编译原理

计算机语言的发展

- 机器语言
  - 二进制指令
- 抱怨，就会提出新问题——太难记

汇编语言

- 机器语言的助记——低级语言

```
-u 100
0B33:0100 B82000        MOV     AX,0020
0B33:0103 BB3000        MOV     BX,0030
0B33:0106 01D8          ADD     AX,BX
0B33:0108 CD09          INT     09
0B33:010A CD20          INT     20
```

- 抱怨，就会提出新问题

高级语言

- 以一个更类似于数学定义或自然语言的简洁形式来编写程序的操作——如何设计一种高级语言？
- 它应与任何机器都无关；
- 方便翻译成二进制代码——如何翻译？

如何设计一种高级语言？

- 1.用数学的表达及逻辑表示

  x=1　y=x+1

- 2.一个正常过程的描述应该包含什么？

  我叫小明帮我买药品.

- 3.语言包括哪些要素？
  - 找出自然语言这个模型进行分析总结
    - 词法
    - 语法

高级语言如何翻译为二进制？

找相似模型:外文翻译

1.翻译的方法:

即席翻译　书面翻译

2.翻译的过程

高级语言的发展过程

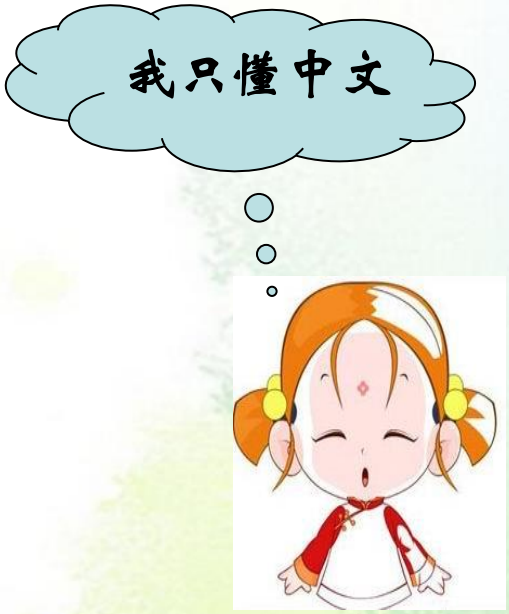
- 在1954年至1957年期间，IBM的John Backus开发了FORTRAN语言编译器。

- Noam Chomsky开始对自然语言结构的研究——用类似数学公式的式子，来建立生成语法体系，并以此描写自然语言

- 两个理论相结合形成一套完善的有关高级程序设计语言的编译原理。

编译？

编写和翻译外文，也称翻译者。



外文翻译过程

以英文翻译为例：

I've got rhythm,
I've got music,
I've got my gal,
Who could ask for anything more?

- I've got rhythm 我有韵律
- I've got music 我有音乐
- I've got my gal 我有女友
- Who could ask for anything more?
  谁还会问我要什么呢?

外文翻译过程的总结

- 抽象观点：

  任何一本外文资料——由字母、标点符号（包括空格和其它符号）并按相应语法规则所组成的字符串。

- 即 符号 → 单词 → 句子 → 书

外文翻译过程的总结

因此，想要翻译外文，必须具备以下能力：

- （1）能认识外语的字母及标点符号；
- （2）能识别出文中的各个单词；
- （3）会查字典；
- （4）懂得此种外语的语法；
- （5）具有目标语言的修辞能力。

分析与综合

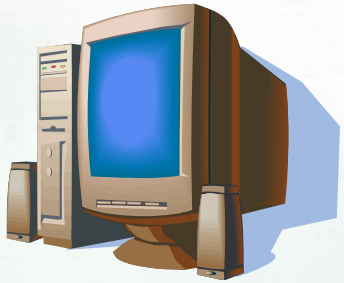
- **分析：**第一个符号开始，依次阅读原文中的各个符号，逐个识别出原文中的各个单词，然后根据语法规则进行语法分析。
- 即分析原文中如何由单词组成短语和句子，以及句子的种类特点等。
  - 在识别单词和进行语法分析的过程中，还要不时查阅字典
  - 做语法正确性的检查，进行相应的语义分析，并做一些必要的信息记录工作等。
- **综合：**根据上述分析所得到的信息，拟定译稿，进行修辞加工，最后写出译文。

