



课程简介

编译原理

华保健

bjhua@ustc.edu.cn



什么是编译器？

- 计算设备包括个人计算机、大型机、嵌入式系统、智能设备等
- 核心的问题都是软件的构造
 - 而目前绝大部分软件都由高级语言书写
 - 成百种高级语言
- 这些语言写成的程序是如何运行在计算机上的？
 - 编译器



示例

```
int main ()
{
    printf ("hello, world\n");
    return 0;
}
```

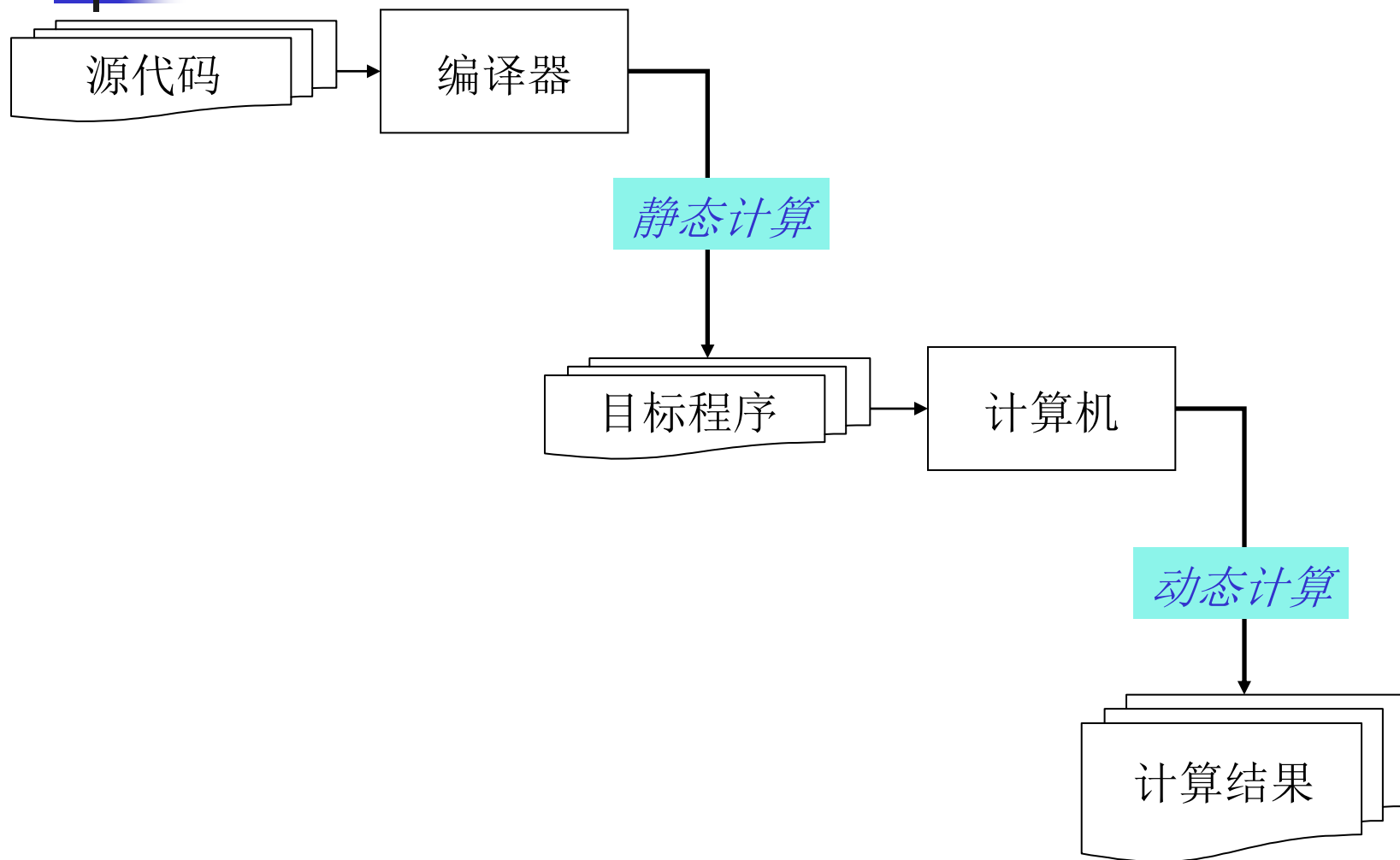
```
.text
str:
    .string "hello,..."
    .globl main
main:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    $str
    call     printf
    leave
    ret
```



什么是编译器？

- 编译器是一个程序
- 核心功能是把源代码翻译成目标代码
 - 源代码：
 - C/C++, Java, C#, html, SQL, ...
 - 目标代码：
 - x86, IA64, ARM, MIPS, ...

编译器的核心功能



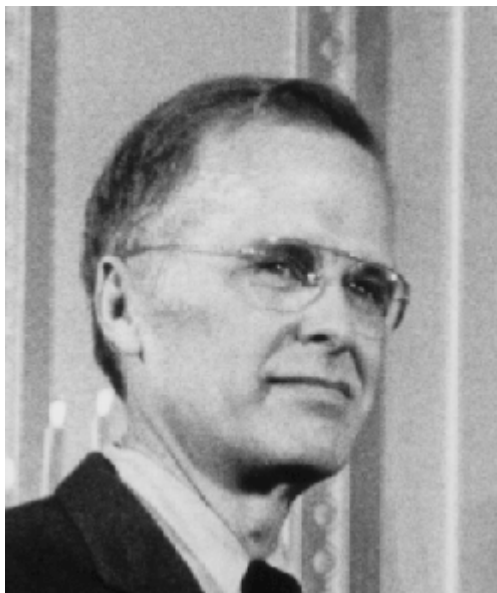


编译器和解释器

- 解释器也是一类处理程序的程序

编译器简史

- 计算机科学史上出现的第一个编译器是 Fortran 语言的编译器
 - 1954-1957年，John Backus





编译器简史

- Fortran编译器的成功给计算机科学发展产生巨大影响
 - 理论上：算法、数据结构、形式语言与自动机等
 - 实践上：软件工程、体系结构等
 - 编译器架构



为什么学习编译原理？

- 编译原理集中体现了计算机科学的很多核心思想
 - 算法，数据结构，软件工程等
- 编译器是其他领域的重要研究基础
- 编译器本身就是非常重要的研究领域
 - 新的语言设计
 - 大型软件的构造和维护



如何学好编译原理？

- 编译器设计是理论和实践高度结合的一个领域，在学习处理好二者关系
 - 理论：深入学习掌握各种算法和数据结构
 - 实践：切实提高将理论应用于解决实际问题的能力



编译器结构

编译原理

华保健

bjhua@ustc.edu.cn



编译器的高层结构

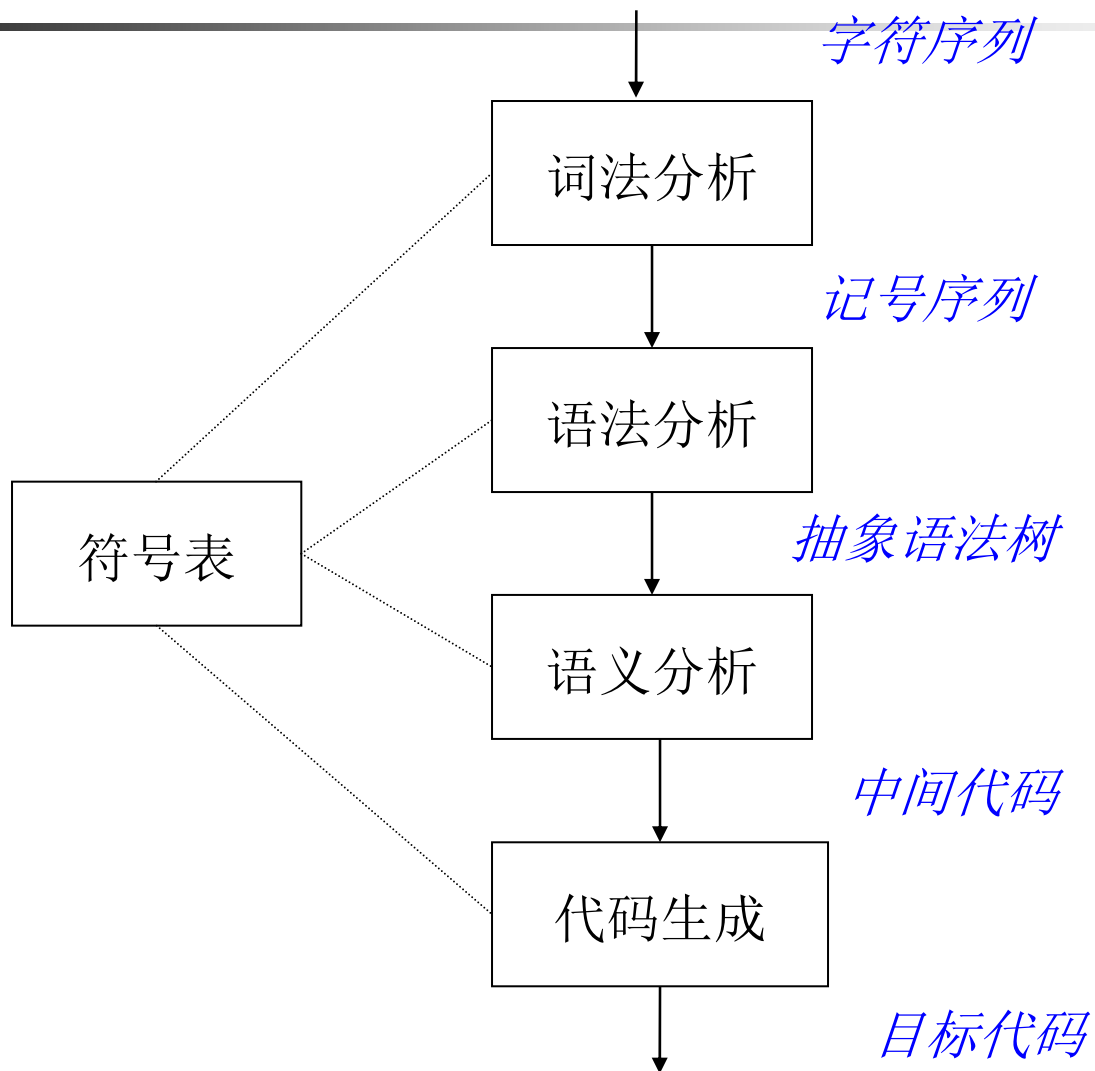
- 编译器具有非常模块化的高层结构



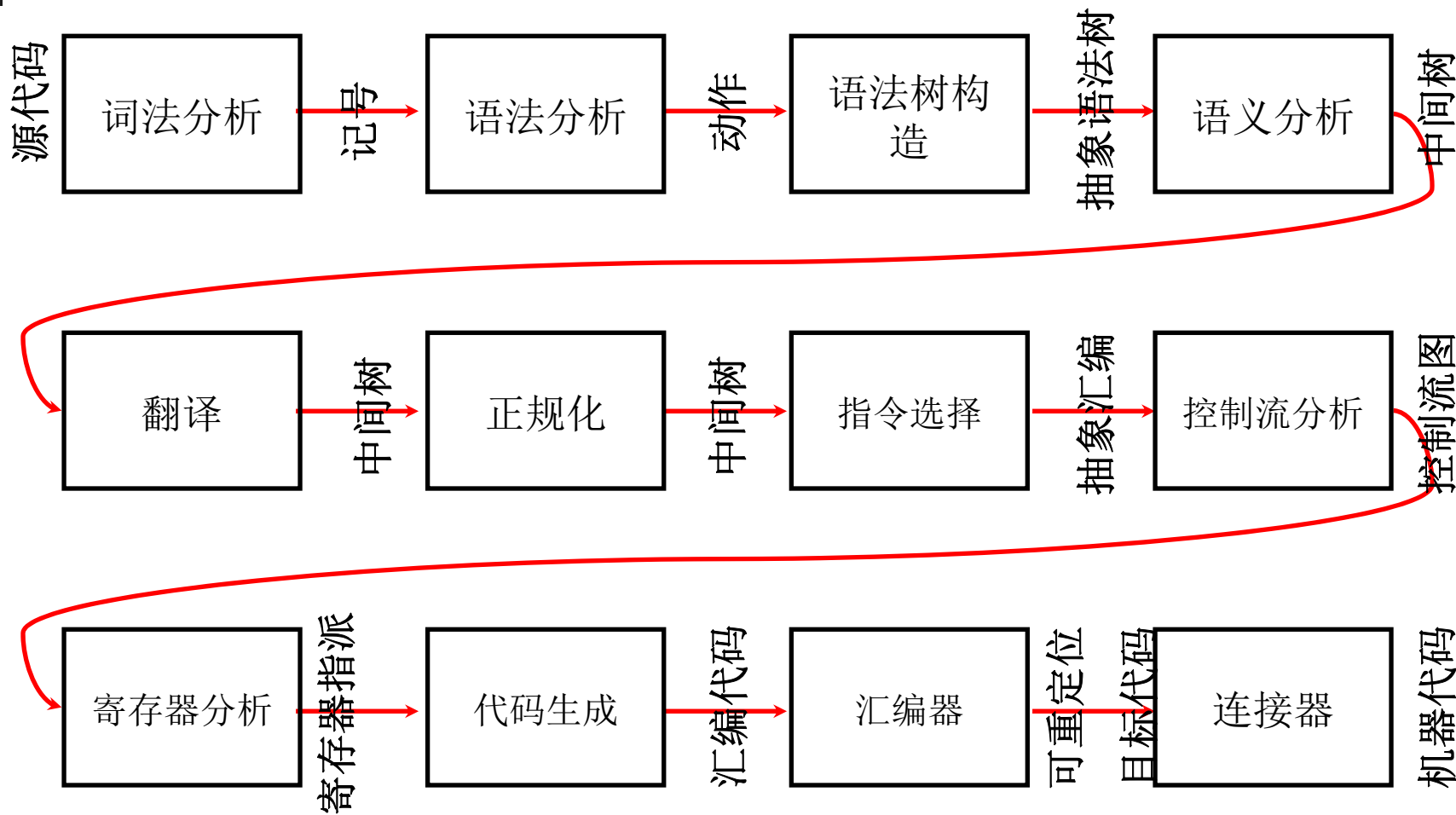
抽象的多个阶段（phase）

- 编译器可看成多个阶段构成的“流水线”结构

一种没有优化的编译器结构



一种更复杂的编译器结构





小结

- 编译器由多个阶段组成，每个阶段都要处理不同的问题
 - 使用不同的理论、数据结构和算法
- 因此，编译器设计中的重要问题是如何合理的划分组织各个阶段
 - 接口清晰
 - 编译器容易实现、维护



编译器例子

编译原理

华保健

bjhua@ustc.edu.cn



简单的编译器实例

- 源语言：加法表达式语言Sum
 - 两种语法形式：
 - 整型数字：n
 - 加法：e1+e2
- 目标机器：栈式计算机Stack
 - 一个操作数栈
 - 两条指令：
 - 压栈指令：push n
 - 加法指令：add



源语言Sum

- 两种语法形式：
 - 整型数字： n
 - 加法： $e_1 + e_2$
- 例子：



栈式计算机Stack

- 一个操作数栈
- 两条指令：
 - 压栈指令：push n
 - 加法指令：add
- 例子：



编译器的阶段

- 任务：编译程序 $1+2+3$ 到栈式计算机
- 阶段一：词法语法分析



编译器实现

- 任务：编译程序 $1+2+3$ 到栈式计算机
- 阶段二：语法树构建



编译器实现

- 任务：编译程序 $1+2+3$ 到栈式计算机
- 阶段三：代码生成



小结

- 编译器构造和具体的编译器目标相关，目前的结构：



思考题

- 任务：编译程序 $1+2+3$ 到栈式计算机
- 阶段四：代码优化（常量折叠优化）
 - 例如： $1+2 ==> 3$



词法分析---简介

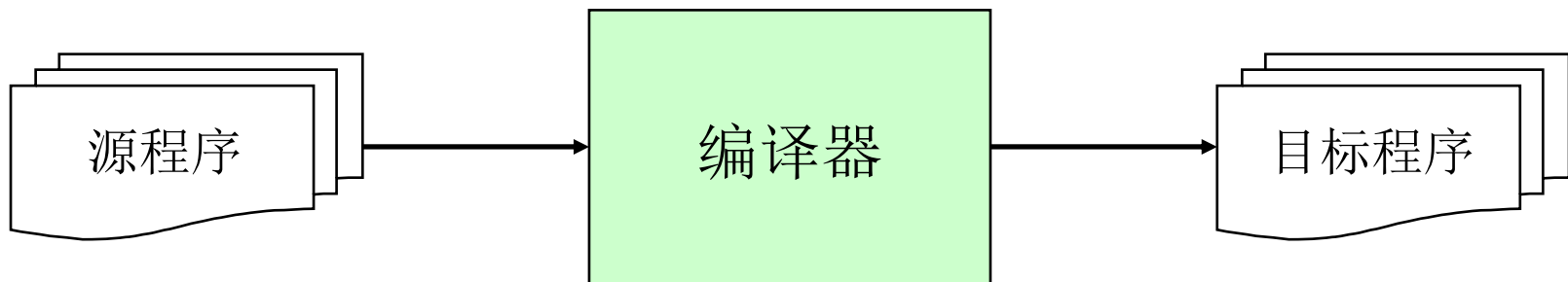
编译原理

华保健

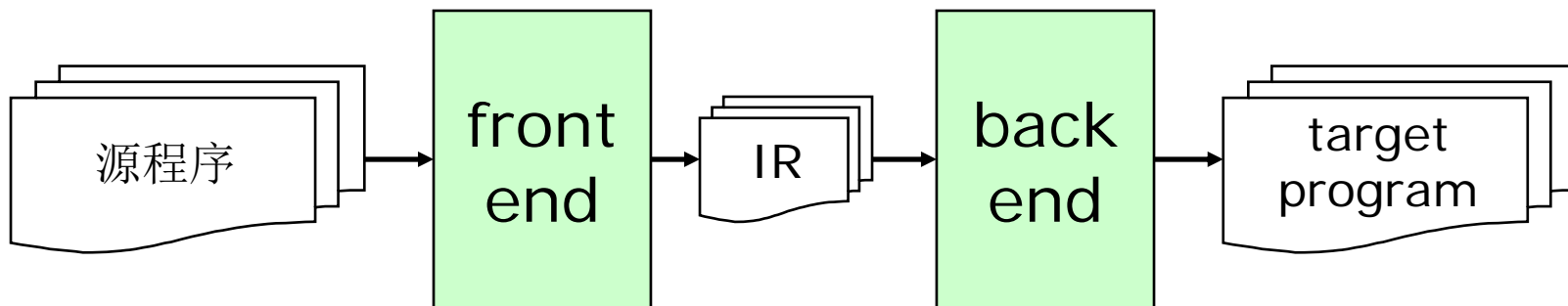
bjhua@ustc.edu.cn



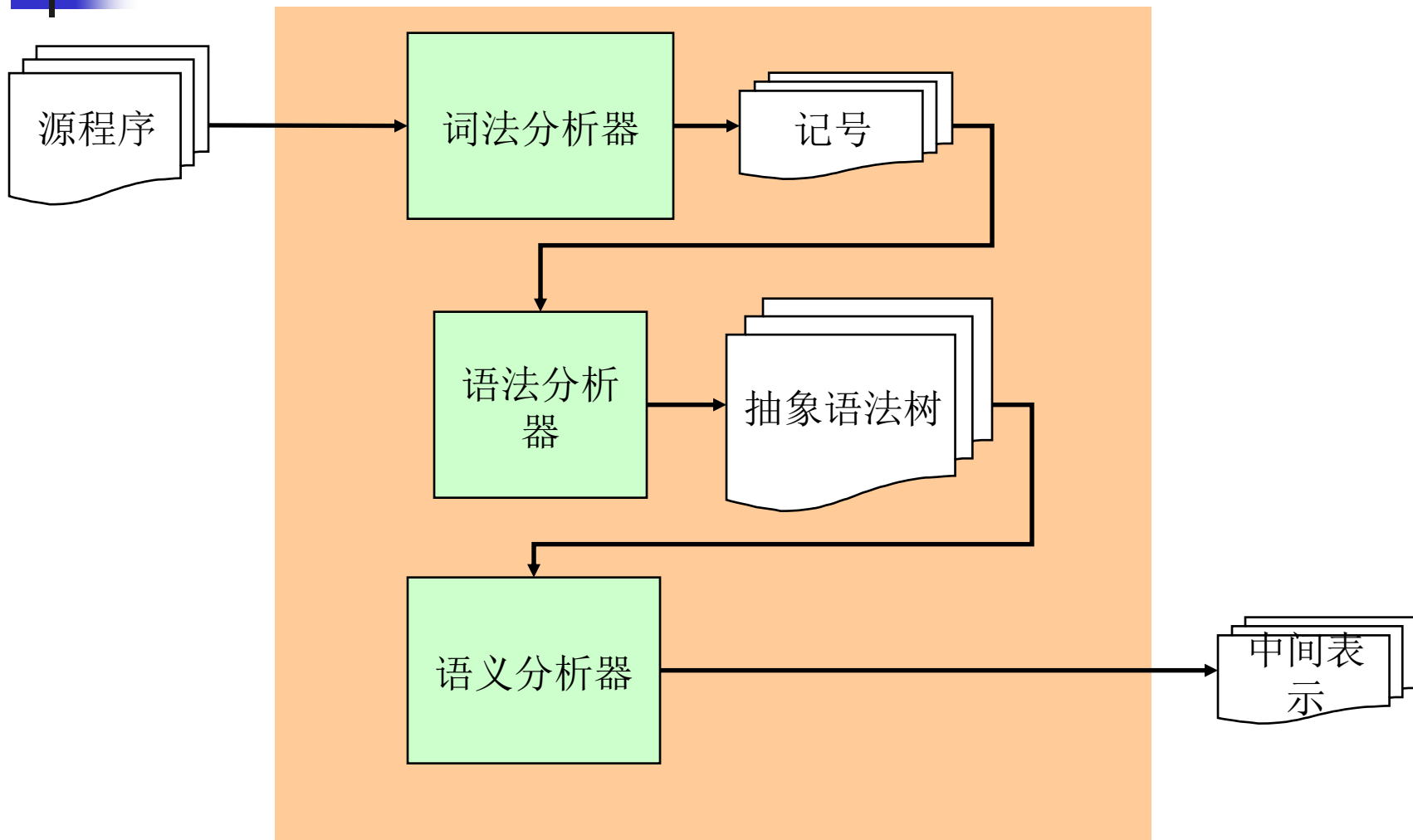
编译器的阶段



阶段

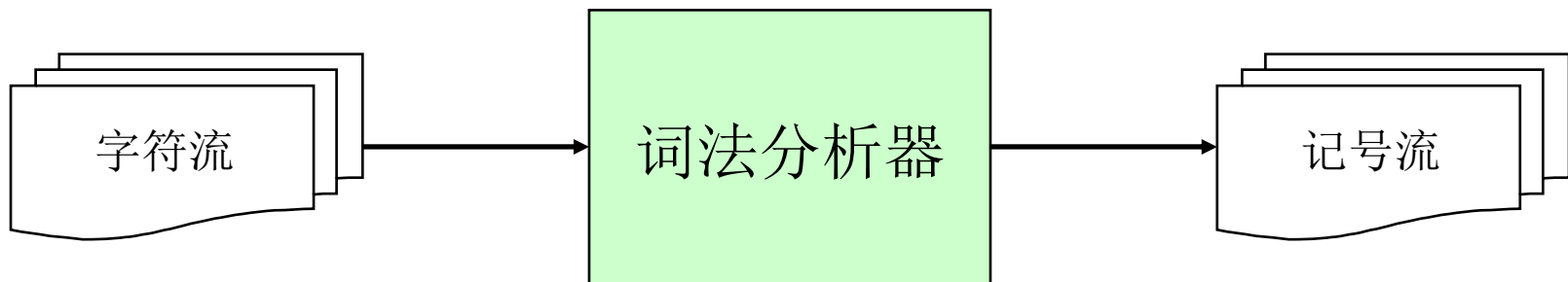


前端






词法分析器的任务





例子

```
if (x > 5)
    y = "hello";
else
    z = 1;
```



词法分析

```
IF LPAREN IDENT(x) GT INT(5) RPAREN
    IDENT(y) ASSIGN STRING("hello") SEMICOLON
ELSE
    IDENT(z) ASSIGN INT(1) SEMICOLON EOF
```



记号的数据结构定义

```
enum kind {IF, LPAREN, ID, INTLIT, ...};  
struct token{  
    enum kind k;  
    char *lexeme;  
};
```

例子:



小结

- 词法分析器的任务：字符流到记号流
 - 字符流：
 - 和被编译的语言密切相关 (ASCII, Unicode, or ...)
 - 记号流：
 - 编译器内部定义的数据结构，编码所识别出的词法单元



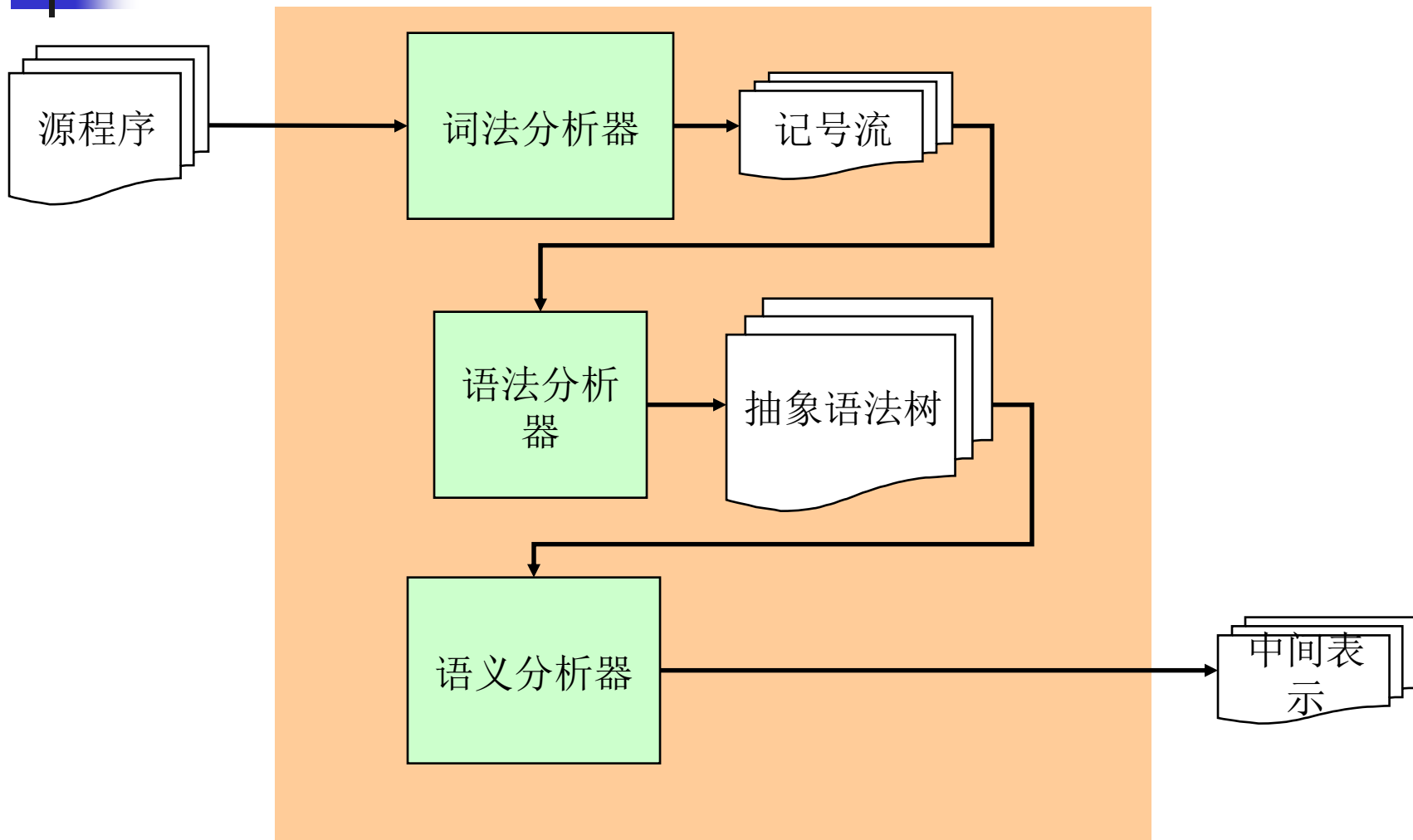
词法分析---手工构造法

编译原理

华保健

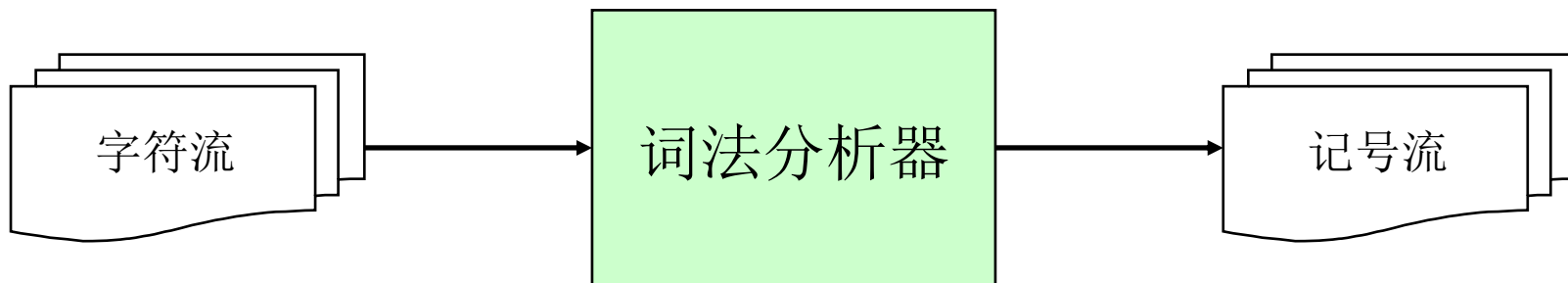
bjhua@ustc.edu.cn

前端





词法分析器的任务

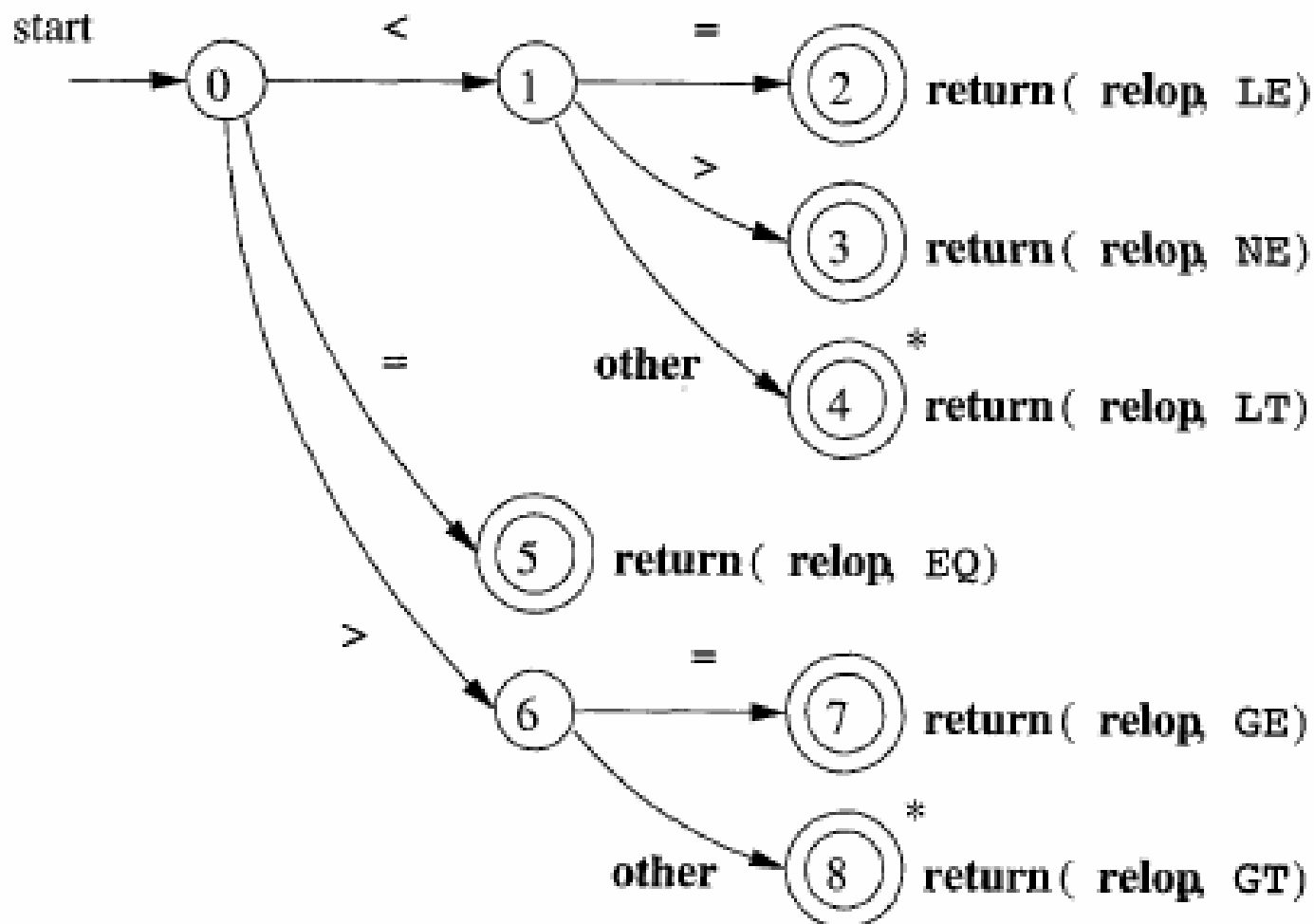




词法分析器的实现方法

- 至少两种实现方案：
 - 手工编码实现法
 - 相对复杂、且容易出错
 - 但是目前非常流行的实现方法
 - GCC, LLVM, ...
 - 词法分析器的生成器
 - 可快速原型、代码量较少
 - 但较难控制细节
- 我们先讨论第一种实现方案
 - 后面几讲会讨论第二种方案

转移图



转移图算法

```
token nextToken ()
```

```
    c = getChar ();
```

```
    switch (c)
```

```
        case '<': c = getChar ();
```

```
            switch (c)
```

```
                case '=': return LE;
```

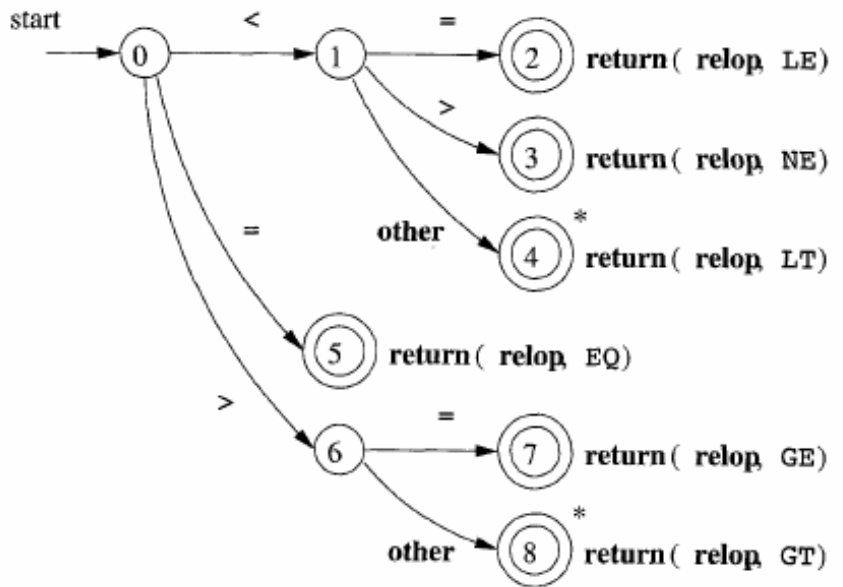
```
                case '>': return NE;
```

```
                default: rollback(); return LT;
```

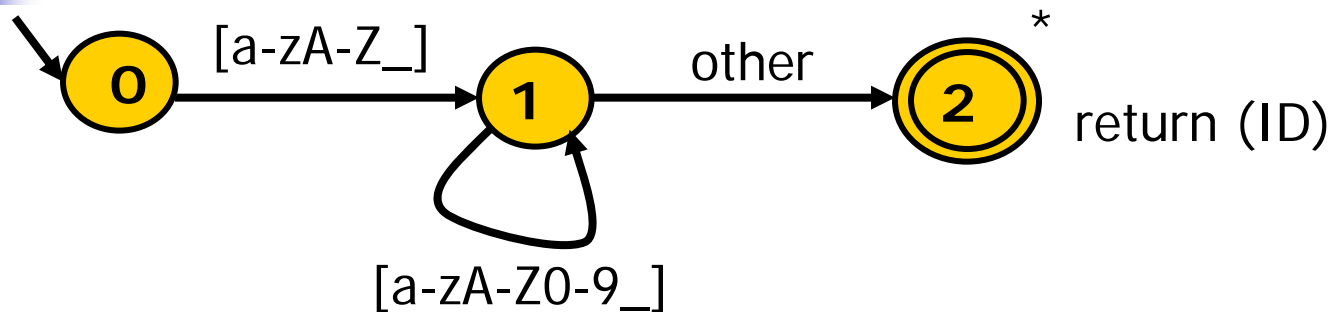
```
        case '=': return EQ;
```

```
        case '>': c = nextChar ();
```

```
            switch (c): // similar
```



标识符的转移图



```
token nextToken ()
```

```
    c = getChar();
```

```
    switch (c)
```

```
    // continued from above cases...
```

```
    case 'a', ..., 'z', 'A', ..., 'Z', '_':
```

```
        c = getChar ();
```

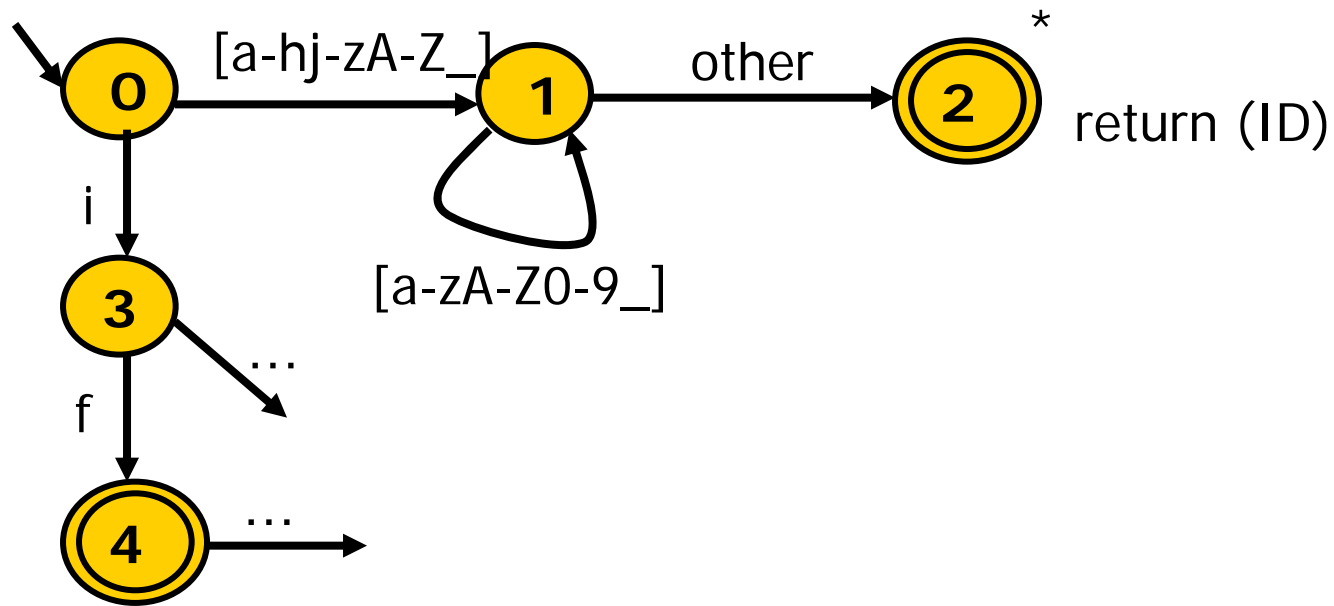
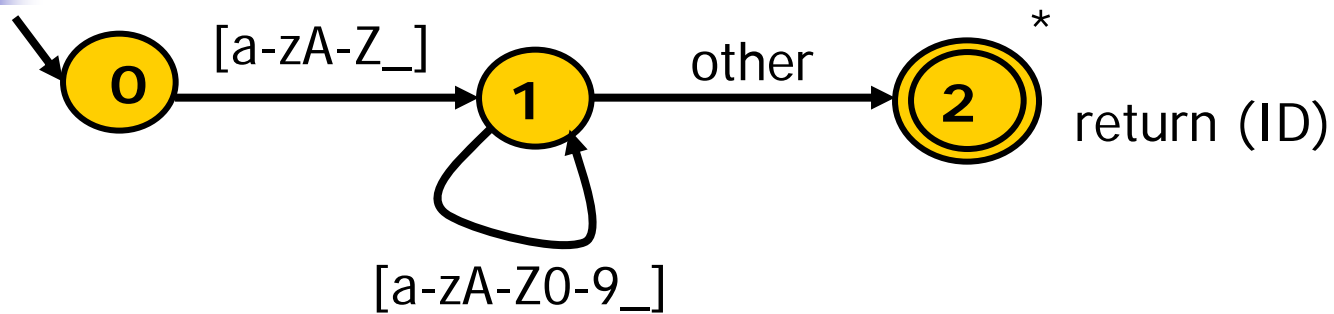
```
        while (c=='a' || c=='b' || ... || c=='_') c=getChar();
```




标识符和关键字

- 很多语言中的标识符和关键字有交集
 - 从词法分析的角度看，关键字是标识符的一部分
- 以C语言为例：
 - 标识符：以字母或下划线开头，后跟零个或多个字母、下划线、或数字
 - 关键字：if, while, else, ...

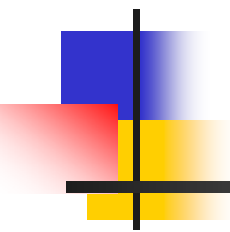
识别关键字（以if为例）





关键字表算法

- 对给定语言中所有的关键字，构造关键字构成的哈希表H
- 对所有的标识符和关键字，先统一按标识符的转移图进行识别
- 识别完成后，进一步查表H看是否是关键字
- 通过合理的构造哈希表H（完美哈希），可以 $O(1)$ 时间完成



词法分析---正则表达式

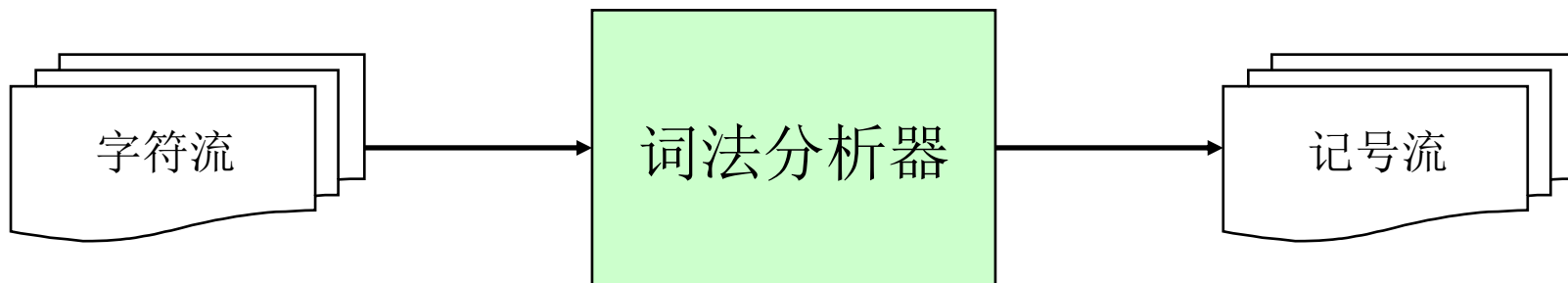
编译原理

华保健

bjhua@ustc.edu.cn



回顾



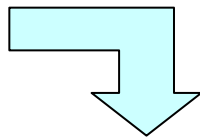


回顾：词法分析器的实现方法

- 至少两种实现方案：
 - 手工编码实现法
 - 相对复杂、且容易出错
 - 但是目前非常流行的实现方法
 - GCC, LLVM, ...
 - 词法分析器的生成器
 - 可快速原型、代码量较少
 - 但较难控制细节
- 我们已经讨论了第一种实现方案
 - 从这一讲开始讨论第二种方案

自动生成

声明式的规范



词法分析器



正则表达式

- 对给定的字符集 $\Sigma = \{c_1, c_2, \dots, c_n\}$
- 归纳定义：
 - 空串 ε 是正则表达式
 - 对于任意 $c \in \Sigma$, c 是正则表达式
 - 如果 M 和 N 是正则表达式, 则以下也是正则表达式
 - 选择 $M \mid N = \{M, N\}$
 - 连接 $MN = \{mn \mid m \in M, n \in N\}$
 - 闭包 $M^* = \{\varepsilon, M, MM, MMM, \dots\}$

正则表达式的形式表示

```
e -> ε
    | c
    | e | e
    | e e
    | e*
```

问题：对于给定字符集 $\Sigma = \{a, b\}$ ，可以写出哪些正则表达式？



例子：关键字

- C语言中的关键字，例如if，while等
 - 如何用正则表达式表示？



例子：标识符

- C语言中的标识符：以字母或下划线开头，后跟零个或多个字母、数字或下划线。
 - 如何用正则表达式表示？



例子：C语言中的无符号整数

- （十进制整型数）规则：或者是0；或者是1到9开头，后跟零个或多个0到9
 - 如何用正则表达式表示？



语法糖

- 可以引入更多的语法糖，来简化构造
 - $[c1-cn]$ == $c1|c2|\dots|cn$
 - $e+$ == 一个或多个 e
 - $e?$ == 零个或一个 e
 - “ a^* ” == a^* 自身, 不是 a 的Kleen闭包
 - $e\{i, j\}$ == i 到 j 个 e 的连接
 - $.$ == 除 ‘ $\backslash n$ ’ 外的任意字符



词法分析---有限状态自动机

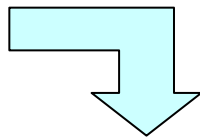
编译原理

华保健

bjhua@ustc.edu.cn

回顾：自动生成

声明式的规范



词法分析器

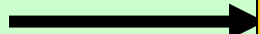


词法分析器的实现方法

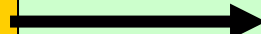
- 至少两种方法：
 - 手工实现算法
 - 自动生成法
- 我们继续讨论第二种方法
 - 首先要用到的第二个数学工具是有限状态自动机（FA）

有限状态自动机 (FA)

输入的字符串



FA



{Yes, No}

$$M = (\Sigma, S, q_0, F, \delta)$$

字母表

状态
集

初始
状态

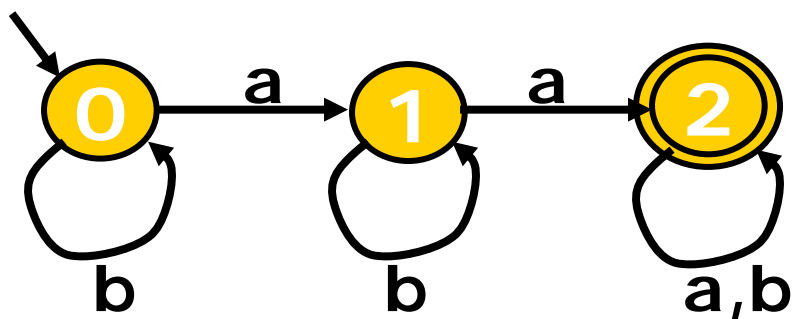
终结状
态集

转移函数

$$M = (\Sigma, S, q_0, F, \delta)$$

自动机例子

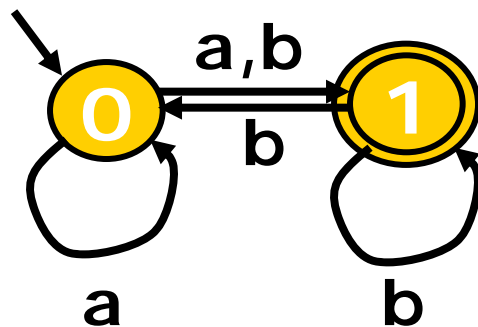
- 什么样的串可被接受?



