# 第十六章 编译程序生成方法和工具

- 编译程序的书写语言
- 自编译性
- 自展
- 编译程序的移植
- 编译程序的自动生成

# 16.1 编译程序的书写语言

- 机器语言或汇编语言

  主要优点：编出来的程序效率高。

  主要缺点：编程效率低，可读性差，不便于修改和移植。

- 高级程序设计语言已基本取代汇编语言

  优点：编程效率高，可读性好，利于移植。

  缺点：编译程运行效率较低。

# 自编译性

**自编译性：如果一个高级语言能用来书写自己的编译程序，则该语言具有自编译性，并称该语言为自编译语言。**

**两点说明：**

1. **通常用自编译语言除可编写本语言的编译程序以外，也可用来编写别的语言的编译程序。**

   **∴如果某台机器上已配备有某种自编译语言，则可利用这种语言为本台机器配置其它的高级语言。**

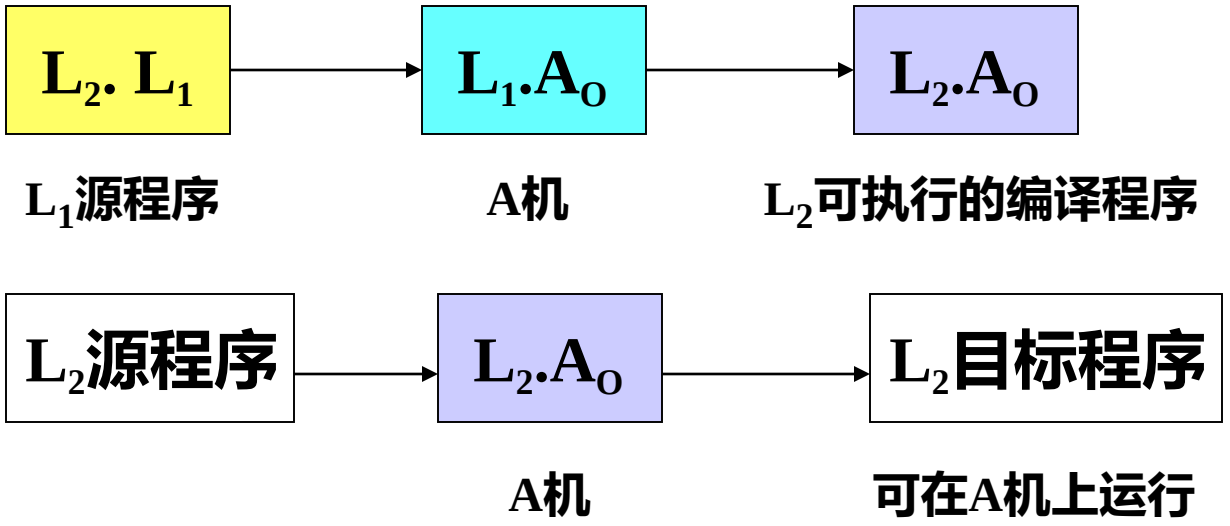
例：A机上有自编译语言$L_1$的编译程序

