# 计算机程序设计语言及编译

类似于数学定义或
自然语言的简洁形式

**高级语言**
**( High Level Language )**

- 接近人类表达习惯
- 不依赖于特定机器
- 编写效率高

引入助记符

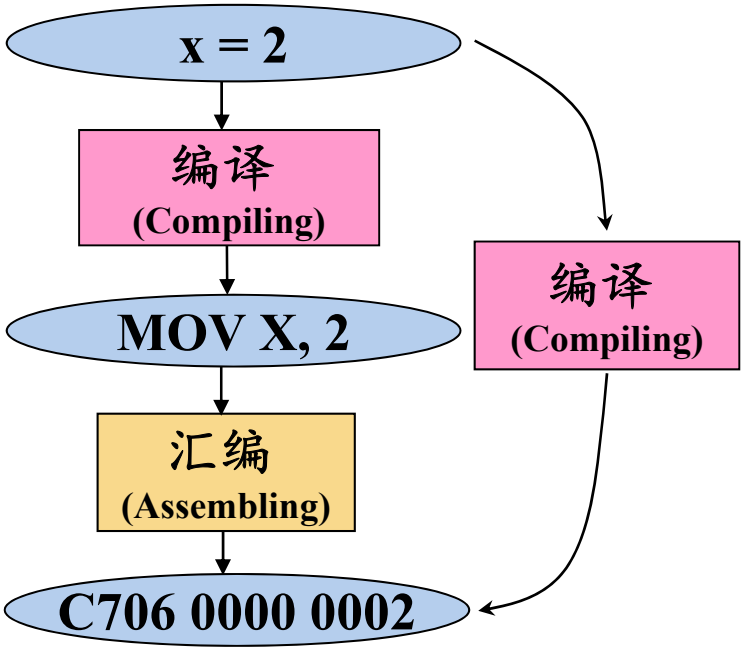
**汇编语言**
**(Assembly Language )**

- 依赖于特定机器，非计算机专业人员使用受限制
- 编写效率依然很低

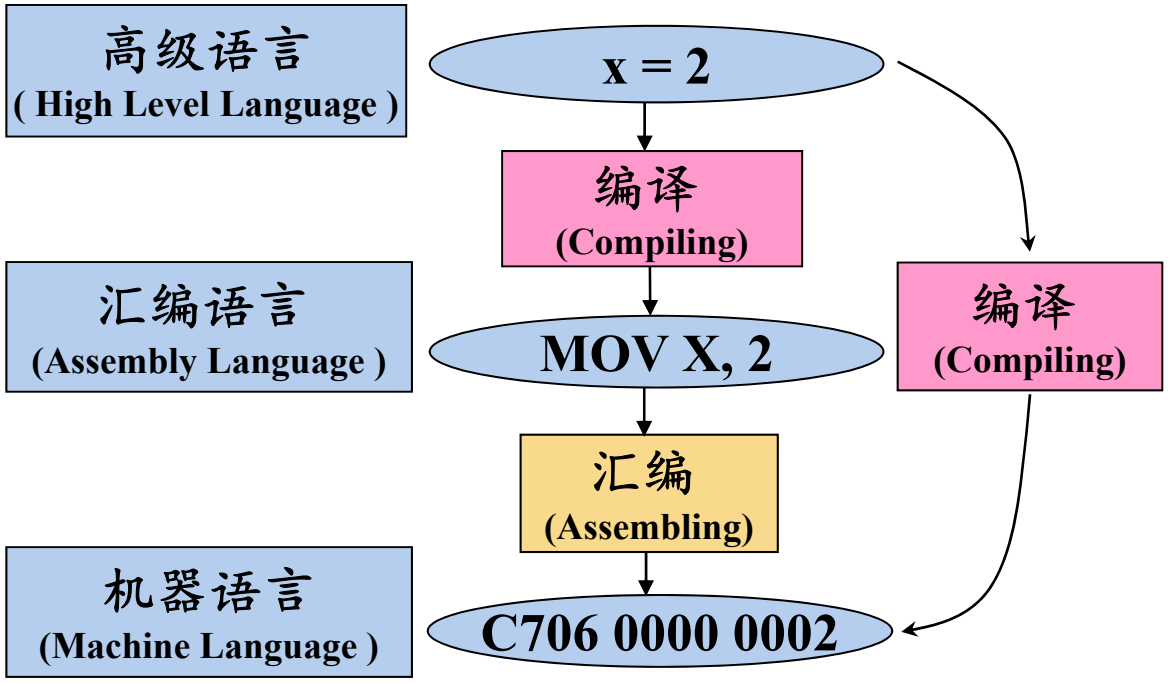
可以被计算机直接理解

**机器语言**
**(Machine Language )**

- 与人类表达习惯相去甚远
- 难记忆
- 难编写、难阅读
- 易写错



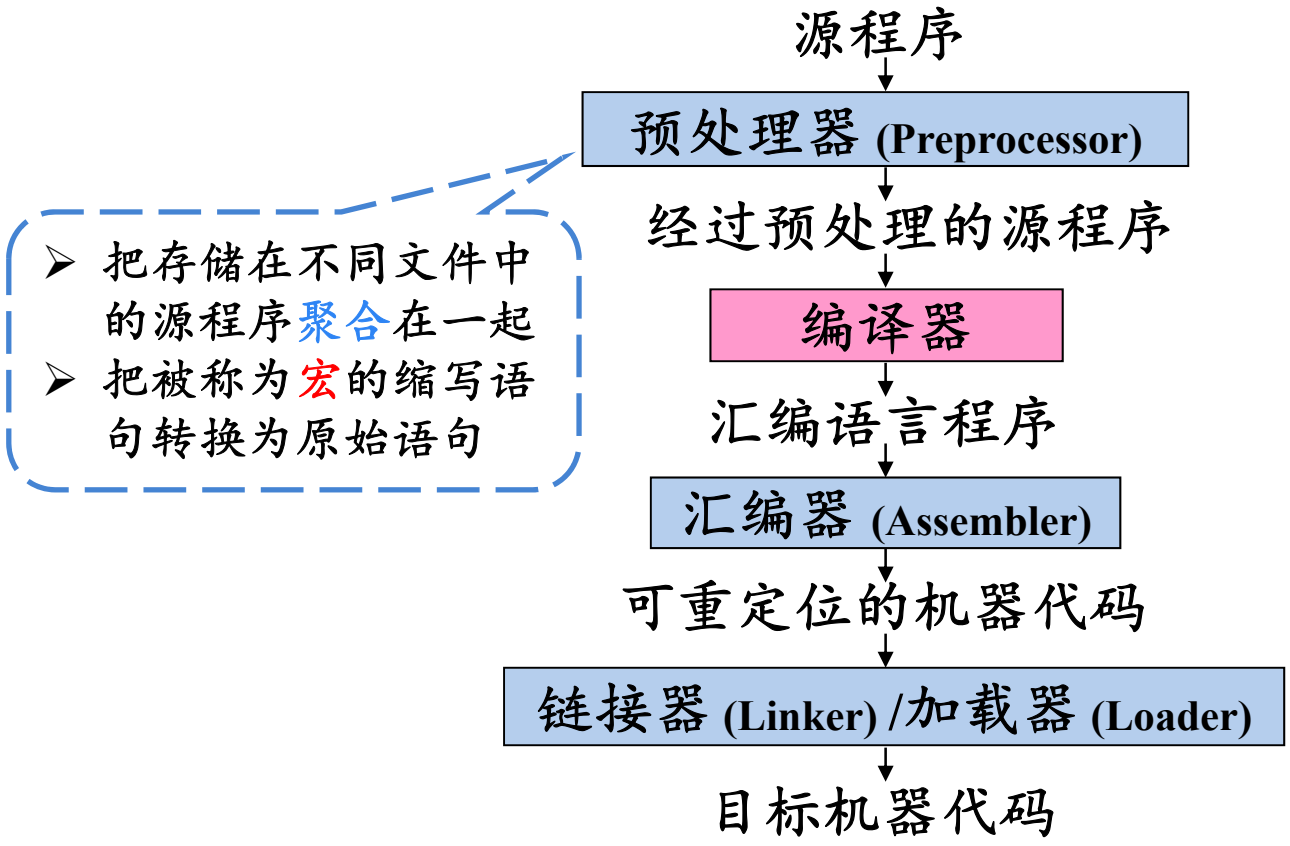
# 计算机程序设计语言及编译



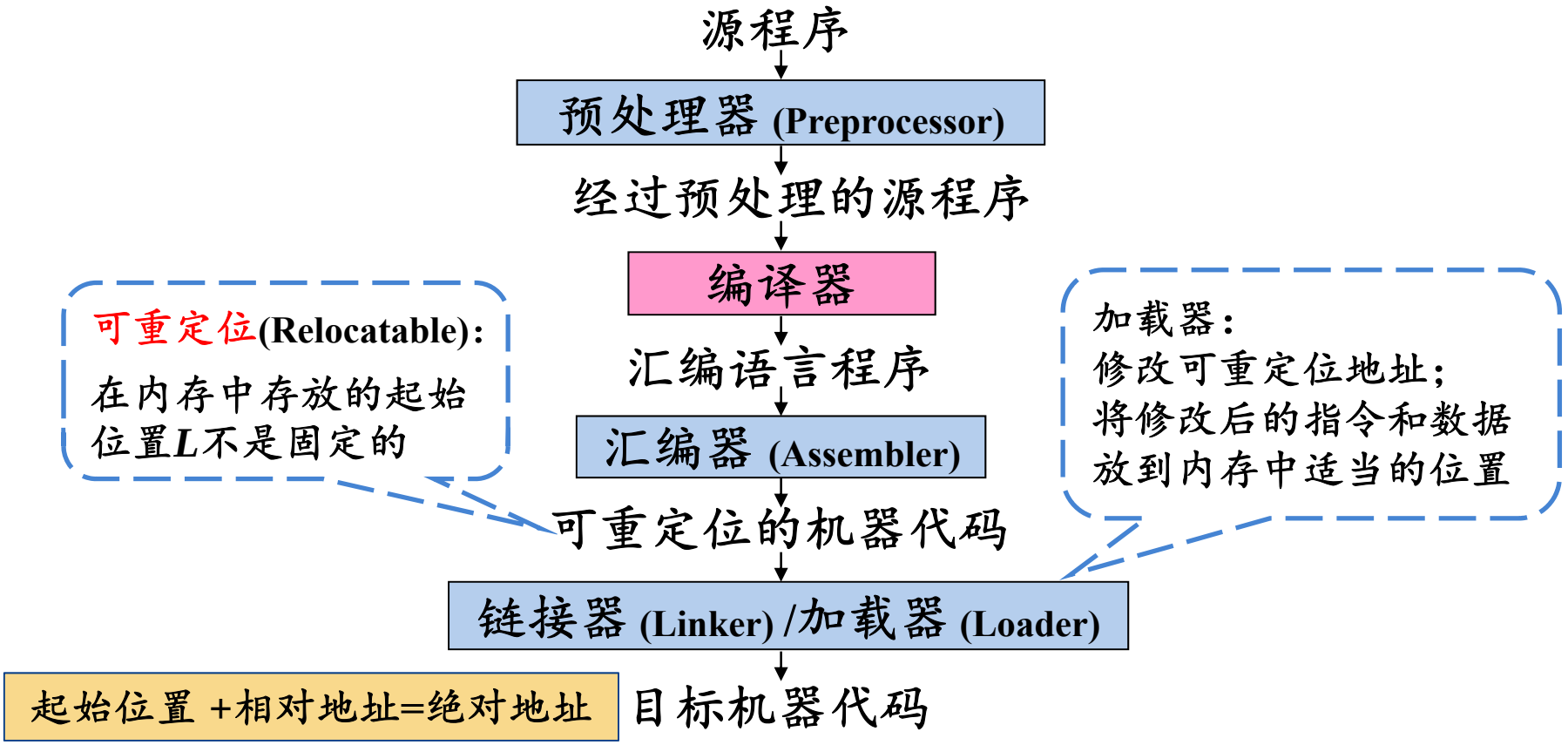

编译：将高级语言翻译成汇编语言或机器语言的过程

源语言　　目标语言

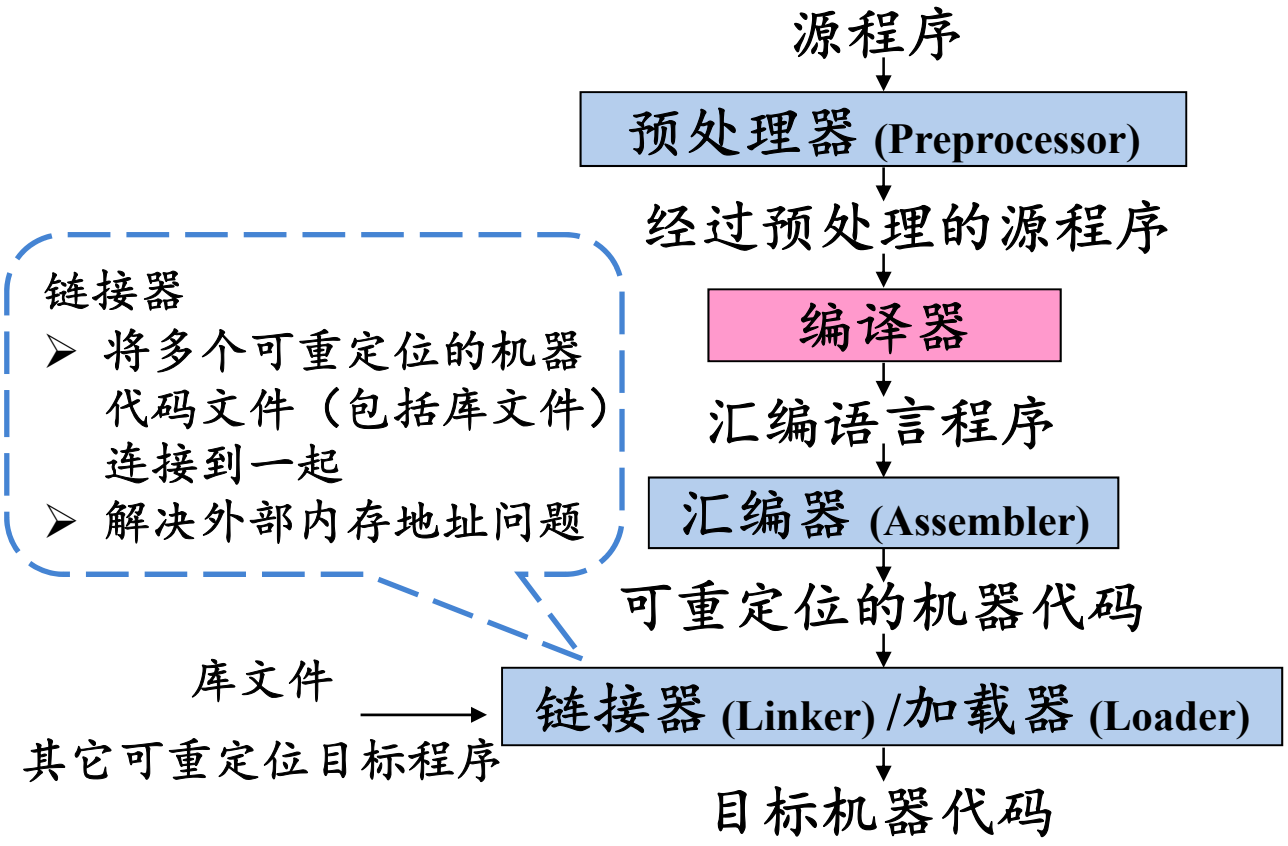
# 编译器在语言处理系统中的位置



# 编译器在语言处理系统中的位置



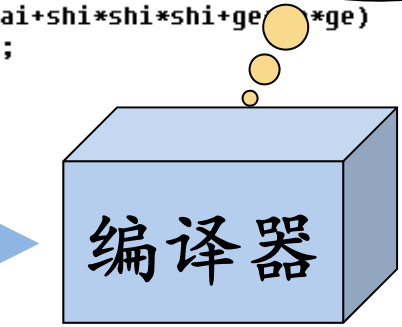
# 编译器在语言处理系统中的位置



# 编译系统的结构

```
#include<stdio.h>
int main()
{
    int a,b,ge,shi,bai,m,n,i,number;
    printf("请输入(输入完两个数按一次回车键): \n");
    scanf("%d %d",&a,&b);
    while(a!=0,b!=0)
        scanf("%d %d",&a,&b);
    if(a>=100,b<=999)
        if(a>b)
            m=b,n=a;
        else
            m=a,n=b;
        for(i=m;i<=n;i
        {
            bai=i/100;
            shi=(i%100)/10;
            ge=i%10;
            if(i==bai*bai*bai+shi*shi*shi+ge
            printf("%5d",i);
            number++;
            printf("\n");
        }
        if(number==0)
            printf("no\n");
}
```

机器是怎么翻译的？




高级语言程序

```
59      call sc
60      mov  al, [si - 1]
61      call sc
62      mov  al, [si]
63      call sc
64      mov  al, ' '
65      call sc
66      ret
67  storechr endp
68  ;清屏，无入口参数
69  clearcrt proc near
70      mov  ax, 0600h
71      mov  bh, 07h
72      mov  cx, 00h
73      mov  dx, 184fh
74      int  10h
75      ret
    rert endp
        行，入口参数，行数 no
        proc near, no:word
        cx, no
        x ==0
        ov  cx, 1
      .endif
      .repeat
        mov  ah, 02h
        mov  dl, cr
        int  21h
        mov  ah, 02h
        mov  dl, lf
        int  21h
      .untilcxz
      ret
92  nextlien endp
93          end
94
```