	翻译外文
分析	阅读原文 识别单词 分析句子
综合	修辞加工 写出译文

查字典
检查错
填表

我只懂人说的语言

我只懂二进制语言



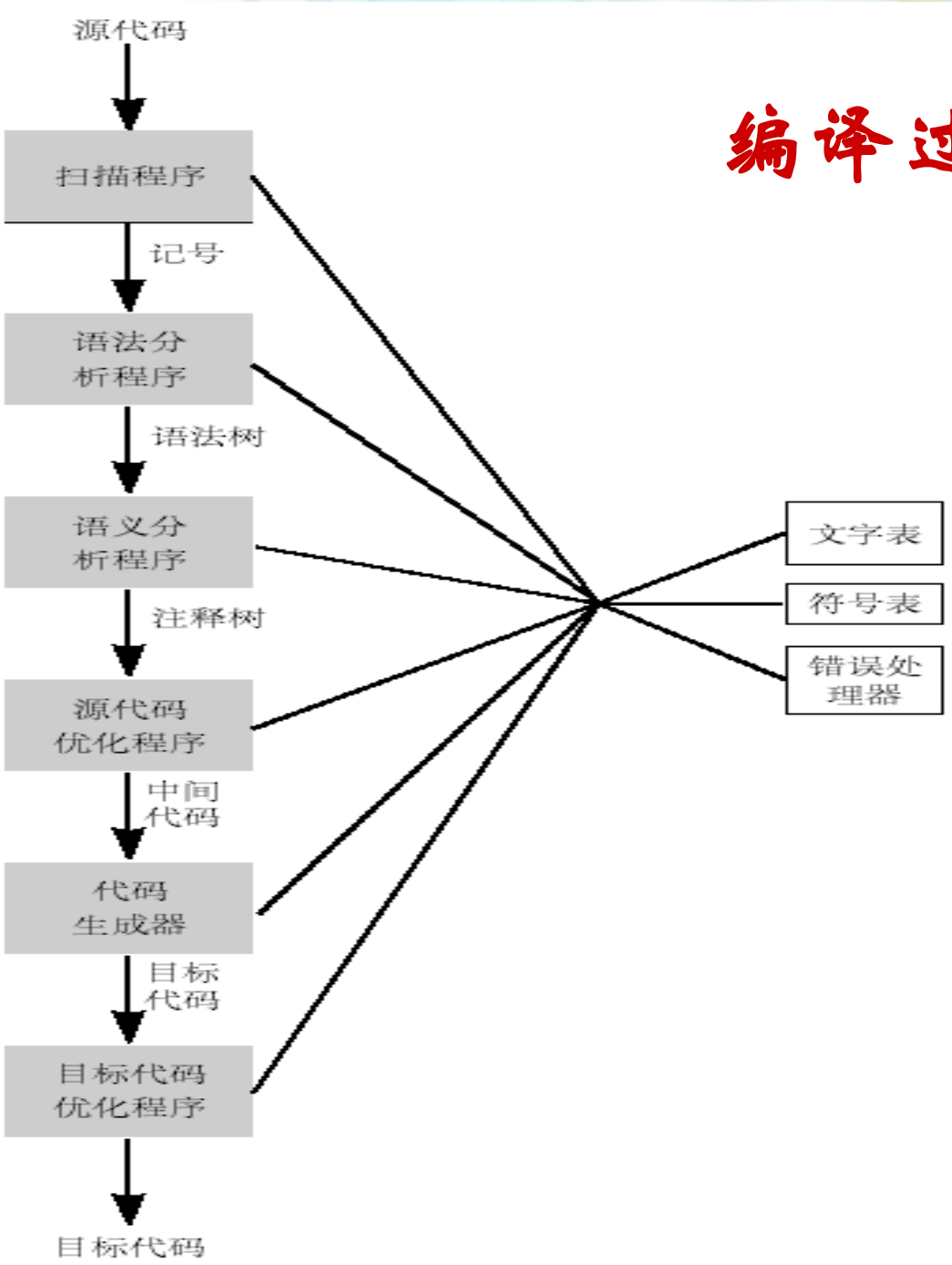
翻译软件包

如何翻译？

	翻译外文	编译源程序
分析	阅读原文 识别单词 分析句子	输入并扫描源程序 词法分析 语法分析
综合	修辞加工 写出译文	修饰优化 目标代码生成

即：先分析源程序，然后再综合为目标程序

编译过程


源代码
扫描程序
记号
语法分析程序
语法树
语义分析程序
注释树
源代码优化程序
中间代码
代码生成器
目标代码
目标代码优化程序
目标代码
文字表
符号表
错误处理器

(1) 扫描程序（scanner）-词法分析程序

源程序的字符 → 单词、 记号（token）

如何实现?

记号（token）类似于自然语言如英语的单词及标点符号

扫描＝拼写

思考

- Good judgment comes from experience and all of that comes from bad judgment.

