

**编译原理课程设计指导书**

|  |  |
| --- | --- |
| **题目：** | **PL/0编译器的设计与实现** |

|  |  |  |
| --- | --- | --- |
| **学院** | ： | 计算机与信息安全学院 |
| **指导教师** | ： | 王慧娇 |

2023年 5 月 15 日

**目录**

[1 Lex和yacc的使用 3](#_Toc104060924)

[1.1 任务描述 3](#_Toc104060925)

[1.2相关知识 3](#_Toc104060926)

[1.3实验步骤 3](#_Toc104060927)

[1.4编程要求 6](#_Toc104060928)

[1.5测试说明 6](#_Toc104060929)

[1.6参考文献 8](#_Toc104060930)

[2词法分析 9](#_Toc104060931)

[2.1任务描述 9](#_Toc104060932)

[2.2相关知识 9](#_Toc104060933)

[2.2.1 PL/0语言编译系统 9](#_Toc104060934)

[2.2.2 扩展巴克斯范式 9](#_Toc104060935)

[2.2.3 词法分析 10](#_Toc104060936)

[2.2.4 词法分析中的算法 11](#_Toc104060937)

[2.2.5设计提示 12](#_Toc104060938)

[2.3实验步骤 14](#_Toc104060939)

[2.4编程要求 14](#_Toc104060940)

[2.5测试说明 14](#_Toc104060941)

[2.6参考文献 20](#_Toc104060942)

[3语法分析 21](#_Toc104060943)

[3.1任务描述 21](#_Toc104060944)

[3.2相关知识 21](#_Toc104060945)

[3.2.1 语法分析 21](#_Toc104060946)

[3.2.2 预测分析法 22](#_Toc104060947)

[3.2.3 递归下降分析法 22](#_Toc104060948)

[3.2.4 LR分析法 22](#_Toc104060949)

[3.2.5 算符优先分析法 23](#_Toc104060950)

[3.2.6 自动工具分析法 23](#_Toc104060951)

[3.2.7 LL（1）文法相关算法 23](#_Toc104060952)

[3.2.8 SLR相关算法描述 24](#_Toc104060953)

[3.3实验步骤 25](#_Toc104060954)

[3.4编程要求 25](#_Toc104060955)

[3.5测试说明 25](#_Toc104060956)

[3.6参考文献 29](#_Toc104060957)

[4 语义分析 30](#_Toc104060958)

[4.1任务描述 30](#_Toc104060959)

[4.2相关知识 30](#_Toc104060960)

[4.2.1 编译过程 30](#_Toc104060961)

[4.2.2 语义分析 31](#_Toc104060962)

[4.2.3 静态语义分析 31](#_Toc104060963)

[4.2.4 中间代码生成 31](#_Toc104060964)

[4.2.5 代码优化 33](#_Toc104060965)

[4.2.6 目标代码生成 33](#_Toc104060966)

[4.3实验步骤 33](#_Toc104060967)

[4.4编程要求 33](#_Toc104060968)

[4.5测试说明 34](#_Toc104060969)

[4.6参考文献 39](#_Toc104060970)

# 1 Lex和yacc的使用

## 1.1 任务描述

学习自动构造词法分析工具lex（Lexical Analyzer）和自动构造语法分析/语义计算工具yacc（Yet Another Compiler Compiler）的安装、配置及使用方法，加深对词法分析、语法分析及语义计算工作过程的理解。

## 1.2相关知识

**lex是词法分析程序的自动构造工具。**lex的运行方式如图1.1所示，lex读入用户编写的一个lex描述文件，生成一个名为lex.yy.c的C源程序文件。其中lex.yy.c中包含一个核心函数yylex()，它是一个扫描子程序，主要任务是读入源程序的字符流，识别并返回下一个单词符号[1]。

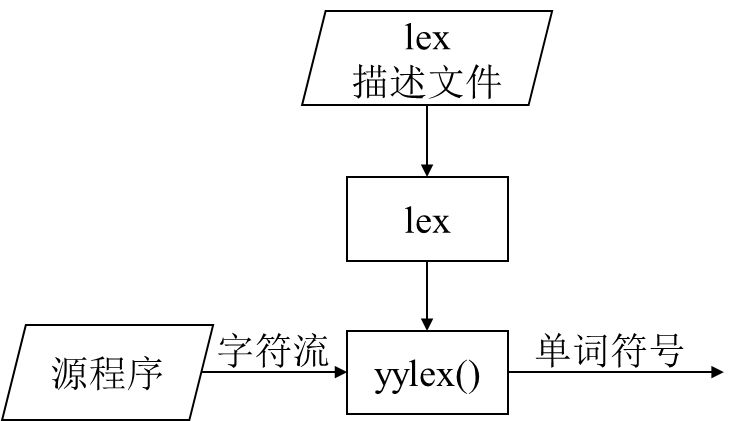


图 1.1 lex简介

**yacc是语法分析/语义计算程序的自动构造工具。**yacc的运行方式如图1.2所示，yacc的核心功能是读入用户编写的一个yacc描述文件，生成一个名为y.tab.c的C源程序文件。其中y.tab.c包含一个函数yyparse()，它描述了一个基于LALR(1)分析表和LALR(1)分析过程，以及基于这一分析过程的语法制导的语义计算。每当yyparse()需要读入终结符时，就调用称为yylex()的词法扫描子程序返回下一个单词符号（包含单词种别以及单词自身的值等信息）。yylex()可以由用户自己编写，也可以通过lex自动生成[1]。

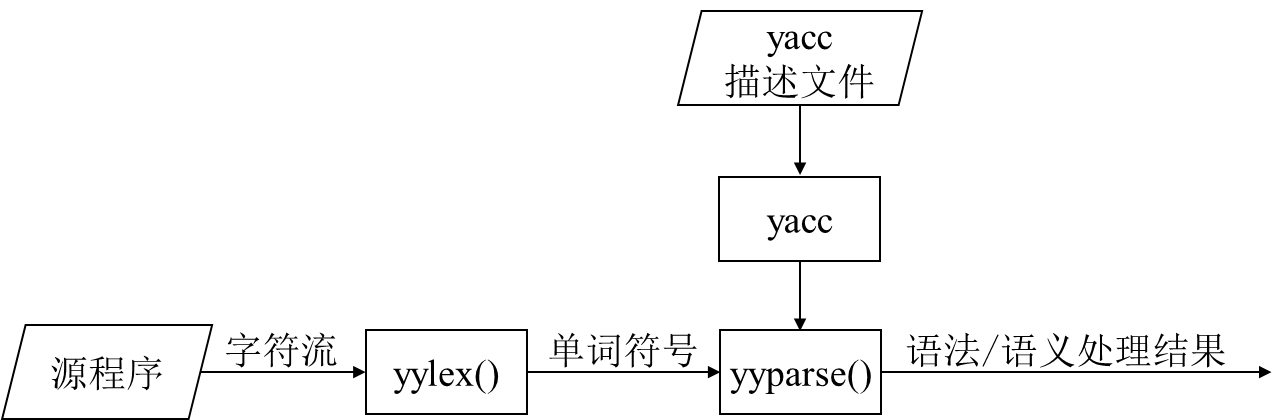


图 1.2 yacc简介

## 1.3实验步骤

1、环境配置

（1）下载flex和bison

本次实验在windows系统进行，lex和yacc对应的版本分别为flex[2]和bison，下载地址为：<https://sourceforge.net/projects/winflexbison/files/latest/download>。