- 转移函数:

- $\{ (q_0, a) \rightarrow q_1, (q_0, b) \rightarrow q_0, (q_1, a) \rightarrow q_2, (q_1, b) \rightarrow q_1, (q_2, a) \rightarrow q_2, (q_2, b) \rightarrow q_2 \}$

自动机第二个例子



- 转移函数:

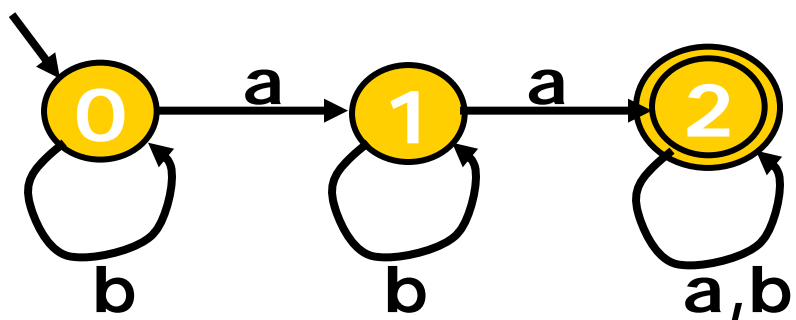
- $\{(q0, \mathbf{a}) \rightarrow \{q0, q1\},$
 $(q0, \mathbf{b}) \rightarrow \{q1\},$
 $(q1, \mathbf{b}) \rightarrow \{q0, q1\}\}$



有限状态自动机小结

- 确定状态有限自动机DFA
 - 对任意的字符，最多有一个状态可以转移
 - $\delta: S \times \Sigma \rightarrow S$
- 非确定的有限状态自动机NFA
 - 对任意的字符，有多于一个状态可以转移
 - $\delta: S \times (\Sigma \cup \epsilon) \rightarrow \wp(S)$

DFA的实现



状态\字符	a	b
0	1	0
1	2	1
2	2	2



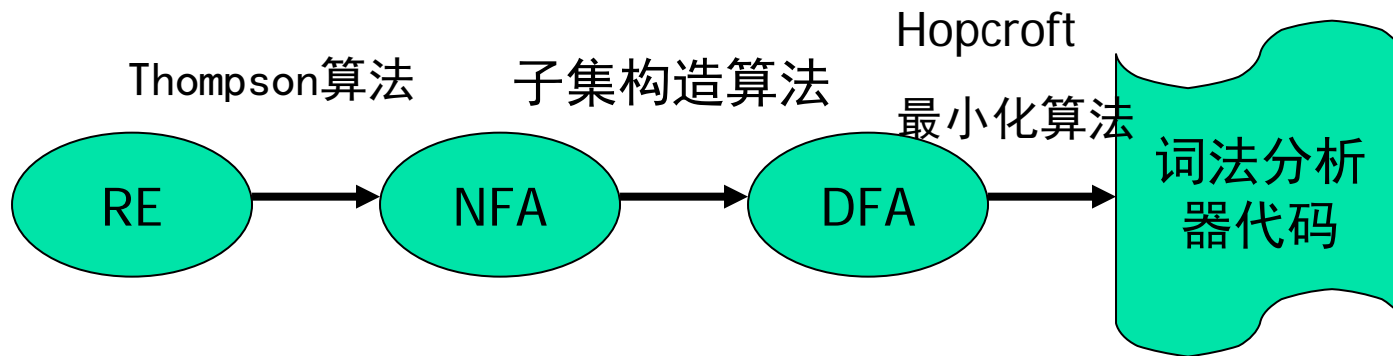
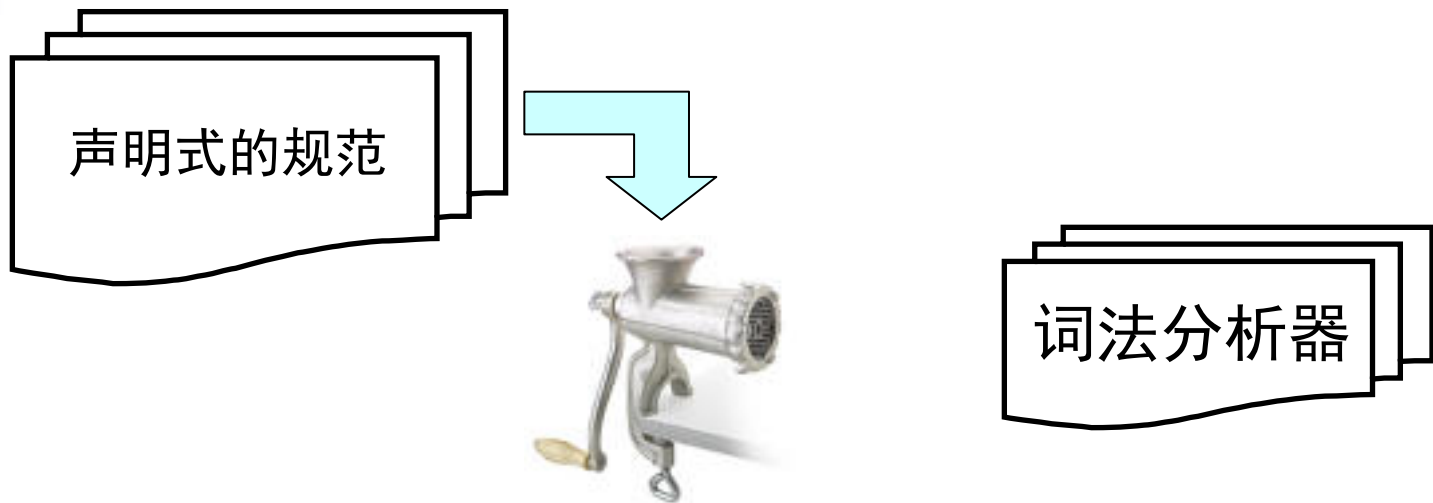
词法分析---正则表达式到非确定有限状态自动机

编译原理

华保健

bjhua@ustc.edu.cn

回顾：自动生成



RE -> NFA:

Thompson算法

- 基于对RE的结构做归纳
 - 对基本的RE直接构造
 - 对复合的RE递归构造
- 递归算法，容易实现
 - 在我们的实现里，不到100行的C代码

RE \rightarrow NFA:

Thompson algorithm

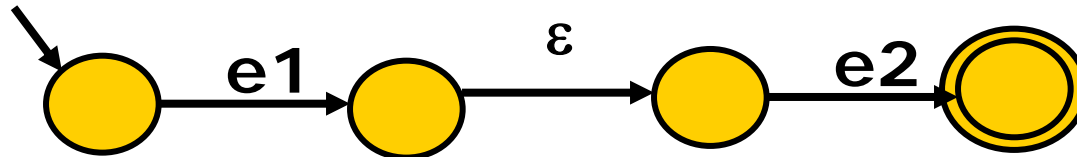
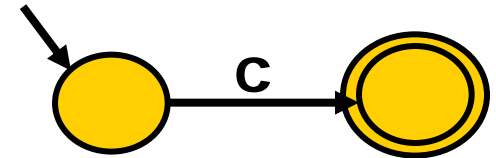
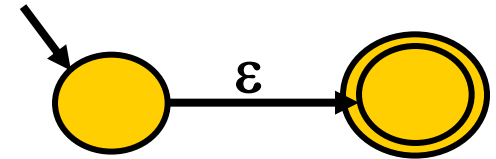
$e \rightarrow \epsilon$

$\rightarrow c$

$\rightarrow e_1 e_2$

$\rightarrow e_1 \mid e_2$

$\rightarrow e_1^*$



RE \rightarrow NFA:

Thompson algorithm

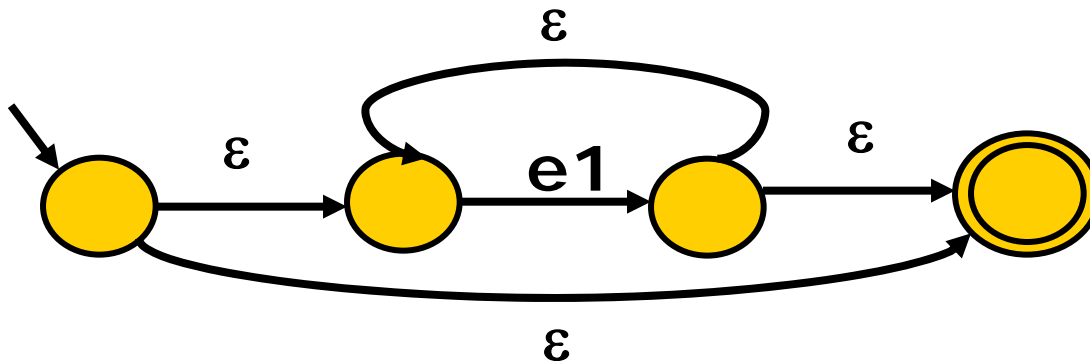
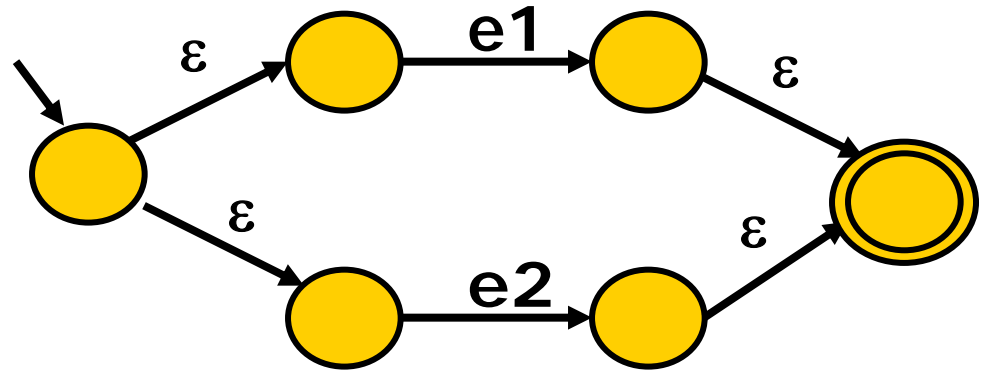
$e \rightarrow \epsilon$

$\rightarrow c$

$\rightarrow e_1 e_2$

$\rightarrow e_1 \mid e_2$

$\rightarrow e_1^*$





示例

$a(b|c)^*$



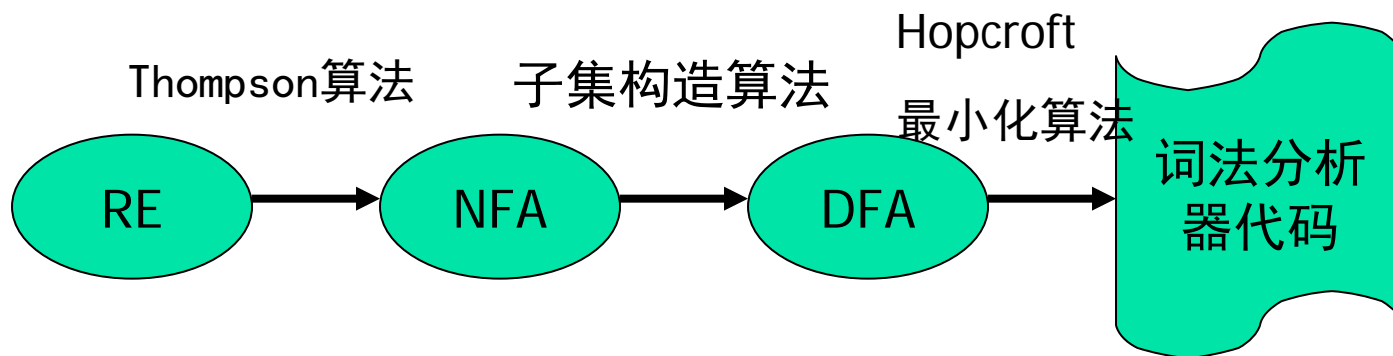
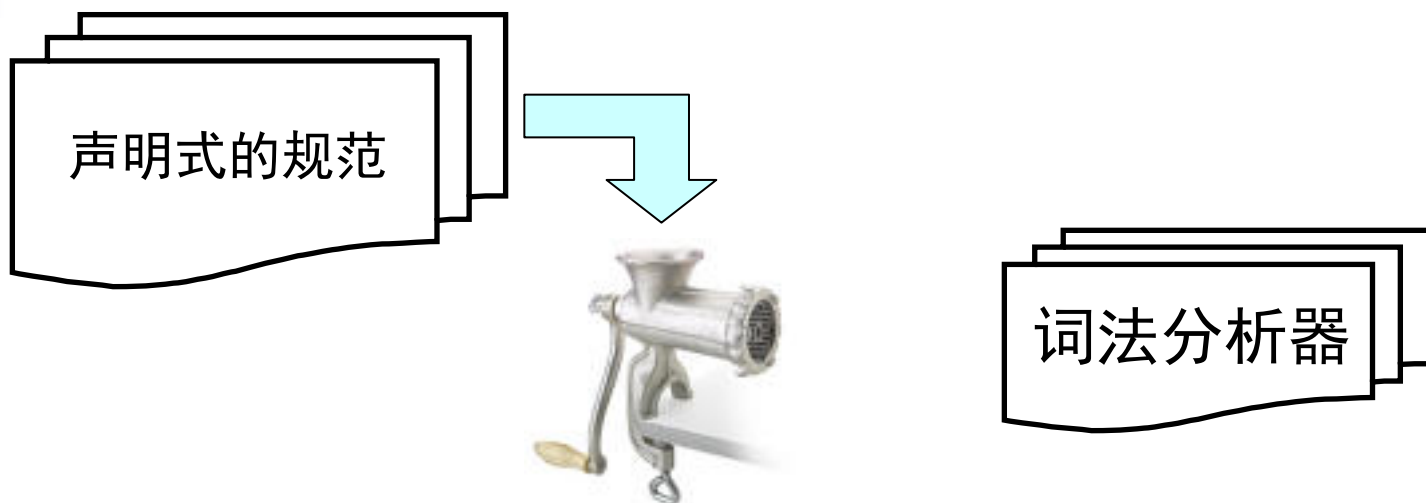
词法分析---NFA转换到DFA

编译原理

华保健

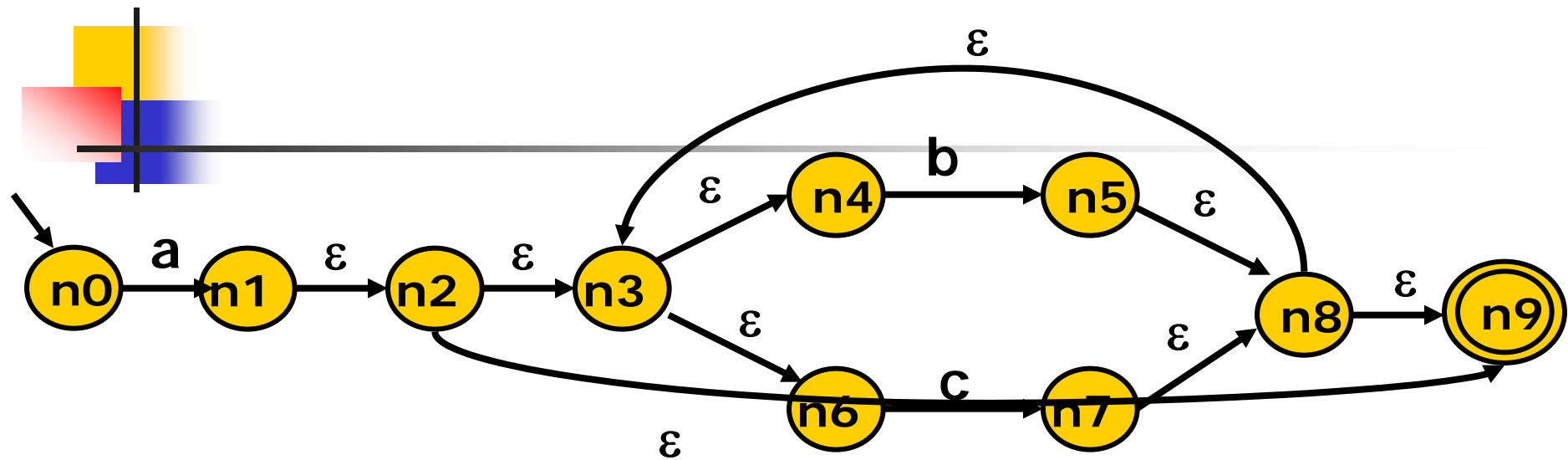
bjhua@ustc.edu.cn

回顾：自动生成



算法思想

$a(b|c)^*$





子集构造算法

(* 子集构造算法：工作表算法 *)

```
q0 <- eps_closure (n0)
```

```
Q <- {q0}
```

```
workList <- q0
```

```
while (workList != [])
```

```
  remove q from workList
```

```
  foreach (character c)
```

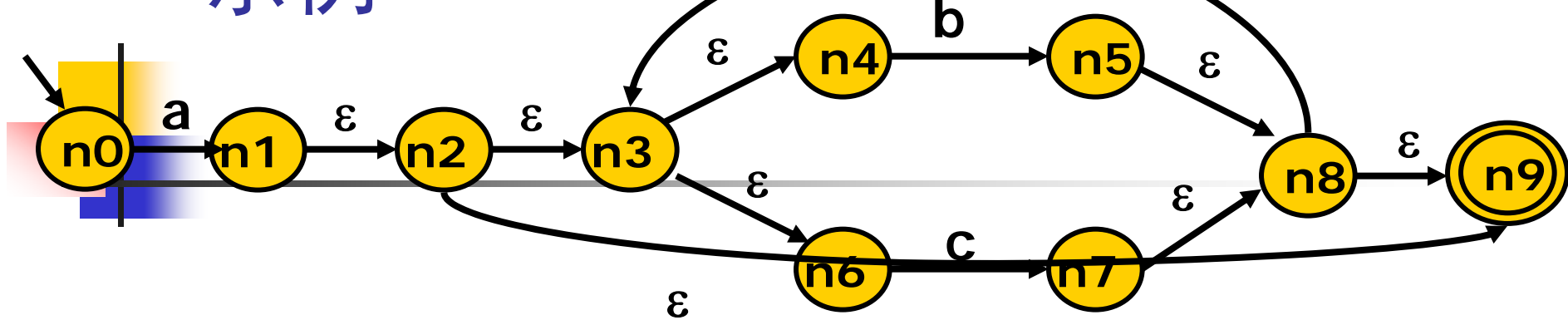
```
    t <- e-closure (delta (q, c))
```

```
    D[q, c] <- t
```

```
    if (t \not\in Q)
```

```
      add t to Q and workList
```

示例



(* 子集构造算法：工作表算法 *)

```
q0 <- eps_closure (n0)
```

```
Q <- {q0}
```

```
workList <- q0
```

```
while (workList != [])
```

```
  remove q from workList
```

```
  foreach (character c)
```

```
    t <- e-closure (delta (q, c))
```

```
    D[q, c] <- t
```

```
    if (t \not\in Q)
```

```
      add t to Q and workList
```



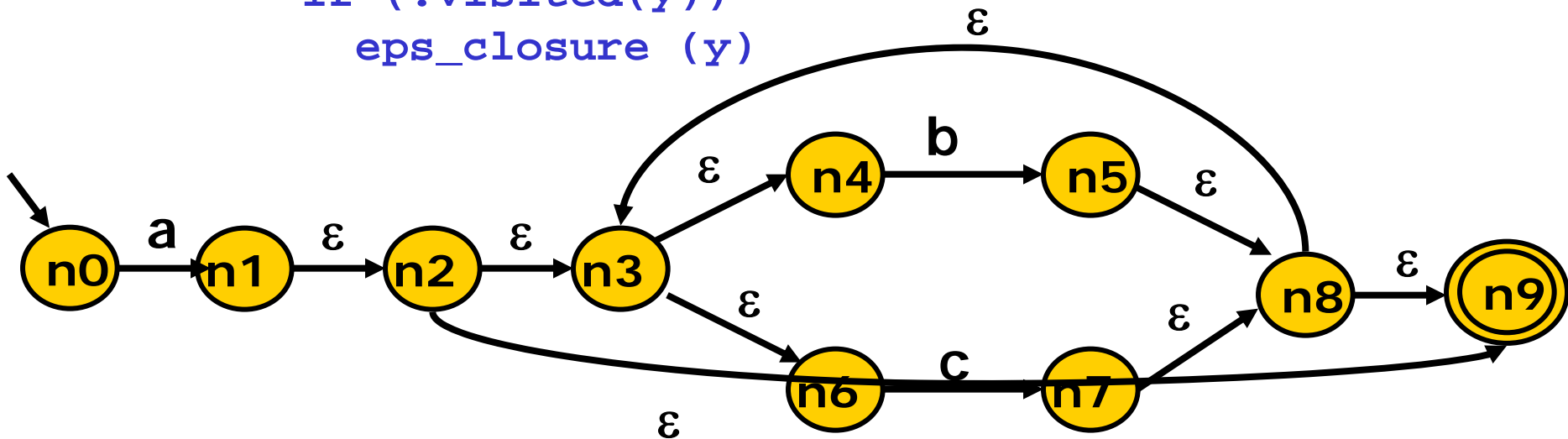

对算法的讨论

- 不动点算法
 - 算法为什么能够运行终止
- 时间复杂度
 - 最坏情况 $O(2^N)$
 - 但在实际中不常发生
 - 因为并不是每个子集都会出现

ϵ -闭包的计算：深度优先

```
/*  $\epsilon$ -closure: 基于深度优先遍历的算法 */  
set closure = {};
```

```
void eps_closure (x)  
    closure += {x}  
    foreach (y: x-- $\epsilon$ --> y)  
        if (!visited(y))  
            eps_closure (y)
```





ε -闭包的计算：宽度优先

```
/*  $\varepsilon$ -closure: 基于宽度优先的算法 */  
set closure = {};  
Q = []; // queue  
void eps_closure (x) =  
    Q = [x];  
    while (Q not empty)  
        q <- deQueue (Q)  
        closure += q  
        foreach (y: q-- $\varepsilon$ --> y)  
            if (!visited(y))  
                enqueue (Q, y)
```



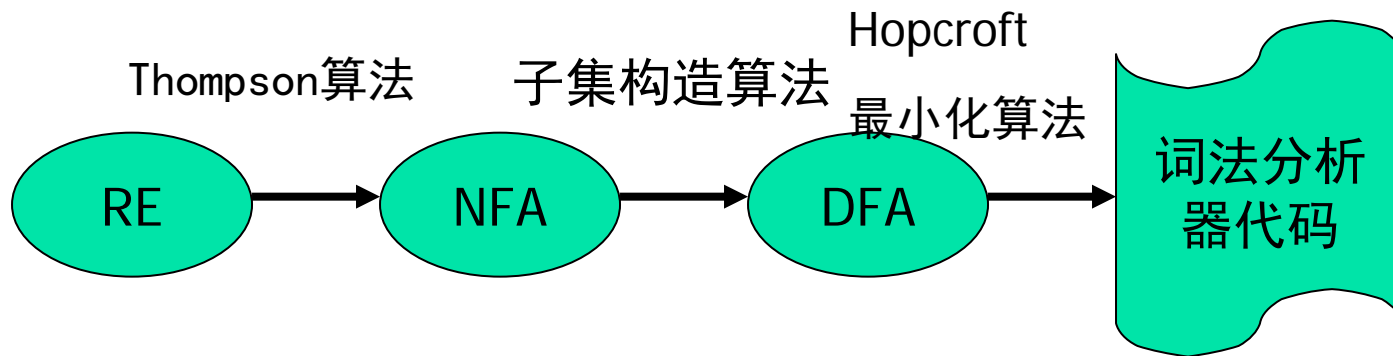
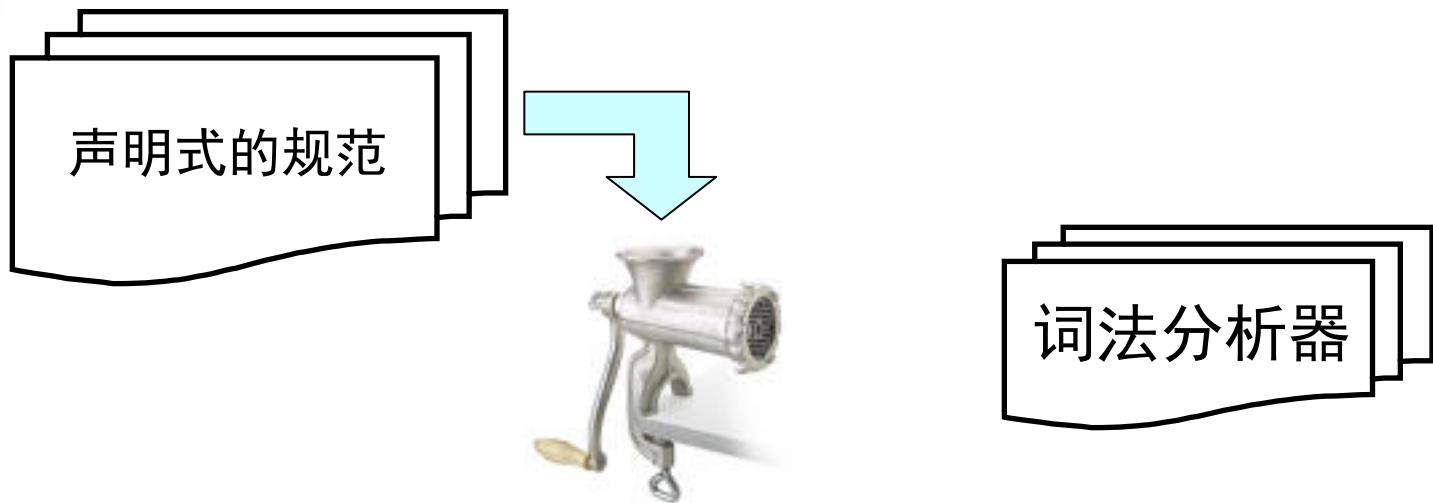
词法分析---DFA的最小化

编译原理

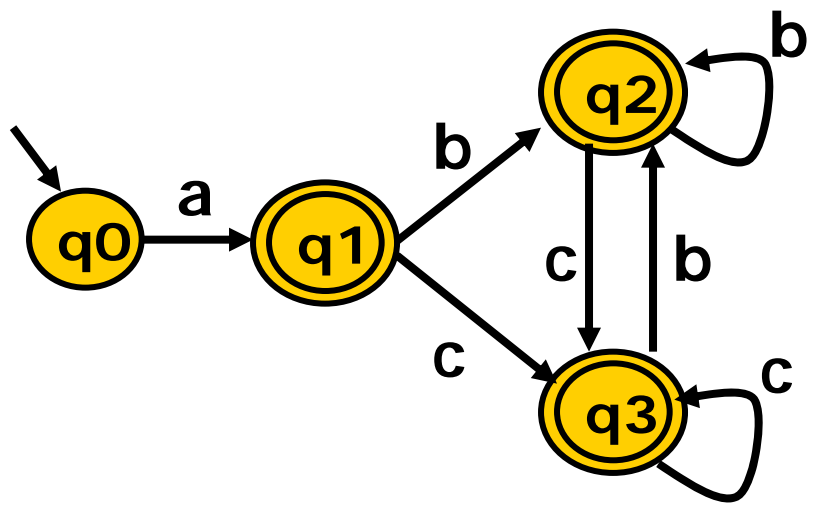
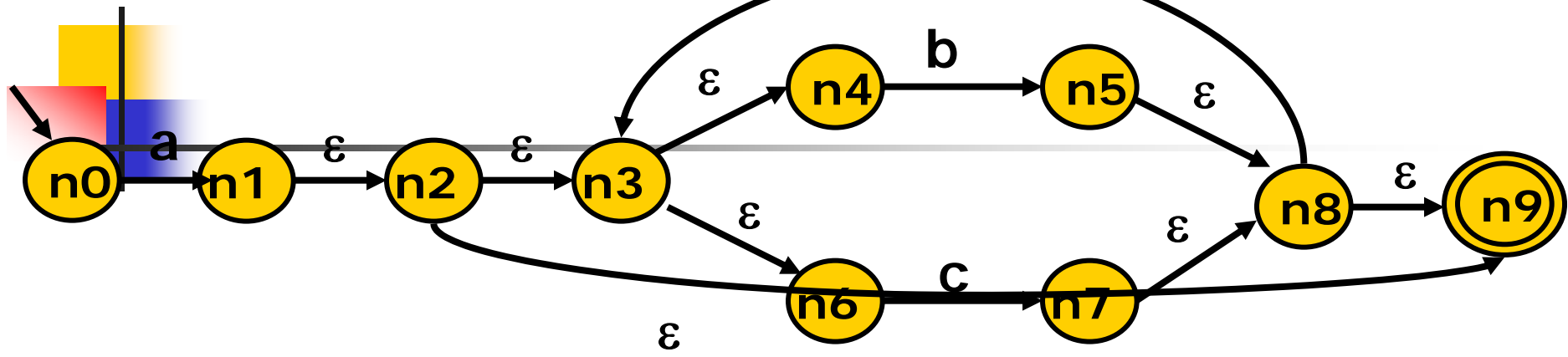
华保健

bjhua@ustc.edu.cn

回顾：自动生成



算法思想





Hopcroft算法

// 基于等价类的思想

```
split(S)
```

```
    foreach (character c)
```

```
        if (c can split S)
```

```
            split S into T1, ..., Tk
```

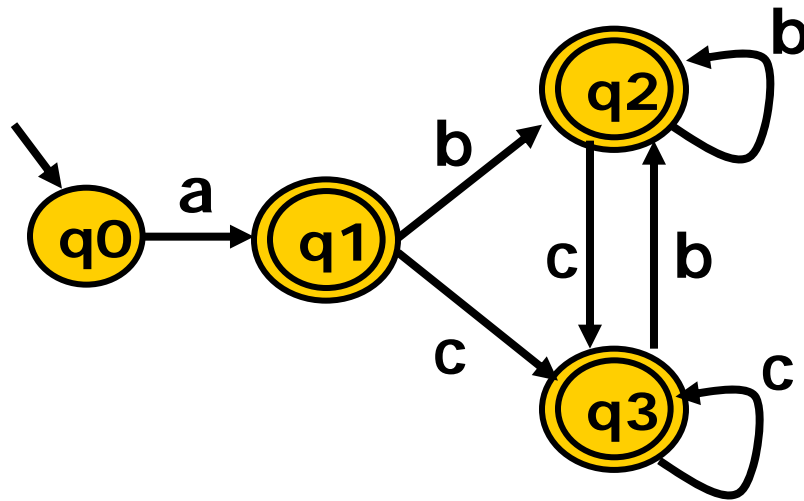
```
hopcroft ()
```

```
    split all nodes into N, A
```

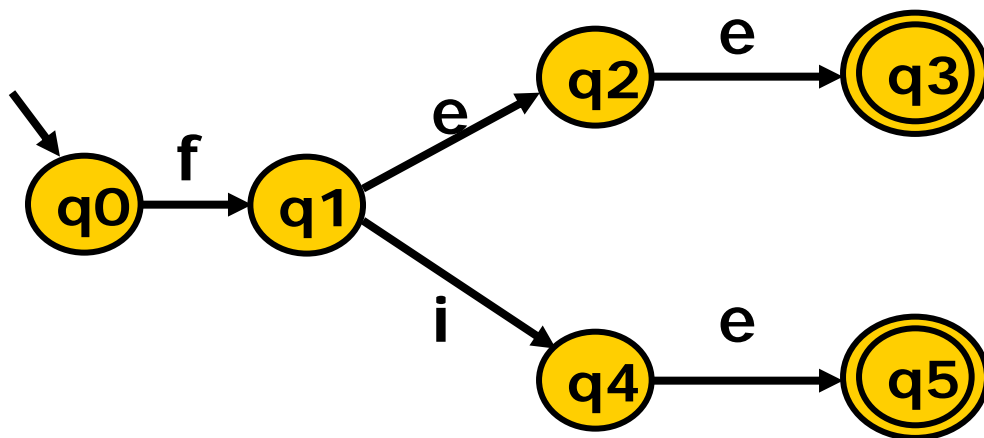
```
    while (set is still changes)
```

```
        split(S)
```

示例1



示例2





对算法的讨论

- 不动点算法
 - 算法为什么能够运行终止
- 时间复杂度
 - 最坏情况 $O(2^N)$?
 - 实际中运行可能会更快
 - 因为并不是每个子集都会分裂



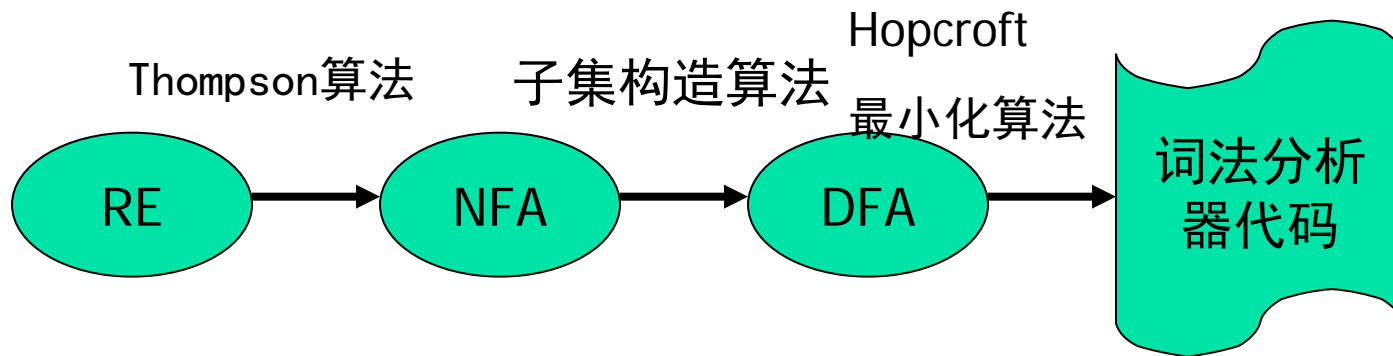
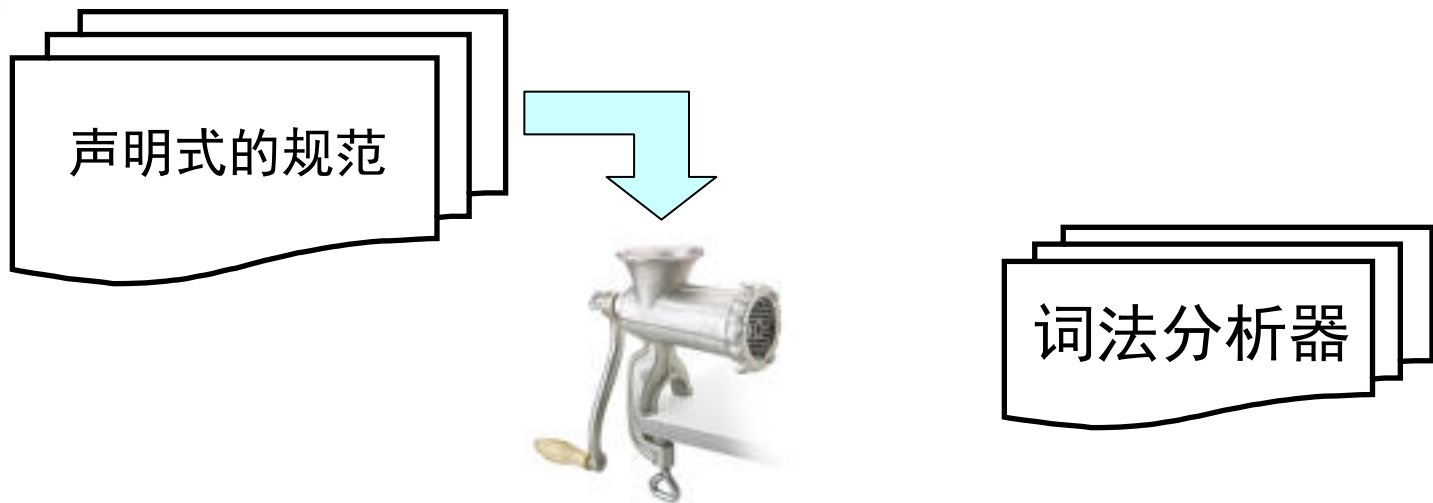
词法分析---DFA的代码表示

编译原理

华保健

bjhua@ustc.edu.cn

回顾：自动生成

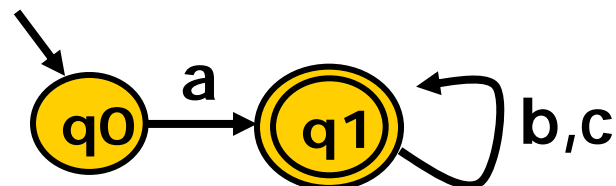




DFA的代码表示

- 概念上讲, DFA是一个有向图
- 实际上, 有不同的DFA的代码表示
 - 转移表 (类似于邻接矩阵)
 - 哈希表
 - 跳转表
 - . . .
- 取决于在实际实现中, 对时间空间的权衡

转移表



状态\字符	a	b	c
0	1		
1		1	1

```
char table[M][N];
```

```
table[0]['a']=1;
```

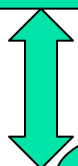
```
table[1]['b']=1;
```

```
table[1]['c']=1;
```

```
// other table entries
```

```
// are ERROR
```

转移表



词法分析
驱动代码

驱动代码

```
nextToken()
```

```
    state = 0
```

```
    stack = []
```

```
    while (state!=ERROR)
```

```
        c = getChar()
```

```
        if (state is ACCEPT)
```

```
            clear(stack)
```

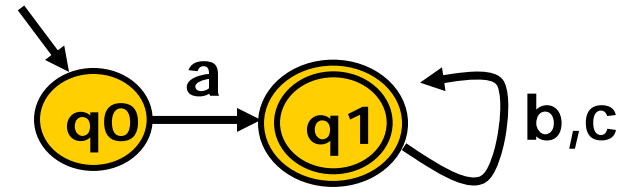
```
            push(state)
```

```
            state = table[state][c]
```

```
    while(state is not ACCEPT)
```

```
        state = pop();
```

```
        rollback();
```



状态\字符	a	b	c
0	1		
1		1	1

```
char table[M][N];
```

```
table[0]['a']=1;
```

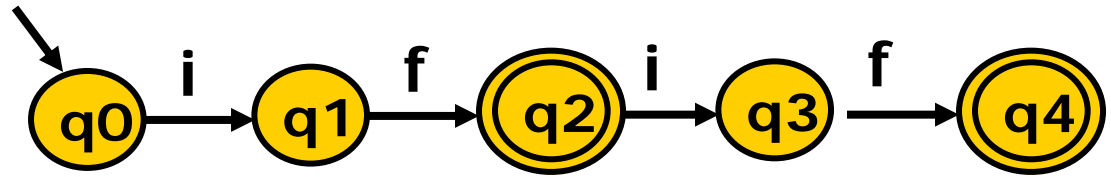
```
table[1]['b']=1;
```

```
table[1]['c']=1;
```

```
// other table entries
```

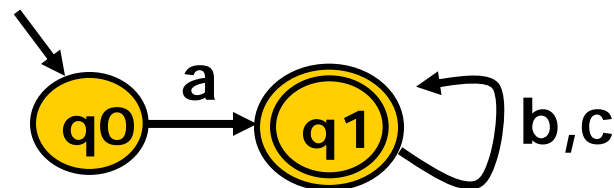
```
// are ERROR
```

最长匹配



```
nextToken()  
    state = 0  
    stack = []  
    while (state!=ERROR)  
        c = getChar()  
        if (state is ACCEPT)  
            clear(stack)  
            push(state)  
            state = table[state][c]  
  
    while(state is not ACCEPT)  
        state = pop();  
        rollback();
```


跳转表



```
nextToken()
```

```
    state = 0
```

```
    stack = []
```

```
    goto q0
```

```
q0:
```

```
    c = getChar()
```

```
    if (state is ACCEPT)
```

```
        clear (stack)
```

```
    push (state)
```

```
    if (c=='a')
```

```
        goto q1:
```

状态\字符	a	b	c
0	1		
1		1	1

```
q1:
```

```
    c = getChar()
```

```
    if (state is ACCEPT)
```

```
        clear (stack)
```

```
    push (state)
```

```
    if (c=='b' || c=='c')
```

```
        goto q1
```



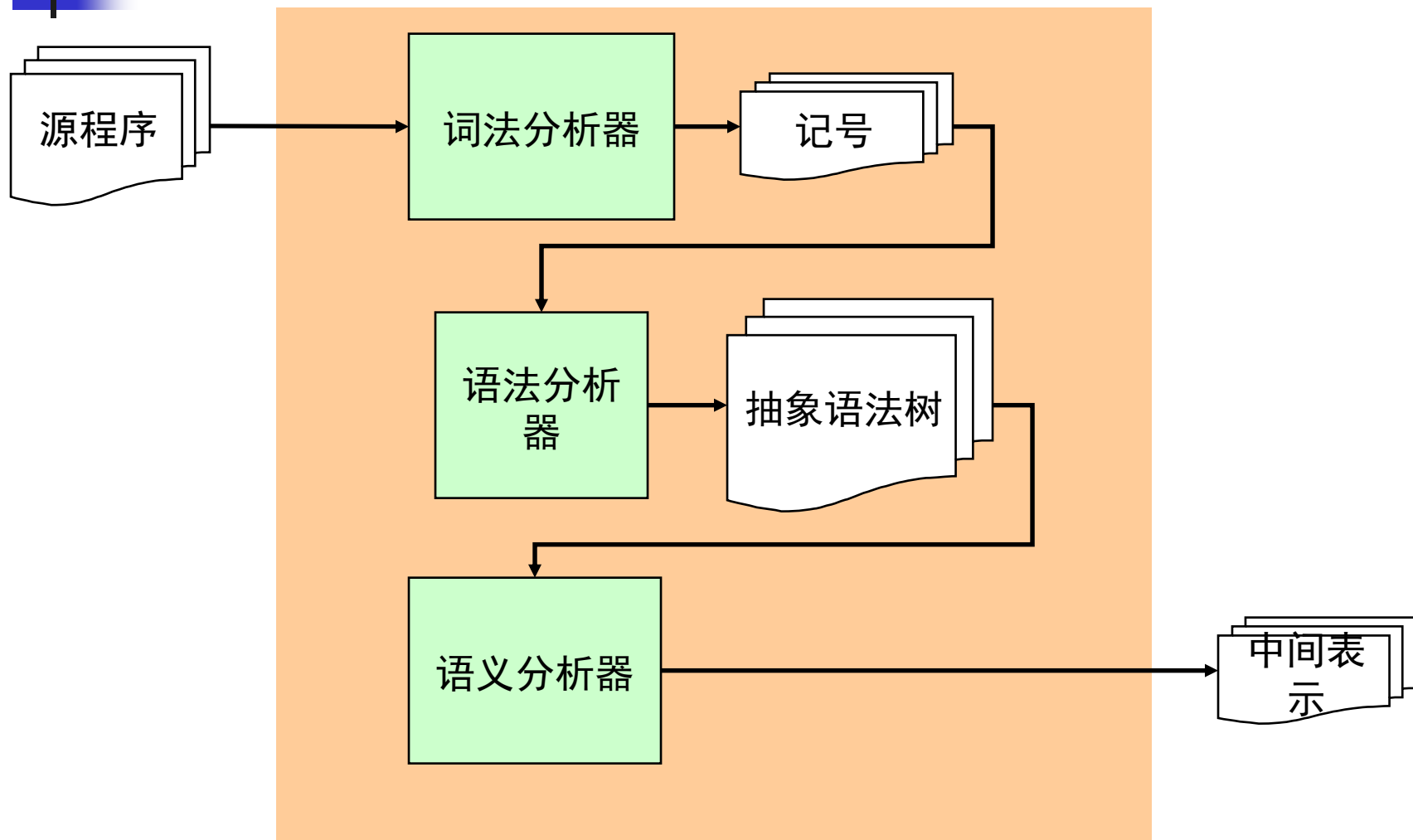
语法分析---简介

编译原理

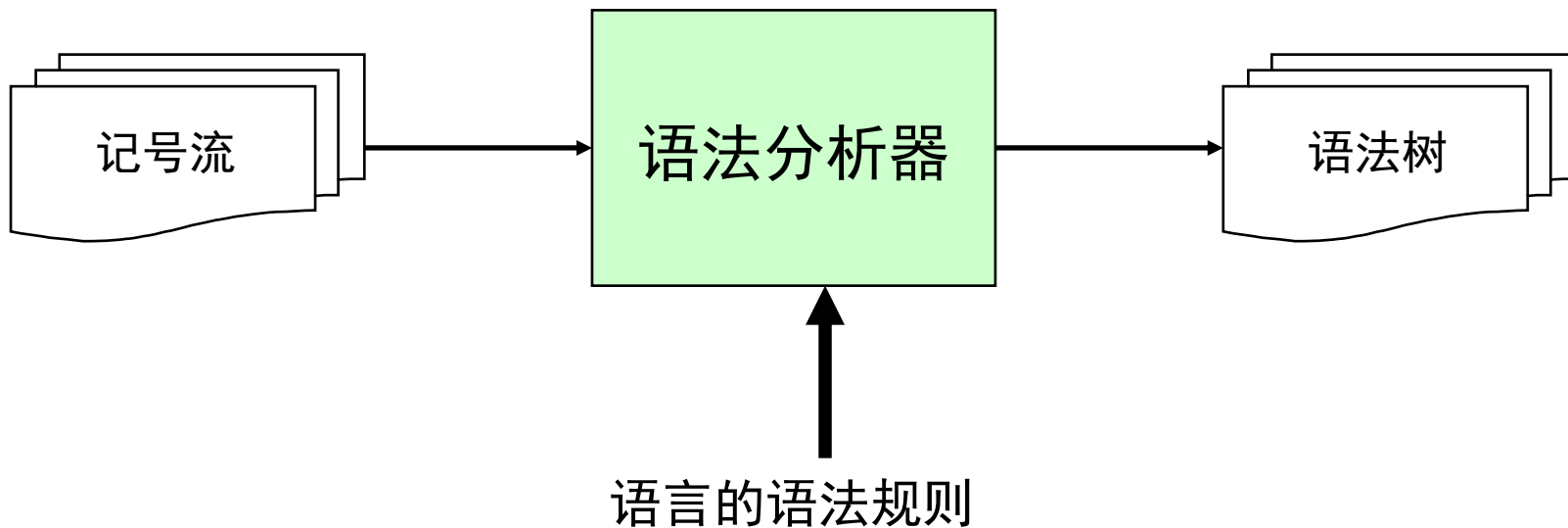
华保健

bjhua@ustc.edu.cn

前端



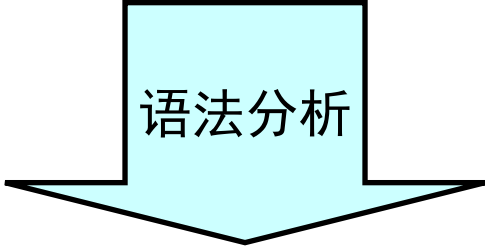
语法分析器的任务





例子：语法错误处理

```
if ((x > 5)
    y = "hello"
else
    z = 1,
```



语法分析

Syntax Error: line 1, missing)

