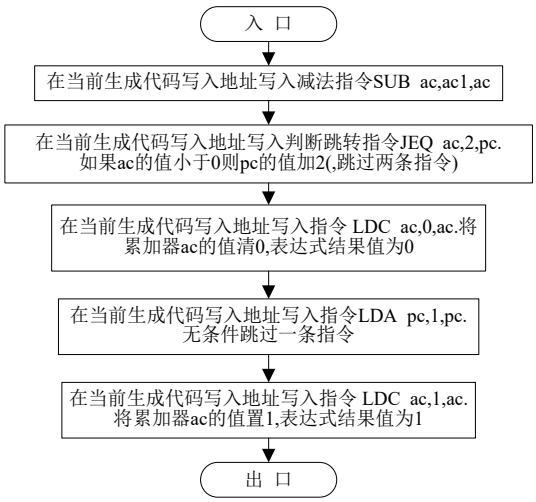


图9.19 等于运算表达式代码生成的算法框图

4． 语法树节点的代码生成函数

函数声明：void cGen(TreeNode * tree)

算法说明：该函数通过遍历语法树,根据语法树的不同类型分别调用不同代码生成函数,递归生成目标代码。

算法框图：见图 9.20。

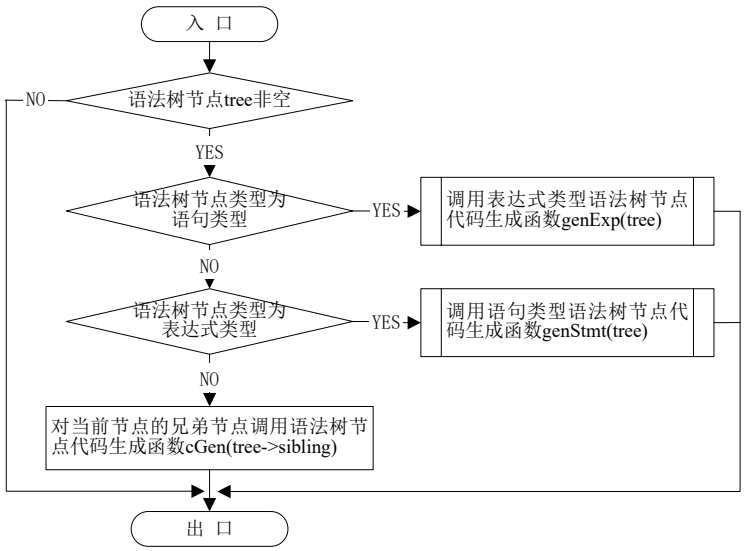


图9.20 语法树节点的代码生成函数cGen的算法框图

5. 代码生成器主函数

函数声明：void codeGen(TreeNode * syntaxTree, char * codefile)

算法说明：为语法分析树生成代码，代码输出到函数参数 codefile 指定的文件。函数为程序加入了一些必要的指令(如设置 mp 指令,程序结束指令)，并调用过程声明部分的代码生成及主程序的代码生成，再将主程序的入口地址送入指令寄存器 pc，最后创建主程序的过程活动记录。

算法框图：见图 9.21。

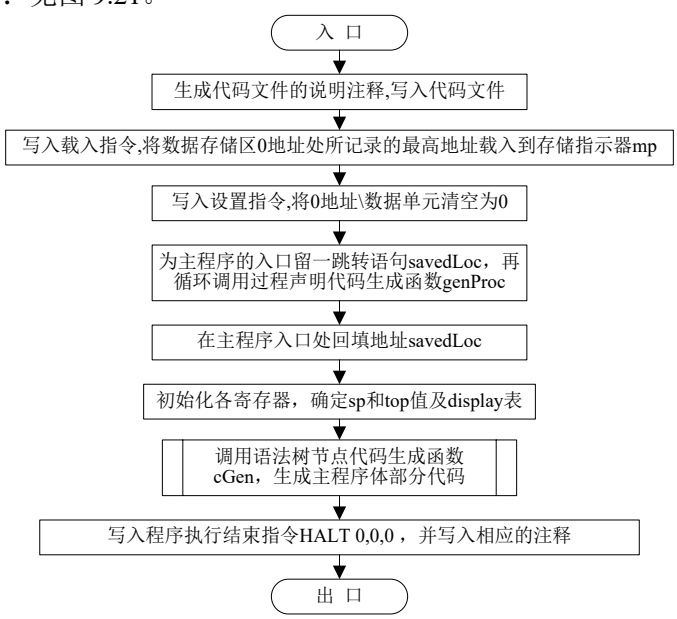


图9.21 代码生成器主函数codeGen的算法框图

9．4　四元式到目标代码的生成

9．4．1　原理

1．取 ARG 结构值对应的目标代码

四元式中间代码的操作分量和目标量以 ARG 结构给出，在生成目标代码的过程中，首先要根据 ARG 结构取得对应的值或者地址，存入累加寄存器 ac 中，再生成运算的目标代码。从 ARG 结构取值的过程如表 9.5 所示：

ARG 结构种类	取值对应的目标代码	
常量 c	LDC，ac，c，0	
标号 lab	LDC，ac，lab，0	
源变量或临时变量 v	直接变量	取 v 的绝对地址到 ac；　(表-2) LD，ac，0，ac
	间接变量	取 v 的绝对地址到 ac；　(表-2) LD，ac1，0，ac LD，ac，0，ac1

表 9.5 取 ARG 结构值对应的目标代码示意表

取变量的绝对地址对应的目标代码如表 9.6 所示：

变量种类	取绝对地址对应的目标代码
源变量 v	LDA，ac，level，displayoff ADD，ac，ac，sp LDC，ac1，off，0 ADD，ac，ac ，ac1 注：level 为变量 v 所在的层数，off 为变量 v 的偏移。
临时变量 t	LDC，ac1，off，0 ADD，ac，sp，ac1 注：off 为临时变量 t 的偏移量

表 9.6 取变量的绝对地址对应的目标代码示意表

3．四元式到目标代码的对应关系

说明：ac,ac1,ac2 三个累加寄存器，用于执行计算；

displayoff 表示记录 sp 到 display 表的偏移的寄存器；
top 寄存器是下一个 AR 指示器，记录下一个可用的地址；
sp 是当前 AR 指示器，记录当前 AR 的始地址；
pc 是程序地址指示器,记录下一条执行指令地址。

种类	中间代码形式	对应的目标代码
算术运算	(ADD, ARG1, ARG2, ARG3)	从 ARG1 取左操作数的值到 ac; LDA, ac2, 0, ac 从 ARG2 取右操作数的值到 ac; LDA, ac1, 0, ac2 **ADD**, ac, ac1, ac LDA, ac2, 0, ac 取 ARG3 的绝对地址到 ac; LDA, ac1, 0, ac2 ST, ac1, 0, ac
	(SUB , ARG1, ARG2, ARG3)	从 ARG1 取左操作数的值到 ac; LDA, ac2, 0, ac 从 ARG2 取右操作数的值到 ac; LDA, ac1, 0, ac2 **SUB**, ac, ac1, ac LDA, ac2, 0, ac 取 ARG3 的绝对地址到 ac; LDA, ac1, 0, ac2 ST, ac1, 0, ac
	(MULT , ARG1, ARG2, ARG3)	从 ARG1 取左操作数的值到 ac; LDA, ac2, 0, ac 从 ARG2 取右操作数的值到 ac; LDA, ac1, 0, ac2 **MUL**, ac, ac1, ac LDA, ac2, 0, ac 取 ARG3 的绝对地址到 ac LDA, ac1, 0, ac2 ST, ac1, 0, ac
	(DIV , ARG1, ARG2, ARG3)	从 ARG1 取左操作数的值到 ac; LDA, ac2, 0, ac 从 ARG2 取右操作数的值到 ac; LDA, ac1, 0, ac2 **DIV**, ac, ac1, ac LDA, ac2, 0, ac 取 ARG3 的绝对地址到 ac； LDA，ac1，0，ac2 ST，ac1，0，ac

<table>
<tr><td rowspan="2">关系运算</td><td>(EQC, ARG1, ARG2, ARG3)</td><td colspan="2">从 ARG1 取左操作数的值到 ac;
LDA,ac2,0,ac
从 ARG2 取右操作数的值到 ac;
LDA,ac1,0,ac2
SUB,ac,ac1,ac
JLT,ac,2,pc
LDC,ac,0,0
LDA,pc,1,pc
LDC,ac,1,0</td></tr>
<tr><td>(LTC, ARG1, ARG2, ARG3)</td><td colspan="2">从 ARG1 取左操作数的值到 ac;
LDA,ac2,0,ac
从 ARG2 取右操作数的值到 ac;
LDA,ac1,0,ac2
SUB,ac,ac1， ac
JEQ,ac,2,pc
LDC,ac,0,0
LDA,pc,1,pc
LDC,ac,1,0</td></tr>
<tr><td rowspan="5">语句</td><td rowspan="2">(READC, ARG1, _ , _)</td><td>ARG1 是
直接变量</td><td>IN,ac2,0,0
取 ARG1 的绝对地址到 ac；
ST,ac2,0,ac</td></tr>
<tr><td>ARG1 是
间接变量</td><td>IN,ac2,0,0
取 ARG1 的绝对地址到 ac；
LD,ac1,0,ac
ST,ac2,0,ac1</td></tr>
<tr><td>(WRITEC, ARG1, _ , _)</td><td colspan="2">从 ARG1 取要输出的值到 ac;
OUT,ac,0,0</td></tr>
<tr><td>(RETURNC, _ , _ , _)</td><td colspan="2">同下面(ENDPROC,_ , _ , _)</td></tr>
<tr><td>(ASSIG, ARG1 , ARG2 , _)</td><td colspan="2">从 ARG2 取赋值左部的地址，存入 ac；(表-2)
LDA,ac2,0,ac
从 ARG1 取赋值右部的值，存入 ac;(表-1)
⇩
ARG2 是直接变量 ST,ac,0,ac2
ARG2 是间接变量 LD,ac1,0,ac2
ST,ac,0,ac1</td></tr>
</table>

<table>
<tr><td rowspan="6"></td><td rowspan="3">(AADD, ARG1, ARG2, ARG3)</td><td>ARG1 是直接变量</td><td>取 ARG1 的地址到 ac;</td></tr>
<tr><td>ARG1 是间接变量</td><td>取 ARG1 的地址到 ac;
LDA,ac2,0,ac</td></tr>
<tr><td colspan="2">⇩
从 ARG2 取值(偏移量)到 ac;
ADD,ac2,ac2,ac
取 ARG3 的绝对地址到 ac;
ST,ac2,0, ac</td></tr>
<tr><td rowspan="2">(LABEL , ARG1 , _ , _)</td><td>前面无标号的使用</td><td>空</td></tr>
<tr><td>前面有标号的使用</td><td>在标号的使用处回填目标代码:
LDC,pc,currentLoc,0
注 : currentLoc 是当前目标代码编号</td></tr>
<tr><td rowspan="2">(JUMP, ARG1 , _ , _)</td><td>前面无对 ARG1 表示的标号的定位</td><td>空</td></tr>
<tr><td>前面有对 ARG1 表示的标号的定位</td><td>LDC,pc,destNum,0
注:destNum 为标号定位所在目标代码号</td></tr>
<tr><td>(JUMP0 , ARG1, ARG2, _)</td><td colspan="2">从 ARG1 取值，存入 ac 中;
LDA,ac2,0,ac
JNE,ac2,notLoc
按照(JUMP,ARG2,_,_)处理跳转部分。
注: notLoc 是处理完跳转部分(JUMP,ARG2,_,_)后的代码地址</td></tr>
</table>

<table>
<tr><td rowspan="4"></td><td>(CALL, ARG1, _ , ARG3)</td><td colspan="2">ST,displayOff,6,top
LDC,displayOff,Noff,0
LDC,ac,returnLoc,0
ST,ac,1,top
按照(JUMP, ARG1,_,_)处理跳转。
注：Noff 是从 ARG3 取得的新的 display 表的偏移; returnLoc 是过程调用的返回地址</td></tr>
<tr><td rowspan="3">(VARACT, ARG1 ,ARG2 , _)</td><td>形参 ARG1 是直接变量</td><td>取形参 ARG1 的地址，存入 ac;
ST,ac, paramoff, top</td></tr>
<tr><td>形参 ARG1 是间接变量</td><td>取形参 ARG1 的地址，存入 ac;
LD,ac,0,ac
ST,ac, paramoff, top</td></tr>
<tr><td colspan="2">注：notLoc 是处理完跳转部分(JUMP,ARG2,_,_)后的代码地址)</td></tr>
<tr><td>(VALACT, ARG1 , ARG2, _)</td><td colspan="2">从 ARG1 中取得实参的值，存入 ac;
ST,ac, paramoff, top
注：paramoff 是从 ARG2 取得的形参的偏移量)</td></tr>
<tr><td>定位</td><td>(PENTRY, ARG1,ARG2 , ARG3)</td><td colspan="2">按照(LABEL,ARG1, _,_)处理标号;
ST,sp,0,top
ST,ac,3,top
ST,ac1,4,top
ST,ac2,5,top
LDC,ac,procLevel
ST,ac,2,top
复制当前 AR 的 display 表到新的 AR; (略)
LDA,ac,procLevel,displayOff
ADD,ac,top,ac
ST,top,0,ac
LDA,sp,0,top
LDA,top,ARsize,top</td></tr>
</table>

	(ENDPROC, _ , _ , _)	(恢复现场和返回) LD,ac,3,sp LD,ac1,4,sp LD,ac2,5,sp LD,displayOff,6,sp LDA,top,0,sp LD,sp,0,sp LD,pc,1,top
	(MENTRY, _ ,ARG2 , ARG3)	在第一条目标代码处写入: LDC,pc,mainLoc,0 在当前位置(初始化部分): LDC,ac,0,0, LDC,ac1,0,0 LDC,ac2,0,0 ST,ac,0,sp, LDC,displayOff,Noff,0 ST,ac,0,displayOff LDA,top, size, sp 注:Noff 是从 ARG3 取得的 display 表的偏移量; size 是从 ARG2 取得的主程序 AR 的大小。

表 9.7 四元式到目标代码的对应关系示意表

9．4．2 四元式到目标代码生成中的关键问题

1． 标号和跳转的处理：处理标号和跳转，需要用到标号地址表；表的结构为：

中间代码标号	目标代码地址	下一项
Label	destnum	next

(1) 遇到标号定位时：设标号为 L，应转向的目标代码为 p1，分为两种情况：
- 在标号地址表中没有 L 项，则填写表项(L，p1,NULL)，并链入表尾；
- 在标号地址表中有 L 项(L，addr，Next)，则根据当前 pc，回填 addr 对应的目标代码。

(2) 遇到跳转代码时：设要跳到的标号为 L，这条语句对应的目标地址为 p2，分为两种情况：
- 在标号地址表中没有 L 项，则构造一个表项(L，p2，NULL)，链入表尾；
- 在标号地址表中有 L 项(L，p1，next)，则从中取出 L 的代码地址 p1，直接生成目标代码。

2．形实参结合的处理：

(1) 形参为值参：

- 实参是常数值：将常数值送入相应存储单元；
- 实参是直接变量：找到变量的存储地址，取值，送相应存储单元；
- 实参是间接变量：此时变量的存储单元存放的是地址。找到变量的存储地址，取内容作为地址，再取内容，得到实参值，送相应存储单元；

(2) 形参为变参：这时实参必须是变量，长度为 1。

- 实参是直接变量：应将实参变量的地址送入形参单元；找到变量的存储地址，送相应存储单元；
- 实参是间接变量：要送实参变量单元的内容；找到变量的存储地址，取内容，送相应存储单元。

3． 过程调用的工作分配：

由于过程调用中的有些工作是相同的，如果都放在调用语句处处理，当一个过程被多个过程调用时，就要重复很多目标代码。从语法树生成目标代码时，所有过程调用的处理工作都必须放在调用语句处进行处理。而从中间代码生成目标代码时，由于有过程入口和过程出口中间代码的存在，我们就可以把有些工作安排在子程序的入口和出口处完成。这样，过程调用中的整个工作(除过程体的执行部分)可分配到过程调用处，过程入口处和过程出口处。这样就可以节省一些目标代码。具体分配如下：

- 过程调用代码完成：保存旧的 display 表的偏移量；
  设置新的 display 表的偏移；
  保存返回地址；
  转向过程入口。
- 过程入口处完成：保存 sp 值；
  保存层数；
  保存累加寄存器的内容，ac，ac1，ac2
  构造 NewAR 的 Display 表；
  修改 sp 和 top 值：sp:=top ； top:= top+NewAR.Size
- 过程出口处完成：恢复寄存器的内容；
  恢复 sp 和 top 的值：top:=sp；sp:=CurrentAR.sp；
  取出返回地址，返回。

9．4．3 程序框图

1．几个实用函数：

(1) 计算操作数数值的函数：
函数声明：void operandGen(ArgRecord *arg)
算法说明：参数是中间代码中的一项，计算结果存在寄存器 ac 中。