（2）配置系统环境变量

将flex和bison所在路径追加到系统的Path环境变量里。

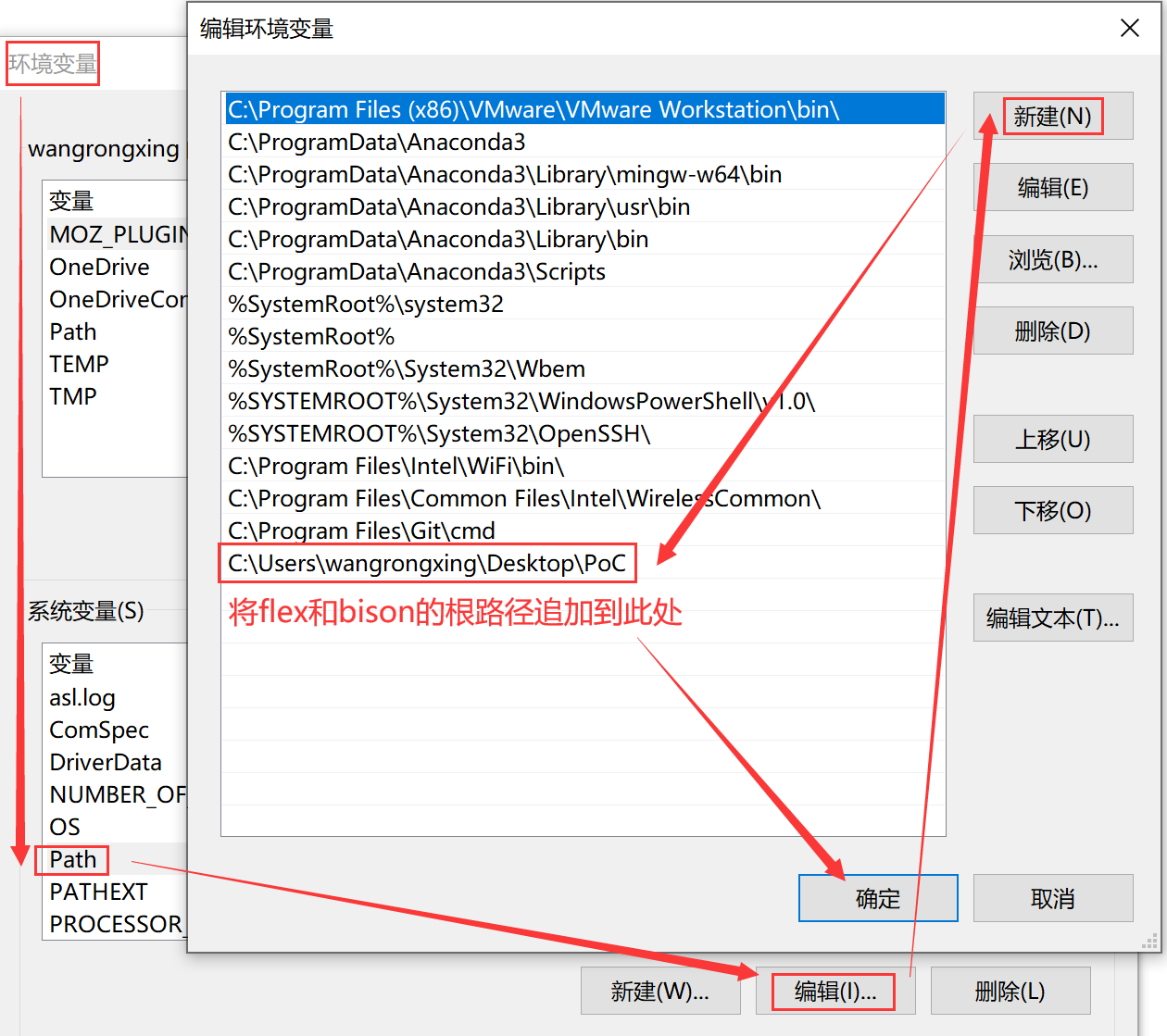


图 1.3 配置系统环境变量

（3）验证

创建test.l和test.y两个测试文件，使用flex执行test.l文件，产生lex.yy.c文件；使用bison执行test.y文件，产生test.tab.c文件。若上述步骤能正常完成，则环境配置成功。

以下为验证步骤里test.l和test.y测试文件的源程序。

**文件：test.l**

%{

    int yywrap(void);

%}

%%

%%

int yywrap(void)

{

    return 1;

}

**文件：test.y**

%{

    void yyerror(const char \*s);

%}

%%

program:

;

%%

void yyerror(const char \*s)

{

}

int main()

{

    yyparse();

    return 0;

}

**lex描述文件的格式**包含辅助定义部分、规则部分和用户子程序部分，三个部分通过%%隔开。三个部分可以按需求选择填写，在没有用户子程序部分时，第二个%%也可省略。

lex描述文件的格式：

辅助定义部分

%%

规则部分

%%

用户子程序部分

**yacc描述文件的格式**包含声明部分、辅助定义部分和用户函数部分，其中声明部分通过%{}%隔开，其它部分通过%%隔开。若声明部分为空，则%{和%}的两行可去掉；若用户函数部分为空，则第二个%%的行也可去掉。

yacc描述文件的格式：

%{

声明部分

%}

辅助定义部分

%%

规则部分

%%

用户函数部分

2、完成以下编程任务：

**任务1-1**：双链DNA分子中，G、C碱基对所占比例越高，其稳定性越强。编写一个lex描述文件，计算指定碱基序列里G、C碱基的比例。

**任务1-2**：编写一个lex描述文件，识别出指定文本串里的单词、数字和符号（空格不作处理）。

**任务1-3**：编写一个yacc描述文件，实现具有加法和乘法功能的计算器。

## 1.4编程要求

完成上述3个编程任务，将C语言源程序复制粘贴到右侧代码编辑器，点击“评测”按钮，运行程序，系统会自动进行结果对比。

## 1.5测试说明

**任务1-1**：

|  |
| --- |
| 输入：ACGTTGATCGGAATCTTCGT  输出：0.450  解释：9÷20=0.450 |
| 输入：TTACGGTACCAATGCTAATGCCTA  输出：0.417  解释：10÷24≈0.417 |

**任务1-2**：

|  |
| --- |
| 输入：  using namespace std;  int main()  {      int year = 2023;      cout << "hello" << endl;      return 0;  }  输出：  using 单词  namespace 单词  std 单词  ; 符号  int 单词  main 单词  ( 符号  ) 符号  { 符号  int 单词  year 单词  = 符号  2023 数字  ; 符号  cout 单词  < 符号  < 符号  " 符号  hello 单词  " 符号  < 符号  < 符号  endl 单词  ; 符号  return 单词  0 数字  ; 符号  } 符号 |

**任务1-3**：

|  |
| --- |
| 输入：5\*7+2  输出：37 |
| 输入：9+3\*6  输出：27 |

## 1.6参考文献

1. 王生原,董渊,张素琴,吕映芝,蒋维杜. 编译原理(第3版). 北京：清华大学出版社,2015.
2. flex: The Fast Lexical Analyzer. https://github.com/westes/flex.