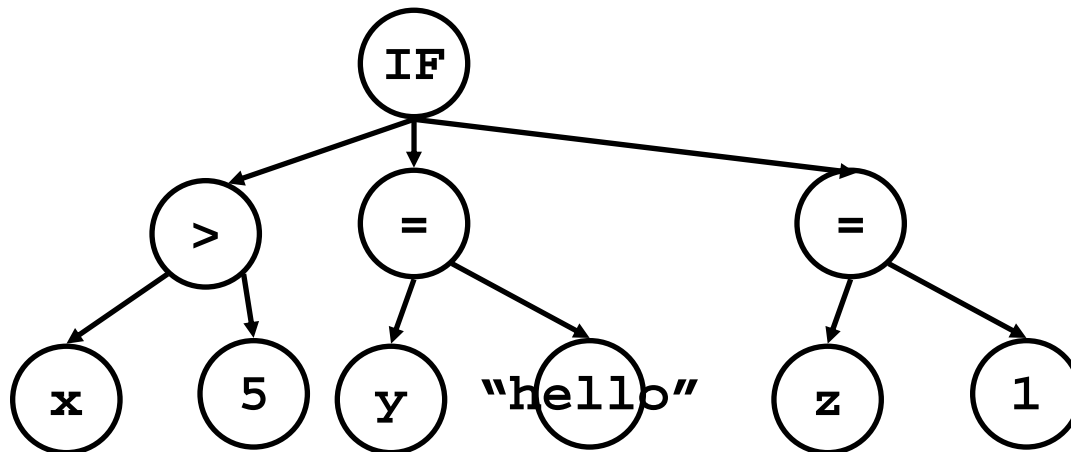
Syntax Error: line 2, missing ;

Syntax Error: line 4, expecting ; but got ,

例子：语法树构建

```
if (x > 5)
    y = "hello";
else
    z = 1;
```

语法分析





路线图

- 数学理论：上下文无关文法（CFG）
 - 描述语言语法规则的数学工具
- 自顶向下分析
 - 递归下降分析算法（预测分析算法）
 - LL分析算法
- 自底向上分析
 - LR分析算法



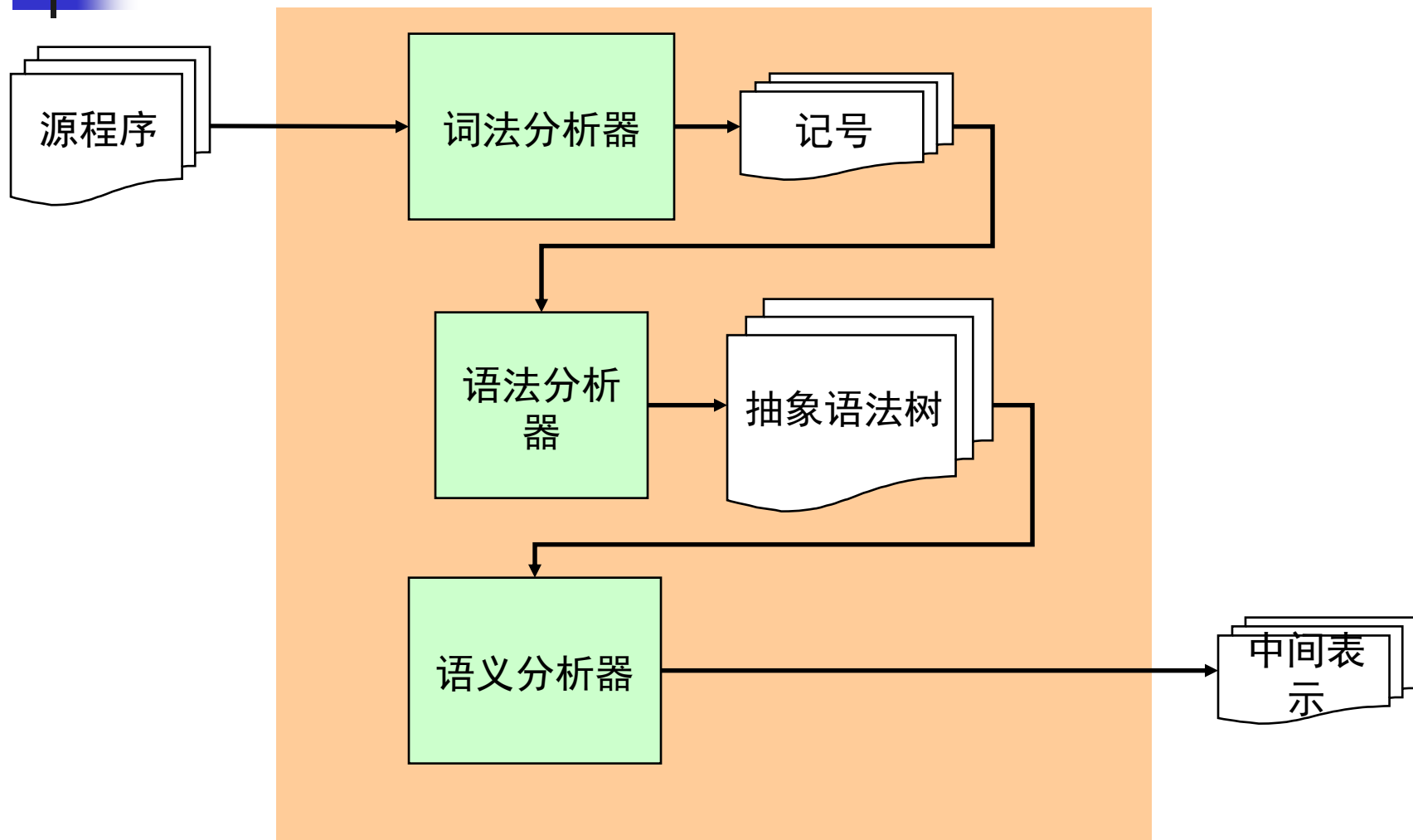
语法分析---简介

编译原理

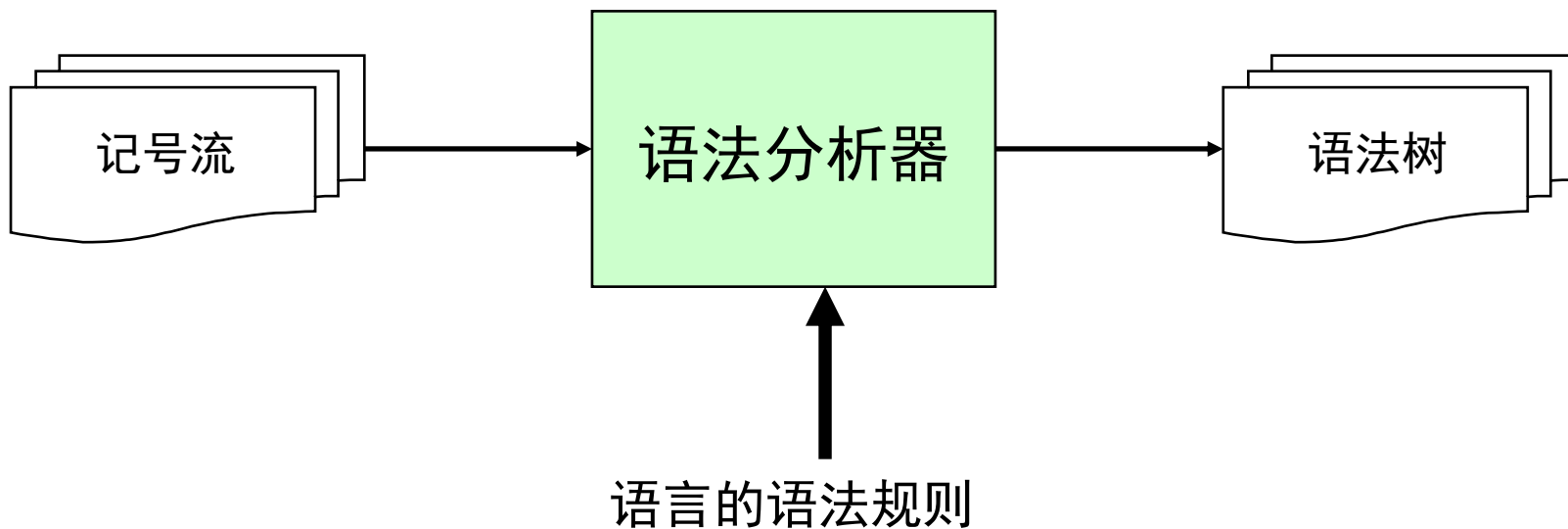
华保健

bjhua@ustc.edu.cn

前端



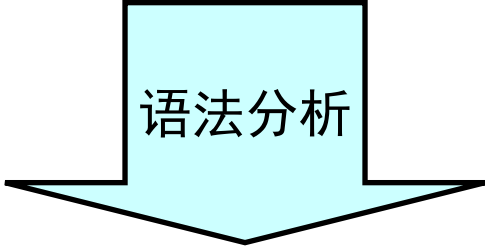
语法分析器的任务





例子：语法错误处理

```
if ((x > 5)
    y = "hello"
else
    z = 1,
```



语法分析

Syntax Error: line 1, missing)

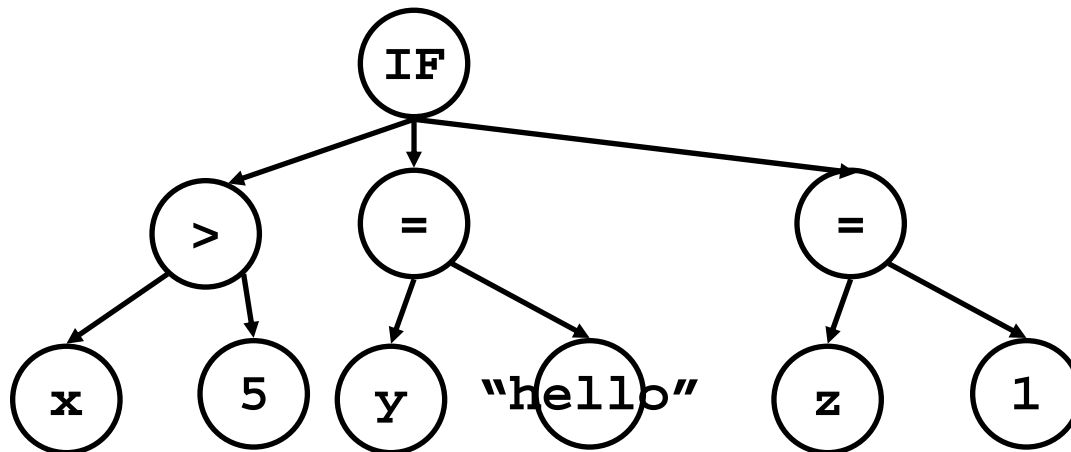
Syntax Error: line 2, missing ;

Syntax Error: line 4, expecting ; but got ,

例子：语法树构建

```
if (x > 5)  
    y = "hello";  
else  
    z = 1;
```

语法分析





路线图

- 数学理论：上下文无关文法（CFG）
 - 描述语言语法规则的数学工具
- 自顶向下分析
 - 递归下降分析算法（预测分析算法）
 - LL分析算法
- 自底向上分析
 - LR分析算法



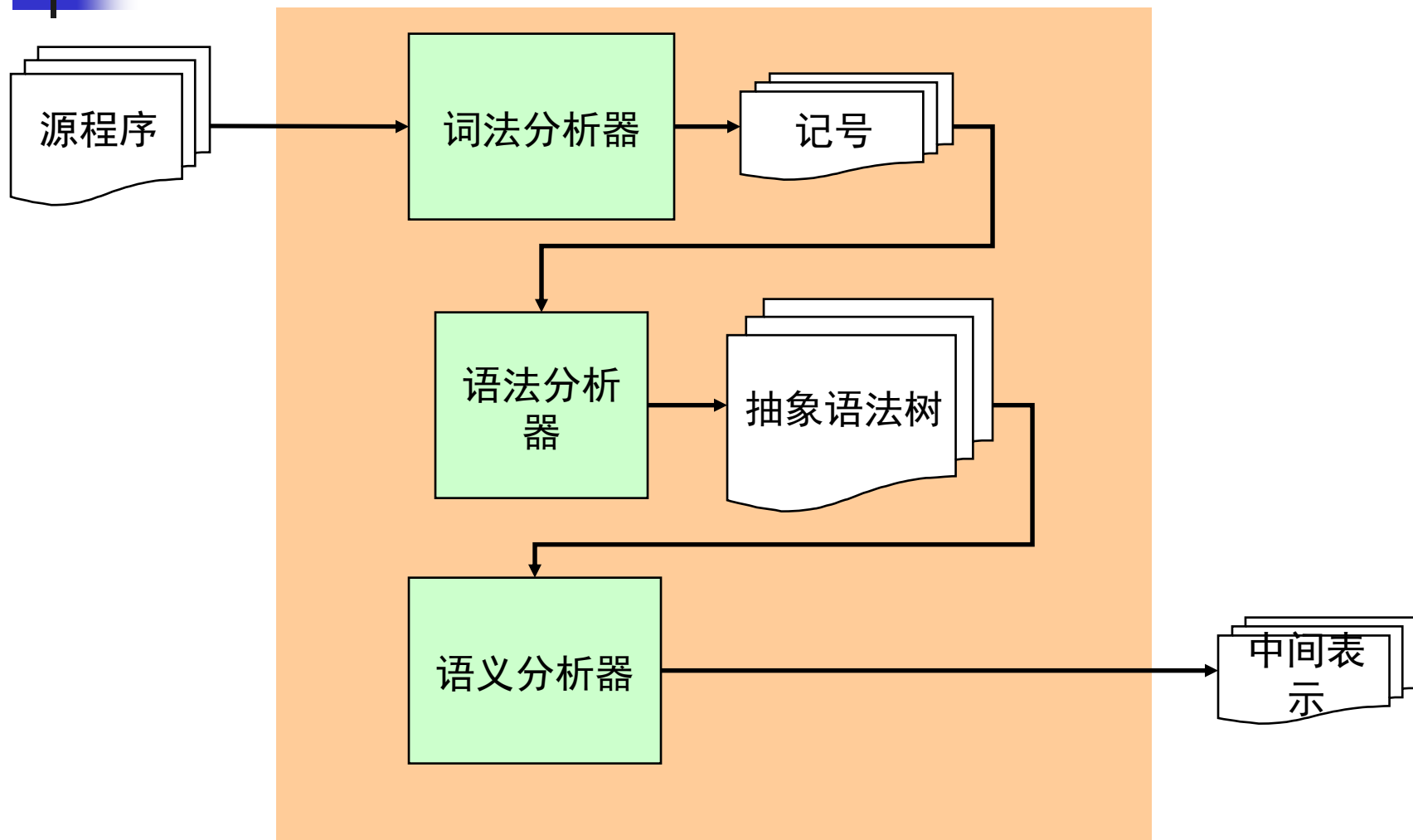
语法分析---上下文无关文法

编译原理

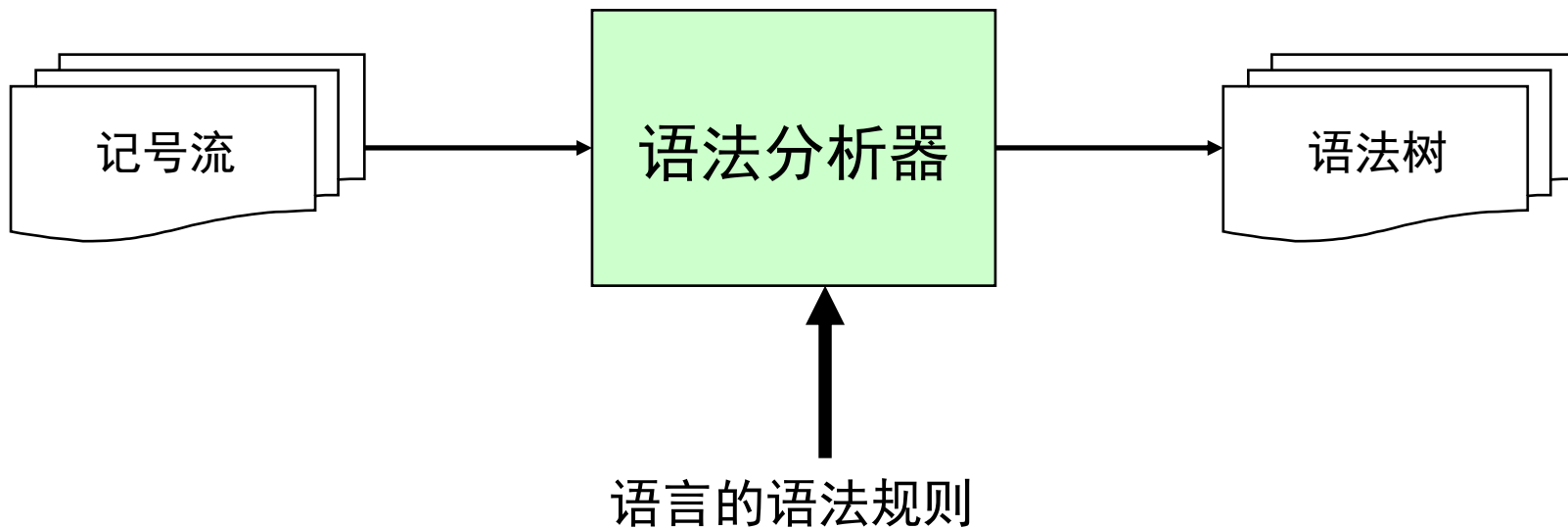
华保健

bjhua@ustc.edu.cn

前端

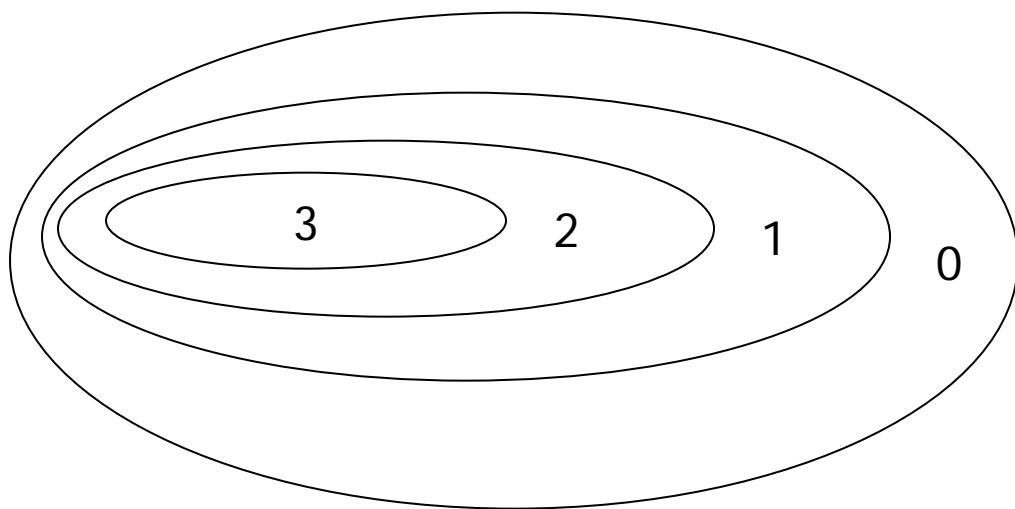


语法分析器的任务



历史背景：乔姆斯基文法体系

- 为研究自然语言构造的一系列数学工具





示例

- 自然语言中的句子的典型结构：
 - 主语 谓语 宾语
 - 名字 动词 名词
- 例子：
 - 名词：{羊、老虎、草、水}
 - 动词：{吃、喝}
- 句子：



形式化

$S \rightarrow N V N$

$N \rightarrow s$

$\quad | t$

$\quad | g$

$\quad | w$

$V \rightarrow e$

$\quad | d$

非终结符: $\{s, N, v\}$

终结符: $\{s, t, g, w, e, d\}$

开始符号: s



上下文无关文法

- 上下文无关文法G是一个四元组：

$$G = (T, N, P, S)$$

- 其中T是终结符集合
- N是非终结符集合
- P是一组产生式规则
 - 每条规则的形式： $X \rightarrow \beta_1 \beta_2 \cdots \beta_n$
 - 其中 $X \in N$, $\beta_i \in (T \cup N)$
- S是唯一的开始符号（非终结符）
 - $S \in N$



上下文无关文法的例子

$S \rightarrow N V N$

$N \rightarrow s$

| t

| g

| w

$V \rightarrow e$

| d

$G = (N, T, P, S)$

非终结符: $N = \{s, N, v\}$

终结符: $T = \{s, t, g, w, e, d\}$

开始符号: s

产生式规则集合:



上下文无关文法的例子

```
E -> num  
    | id  
    | E + E  
    | E * E
```

$G = (N, T, P, S)$

非终结符: $N = \{E\}$

终结符: $T = \{\text{num}, \text{id}, +, *\}$

开始符号: E

产生式规则集合:



推导

- 给定文法G，从G的开始符号S开始，用产生式的右部替换左侧的**非终结符**
- 此过程不断重复，直到不出现**非终结符**为止
- 最终的串称为**句子**

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```



最左推导和最右推导

- **最左推导**：每次总是选择**最左侧**的符号进行替换

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

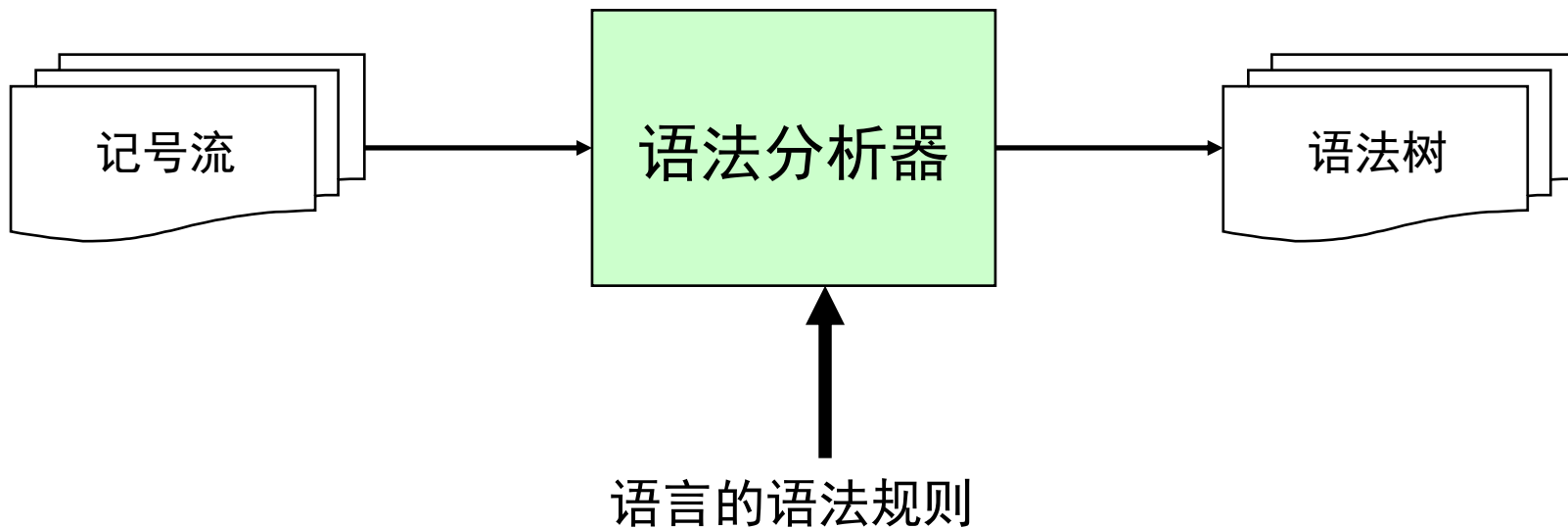


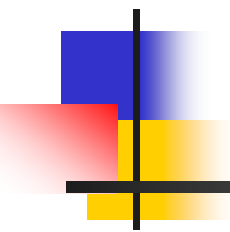

语法分析

- 给定文法 G 和句子 s ，语法分析要回答的问题：是否存在对句子 s 的推导？

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

语法分析器的任务





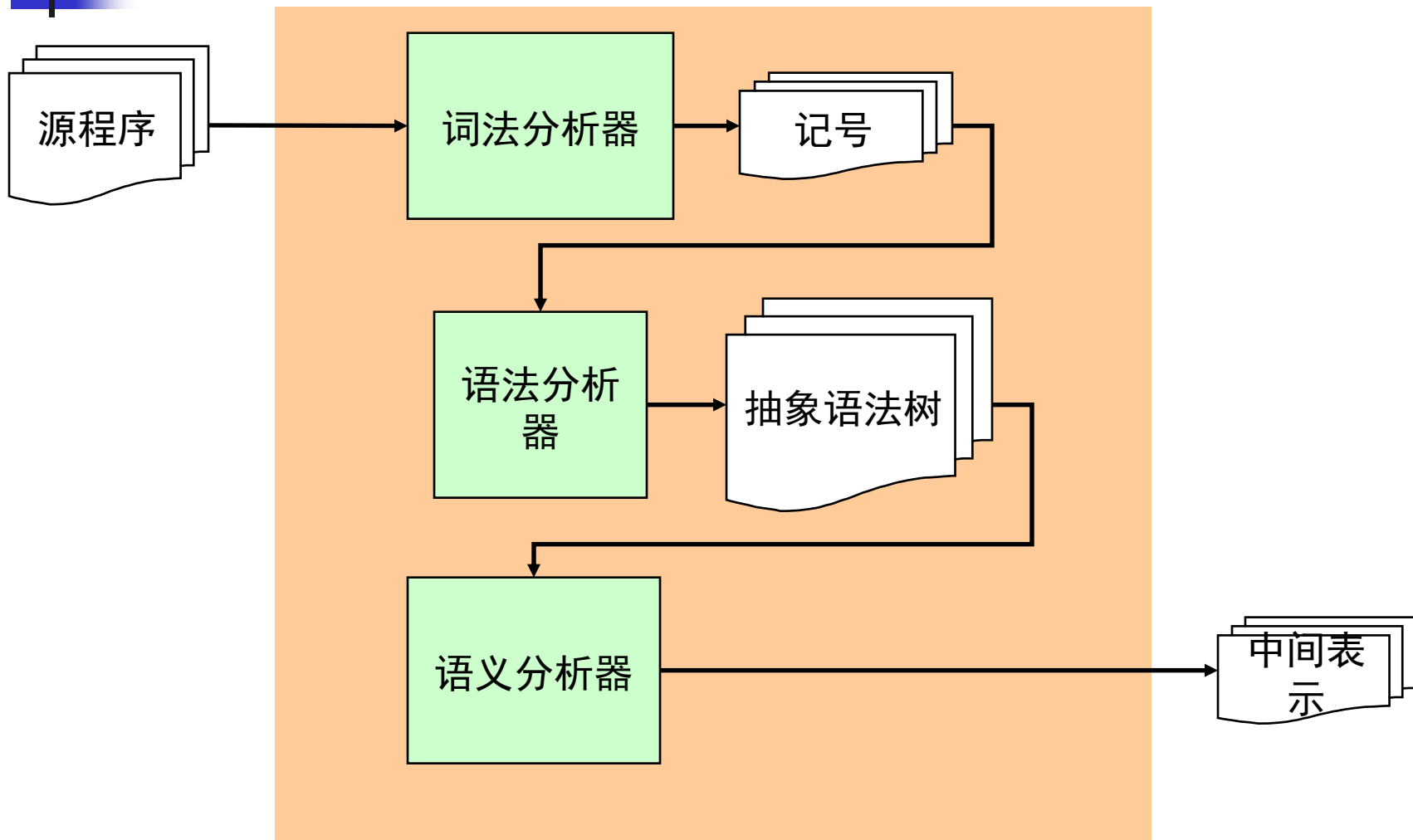
语法分析： 分析树与二义性

编译原理

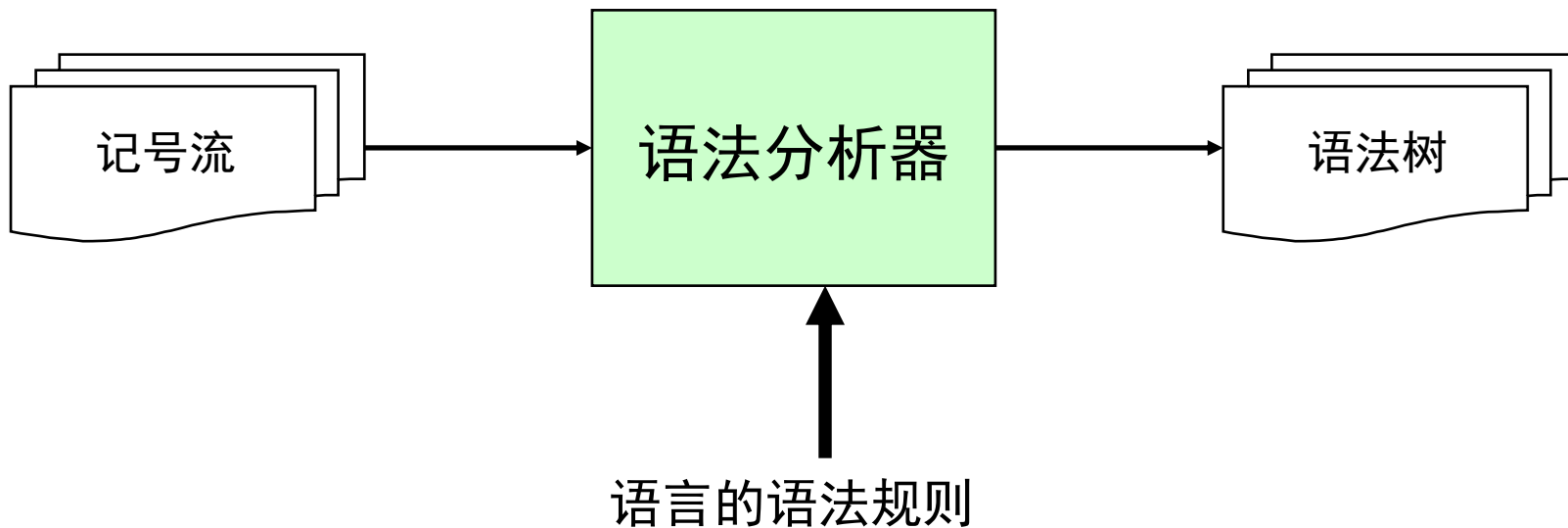
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务





推导与分析树

$S \rightarrow N V N$

$N \rightarrow s$

| t

| g

| w

$V \rightarrow e$

| d



分析树

- 推导可以表达成树状结构
 - 和推导所用的顺序无关（最左、最右、其他）
- 特点：
 - 树中的每个内部节点代表非终结符
 - 每个叶子节点代表终结符
 - 每一步推导代表如何从双亲节点生成它的直接孩子节点

表达式的例子

```
E -> num
    | id
    | E + E
    | E * E
```

```
E -> E + E
    -> 3 + E
    -> 3 + E * E
    -> 3 + 4 * E
    -> 3 + 4 * 5

E -> E * E
    -> E + E * E
    -> 3 + E * E
    -> 3 + 4 * E
    -> 3 + 4 * 5
```

推导这个句子

3+4*5

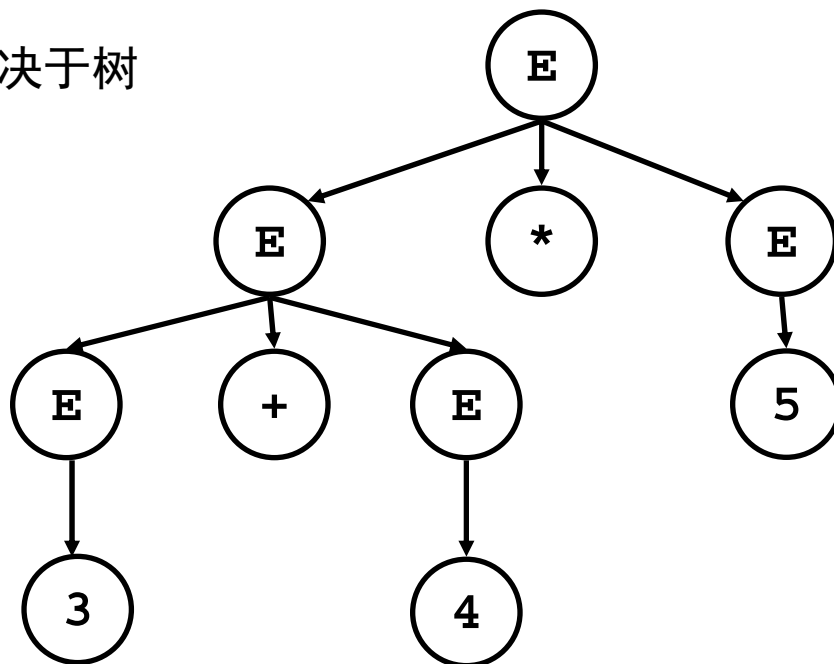
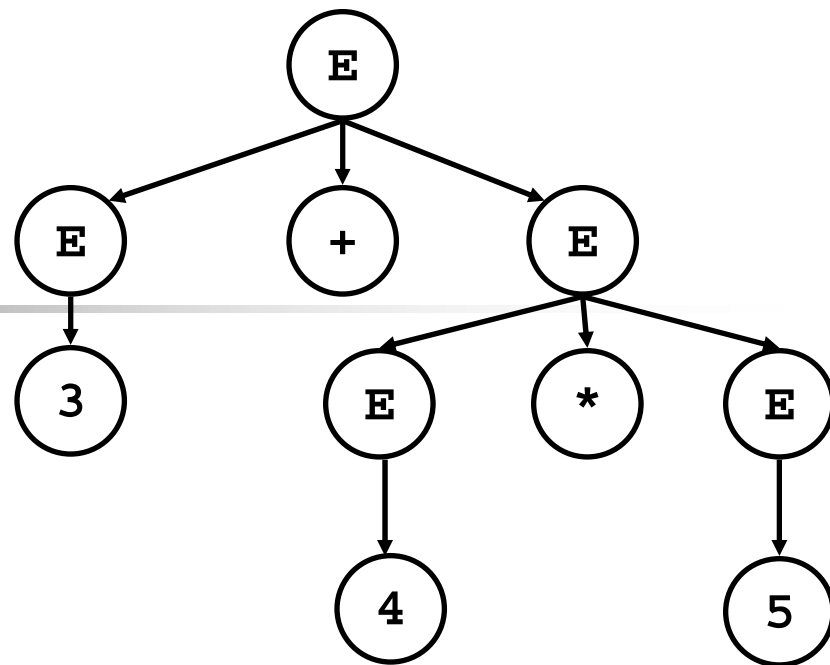
分析树

```
E -> num
    | id
    | E + E
    | E * E
```

```
E -> E + E
-> 3 + E
-> 3 + E * E
-> 3 + 4 * E
-> 3 + 4 * 5

E -> E * E
-> E + E * E
-> 3 + E * E
-> 3 + 4 * E
-> 3 + 4 * 5
```

分析树的含义取决于树的
后序遍历。





二义性文法

- 给定文法G，如果存在句子s，它有两棵不同的分析树，那么称G是二义性文法
- 从编译器角度，二义性文法存在问题：
 - 同一个程序会有不同的含义
 - 因此程序运行的结果不是唯一的
- 解决方案：文法的重写

表达式文法重写

```
E -> E + T
    | T
T -> T * F
    | F
F -> num
    | id
```

```
E -> E + T
-> T + T
-> F + T
-> 3 + T
-> 3 + T * F
-> 3 + F * F
-> 3 + 4 * F
-> 3 + 4 * 5
```

推导这个句子

3+4*5

表达式文法重写

```
E -> E + T
    | T
T -> T * F
    | F
F -> num
    | id
```

```
E -> E + T
-> E + T + T
-> T + T + T
-> F + T + T
-> 3 + T + T
-> 3 + F + T
-> 3 + 4 + T
-> 3 + 4 + F
-> 3 + 4 + 5
```

推导这个句子

3+4+5



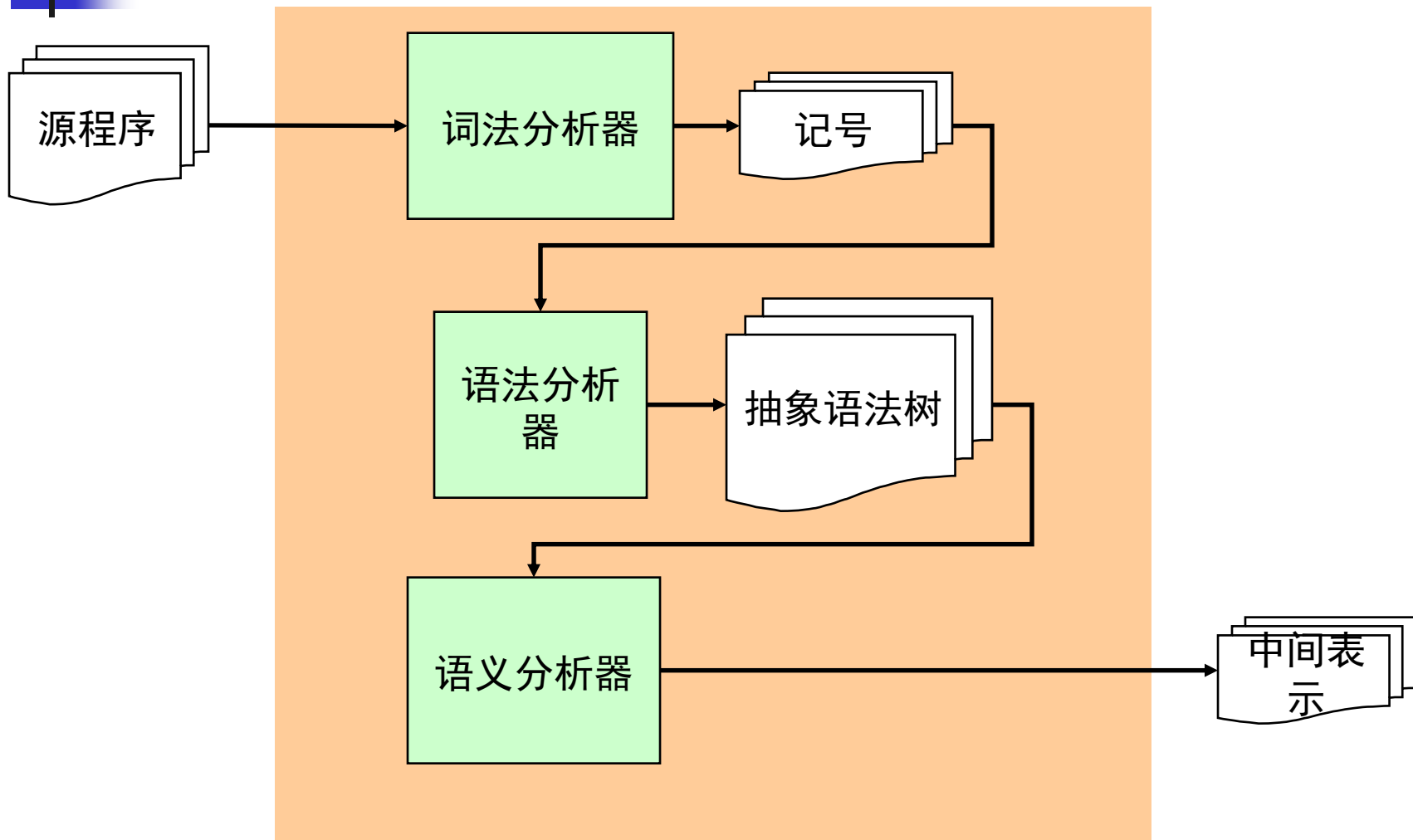
语法分析： 自顶向下分析

编译原理

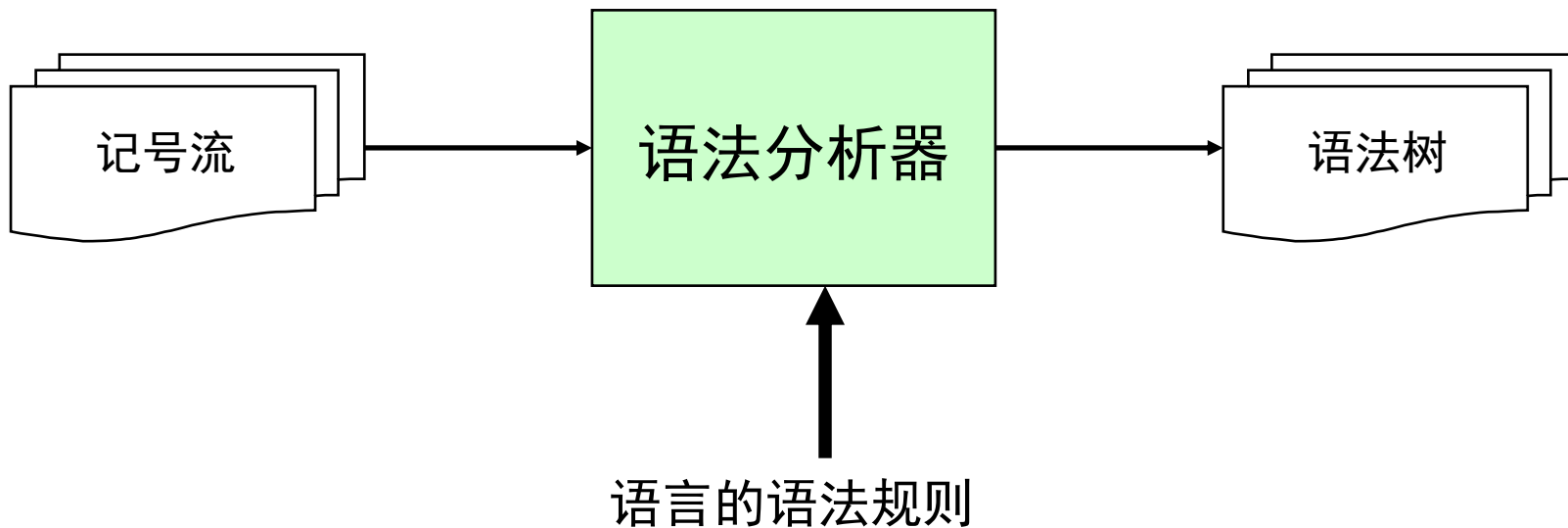
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务





自顶向下分析的算法思想

- 语法分析：给定文法 G 和句子 s ，回答 s 是否能够从 G 推导出来？
- 基本算法思想：从 G 的开始符号出发，随意推导出某个句子 t ，比较 t 和 s
 - 若 $t=s$ ，则回答“是”
 - 若 $t \neq s$ ，则？
- 因为这是从开始符号出发推出句子，因此称为自顶向下分析
 - 对应于分析树自顶向下的构造顺序



示例

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

推导这个句子

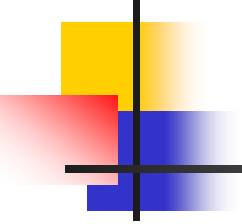


g d w



算法

```
tokens[];    // all tokens
i=0;
stack = [S]  // s是开始符号
while (stack != [])
    if (stack[top] is a terminal t)
        if (t==tokens[i++])
            pop();
        else backtrack();
    else if (stack[top] is a nonterminal T)
        pop(); push(the next right hand side of T)
```



```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