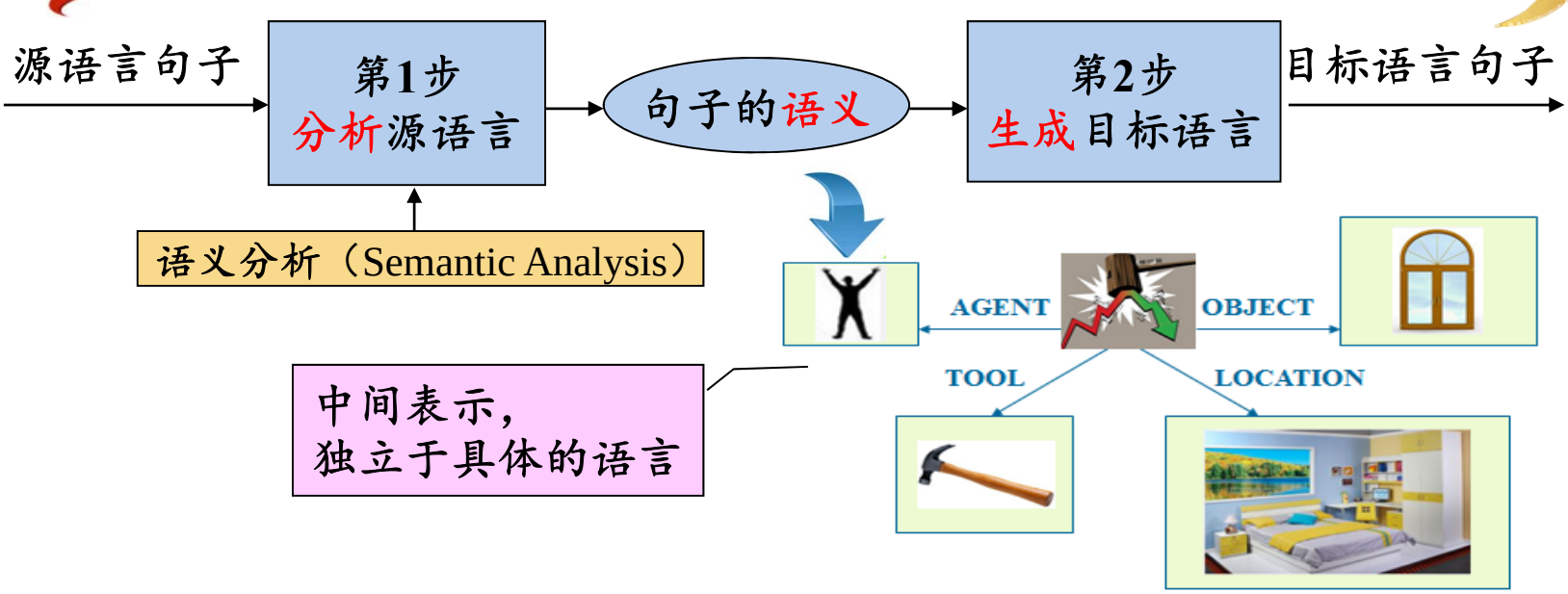
汇编语言程序/机器语言程序

# 人工英汉翻译的例子

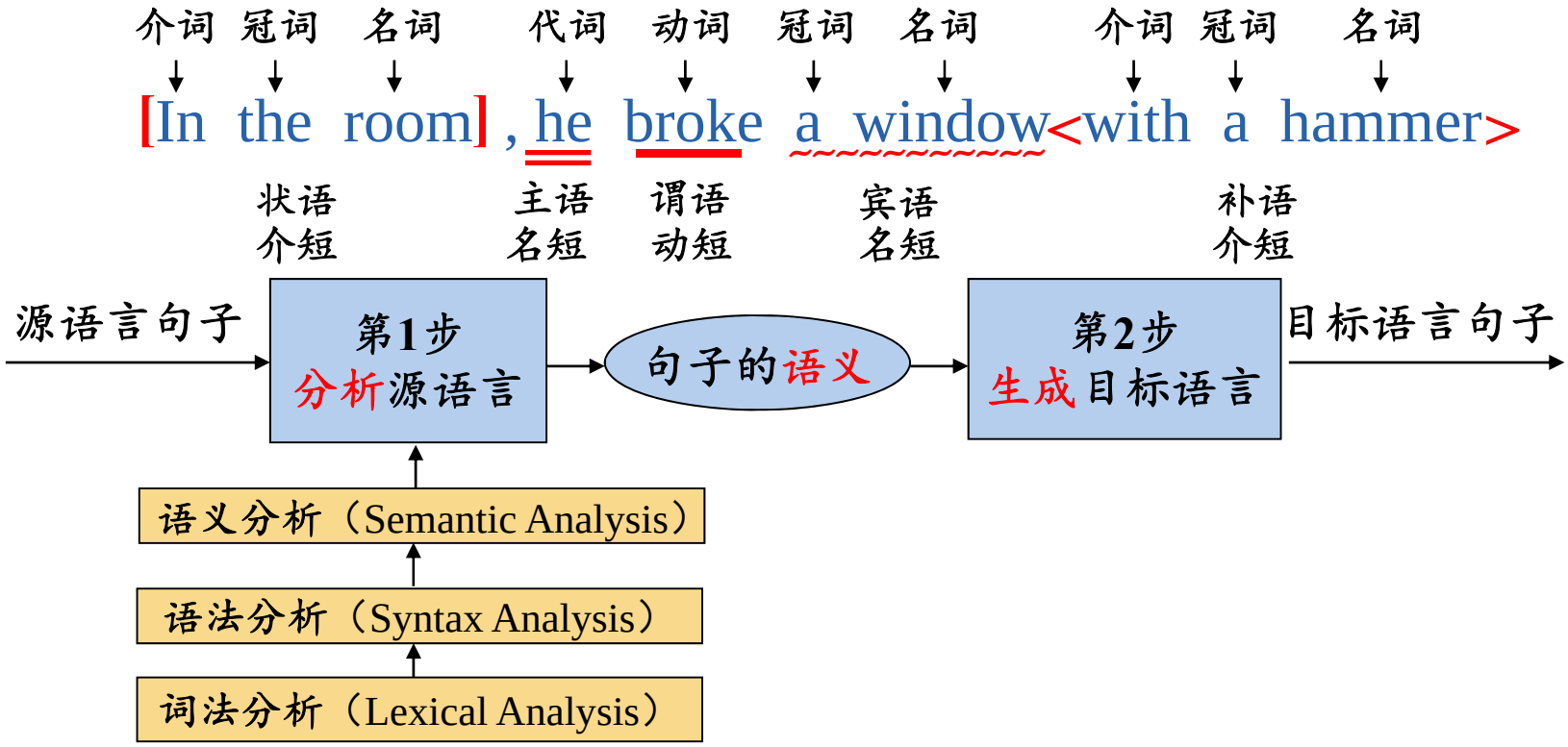
在房间里，他用锤子砸了一扇窗户。

[In the room] , he broke a window<with a hammer>

状语　主语　谓语　宾语　补语



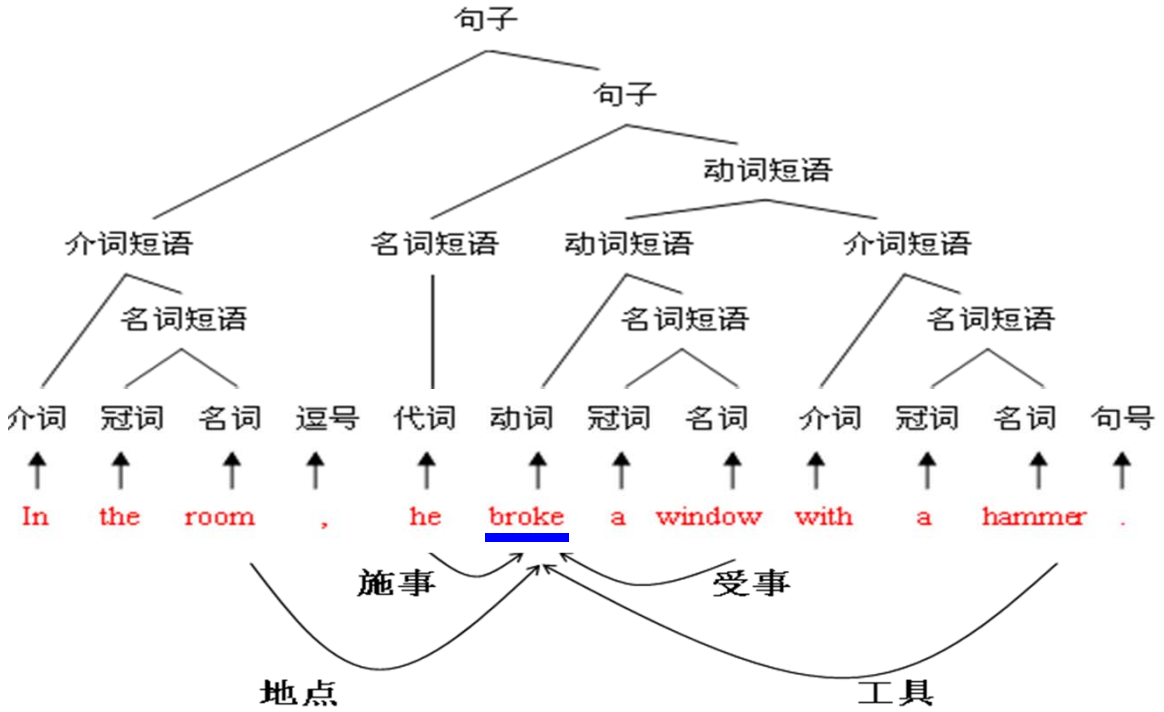
# 人工英汉翻译的例子


介词 冠词 名词 代词 动词 冠词 名词 介词 冠词 名词
[In the room] , he broke a window <with a hammer>
状语 介短
主语 名短
谓语 动短
宾语 名短
补语 介短
源语言句子
第1步 分析源语言
句子的语义
第2步 生成目标语言
目标语言句子
语义分析（Semantic Analysis）