- 单词分析

单词分析

- Good judgment comes from experience and all of that comes from bad judgment.

- Good 形容词
- judgment 名词
- comes 动词
- from 介词
- experience 名词
- and 连词
- ……

(1) 扫描程序（scanner）

例如

a[index] = 4 + 2

记号：

a 标识符

[左括号

index 标识符

] 右括号

= 赋值

4 数字

+ 加号

2 数字

如何实现

主控代码

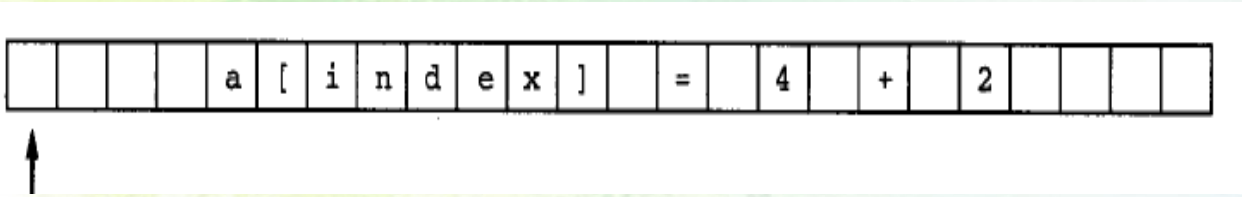
```
switch(ch)
{
    case 字母:
    case 数字:
    case '[':
    case '=':
    case '] ':
    case '+':
    …
}
```

如何组织输入？

输入→程序→输出

- 1.逐个符号读入，逐个符号分析
- 2.逐行读入，再逐个符号分析
- 3.整个程序读入，再逐个符号分析
- 例如：

a[index] = 4 + 2　　逐行读入



尝试性实验：预编译系统的实现
----打造具有个人风格的C++语言（词法分析）

运行效果样本

具有个人风格的C++源程序：
Test-in.cpp

```
#include<iostream.h>
(* This is a test file *)
main()
begin
  integer i;
  read>>i;
  i:=i+1
  if (i<>3) write<< “ok” ;
end
```

改写后的C++源程序：
Test-out.cpp

```
#include<iostream.h>
/* This is a test file */
main()
{
  int i;
  cin>>i;
  i=i+1;
  if (i!=3) cout<< “ok” ;
}
```

实验1.C++源程序的压缩器与解压器

压缩效果分析

C++源程序：

Test.cpp

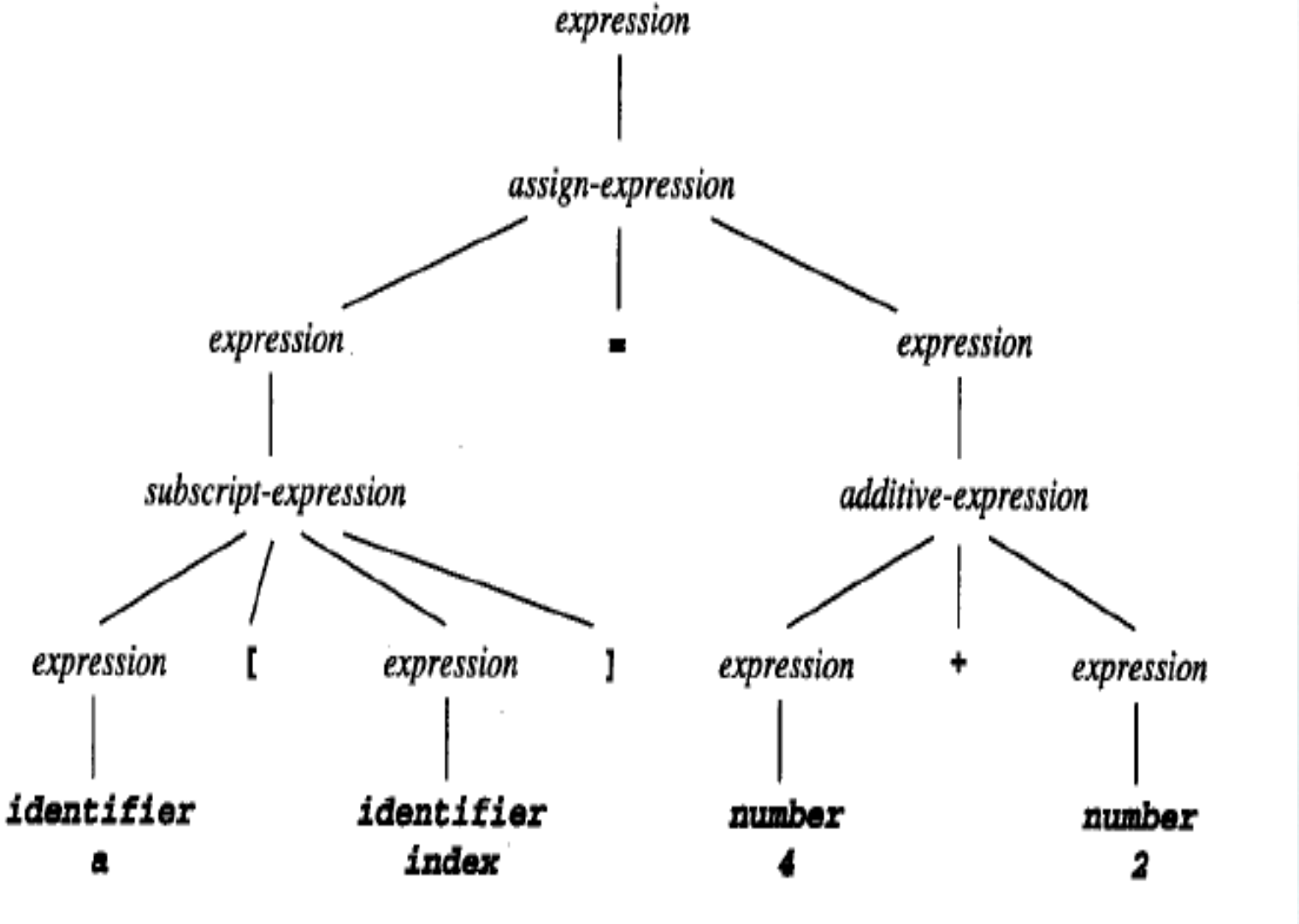
```
#include<iostream.h>
main( )
{
   int i;
   cin>>i;
   if (i>3) cout<< “ok”;
}
```