算法框图：见图 9.22。

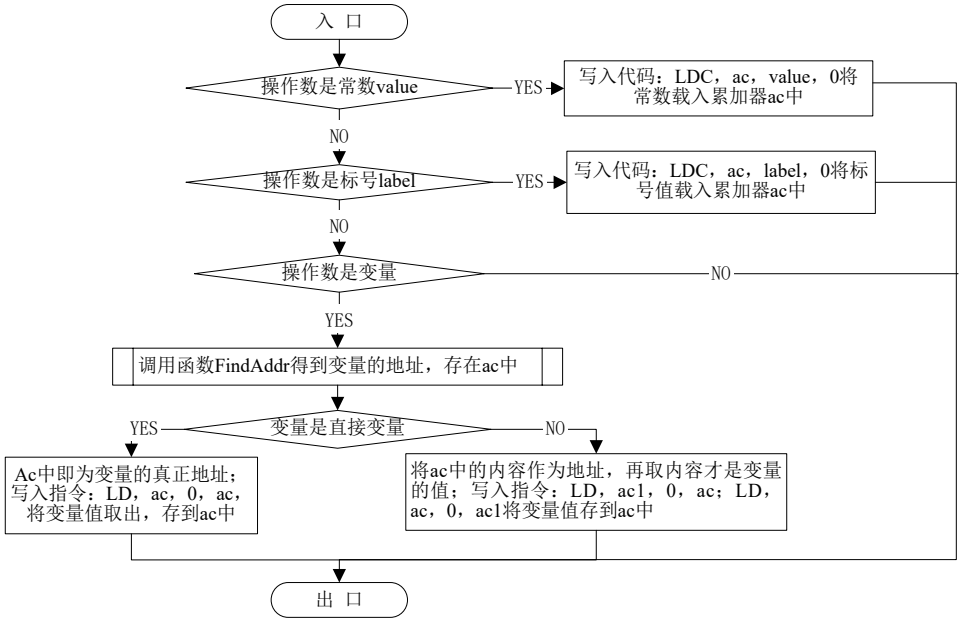


图9.22 计算操作数值的函数operandGen的算法框图

(2) 计算变量地址的函数：

函数声明：void FindAddr(ArgRecord *arg)

算法说明：将由 arg 指出的变量(包括源变量和临时变量)所在的绝对地址计算出来，存入寄存器 ac 中返回。

算法框图：见图 9.23。

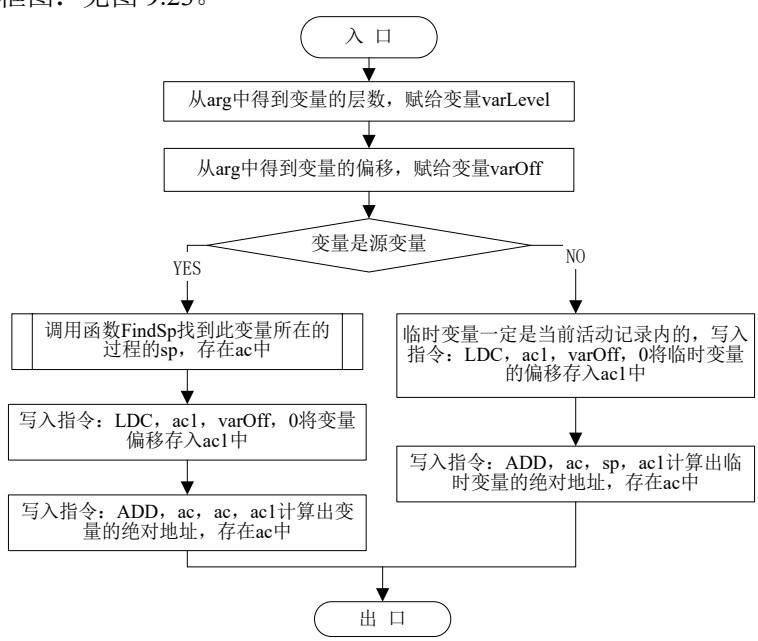


图9.23 计算变量地址的函数FindAddr的算法框图

(3) 计算变量所在的过程活动记录的首地址的函数

函数声明：void FindSp(int varlevel)

算法说明：找到该变量所在 AR 的 sp,存入 ac 中。

算法框图：见图 9.24。

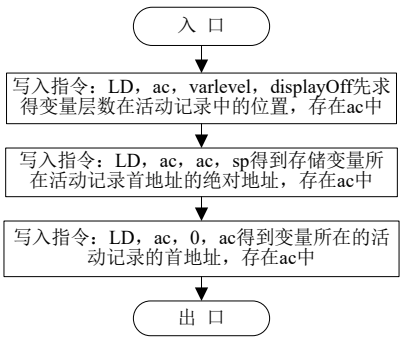


图9.24 计算变量所在的过程活动记录的首地址值的函数FindSp的算法框图

2．目标代码生成函数：

(1) 目标代码生成主函数

函数声明： void codeGen(CodeFile *midcode, char * destcode)

算法说明：该函数通过扫描中间代码序列产生目标代码。文件第二个参数 codefile 为目标代码文件名。

算法框图： 见图 9.25。

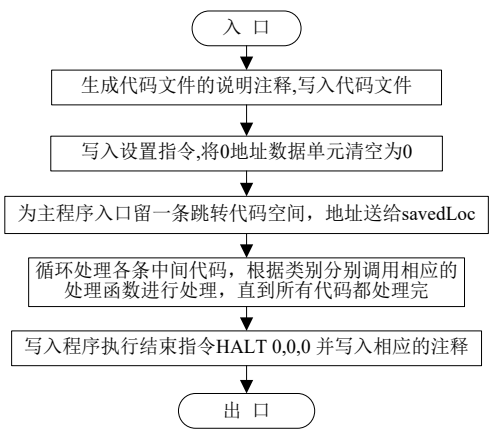


图9.25 目标代码生成主函数codeGen的算法框图

根据中间代码类别选择调用函数部分框图 9.26。

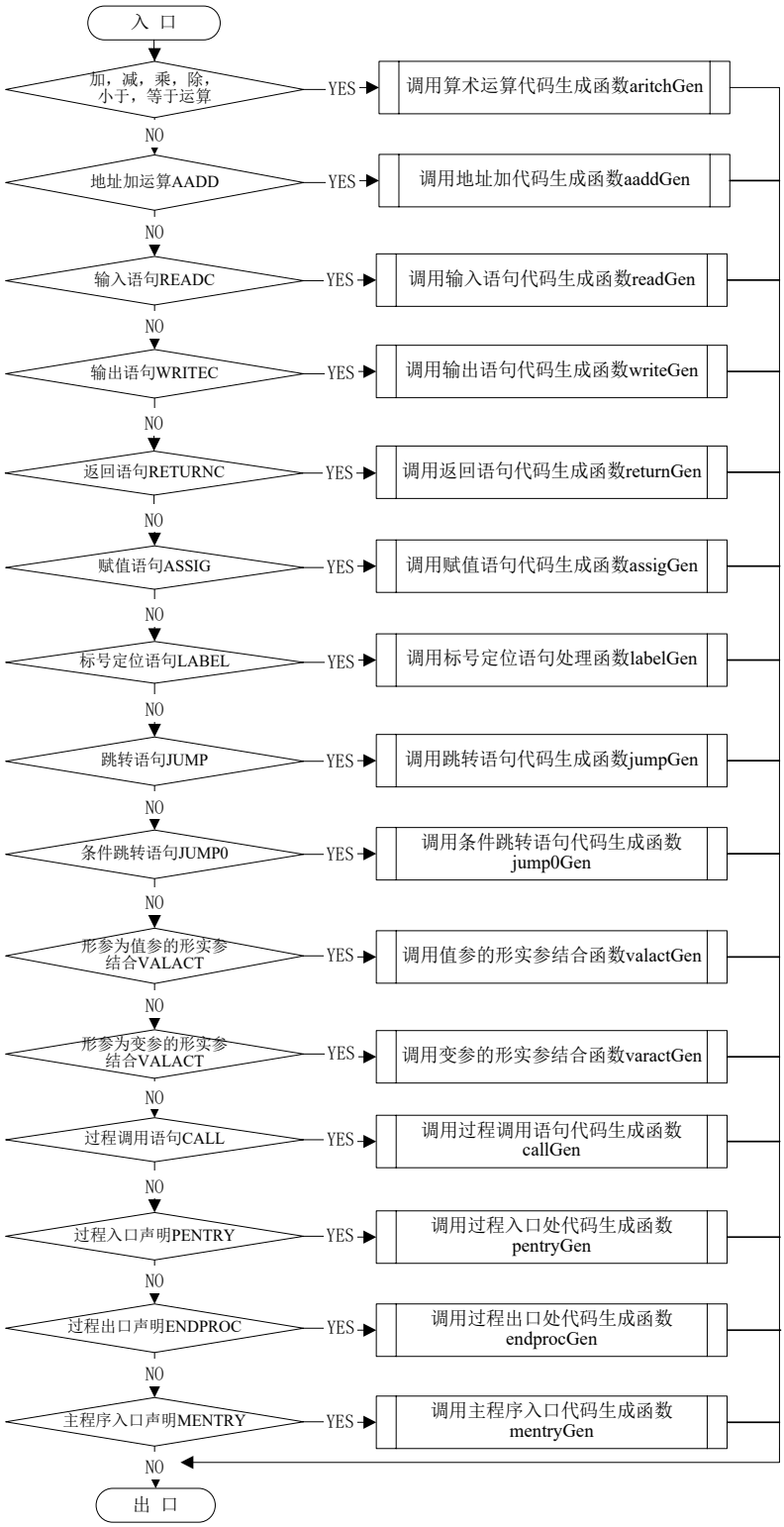


图9.26　根据中间代码类别选择调用函数的算法框图

(2) 运算的目标代码生成函数

函数声明： void arithGen(CodeFile *midcode)

算法说明： 分别生成左、右操作数的目标代码，最后根据操作符生成运算的目标代码。

算法框图： 见图 9.27。

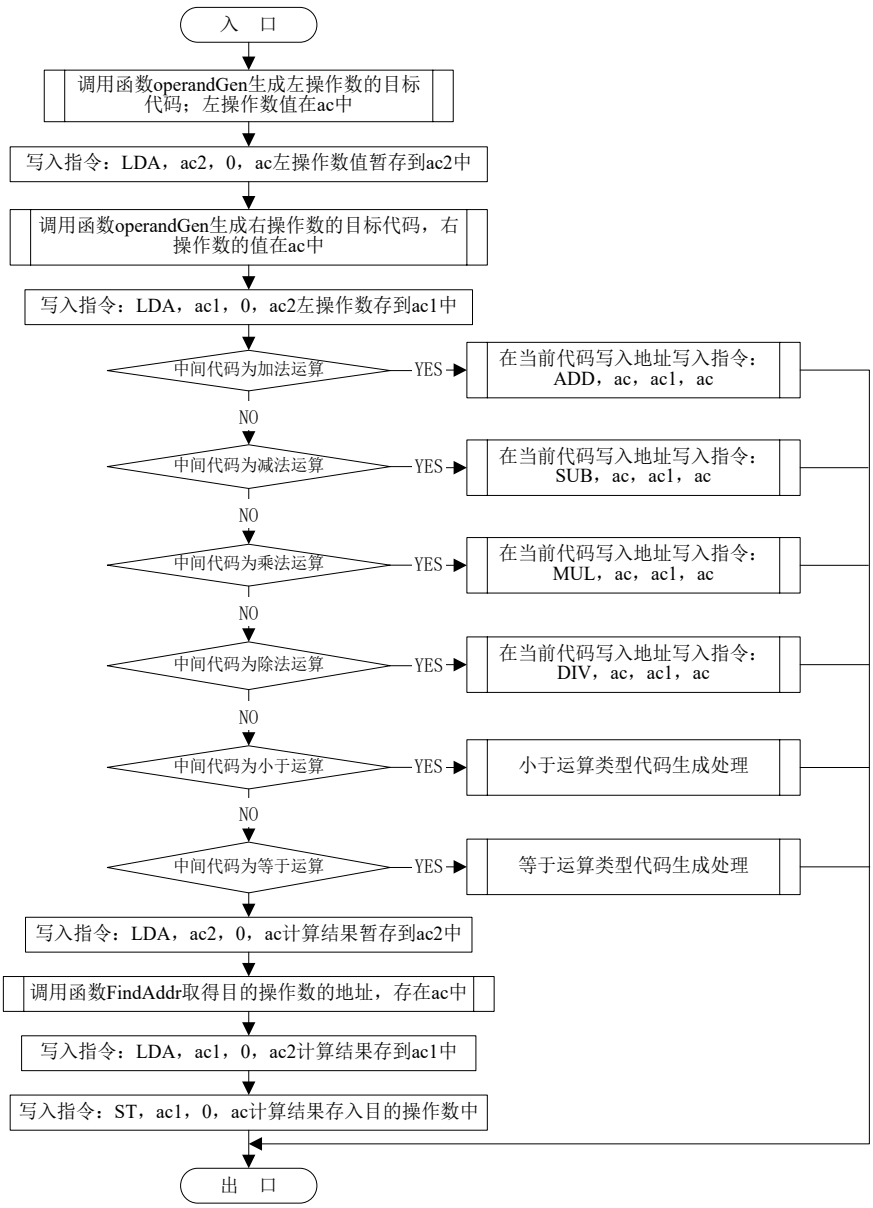


图9.27 运算的目标代码生成函数arithGen的算法框图

(3) 地址加操作的目标代码生成函数

函数声明：void aaddGen(CodeFile *midcode)

算法说明：取出基地址和偏移量相加，暂存到寄存器中，再将此地址相加的结果写入目标变量。

算法框图：见图 9.28。

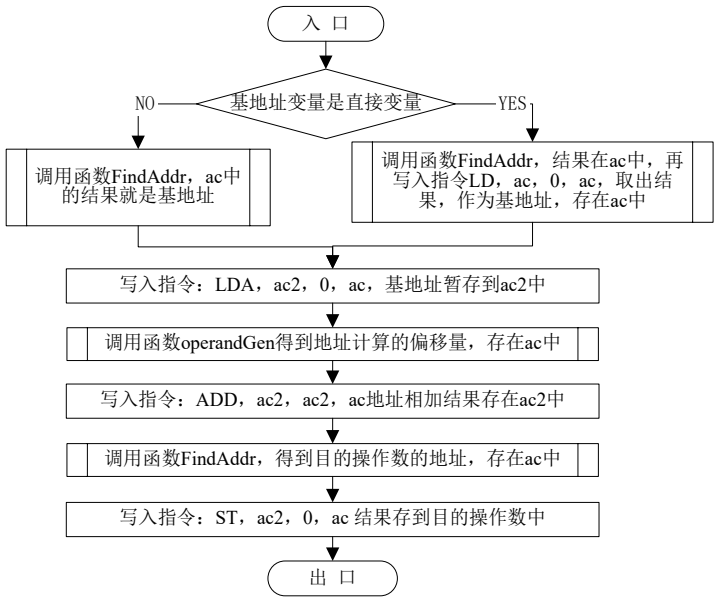


图9.28 地址加操作的目标代码生成函数arithGen的算法框图

(4) 读操作的目标代码生成函数

函数声明：void readGen(CodeFile *midcode)

算法说明：根据变量是直接变量还是间接变量进行不同的处理。

算法框图：见图 9.29。

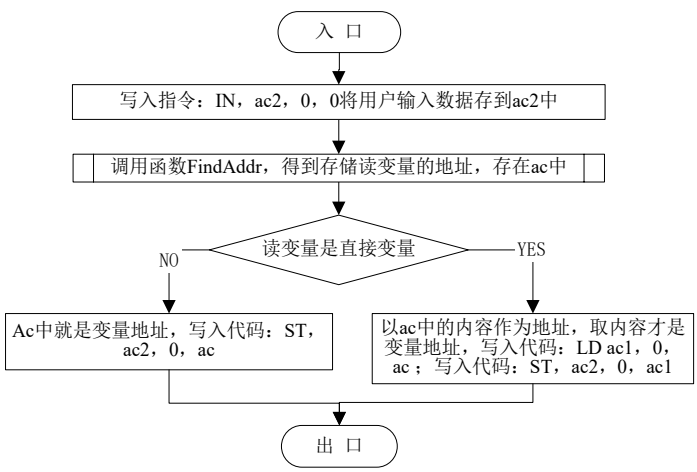


图9.29 读操作的目标代码生成函数readGen的算法框图

(5) 写操作的目标代码生成函数

函数声明：void writeGen(CodeFile *midcode)

算法说明：调用取值函数得到要输出的值，并产生输出代码。

算法框图：见图 9.30。

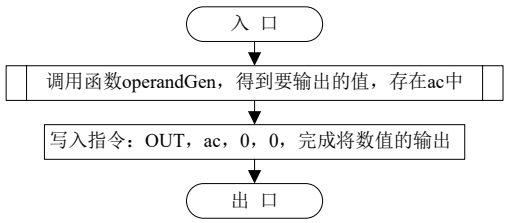


图9.30 写操作的目标代码生成函数writeGen的算法框图

(6) 返回语句的目标代码生成函数

函数声明： void returnGen(CodeFile *midcode)

算法说明：遇到 return 语句，过程结束，故这里调用过程出口处理函数 endprocGen 完成过程结束的恢复操作。

算法框图： 略。

(7) 赋值语句的目标代码生成函数

函数声明：void assigGen(CodeFile *midcode)