语法分析（Syntax Analysis）
词法分析（Lexical Analysis）

# 人工英汉翻译的例子

In the room , he broke a window with a hammer

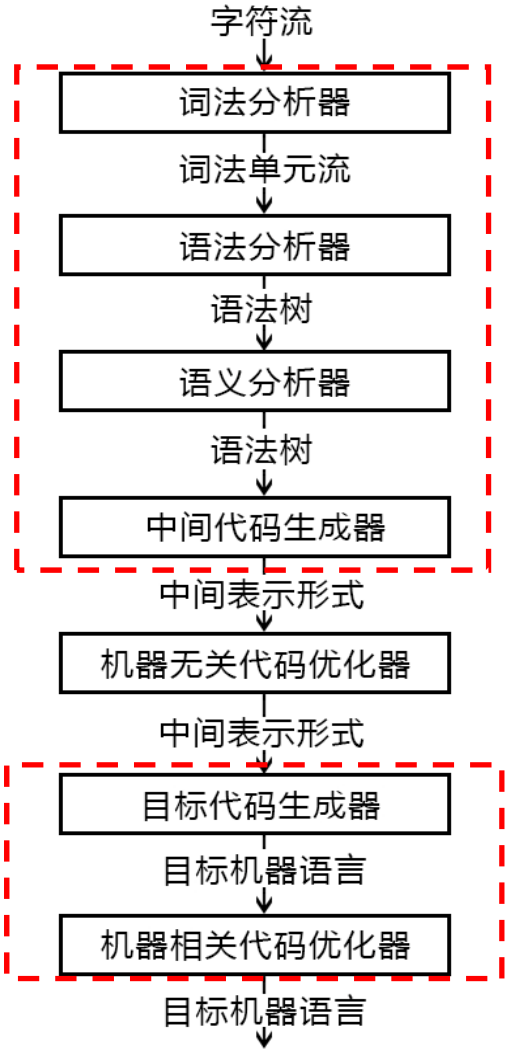
- 词法分析
- 语法分析
- 语义分析

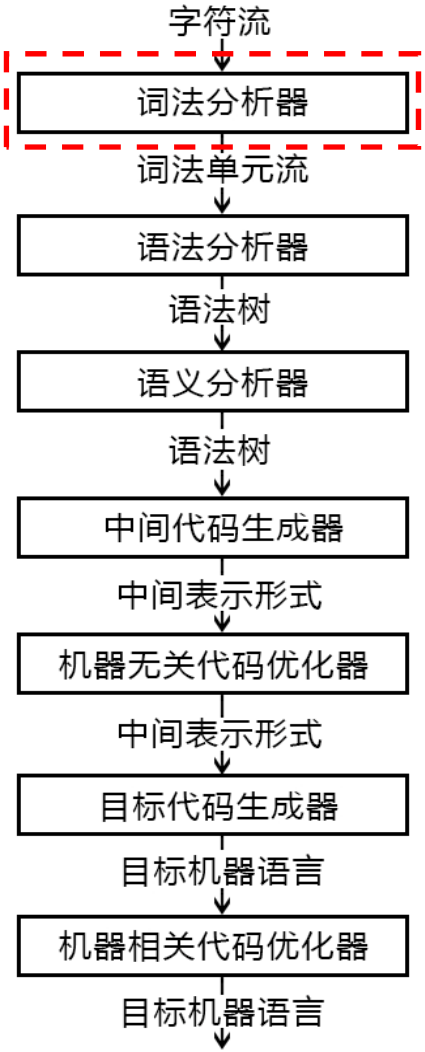
# 编译器的结构

分析部分/
前端(front end):
与源语言相关

综合部分/
后端(back end):
与目标语言相关



# 编译器的结构


字符流
词法分析器
词法单元流
语法分析器
语法树
语义分析器
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
目标代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