# 2词法分析

2.1任务描述

加深对词法分析工作过程的理解；强化对词法分析方法的掌握；能够采用C/C++或Java编程语言实现PL/0编译程序的词法分析程序，并使用自己编写的程序对PL/0编程语言程序段进行词法分析。

2.2相关知识

2.2.1 PL/0语言编译系统

**PL/0语言编译系统由编译程序和解释程序两部分组成，分别称为PL/0编译程序和类P-code解释程序。**PL/0语言编译系统的运行过程如图2.1所示，PL/0语言程序被PL/0编译程序转换为等价的类P-code程序，当编译程序正常结束时，PL/0语言编译系统会调用解释程序（也称类P-code虚拟机），解释执行所生成的目标程序[1]。其中，PL/0编译程序可以使用C/C++、Pascal或Java等各种语言进行编写。

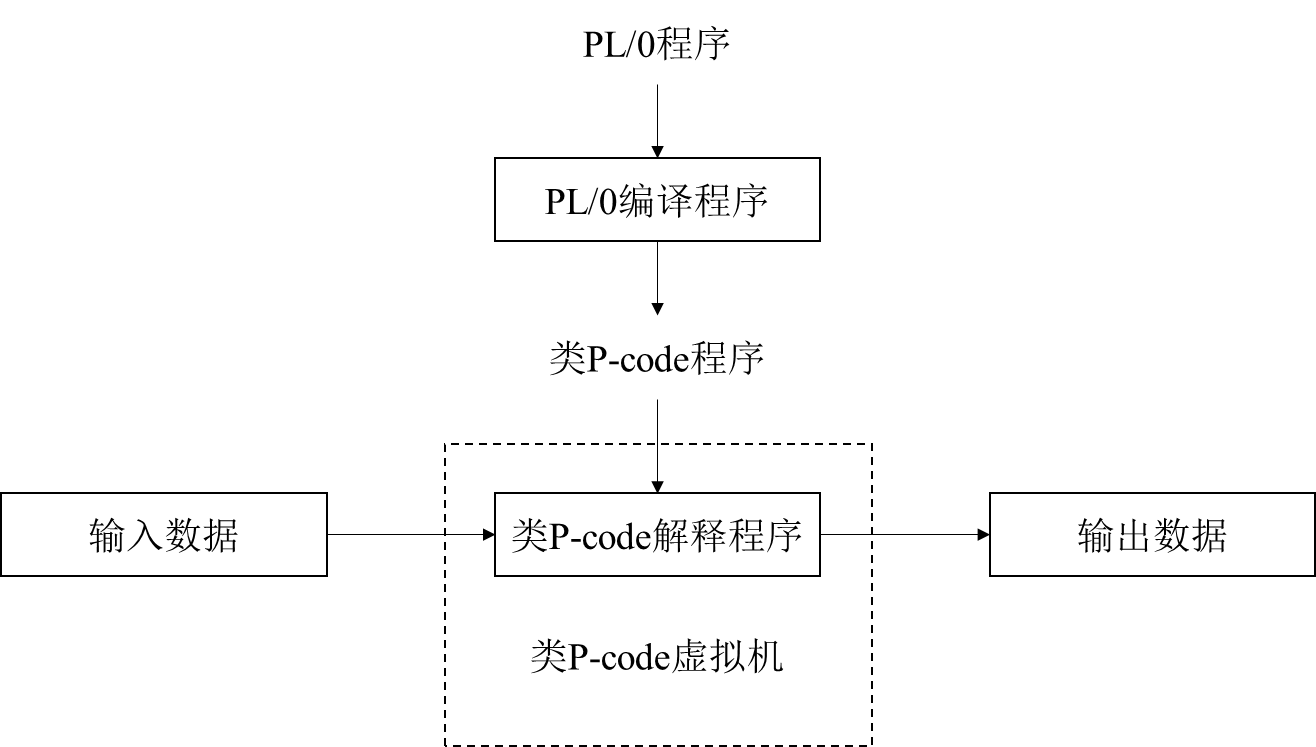


图 2.1 PL/0语言编译系统

2.2.2 扩展巴克斯范式

**扩展巴克斯范式（Extended Backus-Naur Form, EBNF）是程序语言语法描述的一种形式**，其用到的元符号及对应的含义如表2.1所示。

表 2.1 EBNF元符号及含义

|  |  |
| --- | --- |
| **元符号** | **含义** |
| **< >** | 用尖括号括起来的中文字表示语法构造成分，或称语法单位，为非终结符；而用尖括号括起来的英文字表示一类词法单元。 |
| ::= | 表示左部的语法单位由右部定义，可读作“定义为”。 |
| | | 表示“或”，即左部可由多个右部定义。 |
| { } | 用花括号括起来的成分可以重复0次到任意多次。 |
| [ ] | 用方括号括起来的成分为任选项，即出现一次或不出现。 |
| ( ) | 用圆括号括起来的成分优先。 |

2.2.3 词法分析

（参考教材37页-44页）

**词法分析是编译的第一个阶段，它的主要任务是从左往右逐个字符地对源程序进行扫描，产生一个个单词序列，用于语法分析。**执行词法分析的程序称为词法分析程序或扫描程序。语法分析程序调用词法分析程序的过程如图2.2所示，其原理为：将词法分析程序设计成一个子程序，每当语法分析程序需要一个单词时，调用该子程序；词法分析程序每得到一次调用，便从源程序文件中读入一些字符，直到识别出一个单词，或是直到下一个单词的第一个字符为止。

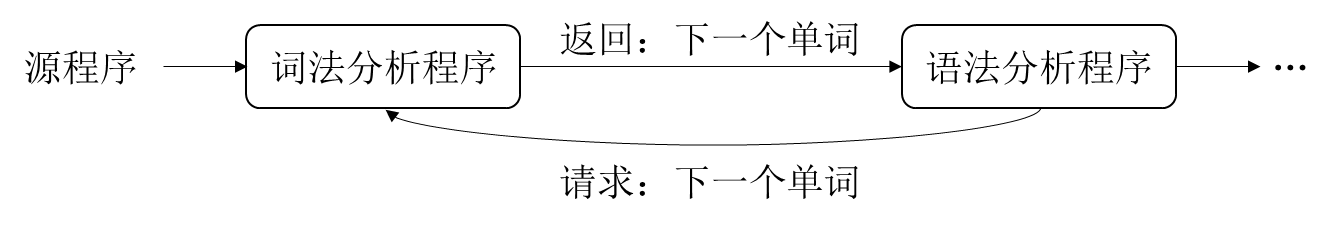


图 2.2 语法分析程序调用词法分析程序

PL/0的单词可以划分为5个大类：保留字（关键字）、标识符、运算符、无符号整数和界符。具体如下：

（1）保留字：共有13个，包括 const , var , procedure , begin , end , odd , if , then , call , while , do , read , write 。