(2) 语法分析程序（parser）

- 语法分析定义了程序的记号元素及其关系

- 记号 → 句子 → 用分析树或语法树表示

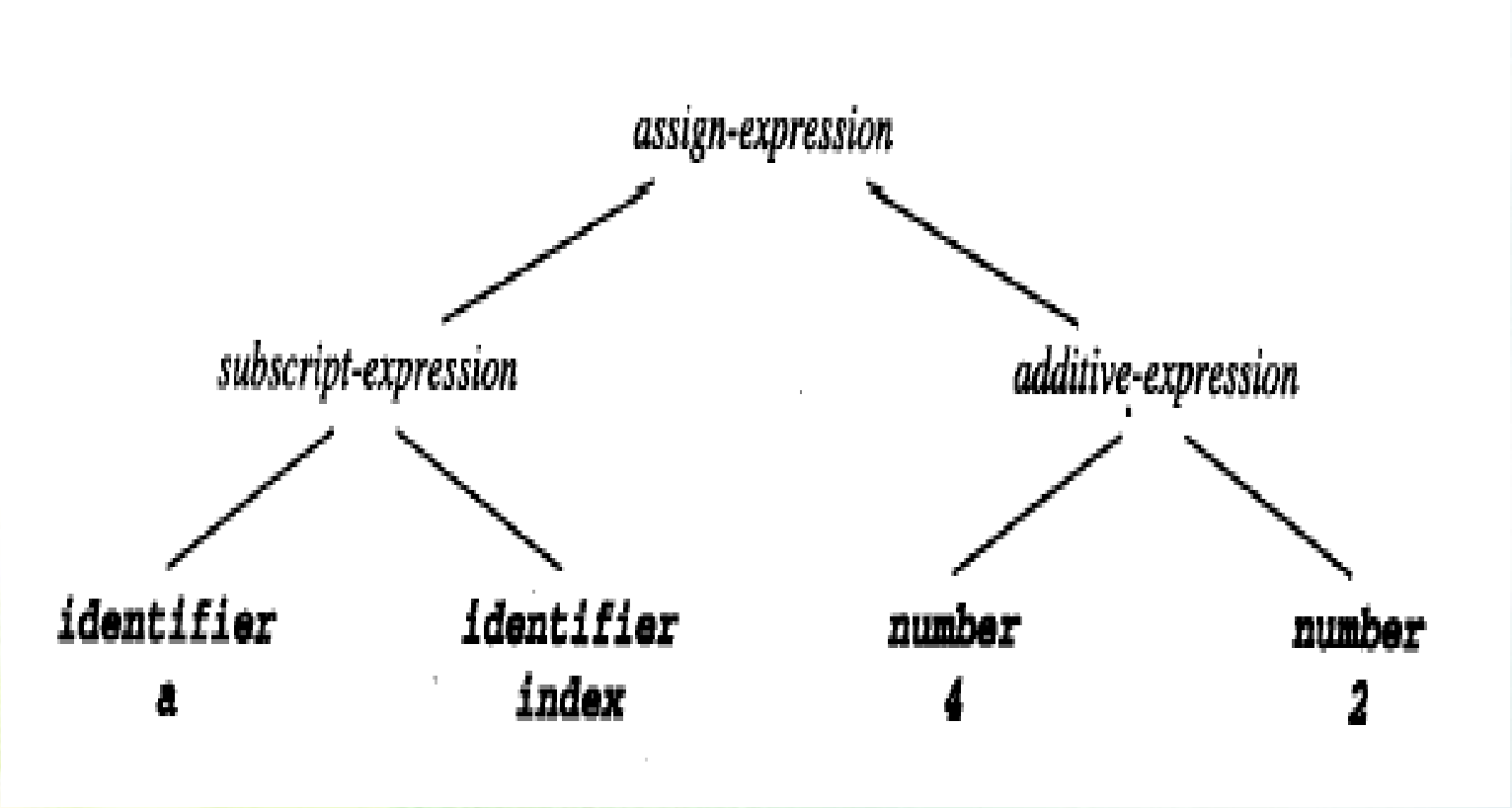
a [index] = 4 + 2


expression
assign-expression
expression
=
expression
subscript-expression
additive-expression
