第九章 错误处理

9.1 概述

1. 错误处理是编译器的必备功能之一

一般情况下，用户开发源程序都难免出现各种错误。一般有语法和语义错误，还有键入疏忽等。

正确的源程序：通过编译，生成目标代码。

错误的源程序：通过编译，发现并指出错误。

编译程序对于语法和语义正确的源程序要正确地编译生成等价的目标代码；而对于错误的源程序不能一发现就停止，而是要能检查出错误的性质和出错位置，并使编译能继续下去，同时尽可能多而准确地发现错误和指出各种错误。

2. 编译器的错误处理能力

(1) 诊察错误的能力。

(2) 报错及时准确（出错位置，错误性质）。

(3) 一次编译找出错误的多少。

(4) 改正错误的能力。

(5) 遏制重复错误信息的能力。

用户希望使用错误处理能力强的编译器，尽可能快地得到一个语法语义正确的源程序。

9.2 错误的分类

从编译角度，将错误分为两类：语法错误和语义错误。

语法错误：程序结构不符合语法（包括词法）规则的错误。

语义错误：程序不符合语义规则或超越具体计算机系统的限制。

超越系统限制：（计算机系统和编译系统）

1. 数据溢出错误，常数太大，计算结果溢出。
2. 符号表、静态存储分配数据区溢出。
3. 动态存储分配数据区溢出。

9.3 错误的诊察和报告

错误诊察:

1. 违反语法和语义规则以及超过编译系统限制的错误。由编译程序在语法和语义分析过程中诊察出来。（语义分析要借助符号表）

2. 下标越界、计算结果溢出以及动态存储数据区溢出等在目标程序运行时才能检测，因此由目标程序诊察。

对此，编译程序要生成相应的目标程序代码进行检查并处理。

错误报告：

1. 出错位置：即源程序中出现错误的位置。

实现：设立行号计数器　　line-no

设立单词序号计数器　char-no

一旦诊察出错误，当前的计数器内容就是出错位置。

2. 出错性质：

可直接显示文字信息

可给出错误编码

3. 报告错误: （两种方式）

(1) 分析以后再报告（显示或者打印）

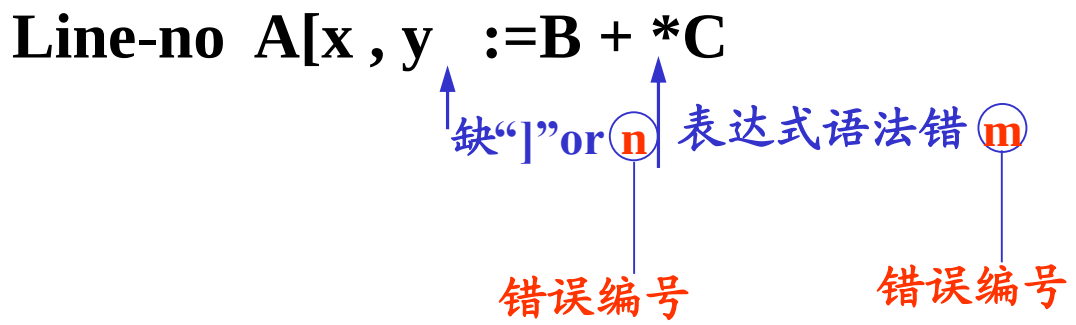
编译程序可设一个保存错误信息的数据区（可用记录型数组)，将语法语义分析所诊断到的错误送数据区保存，待源程序分析完以后，统一显示或打印错误信息。

例： A[x , y :=B +*C

源程序行号	单词序号	错误性质
x x	6	缺少“]”
x x	10	表达式语法错误

(2) 边分析边报告

在分析一行源程序时若发现有错，可以立即输出该行源程序，并在其下输出错误信息。



有时报错不一定十分准确（位置和性质），需进一步分析。

例如有**ALGOL**程序段:

```
begin
    .......
for i := 1  step 1   until  n  do
    .......

end
```

9.4 错误处理技术

发现错误后，在报告错误的同时还要对错误进行处理，以方便编译能继续进行下去。目前有两种处理办法：

1. 错误改正：指编译诊察出错误以后，根据文法进行错误改正。

但不是总能做到，如：A := B – C * D + E)