（2）运算符：共有11个，包括4个整型算数运算符号 + 、 - 、 \* 和 / ，6个比较运算符号 < 、 <= 、 > 、 >= 、 # 和 = ，1个赋值运算符 := 。

（3）界符：共有5个，包括 ( 、 ) 、 , 、 ; 和 . 。

（4）无符号整数：<integer>是由一个或多个数字组成的序列，数字为 0 , 1 , 2 , … , 9 。

（5）标识符：<id>是字母开头的字母数字序列，字母包括大小写英文字母： a , b , ..., z , A , B , …, Z 。

**PL/0语言中5类单词的EBNF描述如下：**

<无符号整数>::=<数字>{<数字>}

<标识符> ::=<字母>{<字母>|<数字>}

<字母> ::= a | b | ... | X | Y | Z

<数字> ::= 0 | 1 | 2 | ... | 8 | 9

<保留字> ::= const | var | procedure | begin | end | odd | if | then | call | while | do | read | write

<运算符> ::= + | - | \* | / | < | <= | > | >= | # | = | :=

<界符> ::= ( | ) | , | ; | .

2.2.4 词法分析中的算法

#### 1、NFA转化为最简DFA算法设计

NFA转化为DFA的算法过程

采用子集构造法。首先定义三个概念如下：

(1)状态集的ε-闭包:状态集I中的任何状态s经任意条ε弧而能到达的所有状态的集合，定义为状态集I的ε-闭包，表示为ε-closure()。

(2)状态集的a弧转换:状态集I中的任何状态s经过一条a弧而能到达的所有状态的集合，定义为状态集1的a弧转换，表示为move(l,a)。

(3)状态集的a弧转换的闭包Ia=ε−closure(move(I,a))

算法步骤如下所示：

（1）每次从队头取出一个集合，(开始队列内只有初态集合I的ε-闭包(I))，然后得到它对于任意一个字符a的Ia=ε−closure(move(I,a))

（2）然后如果当前状态之前没有出现过，那么当前状态作为一个新的状态I,放入队列。

（3）一直做如上操作，直到队列为空。

#### 2、DFA最小化的算法过程

先了解几个概念：

1.多于状态：对于一个状态Si，若从开始状态出发，不可能到达改状态Si，则Si为多余（无用）状态。

2.死状态：对于一个状态Si，对于任意输入符号a，若转到它本身后，不可能从它到达终止状态，则称Si为死状态。

都称为无关状态

3.等价状态：若Si为自动机的一个状态，我们把从Si出发能导出的所有符号串的集合记为L（Si）。

设有两个状态Si和Sj，若有L（Si）=L（Sj），则称Si和Sj是等价状态。

4.可区别状态：自动机中两个状态Si和Sj，如果它们不等价，则称它们可区别。

5.两个状态(Si和Sj)等价的判断条件:

（1）状态Si和Sj必须同时为终止状态或同时为非终止状态。即终止状态和非终止状态是可区别的。

（2）状态Si和Sj对于任意输入符a∈Σ，必须转到等价的状态里，否则Si和Sj是可区别的。

DFA的化简算法：对于DFA M=(S,Σ,f,S0,Z)

(1)首先将DFA的状态集进行初始化，分成Π=(Z,S-Z);

(2)用下面的过程对Π构造新的划分Πnew

for (Π中每个组G) do//每个组都是一个状态集

begin

把G划分成小组，G中的任意两个状态Si和Sj在同一组中，当且仅当对于Σ中任意输入符号a，Si和Sj的a转换是到同一组中，move(Si,a)∈Gi，move(Sj,a)∈Gi。这样，只要Si和Sj的a转换是到不同的组中，则说明Si和Sj是可区别的，可进行划分。在Πnew中用刚完成的对G的划分代替原来的G。

end ;

Π:=Πnew；

（3）重复执行（2），直到Π中每个状态集不能再划分(Πnew=Π)为止;

（4）合并等价状态 ,在每个G中，取任意状态作为代表，删去其它状态;

（5）删去无关状态，从其它状态到无关状态的转换都成为无定义。

2.2.5设计提示

#### 1、处理文件

1）读取源文件的内容，标准套路，（参考C语言的文件操作函数，fopen函数等）

2）把源文件的内容分解为词法单元，词法单元就是单词。

单词有类别码（本次课程设计，请设计好单词的分类，参考教材41页）

#### 2、怎样分解词法单元：有穷自动机，递归分析程序

怎样编写词法分析程序：

1）预处理：把源文件一个字符一个字符的读入词法分析程序设置的输入字符结构体数组中（输入缓冲区），读入过程要删除注释，删除多余的白空格.

2）输入缓冲区要记录每个符号的ASCII码，以及该符号在源程序中的坐标值（行号和列号）

输入符号结构体数组：

struct inchar

{

char input；

int x，y；

}INCHAR[]；

3）从源程序字符数组中获得单词, 编码为二元式.

二元式采用结构体数组存储, 把单词类型和词元记录下来

例如:

struct dual

{ int dual\_type;

union {

char lexeme\_text[]；

int lexeme\_num[];

} lexeme

} DUAL[]

4）分解单词的方法：

利用switch-case多分支语句根据单词的特点直接编写。

通过描述单词的正规文法得到相应的有穷自动机，通过case多路转换语句完成有穷自动机的处理流程。参考课本P41-43的例子

#### 4、编写词法分析程序要注意的问题：

1)检查词法是否有错误

检查是否有非法字符：如a, aBC, @, &, !

检查是否有非法单词：如12A, BC\_1,

检查标志符和数字是否满足限制条件

检查注释符号是否配对

2)界符分隔单词

能够区分两个单词的符号为界符

有些界符不是单词：如白空格

有些界符仅仅用来分隔：如；

有些界符本身还是源程序不可缺少的单词，如(, ), +, /, 等等

有些界符包含两个字符：如<>, >=等等

3）输出词法错误

如果没有错误，返回0

如果有错误，需要报告词法错误在源程序中的位置。并且，要能够越过错误，分解下一个单词，直到源程序结束。

4) 输出的二元式流保存在二元式结构体数组中，供语法分析使用。

为了方便接下来的语法分析能够定位语法错误在源程序中的位置，需要在二元式结构体数组中保留每个单词在源程序中的坐标（单词的首字母所在的行号列号）。

实际的二元式结构体数组应该是：

struct dual

{ int dual\_type;

union {

char lexeme\_text[]；

int lexeme\_num[];

} lexeme

int x, y;

} DUAL[]