```
tokens[];    // holding all tokens
i=0;
stack = [S]  // s是开始符号
while (stack != [])
    if (stack[top] is a terminal t)
        if (t==tokens[i++])
            pop();
        else backtrack();
    else if (stack[top] is a nonterminal T)
        pop(); push(the next right hand side of T)
```

推导这个句子



g d w



算法的讨论

- 算法需要用到回溯
 - 给分析效率带来问题
- 而就这部分而言（就所有部分），编译器必须高效
 - 编译上千万行的内核等程序
- 因此，实际上我们需要线性时间的算法
 - 避免回溯
 - 引出递归下降分析算法和LL(1)分析算法



重新思考示例

- 用前看符号避免回溯

S -> N V N

N -> s

| t

| g

| w

V -> e

| d

推导这个句子



g d w



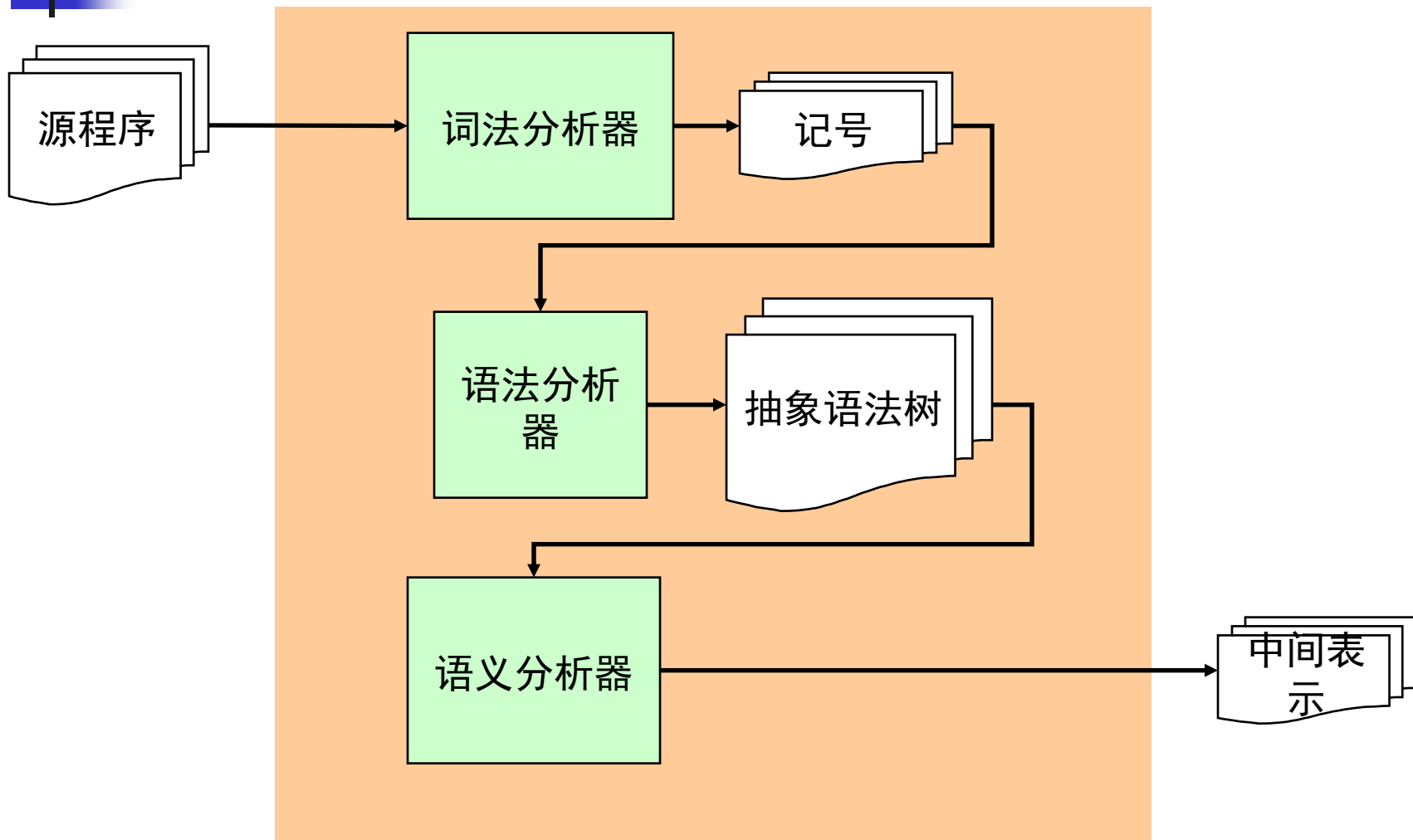
语法分析： 递归下降分析

编译原理

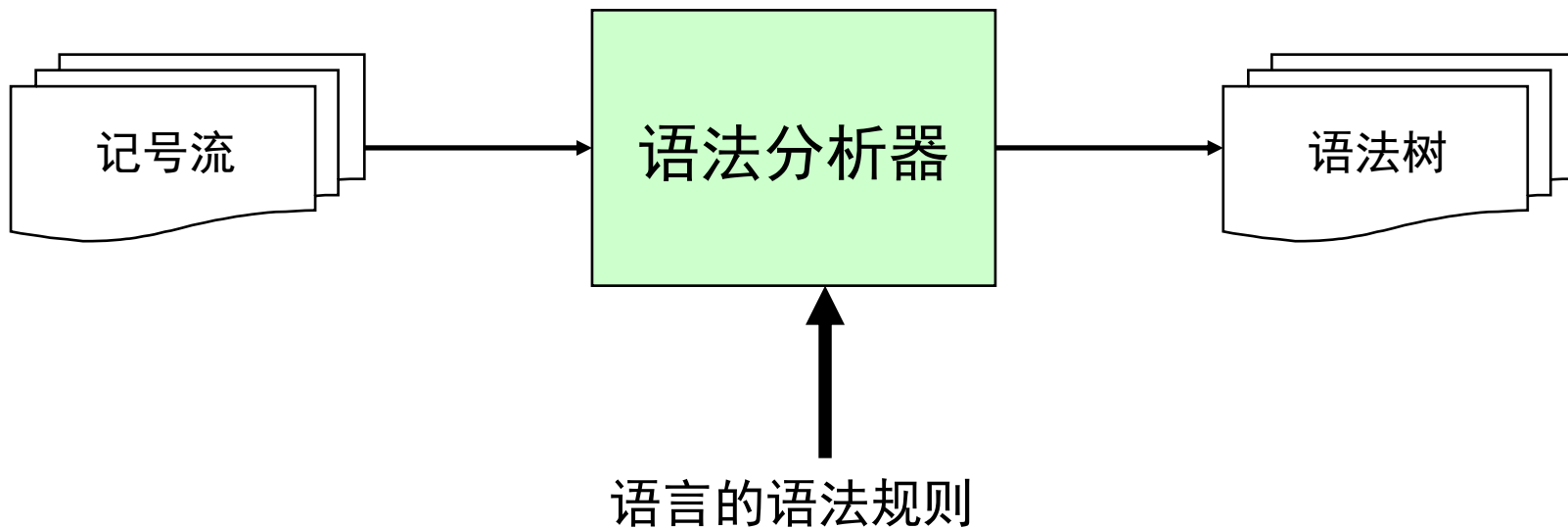
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务





递归下降分析算法

- 也称为预测分析
 - 分析高效（线性时间）
 - 容易实现（方便手工编码）
 - 错误定位和诊断信息准确
 - 被很多开源和商业的编译器所采用
 - GCC 4.0, LLVM, ...
- 算法基本思想：
 - 每个非终结符构造一个分析函数
 - 用前看符号指导产生式规则的选择



示例

```
S -> N V N
N -> s
    | t
    | g
    | w
V -> e
    | d
```

推导这个句子



g d w



算法

```
parse_S()  
    parse_N()  
    parse_V()  
    parse_N()
```

```
parse_N()  
    token = tokens[i++]  
    if (token==s || token==t ||  
        token==g || token==w)  
        return;  
    error("...");
```

```
parse_V()  
    token = tokens[i++]  
    ...// leave this part to you ☺
```

```
S -> N V N  
N -> s  
    | t  
    | g  
    | w  
V -> e  
    | d
```

推导这个句子



g d w



一般的算法框架

```
parse_X()  
    token = nextToken()  
    switch(token)  
    case ...: //  $\beta_{11} \dots \beta_{1i}$   
    case ...: //  $\beta_{21} \dots \beta_{2j}$   
    case ...: //  $\beta_{31} \dots \beta_{3k}$   
    ...  
    default: error ("...");
```

```
x ->  $\beta_{11} \dots \beta_{1i}$   
      |  $\beta_{21} \dots \beta_{2j}$   
      |  $\beta_{31} \dots \beta_{3k}$   
      | ...
```

对算术表达式的递归下降分析

```
// a first try
parse_E()
    token = tokens[i++]
    if (token==num)
        ? // E+T or T
    else error("...");
```

```
E -> E + T
    | T
T -> T * F
    | F
F -> num
```

对这个句子做语法分析

3+4*5

对算术表达式的递归下降分析

```
// a second try
parse_E()
    parse_T()
    token = tokens[i++]
    while (token == '+')
        parse_T()
        token = tokens[i++]

parse_T()
    parse_F()
    token = tokens[i++]
    while (token == '*')
        parse_F()
        token = tokens[i++]
```

```
E -> E + T
    | T
T -> T * F
    | F
F -> num
```

对这个句子做语法分析

3+4*5



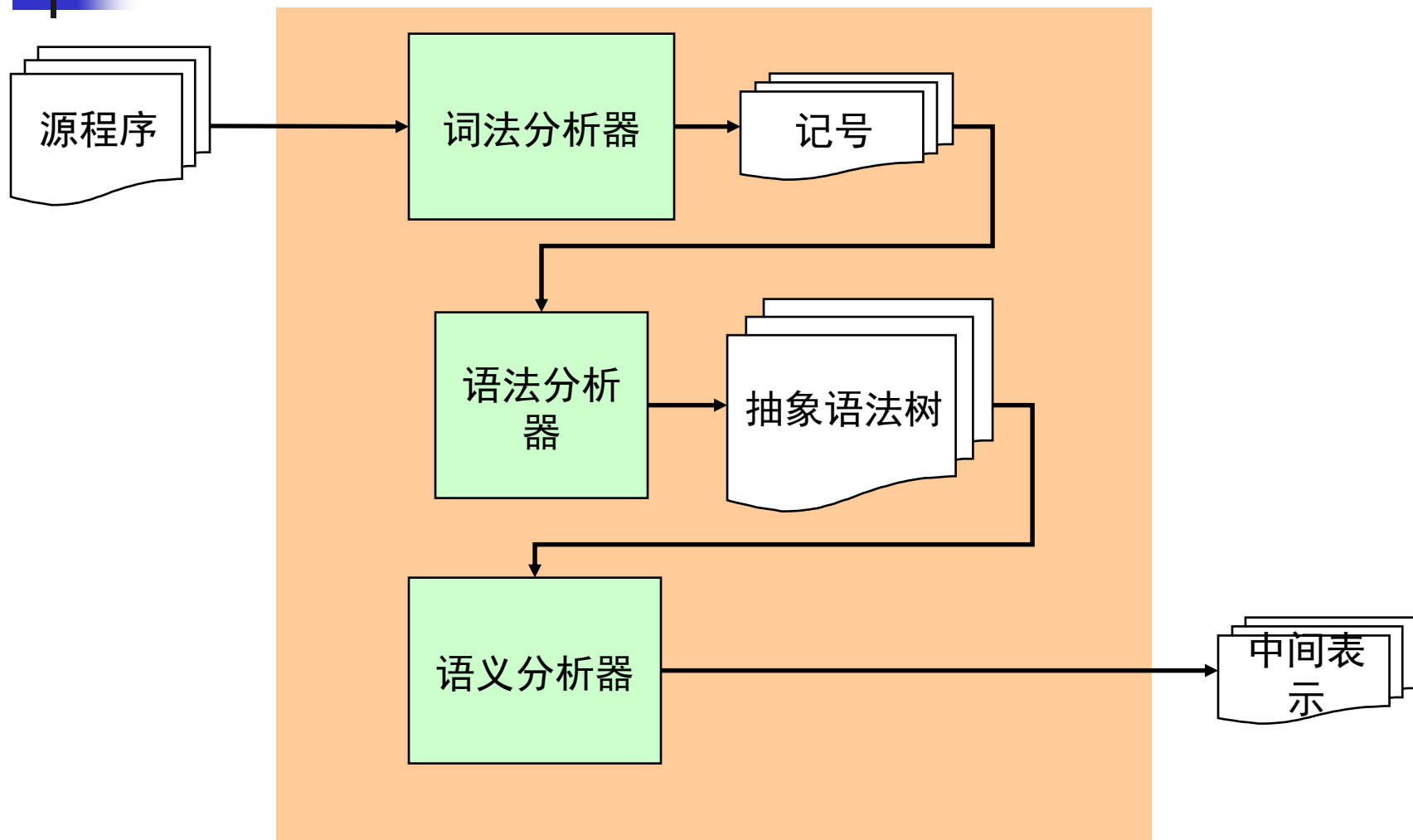
语法分析： LL(1)分析算法

编译原理

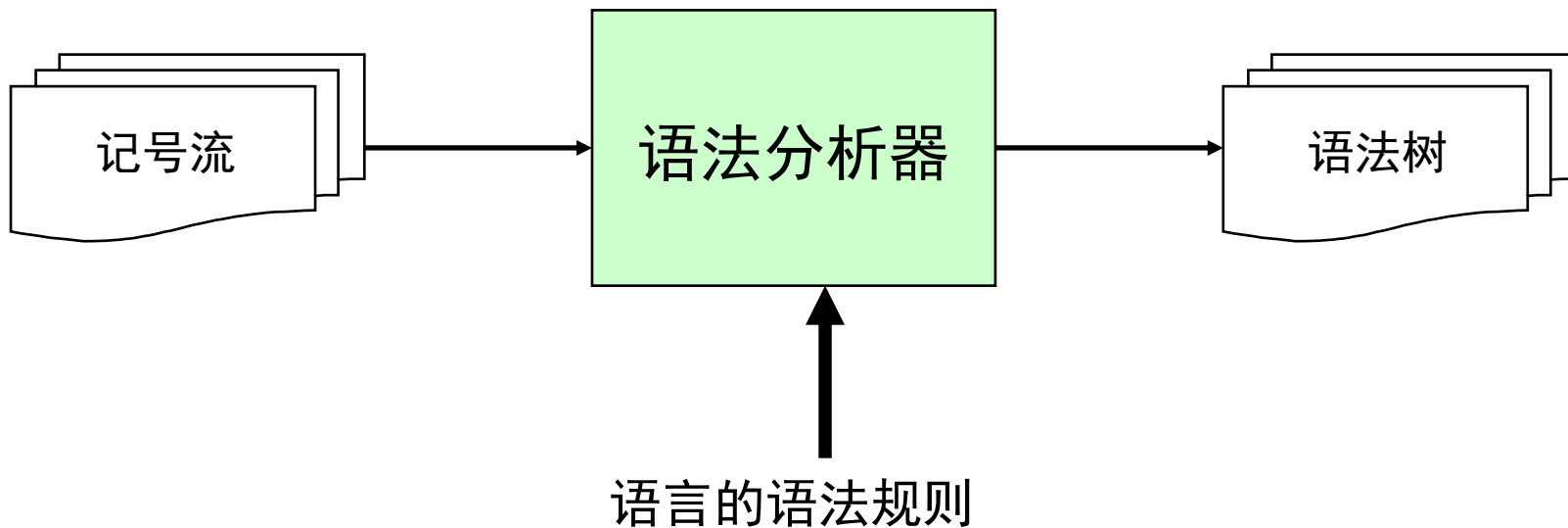
华保健

bjhua@ustc.edu.cn

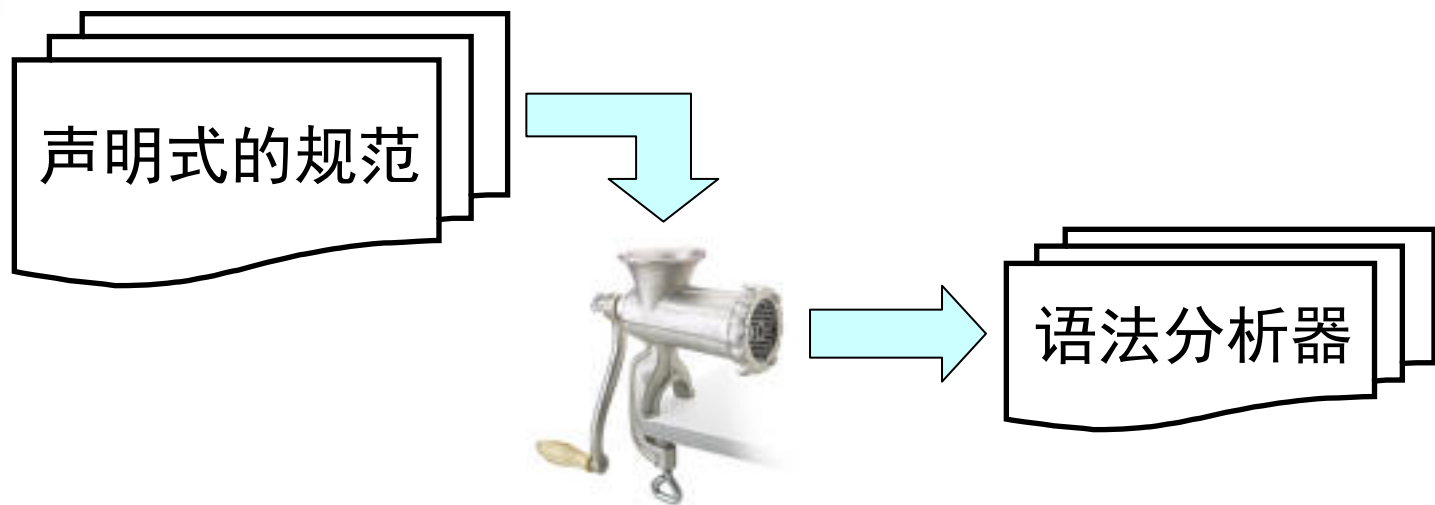
前端



语法分析器的任务



自动生成

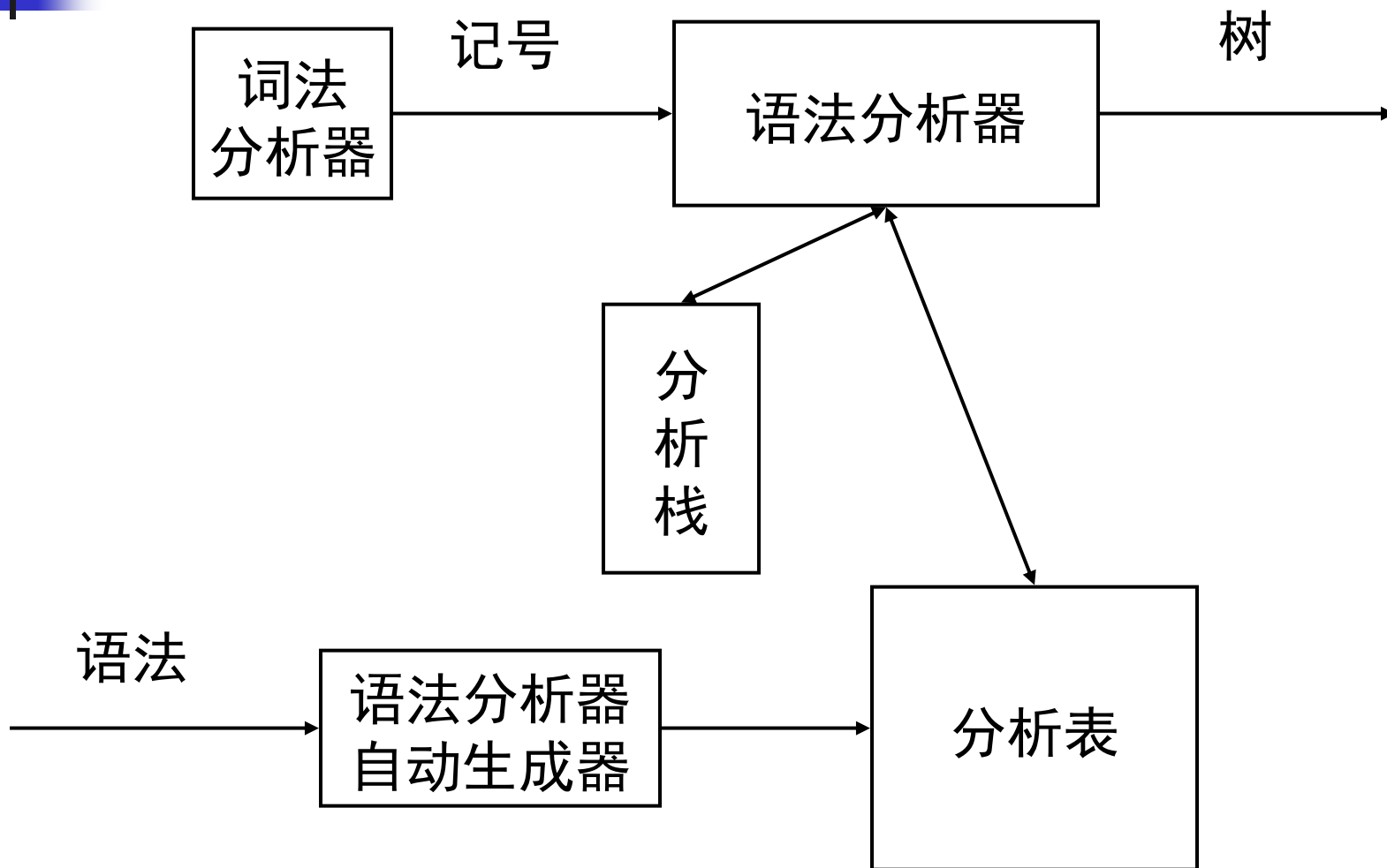




LL(1)分析算法

- 从左（L）向右读入程序，最左（L）推导，采用一个（1）前看符号
 - 分析高效（线性时间）
 - 错误定位和诊断信息准确
 - 有很多开源或商业的生成工具
 - ANTLR, ...
- 算法基本思想：
 - 表驱动的分析算法

表驱动的LL分析器架构



回顾：自顶向下分析算法

```
tokens[];    // all tokens
```

```
i=0;
```

```
stack = [S]  // s是开始符号
```

```
while (stack != [])
```

```
    if (stack[top] is a terminal t)
```

```
        if (t==tokens[i++])
```

```
            pop();
```

```
        else backtrack(); error(...)
```

```
    else if (stack[top] is a nonterminal T)
```

```
        pop(); push(the next right hand side of T)
```

correct

0: S -> N V N

1: N -> s

2: | t

3: | g

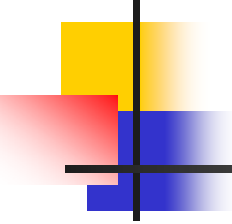
4: | w

5: V -> e

6: | d

分析这个句子

g d w



N\T	s	t	g	w	e	d
S	0	0	0	0		
N	1	2	3	4		
V					5	6

```

tokens[];    // all tokens
i=0;
stack = [S]  // s是开始符号
while (stack != [])

```

```

    if (stack[top] is a terminal t)
        if (t==tokens[i++])
            pop();

```

```

    else backtrack(), error(...)
else if (stack[top] is a nonterminal T)
    pop(); push(the next right hand side of T)
               correct table[N, T]

```

```

0: S -> N V N
1: N -> s
2:   | t
3:   | g
4:   | w
5: V -> e
6:   | d

```

分析这个句子

g d w

FIRST集

// 定义:

// $\text{FIRST}(N)$ = 从非终结符 N 开始推
// 导得出的句子开头的
// 所有可能终结符集合

// 计算公式 (第一个版本, 近似!) :

对 $N \rightarrow a \dots$

$\text{FIRST}(N) \cup = \{a\}$

对 $N \rightarrow M \dots$

$\text{FIRST}(N) \cup = \text{FIRST}(M)$

0: $S \rightarrow N V N$

1: $N \rightarrow s$

2: | t

3: | g

4: | w

5: $V \rightarrow e$

6: | d

推导这个句子

$g d w$



FIRST集的不动点算法

```
foreach (nonterminal N)
```

```
    FIRST(N) = {}
```

```
while(some set is changing)
```

```
    foreach (production p: N → β1 ... βn)
```

```
        if (β1 == a ...)
```

```
            FIRST(N) ∪= {a}
```

```
        if (β1 == M ...)
```

```
            FIRST(N) ∪= FIRST(M)
```

```
0: S → N V N
```

```
1: N → s
```

```
2:   | t
```

```
3:   | g
```

```
4:   | w
```

```
5: V → e
```

```
6:   | d
```

N\FIRST	0	1	2	3	4	5
S	{}					
N	{}					
V	{}					

把FIRST集推广到任意串上

$\text{FIRST}_S(\beta_1 \dots \beta_n) =$

$\text{FIRST}(N), \quad \text{if } \beta_1 == N;$

$\{a\}, \quad \text{if } \beta_1 == a.$

// 在右侧产生式上标记这个 FIRST_S 集合

0: $S \rightarrow N V N$

1: $N \rightarrow s$

2: $\quad \quad | t$

3: $\quad \quad | g$

4: $\quad \quad | w$

5: $V \rightarrow e$

6: $\quad \quad | d$

$N \backslash \text{FIRST}$	
S	{s, t, g, w}
N	{s, t, g, w}
V	{e, d}

构造LL(1)分析表

N\T	s	t	g	w	e	d
S	0	0	0	0		
N	1	2	3	4		
V					5	6

0: S \rightarrow N V N $\{s, t, g, w\}$
1: N \rightarrow s {s}
2: | t {t}
3: | g {g}
4: | w {w}
5: V \rightarrow e {e}
6: | d {d}

N\FIRST	
S	{s, t, g, w}
N	{s, t, g, w}
V	{e, d}

LL(1)分析表中的冲突

N\T	s	t	g	w	e	d
S	0	0	0	0		
N	1	2	3	4,5		
V					5	6

冲突检测:

对N的两条产生式规则 $N \rightarrow \beta$ 和 $N \rightarrow \gamma$, 要求
 $FIRST_S(\beta) \cap FIRST_S(\gamma) = \{\}$ 。

N\FIRST	
S	{s, t, g, w}
N	{s, t, g, w}
V	{e, d}

```

0: S -> N V N {s,t,g,w}
1: N -> s {s}
2:   | t {t}
3:   | g {g}
4:   | w {w}
5:   | w V {w}
6: V -> e {e}
7:   | d {d}
  
```

一般条件下的LL(1)分析表构造

- 首先研究右侧的例子：
 - FIRST_S(X Y Z)?
 - 一般情况下需要知道某个非终结符是否可以推出空串
 - NULLABLE
 - 并且一般需要知道在某个非终结符后面跟着什么符号
 - 跟随集FOLLOW

```
z -> d
      | x y z
y -> c
      |
x -> y
      | a
```



NULLABLE集合

- 归纳定义：
- 非终结符 X 属于集合NULLABLE，当且仅当：
 - 基本情况：
 - $X \rightarrow$
 - 归纳情况：
 - $X \rightarrow Y_1 \cdots Y_n$
 - Y_1, \cdots, Y_n 是 n 个非终结符，且都属于NULLABLE集



NULLABLE集合算法

```
NULLABLE = {};
```

```
while (NULLABLE is still changing)
```

```
    foreach (production p:  $X \rightarrow \beta$  )
```

```
        if (  $\beta == \varepsilon$  )
```

```
            NULLABLE  $\cup$  = {x}
```

```
        if (  $\beta == Y_1 \dots Y_n$  )
```

```
            if (  $Y_1 \in \text{NULLABLE} \ \&\& \dots \ \&\& \ Y_n \in \text{NULLABLE}$  )
```

```
                NULLABLE  $\cup$  = {x}
```



示例

```
NULLABLE = {};
```

```
while (NULLABLE is still changing)
```

```
    foreach (production p:  $X \rightarrow \beta$  )
```

```
        if (  $\beta == \epsilon$  )
```

```
            NULLABLE  $\cup$  = {X}
```

```
        if (  $\beta == Y_1 \dots Y_n$  )
```

```
            if (  $Y_1 \in \text{NULLABLE} \ \&\& \dots \ \&\& \ Y_n \in \text{NULLABLE}$  )
```

```
                NULLABLE  $\cup$  = {X}
```

```
z -> d
    | x y z
Y -> c
    |
X -> Y
    | a
```



FIRST集合的完整计算公式

- 基于归纳的计算规则：
 - 基本情况：
 - $X \rightarrow a$
 - $\text{FIRST}(X) \cup = \{a\}$
 - 归纳情况：
 - $X \rightarrow Y_1 Y_2 \cdots Y_n$
 - $\text{FIRST}(X) \cup = \text{FIRST}(Y_1)$
 - if $Y_1 \in \text{NULLABLE}$, $\text{FIRST}(X) \cup = \text{FIRST}(Y_2)$
 - if $Y_1, Y_2 \in \text{NULLABLE}$, $\text{FIRST}(X) \cup = \text{FIRST}(Y_3)$
 - ...



FIRST集的不动点算法

```
foreach (nonterminal N)
```

```
    FIRST(N) = {}
```

```
while(some set is changing)
```

```
    foreach (production p:  $N \rightarrow \beta_1 \dots \beta_n$ )
```

```
        foreach ( $\beta_i$  from  $\beta_1$  upto  $\beta_n$ )
```

```
            if ( $\beta_i == a \dots$ )
```

```
                FIRST(N)  $\cup$  = {a}
```

```
                break
```

```
            if ( $\beta_i == M \dots$ )
```

```
                FIRST(N)  $\cup$  = FIRST(M)
```

```
                if (M is not in NULLABLE)
```

```
                    break;
```

FIRST集计算示例

```
foreach (nonterminal N)
```

```
    FIRST(N) = {}
```

```
while(some set is changing)
```

```
    foreach (production p:  $N \rightarrow \beta_1 \dots \beta_n$ )
```

```
        foreach ( $\beta_i$  from  $\beta_1$  upto  $\beta_n$ )
```

```
            if ( $\beta_i == a \dots$ )
```

```
                FIRST(N)  $\cup$  = {a}
```

```
            break
```

```
            if ( $\beta_i == M \dots$ )
```

```
                FIRST(N)  $\cup$  = FIRST(M)
```

```
                if (M is not in NULLABLE)
```

```
                    break;
```

```
z -> d
    | x y z
y -> c
    |
x -> y
    | a
```

N\FIRST	0	1	2
Z	{}		
Y	{}		
X	{}		



FOLLOW集的不动点算法

```
foreach (nonterminal N)
```

```
    FOLLOW(N) = {}
```

```
while(some set is changing)
```

```
    foreach (production p:  $N \rightarrow \beta_1 \dots \beta_n$ )
```

```
        temp = FOLLOW(N)
```

```
        foreach ( $\beta_i$  from  $\beta_n$  downto  $\beta_1$ ) // 逆序!
```

```
            if ( $\beta_i == a \dots$ )
```

```
                temp = {a}
```

```
            if ( $\beta_i == M \dots$ )
```

```
                FOLLOW(M)  $\cup$  = temp
```

```
                if (M is not NULLABLE)
```

```
                    temp = FIRST(M)
```

```
                else temp  $\cup$  = FIRST(M)
```



FOLLOW集计算示例

NULLABLE = {X, Y}

	X	Y	Z
FIRST	{a, c}	{c}	{a, c, d}

N\FOLLOW	0	1	2
Z	{}		
Y	{}		
X	{}		

0: Z -> d

1: | x y z

2: Y -> c

3: |

4: x -> y

5: | a



计算FIRST_S集合

```
foreach (production p)
```

```
    FIRST_S(p) = {}
```

```
calcuete_FIRST_S(production p:  $N \rightarrow \beta_1 \dots \beta_n$ )
```

```
    foreach ( $\beta_i$  from  $\beta_1$  to  $\beta_n$ )
```

```
        if ( $\beta_i == a \dots$ )
```

```
            FIRST_S(p)  $\cup$  = {a}
```

```
            return;
```

```
        if ( $\beta_i == M \dots$ )
```

```
            FIRST_S(p)  $\cup$  = FIRST(M)
```

```
            if (M is not NULLABLE)
```

```
                return;
```

```
    FIRST_S(p)  $\cup$  = FOLLOW(N)
```



示例：构造FIRST_S集

NULLABLE = {X, Y}

	X	Y	Z
FIRST	{a, c}	{c}	{a, c, d}
FOLLOW	{a, c, d}	{a, c, d}	{}

0: z -> d

1: | x y z

2: y -> c

3: |

4: x -> y

5: | a

	0	1	2	3	4	5
FIRST_S	{d}	{a, c, d}	{c}	{a, c, d}	{c, a, d}	{a}

示例：构造LL(1)分析表

	a	c	d
Z	1	1	0, 1
Y	3	2, 3	3
X	4, 5	4	4

0: z -> d

1: | x y z

2: y -> c

3: |

4: x -> y

5: | a

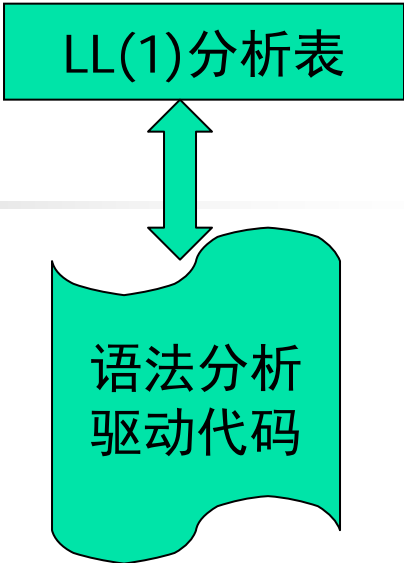
	0	1	2	3	4	5
FIRST_S	{d}	{a, c, d}	{c}	{a, c, d}	{c, a, d}	{a}



LL(1)分析器

```
tokens[];    // all tokens
i=0;
stack = [S]  // s是开始符号
while (stack != [])
    if (stack[top] is a terminal t)
        if (t==tokens[i++])
            pop();
        else error(...);
    else if (stack[top] is a nonterminal T)
        pop()
        push(table[T, tokens[i]])
```

LL(1)分析表



语法分析
驱动代码



语法分析： LL(1)分析冲突处理

编译原理

华保健

bjhua@ustc.edu.cn

LL(1)文法中的冲突

	a	c	d
Z	1	1	0, 1
Y	3	2, 3	3
X	4, 5	4	4

0: $z \rightarrow d$

1: | $x y z$

2: $y \rightarrow c$

3: |

4: $x \rightarrow y$

5: | a

	0	1	2	3	4	5
FIRST_S	{d}	{a, c, d}	{c}	{a, c, d}	{c, a, d}	{a}

另外一个示例

	n	+	*
E	0, 1		
T	2, 3		
F	4		

0: $E \rightarrow E + T$

1: $\quad \mid T$

2: $T \rightarrow T * F$

3: $\quad \mid F$

4: $F \rightarrow n$

	0	1	2	3	4
FIRST_S	{n}	{n}	{n}	{n}	{n}

消除左递归

	n	+	*
E	0		
E'		1	
T	3		
T'		5	4
F	6		

0: $E \rightarrow E + T$

1: $\quad \mid T$

2: $T \rightarrow T * F$

3: $\quad \mid F$

4: $F \rightarrow n$

0: $E \rightarrow T E'$

1: $E' \rightarrow + T E'$

2: $\quad \mid$

3: $T \rightarrow F T'$

4: $T' \rightarrow * F T'$

5: $\quad \mid$

6: $F \rightarrow n$



提取左公因子

0: $X \rightarrow aY$

1: | aZ

2: $Y \rightarrow b$

3: $Z \rightarrow c$

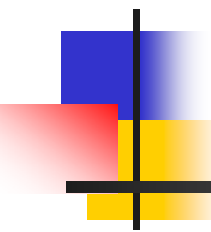
0: $X \rightarrow aX'$

1: $X' \rightarrow Y$

2: | Z

3: $Y \rightarrow b$

4: $Z \rightarrow c$



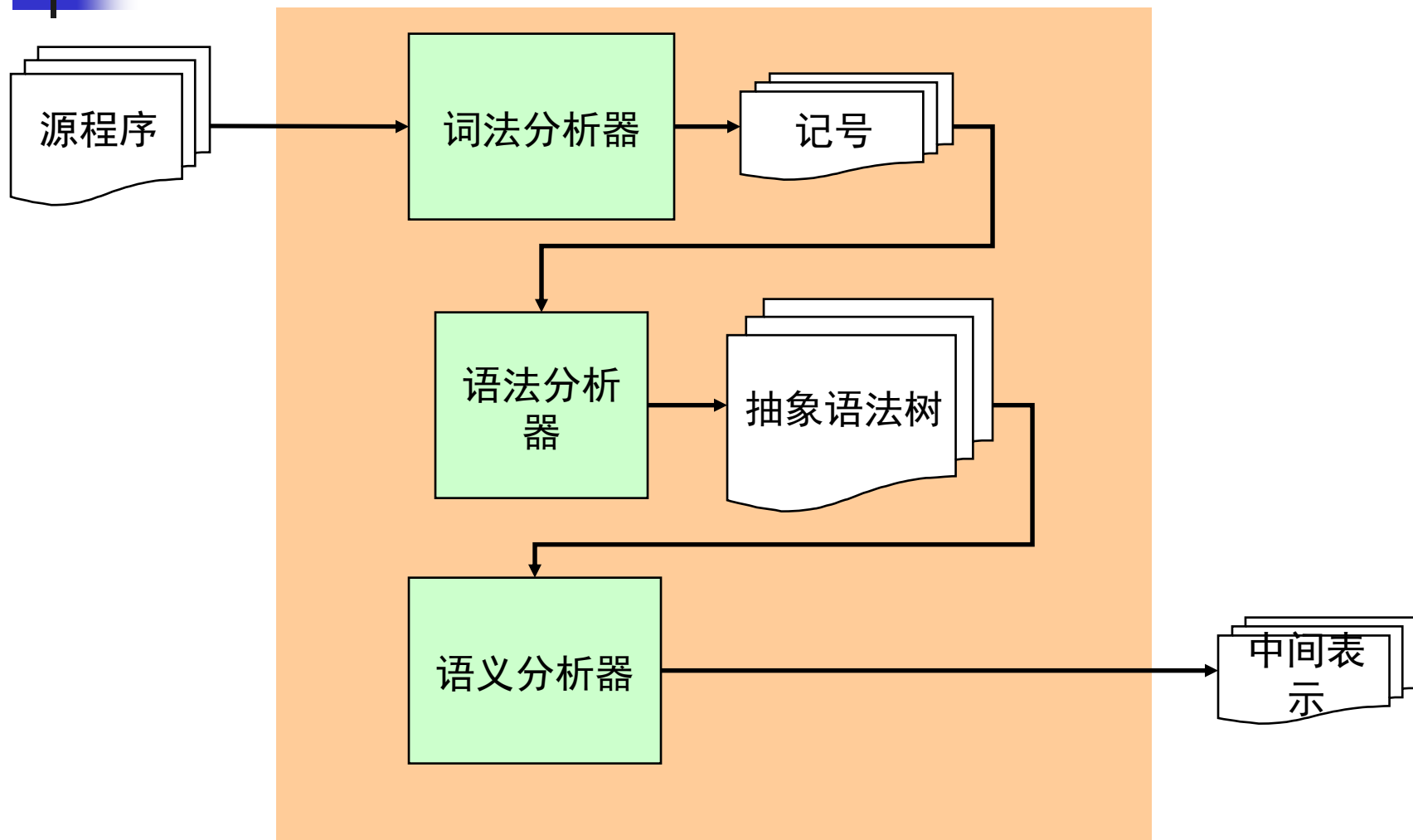
语法分析： LR(0)分析算法

编译原理

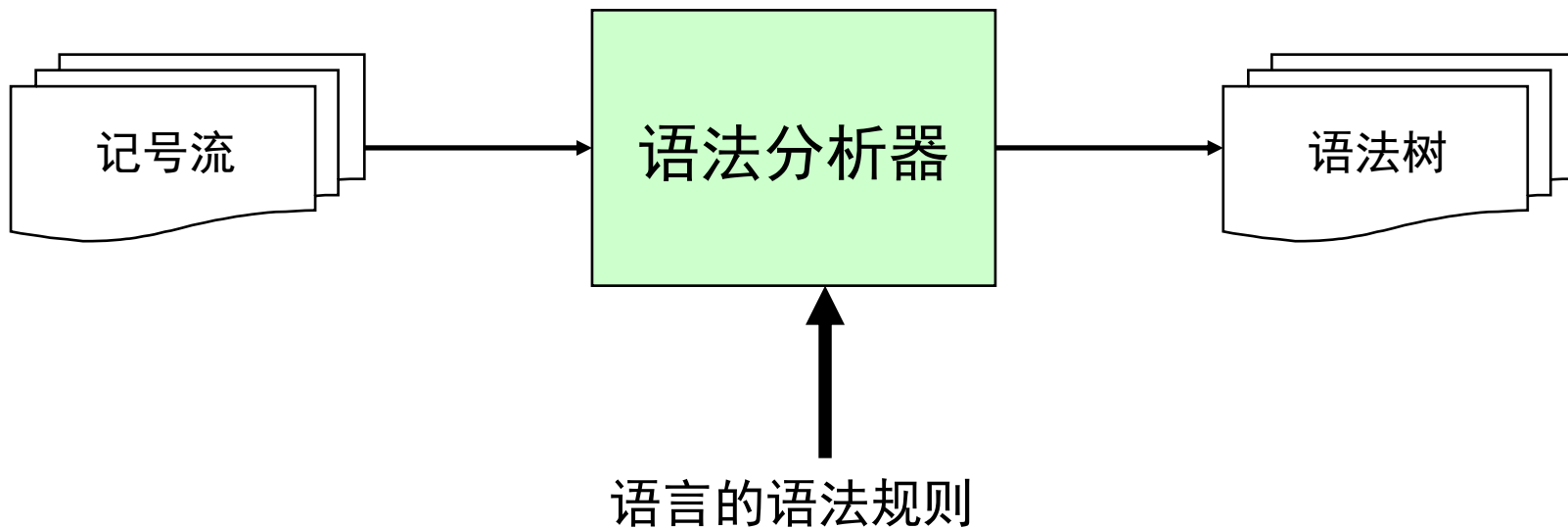
华保健

bjhua@ustc.edu.cn

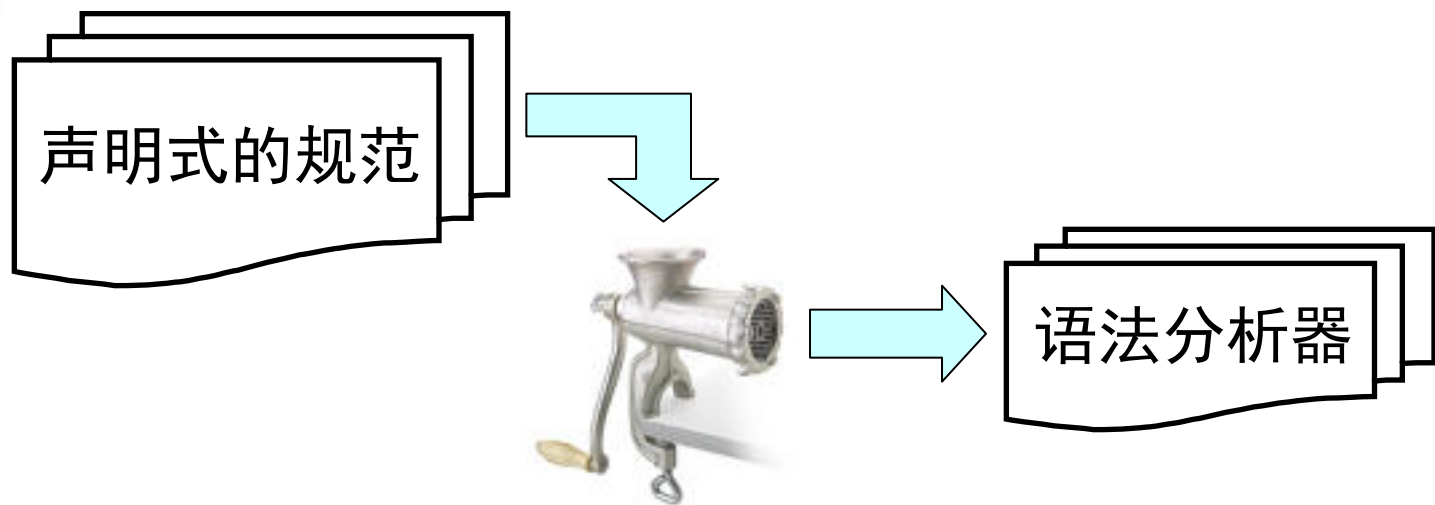
前端



语法分析器的任务



自动生成





LL(1)分析算法

- 从左（L）向右读入程序，最左（L）推导，采用一个（1）前看符号
 - 优点：
 - 算法运行高效
 - 有现成的工具可用
 - 缺点：
 - 能分析的文法类型受限
 - 往往需要文法的改写



自底向上分析算法

- 研究其中最重要也是最广泛应用的一类
 - LR分析算法（移进-归约算法）
 - 算法运行高效
 - 有现成的工具可用
 - 这也是目前应该广泛的一类语法分析器的自动生成器中采用的算法
 - YACC, bison, CUP, C#yacc, 等

自底向上分析的基本思想

0: $S \rightarrow E$

1: $E \rightarrow E + T$

2: $\quad \mid T$

3: $T \rightarrow T * F$

4: $\quad \mid F$

5: $F \rightarrow n$

最右推导的逆过程!

2 + 3 * 4

F + 3 * 4

T + 3 * 4

E + 3 * 4

E + F * 4

E + T * 4

E + T * F

E + T

E

S



点记号

- 为了方便标记语法分析器已经读入了多少输入，我们可以引入一个点记号 •

$E + 3 \bullet * 4$

已经读入的

剩余的输入



自底向上分析

2 + 3 * 4

F + 3 * 4

T + 3 * 4

E + 3 * 4

E + F * 4

E + T * 4

E + T * F

E + T

E

S

2 ● + 3 * 4

F ● + 3 * 4

T ● + 3 * 4

E + 3 ● * 4

E + F ● * 4

E + T * 4 ●

E + T * F ●

E + T ●

E ●

S ●

另外的写法

	2	+	3	*	4
	●	+	3	*	4
F	●	+	3	*	4
T	●	+	3	*	4
E	●	+	3	*	4
E +	●	3	*	4	
E + 3	●	*	4		
E + F	●	*	4		
E + T	●	*	4		
E + T *	●	4			
E + T * 4	●				
E + T * F	●				
E + T	●				
E	●				
S	●				