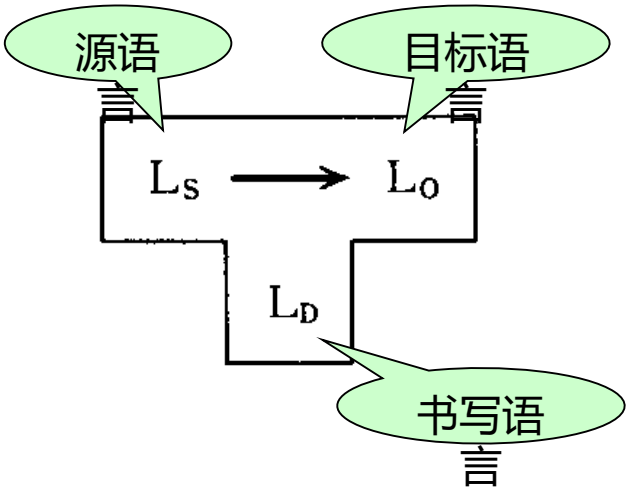
$L_1 . A_O$

$L_1$——语言$L_1$的编译程序

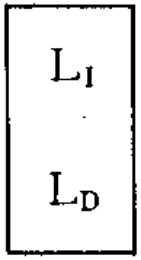
$A_O$——以A机的机器指令形式给出

利用语言$L_1$可为A机生成语言$L_2$的编译程序



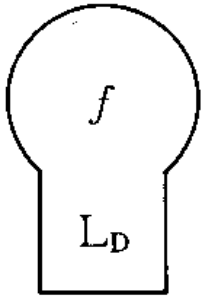



(a)编译程序

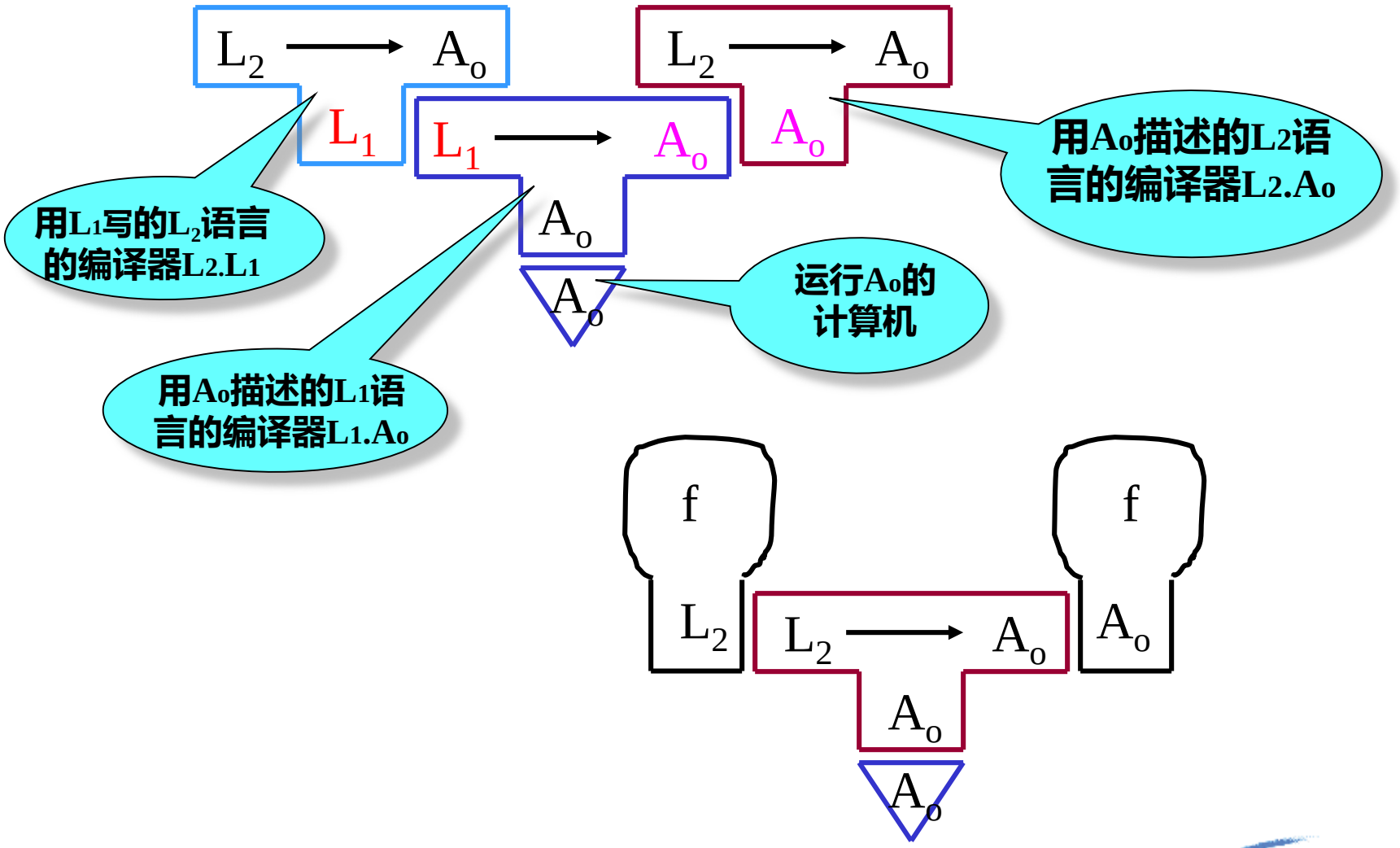



(b)解释程序


(c)计算机




(d)程序


L2 → Ao
L1
L1 → Ao
Ao
Ao
L2 → Ao
Ao
用L1写的L2语言的编译器L2.L1
用Ao描述的L1语言的编译器L1.Ao
运行Ao的计算机
用Ao描述的L2语言的编译器L2.Ao
f
L2
L2 → Ao
Ao
Ao
f
Ao