2.3实验步骤

完成以下编程任务（二选一）：

**任务1-1**：使用C/C++语言编写PL/0编译程序的词法分析程序。

**任务1-2**：使用Java语言编写PL/0编译程序的词法分析程序。

需要注意的点：

（1）识别非法字符：如 @ 、 & 和 ! 等；

（2）识别非法单词：数字开头的数字字母组合；

（3）标识符和无符号整数的长度不超过8位；

（4）能自动识别并忽略/\* \*/及//格式的注释信息；

（5）词法分析过程中遇到错误后能继续往下识别，并输出错误信息。

2.4编程要求

完成上述的任一编程任务，将源程序复制粘贴到右侧代码编辑器，点击“评测”按钮，运行程序，系统会自动进行结果对比。

## 2.5测试说明

**任务1-1/1-2**：

|  |
| --- |
| 输入：  const a = 10;  var   b, c;  procedure fun1;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0 do          begin              call fun1;              write(2 \* c);              read(b);          end  end.  输出：  (保留字,const)  (标识符,a)  (运算符,=)  (无符号整数,10)  (界符,;)  (保留字,var)  (标识符,b)  (界符,,)  (标识符,c)  (界符,;)  (保留字,procedure)  (标识符,fun1)  (界符,;)  (保留字,if)  (标识符,a)  (运算符,<=)  (无符号整数,10)  (保留字,then)  (保留字,begin)  (标识符,c)  (运算符,:=)  (标识符,b)  (运算符,+)  (标识符,a)  (界符,;)  (保留字,end)  (界符,;)  (保留字,begin)  (保留字,read)  (界符,()  (标识符,b)  (界符,))  (界符,;)  (保留字,while)  (标识符,b)  (运算符,#)  (无符号整数,0)  (保留字,do)  (保留字,begin)  (保留字,call)  (标识符,fun1)  (界符,;)  (保留字,write)  (界符,()  (无符号整数,2)  (运算符,\*)  (标识符,c)  (界符,))  (界符,;)  (保留字,read)  (界符,()  (标识符,b)  (界符,))  (界符,;)  (保留字,end)  (保留字,end)  (界符,.) |
| 输入：  const 2a = 123456789;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure function1;      if 2a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b @ 0 do          begin              call function1;              write(2 \* c);              read(b);          end  end.  输出：  (保留字,const)  (非法字符(串),2a,行号:1)  (运算符,=)  (无符号整数越界,123456789,行号:1)  (界符,;)  (保留字,var)  (标识符,b)  (界符,,)  (标识符,c)  (界符,;)  (保留字,procedure)  (标识符长度超长,function1,行号:10)  (界符,;)  (保留字,if)  (非法字符(串),2a,行号:11)  (运算符,<=)  (无符号整数,10)  (保留字,then)  (保留字,begin)  (标识符,c)  (运算符,:=)  (标识符,b)  (运算符,+)  (标识符,a)  (界符,;)  (保留字,end)  (界符,;)  (保留字,begin)  (保留字,read)  (界符,()  (标识符,b)  (界符,))  (界符,;)  (保留字,while)  (标识符,b)  (非法字符(串),@,行号:17)  (无符号整数,0)  (保留字,do)  (保留字,begin)  (保留字,call)  (标识符长度超长,function1,行号:19)  (界符,;)  (保留字,write)  (界符,()  (无符号整数,2)  (运算符,\*)  (标识符,c)  (界符,))  (界符,;)  (保留字,read)  (界符,()  (标识符,b)  (界符,))  (界符,;)  (保留字,end)  (保留字,end)  (界符,.) |

注：**直接print测试集的输出，视为本关不通过**。

2.6参考文献

1. 王生原,董渊,张素琴,吕映芝,蒋维杜. 编译原理(第3版). 北京：清华大学出版社,2015.

# 3语法分析

3.1任务描述

加深对语法分析工作过程的理解；强化对语法分析方法的掌握；基于词法分析程序，能够采用C/C++或Java编程语言实现PL/0编译程序的语法分析程序，并使用自己编写的程序对PL/0编程语言程序段进行语法分析。

3.2相关知识

3.2.1 语法分析

语法分析的作用是识别由词法分析给出的单词符号串是否是给定文法的正确句子（程序）。PL/0语言语法可用EBNF形式描述，其中的一个EBNF描述如表3.1所示[1]。

表 3.1 PL/0语言语法的EBNF描述

|  |  |
| --- | --- |
| PL/0语法单位 | EBNF描述 |
| <程序> | ::=<分程序>. |
| <分程序> | ::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句> |
| <常量说明部分> | ::=const<常量定义>{,<常量定义>}; |
| <常量定义> | ::=<标识符>=<无符号整数> |
| <变量说明部分> | ::=var<标识符>{,<标识符>}; |
| <过程说明部分> | ::=<过程首部><分程序>{;<过程说明部分>}; |
| <过程首部> | ::=procedure<标识符>; |
| <语句> | ::=<赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<空语句> |
| <赋值语句> | ::=<标识符>:=<表达式> |
| <复合语句> | ::=begin<语句>{;<语句>}end |
| <空语句> | ::=ɛ |
| <条件> | ::=<表达式><关系运算符><表达式>|odd<表达式> |
| <表达式> | ::=[+|-]<项>{<加减运算符><项>} |
| <项> | ::=<因子>{<乘除运算符><因子>} |
| <因子> | ::=<标识符>|<无符号整数>|’（’<表达式>’）’ |
| <加减运算符> | ::=+|- |
| <乘除运算符> | ::=\*|/ |
| <关系运算符> | ::==|#|<|<=|>|>= |
| <条件语句> | ::=if<条件>then<语句> |
| <过程调用语句> | ::=call<标识符> |
| <当型循环语句> | ::=while<条件>do<语句> |
| <读语句> | ::=read’（’<标识符>{,<标识符>}’）’ |
| <写语句> | ::=write’（’<表达式>{,<表达式>}’）’ |

注：PL/0中变量的数据类型只有整型。

**语法分析常用的程序设计方法可分为自顶向下分析和自底向上分析两类。**其中，自顶向下分析包括预测分析法和递归下降分析法，自底向上分析包括LR分析法和算符优先分析法[2]。此外，还可以通过语法分析器自动生成工具（例如yacc）编写语法分析程序。

3.2.2 预测分析法

预测分析法（Forecasting Parse）要求文法必须是LL(1)文法，如图3.1所示，分析过程将按照自左至右的顺序读入输入符号串，并在此过程中产生一个句子的最左推导。它采用表驱动的方式，通过显示地维护一个状态栈和一个二维预测分析表，在总控程序的控制下实现分析过程。按此方式执行语法分析任务的程序称为预测分析程序或预测分析器。