```

0: S -> E
1: E -> E + T
2:   | T
3: T -> T * F
4:   | F
5: F -> n
    
```

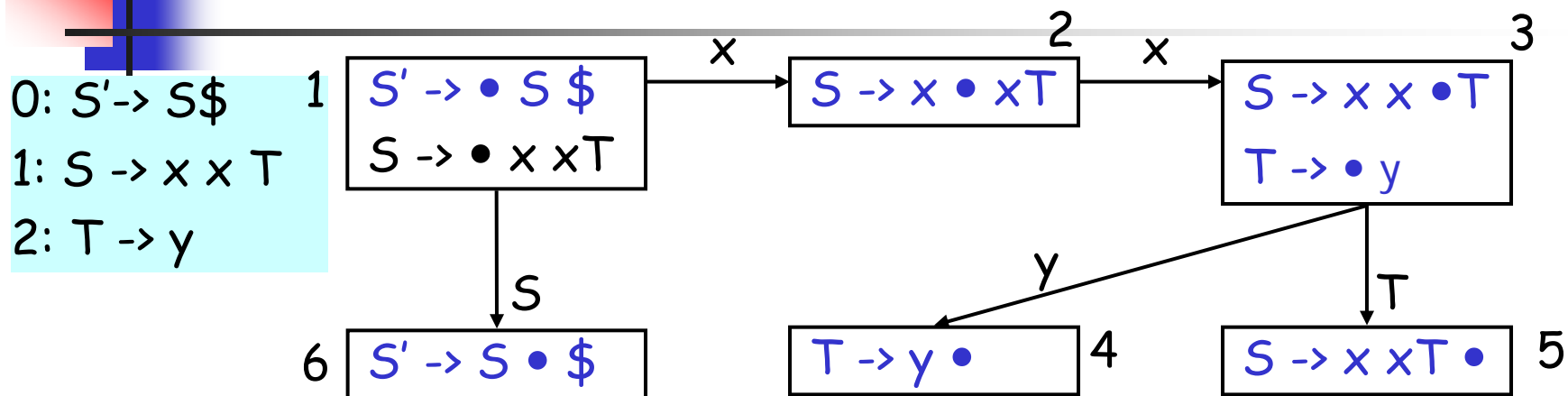
左方是什么数据结构？



生成一个逆序的最右推导

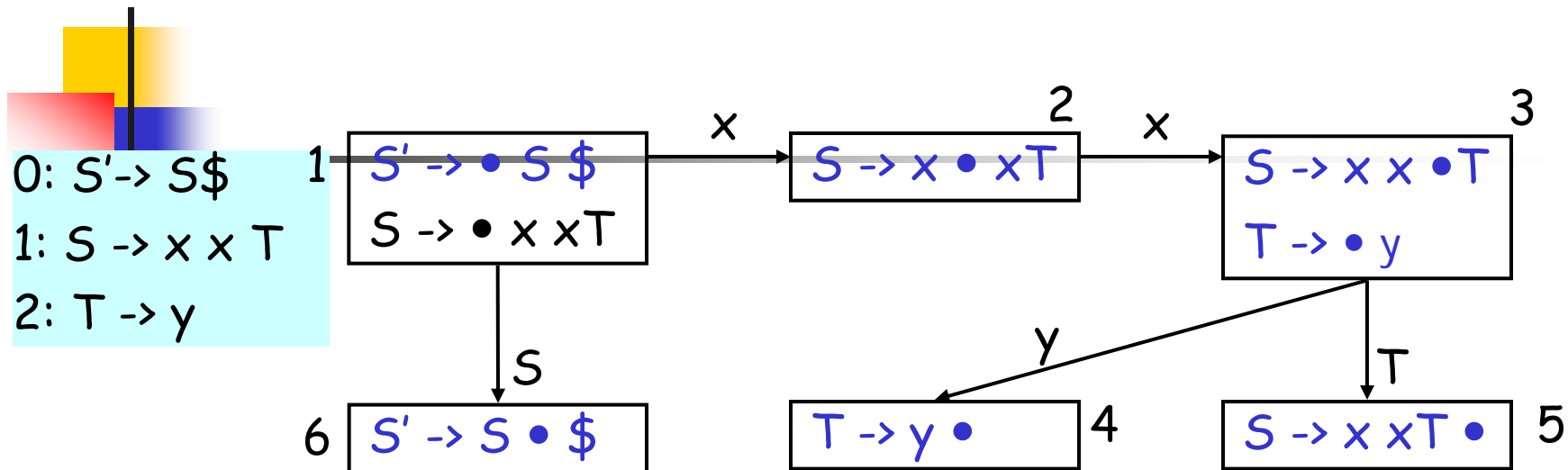
- 需要两个步骤：
 - **移进** 一个记号到栈顶上，或者
 - **归约** 栈顶上的 n 个符号（某产生式的右部）到左部的非终结符
 - 对产生式 $A \rightarrow \beta_1 \dots \beta_n$
 - 如果 $\beta_n \dots \beta_1$ 在栈顶上，则弹出 $\beta_n \dots \beta_1$
 - 压入 A
- 核心的问题：如何确定**移进**和**归约**的时机？

算法思想



分析这个输入串: $x x y \$$

LR(0)分析表

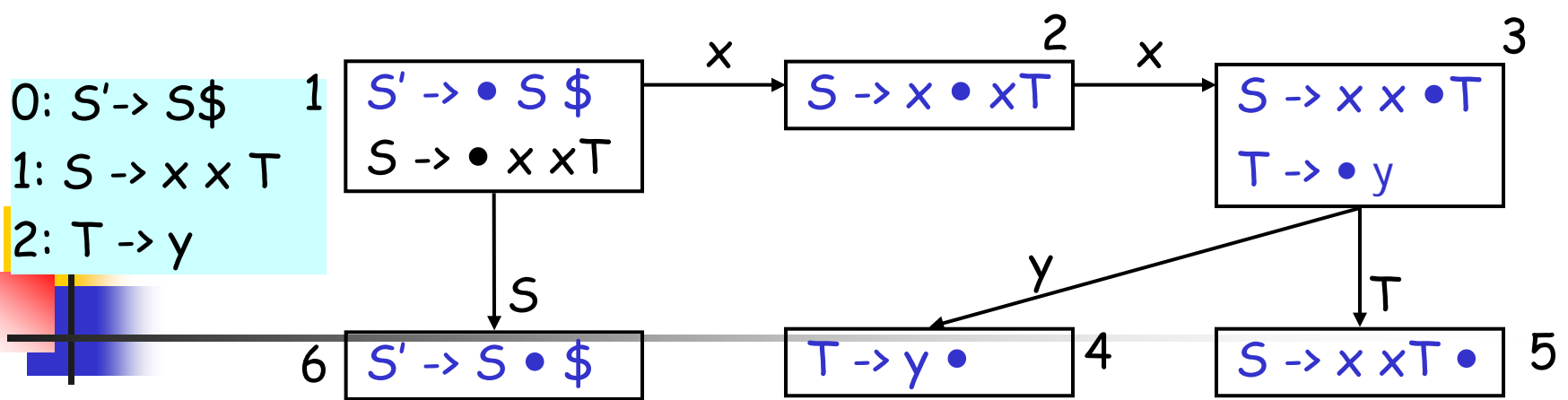


状态\符号	动作 (ACTION)			转移 (GOTO)	
	x	y	\$	S	T
1	s2			g6	
2	s3				
3		s4			g5
4	r2	r2	r2		
5	r1	r1	r1		
6			accept		



LR(0)分析算法

```
stack = []
push ($)          // $: end of file
push (1)          // 1: initial state
while (true)
    token t = nextToken()
    state s = stack[top]
    if (ACTION[s, t] == "si")
        push (t); push (i)
    else if (ACTION[s, t] == "rj")
        pop (the right hand of production "j:  $x \rightarrow \beta$ ")
        state s = stack[top]
        push (X); push (GOTO[s, X])
    else error (...)
```



分析这个输入串: $x x y \$$

```
stack = []
push ($)      // $: end of file
push (1)      // 1: initial state
while (true)
    token t = nextToken()
    state s = stack[top]
    if (ACTION[s, t] == "si")
        push (t); push (i)
    else if (ACTION[s, t] == "rj")
        pop (the right hand of production "j:  $x \rightarrow \beta$ ")
        state s = stack[top]
        push (X); push (GOTO[s, X])
    else error (...)
```



LR(0)分析表构造算法

```
C0 = closure (S' -> • S $) // the init closure
SET = {C0} // all states
Q = enqueue(C0) // a queue
while (Q is not empty)
    C = dequeue (Q)
    foreach (x ∈ (N ∪ T))
        D = goto (C, x)
        if (x ∈ T)
            ACTION[C, x] = D
        else GOTO[C, x] = D
        if (D not ∈ SET)
            SET ∪= {D}
            enqueue (D)
```



goto和closure

```
goto (C, x)
  temp = {}           // a set
  foreach (C's item i: A ->  $\beta \bullet x \gamma$ )
    temp U= {A ->  $\beta x \bullet \gamma$ }
  return closure (temp)

closure (C)
  while(C is still changing)
    foreach (C's item i: A ->  $\beta \bullet B \gamma$ )
      C U= {B -> ...}
```



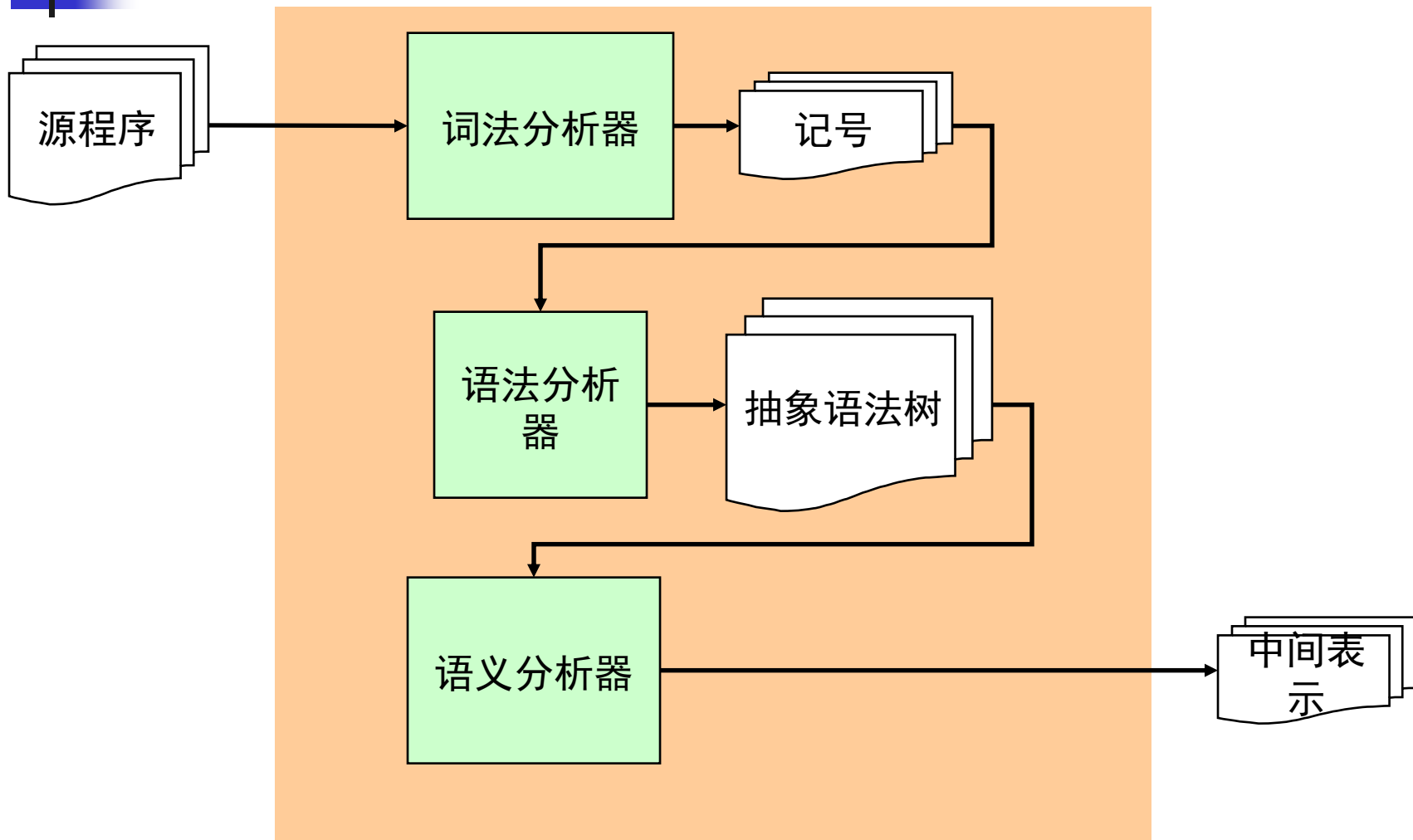
语法分析： SLR分析算法

编译原理

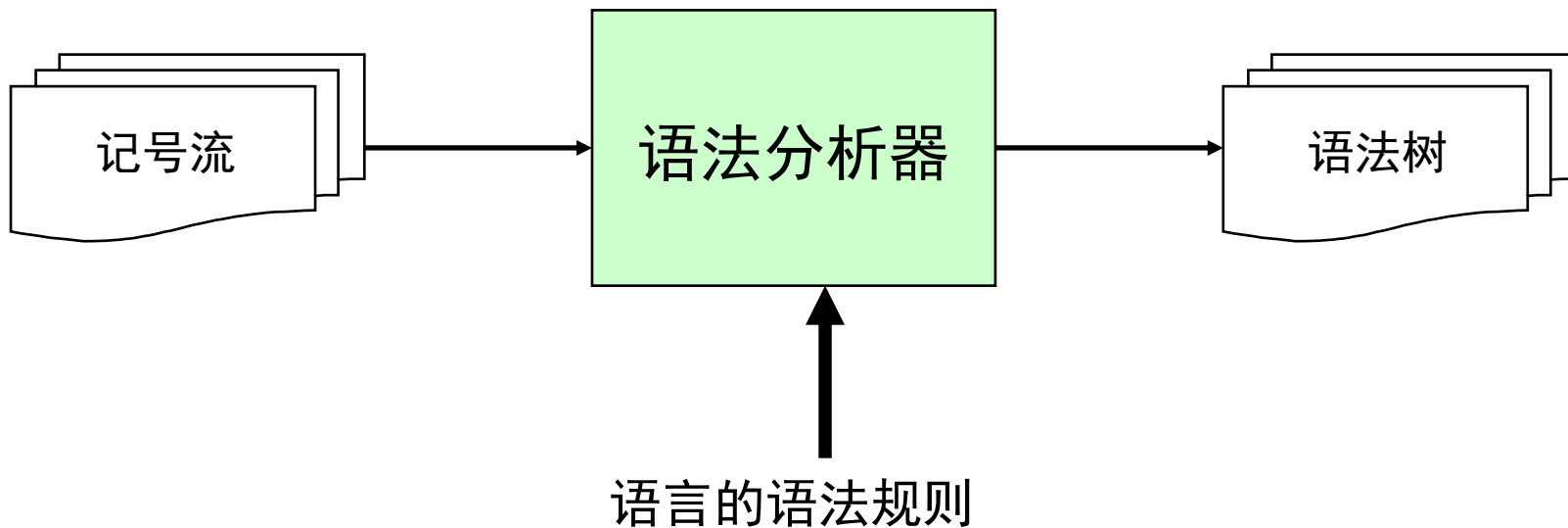
华保健

bjhua@ustc.edu.cn

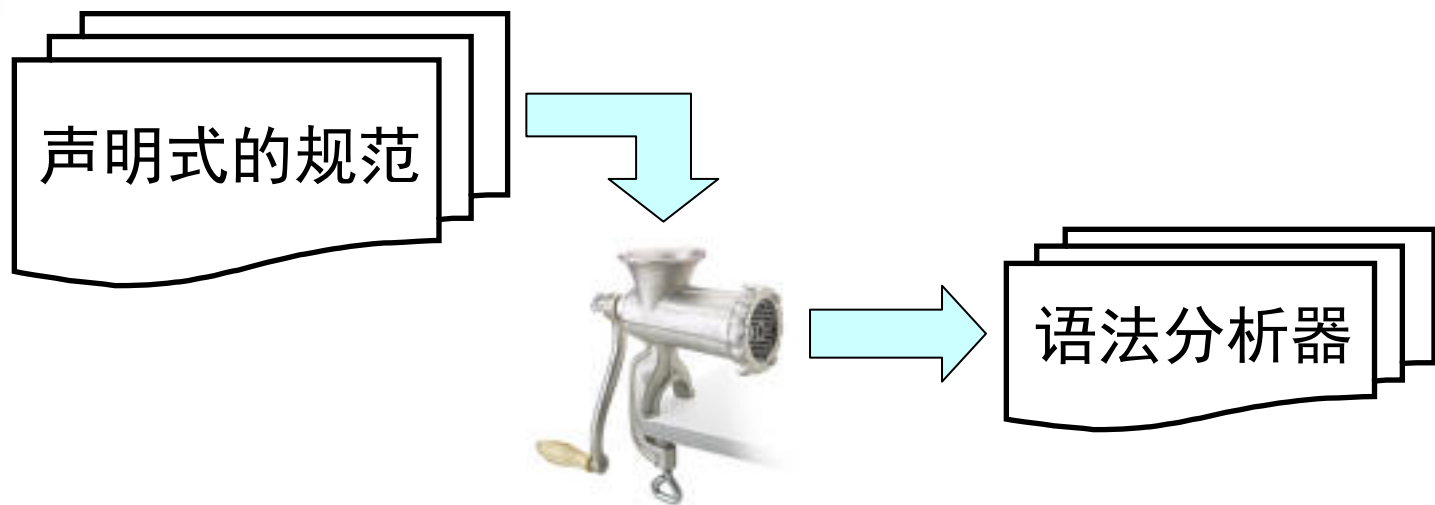
前端



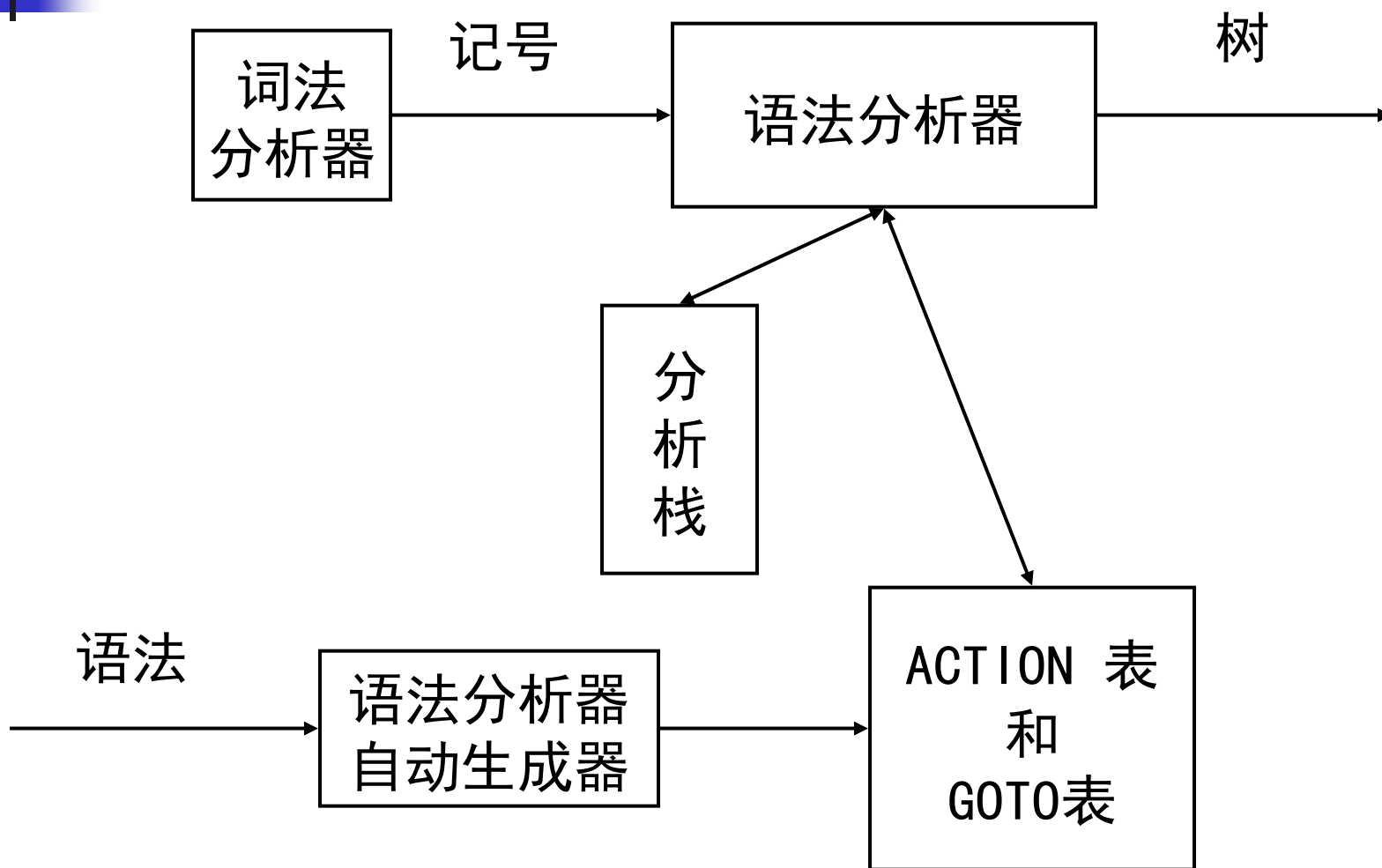
语法分析器的任务



自动生成



表驱动的LR分析器架构





LR(0)分析算法

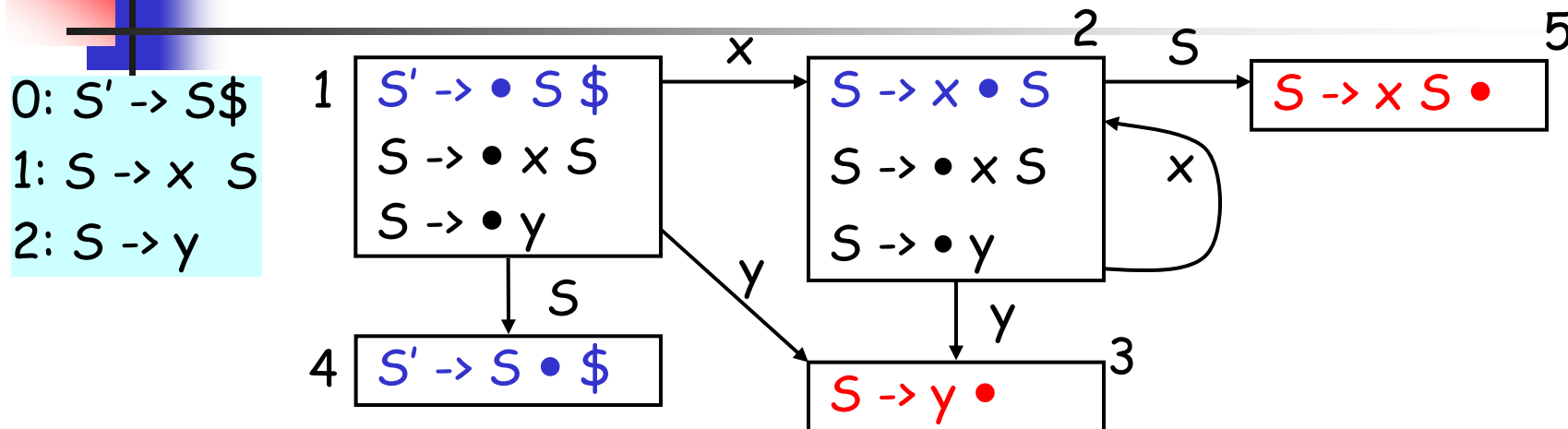
- 从左（L）向右读入程序，最右（L）推导，不用前看符号来决定产生式的选择（0个前看符号）
 - 优点：
 - 容易实现
 - 缺点：
 - 能分析的文法有限



LR(0)分析算法的缺点

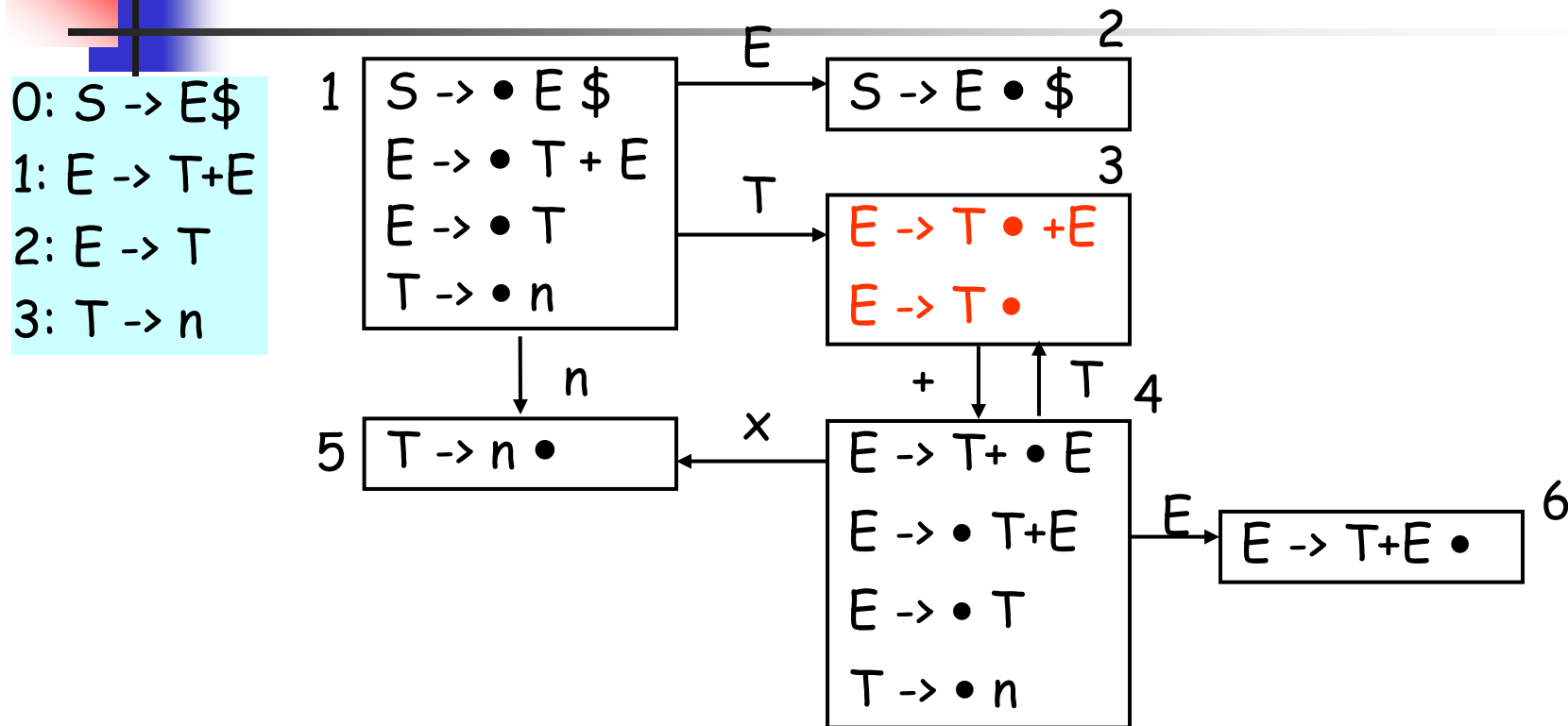
- 对每一个形如 $X \rightarrow \alpha \bullet$ 的项目
 - 直接把 α 归约成 X , 紧跟一个 “goto”
 - 尽管不会漏掉错误, 但会延迟错误发现时机
 - 练习: 尝试 “x x y x”
- LR(0)分析表中可能包含冲突

问题1：错误定位



	ACTION			GOTO
状态\符号	x	y	\$	S
1	s2	s3		g4
2	s2	s3		g5
3	r2	r2	r2	
4			accept	
5	r1	r1	r1	

问题2：冲突



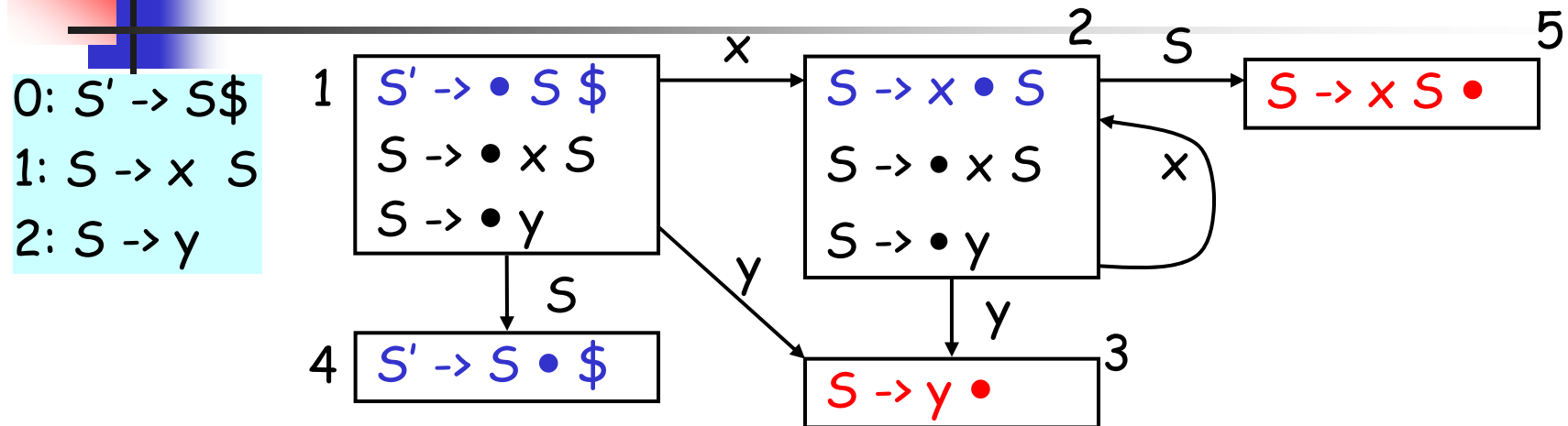
状态3包含移进-归约冲突！



SLR分析算法

- 和LR(0)分析算法基本步骤相同
- 仅区别于对归约的处理
 - 对于状态 i 上的项目 $X \rightarrow \alpha \bullet$
 - 仅对 $y \in \text{FOLLOW}(X)$ 添加 $\text{ACTION}[i, y]$

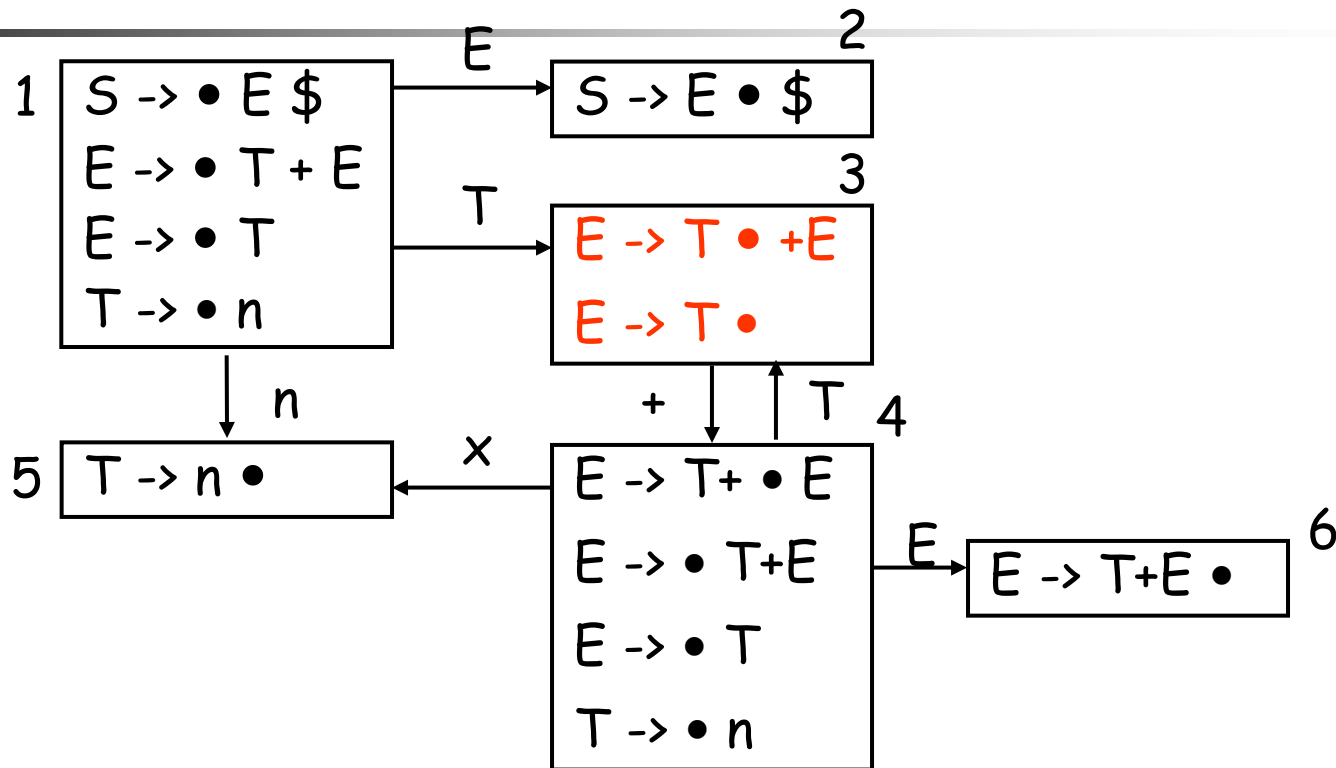
示例1

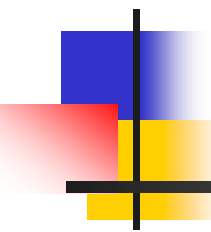


	ACTION			GOTO
状态\符号	x	y	\$	S
1	s2	s3		g4
2	s2	s3		g5
3	r2	r2	r2	
4			accept	
5	r1	r1	r1	

示例2

0: $S \rightarrow E\$$
1: $E \rightarrow T+E$
2: $E \rightarrow T$
3: $T \rightarrow n$





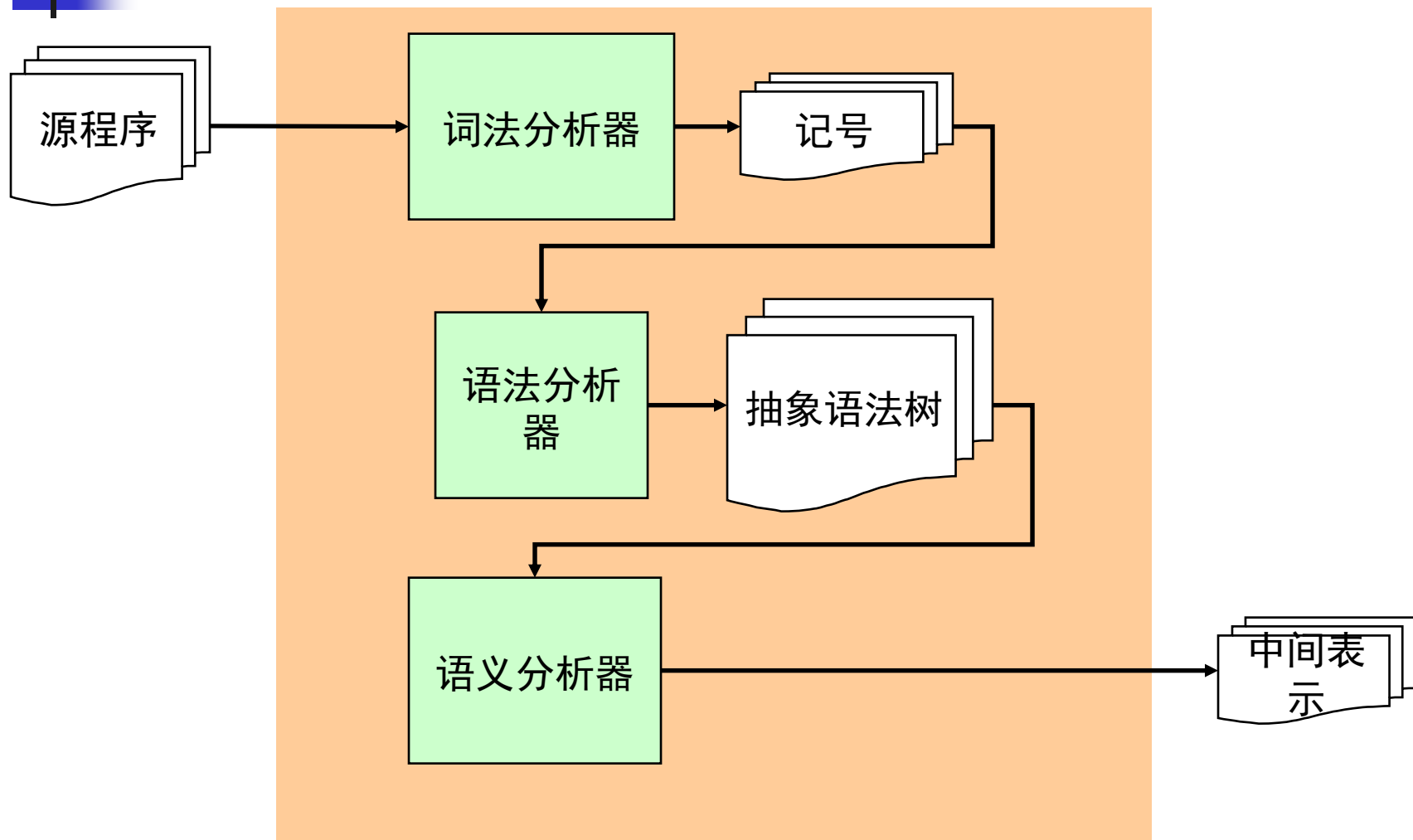
语法分析： LR(1)分析算法

编译原理

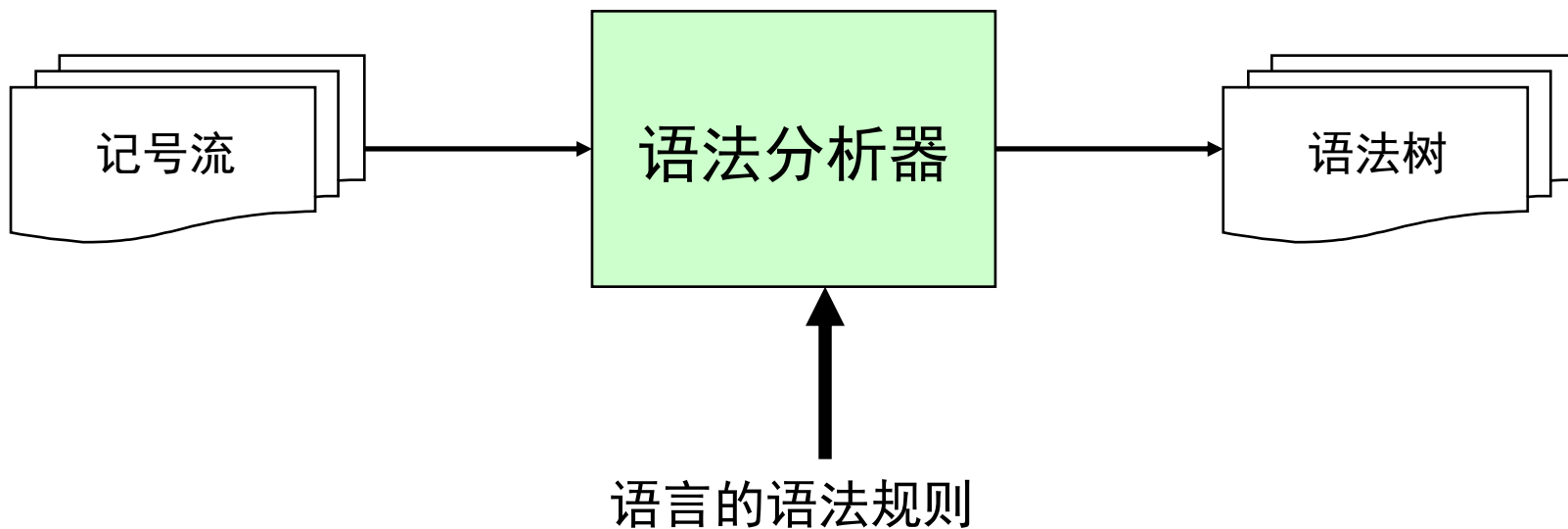
华保健

bjhua@ustc.edu.cn

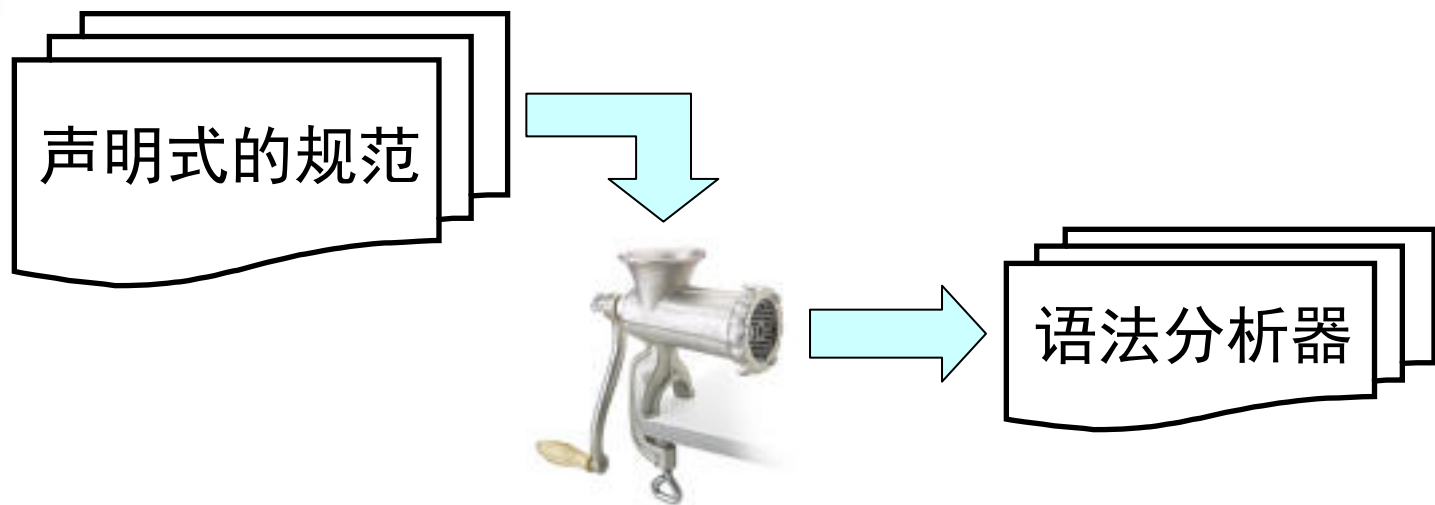
前端



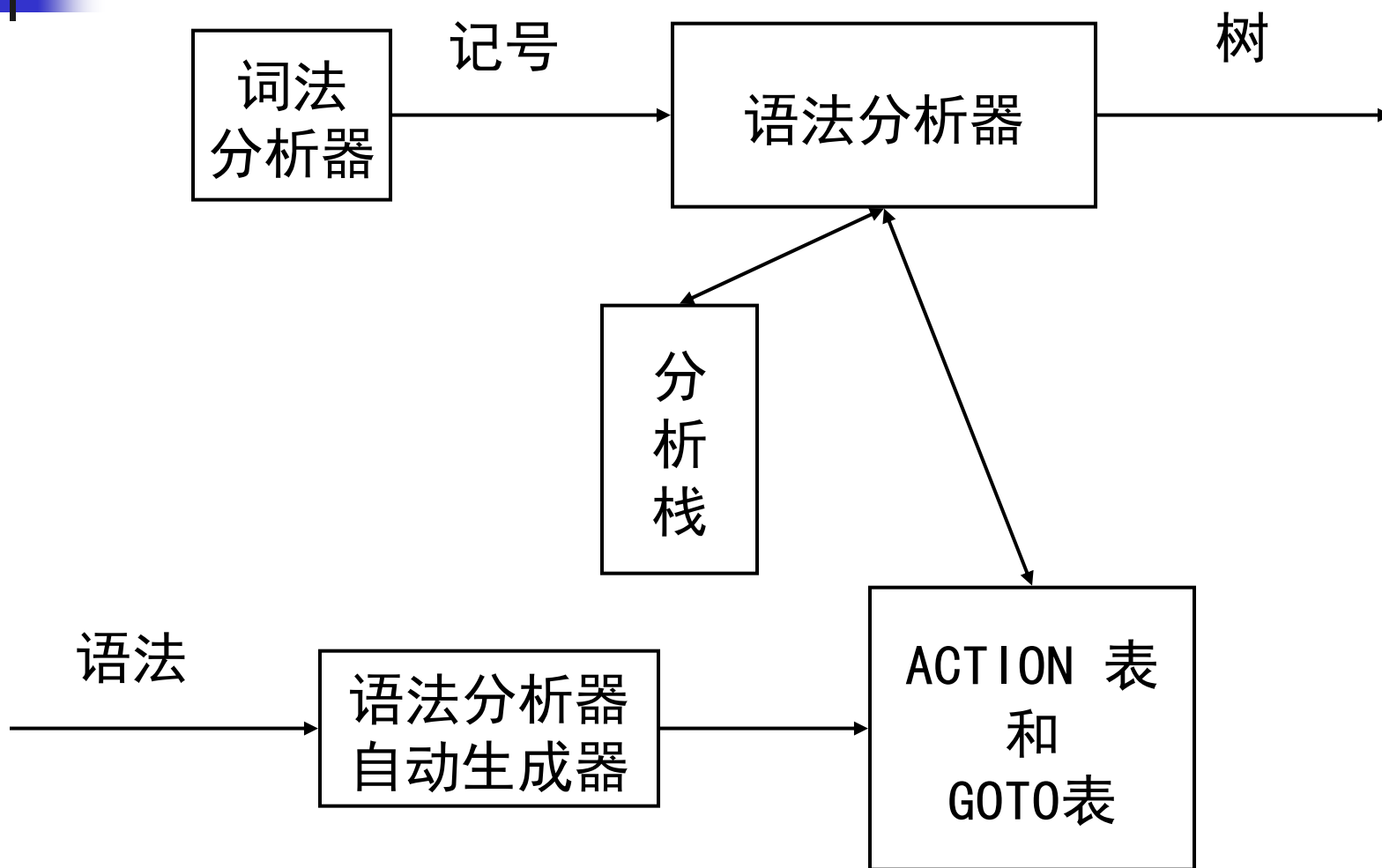
语法分析器的任务



自动生成



表驱动的LR分析器架构



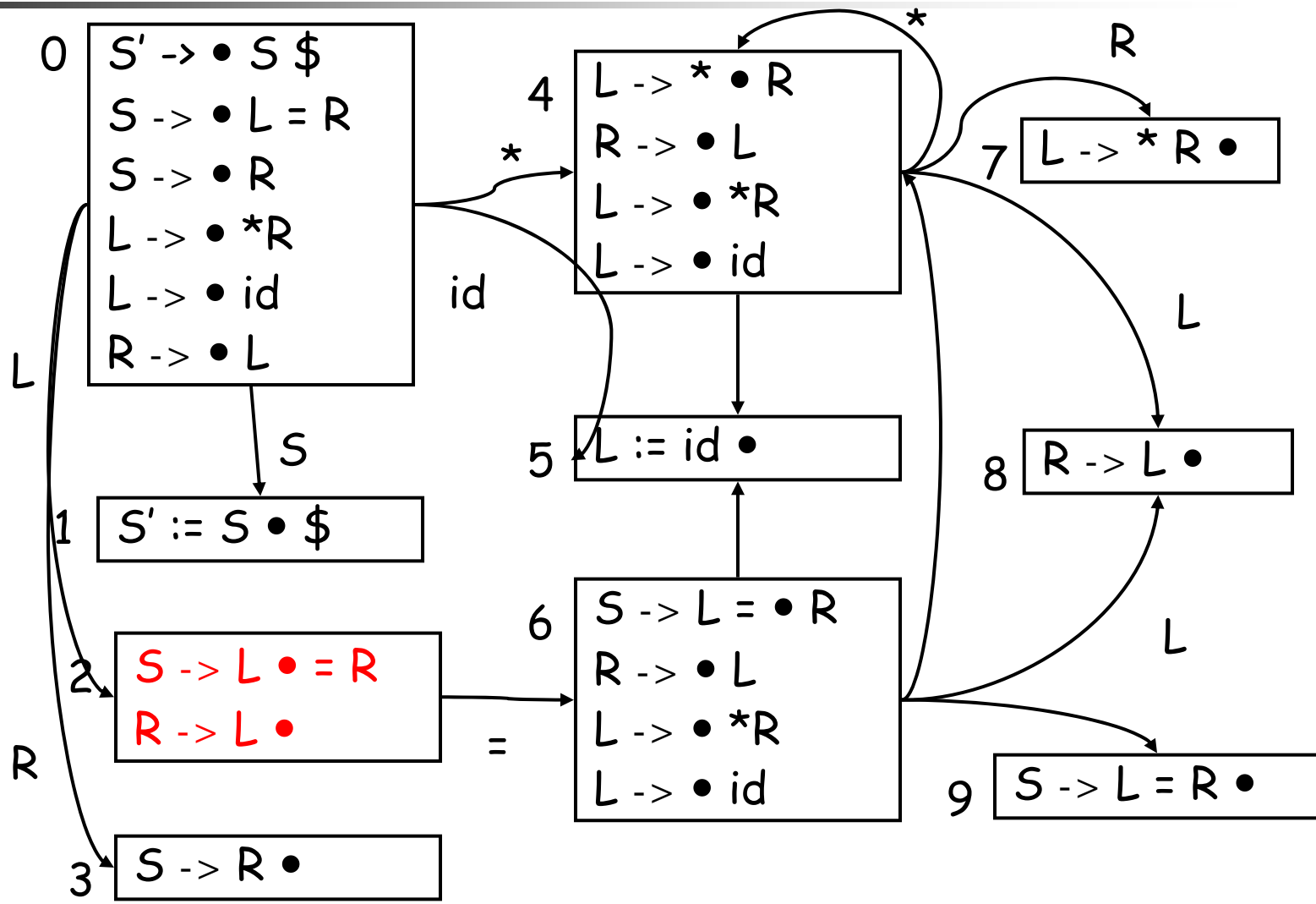


SLR分析算法的思想

- 基于LR(0)，通过进一步判断一个前看符号，来决定是否执行归约动作
 - $X \rightarrow \alpha$ • 归约，当且仅当 $y \in FOLLOW(X)$
- 优点：
 - 有可能减少需要归约的情况
 - 有可能去除需要移进-归约冲突
- 缺点：
 - 仍然有冲突出现的可能

SLR分析表中的冲突

$S' \rightarrow S\$$
 $S \rightarrow L = R$
 $S \rightarrow R$
 $L \rightarrow * R$
 $L \rightarrow id$
 $R \rightarrow L$





LR(1)项目

- $[X \rightarrow \alpha \bullet \beta, a]$ 的含义是:
 - α 在栈顶上
 - 剩余的输入能够匹配 βa
- 当归约 $X \rightarrow \alpha\beta$ 时, a 是前看符号
 - 把 'reduce by $X \rightarrow \alpha\beta$ ' 填入 $\text{ACTION}[s, a]$

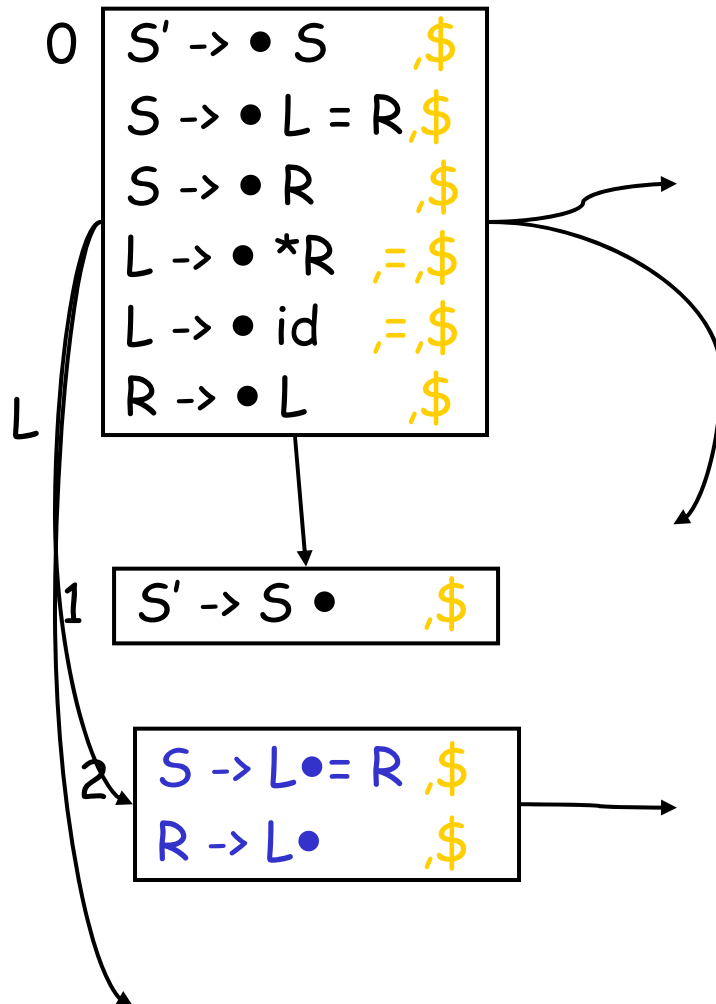


LR(1)项目的构造

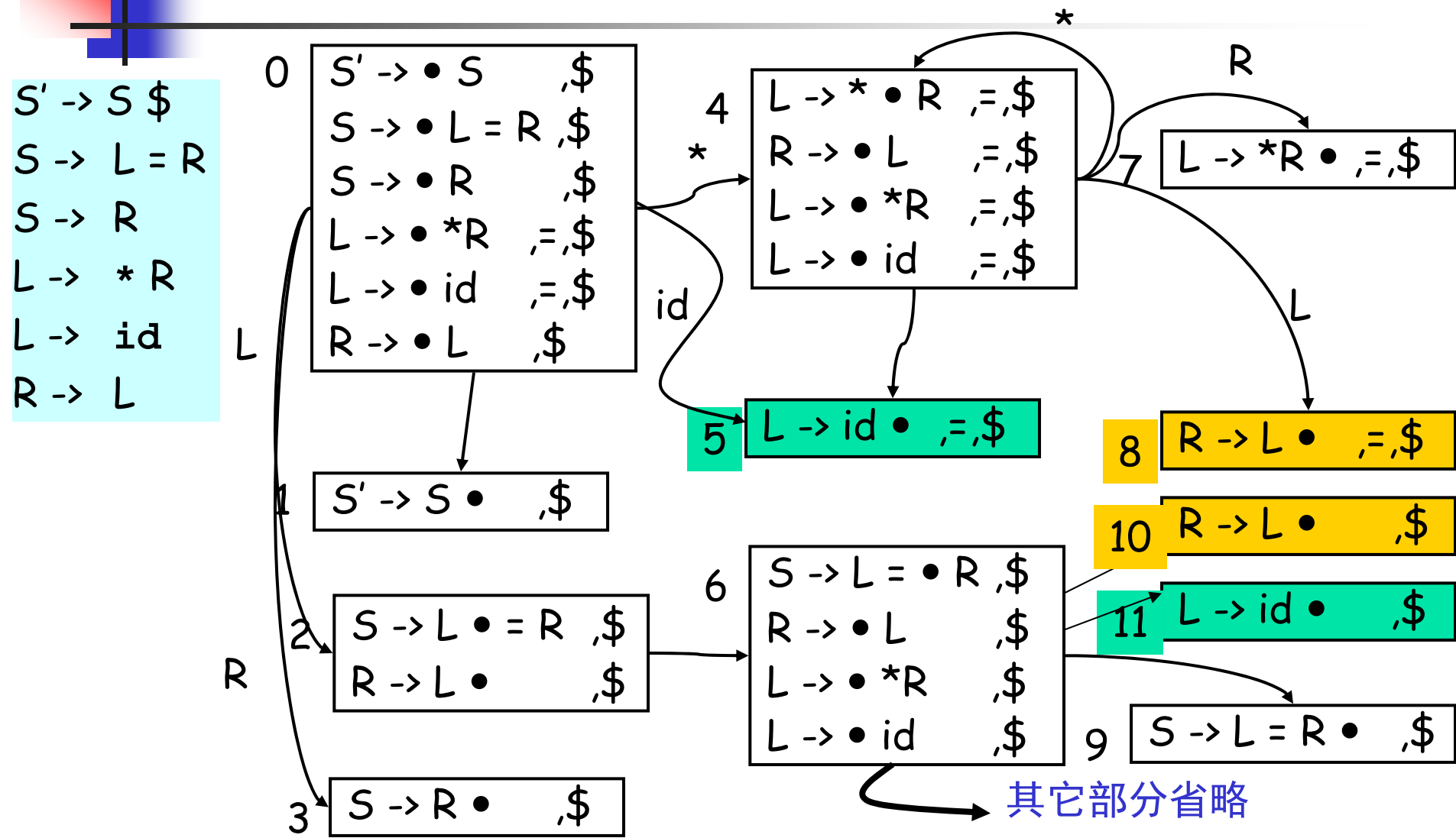
- 其他和LR(0)相同，仅闭包的计算不同：
 - 对项目 $[X \rightarrow \alpha \bullet Y \beta, a]$
 - 添加 $[Y \rightarrow \bullet \gamma, b]$ 到项目集，
 - 其中 $b \in \text{FIRST}_S(\beta a)$

LR(1)项目集 (部分)

$S' \rightarrow S\$$
 $S \rightarrow L = R$
 $S \rightarrow R$
 $L \rightarrow * R$
 $L \rightarrow id$
 $R \rightarrow L$



更多项目集





LALR分析算法

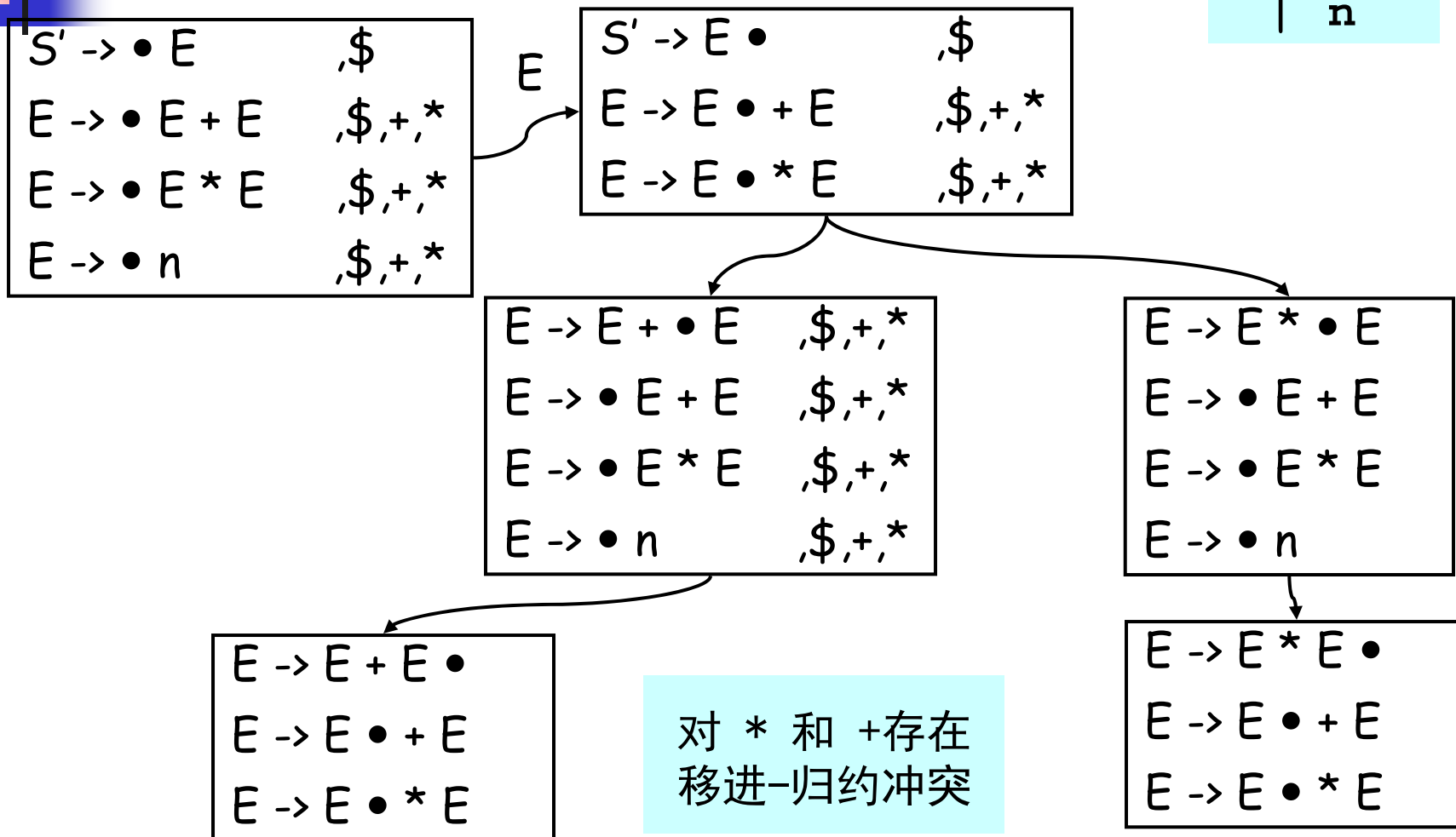
- 把类似的项目集进行合并
- 需要修改ACTION表和GOTO表，以反映合并的效果

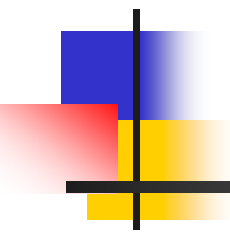


对二义性文法的处理

- 二义性文法无法使用LR分析算法分析
- 不过，有几类二义性文法很容易理解，因此，在LR分析器的生成工具中，可以对它们特殊处理
 - 优先级
 - 结合性
 - 悬空else
 - 。 。 。

优先级和结合性

$$\begin{array}{l} E \rightarrow E + E \\ | E * E \\ | n \end{array}$$




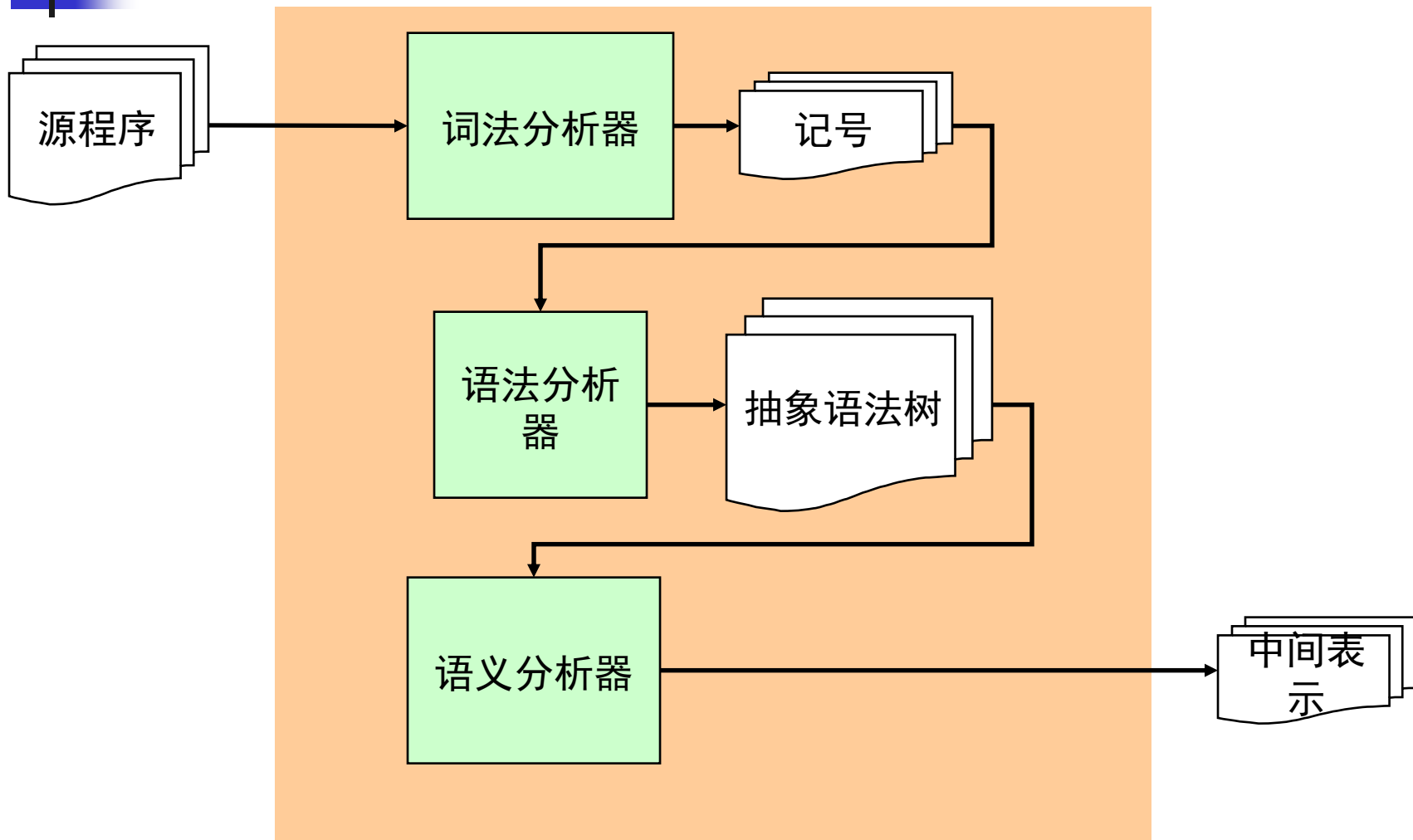
语法分析： LR(1)分析工具

编译原理

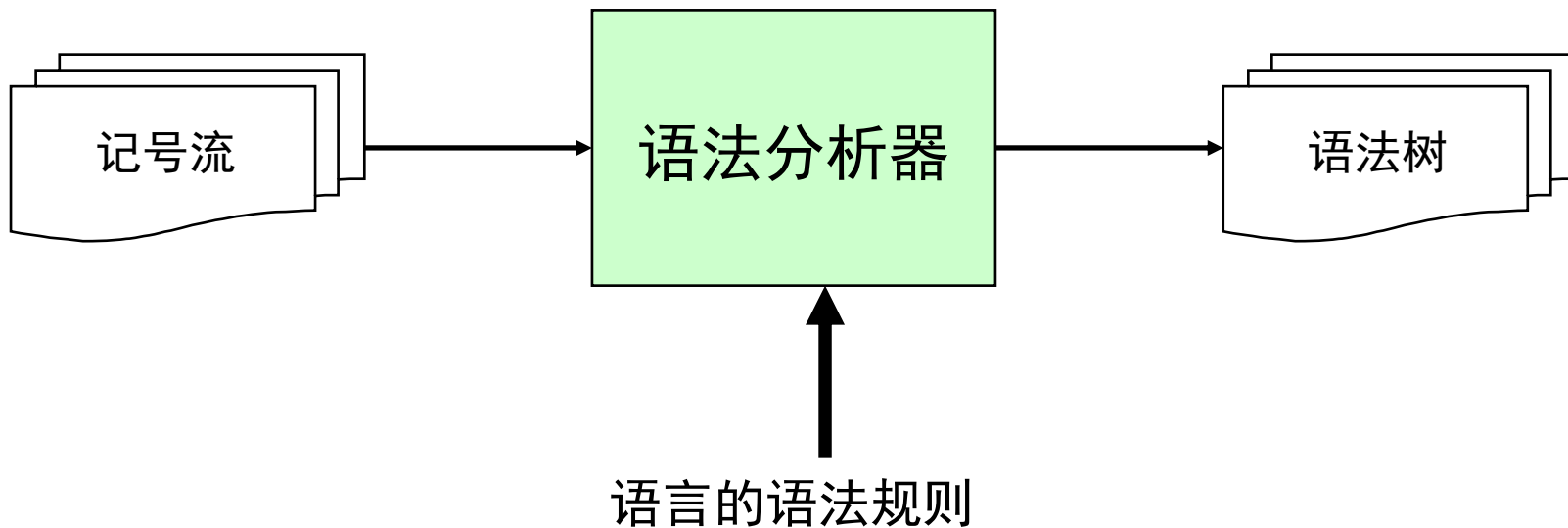
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务

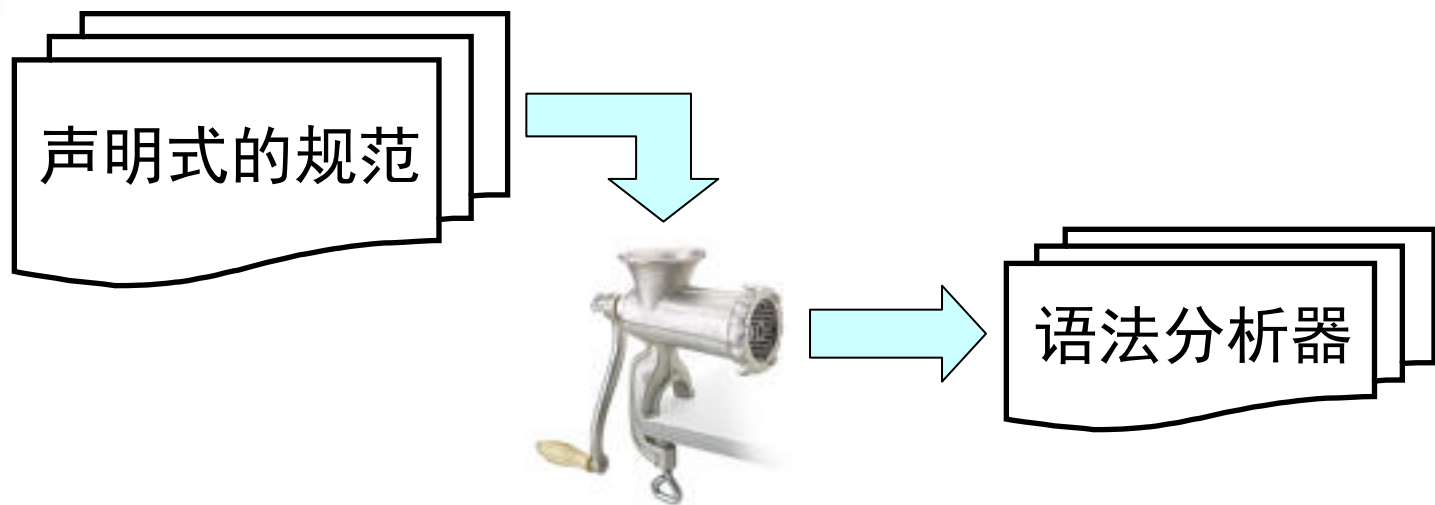




语法分析器的实现方法

- 手工方式
 - 递归下降分析器
- 使用语法分析器的自动生成器
 - $LL(1)$, $LR(1)$
- 两种方式在实际的编译器中都有广泛的应用
 - 自动的方式更适合快速对系统进行原型

自动生成





历史发展

- YACC 是 Yet Another Compiler-Compiler缩写
- 在1975年首先由Steve Johnson在Unix上实现
- 后来，很多工具在此基础上做了改进：
 - 例如GNU Bison
 - 并且移植到了很多其他语言上
- YACC 现在是一个标准的工具（见IEEE Posix 标准 P1003.2）



Yacc

用户代码和yacc声明： 可以在接下来的部分使用

%%

语法规则： 上下文无关文法的规则及相应语义动作

%%

用户代码： 用户提供的代码



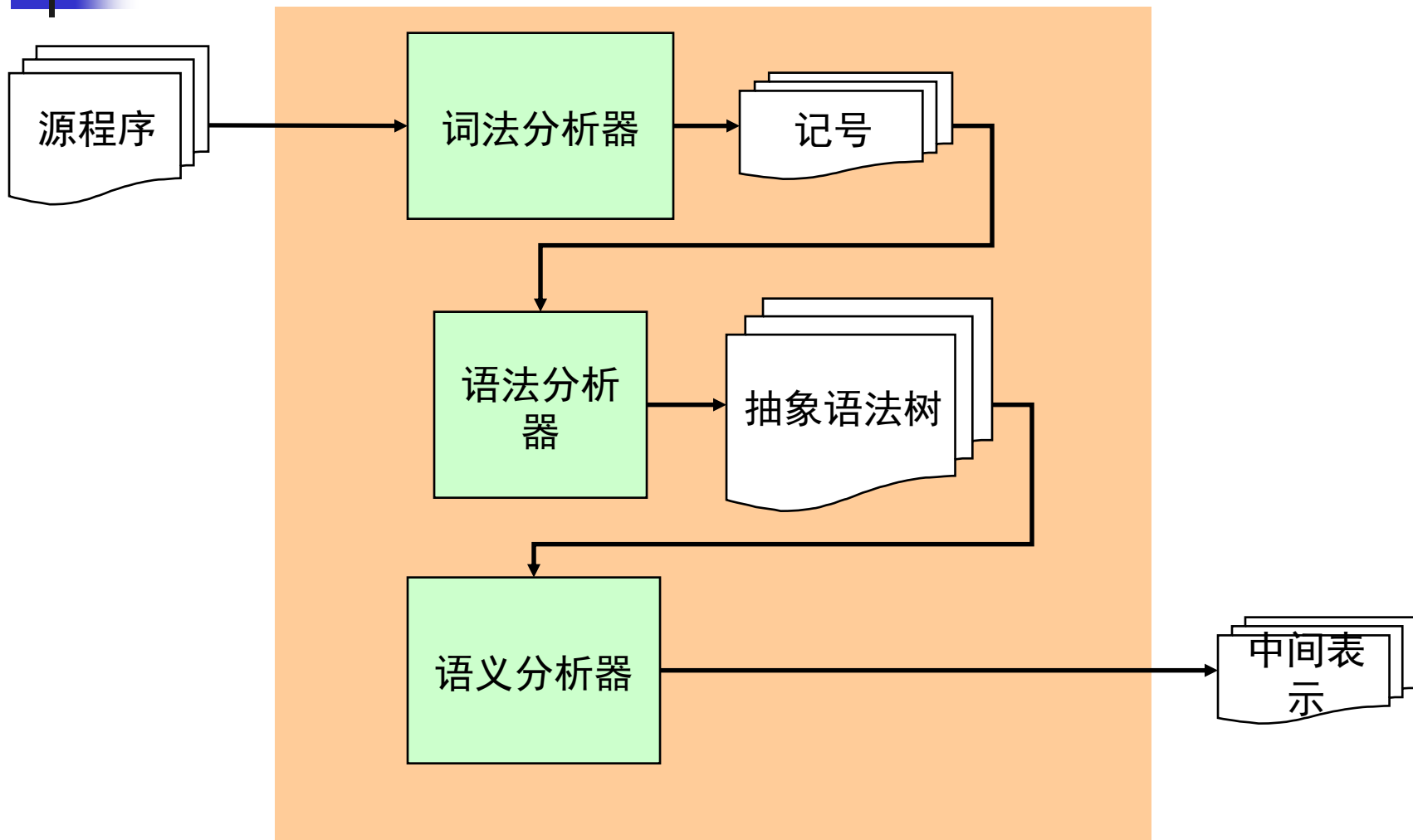
语法制导翻译

编译原理

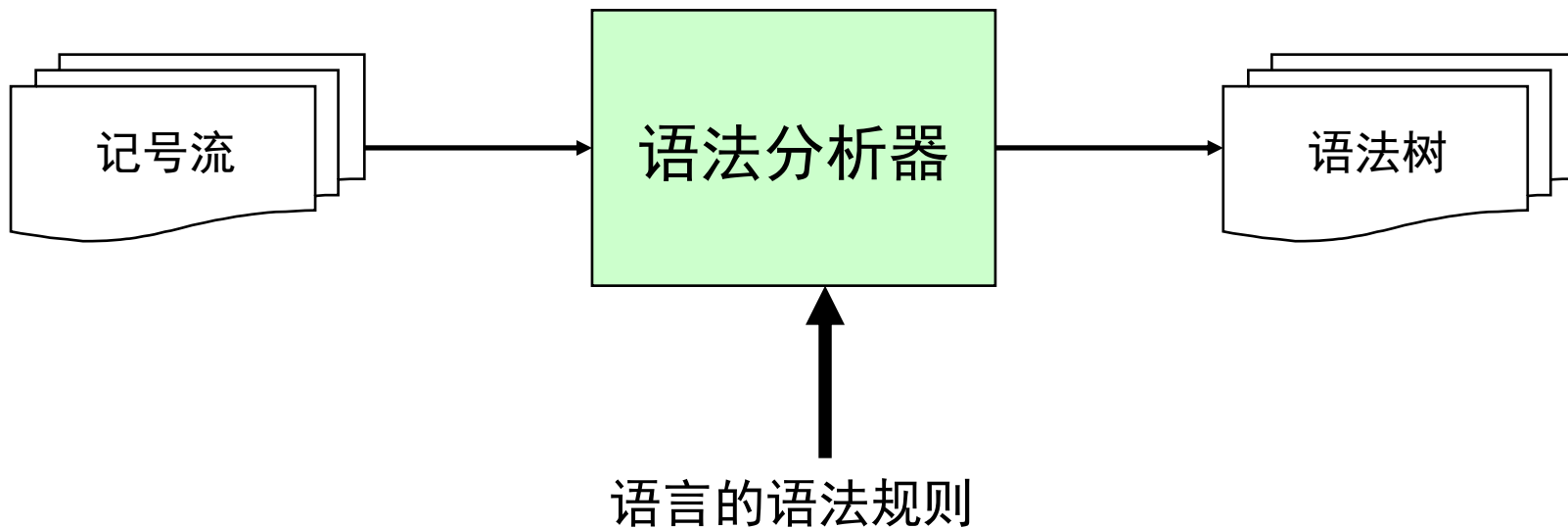
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务





语法制导的翻译

- 编译器在做语法分析的过程中，除了回答程序语法是否合法外，还必须完成后续工作
 - 可能的工作包括（但不限于）
 - 类型检查
 - 目标代码生成
 - 中间代码生成
 - 。 。 。
- 这些后续的工作一般可通过语法制导的翻译完成



基本思想

- 给每条产生式规则附加一个语义动作
 - 一个代码片段
- 语义动作在产生式“**归约**”时执行
 - 即当**右部**分析完毕时刻
 - 由**右部**的值计算**左部**的值
 - 自顶向下分析和自底向上分析采用的技术类似
 - 接下来重点讨论在自底向上的技术中的语法制导翻译

1:	$X \rightarrow \beta_1$	a_1
2:	$\mid \beta_2$	a_2
3:	$\mid \beta_3$	a_3
...		...
n:	$\mid \beta_n$	a_n



LR分析中的语法制导翻译