## 2.自编译性不是绝对的，只是强弱不同

数据类型丰富的语言
控制结构丰富的语言
} 自编译性强

**数据类型：** 除一般的外还有字符串类型，数组，结构，枚举，指针等类型。

**控制结构：** 应适于进行多分支的程序设计，如有CASE语句等

FORTRAN , ALGOL——自编译性差

PASCAL , C , ADA，C + + ，JAVA——自编译性强

**实践示例：** 用PASCAL语言编写一个简单的编译程序，就是利用PASCAL的自编译性。

# 16.2 自展

利用高级语言的自编译性，还可以通过自展方式生成语言的编译程序。

设L为自编译语言，自展生成

L. Ao（A机目标形式的语言L的编译器，可在A机上运行）

步骤：1.首先，将语言划分为N个部分：

$$L = L_1 + L_2 + \ldots + L_n$$

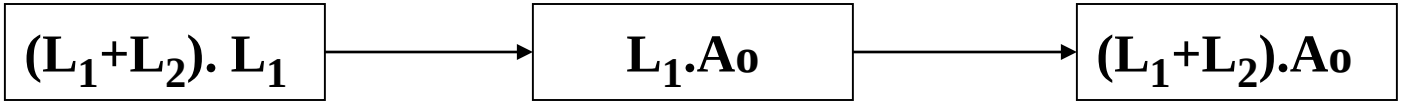
$L_1$ ——核心部分

$L_2$ ~ $L_n$——扩充部分

2.先用A机上的汇编编写$L_1$的编译程序，$L_1$.Aa

$L_1$.Aa→Assember→$L_1$.Ao

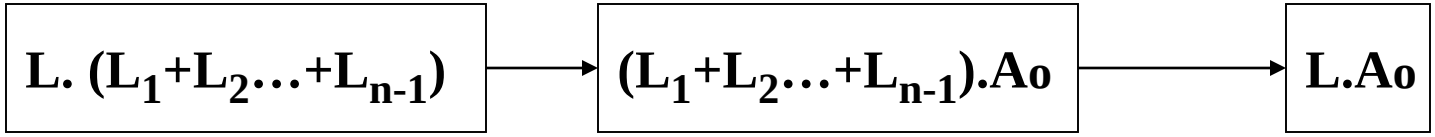
3.用$L_1$编写$L_1+L_2$的编译程序

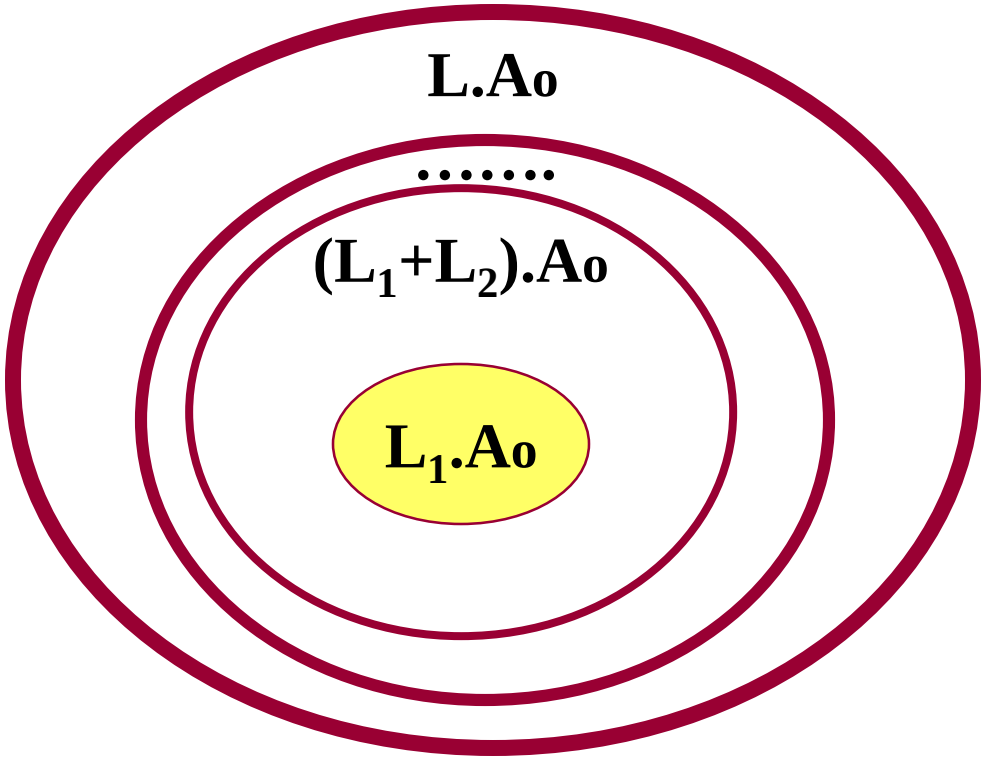



4.用$(L_1+L_2)$编写$L_1+L_2+L_3$的编译程序

…

5.