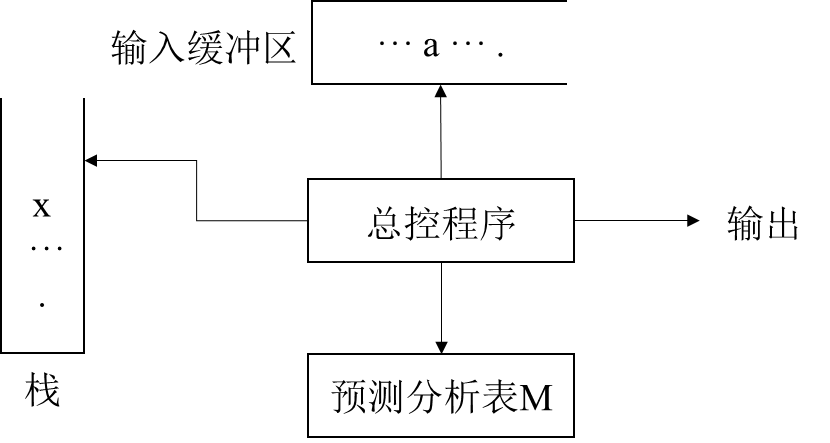


图 3.1 预测分析器模型

3.2.3 递归下降分析法

递归下降分析（Recursive Descent Parse）是直接以程序的方式模拟产生式产生语言的过程，它的基本思想是：为每个非终结符构造一个函数，每个函数的函数体按非终结符的候选式分情况展开，遇到终结符就进行比较，看是否与输入符号串匹配；遇到非终结符就调用该非终结符对应的函数。分析过程从调用文法开始符号对应的函数开始，直到所有非终结符都展开为终结符并得到匹配为止。如果分析过程中达到这一步则表明分析成功，否则表明输入符号串有语法错误。由于文法是递归定义的，因此函数也是递归的。如图3.2所示，递归下降分析器的输入是词法分析输出的token串，通过分析检查输入的token串是否符合文法要求，输出语法树和错误信息。

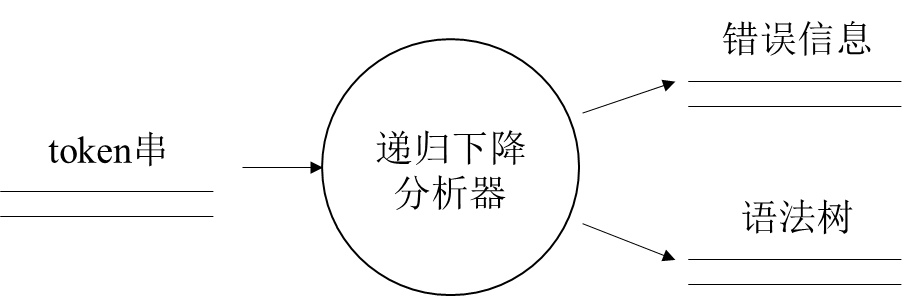


图 3.2 递归下降语法分析器语法分析程序接口

3.2.4 LR分析法

LR分析（Left to Right Parse）的基本思想是：在规范归约过程中，一方面用栈存放已移进和归约出的整个符号串，即记住“历史”；另一方面，LR分析器还要面对“现实”的当前输入符号，再根据所用产生式推测未来可能碰到的输入符号，即对未来的“展望”。当某可归约符号串出现在栈顶时，需要根据已记载的“历史”、“展望”和“现实”的输入符号三方面的内容来决定栈顶的符号串是否构成了真正的句柄，是否能够进行归约。LR分析器的工作过程如图3.3所示。

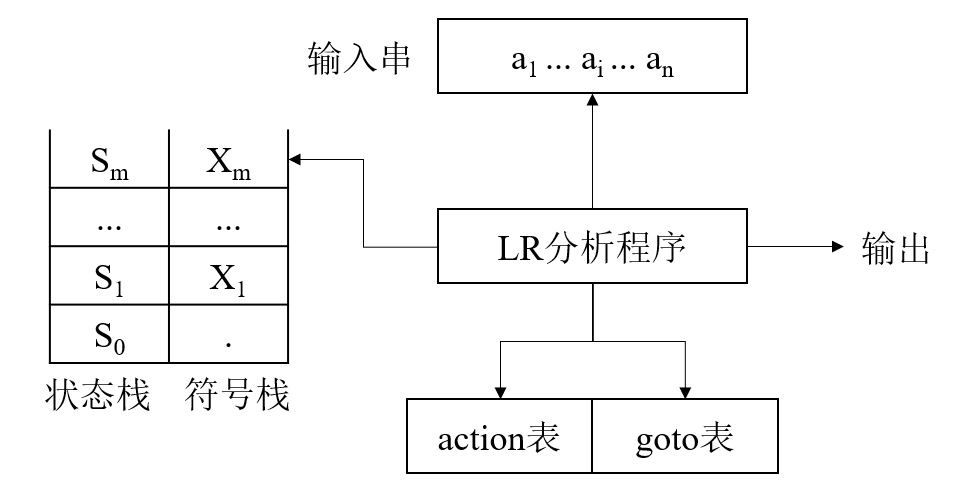


图 3.3 LR分析器工作过程

### 3.2.5 算符优先分析法

算符优先分析（Operator Precedence Parse）需要构造算符优先关系表，它通过对任意给定的符号串进行算符优先分析，判断输入符号串是否为对应文法的句子。具体思想为：通过比较相邻终结符号间的优先关系来进行分析，仍然采用“移进-归约”方式，不断移进输入符号，识别可归约串，并进行归约。但是，由于该分析法仅考虑了终结符之间的优先关系，没有考虑非终结符之间的优先关系，所以每次归约的并不一定是当前句型的句柄。实际上，算符优先分析法不是用句柄来刻画“可归约串”，而是用最左素短语来刻画“可归约串”。此外，需要注意的是，并不是所有文法都能用算符优先分析法进行分析，只有算符优先文法才能进行算符优先分析。

3.2.6 自动工具分析法

利用语法分析器自动生成工具（如yacc）编写语法分析程序，可以克服其他分析方法中（如LR分析法）手工构造分析表的缺点。yacc的使用已在第一章中介绍过，感兴趣的同学可自行编写yacc的源程序，实现语法分析过程。

3.2.7 LL（1）文法相关算法

#### 1、求first集的算法过程

求解FIRST(X)

1）若X∈VT，则FIRST(X)={X}。

2）若X∈VN，且有产生式X→a……，a∈VT，则a∈FIRST(X)

3）若X∈VN，X→ε，则ε∈FIRST(X)

4）若X,Y1,Y2,……,Yn∈VN，而有产生式X→Y1Y2,……Yn。当Y1,Y2,……,Yi-1推出ε时，则FIRST(Y1)-ε，FIRST(Y2)-ε,……,FIRST(Y(i-1)-ε)，FIRST(Yi)都包含在FIRST(X)中

5）当4）中所有Yi都推出ε时，则最后的FIRST(X)=FIRST(Y1)∪FIRST(Y2)∪……∪FIRST(Yn)∪{ε}

#### 2、求follow集的算法过程

求解FOLLOW(X)

1)S是文法开始符号，初始化follow(S)={#}

2) D→BAC follow(A)=first(C)-ε

3)如果上述产生式D→BAC 中，存在Cε，即D→BA，那么还要将follow(D)加入到follow(A)中

#### 3、求select集的算法

A→，若不能推导出空，那么select(A→) = first()，如果经过有限次推导能推出ε，那么那么select(A→)={first()- ε}∪Follow(A)