- 是多一个右括号还是少一个左括号？
- 如果是少了左括号，应该少在何处？

所以，要正确地改写错误是很困难的！

2. 错误局部化处理： 指当编译程序发现错误后，尽可能将把错误的影响限制在一个局部的范围，避免错误扩散和影响程序其它部分的分析。

(1) 一般原则

(2) 错误局部化处理的实现

(3) 提高错误局部化程度的方法

(1) 一般原则

当诊断到错误以后，就暂停对后面符号的分析，跳过错误所在的语法成分（一旦跳过就认为该语法成分是正确的）然后继续往下分析。

词法分析：发现不合法字符，显示错误，并跳过该标识符（单词）继续往下分析。

语法语义分析：跳过所在的语法成分（短语或语句），一般是跳到语句右界符，然后从新语句继续往下分析。

(2) 错误局部化处理的实现（递归下降分析法）

CX：全局变量，存放错误信息。

- 用递归下降分析时，如果发现错误，便将有关错误信息（字符串或者编号）送CX，然后转出错误处理程序；
- 出错程序先打印或显示出错位置以及出错信息，然后跳出一段源程序，直到跳到语句的右界符（如end、；），或正在分析的语法成分的合法后继符号为止，然后再往下分析。

```
procedure  if statement;
 begin
    nextsym;                          /*读下个单词符号*/
    B;                                /*调用布尔表达式处理程序*/
    if  not  class = ‘then’ then
      begin
        cx := ‘缺then’;               /*错误性质送cx*/
        error;                        /*调用出错处理程序*/
       end;
    else
       begin
          nextsym;
          statement;
        end;
    if class = ‘else’ then
       begin
            nextsym;
            statement;
       end;
end  if  statement;
```

局部化处理的出错程序为:

```
procedure error;
      begin
        write(源程序行号, 序号, cx)
        repeat
          nextsym;
        until class =‘ ; ’ or class =‘ end ’ or class =‘ else ’
end  error;
```

可以看出，如发现错误就立即跳到语句结尾处（语句右界符 end , ; 等）。这样的处理比较粗糙，将跳过太多。上例中缺then，就将跳过整个条件语句，使得then后面的语句都被跳过而不分析，其中若有错误就发现不了。

(3) 提高错误局部化程度的方法

设 S_1：合法后继符号集（某语法成分的后继符号）

S_2：停止符号（跳读必须停止的符号集）

当发现错误时: error(S_1, S_2)

```
procedure   error( S1, S2 )
      begin
            write( line-no, char-no, cx);
             repeat
                   nextsym
              until( class in  S1 or class in  S2 );
      end error;
```

上面例题中的 if 语句中，若<B>有错，则可跳到then;

若statement有错，则可跳到 else。

if < B > then < statement > [else < statement >] ;

4、LL分析的错误恢复----补充（第四章内容！）

当符号栈顶的终结符和下一个输入符号不匹配，或栈顶是非终结符A，输入符号a，而M[A,a]为空白（即error）时，则分析发现错误。

错误恢复的基本思想是：跳过一些输入符号，直到期望的同步符号之一出现为止。

同步符号（可重新开始继续分析的输入符号）集合通常可按以下方法确定：

1) 把FOLLOW(A)的所有符号加入A的同步符号集。如果我们跳读一些输入符号直到出现FOLLOW(A)的符号，之后把 A 从栈中弹出，继续往下分析即可。

2) 只用FOLLOW(A)作为非终结符A的同步符号集是不够的(容易造成跳读过多，如输入串中缺少语句结束符分号时)。此时可将作为语句开头的关键字加入它的同步符号集，从而避免这种情况的发生。

3) 把FIRST(A)的符号加入非终结符A的同步符号集中。

4) 如果非终结符A可以产生空串，那么推导ε的产生式可以作为缺省的情况。这样做可以推迟某些错误检查，但不会漏过错误。

5) 如果终结符在栈顶而不能匹配，则可弹出该终结符并发出一条信息后继续分析。这好比把所有其他符号均作为该符号的同步集合元素。

3. 目标程序运行时错误检测与处理

- 下标变量下标值越界
- 计算结果溢出
- 动态存储分配数据区溢出

在编译时生成检测该类错误的代码。

对于这类错误，要正确地报告出错误位置很难，因为目标程序与源程序之间难以建立位置上的对应关系。

一般处理办法：

当目标程序运行检测到这类错误时，就调用异常处理，打印错误信息和运行现场（寄存器和存储器中的值）等，然后停止程序运行。