算法说明：取得赋值左部变量的地址；生成赋值右部的目标代码；并根据左部是直接变量还是间接变量生成不同的赋值的目标代码。

算法框图：见图 9.31。

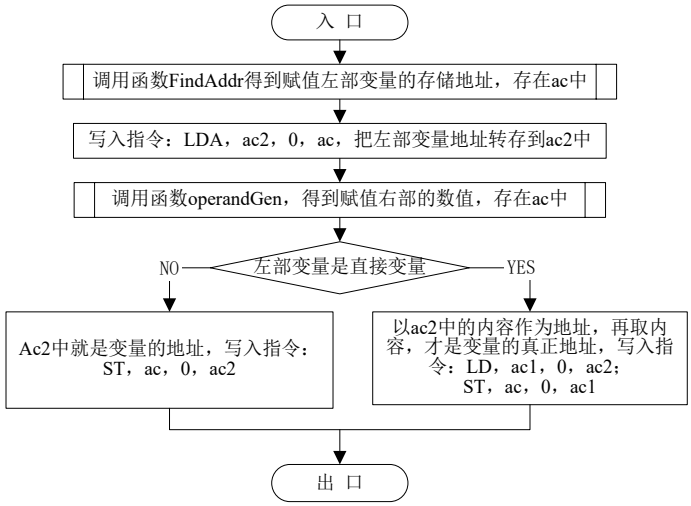


图9.31 赋值语句的目标代码生成函数assignGen的算法框图

(8) 标号定位语句的目标代码生成函数

函数声明： void labelGen(CodeFile *midcode)

算法说明： 查找标号地址表，若表中没有此标号对应的项,填表；若表中有此标号项，则回填目标代码。

算法框图： 见图 9.32。

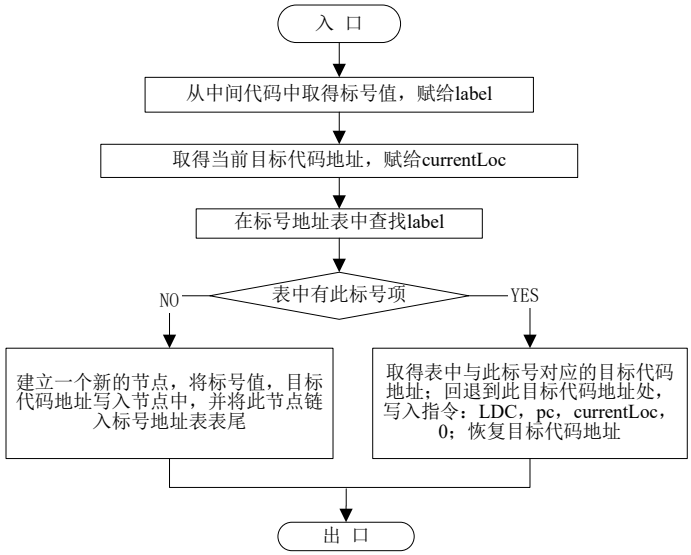


图9.32 标号定位语句的目标代码生成函数labelGen的算法框图

(9) 跳转语句的目标代码生成函数

函数声明： void jumpGen(CodeFile *midcode, int i)

算法说明： 参数 i 是为了复用此函数而设，根据 i 决定从中间代码中哪个分量取标号值。

算法框图： 见图 9.33。

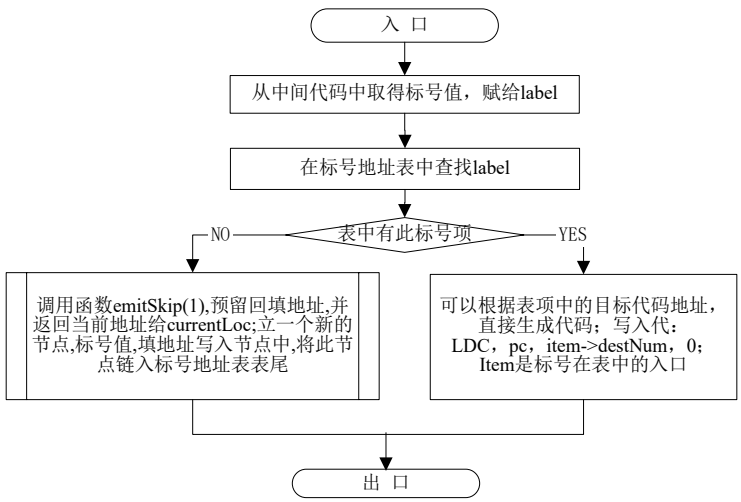


图9.33 跳转语句的目标代码生成函数jumpGen的算法框图

(10) 条件跳转语句的目标代码生成函数

函数声明： void jump0Gen(CodeFile *midcode, int i)

算法说明： 通过跳转语句的目标代码生成函数实现此函数。

算法框图： 见图 9.34。

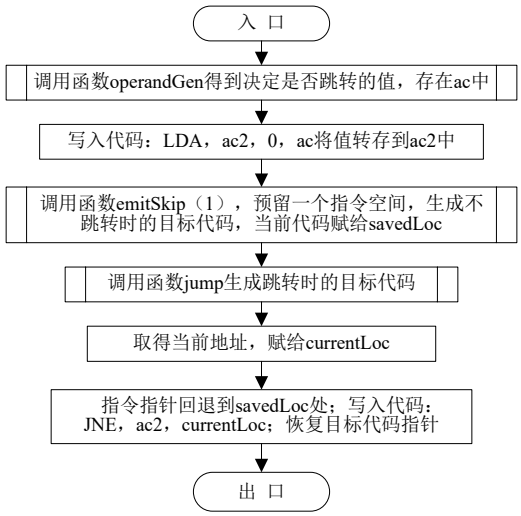


图9.34 条件跳转语句的目标代码生成函数jump0Gen的算法框图

(11) 形参为值参的形实参结合语句的目标代码生成函数

函数声明：void valactGen(CodeFile *midcode)

算法说明：取得形参的偏移，取得实参的值，生成形实参结合的目标代码。

算法框图：见图 9.35。

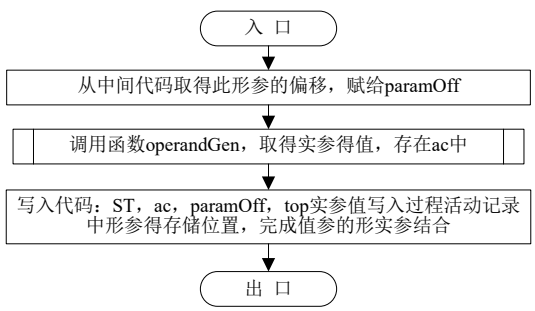


图9.35 形参为值参的形式参结合语句的目标代码生成函数valactGen的算法框图

(12) 形参为变参的形实参结合语句的目标代码生成函数

函数声明： void varactGen(CodeFile *midcode)

算法说明： 取得形参的偏移，根据实参是直接变量还是间接变量生成不同的取得实参地址的代码，最后生成形实参结合的目标代码。

算法框图： 见图 9.36。

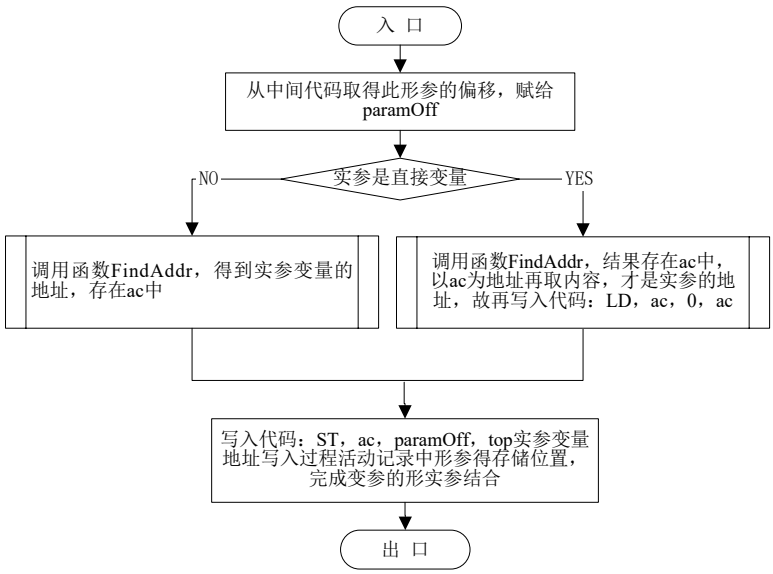


图9.36 形参为变参的形式参结合语句的目标代码生成函数varactGen的算法框图

(13) 过程调用语句的目标代码生成函数

函数声明：void callGen(CodeFile *midcode)

算法说明：为了节约目标代码，将过程调用中的整体工作分配到三部分：过程调用处、过程入口处和过程出口处。

算法框图：见图 9.37。

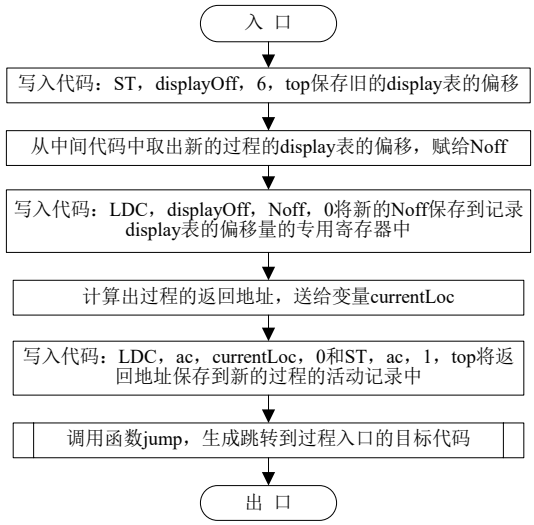


图9.37 过程调用语句的目标代码生成函数callGen的算法框图

(14) 过程入口语句的目标代码生成函数

函数声明：void pentryGen(CodeFile *midcode)

算法说明：过程入口中间代码中，ARG1 是过程入口标号，ARG2 是 display 表的偏移量，ARG3 是过程的层数。

算法框图：见图 9.38。

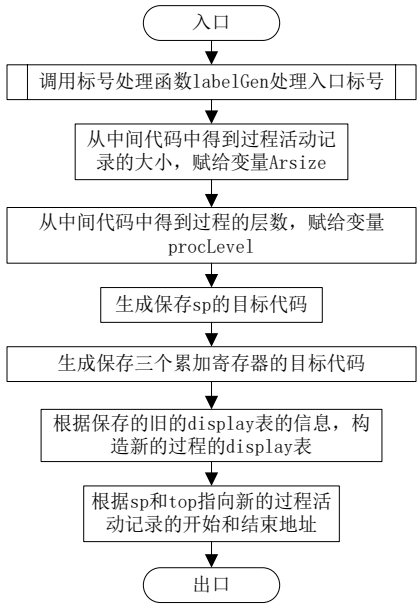


图9.38 过程入口语句的目标代码生成函数pentryGen的算法框图

(15) 过程出口语句的目标代码生成函数

函数声明：void endprocGen(CodeFile *midcode)

算法说明：恢复寄存器值；恢复 sp 和 top 值；取出返回地址，返回。

算法框图：见图 9.39。

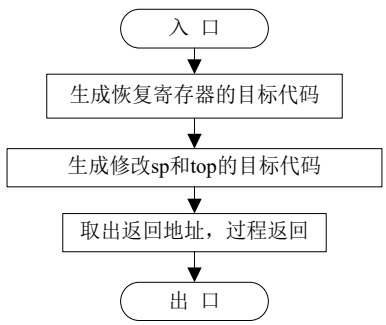


图9.39 过程出口语句的目标代码生成函数endprocGen的算法框图

(16) 主程序入口语句的目标代码生成函数

函数声明：void mentryGen(CodeFile *midcode ,int savedLoc)

算法说明：主程序入口中间代码的 ARG2 记录了主程序 AR 的 display 表的偏移；参数 savedLoc 记录跳转改变 pc 的指令应在的位置。

算法框图：见图 9.40。

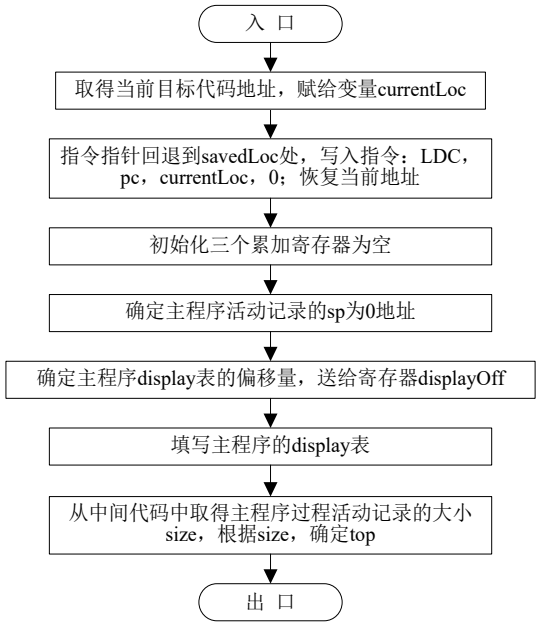


图9.40 主程序入口语句的目标代码生成函数mentryGen的算法框图

第十章 虚拟目标代码的解释程序

10．1 解释程序

解释程序是源程序的一个执行程序，它把源程序（或者源程序的某种中间表示）看成是自己的部分输入，而源程序原来的输入 Input 看成是另一部分输入，最终输出的是源程序在输入 Input 下的运行结果。

在这里，我们要实现虚拟目标机 TM 的指令系统的解释程序，它读入含有 TM 的指令序列的文件，并依次对每条指令进行解释执行，这里源程序没有输入。

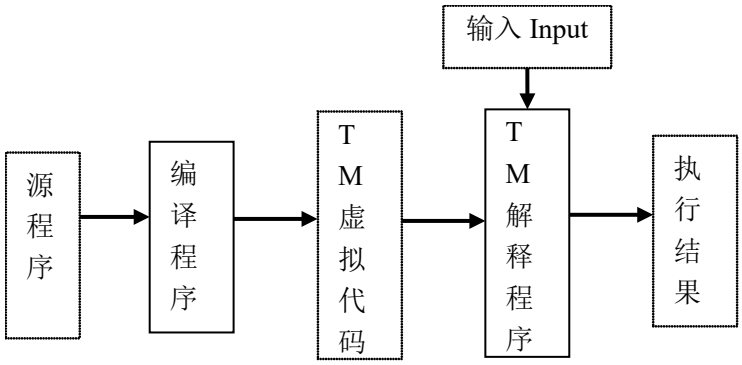


图 10.1 解释执行过程

10．2 虚拟目标机 TM 的可执行命令

TM 虚拟机的指令系统在第九章已经做了介绍，这里不再重复，主要介绍 TM 的可执行命令。

TM 机可执行命令列表:

S(tep) 按步执行(step)命令:可输入"s n"来执行,可执行 n(默认为 1)条 tm 指令。
G(o) 执行到结束(go)命令:可输入"g"来执行,顺序执行 tm 指令直到遇到 HALT 指令。
R(egs) 显示寄存器(regs)命令:可输入"r"来执行,显示各寄存器的内容。
I(Mem) 输出指令(iMem)命令:可输入"i b n"来执行,从地址 b 处输出 n 条指令。
D(Mem) 输出数据(dMem)命令:可输入"d b n"来执行,从地址 b 处输出 n 条数据。
T(race) 跟踪(trace)命令:可输入"t"来执行,反置追踪标志 traceflag,如果 traceflag 为 TRUE,则执行每条指令时候显示指令。

P(rint) 显示执行指令数量(print)命令:可输入"p"来执行,反置追踪标志icountflag,如果 icountflag 为 TRUE,则显示已经执行过的指令数量。只在执行"go"命令时有效。

C(lear) 重置 tm 机用(clear)命令:可输入"c"来执行,重新设置 tm 虚拟机,用以执行新的程序。

H(elp) 帮助(help)命令:可输入"h"来执行,显示命令列表。

Q(uit) 终止(quit)命令,可输入"q"来执行,结束虚拟机的执行。

10．3 解释程序的实现

10．3．1 解释程序的实现框图

目标代码的解释执行程序首先将目标代码文件中的指令读入 TM 虚拟机中的指令存储器 iMem。由于读入的是目标代码的文本文件，所以需要对文本字符流进行识别,将文本形式转化为指令结构才能进行存储。就像编译器的词法扫描器一样,需要具有识别指令的功能部分。此功能由 tm.c 文件中定义的函数 readInstructions 完成。

指令全读入之后就可以进行解释执行了。TM 虚拟机提供了各种命令,由函数 doCommand 解释执行。可用它们对存储在指令存储器中的指令进行操作。

主要函数之间调用关系如下：

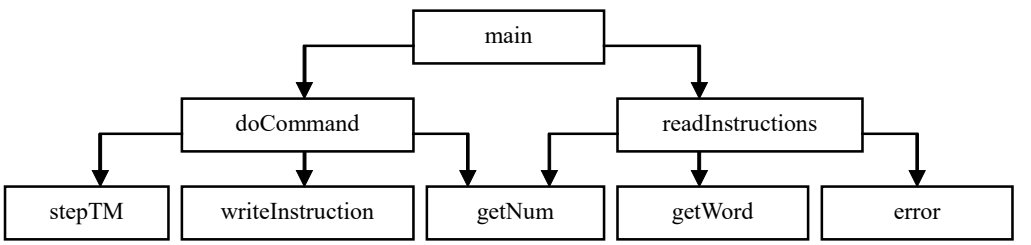


图 10.2 解释程序的调用关系图

所用各函数详细说明和算法框图如下：

1． 指令输出函数 writeInstruction

函数声明：void writeInstruction (int loc)