# 词法分析/扫描(*Scanning*)

- 词法分析的主要任务

从左向右逐行扫描源程序的字符，识别出各个单词，确定单词的类型。

将识别出的单词转换成统一的机内表示——词法单元(token)形式

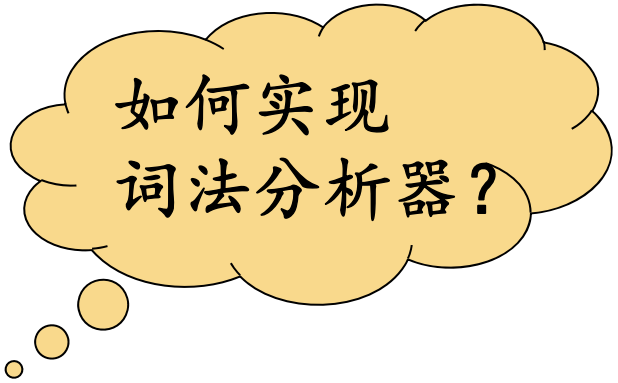
token：＜种别码，属性值＞

| | 单词类型 | 种别 | 种别码 |
|---|---|---|---|
| 1 | 关键字 | program、if、else、then、… | 一词一码 |
| 2 | 标识符 | 变量名、数组名、记录名、过程名、… | 多词一码 |
| 3 | 常量 | 整型、浮点型、字符型、布尔型、… | 一型一码 |
| 4 | 运算符 | 算术（+ - * / ++ --）<br>关系（> < == != >= <=）<br>逻辑（& \| ~） | 一词一码<br>或<br>一型一码 |
| 5 | 界限符 | ; ( ) = { } … | 一词一码 |

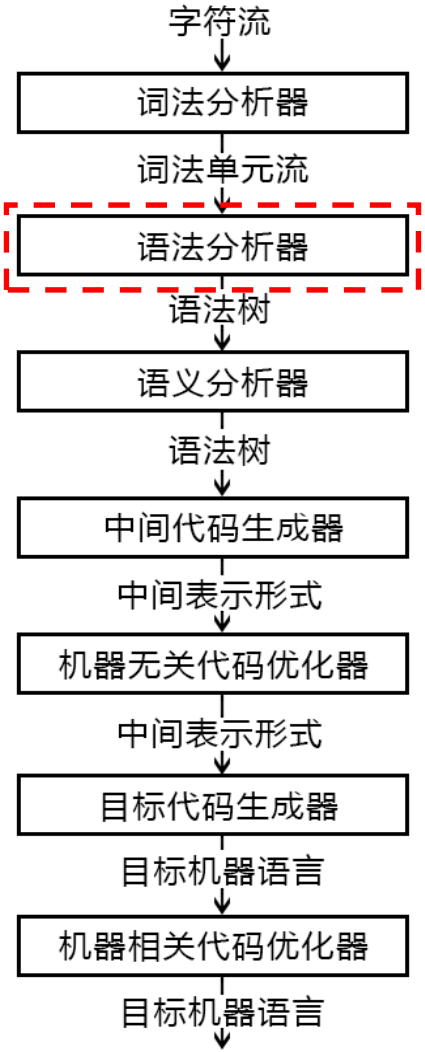
# 例：词法分析后得到的token序列

- 输入　while(value!=100){num++;}
- 输出

```
1   while    < WHILE ,    -     >
2     (      <   SLP   ,    -     >
3   value    <   IDN   ,  value   >
4     !=     <   NE    ,    -     >
5    100     < CONST ,   100    >
6     )      <   SRP   ,    -     >
7     {      <   LP    ,    -     >
8   num      <   IDN   ,   num    >
9     ++     <   INC   ,    -     >
10    ;      <  SEMI   ,    -     >
11    }      <   RP    ,    -     >
```

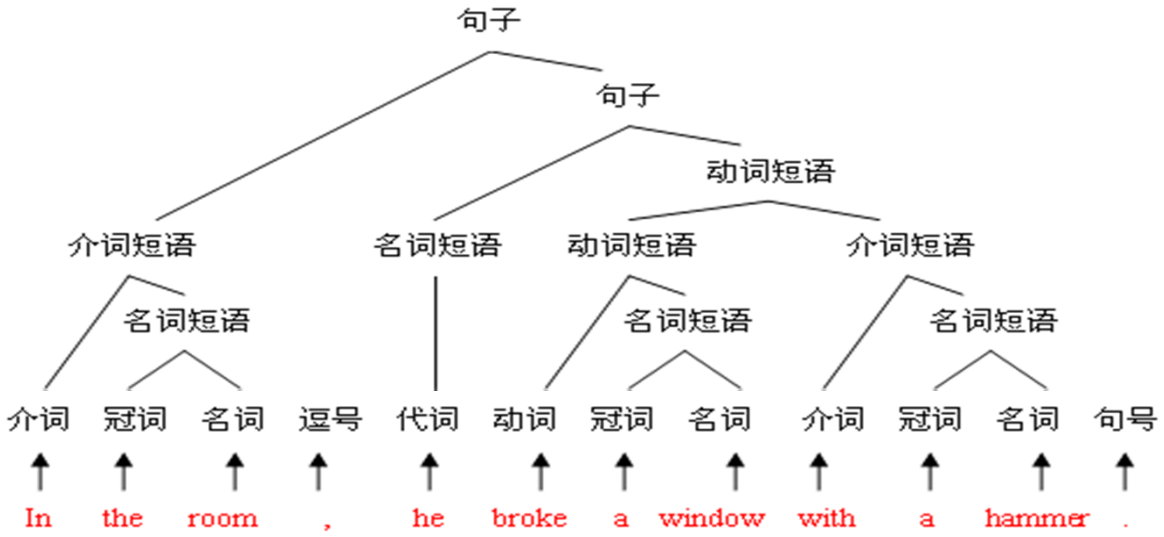


# 编译器的结构


字符流
词法分析器
词法单元流
语法分析器
语法树
语义分析器
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
目标代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