#### 4、求预测分析表的算法

根据select集合来填写预测分析表。

3.2.8 SLR相关算法描述

#### 1、构造LR(0) 项目集规范族

（1）假定I是文法G’的任一项目集，定义和构造I的闭包CLOSURE(I)的办法是：

I的任何项目都属于CLOSURE(I)；

若A→α•Bβ属于CLOSURE(I)，那么，对任何关于B的产生式B→γ，项目B→•γ也属于CLOSURE(I)；

重复执行上述两步骤直至CLOSURE(I)不再增大为止。

（2）Go(I,a)函数构造算法

遍历所有的项目，如果任意两个项目之间存在边（有向），那么这两个项目所在的项目规范族之间连上对应的有向边。

#### 2、SLR解决冲突的方法：

对于项目集

考察集合FOLLOW(A)和FOLLOW(C)，如果这两个集合不相交，而且也不包含b，那么当状态Ii面临输入符号a时，我们可以使用如下策略：

1. 若a=b，则移进。
2. 若a∈FOLLOW(A)，则用产生式A→α进行归约；
3. 若a∈FOLLOW(B)，则用产生式C→α进行归约；
4. 其他均置error

#### 3、SLR语法分析表的构造方法：

根据构造的LR(0)项目集规范族C和活前缀识别自动机的状态转换函数GO。函数ACTION和GOTO可按如下方法构造：

1. 若项目A→ɑ••bβ属于Ik，GO(Ik,a)= Ij, a为终结符，置ACTION[k,a]为“把状态j和符号a移进栈”，简记为“sj”；
2. 若项目A→α•属于Ik，那么，对任何非终结符a，a∈FOLLOW(A)，置ACTION[k,a]为“用产生式A→α进行归约”，简记为“rj”；其中，假定A→α为文法G’的第j个产生式
3. 若项目S’→S•属于Ik，则置ACTION[k,#]为可“接受”，简记为“acc”；
4. 若GO(Ik, A)= Ij，A为非终结符，则置GOTO[k, A]=j；
5. 分析表中凡不能用规则1至4填入信息的空白格均填上“出错标志”。

3.3实验步骤

1、选择上述一种语法分析方法，完成以下编程任务（二选一）：

**任务1-1**：基于第二章的词法分析程序，使用C/C++语言编写PL/0编译程序的语法分析程序。

**任务1-2**：基于第二章的词法分析程序，使用Java语言编写PL/0编译程序的语法分析程序。

3.4编程要求

完成上述的任一编程任务，将源程序复制粘贴到右侧代码编辑器，点击“评测”按钮，运行程序，系统会自动进行结果对比。

3.5测试说明

**任务1-1/1-2**：

|  |
| --- |
| 输入：  const a = 10;  var   b, c;  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  语法正确 |
| 输入：  const a := 10;  var   b, c;  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  (语法错误,行号:1) |
| 输入：  const a = 10;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a          end;  begin      read(b);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  (语法错误,行号:13) |
| 输入：  const a = 10;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  (语法错误,行号:17) |
| 输入：  const a := 10;  var   b, c d;  //单行注释  /\*  \* 多行注释  \*/  procedure procedure fun1;      if a <= 10          begin              c = b + a          end;  begin      read(b;      while b # 0          begin              call fun1;              write 2 \* c);              read(b);          end  end.  输出：  (语法错误,行号:1)  (语法错误,行号:2)  (语法错误,行号:10)  (语法错误,行号:11)  (语法错误,行号:13)  (语法错误,行号:16)  (语法错误,行号:17)  (语法错误,行号:20) |

注：本次任务各小组选择的语法分析方法及程序设计存在差异，因此通过部分或全部测试集均可通关**(直接print测试集的输出，视为本关不通过)**。

3.6参考文献

1. 王生原,董渊,张素琴,吕映芝,蒋维杜. 编译原理(第3版). 北京：清华大学出版社,2015.
2. 黄贤英,王柯柯,曹琼,魏星. 编译原理及实践教程(第3版). 北京：清华大学出版社,2019.

# 4 语义分析

4.1任务描述

加深对语义分析工作过程的理解；强化对语义分析方法的掌握；基于词法分析和语法分析程序，能够采用C/C++或Java编程语言实现PL/0编译程序的语义分析程序，并使用自己编写的程序对PL/0编程语言程序段进行语义分析；能够生成PL/0编程语言四元式形式的中间代码。

4.2相关知识

4.2.1 编译过程

**编译程序完成从源程序到目标程序的翻译工作，是一个复杂的整体的过程。**从概念上来讲，一个编译程序的整个工作是划分成阶段进行的，每个阶段将源程序的一种表示形式转换成另一种表示形式，各个阶段进行的操作在逻辑上是紧密连接在一起的。图4.1给出了一个编译过程的各个阶段，这是一种典型的划分方法，它将编译过程划分成词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成6个阶段[1]。

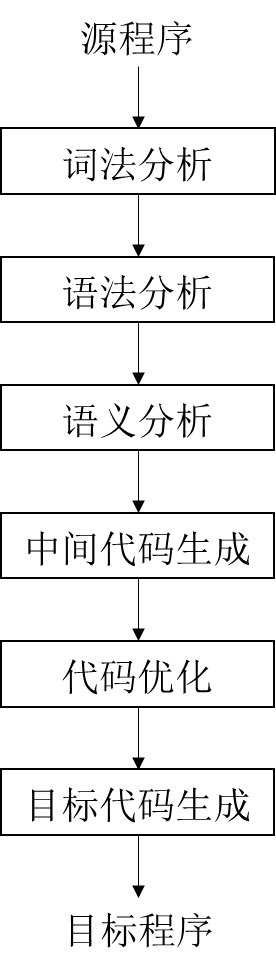


图 4.1 编译过程的各个阶段

4.2.2 语义分析

**语义是指源程序及其组成部分所表述的含义。**语义与语法的区别在于：语法是关于程序及其组成部分的构成规则的描述，是上下文无关的，而语义是关于语法结构的含义及其使用规则的描述，是上下文有关的。**在语法分析正确的基础上，需要进行语义检查。**只有当源程序的语义正确了，编译程序才能用另一种形式的语言将其含义表达出来，实现翻译，即“忠实地”将源程序所表达的含义传递给目标程序。也就是说，源程序编译后的目标程序虽然在语法结构上不同，但它们所表达的语义必须是一致的，否则编译就失去了意义。因此，在进行翻译之前，必须检查这些语法正确的语法单位其内部逻辑含义是否正确，称为**语义分析**[2]。

4.2.3 静态语义分析

**静态语义刻画程序在静态一致性或完整性方面的特征，而动态语义刻画程序执行时的行为。**编译器根据语言的静态语义规则完成静态语义分析。静态语义分析过程中若发现程序有不符合静态语义规则之处，则报告语义错误；若没有语义错误，则称该程序通过了静态语义检查。语义分析过程中与静态语义检查相关的部分称为**静态语义分析**。