函数功能：函数将参数 loc 指定地址上的指令按寻址模式分类,以指定格式输出到屏幕。

算法框图：略。

2． 数值获取函数 getNum

函数声明：int getNum (void)

函数功能：函数将在输入缓冲区中连续出现的有加减运算的数 term 合并计数,所得数值送入 num。如果成功得到数值,则函数返回 TRUE,否则函数

返回 FALSE。

算法框图：略。

3. 单词获取函数 getWord

函数声明： int getWord (void)

函数功能： 函数将输入缓冲区中当前字符序列组合生成一个单词。如果成功,则函数返回 TRUE,否则,函数返回 FALSE。

算法框图： 略。

4. 字符略过函数 skipCh

函数声明： int skipCh (char c)

函数功能： 该函数略过指定的字符,如果成功略过,则函数返回值为 TRUE,否则为 FALSE。

算法框图： 略。

5. 行结束判断函数 atEOL

函数声明： int atEOL(void)

函数功能： 该函数判断当前位置是否为行的结尾,是则函数返回值为 TRUE,否则为 FALSE。

算法框图： 略。

6. 错误处理函数 error

函数声明： int error(char * msg, int lineNo, int instNo)

函数功能： 函数输出信息包括:错误行号,指令地址标号和错误信息。分别由函数参数 msg,lineNo,instNo 指定。

算法框图： 略。

7. 读入指令函数 readInstructions

函数声明： int readInstructions (void)

函数功能： 函数从指令文件中的字符流中识别指令,并根据其所属寻址模式,读入到指令存储区 iMem 中。

算法框图： 略。

■ 指令读入函数 readInstructions 中的指令识别处理部分：

对目标代码文件输入的字符流处理。按格式“地址标号 : 操作码 操作数部分”识别指令,类似于词法扫描和语法分析操作。其中对操作数部分,根据指令操作码所属的寻址模式具体处理。

- 寄存器-寄存器操作数识别

  寄存器-寄存器操作数部分按格式“操作数 1,操作数 2,操作数 3”识别处理,三个操作数指明所用寄存器,应为 0-7 之间整数。

- 寄存器-内存;寄存器-立即数操作数识别

  寄存器-内存操作数部分按格式“操作数 1,操作数 2(操作数 3)”识别处理。寄存器-立即数操作数部分按格式“ 操作数 1,操作数 2,操作数 3”识别处理,第一,第三操作数指明所用寄存器,应为 0-7 之间整数。第二操作数为数据存储区单元偏移地址,应为 0-1023。

8．TM 机单步执行函数 stepTM

函数声明： STEPRESULT stepTM (void)

函数功能： 函数从指令存储区 iMem 中读入指令指示器 pc 指定的指令,并分类执行返回相应的结果状态。

tm 机指令执行状态： srOKAY 正常
srHALT 停止
srIMEM_ERR 指令存储错
srDMEM_ERR 数据存储错
srZERODIVIDE 除数为零错

算法框图： 见图 10.3。

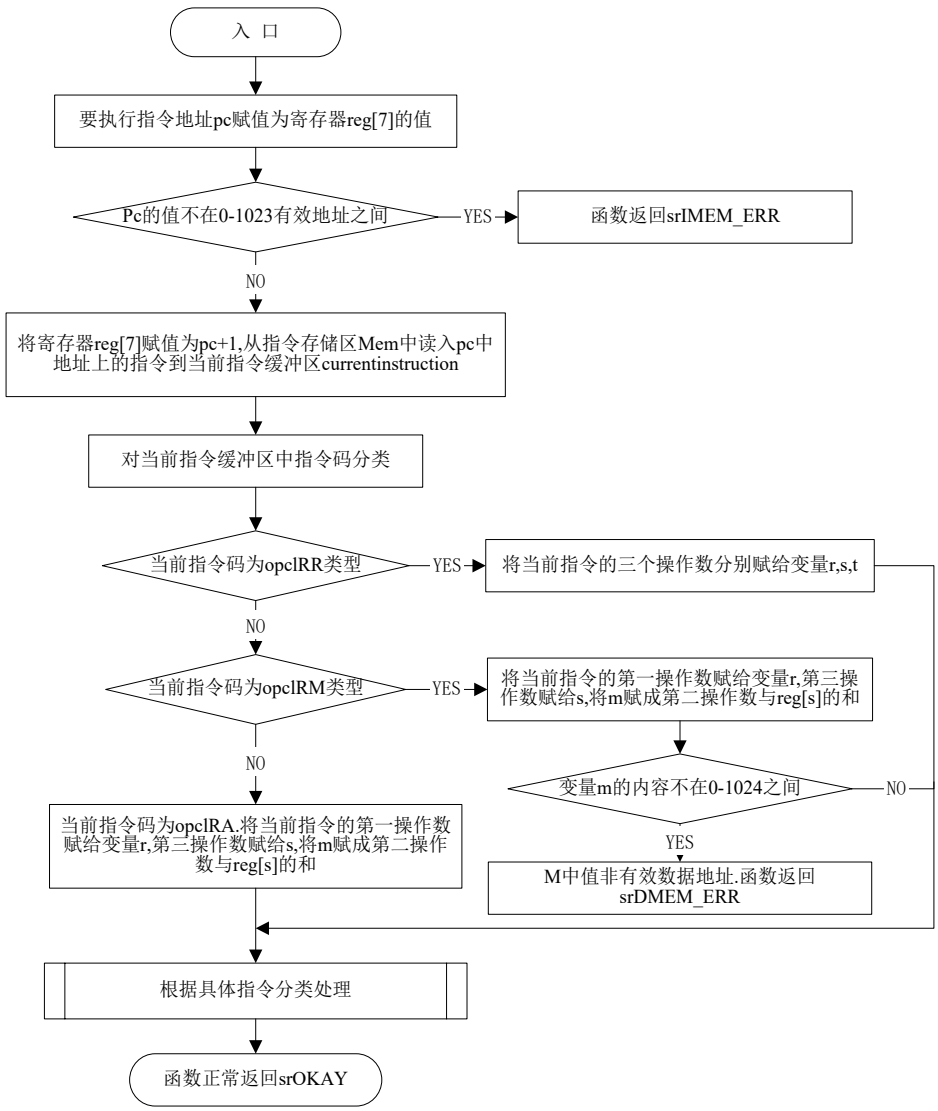


图10.3 单步执行函数stepTM的算法框图

分类处理部分中对寄存器-寄存器；寄存器-内存模式指令的处理算法框图:

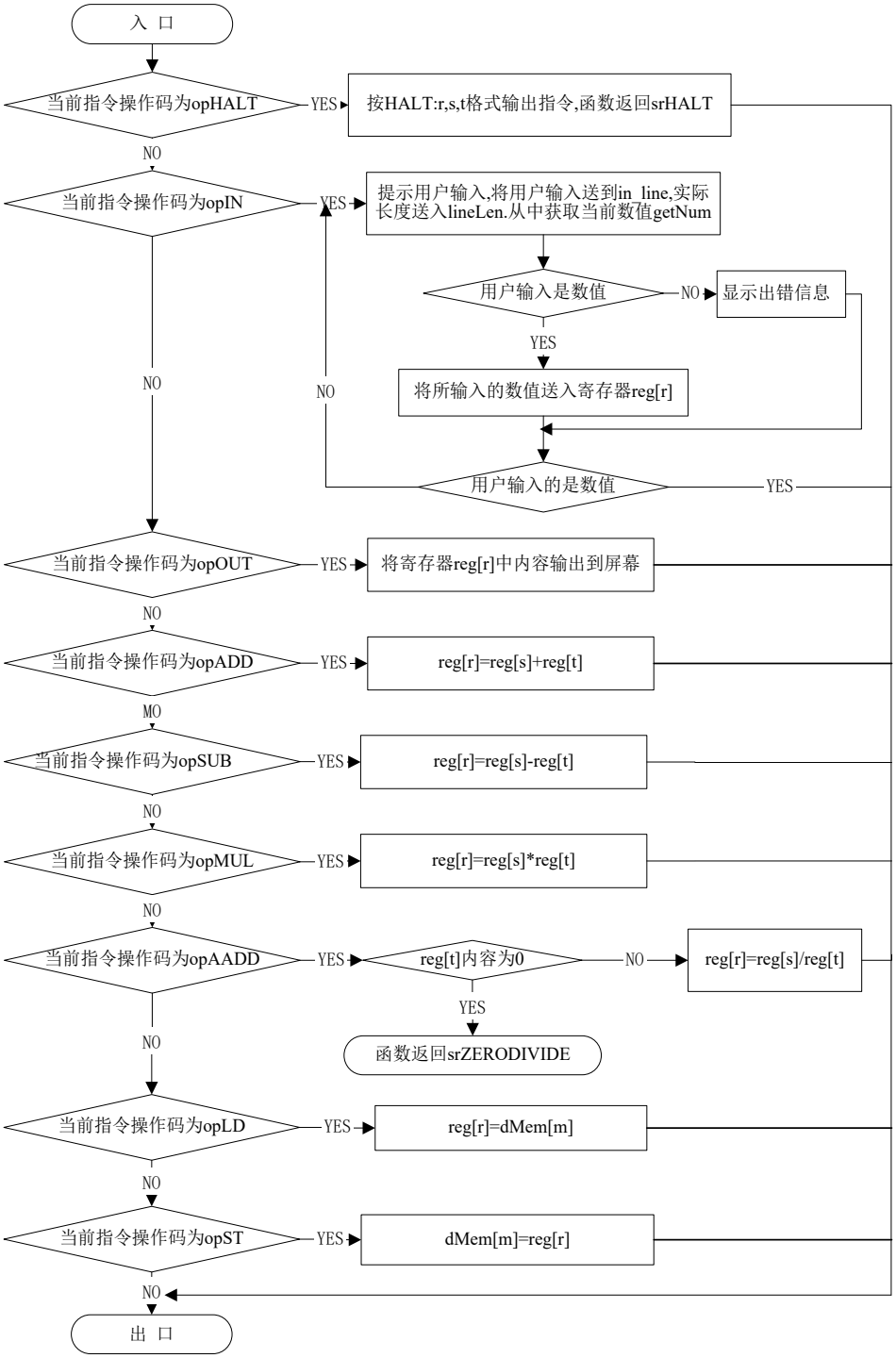


图10.4 寄存器-寄存器;寄存器-内存模式指令的处理算法框图

分类执行部分中对寄存器-立即数模式指令的处理算法框图:

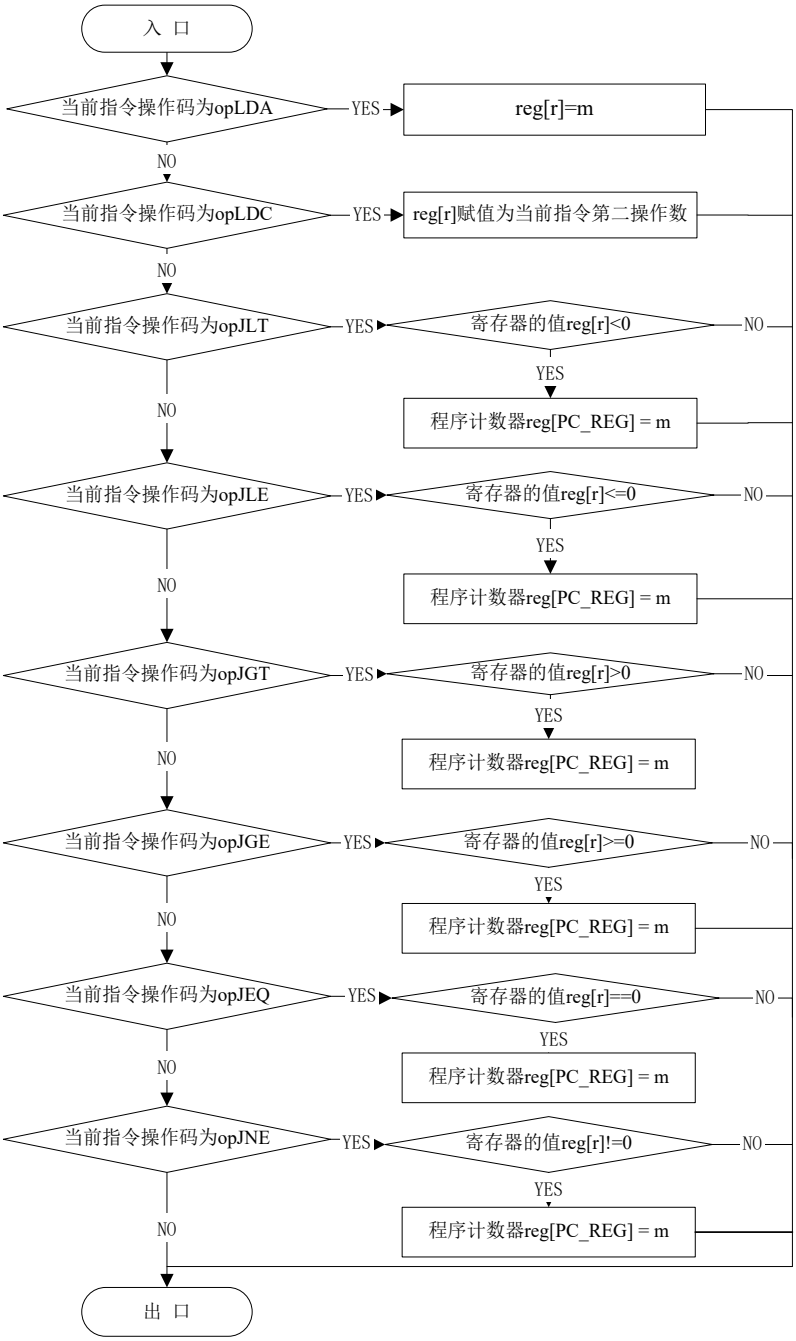


图10.5 寄存器-立即数模式指令的处理算法框图

9. TM 机命令处理函数 doCommand
函数声明：int doCommand (void)

函数功能：函数处理用户输入的 TM 命令,完成对 TM 机的相应处理。
算法框图：见图 10.6。

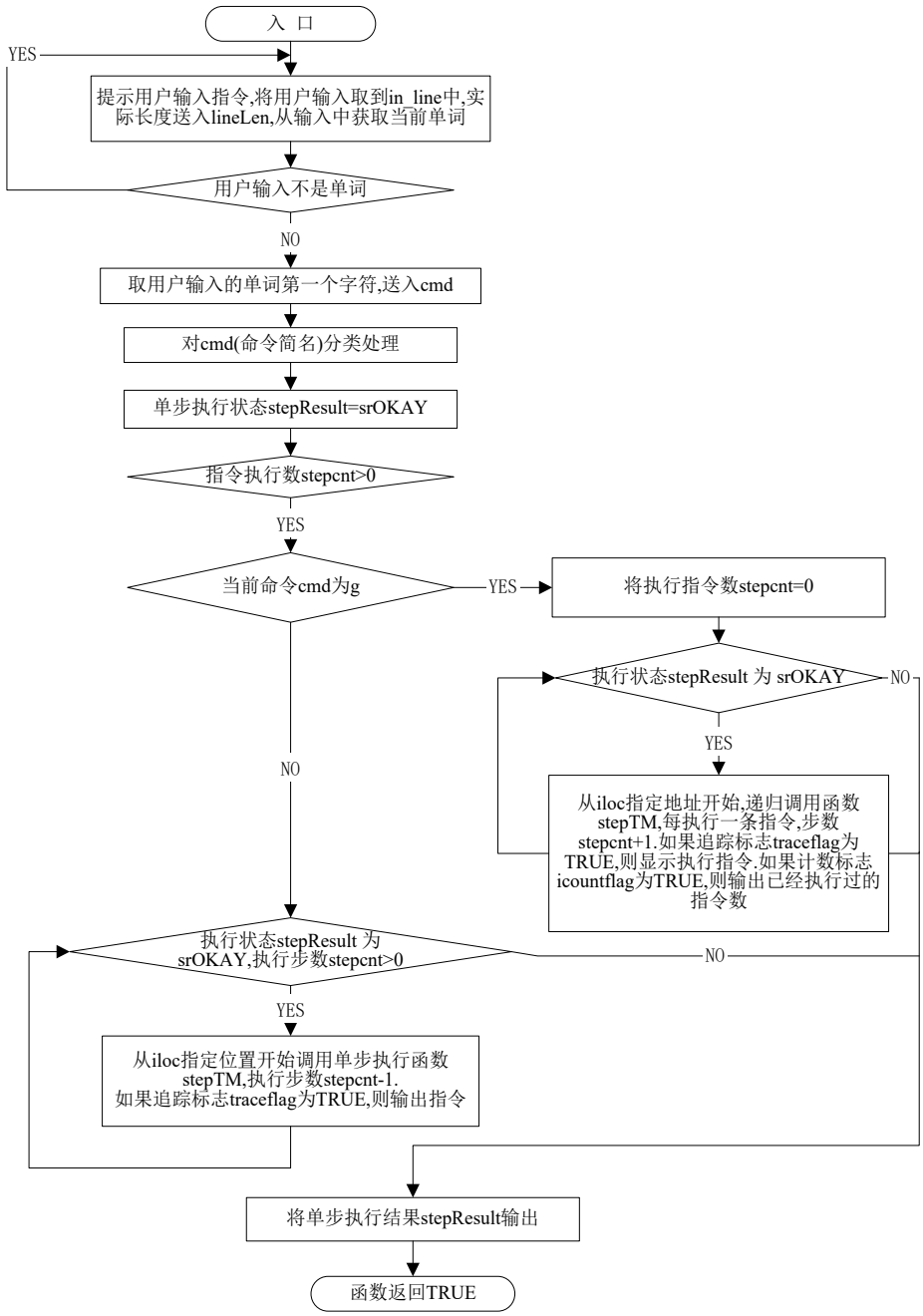
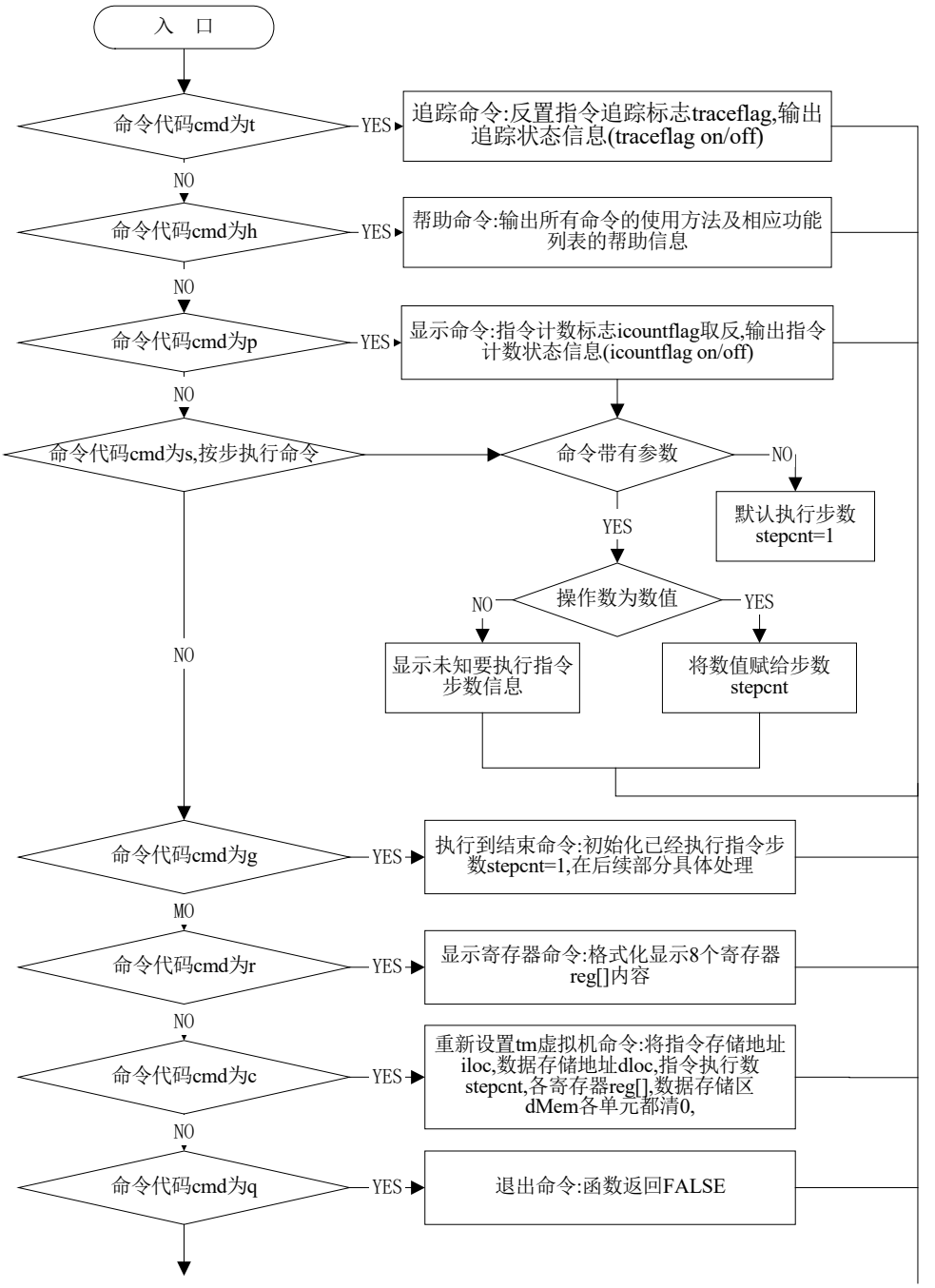


图10.6 命令处理函数doCommand

具体命令处理算法框图:



续下页

接上页

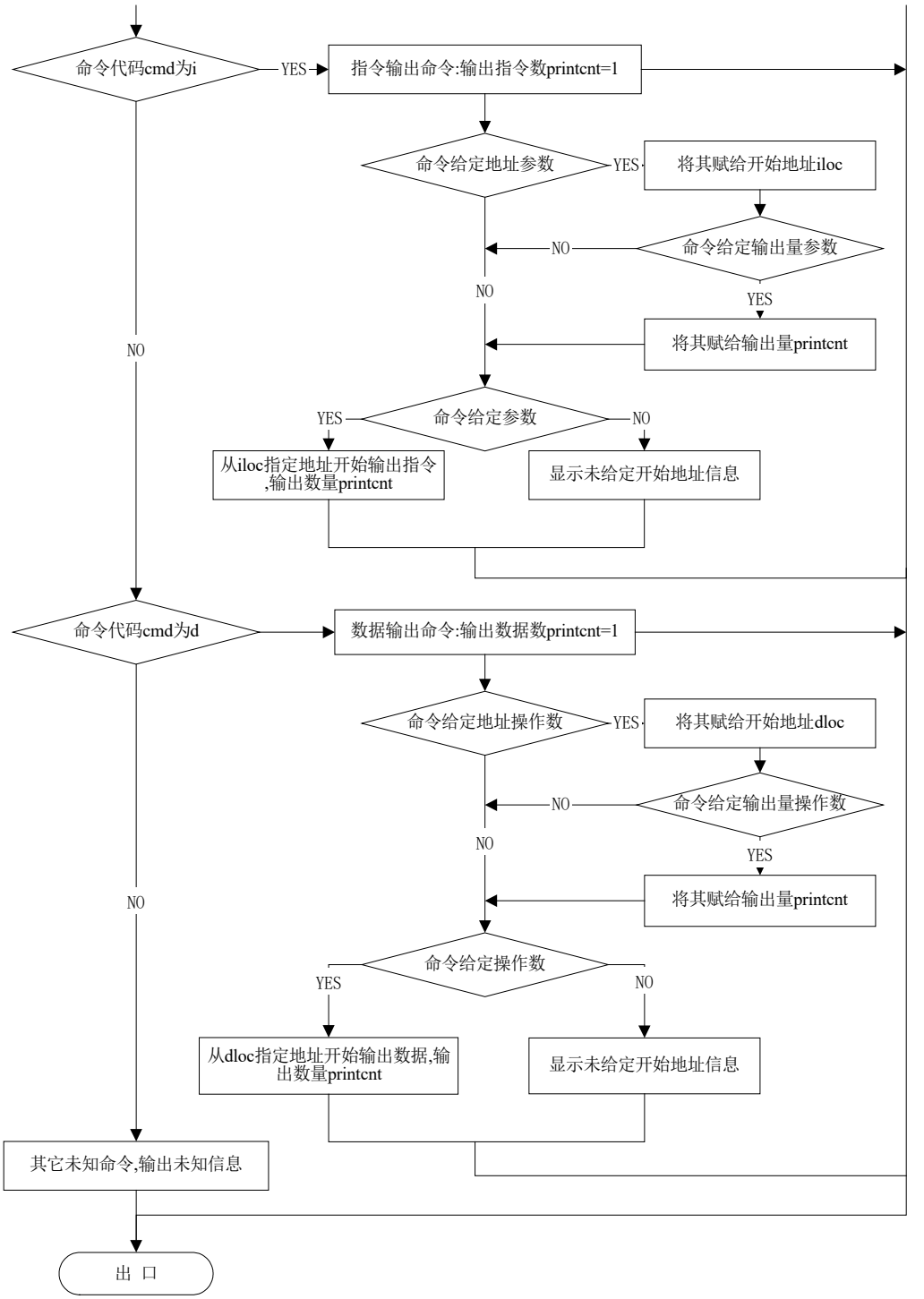


图10.7 具体命令处理算法框图

10. TM 虚拟机主程序函数 main

函数声明: main(int argc, char * argv[])

函数功能: 函数需要给定目标代码文件目录名参数,函数从参数指定文件识别指令并读入到 TM 机中指令存储区。显示 tm 可执行的命令。与用户交互,完成用户输入命令指定的操作。

算法框图: 见图 10.8。

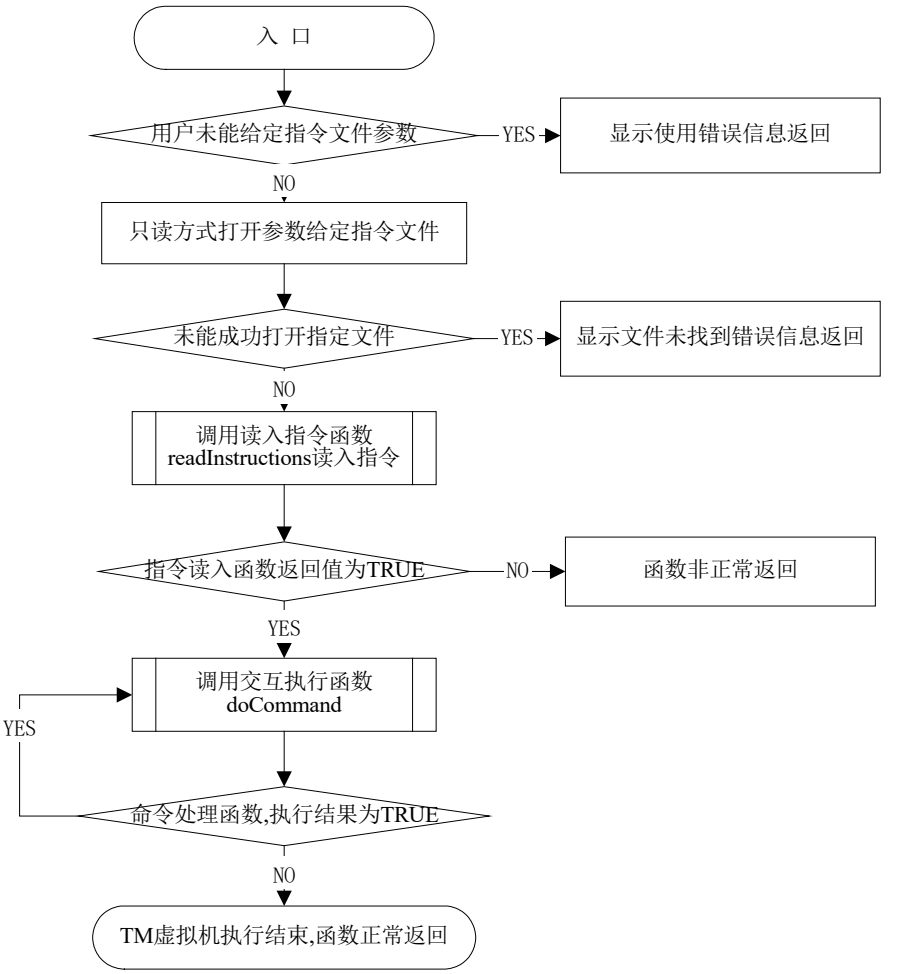


图10.8 主程序函数main的算法框图

第十一章 实践课题

本教材针对一个类 PASCAL 的教学程序设计语言 SNL，给出了其编译程序的每个功能部分的详细的实现过程，并且对源程序进行了详细的分析和说明，可以作为学生的实践教材，使学生在课上学习基本原理和实现技术的同时，通过本教材中编译实例的分析，以及亲身上机实践，加强对原理的理解，同时增强大型软件系统的开发能力。

为了简单清楚地说明编译程序的实现方法和具体实现过程，本教材采用的程序设计语言中没有包含一些比较复杂的数据结构和相应的操作，而对于编译程序的实现过程也采用了一些相对简单实用的方法。在学习和掌握本教材的基础上，作为扩充课题，还可以在以下一些方面进行实践：

11．1 语言的扩充和实现

1．运算的扩充： SNL 虽然给出了字符类型,但是并没有定义字符类型的操作，而表达式只能处理整数，可以对程序进行扩充，增加字符的操作。

2．类型的扩充： 记录类型、文件类型、指针类型等是程序设计语言中的常用类型,可以考虑扩充程序以支持以上类型。

3．语句的扩充： SNL 包含了进行结构化程序设计必须的所有语句。作为实践课题,可以考虑增加对 GOTO 语句的处理。

11．2 实现方法的扩充

1．符号表的组织： 为了简单起见，符号表的组织采用了简单的顺序查表法，可以考虑用散列法，二叉树法等其他的更有效率的符号表组织形式。

2．语法分析方法： 在语法分析阶段，我们给出了递归下降分析方法和 LL(1)分析方法，这两者都属于自顶向下分析方法，可以尝试自底向上的语法分析方法，如各种 LR 分析方法。

11．3 应用自动生成工具

1．词法分析： 应用 LEX 构造扩充后的语言的词法分析程序。

2．语法分析： 应用 ACCENT 构造扩充后的语言的语法分析程序，并增加语法错误处理。

11．4 实现语言

SNL 语言的编译器是基于 C 语言实现的，可以考虑按照对象式程序设计的思想，使用其它的开发工具如 C++，Java 等实现。

第十二章 SNLC (SNL Compiler) 软件使用指南

12．1 SNLC 概述

12．1．1 SNLC 的特色

SNLC 作为小的嵌套式语言（SNL）的集成开发环境，为使用者能够更加方便的进行 SNL 语言源程序的编写，更深入的学习编译器的各个组成部分的功能，以及更直观的了解 SNL 的内部实现方法提供了便捷的途径。

- SNLC 利用 SNL 多遍扫描的特性，提供了特有的分步式编译操作，用户可以在每步操作之后直观的看到该步分析之后的结果。
- SNLC 丰富了编译过程中各种编译方法的可选性，用户可在对源语言进行编译时按照需要选择不同的语法分析方法、不同的代码生成方法以及不同的优化方法，这样能够使使用者清晰的看到编译过程中每个步骤的运行结果，以便于进一步了解编译器的内部操作过程。
- SNLC 提供了直接查看源代码的功能。