expression
[
expression
]
expression
+
expression
identifier
a
identifier
index
number
4
number
2

- 分析树的缺点：
  - 过于庞大，耗费内存

- 解决方法：

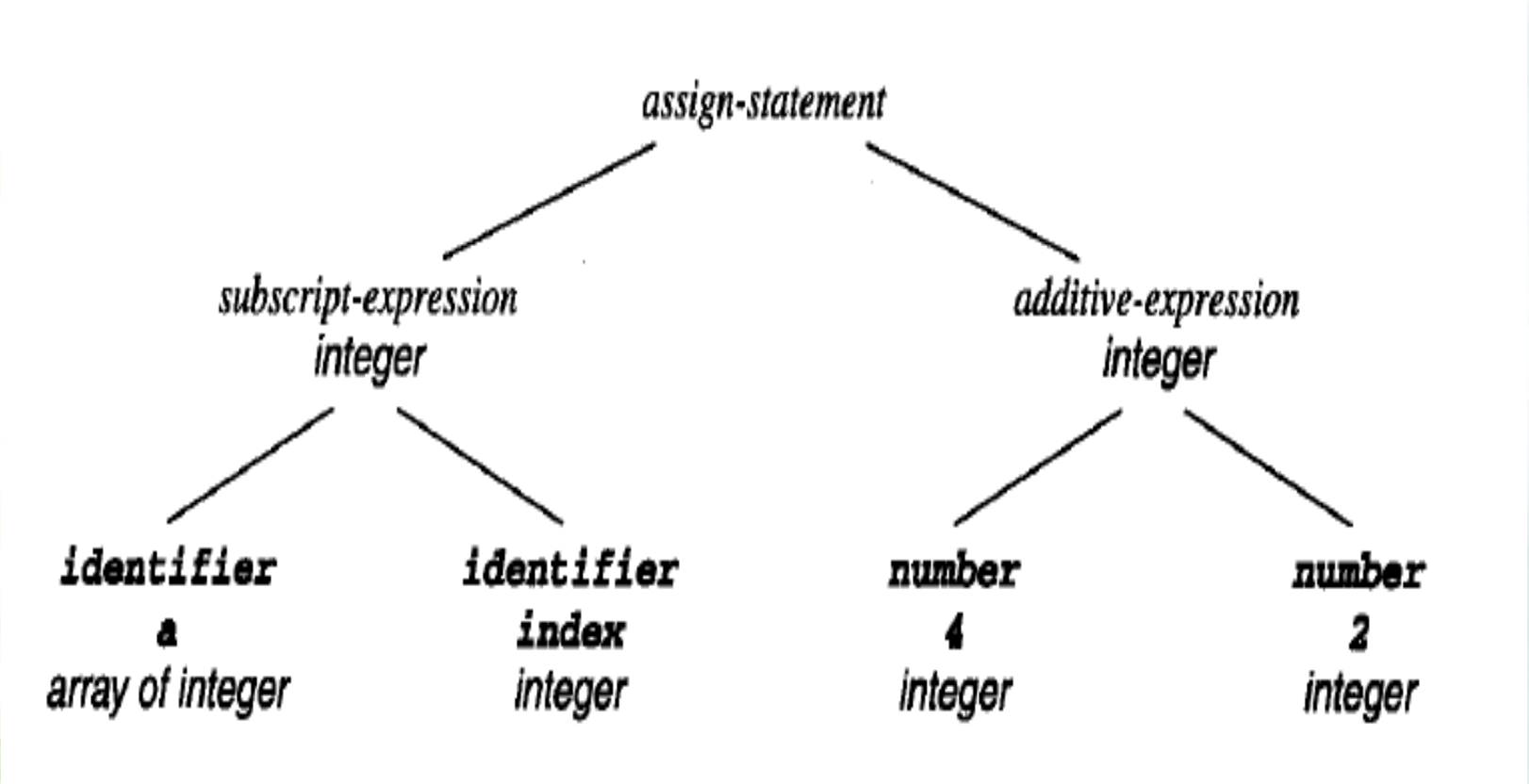
- 压缩→只保留有用信息→抽象的语法树（abstract syntax tree）