# 语法分析（*parsing*）

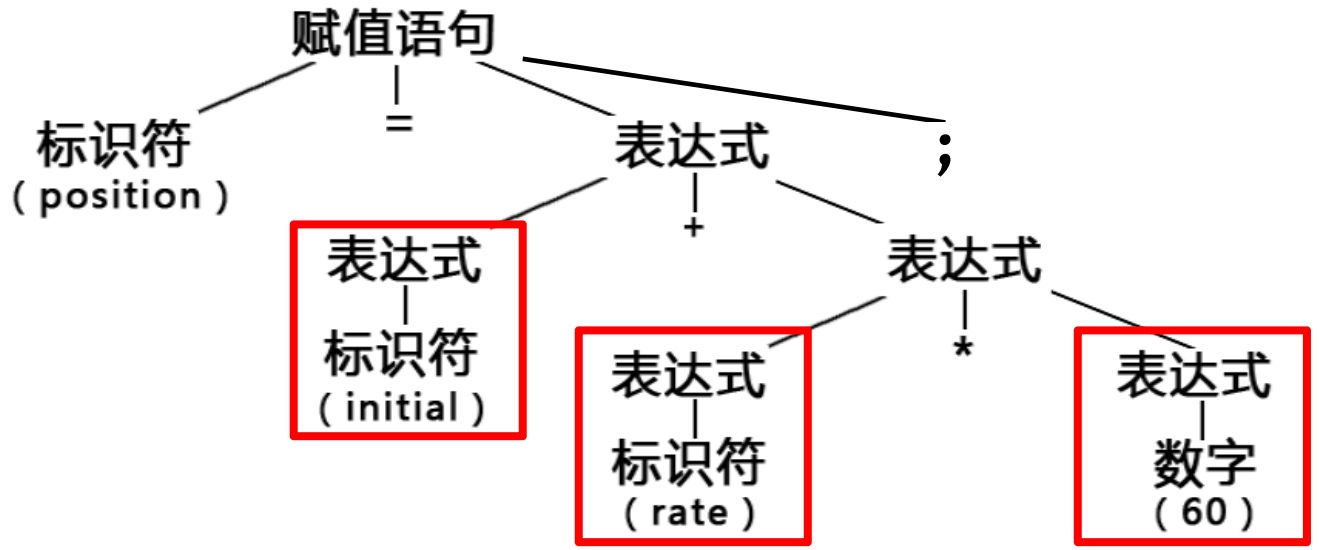
- 语法分析器(parser)从词法分析器输出的token序列中识别出各类短语，并构造语法分析树(parse tree)
  - 语法分析树描述了句子的语法结构



# 例1：赋值语句的分析树

**position = initial + rate * 60 ;**

**<id, position> <=> <id,initial> <+> <id, rate> <*> <num,60> <;>**



# 例2：变量声明语句的分析树

- 文法：

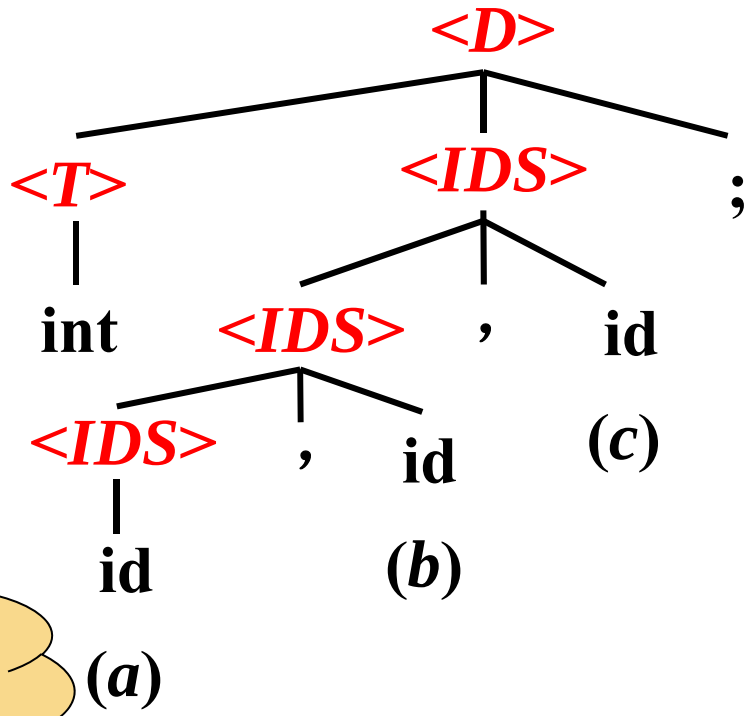
  <*D*> → <*T*> <*IDS*>;

  <*T*> → int | real | char | bool

  <*IDS*> → id | <*IDS*>, id

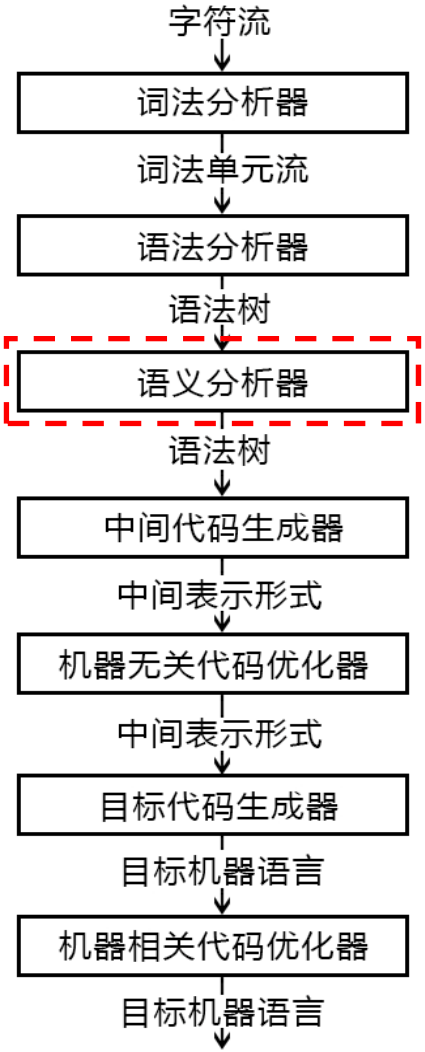
- 输入：

  int *a* , *b* , *c* ;




如何根据语法规则为
输入句子构造分析树？

# 编译器的结构


字符流
词法分析器
词法单元流
语法分析器
语法树
语义分析器
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
目标代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