```
if (action[s, t]=="ri")  
    ai  
    pop( $\beta_i$ )  
    state s' = stack[top]  
    push (X)  
    push (goto[s', X])
```

1:	$X \rightarrow \beta_1$	a1
2:	β_2	a2
3:	β_3	a3
...		...
n:	β_n	a _n



示例

- 计算表达式的值

0: $E \rightarrow E + E$	$\{E = E1 + E2\}$
1: n	$\{E = n\}$

- 对应后序遍历的序
- 接下来看Yacc中的实现示例



语法制导翻译的实现原理

编译原理

华保健

bjhua@ustc.edu.cn

语法制导翻译的基本思想

- 给每条产生式规则附加一个语义动作
 - 一个代码片段
- 语义动作在产生式“**归约**”时执行
 - 即由**右部**的值计算**左部**的值
 - 以自底向上的技术为例进行讨论
 - 自顶向下的技术与此类似

0:	$X \rightarrow \beta_1$	a_1
1:	$\mid \beta_2$	a_2
2:	$\mid \beta_3$	a_3
...		...
n-1:	$\mid \beta_n$	a_n

LR分析中的语法制导翻译

```
if (action[s, t]=="ri")  
    ai  
    pop( $\beta_i$ )  
    state s' = stack[top]  
    push (X)  
    push (goto[s', X])
```

1:	X \rightarrow β_1	a1
2:	β_2	a2
3:	β_3	a3
...		...
n:	β_n	a _n

在分析栈上维护三元组: $\langle \text{symbol}, \text{value}, \text{state} \rangle$

其中symbol是终结符或非终结符, value是symbol所拥有的值, state是当前的分析状态

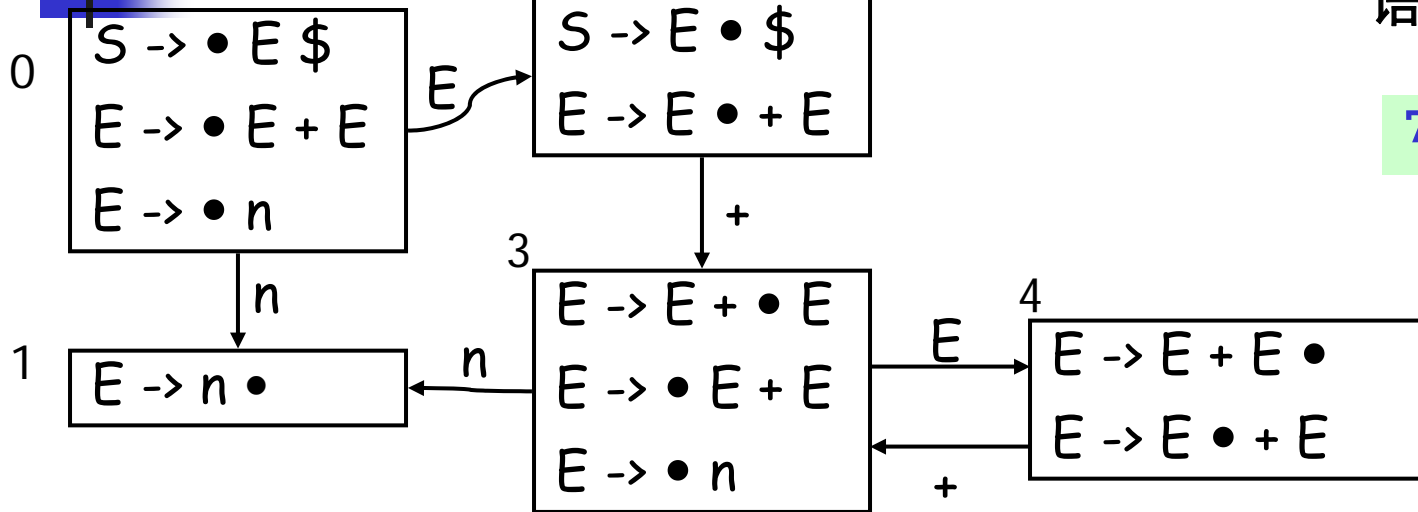
$E \rightarrow E + E \quad \{\$ \$ = \$1 + \$3;\}$
 $\quad \quad \quad | n \quad \quad \quad \{\$ \$ = n;\}$

示例

2

对这个输入的
语法制导翻译

7+8+9



+存在移进-
归约冲突



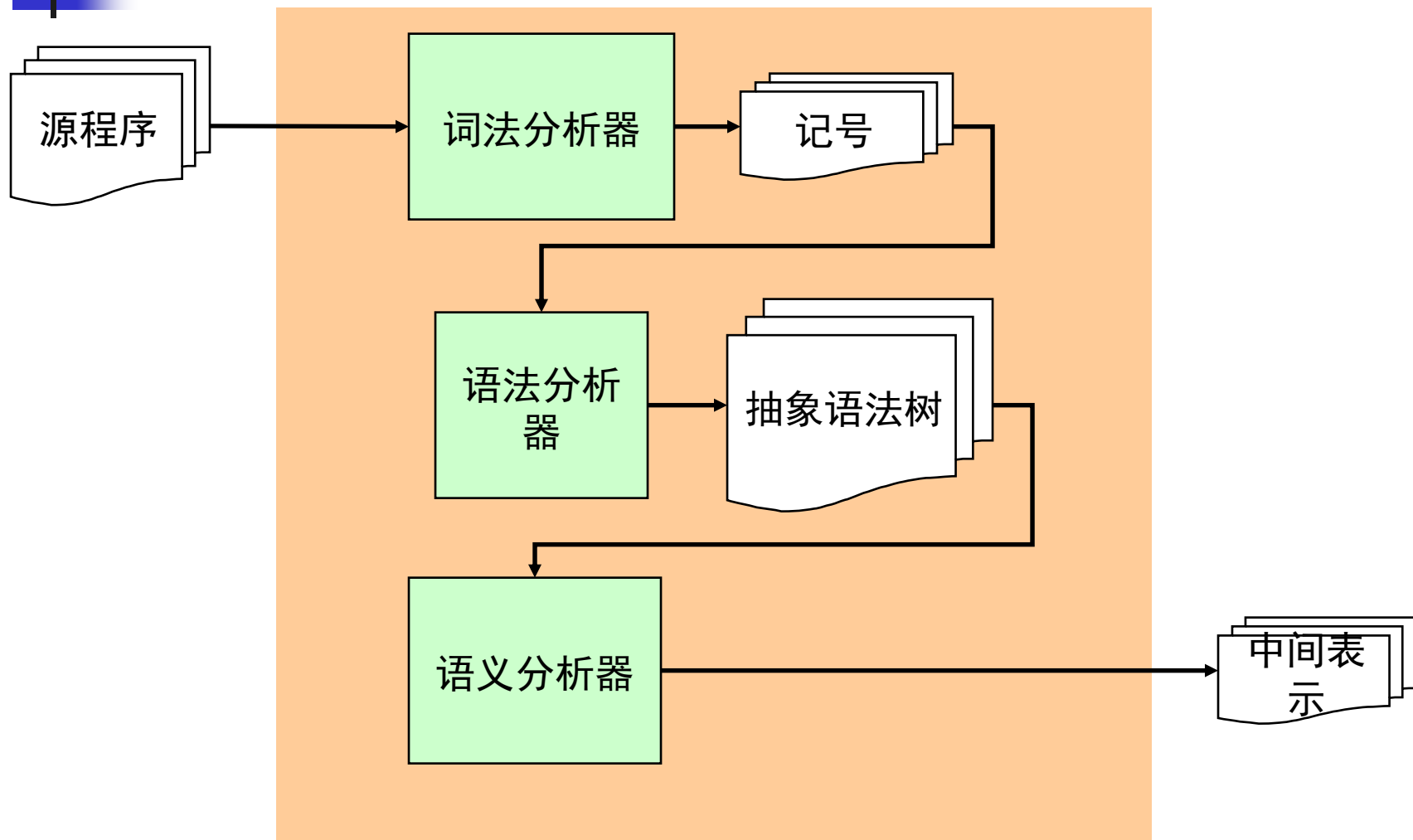
抽象语法树

编译原理

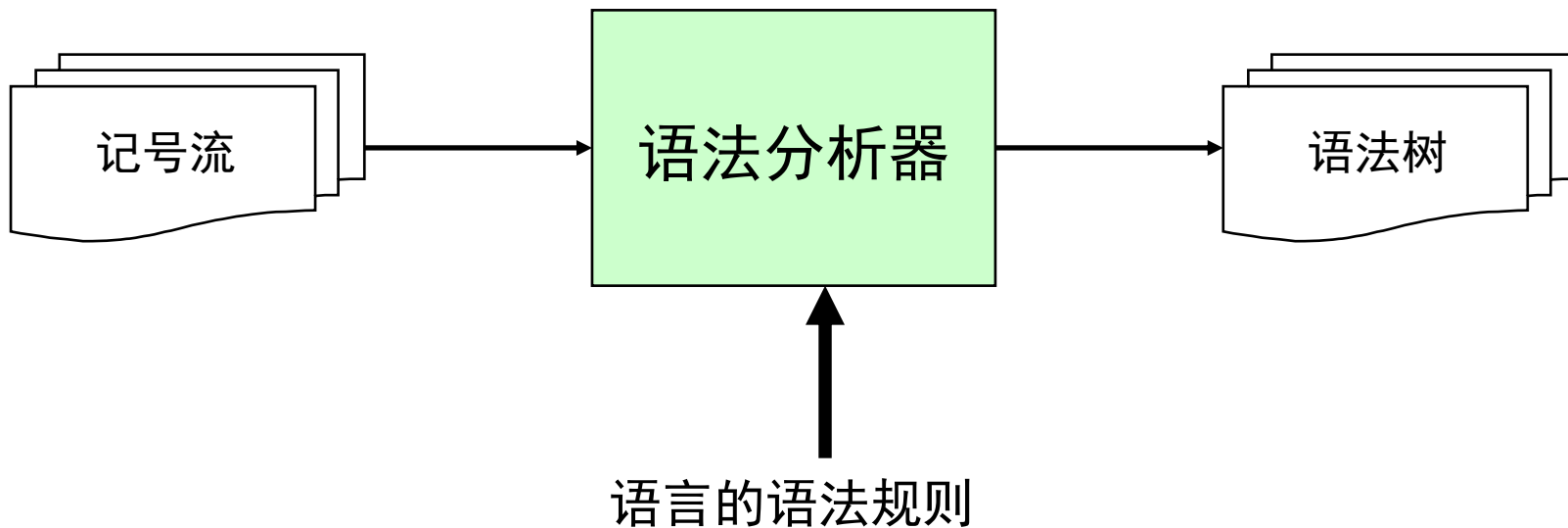
华保健

bjhua@ustc.edu.cn

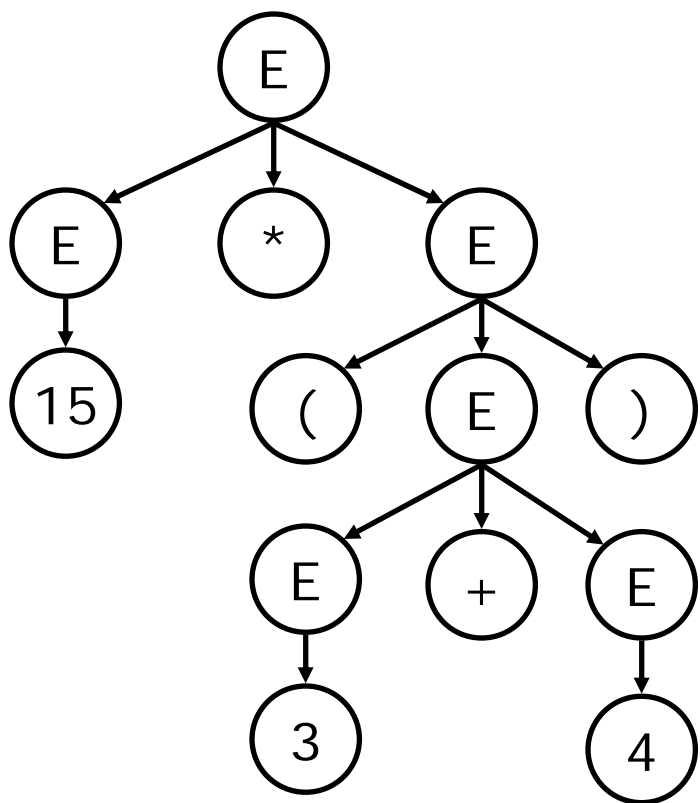
前端



语法分析器的任务



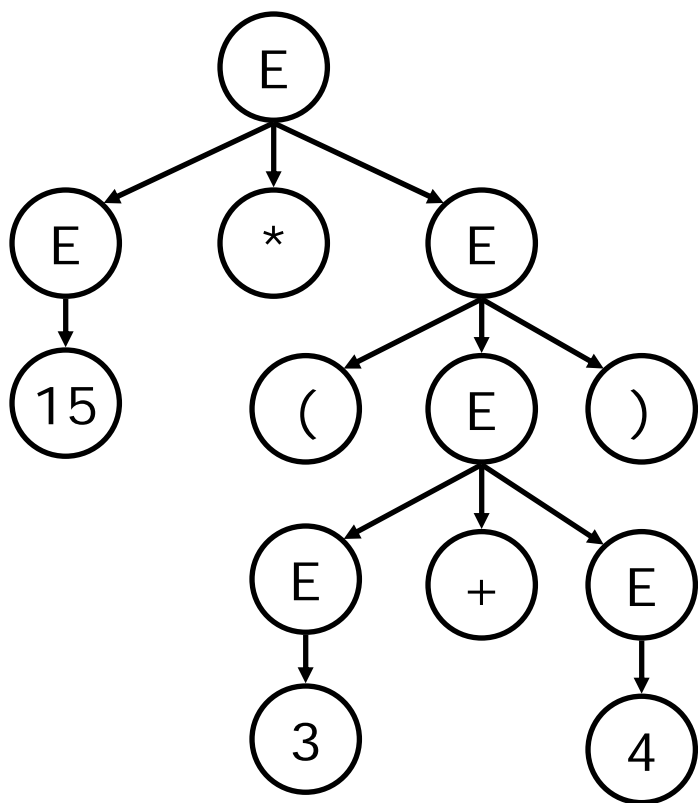
分析树



15 * (3 + 4)

- 分析树编码了句子的推导过程
- 但是包含很多不必要的信息
 - 注意：这些节点要占用额外的存储空间
- 本质上，这里的哪些信息是重要的？

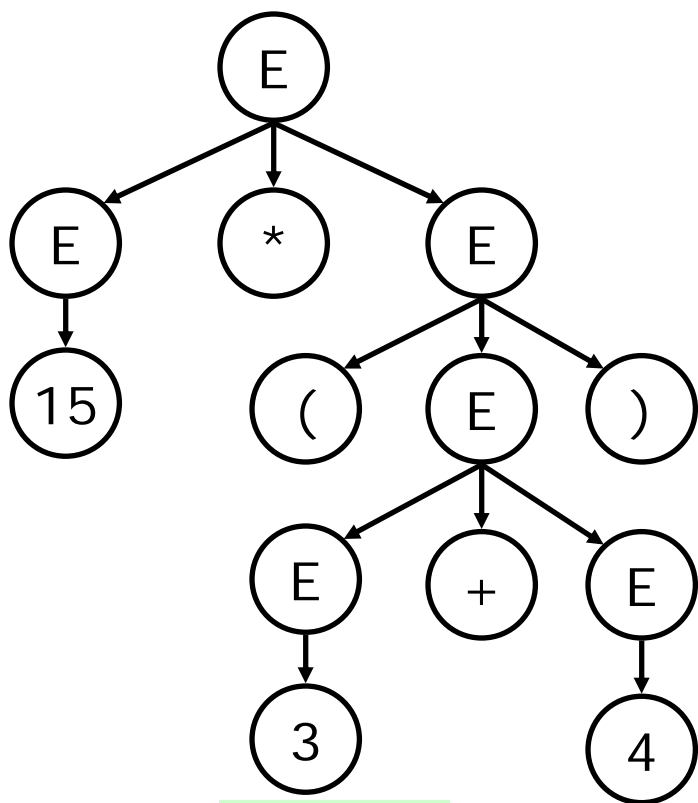
分析树



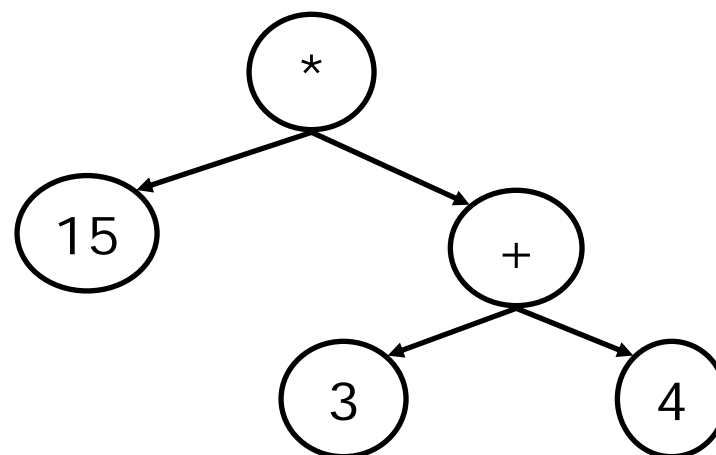
15 * (3+4)

- 对于表达式而言，编译只需要知道运算符和运算数
 - 优先级、结合性等已经在语法分析部分处理掉了
- 对于语句、函数等语言其他构造而言也一样
 - 例如，编译器不关心赋值符号是=还是:=或其它

抽象语法树



分析树



抽象语法树



具体语法和抽象语法

- 具体语法是语法分析器使用的语法
 - 必须适合于语法分析，如各种分隔符、消除左递归、提取左公因子，等等
- 抽象语法是用来表达语法结构的内部表示
 - 现代编译器一般都采用抽象语法作为前端（词法语法分析）和后端（代码生成）的接口

具体语法和抽象语法

$E \rightarrow E + T$

| T

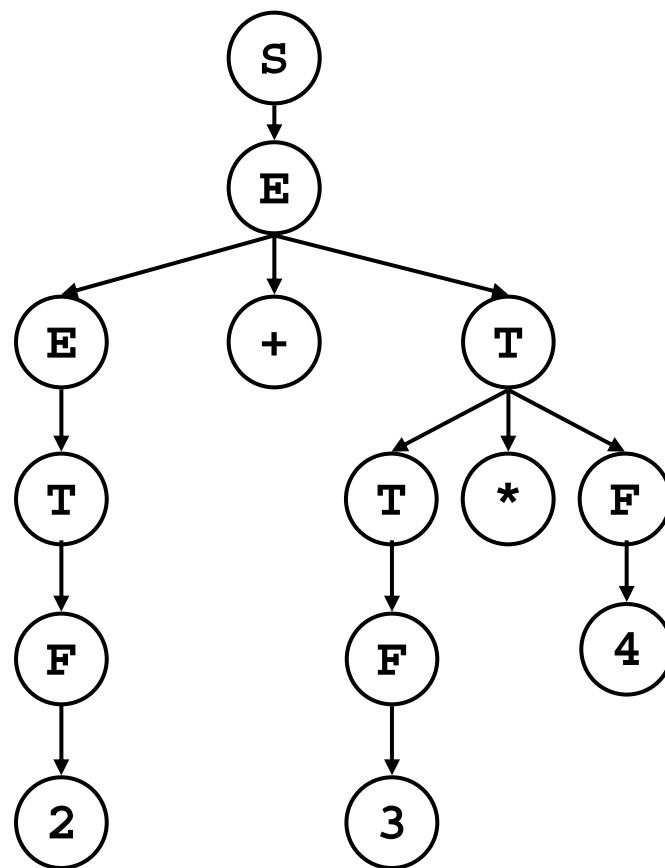
$T \rightarrow T * F$

| F

$F \rightarrow n$

| (E)

2 + 3 * 4



具体语法和抽象语法

2 + 3 * 4

$E \rightarrow E + T$

| T

$T \rightarrow T * F$

| F

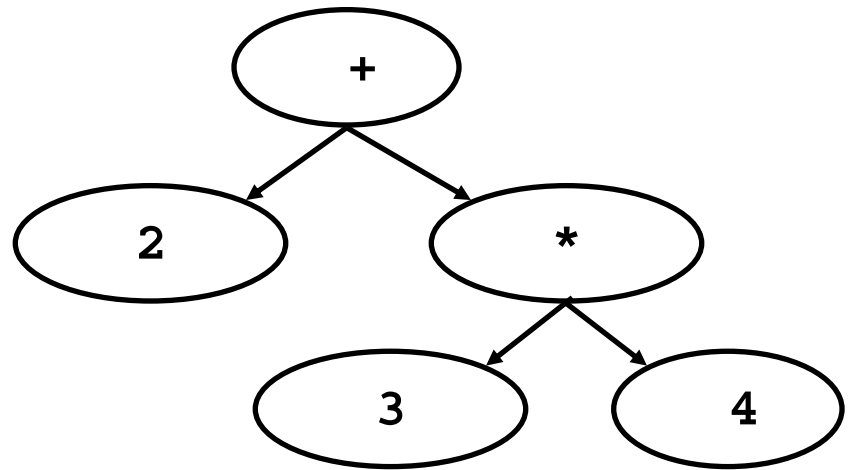
$F \rightarrow n$

| (E)

$E \rightarrow n$

| $E + E$

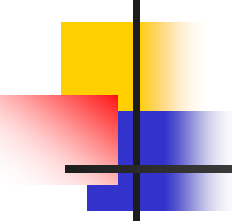
| $E * E$





抽象语法树数据结构

- 在编译器中，为了定义抽象语法树，需要使用实现语言来定义一组数据结构
 - 和实现语言密切相关
- 早期的编译器有的不采用抽象语法树数据结构
 - 直接在语法制导翻译中生成代码
 - 但现代的编译器一般采用抽象语法树作为语法分析器的输出
 - 更好的系统的支持
 - 简化编译器的设计



抽象语法树的定义

C语言版



数据结构的定义