12．1．2 SNLC 的运行环境

SNLC 可安装运行于 Windows 98 / Me / 2000 / XP。

12．1．3 SNLC 的安装和卸载

可通过如下步骤对 SNLC 进行安装：

1．双击 SNLC 的安装程序 snl.exe，进入图如 12.1 的界面：

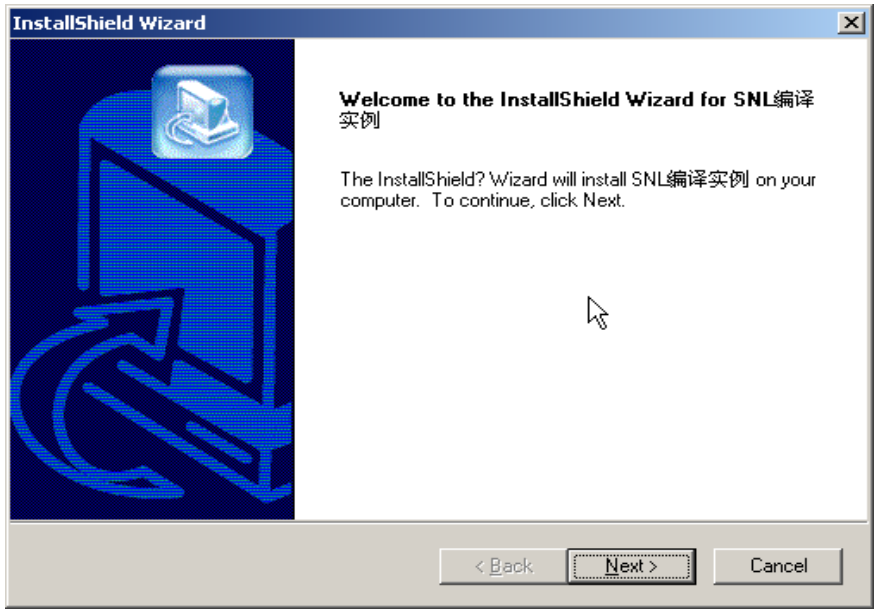


图 12.1 SNLC 开始安装界面

2．选择“Next”，进入如图 12.2 所示的界面：

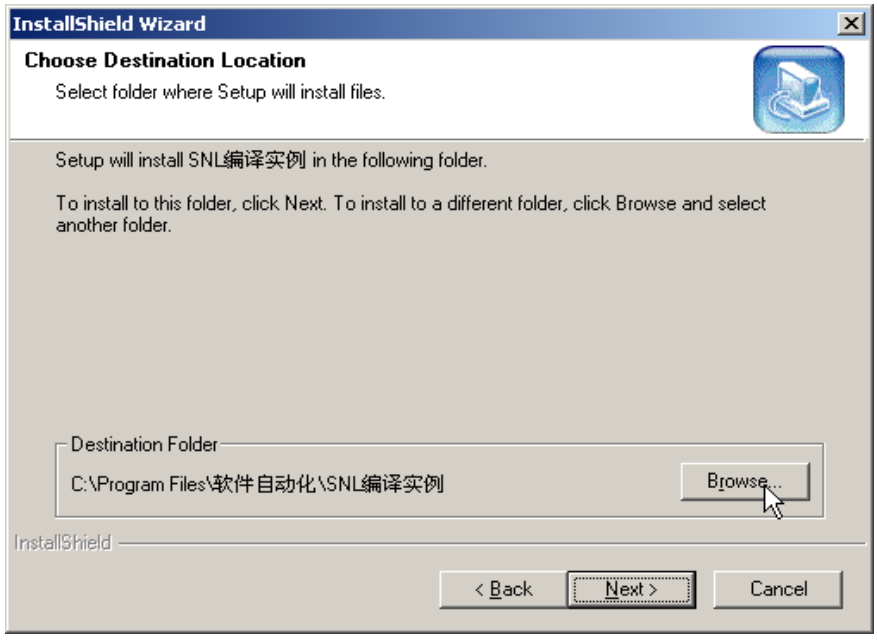


图 12.2 SNLC 安装路径选择界面

3．单击 Browse 选择程序将要安装到的文件夹，如图 12.3 所示；若采取默认值则选择 Next，进入下一步。

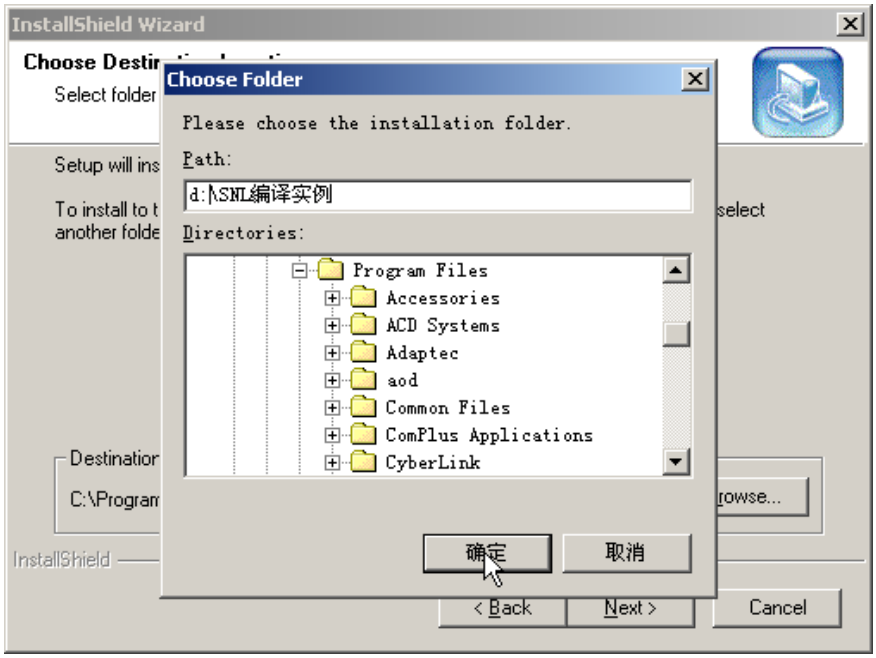


图 12.3 SNLC 安装文件夹的路径选择界面

4．继续选择 Next，直到安装程序完成，如图 12.4 所示：

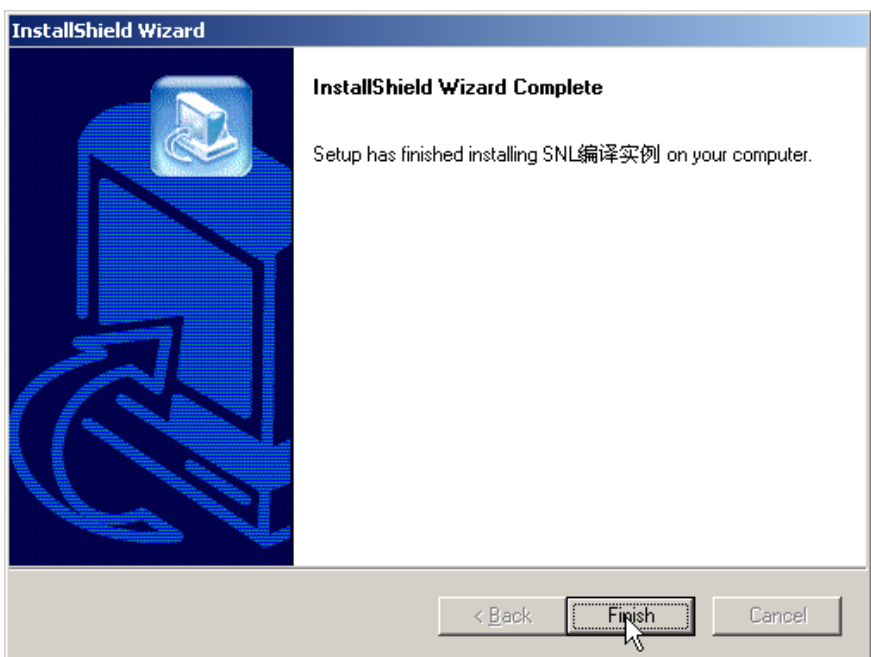


图 12.4 SNLC 安装完成界面

5. 单击 Finish，完成安装过程；程序被安装在用户指定的位置，并在桌面上自动创建了快捷方式以方便用户使用。

按照如下步骤进行卸载：

可以通过两种方式进行完全卸载 SNLC 程序，一种是通过控制面板里的添加/删除程序卸载，另一种是在机器已经装有 SNLC 程序的情况下，再次双击 SNLC 安装程序 snl.exe，则自动卸载原有的 SNLC 文件夹。

12．1．4 SNLC 的启动和退出

双击“SNL 编译实例”图标，进入 SNLC 的运行环境，如图 12.5 所示：

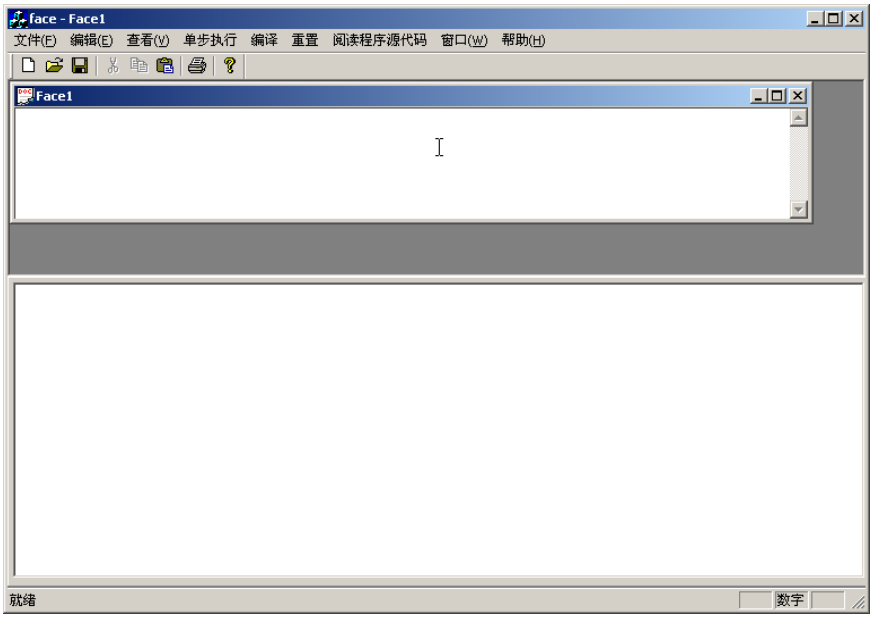


图 12.5 SNLC 的运行环境

用户在新建文档中写入源程序或者打开已有源程序进行编译即可。通过菜单“文件”→“退出”，或者通过单击窗口右上角的 close，可以退出 SNLC 界面。

12．2 SNLC 的使用

通过菜单“编译”→“编译”选项选择直接编译源程序而不需要单步执行，此时将直接生成目标代码，点击“执行”选项，则直接进行执行。

图 12.6 选择直接进行编译的 SNLC 运行环境

如果用户想单步执行，则可通过单步执行菜单，可以对源文件进行分阶段编译，并将各阶段的分析结果显示在输出栏中。

12．2．1 SNL 文件的操作

选择一个源文件：通过工具栏或文件菜单，新建一个 SNL 程序或打开一个已有的 SNL 程序，本例打开一个已有的 SNL 程序文件 factorial.txt，结果如图 12.7 所示：

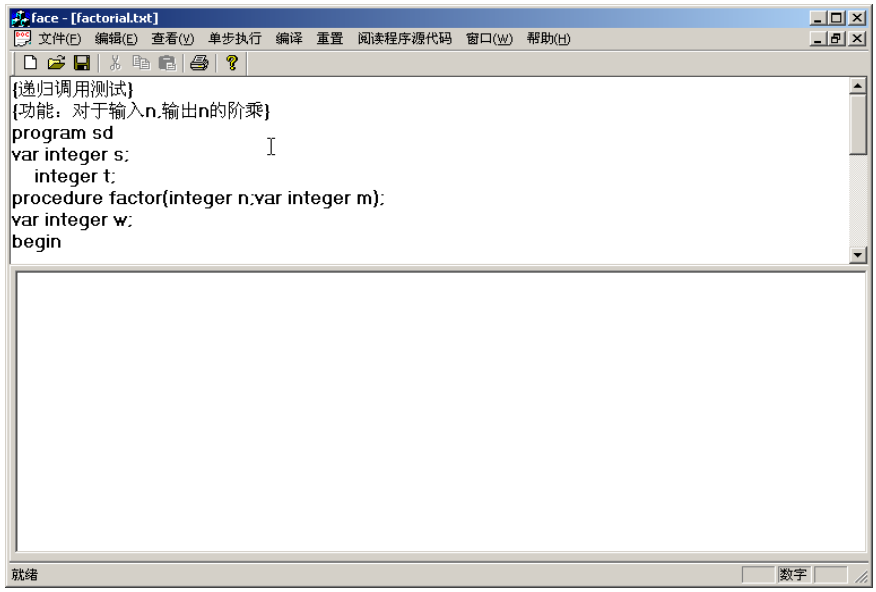


图 12.7 打开或新建 SNL 程序的操作示意图

12．2．2 SNL 程序的词法分析

对于一个已完成的源程序，可以通过菜单“单步执行”→“词法分析”选择对该源程序进行词法分析。

若程序没有词法错误，则输出框将给出提示“词法分析无错，Token 序列如下：”，否则，输出框将给出提示“词法分析有错，以 ERROR 打头的为错误的 Token”，两种情况都会在后面给出词法分析后得到的 Token 序列。