# 语义分析的主要任务

- 收集标识符的属性信息
  - 种属 (Kind)
    - 简单变量、复合变量（数组、记录、…）、过程、…

# 语义分析的主要任务

- 收集标识符的属性信息
  - 种属 (Kind)
  - 类型 (Type)
    - 整型、实型、字符型、布尔型、指针型、…

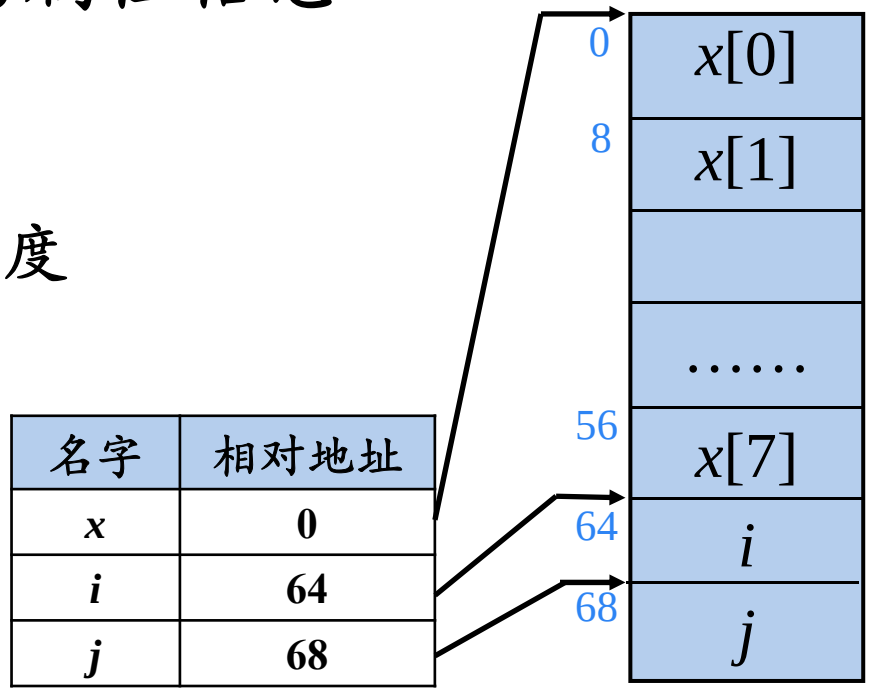
# 语义分析的主要任务

- 收集标识符的属性信息
  - 种属 (Kind)
  - 类型 (Type)
  - 存储位置、长度

例：

```
begin
    real x[8];
    integer i,j;
    ……
end
```

| 名字 | 相对地址 |
|---|---|
| *x* | 0 |
| *i* | 64 |
| *j* | 68 |



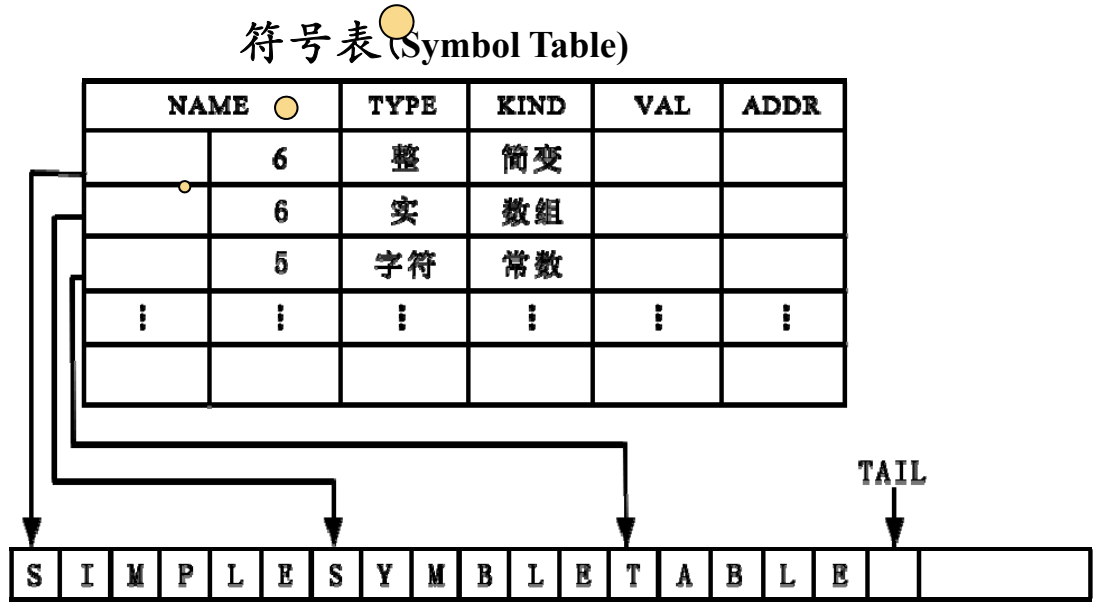
# 语义分析的主要任务

- 收集标识符的属性信息
  - 种属 (Kind)
  - 类型 (Type)
  - 存储位置、长度
  - 值
  - 作用域
  - 参数和返回值信息
    - 参数个数、参数类型、参数传递方式、返回值类型、…

# 语义分析的主要任务

符号表中为什么要设计字符串表这样一种数据结构？

- 收集标识符的属性信息
  - 种属 (Kind)
  - 类型 (Type)
  - 存储位置、长度
  - 值
  - 作用域
  - 参数和返回值信息

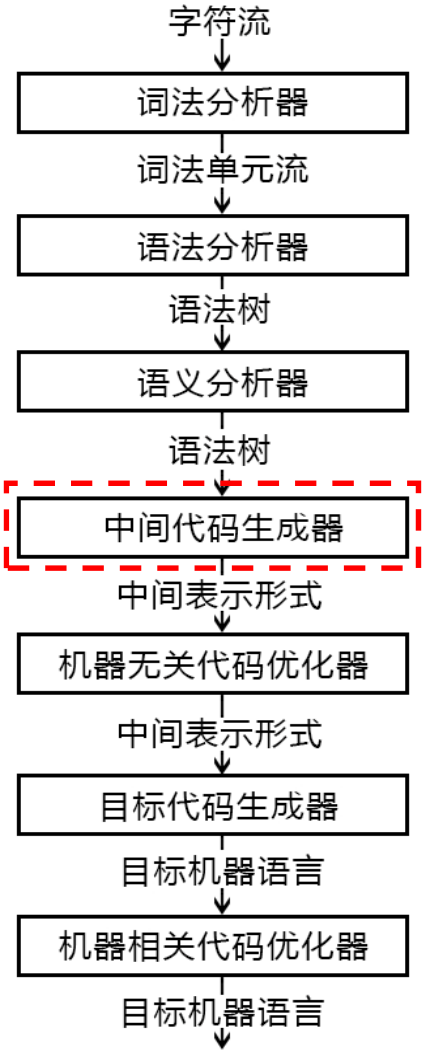



符号表是用于存放标识符的属性信息的数据结构

# 语义分析的主要任务

- 收集标识符的属性信息
- 语义检查
  - 变量或过程未经声明就使用
  - 变量或过程名重复声明
  - 运算分量类型不匹配
  - 操作符与操作数之间的类型不匹配
    - 数组下标不是整数
    - 对非数组变量使用数组访问操作符
    - 对非过程名使用过程调用操作符
    - 过程调用的参数类型或数目不匹配
    - 函数返回类型有误

# 编译器的结构


字符流
词法分析器
词法单元流
语法分析器
语法树
语义分析器
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
目标代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

# 常用的中间表示形式

- 三地址码 (Three-address Code)
  - 三地址码由类似于汇编语言的指令序列组成，每个指令最多有三个操作数(operand)
- 语法结构树/语法树 (Syntax Trees)

# 常用的三地址指令

| 序号 | 指令类型 | 指令形式 |
|---|---|---|
| 1 | 赋值指令 | $x = y$ op $z$<br>$x =$ op $y$ |
| 2 | 复制指令 | $x = y$ |
| 3 | 条件跳转 | if $x$ relop $y$ goto $n$ |
| 4 | 非条件跳转 | goto $n$ |
| 5 | 参数传递 | param $x$ |
| 6 | 过程调用 | call $p$, $n$ |
| 7 | 过程返回 | return $x$ |
| 8 | 数组引用 | $x = y[i]$ |
| 9 | 数组赋值 | $x[i] = y$ |
| 10 | 地址及<br>指针操作 | $x =\& y$<br>$x =* y$<br>$*x = y$ |

地址可以具有如下形式之一

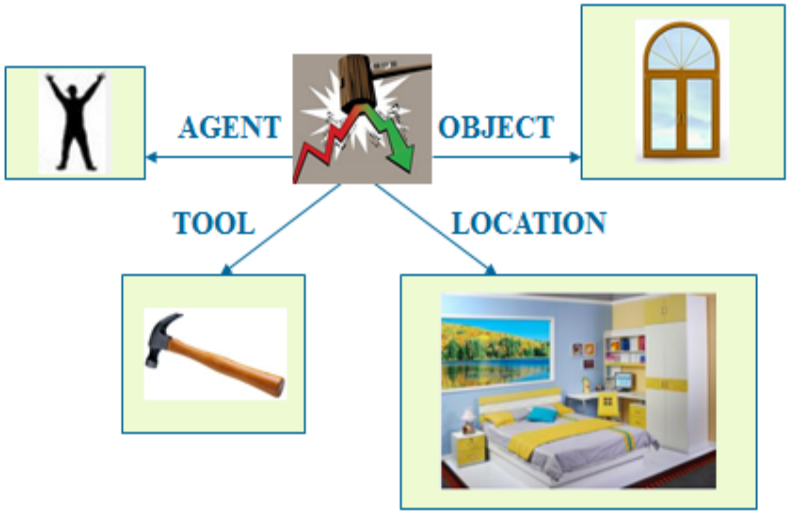
- 源程序中的名字 (*name*)
- 常量 (*constant*)
- 编译器生成的临时变量(*temporary*)