assign-expression
subscript-expression
additive-expression
identifier
a
identifier
index
number
4
number
2

(3) 语义分析程序（semantic analyzer）

- 程序的语义→ 理解其“意思”
- 一般的程序设计语言的典型静态语义包括**声明和类型检查**。
- 解决方法：
- 在语法树上添加**属性**（attribute）——如数据类型）。

a [index] = 4 + 2


assign-statement
subscript-expression
integer
additive-expression
integer
identifier
a
array of integer
identifier
index
integer
number
4
integer
number
2
integer

(4) 源代码优化程序（source code optimizer）

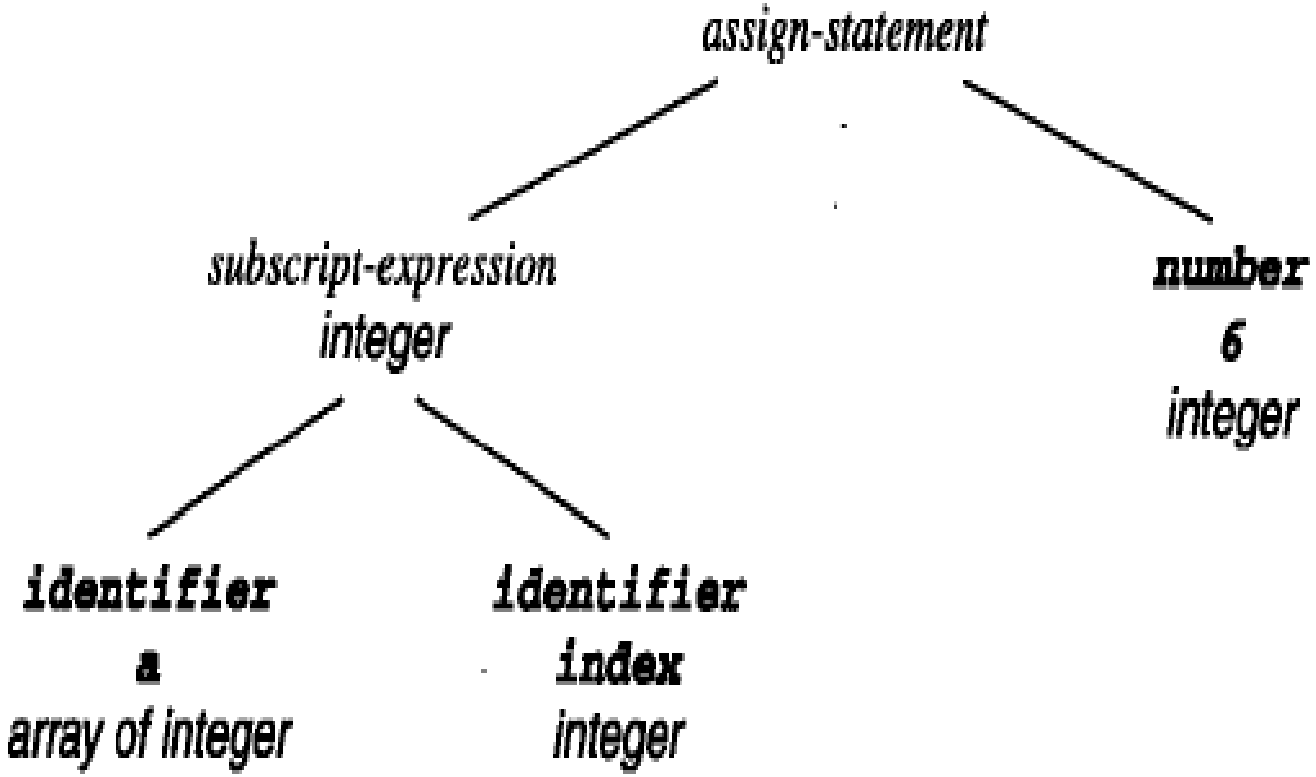
即对代码进行改进或优化步骤。

- 与机器无关的优化

如：表达式4＋2

→可以先计算结果6

→常量合并(constant folding)。


assign-statement
subscript-expression
integer
number
6
integer
identifier
a
array of integer
identifier
index
integer