示例程序 factorial.txt 没有词法错误，结果如图 12.8 所示：

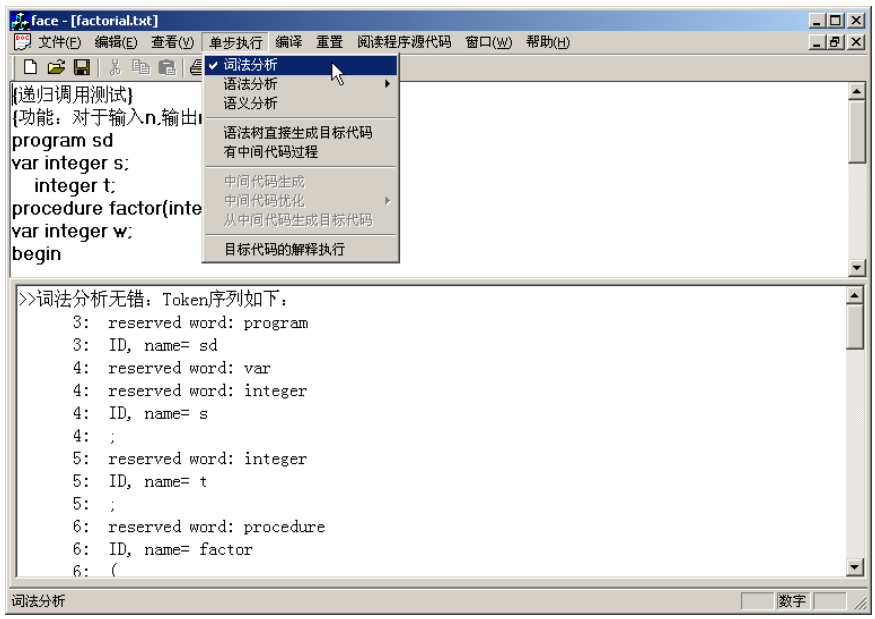


图 12.8　单步执行 SNL 源程序的词法分析

其中 token 序列的显示格式为：

行号 ： 词法信息 ： 语义信息

具体的 token 结构定义详见本书第四章词法分析。

12．2．3 SNL 程序的语法分析

对于一个已经进行完词法分析后的 token 序列，可以选择递归下降法或者 LL(1)方法进行语法分析。

若上一步检查出程序有错，那么用户选择进行语法分析时，软件会提示用户先纠正错误，再进行后面的操作；否则，给出语法分析的结果，若无语法错误，则输

出框将提示“无语法错误，构造的语法树如下：”，并显示语法树信息；否则，输出框给出语法错误的位置和错误单词。

本示例 factorial.txt 无语法错误，结果如图 12.9 所示：

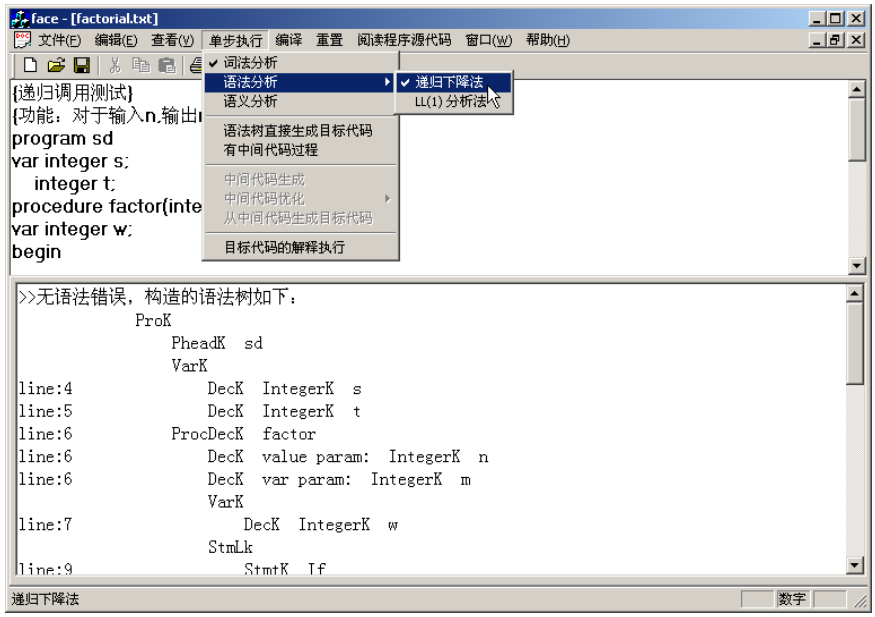


图 12.9 递归下降法执行 SNL 源程序的语法分析

其中语法树的显示结构为：

Token 的行号 ： 节点类型 节点的各个域

如果是儿子节点，则向后缩进 4 个字符的空位，如果是兄弟节点，则在同一条直线上。

具体的语法格式详见本书第五章语法分析。

12．2．4 SNL 程序的语义分析

对于一个通过语法分析生成的语法树可以进行语义分析来检查语义错误。

若无语义错误，则输出框将提示“无语义错误，生成的符号表如下：”，并显示符号表信息；否则，输出栏给出语义错误的位置和错误类型。

本示例 factorial.txt 无语义错误，结果如图 12.10 所示：

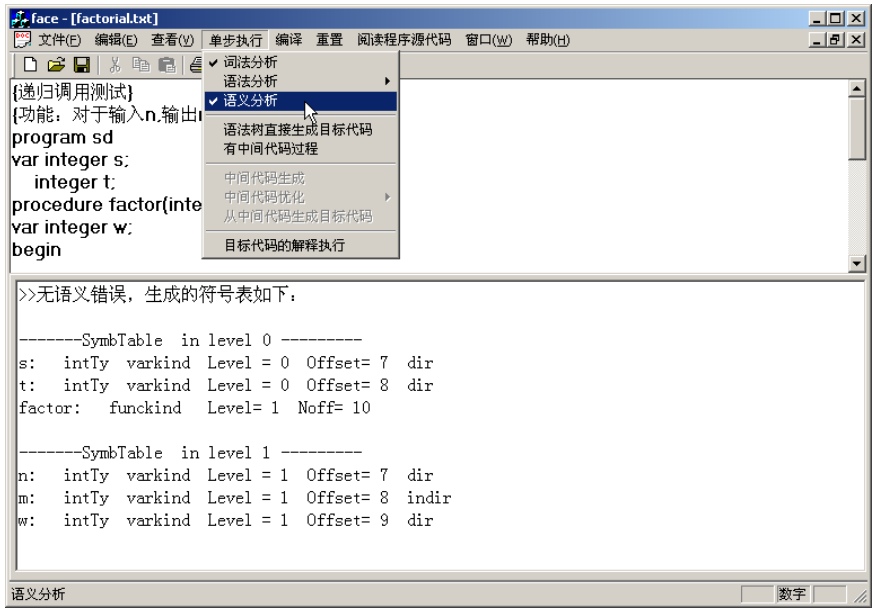


图 12.10 单步执行 SNL 源程序的语义分析

其中符号表的显示格式如下：

----------层 号----------
标识符名： 类型信息 层号 偏移 直/间接变量

具体的语义信息存储格式详见第六章语义分析部分。

12．2．5 SNL 程序的中间代码生成

对于语义分析后的无语义错误的语法树，本软件提供两种生成目标代码的途径，一种是先对语法树生成中间代码，再生成目标代码；另一种是从语法树直接生成目标代码。

打开菜单项，选择“有中间代码过程”，如图 12.11 所示：

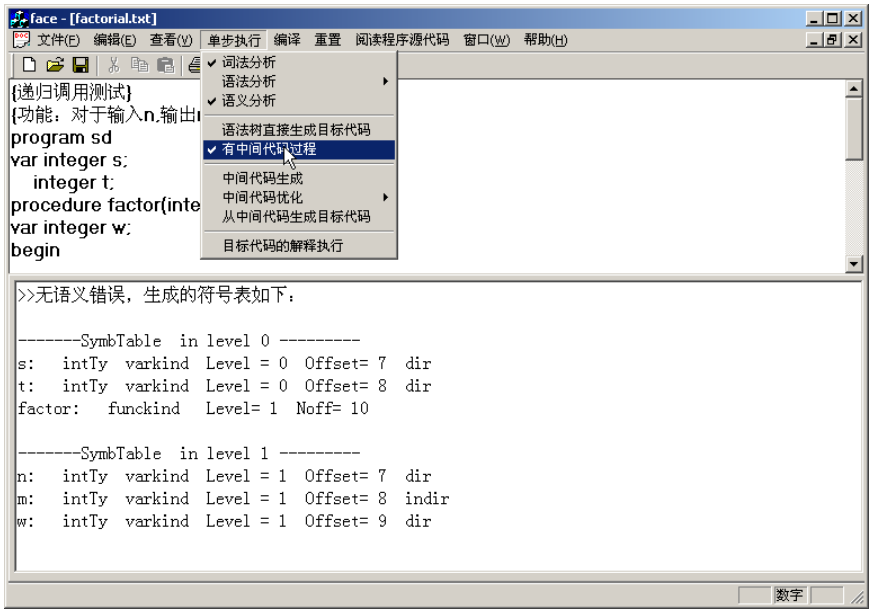


图 12.11 选择 SNL 有中间代码生成过程

然后选择“中间代码生成”以生成中间代码：

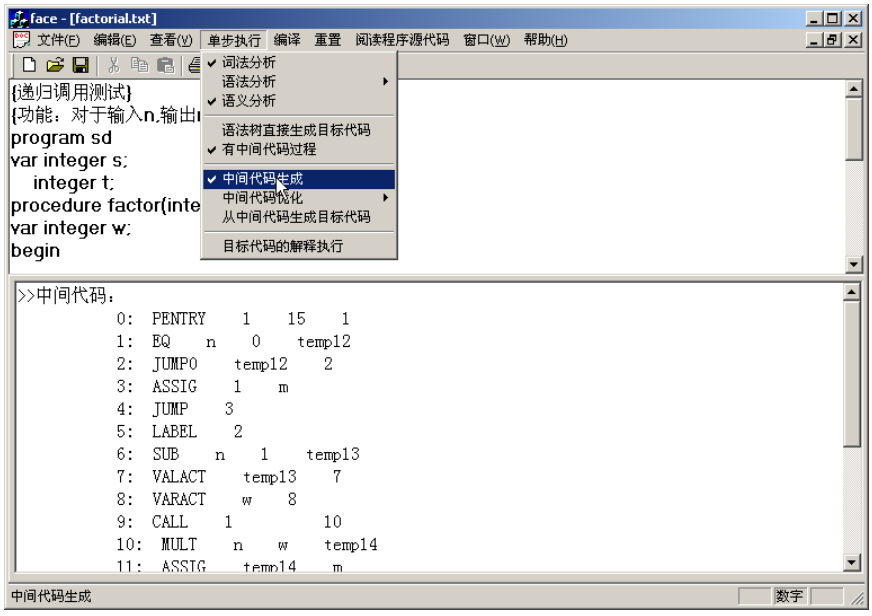


图 12.12 单步执行 SNL 源程序的中间代码生成

执行后生成的中间代码将在下方的文本框中显示出来。其具体显示格式如下：

行号 ： 四元式

具体的四元式的结构定义见第七章中间代码生成。

12．2．6 SNL 程序的优化

本软件提供三种中间代码的优化方案：常量表达式优化、公共表达式优化、循环不变式优化。用户可以任选其中的某几个方案进行中间代码优化，也可以不选择任何项，即不对中间代码进行任何优化，而直接生成目标代码。

- 选择常量表达式优化，结果如图 12.13 所示：

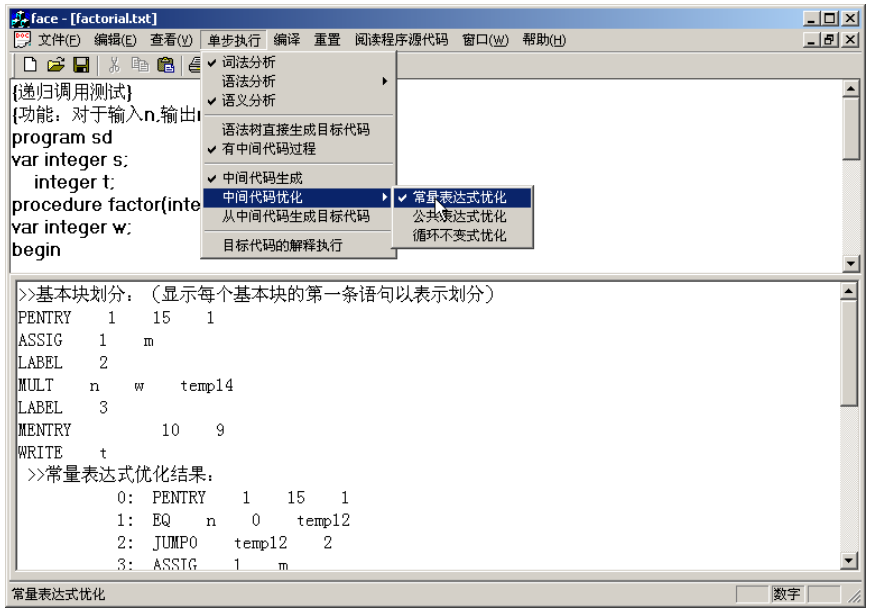


图 12.13 单步执行 SNL 中间代码的常表达式优化

- 进行公共表达式优化，结果如图 12.14 所示：

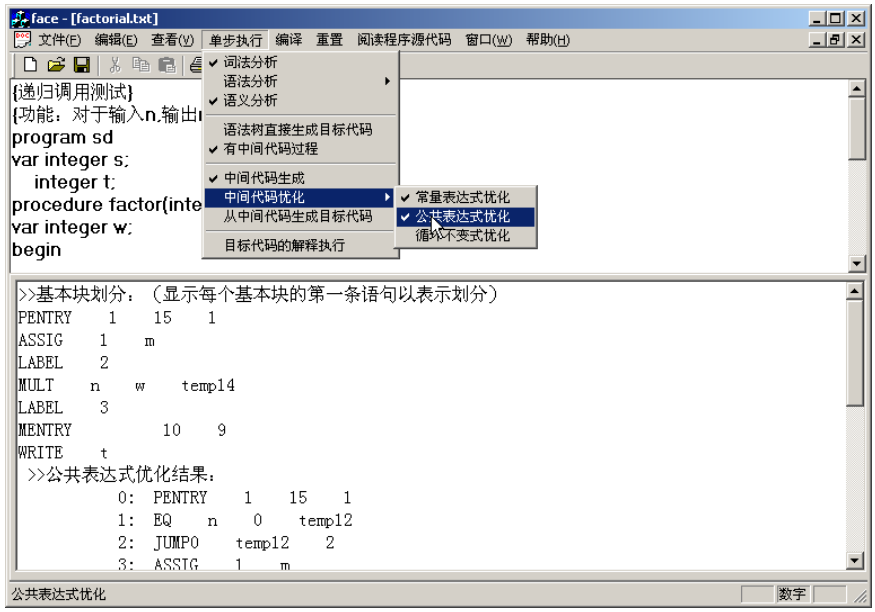


图 12.14 单步执行 SNL 中间代码的公共表达式优化

■ 进行循环不变式优化，结果如图 12.15 所示：

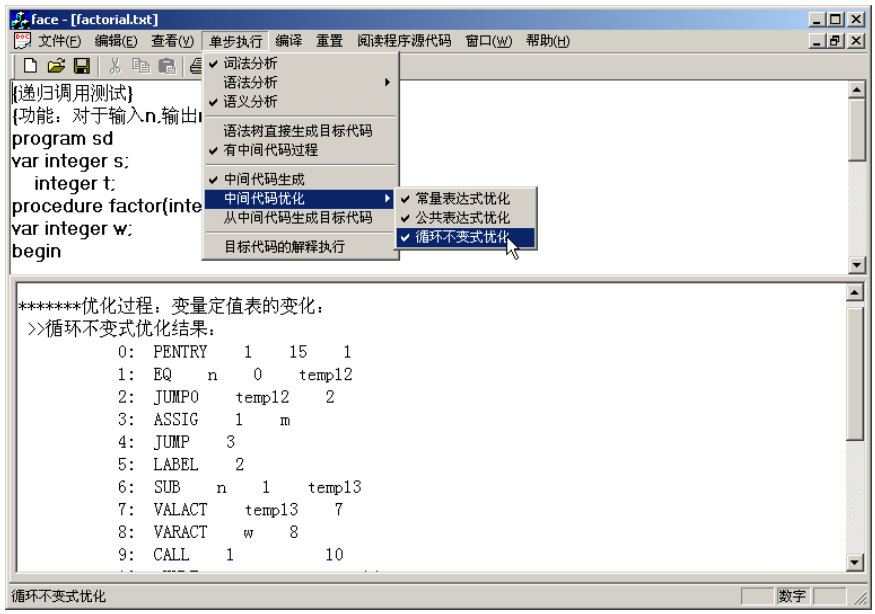


图 12.15 单步执行 SNL 中间代码的循环不变式优化

12．2．7 SNL 程序的目标代码生成

如果选择了从语法树直接生成目标代码，则如图 12.16 所示：

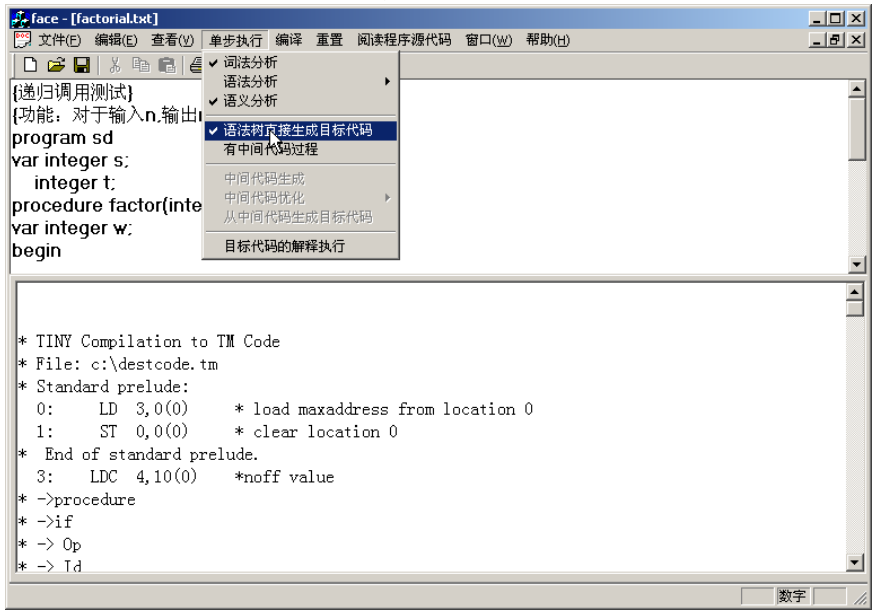


图 12.16 选择从语法树直接生成目标代码

在界面的下方文本框中将显示出适用于 TM 目标机的目标代码，其具体格式见第十章虚拟机 TM 的简介。

如果选择了生成中间代码，则可以在生成中间代码之后的任意一个步骤之后选择“从中间代码生成目标代码”，如图 12.17 所示：

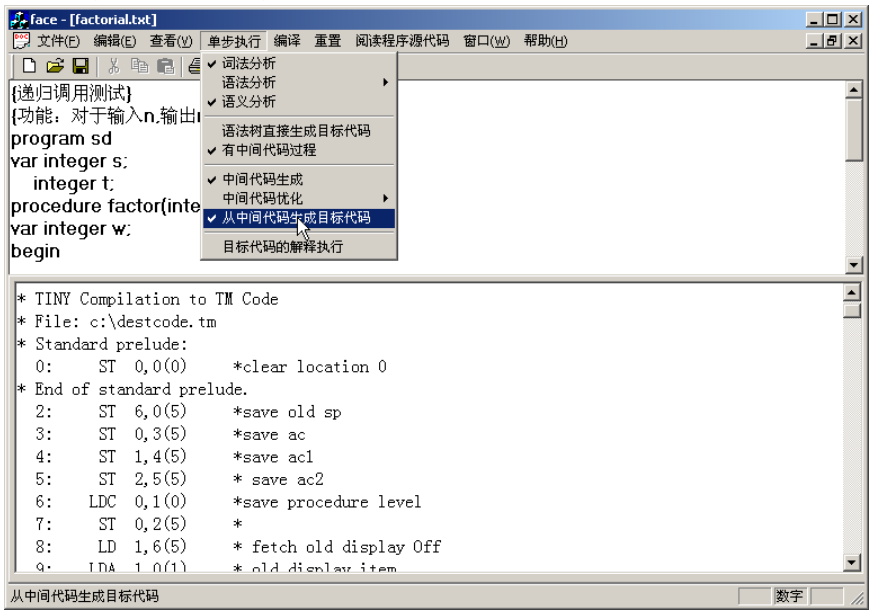


图 12.17 选择从中间代码生成目标代码

12．2．8 SNL 程序的虚拟执行

生成目标代码之后，可以通过从菜单项中选择“目标代码的解释执行”，进入执行虚拟机状态：

- 开始执行虚拟机。

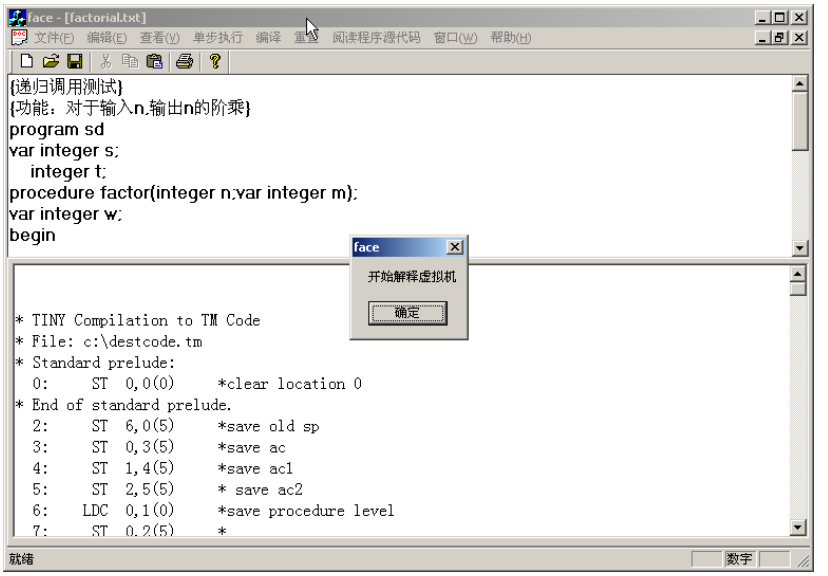


图 12.18 开始解释虚拟机

- 根据屏幕提示的信息，输入想要执行的指令。注意必须命令输入是区分大小写的，请输入小写指令标记，进行相应的动作。
  例如：若要执行目标代码，输入命令 g ，点击 “确定”。

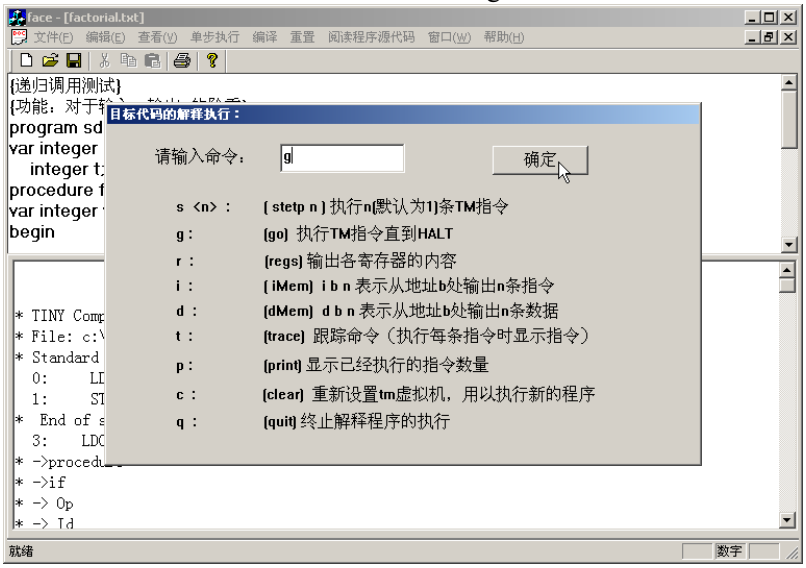


图 12.19 输入命令解释执行目标代码

■ 若 SNL 源程序有输入，则进入输入提示对话框。例如：此示例程序 factorial.txt 的作用是对于输入整数 n，计算 n 的阶乘，在输入对话框中输入数字 4，点击“确定”。