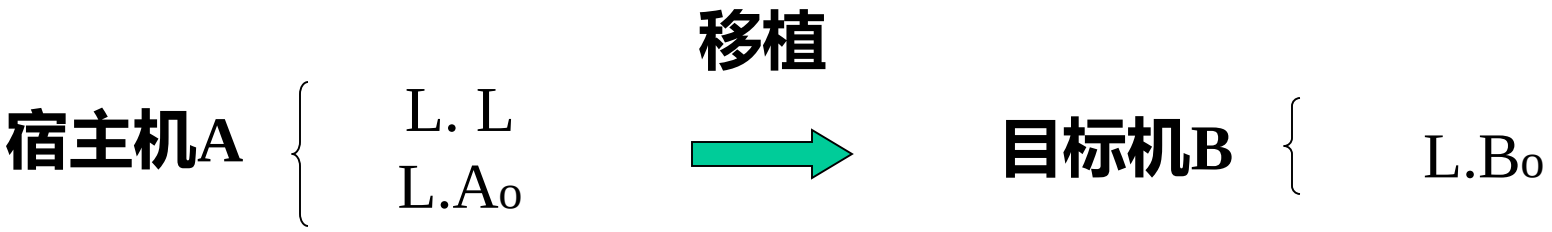

滚雪球式

用自展方式进行编译，可提高生产率。因核心语言小，可用汇编实现。其余部分高级语言编写。比全用低级语言效率高。

# 12.3 编译程序的移植

**移植：将某台机上的成熟软件移植到另一台机器上，也就是将宿主机上的软件移植到目标机上。**

**具有自编译性的高级语言来书写程序，则移植是方便的。**




**通过移植，在B机上可得到语言L的编译程序，具B机目标形式，可在B机上运行。**

## 移植步骤：

1. 将L. L分为两部分：

   一部分与机器无关 F.L　一部分与机器有关　A.L

   ∴ L. L = F.L+ A.L

2. 根据目标机用语言L改写与具体机器有关的部分：

   A.L $\xrightarrow{L}$ B.L

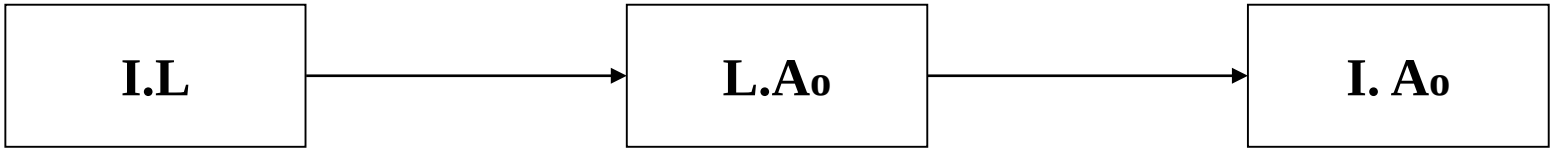
   产生A机代码　　产生B机代码

   ∴交叉编译器：　　I.L= F.L+ B.L

   用A机上的L语言所写的能生成B机目标代码的语言L的编译程序。

## 3. 第一次编译

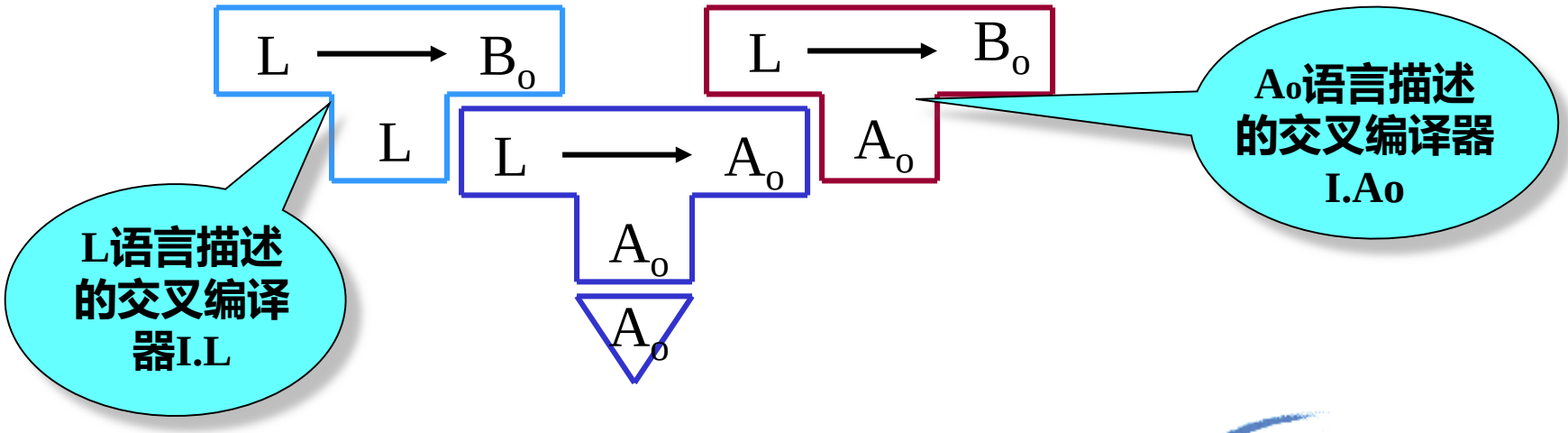
将I.L在宿主机A上用L的编译程序进行编译，生成能在宿主机A上运行的语言L的交叉编译器，它能生成目标机B的代码。