如何更好地进行优化？

→多遍扫描

→中间代码

→中间代码的存储结构

中间代码

- 中间代码一直是指一种位于源代码和目标代码之间的代码表示形式。
- 三元组、四元组、树型、伪代码、逆波兰

(5) 代码生成器（code generator）

- 代码生成器得到中间代码(IR),并生成目标机器代码。

- 尽管大多数编译器直接生成目标代码，但是为了便于理解，本书用汇编语言来编写目标代码。

- 例如,下面是上述表达式的样本代码序列(在假设的汇编语言中):

```
MOV  R0, index        ;; value of index -> R0
MUL  R0, 2            ;; double value in R0
MOV  R1, &a           ;; address of a -> R1
ADD  R1, R0           ;; add R0 to R1
MOV  *R1, 6           ;; constant 6 -> address in R1
```

(6) 目标代码优化程序（target code optimizer）

- 与机器有关的优化，利用机器指令特征进行优化
- 使用了这两种优化后，目标代码就变成：

```
MOV   R0, index          ;; value of index -> R0
SHL    R0                ;; double value in R0
MOV   &a[R0], 6          ;; constant 6 -> address a + R0
```

- 优化的目的：运行效率和节省内存空间

基本概念

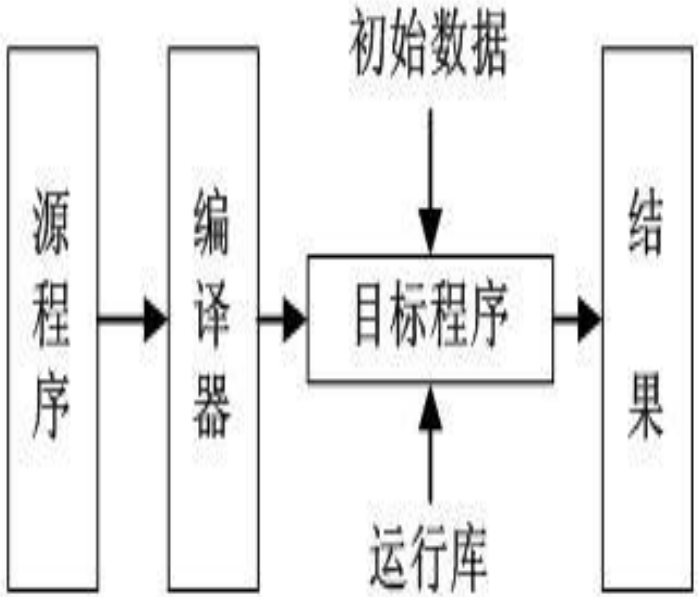
- 汇编程序(assembler)：
  - 从汇编语言到机器语言的翻译程序称为汇编程序，它的源语言和目标语言分别是相应的汇编语言和机器语言。

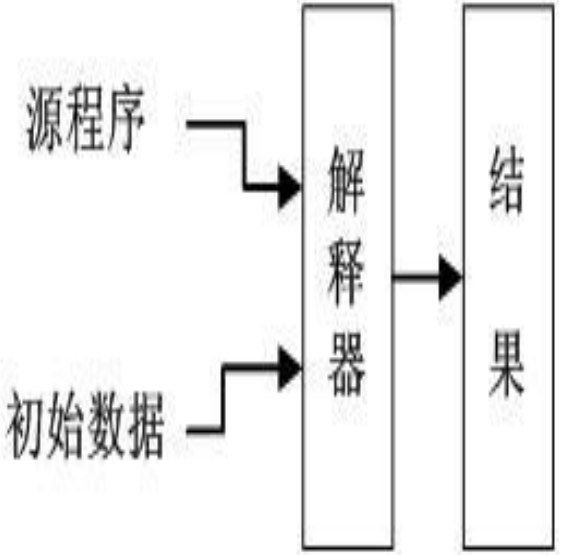
- 编译程序(compiler)或解释程序(interpreter)：
  - 如果一个翻译程序的源语言是某种高级语言，其目标语言是相应于某一计算机的汇编语言或机器语言，则称这种翻译程序为编译程序或解释程序。