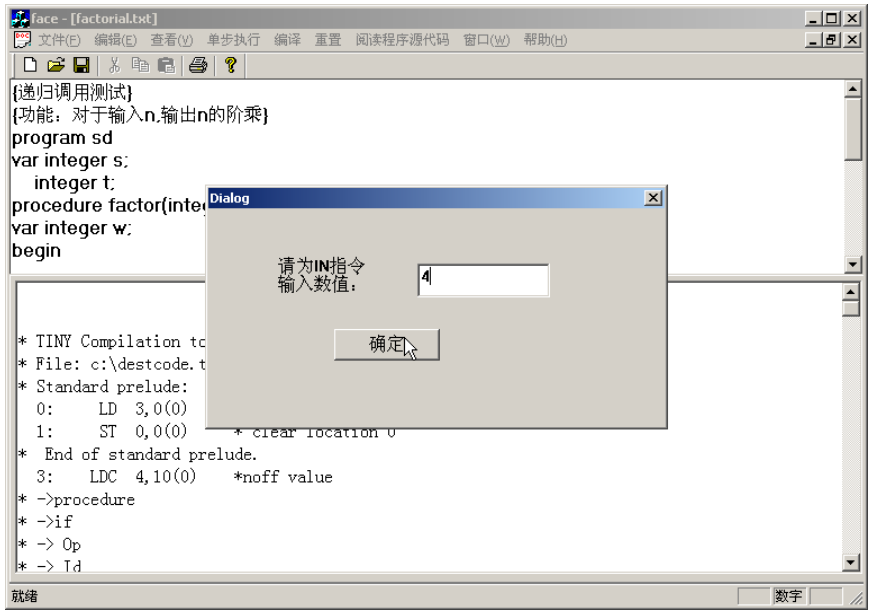


图 12.20 按照源程序要求输入数值

■ 显示输出结果，如图 12.21 所示，编译器计算 4 的阶乘,并将结果显示在输出框中：

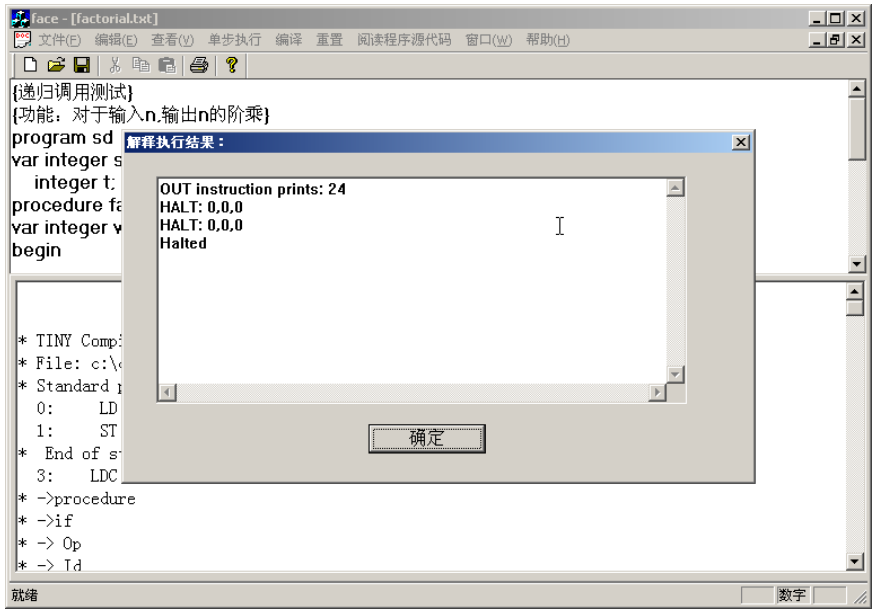


图 12.21 解释执行程序 factorial 的结果

■ 单击“确定”，回到执行指令选择对话框。若想执行其他的命令，可以根据提

示信息输入相应的指令；选择 q 终止程序的执行，并返回 SNL 编译器主菜单，如图 12.22 所示：

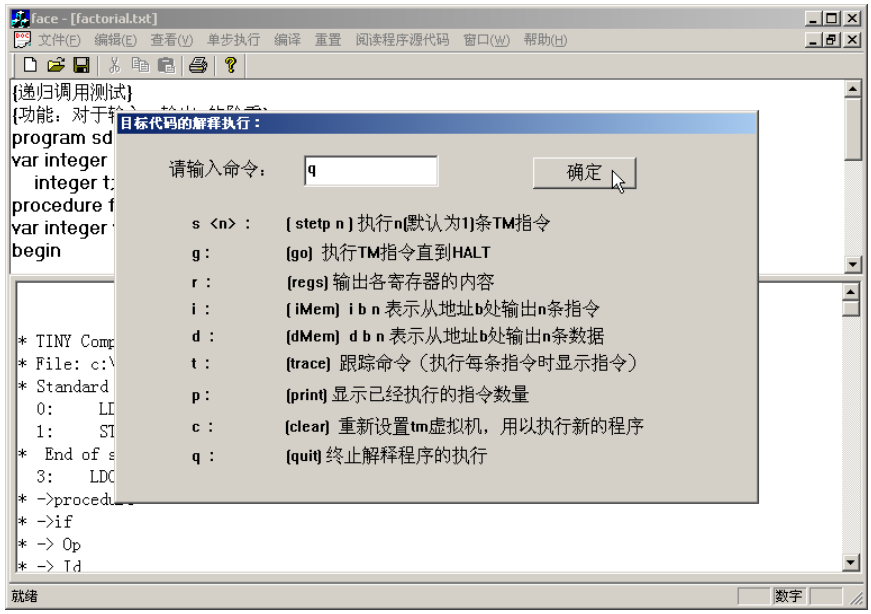


图 12.22 中止目标代码的解释执行

- 虚拟机执行完毕，回到编译主界面，如图 12.23 所示：

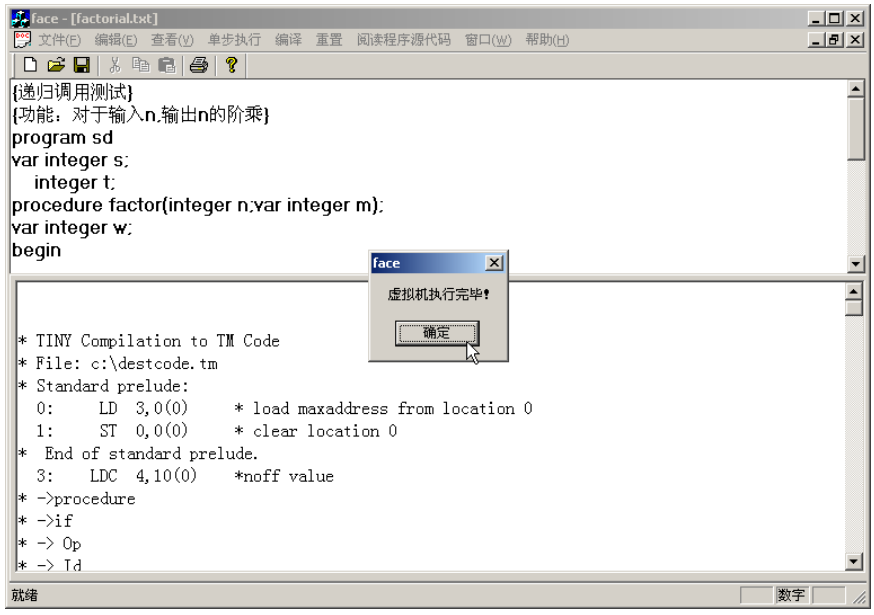


图 12.23 虚拟机解释执行完毕

12．3 有关问题的说明

12．3．1 SNLC 的维护和出错处理

如果在编译的过程中出现错误（包括各种词法、语法、语义错误），都会在界面下方的输出框中显示出错误提示，用户可根据错误提示，到源程序中进行适当的修改。此时用户可以使用系统的“重置”功能进行重置，使整个编译系统回到最初始的状态。

菜单中的“重置”选项为用户提供了这种恢复初始状态的方式，当用户误选了某一步骤或者中途退出编译过程时，可以通过选择“恢复初始值”而简单的完成。

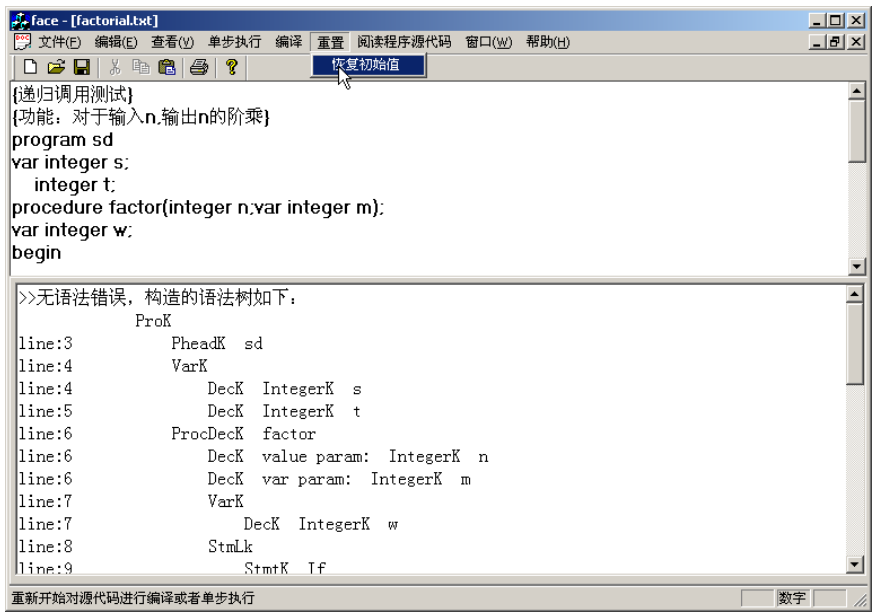


图 12.24 恢复初始状态

12．3．2 SNLC 的帮助功能

- 阅读源程序

单击“阅读程序源代码”菜单项，选择想要学习的源代码，例如选择查看“词法分析程序”，则词法分析程序的源代码将显示在文件框中。

如图 12.25 所示：

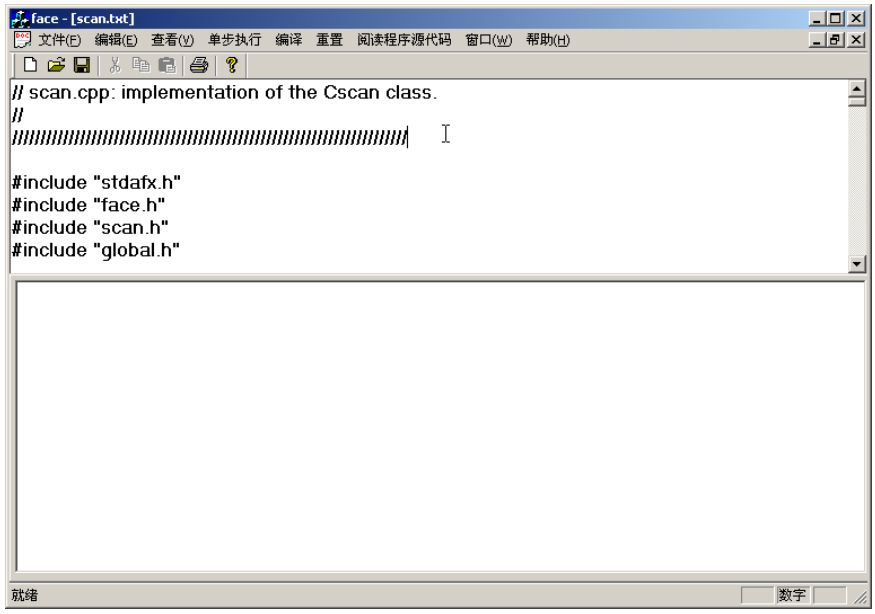


图 12.25 阅读 SNL 源码

■ LL(1)文法

点击“帮助”→“LL(1)文法”，界面上方将显示出 SNL 语言的上下文无关文法。如图 12.26 所示：

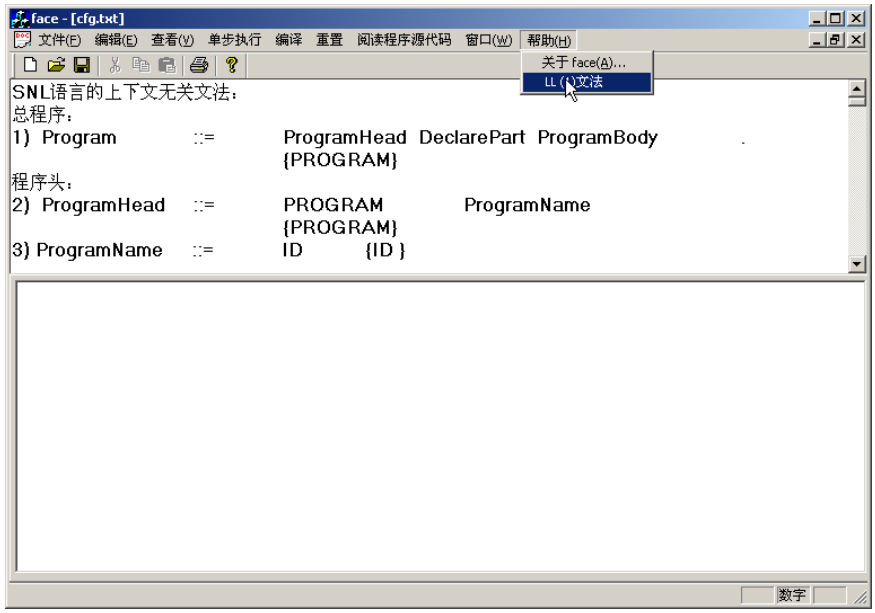


图 12.26 SNL 语言的上下文无关文法

■ 关于 SNL 编译器
通过帮助菜单，可以了解 SNL 编译器的相关信息，谢谢使用。

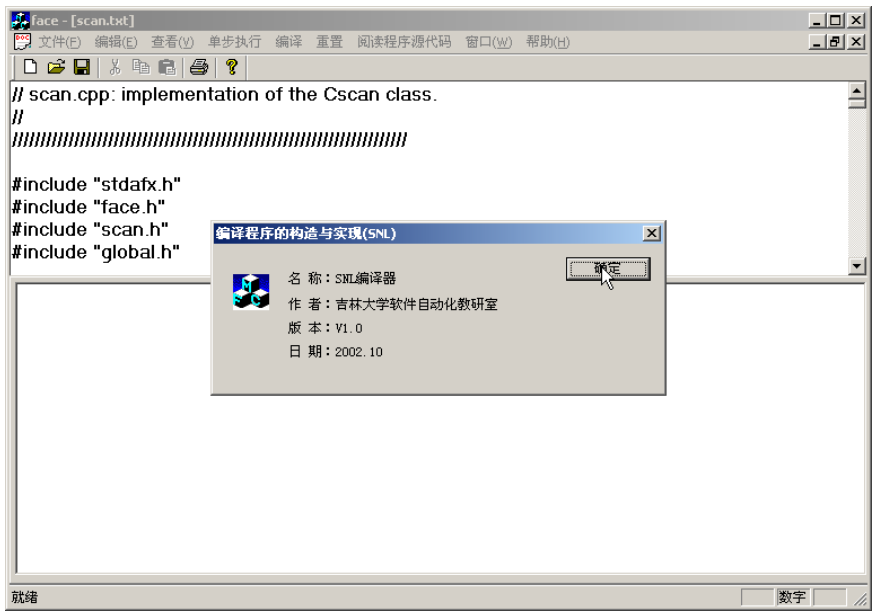


图 12.27 关于 SNL 编译器的相关信息

参 考 文 献

[1] 金成植．编译程序构造原理与实现技术．北京： 高等教育出版社，2000
[2] Alfred V.Aho,Ravi Sethi,Jeffrey D.Ullman．Compilers:Principles,Techniques and Tools．北京：人民邮电出版社，2002
[3] Kenneth C.Louden．Compiler Construction Principles and Practice．北京：机械工业出版社，2002
[4] 陈火旺．程序设计语言编译原理．北京：国防工业出版社，1984
[5] 陈意云．编译原理和技术．合肥：中国科学技术大学出版社，1989
[6] Ravi Sethi ．Programming Languages:Concepts and Structures(2E)．北京：机械工业出版社，2002
[7] Appel,Andrew W．Modern Compiler Implementation in C/ML/Java．Cambridge University Publishers，1997
[8] Muchnick,Steven ．Advanced Compiler Design and Implementation．Morgan Kaufmann Publishers，1997