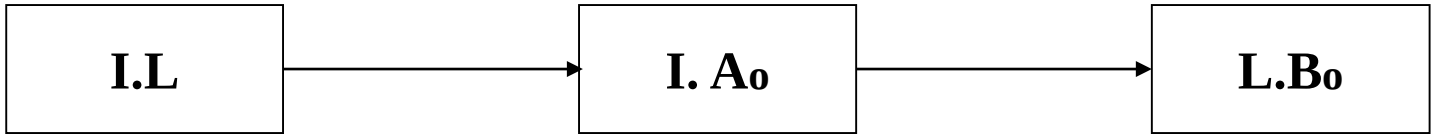

用L所写的生成目标机B代码的L语言交叉编译器源程序

宿主机A的L编译程序

语言L的交叉编译器，能在宿主机A上运行，生成目标机B的代码

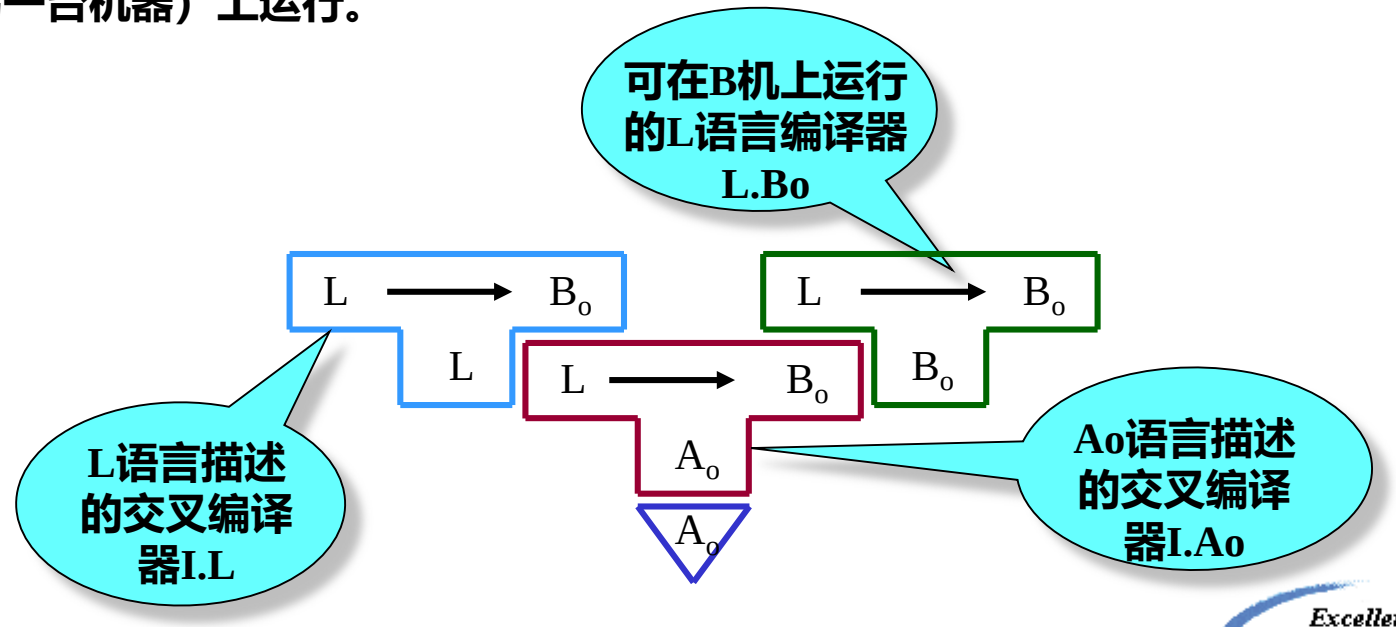


# 4. 第二次编译（交叉编译）




**交叉编译程序：** 在宿主机A上运行，→ A机
但所生成的目标只能在目标机B
（另一台机器）上运行。

可在目标机B上运行并生
成目标机B代码的L编译程序



可以设想，只要在某台机器上为某目标机配置一个L语言的交叉编译程序，就能将宿主机上的L语言所写的所有软件移植到其他目标机上。

采用软件移植的办法来开发软件，可提高软件生产率，并提高软件的可靠性。由于上述优点，所以软件的可移植性是软件开发所追求的目标之一。

目前有许多编译程序都考虑到可移植性的要求。

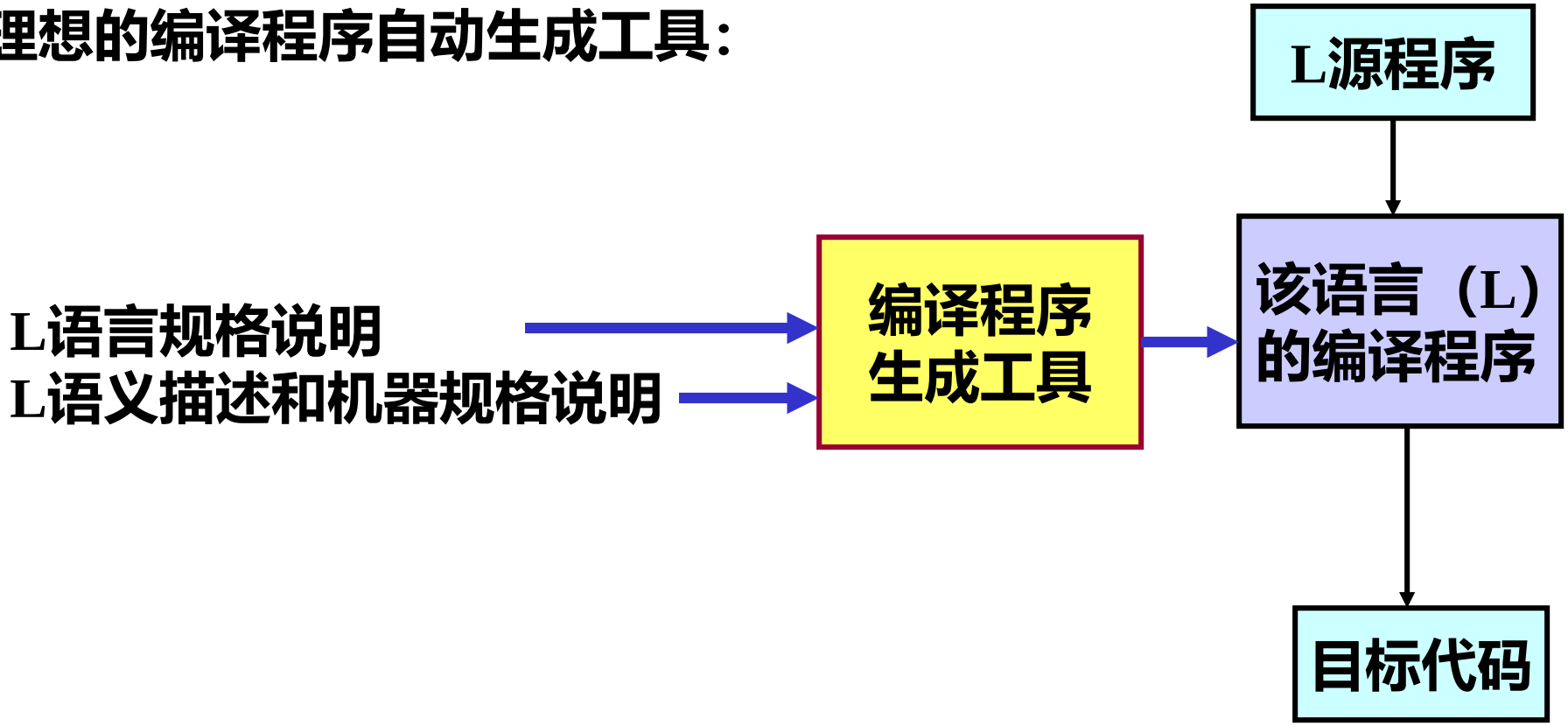