# 三地址指令的表示

- 四元式 (Quadruples)
  - (op, *y*, z, *x*)
- 三元式 (Triples)
- 间接三元式 (Indirect triples)

# 三地址指令的四元式表示

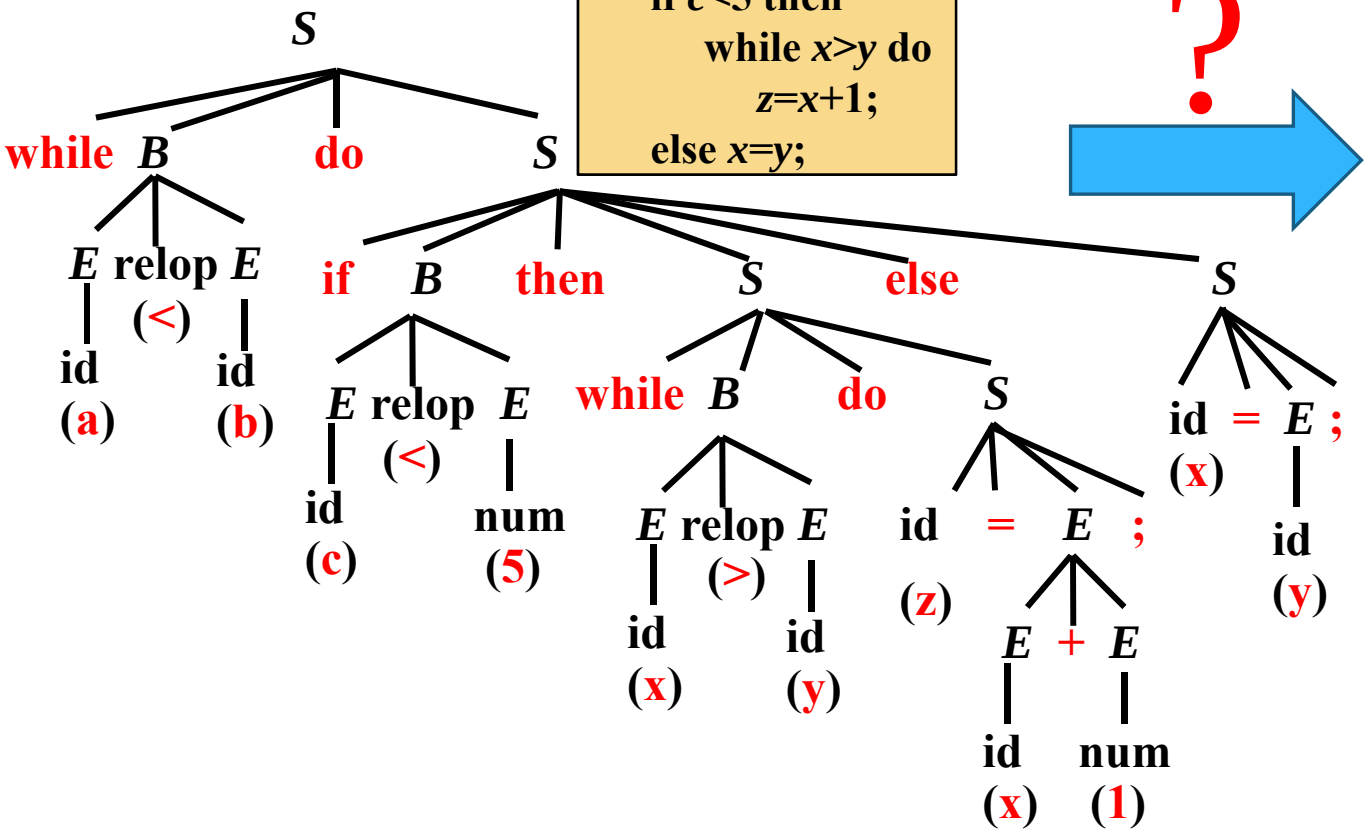
| | |
|---|---|
| ➢ $x = y$ op $z$ | ( op , $y$ , $z$ , $x$ ) |
| ➢ $x =$ op $y$ | ( op , $y$ , _ , $x$ ) |
| ➢ $x = y$ | ( = , $y$ , _ , $x$ ) |
| ➢ if $x$ relop $y$ goto $n$ | ( relop , $x$ , $y$ , $n$ ) |
| ➢ goto $n$ | ( goto , _ , _ , $n$ ) |
| ➢ param $x$ | (param, _ , _ , $x$ ) |
| call $p$, $n$ | ( call , $p$ , $n$ , _ ) |
| return $x$ | (return, _ , _ , $x$ ) |
| ➢ $x = y[i]$ | ( =[] , $y$ , $i$ , $x$ ) |
| $x[i] = y$ | ( []= , $y$ , $x$ , $i$ ) |
| ➢ $x = \&y$ | ( & , $y$ , _ , $x$ ) |
| $x = *y$ | ( =* , $y$ , _ , $x$ ) |
| $*x = y$ | ( *= , $y$ , _ , $x$ ) |




三地址指令序列唯一确定了运算完成的顺序

# 中间代码生成的例子

```
while a<b do
    if c<5 then
        while x>y do
            z=x+1;
    else x=y;
```

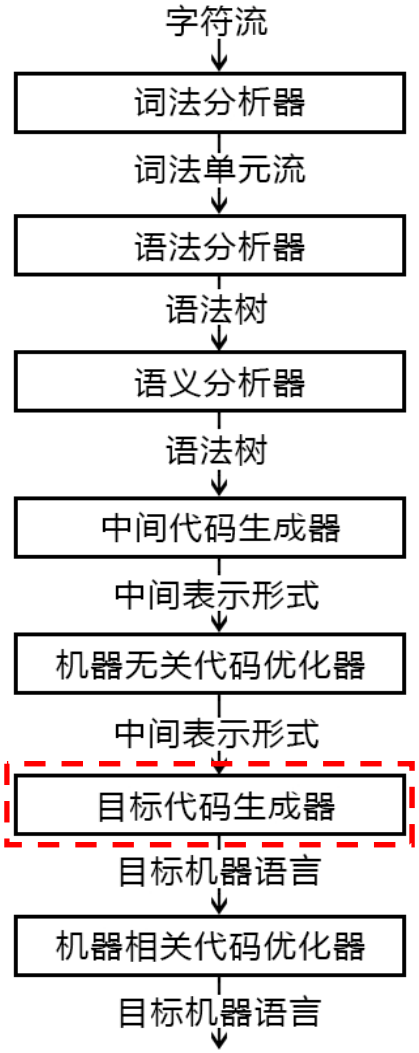



?

```
100: ( j<, a ,b , 102 )
101: ( j  , - , - , 112 )
102: ( j<, c , 5 , 104 )
103: ( j  , - , - , 110 )
104: ( j>, x , y , 106 )
105: ( j  , - , - , 100 )
106: ( + , x , 1 ,  t1 )
107: ( = , t1 , - ,  z  )
108: ( j  , - , - , 104 )
109: ( j  , - , - , 100 )
110: ( = , y , - ,  x  )
111: ( j  , - , - , 100 )
112:
```

# 编译器的结构

- 目标代码生成以源程序的中间表示形式作为输入，并把它映射到目标语言
- 目标代码生成的一个重要任务是为程序中使用的变量合理分配寄存器



# 编译器的结构

- 代码优化
  - 为改进代码所进行的等价程序变换，使其运行得更快一些、占用空间更少一些，或者二者兼顾