翻译方式

编译程序：书面翻译

解释程序：即席翻译


源程序
编译器
初始数据
目标程序
运行库
结果


源程序
初始数据
解释器
结果

高级语言的设计

- 1.单词分类（保留字、标识符、专用符号等）
- 2.接近于数学定义及数学运算方式
- 3.控制结构简单：顺序、分支、循环
- 4.应该要能做输入及输出
- 5.为了提高程序的可读性：可以加注释

样本语言

- Tiny语言

- Tiny样本语言：求输入值阶乘TINY语言程序

```
{ Sample program
   in TINY language -
   computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1
   until x = 0;
   write fact { output factorial of x }
end
```

TINY 样本语言

(1) 由分号分隔开的语句序列。

(2) 无过程（函数）也无数据类型的声明（只有整型变量）。

(3) 只有两个控制语句：if语句和repeat语句，这两个控制语句本身也可包含语句序列。

If语句有一个可选的else部分且必须由关键字end结束。

（4）read语句和write语句完成输入/输出。

（5）用花括号表示注释，但注释不能嵌套。

（6）只有布尔（比较）表达式和整型算术表达式。

- 比较表达式只能使用＜与＝比较运算符号。
- 比较表达式可能用在控制语句中（即没有布尔型变量和布尔型的赋值）
- 算术表达式可以包括整型常数、变量、参数以及4个整型算符+、－、*、/。
- 为了区分比较运算符＝，赋值运算符号用 :=

TINY 样本语言的单词分类

1. 关键词：

If then else end repeat until read write 不区分大小写

2. 专用符号： + - * / < = { } ; :=

3. identifier和*number*，则定义：

identifier： 字母开头，后面可跟若干个字母，且不区分大小写

number： 数字符号开头，后面可跟若干个数字

也可以抽象地表示为：

identifier = letter letter*

number = digit digit*

letter = a|..|z|A|..|Z

digit = 0|..|9

4. 空格由空白、换行符和制表符组成。

5. 用表示{ }注释,注释可以放在任何空白出现的位置(即注释不能放在标记内)上，且可以超过一行。注释不能嵌套。

TINY 样本语言的语法规则

program → *stmt-sequence*

stmt-sequence → *stmt-sequence* **;** *statement* | *statement*

statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

if-stmt → **if** *exp* **then** *stmt-sequence* **end**
| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**

repeat-stmt → **repeat** *stmt-sequence* **until** *exp*

assign-stmt → **identifier** **:=** *exp*

read-stmt → **read identifier**

write-stmt → **write** *exp*

exp → *simple-exp comparison-op simple-exp* | *simple-exp*

comparison-op → **<** | **=**

simple-exp → *simple-exp addop term* | *term*

addop → **+** | **-**

term → *term mulop factor* | *factor*

mulop → ***** | **/**

factor → **(** *exp* **)** | **number** | **identifier**

Tiny语言的缺陷

- 1.不方便利用计算机内存：增加数据类型
- 2.运算不丰富，如比较运算只有<和=，应该是每种数据类型都要有自己的运算
- 3.控制结构单一，表达不方便，不灵活：
  - 增加多分支结构的表达,如switch
  - 增加循环结构的表达,如while，for
- 4.方便利用已写的代码：
  - 引入函数
  - 引入面向对象的方法

TINY编译器

TINY编译器包括以下的C文件，它的头文件放在左边，它的代码文件放在右边：

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c
symtab.h	symtab.c
analyze.h	analyze.c
code.h	code.c
cgen.h	cgen.c

- 编译器有4遍：
- 第1遍由构造语法树的扫描程序和分析程序组成；
- 第2遍和第3遍执行语义分析，其中第2遍构造符号表
- 第3遍完成类型检查；
- 最后一遍是代码生成器。
- 在main.c中驱动这些遍的代码十分简单。当忽略了标记和编辑肘，它的中心代码如下（请参看附录B中的第69、77、79和94行）：

```
syntaxTree = parse( );
buildSymtab (syntaxTree);
typeCheck (syntaxTree);
codeGen (syntaxTree, codefile);
```

• 条件编译

标志	设置效果	编译所需文件（附加）
NO_PARSE	创建只扫描的编译器	globals.h, main.c, util.h ,util.c, scan.h, scan.c
NO_ANALYZE	创建只分析和扫描的编译器	parse.h, parse.c
NO_CODE	创建执行语义分析，但不生成代码的编译器	symtab.h, symtab.c, analyze.h, analyze.c

- 编译命令：

tiny　sample. tny

生成结果文件：sample.tm

TM机

- 执行程序命令：

```
tm sample.tm
```

- 进入TM运行模拟环境。