例如有：可移植的PASCAL编译程序。

P.J.Brown ， Software Portablility.

朱关铭等译，1982.12

# 16.4 编译程序的自动生成

**理想的编译程序自动生成工具：**

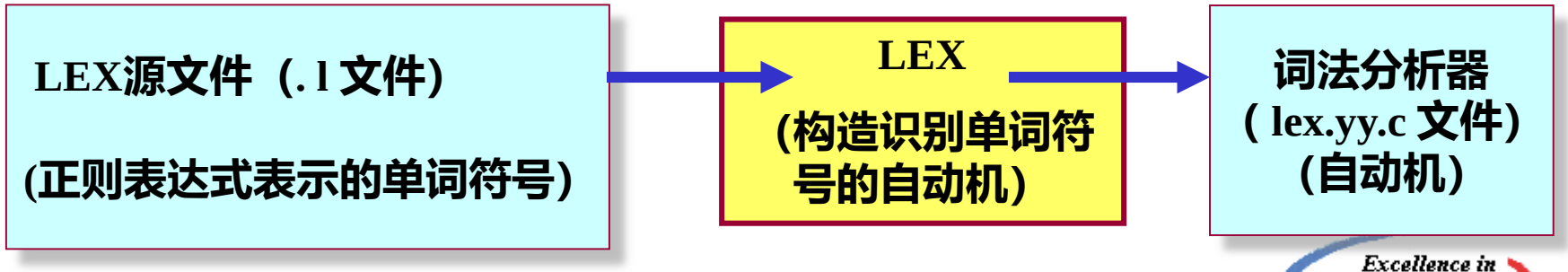


目前还没有一个系统能自动生成整个编译系统。

早期的工作集中在分析部分，即针对语法规则的形式化描述。对编译程序后端，即与目标机有关的代码生成与代码优化部分，由于对语义和目标机进行形式化描述方面所存在的困难，最近有所突破，但未见到流行的产品。（样机——未形成真正产品）

- 有词法分析器的自动生成器和语法分析器的自动生成器。

  词法分析器生成器（在第十一章已作介绍） LEX:



## 语法分析器生成器：

YACC ( YET ANOTHER COMPILER - COMPILER )


Bison: 美国GNU开发的语法分析器生成器)和YACC一样都在UNIX系统下运行。 (已有PC版)