```
/* 数据结构 */
enum kind {E_INT, E_ADD, E_TIMES};
struct Exp {
    enum kind kind;
};
struct Exp_Int{
    enum kind kind;
    int n;
};
struct Exp_Add{
    enum kind kind;
    struct Exp *left;
    struct Exp *right;
};
```

```
E -> n
      | E + E
      | E * E
```

```
struct Exp_Times{
    enum kind kind;
    struct Exp *left;
    struct Exp *right;
};
```



“构造函数”的定义

```
struct Exp_Int *Exp_Int_new (int n)
{
    struct Exp_Int *p
        = malloc (sizeof(*p));
    p->kind = E_INT;
    p->n = n;
    return p;
}
```

```
struct Exp_Add *Exp_Add_new(struct Exp *left
    , struct Exp *right)
{
    struct Exp_Add *p = malloc (sizeof(*p));
    p->kind = E_ADD;
    p->left = left; p->right = right;
    return p;
}
```

```
E -> n
    | E + E
    | E * E
```



示例

```
/* 用数据结构来编码程序 "2+3*4" */  
e1 = Exp_Int_new (2);  
e2 = Exp_Int_new (3);  
e3 = Exp_Int_new (4);  
e4 = Exp_Times_new (e2, e3);  
e5 = Exp_Add_new (e1, e4);
```

```
E -> n  
    | E + E  
    | E * E
```




AST上的操作成为树的遍历

```
/* 优美打印 */
void pretty_print (e){
    switch (e->kind) {
        case E_INT: printf ("%d", e->n); return;
        case E_ADD:
            printf ("("); pretty_print (e->left);
            printf (")"); // 需要适当的类型转换
            printf (" + ");
            printf ("("); pretty_print (e->right);
            printf (")");
            return;
        other cases: /* similar */
    }
}
```



示例：树的规模

```
/* 节点的个数 */
int numNodes (E e)
{
    switch (e->kind) {
        case E_INT: return 1;
        case E_ADD:
        case E_TIMES:
            return 1 + numNodes (e->left)
                    + numNodes (e->right);
        default:
            error ("compiler bug");
    }
}
```



示例：从表达式到栈式计算机 Stack的编译器

```
/* 编译器：请参考课程第一部分的作业内容：Sum -> Stack*/  
List all;    // 存放生成的所有指令  
void compile (E e)  
{  
    switch (e->kind) {  
        case E_INT: emit(push e->n); return;  
        case E_ADD:  
        case E_TIMES:  
            // 留作练习  
        default:  
            error ("compiler bug");  
        }  
    }  
}
```



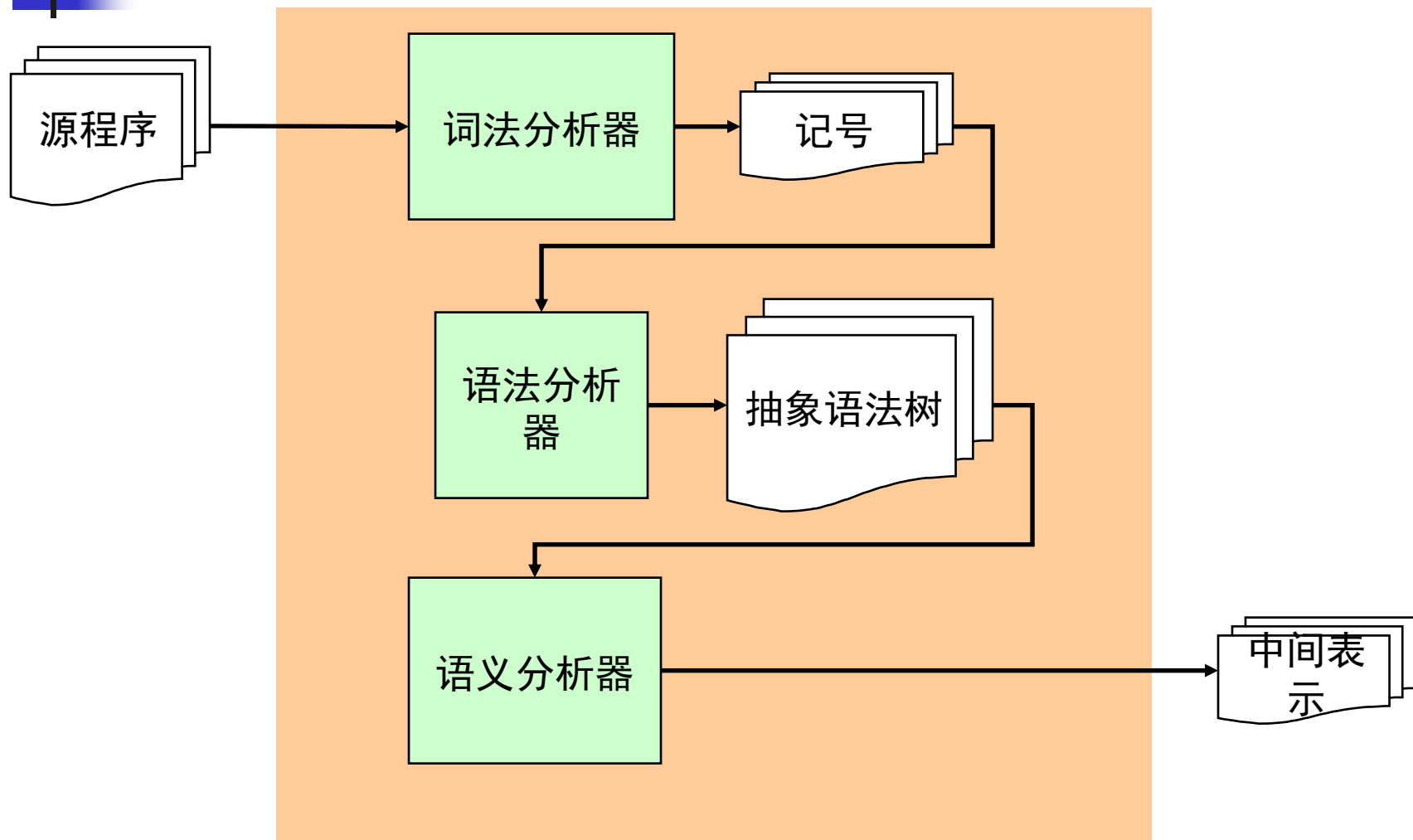
抽象语法树的自动生成

编译原理

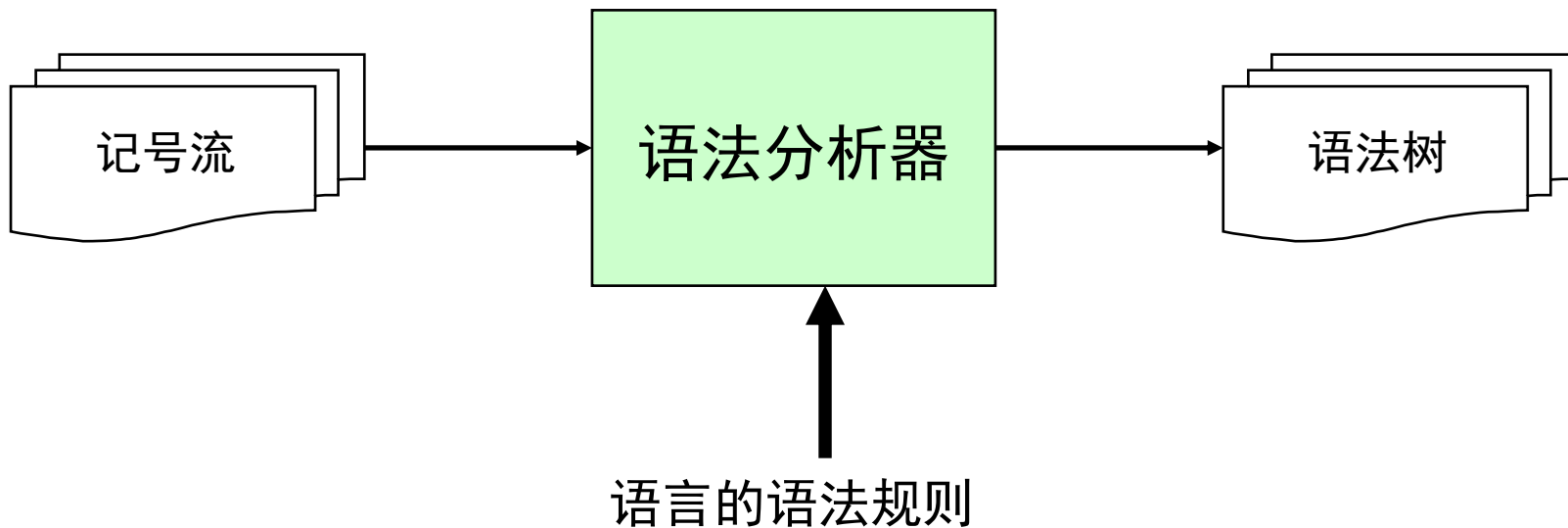
华保健

bjhua@ustc.edu.cn

前端



语法分析器的任务





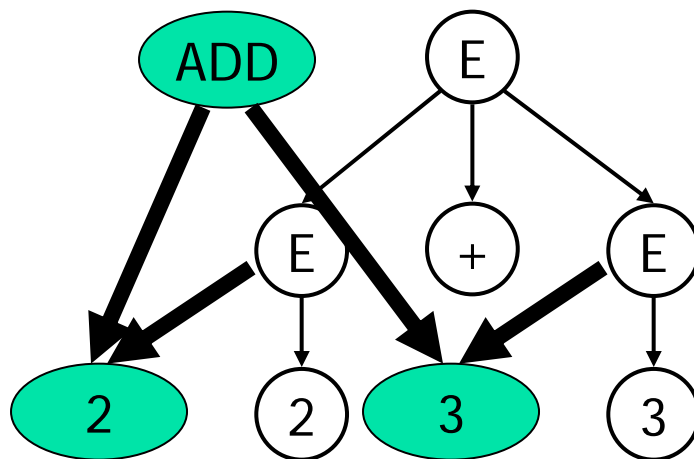
LR分析中生成抽象语法树

- 在语法动作中，加入生成语法树的代码片段
 - 片段一般是语法树的“构造函数”
- 在产生式归约的时候，会自底向上构造整棵树
 - 从叶子到根

示例：LR分析中生成抽象语法树

$E \rightarrow E + E \quad \{\$ \$ = \text{Exp_Add_new} (\$1, \$3); \}$
 $\quad \quad \quad | E * E \quad \{\$ \$ = \text{Exp_Times_new} (\$1, \$3); \}$
 $\quad \quad \quad | n \quad \quad \quad \{\$ \$ = \text{Exp_Int_new} (\$1); \}$

	2 + 3 + 4
2	● + 3 + 4
E	● + 3 + 4
E +	● 3 + 4
E + 3	● + 3 + 4
E + E	● 3 + 4
E	● + 4
E +	● 4
E + 4	●
E + E	●





源代码信息的保留和传播

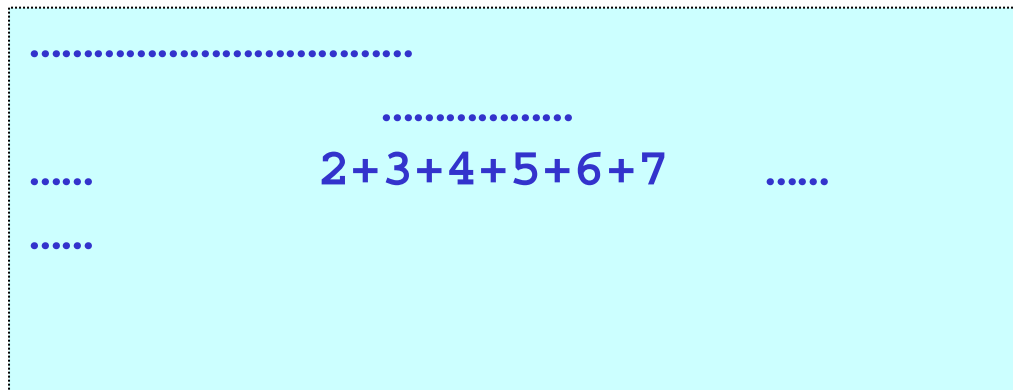
- 抽象语法树是编译器前端和后端的接口
 - 程序一旦被转换成抽象语法树，则源代码即被丢弃
 - 后续的阶段只处理抽象语法树
- 所以抽象语法树必须编码足够多的源代码信息
 - 例如，它必须编码每个语法结构在源代码中的位置（文件、行号、列号等）
 - 这样，后续的检查阶段才能精确的报错
 - 或者获取程序的执行剖面
- 抽象语法树必须仔细设计！



示例：位置信息

```
struct position_t{
    char *file;
    int line;
    int column;
};

struct Exp_Add{
    enum kind kind;
    Exp *left;
    Exp *right;
    struct position_t from;
    struct position_t to;
};
```



.....

.....

..... 2+3+4+5+6+7

.....



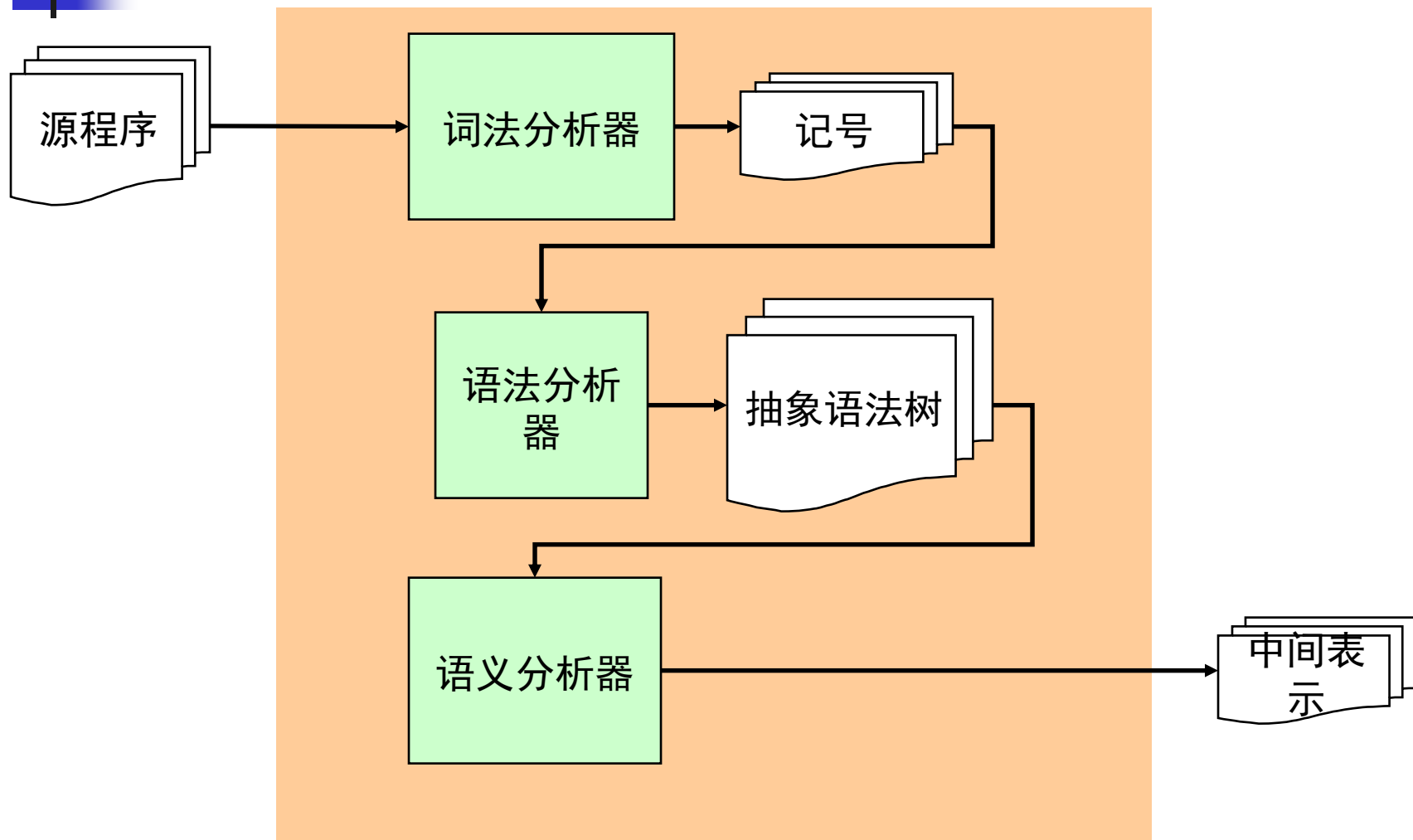
语义分析

编译原理

华保健

bjhua@ustc.edu.cn

前端





语义分析的任务

- 语义分析也称为类型检查、上下文相关分析
- 负责检查程序（抽象语法树）的上下文相关的属性：
 - 这是具体语言相关的，典型的情况包括：
 - 变量在使用前先进行声明
 - 每个表达式都有合适的类型
 - 函数调用和函数的定义一致
 - ...



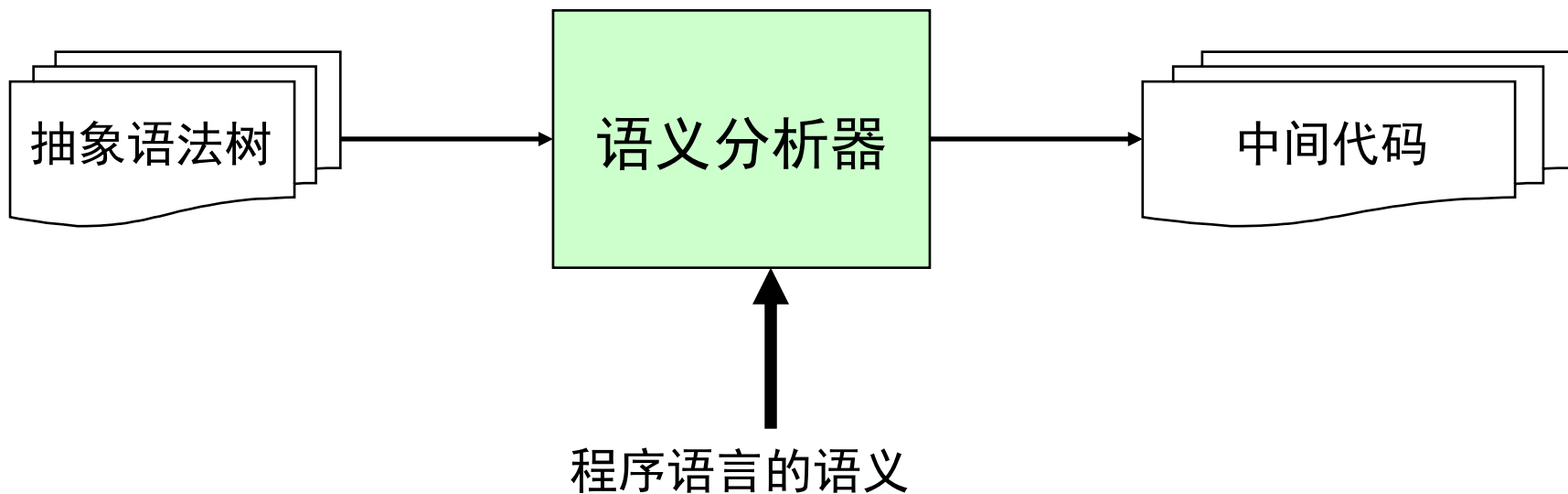
语义分析的示例

// 示例C代码。有哪些错误？

```
void f (int *p)
{
    x += 4;
    p (23);
    "hello" + "world";
}
```

```
int main ()
{
    f () + 5;
    break;
    return;
}
```

语义分析器概念上的结构





程序语言的语义

- 传统上，大部分的程序设计语言都采用自然语言来表达程序语言的语义
 - 例如，对于“+”运算：
 - 要求左右操作数都必须是整型数
- 编译器的实现者必须对语言中的语义规定有全面的理解
 - 主要的挑战是如何能够正确高效的实现
 - 接下来的内容，我们将一起讨论涉及的主要问题和解决方案



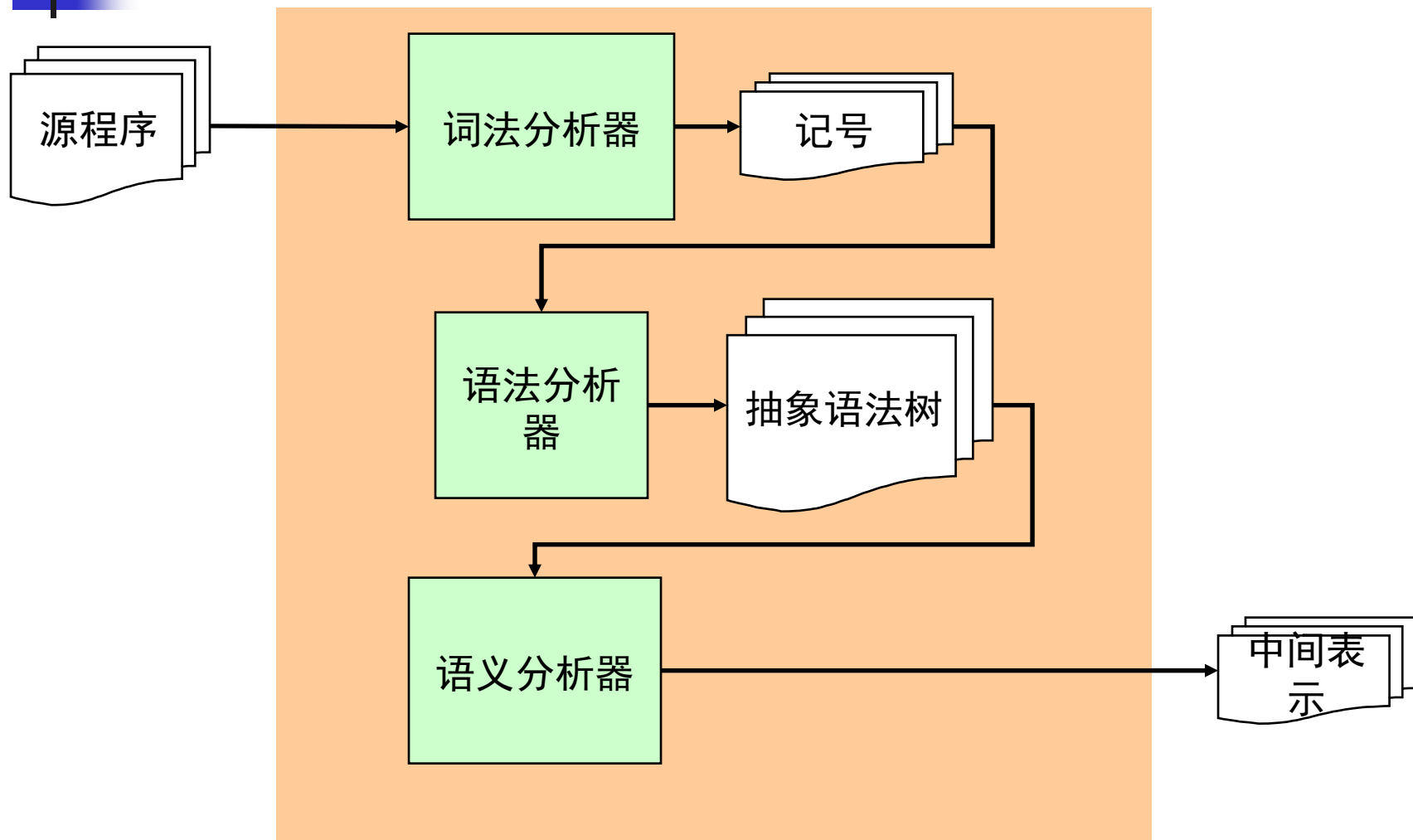
语义分析：语义检查

编译原理

华保健

bjhua@ustc.edu.cn

前端





语义分析的任务

- 语义分析也称为类型检查、上下文相关分析
- 负责检查程序（语法树）的上下文相关的属性：
 - 这是具体语言相关的，典型的情况包括：
 - 变量在使用前先进行声明
 - 每个表达式都有合适的类型
 - 函数调用和函数的定义一致
 - ...



C--语言

```
E -> n
    | true
    | false
    | E + E
    | E && E
```

// 类型合法的程序：

`3+4`

`false && true`

// 类型不合法的程序：

`3 + true`

`true + false`

// 对这个语言，语义分析的任务是：对给定的一个表达式`e`，写一个函数

`type check(e);`

// 返回表达式`e`的类型；若类型不合法，则报错。



类型检查算法

```
E -> n
      | true
      | false
      | E + E
      | E && E
```

```
enum type {INT, BOOL};
```

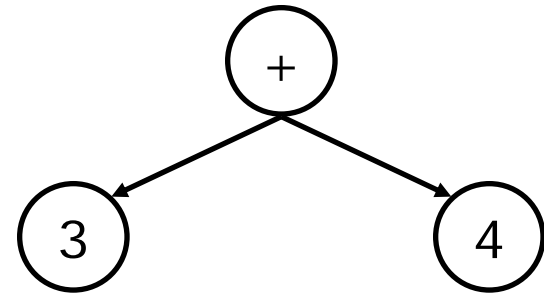
```
enum type check_exp (Exp_t e)
{
    switch(e->kind)
    {
        case EXP_INT: return INT;
        case EXP_TRUE: return BOOL;
        case EXP_FALSE: return BOOL;
        case EXP_ADD: t1 = check_exp (e->left);
                      t2 = check_exp (e->right);
                      if (t1!=INT || t2!=INT)
                          error ("type mismatch");
                      else return INT;
        case EXP_AND: ... // 类似; 留作练习
    }
}
```

示例1

```
E -> n
      | true
      | false
      | E + E
      | E && E
```

```
enum type {INT, BOOL};
```

```
enum type check_exp (Exp_t e)
{
    switch(e->kind)
    {
        case EXP_INT: return INT;
        case EXP_TRUE: return BOOL;
        case EXP_FALSE: return BOOL;
        case EXP_ADD: t1 = check_exp (e->left);
                      t2 = check_exp (e->right);
                      if (t1!=INT || t2!=INT)
                          error ("type mismatch");
                      else return INT;
        case EXP_AND: ... // 类似; 留作练习
    }
}
```



示例2

```
E -> n
      | true
      | false
      | E + E
      | E && E
```

```
enum type {INT, BOOL};
```

```
enum type check_exp (Exp_t e)
```

```
switch(e->kind)
```

```
case EXP_INT: return INT;
```

```
case EXP_TRUE: return BOOL;
```

```
case EXP_FALSE: return BOOL;
```

```
case EXP_ADD: t1 = check_exp (e->left);
```

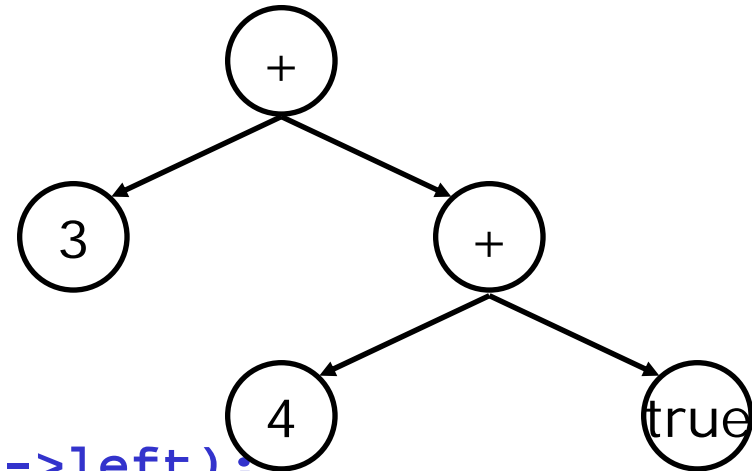
```
                t2 = check_exp (e->right);
```

```
                if (t1!=INT || t2!=INT)
```

```
                    error ("type mismatch");
```

```
                else return INT;
```

```
case EXP_AND: ... // 类似; 留作练习
```





变量声明的处理

// 类型合法的程序:

```
int x;
```

```
x+4
```

// 类型合法的程序:

```
bool y;
```

```
false && y
```

// 类型不合法的程序:

```
x + 3
```

// 类型不合法的程序:

```
int x;
```

```
x + false
```

```
P -> D E
D -> T id; D
    |
T -> int
    | bool
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

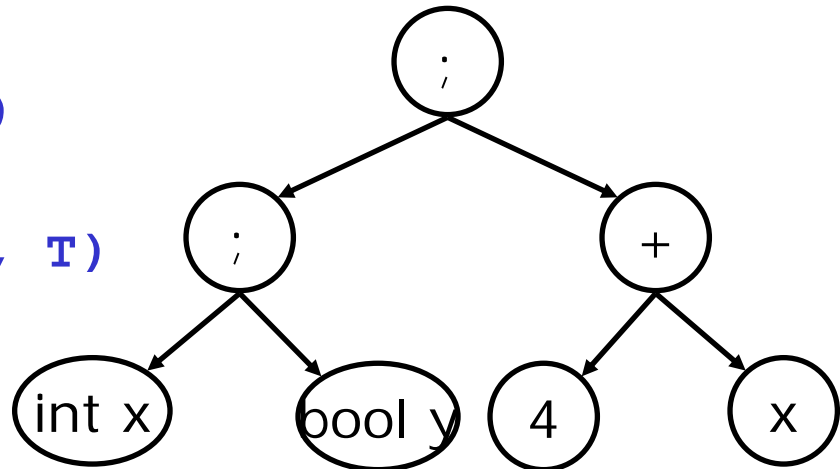

类型检查算法

```
enum type {INT, BOOL};  
Table_t table;
```

```
enum type check_prog (Dec_t d, Exp_t e)  
    table = check_dec (d)  
    return check_exp (e)
```

```
Table_t check_dec (Dec_t d)  
    foreach (T id ∈ d)  
        table_enter (table, id, T)
```

```
P -> D E  
D -> T id; D  
    |  
T -> int  
    |  
    bool  
E -> n  
    |  
    id  
    |  
    true  
    |  
    false  
    |  
    E + E  
    |  
    E && E
```

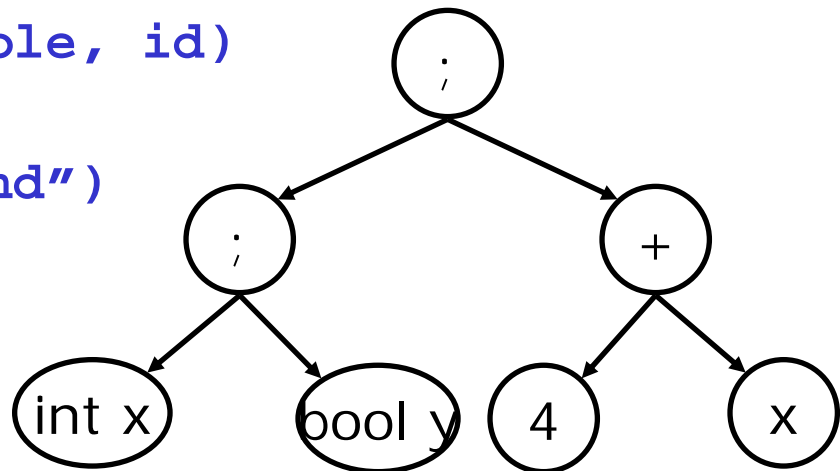


类型检查算法（续）

```
enum type {INT, BOOL};  
Table_t table;
```

```
enum type check_exp (Exp_t e)  
{  
    switch (e->kind)  
    {  
        case EXP_ID:  
            t = Table_lookup (table, id)  
            if (id not exist)  
                error ("id not found")  
            else return t  
    }  
}
```

```
P -> D E  
D -> T id; D  
    |  
T -> int  
    |  
    bool  
E -> n  
    |  
    id  
    true  
    false  
    E + E  
    E && E
```





语句的处理

```
void check_stm (Table_t table, Stm_t s)
switch(s->kind)
    case STM_ASSIGN:
        t1 = Table_lookup (s->id)
        t2 = check_exp (table, s->exp)
        if (t1!=t2)
            error("type mismatch")
        else return INT;
    case STM_PRINTI:
        t = check_exp(s->exp)
        if (t!=INT)
            error ("type mismatch")
        else return;
    case STM_PRINTB: ... // 类似; 留作练习
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```



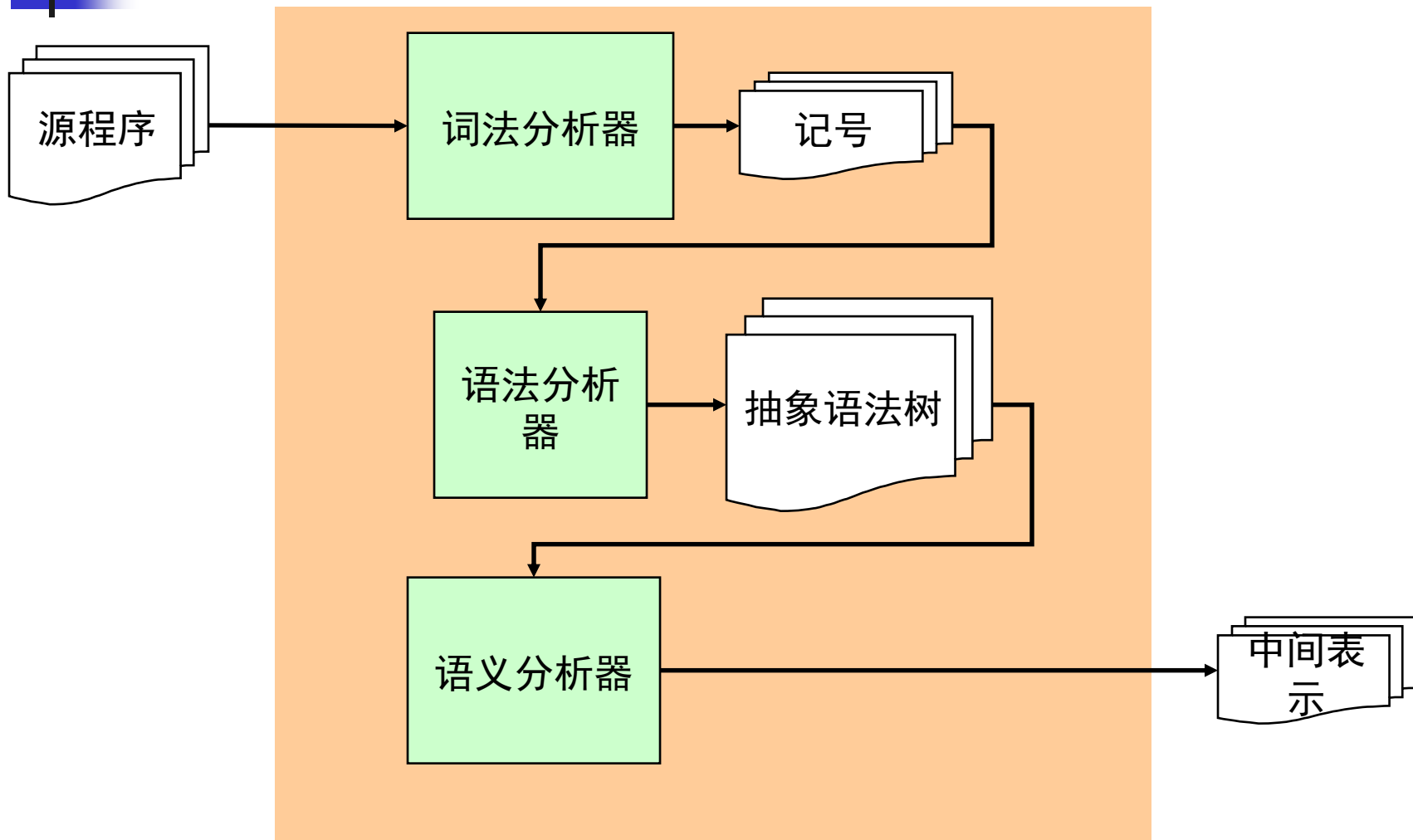
语义分析：符号表

编译原理

华保健

bjhua@ustc.edu.cn

前端





符号表

- 用来存储程序中的变量相关信息
 - 类型
 - 作用域
 - 访问控制信息
 - 。 。 。
- 必须非常高效
 - 程序中的变量规模会很大



符号表的接口

```
#ifndef TABLE_H
#define TABLE_H

typedef ... Table_t; // 数据结构

// 新建一个符号表
Table_t Table_new ();
// 符号表的插入
void Table_enter (Table_t, Key_t, Value_t);
// 符号表的查找
Value_t Table_lookup (Table_t, Key_t);

#endif
```



符号表的典型数据结构

```
// 符号表是典型的字典结构：  
symbolTable: key ->  
value  
// 一种简单的数据结构的定义（概  
念上的）：
```

```
typedef char *key;  
typedef struct value{  
    Type_t type;  
    Scope_t scope;  
    ... // 必要的其他字段  
} value;
```

变量\映射	type	scope	...
x	INT	0	...
y	BOOL	1	...
...



符号表的高效实现

- 为了高效，可以使用哈希表等数据结构来实现符号表
 - 查找是 $O(1)$ 时间
- 为了节约空间，也可以使用红黑树等平衡树
 - 查找是 $O(\lg N)$ 时间



作用域

```
int x;  
int f ()  
{  
    if (4) {  
        int x;  
        x = 6;  
    }  
    else {  
        int x;  
        x = 5;  
    }  
    x = 8;  
}
```



符号表处理作用域的方法

- 方法#1：一张表的方法
 - 进入作用域时，插入元素
 - 退出作用域时，删除元素

示例

```
int x;            $\sigma = \{x \rightarrow \text{int}\}$ 
int f ()          $\sigma_1 = \sigma + \{f \rightarrow \dots\} = \{x \rightarrow \text{int}, f \rightarrow \dots\}$ 
{
    if (4) {
        int x;    $\sigma_2 = \sigma_1 + \{x \rightarrow \text{int}\} = \{x \rightarrow \dots, f \rightarrow \dots, x \rightarrow \dots\}$ 
        x = 6;
    }
    else {
        int x;    $\sigma_4 = \sigma_1 + \{x \rightarrow \text{int}\} = \{x \rightarrow \dots, f \rightarrow \dots, x \rightarrow \dots\}$ 
        x = 5;
    }
    x = 8;
}
```

屏蔽



符号表处理作用域的方法

- 方法#1：一张表的方法
 - 进入作用域时，插入元素
 - 退出作用域时，删除元素
- 方法#2：采用符号表构成的栈
 - 进入作用域时，插入新的符号表
 - 退出作用域时，删除栈顶符号表

示例

```
int x;  
int f ()  
{  
    if (4) {  
        int x;  
        x = 6;  
    }  
    else {  
        int x;  
        x = 5;  
    }  
    x = 8;  
}
```

变量\映射	type	scope
x	INT	0
f	...	0

变量\映射	type	scope
x	INT	1



名字空间

```
struct list
{
    int x;
    struct list *list;
} *list;

void walk (struct list *list)
{
    list:
    printf ("%d\n", list->x);
    if (list = list->list)
        goto list;
}
```



用符号表处理名字空间

- 每个名字空间用一个表来处理
- 以C语言为例
 - 有不同的名字空间:
 - 变量
 - 标签
 - 标号
 - 。 。 。
 - 可以每类定义一张符号表



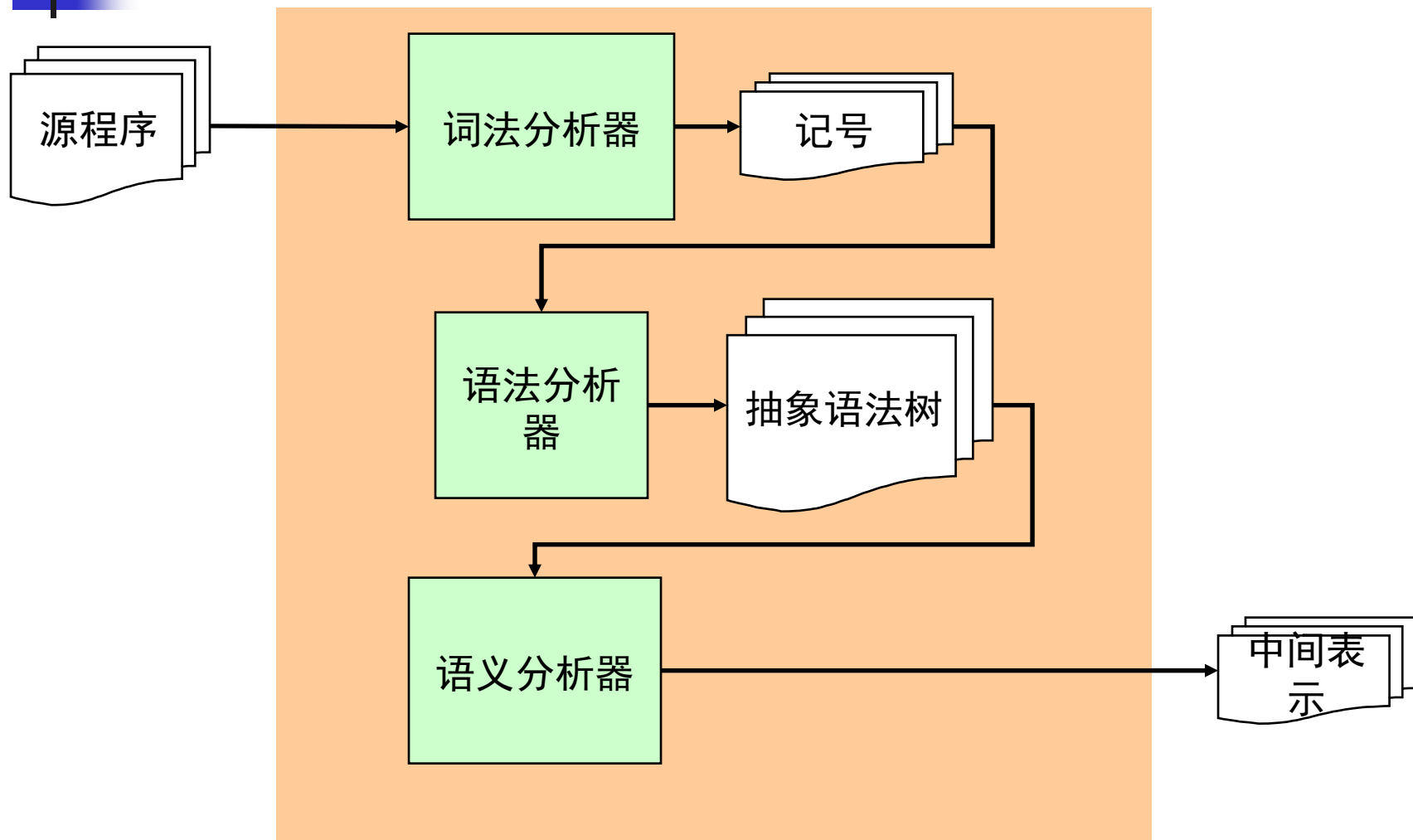
语义分析：其它问题

编译原理

华保健

bjhua@ustc.edu.cn

前端





其它问题

- 语义分析中要考虑的其它问题：
 - 类型相容性？
 - 错误诊断？
 - 代码翻译？



类型相等

- 类型检查问题往往归结为判断两个类型是否相等 $t1 == t2$?
 - 在实际的语言中，这往往是个需要谨慎处理的复杂问题
- 示例1：名字相等 vs 结构相等
 - 对采用名字相等的语言，可直接比较
 - 对采用结构相等的语言，需要递归比较各个域