静态语义分析的**主要任务**包括：

（1）借助符号表进行上下文相关的静态语义分析；

（2）确保符号表可以体现作用域规则；

（3）确保标识符属性与上下文环境一致；

（4）确保标识符先声明后引用；

（5）确保标识符唯一；

（6）确保过程的嵌套层数符合语言的约定；

（7）提示语义错误信息。

4.2.4 中间代码生成

**中间代码是源程序的不同表现形式，也称为中间表示。**对于已通过静态语义检查的程序，编译程序可将其翻译为后续的中间表示形式，即**中间代码生成**。中间代码生成的过程体现了如何在更低的级别诠释程序的动态语义。其作用包括：（1）搭建源语言和目标语言之间的桥梁，避开二者之间较大的语义跨度，使编译程序的逻辑结构更加简单明确；（2）有利于编译程序的重定向；（3）有利于进行与目标机无关的优化。

**三地址代码**（Three-Address Code, TAC）是中间代码的一种抽象形式，其比较接近汇编语言。**四元式**是三地址代码的一种具体实现，它是具有四个域的记录结构，具体表示为：

（op,arg1,arg2,result）

其含义为：arg1和arg2进行op指定的操作，结果存放到result中。其中，op为运算符，arg1、arg2和result分别为第一、第二运算对象和结果，它们可以是用户定义的变量或临时变量，arg1和arg2还可以是常量。如果op是单目运算符，只需要一个运算对象，则arg2用下划线“\_”来代替，为它留出位置。**通过静态语义检查的程序，可以将其翻译为四元式表示的中间代码。**表4.1为PL/0编程语言常见的四元式代码形式，另外，**标识符的声明通常存放在符号表里。**

表 4.1 常见的四元式代码形式

|  |  |  |
| --- | --- | --- |
| **编号** | **四元式代码** | **含义** |
| (1) | (syss,\_,\_,\_) | 程序开始（system start） |
| (2) | (syse,\_,\_,\_) | 程序结束（system end） |
| (3) | (const,ident,\_,\_) | 声明标识符ident（常量） |
| (4) | (var,ident,\_,\_) | 声明标识符ident（变量） |
| (5) | (procedure,ident,\_,\_) | 声明标识符ident（函数） |
| (6) | (+,A,B,T) | 加法运算，将A+B的结果赋给T |
| (7) | (-,A,B,T) | 减法运算，将A-B的结果赋给T |
| (8) | (\*,A,B,T) | 乘法运算，将A\*B的结果赋给T |
| (9) | (/,A,B,T) | 除法运算，将A/B的结果赋给T |
| (10) | (<,A,B,T) | 小于，将A<B的结果赋给T |
| (11) | (<=,A,B,T) | 小于或等于，将A<=B的结果赋给T |
| (12) | (>,A,B,T) | 大于，将A>B的结果赋给T |
| (13) | (>=,A,B,T) | 大于或等于，将A>=B的结果赋给T |
| (14) | (#,A,B,T) | 不等于，将A#B的结果赋给T |
| (15) | (=,A,B,T) | 等于，将A=B的结果赋给T |
| (16) | (=,A,\_,T) | 将A（的值）赋给常量T |
| (17) | (:=,A,\_,T) | 将A（的值）赋给变量T |
| (18) | (jrop,A,B,P) | 当A rop B为真时跳转到四元式P |
| (19) | (read,A,\_,\_) | 为变量A输入值 |
| (20) | (write,A,\_,\_) | 输出A的值 |
| (21) | (call,fun,\_,A) | 调用fun函数，返回值存入变量A |
| (22) | (call,fun,\_,\_) | 调用fun函数，不保存返回值 |
| (23) | (ret,A,\_,\_) | 返回返回值A |
| (24) | (ret,\_,\_,\_) | 返回，无返回值 |

4.2.5 代码优化

代码优化工作可以在编译的各个阶段进行。本质上讲，在保证程序的含义保持一致的情况下，可以对代码进行任何修改。也就是说，代码优化不应该改变程序运行的结果，必须要保证优化后的代码和原来的代码完成相同的工作。常见的代码优化技术有窥孔优化、局部优化、循环优化和全局优化等。

（注：感兴趣的同学可以自行实现代码优化过程。）

4.2.6 目标代码生成

编译过程最后阶段的工作是生成目标代码生成，完成目标代码生成的程序称为目标代码生成器。目标代码生成就是要将与机器无关的中间代码翻译为某个具体机器的指令代码，目标代码可以是绝对机器代码、可重定位机器代码或汇编代码等。

（注：感兴趣的同学可以自行实现目标代码生成过程。）

4.3实验步骤

1、完成以下编程任务（二选一）：

**任务1-1**：基于第二章的词法分析程序和第三章的语法分析程序，使用C/C++语言编写PL/0编译程序的语义分析程序，并生成四元式形式的中间代码。

**任务1-2**：基于第二章的词法分析程序和第三章的语法分析程序，使用Java语言编写PL/0编译程序的语义分析程序，并生成四元式形式的中间代码。

4.4编程要求

完成上述的任一编程任务，将源程序复制粘贴到右侧代码编辑器，点击“评测”按钮，运行程序，系统会自动进行结果对比。

4.5测试说明

**任务1-1/1-2**：

|  |
| --- |
| 输入：  const a = 10;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  语义正确  中间代码:  (1)(syss,\_,\_,\_)  (2)(const,a,\_,\_)  (3)(=,10,\_,a)  (4)(var,b,\_,\_)  (5)(var,c,\_,\_)  (6)(procedure,p,\_,\_)  (7)(j<=,a,10,$8)  (8)(+,b,a,c)  (9)(ret,\_,\_,\_)  (10)(read,b,\_,\_)  (11)(j#,b,0,$13)  (12)(j=,b,0,$17)  (13)(call,p,\_,\_)  (14)(\*,2,c,T1)  (15)(write,T1,\_,\_)  (16)(read,b,\_,\_)  (17)(syse,\_,\_,\_)  符号表:  const a 10  var b 0  var c 0  procedure p |
| 输入：  const a = 10;  var   a, b, c;  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(b);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  (语义错误,行号:2) |
| 输入：  const a = 10;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(p);      while b # 0 do          begin              call p;              write(2 \* c);              read(b);          end  end.  输出：  (语义错误,行号:16) |
| 输入：  const a = 10;  var   b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(p);      while b # 0 do          begin              call q;              write(2 \* c);              read(b);          end  end.  输出：  (语义错误,行号:19) |
| 输入：  const a = 10;  var   a, b, c;  //单行注释  /\*  \* 多行注释  \*/  procedure p;      if a <= 10 then          begin              c := b + a;          end;  begin      read(p);      while b # 0 do          begin              call q;              write(2 \* d);              read(b);          end  end.  输出：  (语义错误,行号:2)  (语义错误,行号:16)  (语义错误,行号:19)  (语义错误,行号:20) |

注：**直接print测试集的输出，视为本关不通过**。

4.6参考文献

1. 王生原,董渊,张素琴,吕映芝,蒋维杜. 编译原理(第3版). 北京：清华大学出版社,2015.
2. 黄贤英,王柯柯,曹琼,魏星. 编译原理及实践教程(第3版). 北京：清华大学出版社,2019.