# 用yacc建立翻译程序

yacc源程序：translate.y

1. 键入命令：

yacc translate.y

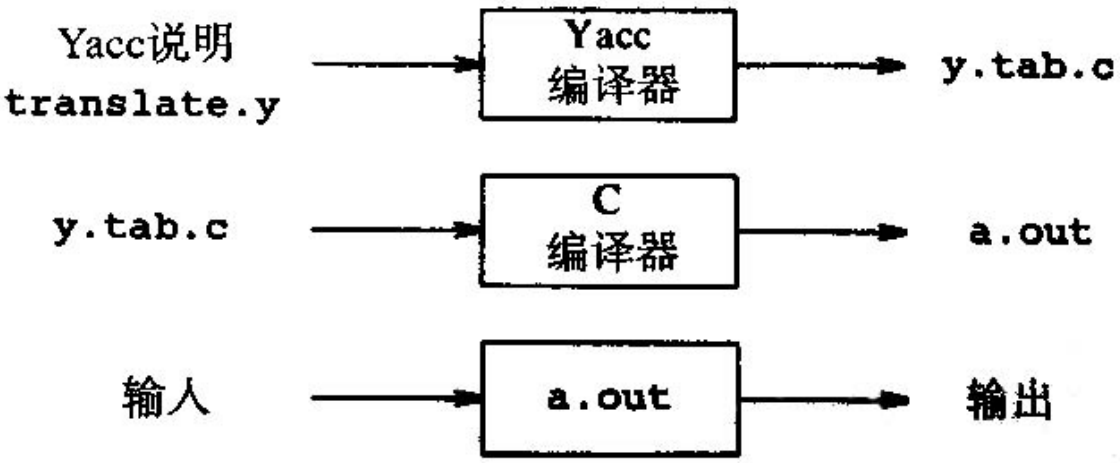
2. 生成进行LALR分析的翻译程序： y.tab.c

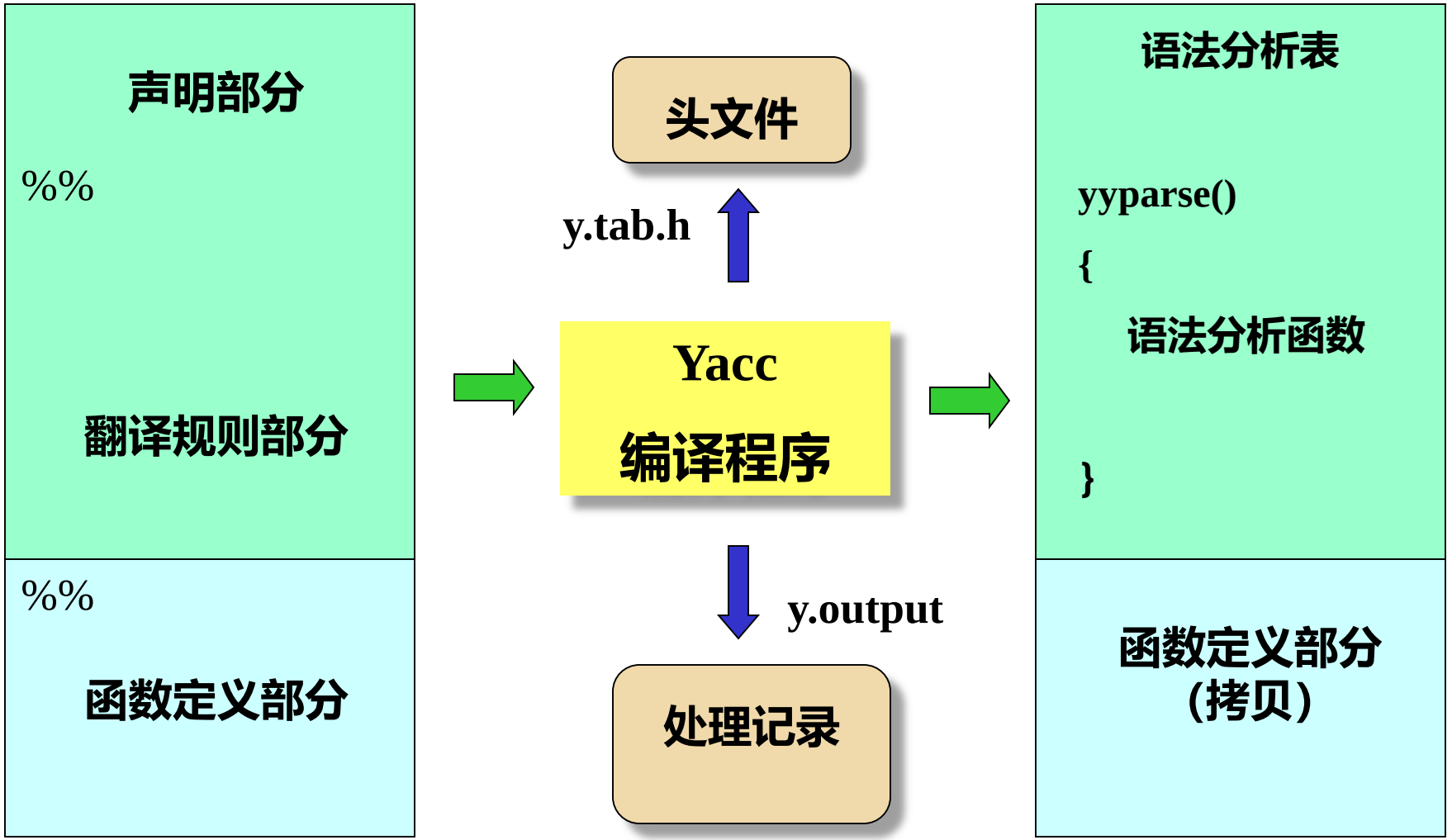
3. 对生成的分析器进行编译：

cc y.tab.c -ly

(ly为使用LR分析器的库)

生成可执行的 翻译程序 a.out




声明部分
%%
翻译规则部分
%%
函数定义部分
yacc源程序
( *. y )
头文件
y.tab.h
Yacc
编译程序
y.output
处理记录
语法分析表
yyparse()
{
语法分析函数
}
函数定义部分
(拷贝)
y.tab.c

```
1.   /* 表达式计算 */
2.   % token NUM
3.   %%
4.   line   : expr '\n'          { printf ('\n', $1);}
5.          ;
6.   expr   : expr '+' term      { $$ = $1 + $3; }
7.          | expr '-' term      { $$ = $1 - $3; }
8.          | term               /* $$ = $1 */
9.          ;
10.  term : term '*' factor      { $$ = $1 * $3; }
11.         | term '/' factor    { $$ = $1 / $3; }
12.         | factor             /* $$ = $1 */
13.         ;
14.  factor: '(' espr ')'        { $$ = $2; }
15.         | NUM                /* $$ = $1 */
16.         ;
```

```
%%
#include   <ctype.h>
yylex( )
{
        int  c;
        while (( c = getchar( )) == ' ' );
        if ( isdigit ( c ) )  {
            yylval = c – '0';
            while ( isdigit( c = getchar ( ) ) )
                yylval = yylval*10 + ( c-'0' );
            ungetc ( c, stdin );
            retun  NUM;      }
         else   return c;
}
```

# 简单台式计算器的yacc源程序

## 简单台式计算器语法：

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow ( E ) \mid digit$

digit为０－９的单个数字

yylex()为词法分析程序，它返回单词（类）和单词值

在本例中，单词为 DIGIT，单词值存入特定的变量 yyval中。

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :   expr '\n'          { printf("%d\n", $1); }
        ;
expr    :   expr '+' term      { $$ = $1 + $3; }
        |   term
        ;
term    :   term '*' factor    { $$ = $1 * $3; }
        |   factor
        ;
factor  :   '(' expr ')'       { $$ = $2; }
        |   DIGIT
        ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

语义动作中$$ 表示规则左部非终结符的值，$i表示规则右部第i个符号的值

# 改进的台式计算器yacc源程序

具有二义性文法的yacc源程序

yacc 缺省的解决冲突策略：

归约—归约冲突，按先出现的规则归约

移进—归约冲突，则移进优先

还可以在yacc源文件中指定 终结符的优先级和结合律。这时，当需在移进符号a 和按规则A→β进行归约之间进行选择时，若该规则的优先级高于a 或优先级相同但规则是左结合时，就进行归约，否则就选择移进。

规则（产生式）的优先级与它最右边的终结符优先级相同

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   :   lines expr '\n'   { printf("%g\n", $2); }
        |   lines '\n'
        |   /* ε */
        ;
expr    :   expr '+' expr     { $$ = $1 + $3; }
        |   expr '-' expr     { $$ = $1 - $3; }
        |   expr '*' expr     { $$ = $1 * $3; }
        |   expr '/' expr     { $$ = $1 / $3; }
        |   '(' expr ')'      { $$ = $2; }
        |   '-' expr  %prec UMINUS  { $$ = - $2; }
        |   NUMBER
        ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

# 用lex建立yacc的词法分析器

**lex源程序lexical.l：**

```
number      [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]         { /* 跳过空格 */ }
{number}    { sscanf(yytext, "%lf", &yylval);
              return NUMBER; }
\n|.        { return yytext[0]; }
```

**Yacc源程序第 3 部分的例程yylex( )语句应由语句 #include “lex.yy.c” 替代，并键入如下命令生成台式计算器程序 a.out：**

```
lex     lexical.l
yacc    translate.y
cc      y.tab.c  -ly  -ll
```

# 包含错误恢复的台式计算器yacc源程序

**Yacc 使用出错产生式**

**A →.error α 进行错误恢复**

**A为主要非终结符**

**error为yacc的保留字**

**α为符号串**

**yyerrok是将语法分析器恢复为正常操作模式的yacc例程**

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

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   :   lines expr '\n'   { printf("%g\n", $2); }
        |   lines '\n'
        |   /* empty */
        |   error '\n'{ yyerror ("重新输入上一行 ; ")
                        yyerrok; }
        ;
expr    :   expr '+' expr     { $$ = $1 + $3; }
        |   expr '-' expr     { $$ = $1 - $3; }
        |   expr '*' expr     { $$ = $1 * $3; }
        |   expr '/' expr     { $$ = $1 / $3; }
        |   '(' expr ')'      { $$ = $2; }
        |   '-' expr  %prec UMINUS  { $$ = - $2; }
        |   NUMBER
        ;
%%
#include "lex.yy.c"
```