```
struct A
{
    int i;
} x;
```

```
struct B
{
    int i;
} y;
```

$x = y;$



类型相等

- 类型检查问题往往归结为判断两个类型是否相等 $t1 == t2$?
 - 在实际的语言中，这往往是个需要谨慎处理的复杂问题
- 示例2：面向对象的继承
 - 需要维护类型间的继承关系

```
class A
{
    int i;
}

class B
extends A
{
    int i;
}

A x;
B y;
x = y;
```



错误诊断

- 要给出尽可能准确的错误信息
- 要给出尽可能多的错误信息
 - 从错误中进行恢复
- 要给出尽可能准确的出错位置
 - 程序代码的位置信息要从前端保留并传递过来



代码翻译

- 现代的编译器中的语义分析模块，除了做语义分析外，还要负责生成中间代码或目标代码
 - 代码生成的过程也同样是对树的某种遍历
 - 代码翻译将在课程后续讨论
- 因此，语义分析模块往往是编译器中最庞大也最复杂的模块



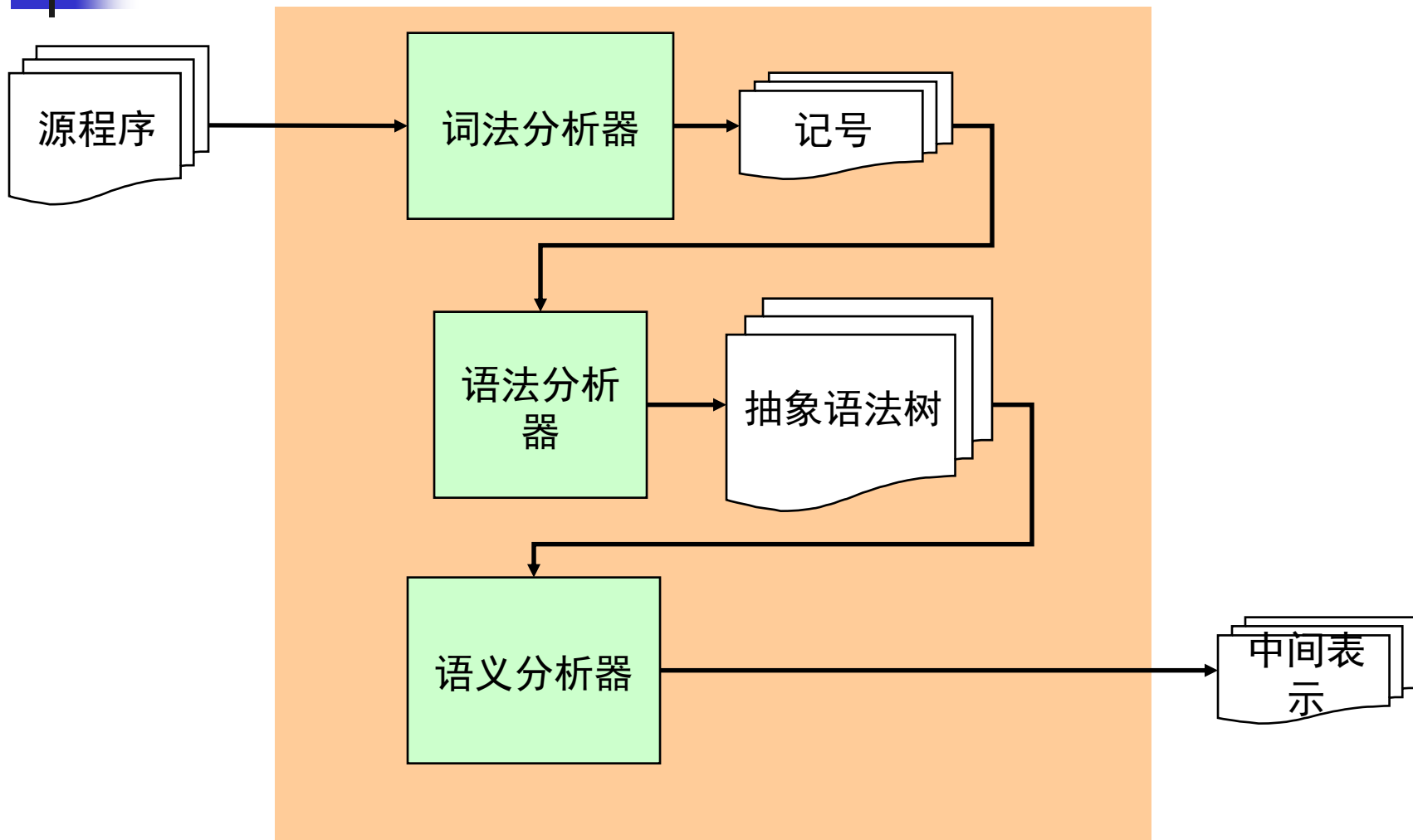
代码生成

编译原理

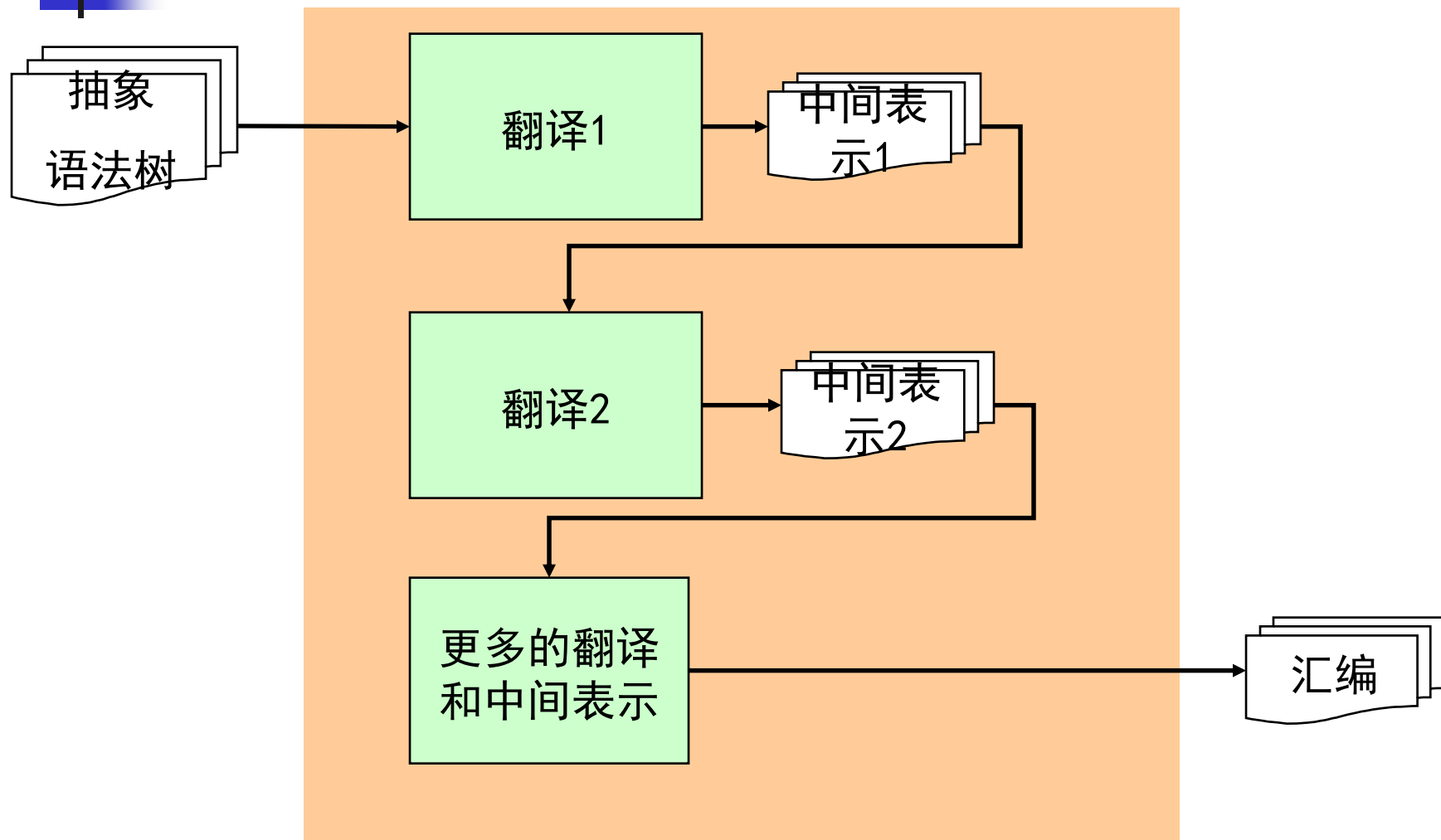
华保健

bjhua@ustc.edu.cn

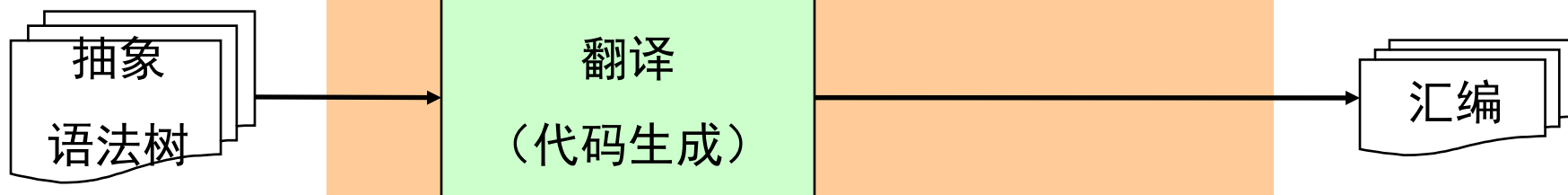
前端



中间端和后端



最简单的结构





代码生成的任务

- 负责把源程序翻译成“目标机器”上的代码
 - 目标机器：
 - 可以是真实物理机器
 - 可以是虚拟机
- 两个重要任务：
 - 给源程序的数据分配计算资源
 - 给源程序的代码选择指令



给数据分配计算资源

- 源程序的**数据**:
 - 全局变量、局部变量、动态分配等
- 机器计算**资源**:
 - 寄存器、数据区、代码区、栈区、堆区
- 根据程序的特点和编译器的设计目标，合理的为数据分配计算资源
 - 例如：变量放在内存里还是寄存器里？



给代码选择合适的机器指令

- 源程序的代码：
 - 表达式运算、语句、函数等
- 机器指令：
 - 算术运算、比较、跳转、函数调用返回
- 用机器指令实现高层代码的语义
 - 等价性
 - 对机器指令集体系统结构（ISA）的熟悉



路线图

- 为了讲解代码生成涉及的重要问题和解决方案，我们研究两种不同的ISA上的代码生成技术
 - 栈计算机Stack
 - 寄存器计算机Reg



代码生成：栈式计算机

编译原理

华保健

bjhua@ustc.edu.cn

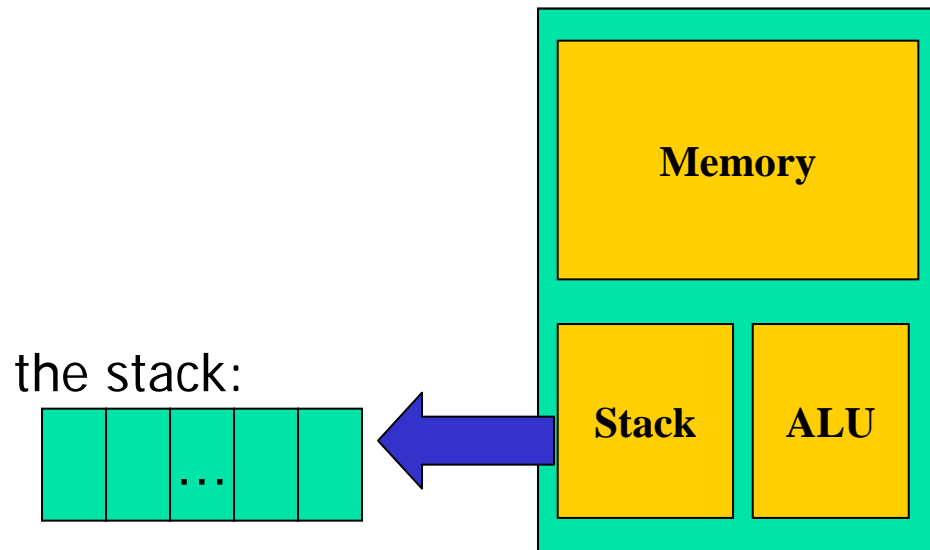


栈式计算机

- 栈式计算机在历史上非常流行
 - 上世纪70年代曾有很多栈式计算机
- 但今天已经基本上已经退出了历史舞台
 - 效率问题
- 我们还要讨论栈式计算机的代码生成技术的原因是：
 - 给栈式计算机生成代码是最容易的
 - 仍然有许多栈式的虚拟机
 - Pascal P code
 - Java virtual machine (JVM)
 - Postscript
 - ...

栈式计算机Stack的结构

- 内存
 - 存放所有的变量
- 栈
 - 进行运算的空间
- 执行引擎
 - 指令的执行



栈计算机的指令集

// 指令的语法

s -> push NUM

| load x

| store x

| add

| sub

| times

| div

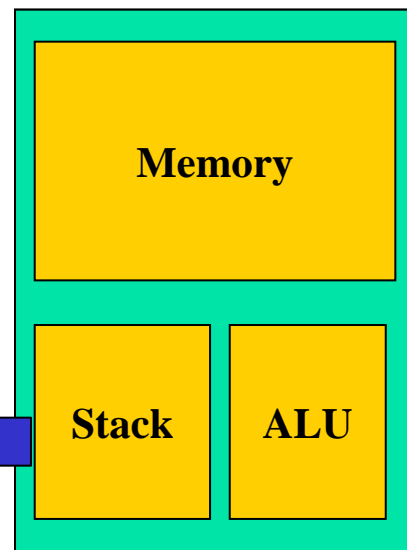
} 栈操作

} 访存

} 算术运算

是Java字节码的一个子集！

the stack:



指令的语义: push

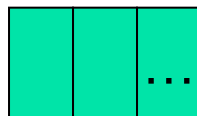
// 指令的语法

```
s -> push NUM
    | load x
    | store x
    | add
    | sub
    | times
    | div
```

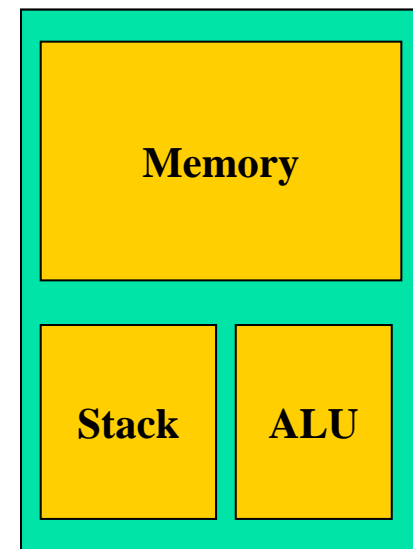
push NUM:

```
top++;
stack[top] = NUM;
```

执行前:



执行后:



指令的语义: load x

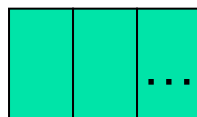
// 指令的语法

```
s -> push NUM
    | load x
    | store x
    | add
    | sub
    | times
    | div
```

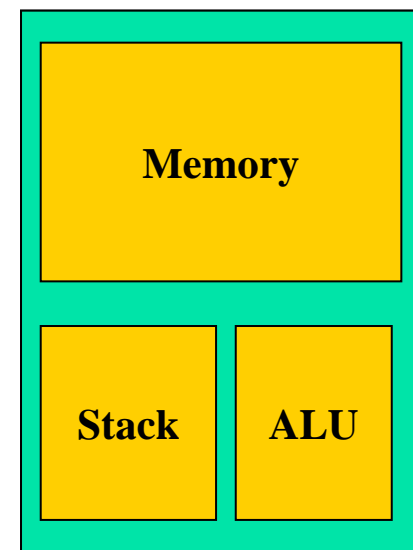
load x:

```
top++;
stack[top] = x;
```

执行前:



执行后:



指令的语义: store x

// 指令的语法

```
s -> push NUM  
    | load x  
    | store x  
    | add  
    | sub  
    | times  
    | div
```

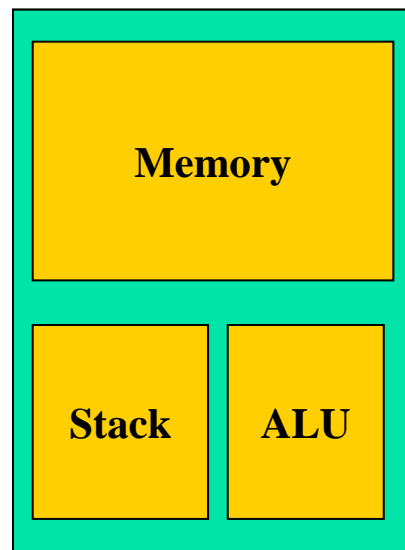
store x:

```
x = stack[top];  
top--;
```

执行前:



执行后:



指令的语义: add

// 指令的语法

```
s -> push NUM  
    | load x  
    | store x  
    | add  
    | sub  
    | times  
    | div
```

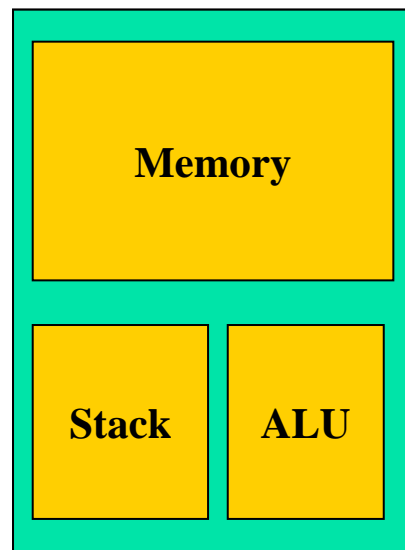
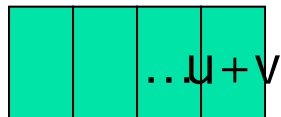
add:

```
temp = stack[top-1]  
      +stack[top];  
  
top -= 2;  
push temp;
```

执行前:



执行后:





变量的内存分配伪指令

- Stack机器只支持一种数据类型int，并且给变量x分配内存的伪指令是：
 - `.int x`
- Stack机器在装载一个程序时，就会读取伪指令，并且给相关变量分配内存空间



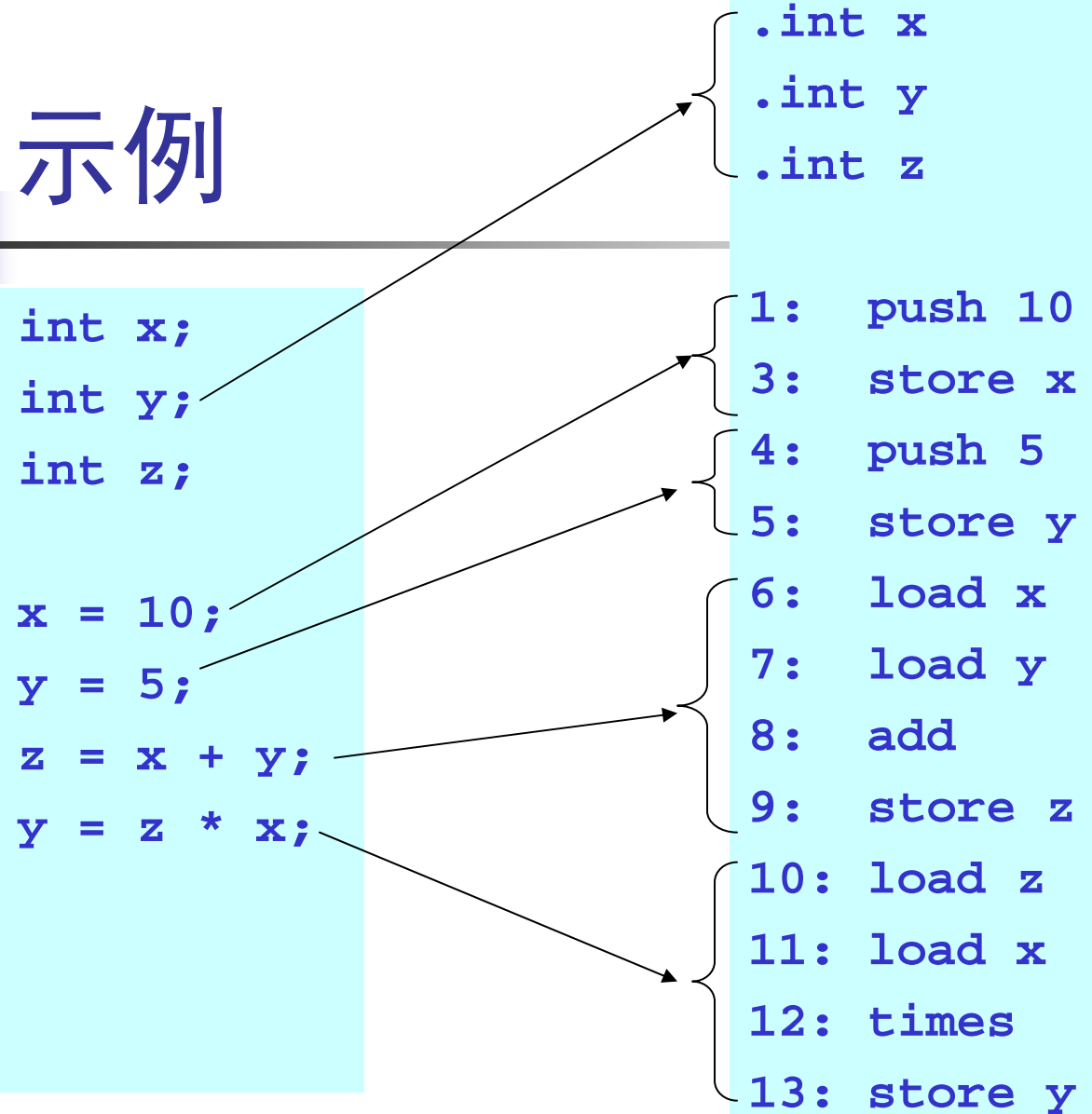
示例

```
int x;  
int y;  
int z;
```

```
x = 10;  
y = 5;  
z = x + y;  
y = z * x;
```

```
.int x  
.int y  
.int z
```

```
1:  push 10  
3:  store x  
4:  push 5  
5:  store y  
6:  load x  
7:  load y  
8:  add  
9:  store z  
10: load z  
11: load x  
12: times  
13: store y
```





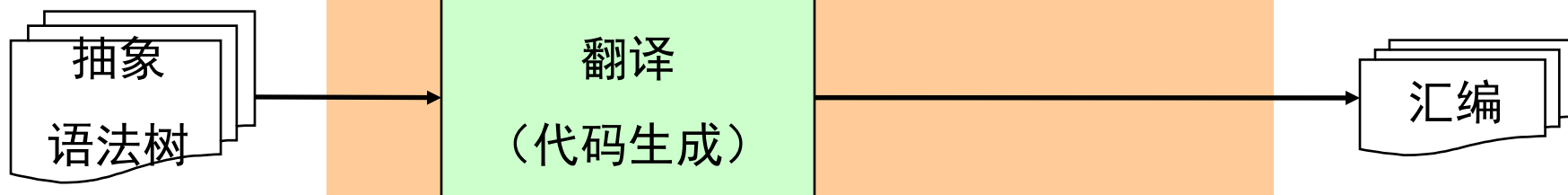
代码生成： 栈式计算机的代码生成

编译原理

华保健

bjhua@ustc.edu.cn

最简单的结构





递归下降代码生成算法： 从C--到Stack

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

```
// 要写如下几个
// 递归函数：
Gen_P(D S);
Gen_D(T id; D);
Gen_T(T);
Gen_S(S);
Gen_E(E);
```

```
// 指令的语法
s -> push NUM
    | load x
    | store x
    | add
    | sub
    | times
    | div
```

递归下降代码生成算法： 表达式的代码生成

// 不变式：表达式的值总在栈顶

```
Gen_E(E e)
    switch (e)
        case n: emit ("push n"); break;
        case id: emit ("load id"); break;
        case true: emit ("push 1"); break;
        case false: emit ("push 0"); break;
        case e1+e2: Gen_E (e1);
                    Gen_E (e2);
                    emit ("add");
                    break;
        case ...: ... // similar
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 语句的代码生成

// 不变式：栈的规模不变

```
Gen_S(S s)
    switch (s)
        case id=e: Gen_E(e);
                    emit("store id");
                    break;
        case printi(e): Gen_E(e);
                        emit ("printi");
                        break;
        case printb(e): Gen_E(e);
                        emit ("printb");
                        break;
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 类型的代码生成

// 不变式：只生成.int类型

```
Gen_T(T t)
    switch (t)
        case int: emit (".int");
                  break;
        case bool: emit (".int");
                  break;
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

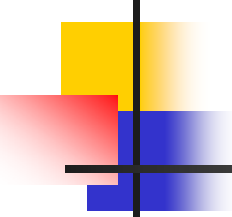
递归下降代码生成算法： 变量声明的代码生成

// 不变式：只生成.int类型

```
Gen_D(T id; D)
    Gen_T (T);
    emit (" id");
    Gen_D (D);
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```


递归下降代码生成算法： 程序的代码生成



```
Gen_P(D S)
  Gen_D (D);
  Gen_S (S);
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```



示例

```
int x;  
int y;  
int z;
```

```
x = 10;
```

```
y = 5;
```

```
z = x + y;
```

```
y = z * x;
```

```
.int x  
.int y  
.int z
```

```
1:  push 10
```

```
3:  store x
```

```
4:  push 5
```

```
5:  store y
```

```
6:  load x
```

```
7:  load y
```

```
8:  add
```

```
9:  store z
```

```
10: load z
```

```
11: load x
```

```
12: times
```

```
13: store y
```



如何运行生成的代码？

- 找一台真实的物理机
- 写一个虚拟机（解释器）
 - 类似于JVM
- 在非栈式计算机上进行模拟：
 - 例如，可以在x86上模拟栈式计算机的行为
 - 用x86的调用栈模拟栈



代码生成：

寄存器计算机及其代码生成

编译原理

华保健

bjhua@ustc.edu.cn

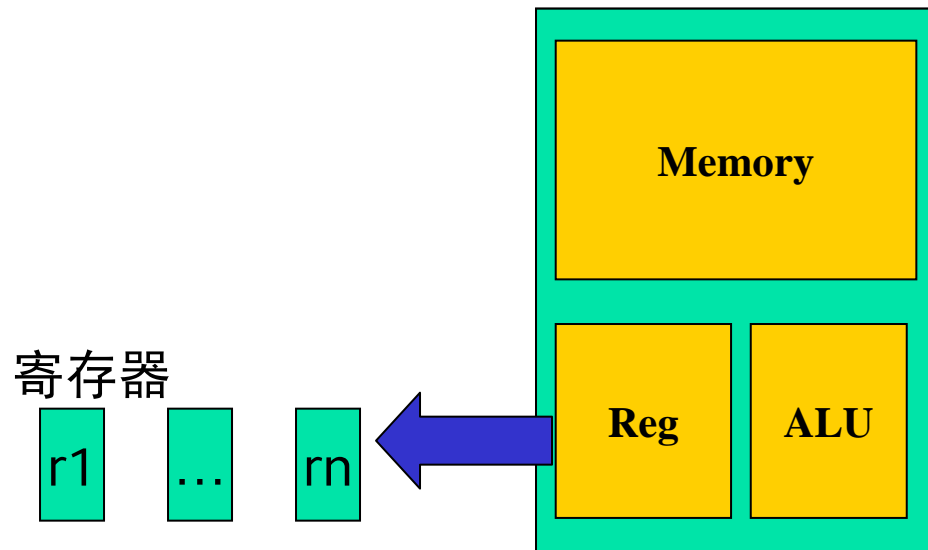


寄存器计算机

- 寄存器计算机是目前最流行的机器体系结构之一
 - 效率很高
 - 机器体系结构规整
- 机器基于寄存器架构：
 - 典型的有16、32或更多个寄存器
 - 所有操作都在寄存器中进行
 - 访存都通过load/store进行
 - 内存不能直接运算

寄存器计算机Reg的结构

- 内存
 - 存放溢出的变量
- 寄存器
 - 进行运算的空间
 - 假设有无限多个
- 执行引擎
 - 指令的执行



寄存器计算机的指令集

// 指令的语法

s -> movn n, r

| mov r1, r2

| load [x], r

| store r, [x]

| add r1, r2, r3

| sub r1, r2, r3

| times r1, r2, r3

| div r1, r2, r3

数据移动

访存

算术运算

寄存器

r1

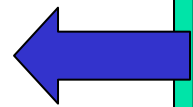
...

rn

Memory

Reg

ALU





变量的寄存器分配伪指令

- Reg机器只支持一种数据类型int，并且给变量x分配寄存器的伪指令是：
 - .int x
- 在代码生成的阶段，假设Reg机器上有无限多个寄存器
 - 因此每个声明变量和临时变量都会占用一个（虚拟）寄存器
 - 把虚拟寄存器分配到物理寄存器的过程称为寄存器分配

递归下降代码生成算法： 从C--到Reg

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

// 要写如下几个

// 递归函数：

```
void Gen_P(D S);
void Gen_D(T id; D);
void Gen_T(T);
void Gen_S(S);
R_t Gen_E(E);
```

// 指令的语法

```
s -> movn n, r
    | mov r1, r2
    | load [x], r
    | store r, [x]
    | add r1, r2, r3
    | sub r1, r2, r3
    | times r1, r2, r3
    | div r1, r2, r3
```

递归下降代码生成算法： 表达式的代码生成

// 不变式：表达式的值在函数返回的寄存器中

```
R_t Gen_E(E e)
    switch (e)
        case n: r = fresh();
                emit ("movn n, r");
                return r;
        case id: r = fresh ();
                emit ("mov id, r");
                return r;
        case true: r = fresh ();
                  emit ("movn 1, r");
                  return r;
        case false: r = fresh ();
                    emit ("movn 0, r");
                    return r;
```

// 下页继续

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 表达式的代码生成

// 不变式：表达式的值在函数返回的寄存器中

```
R Gen_E(E e)
  switch (e)
    case e1+e2: r1 = Gen_E(e1);
                r2 = Gen_E(e2);
                r = fresh();
                emit ("add r1, r2, r");
                return r;
    case e1&&e2: r1 = Gen_E(e1);
                r2 = Gen_E(e2);
                r = fresh();
                emit ("and r1, r2, r");
                return r; // 非短路
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 语句的代码生成

```
Gen_S(S s)
  switch (s)
    case id=e: r = Gen_E(e);
               emit("mov r, id");
               break;
    case printi(e): r = Gen_E(e);
                   emit ("printi r");
                   break;
    case printb(e): r = Gen_E(e);
                   emit ("printb r");
                   break;
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 类型的代码生成

// 不变式：只生成.int类型

```
Gen_T(T t)
    switch (t)
        case int: emit (".int");
                  break;
        case bool: emit (".int");
                  break;
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

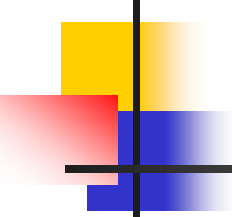
递归下降代码生成算法： 变量声明的代码生成

// 不变式：只生成.int类型

```
Gen_D(T id; D)
    Gen_T (T);
    emit (" id");
    Gen_D (D);
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 程序的代码生成



```
Gen_P(D S)
  Gen_D (D);
  Gen_S (S);
```

```
P -> D S
D -> T id; D
    |
T -> int
    | bool
S -> id = E
    | printi (E)
    | printb (E)
E -> n
    | id
    | true
    | false
    | E + E
    | E && E
```



示例

```
int x;  
int y;  
int z;
```

```
x = 1+2+3+4;  
y = 5;  
z = x + y;  
y = z * x;
```

```
.int x  
.int y  
.int z
```

```
1:  movn 1, r1  
3:  movn 2, r2  
4:  add r1, r2, r3  
5:  movn 3, r4  
6:  add r3, r4, r5  
7:  movn 4, r6  
8:  add r5, r6, r7  
9:  mov r7, x  
10: movn 5, r8  
11: mov r8, y
```




如何运行生成的代码？

- 写一个虚拟机（解释器）
- 在真实的物理机器上运行：
 - 需进行寄存器分配



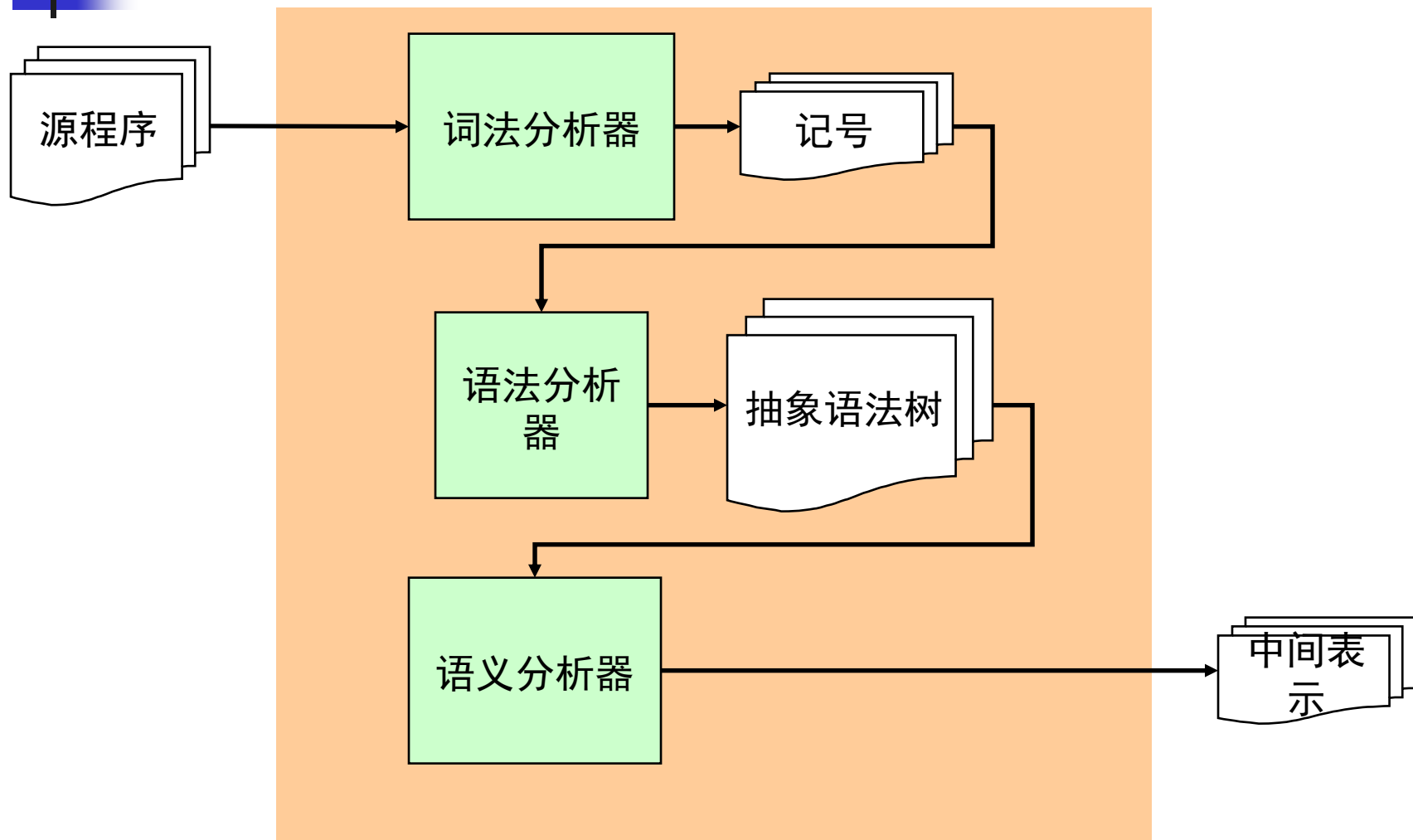
中间表示： 中间代码的地位和作用

编译原理

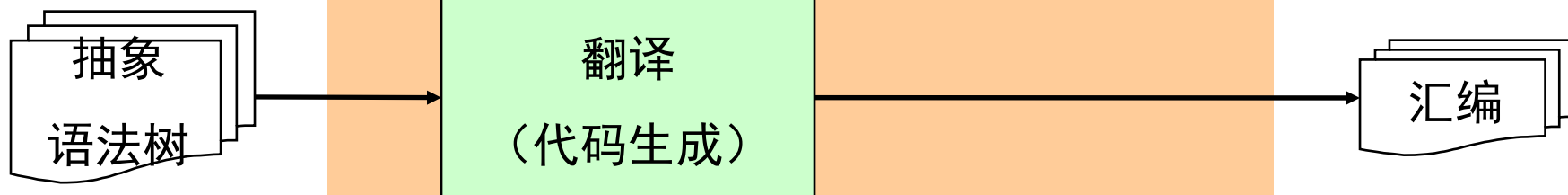
华保健

bjhua@ustc.edu.cn

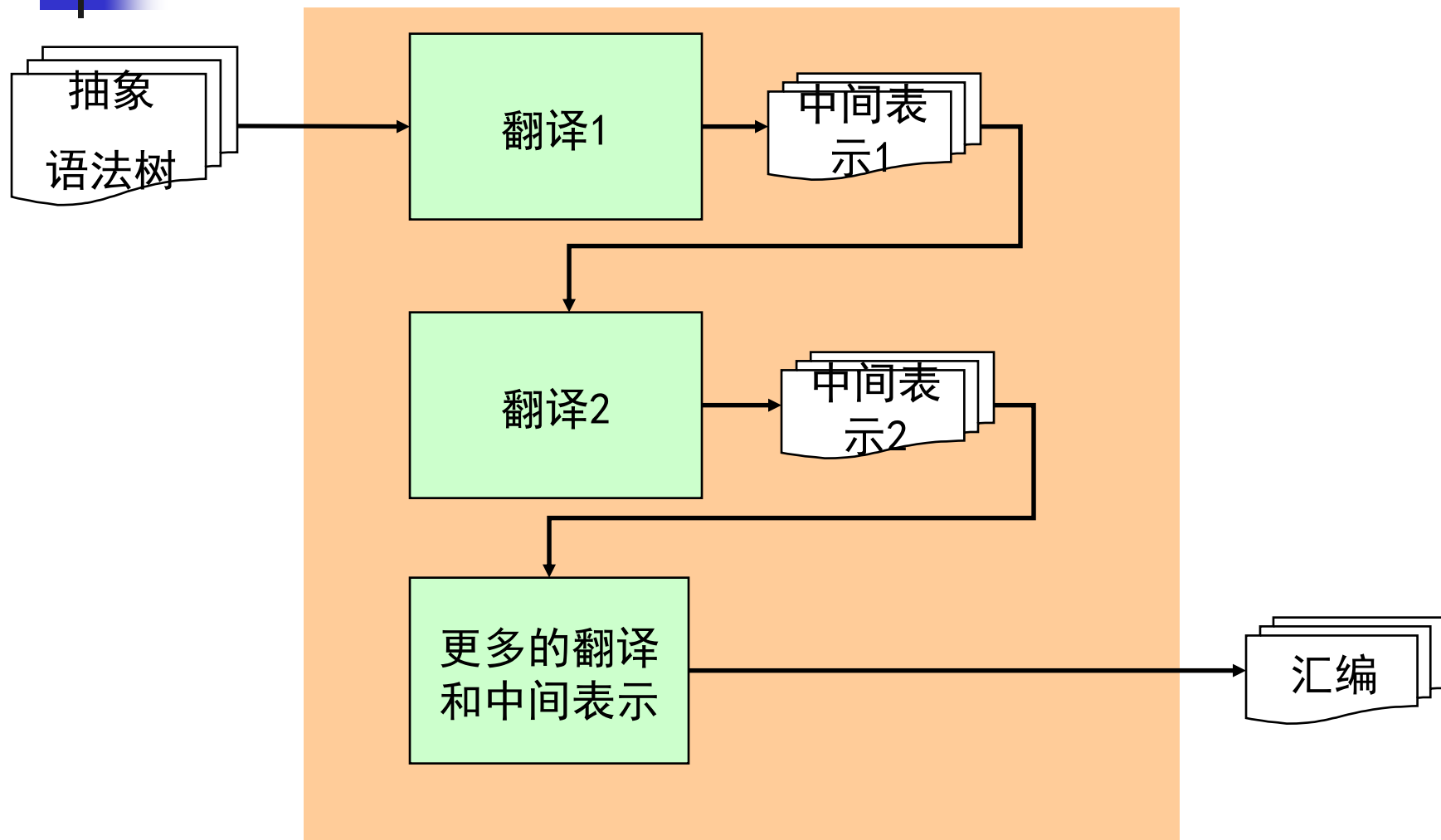
前端



最简单的结构



中间端和后端

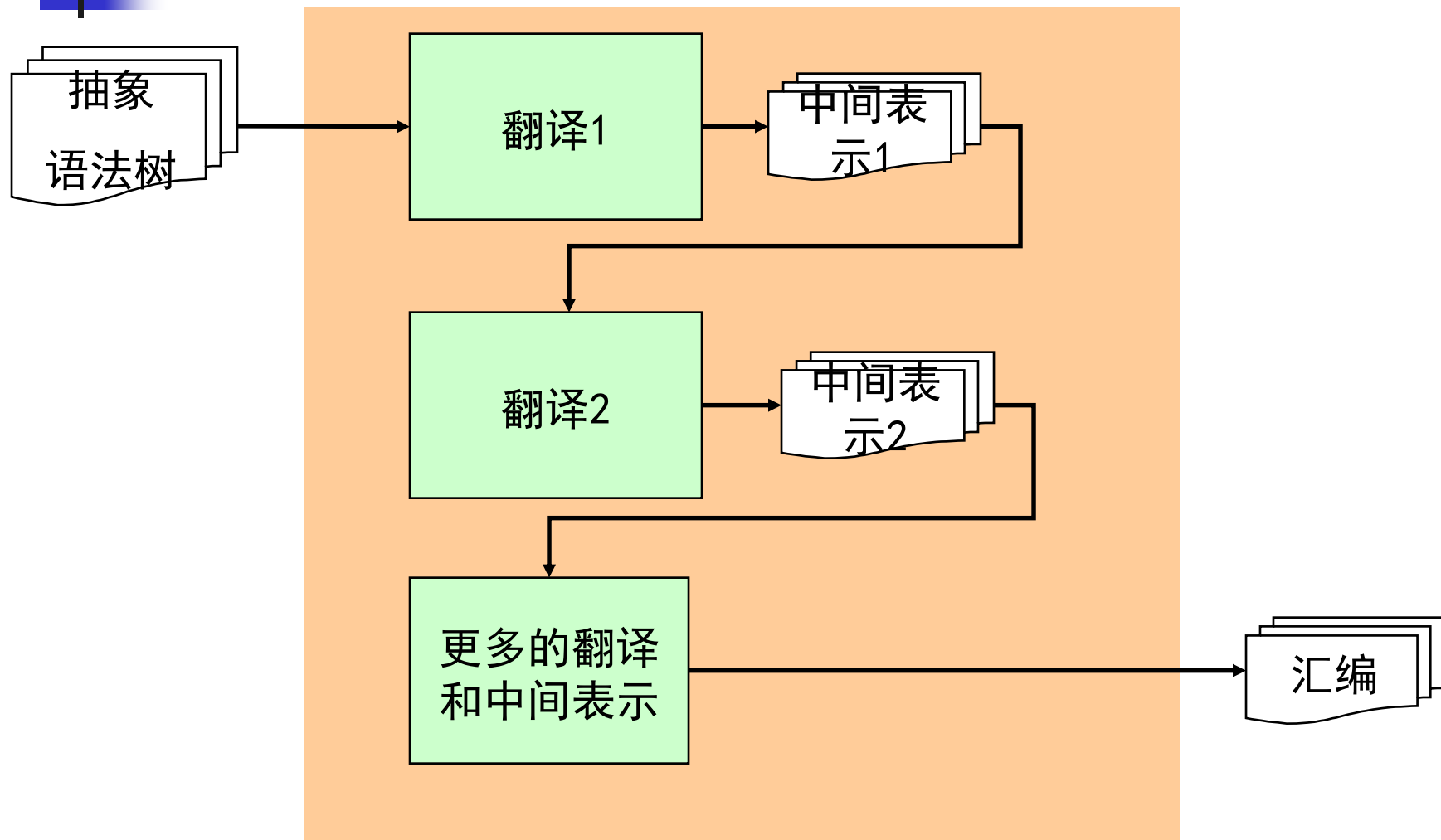




中间代码

- 树和有向无环图（DAG）
 - 高层表示，适用于程序源代码
- 三地址码（3-address code）
 - 低层表示，靠近目标机器
- 控制流图（CFG）
 - 更精细的三地址码，程序的图状表示
 - 适合做程序分析、程序优化等
- 静态单赋值形式（SSA）
 - 更精细的控制流图
 - 同时编码控制流信息和数据流信息
- 连续传递风格（CPS）
 - 更一般的SSA

为什么要划分成不同的中间表示？

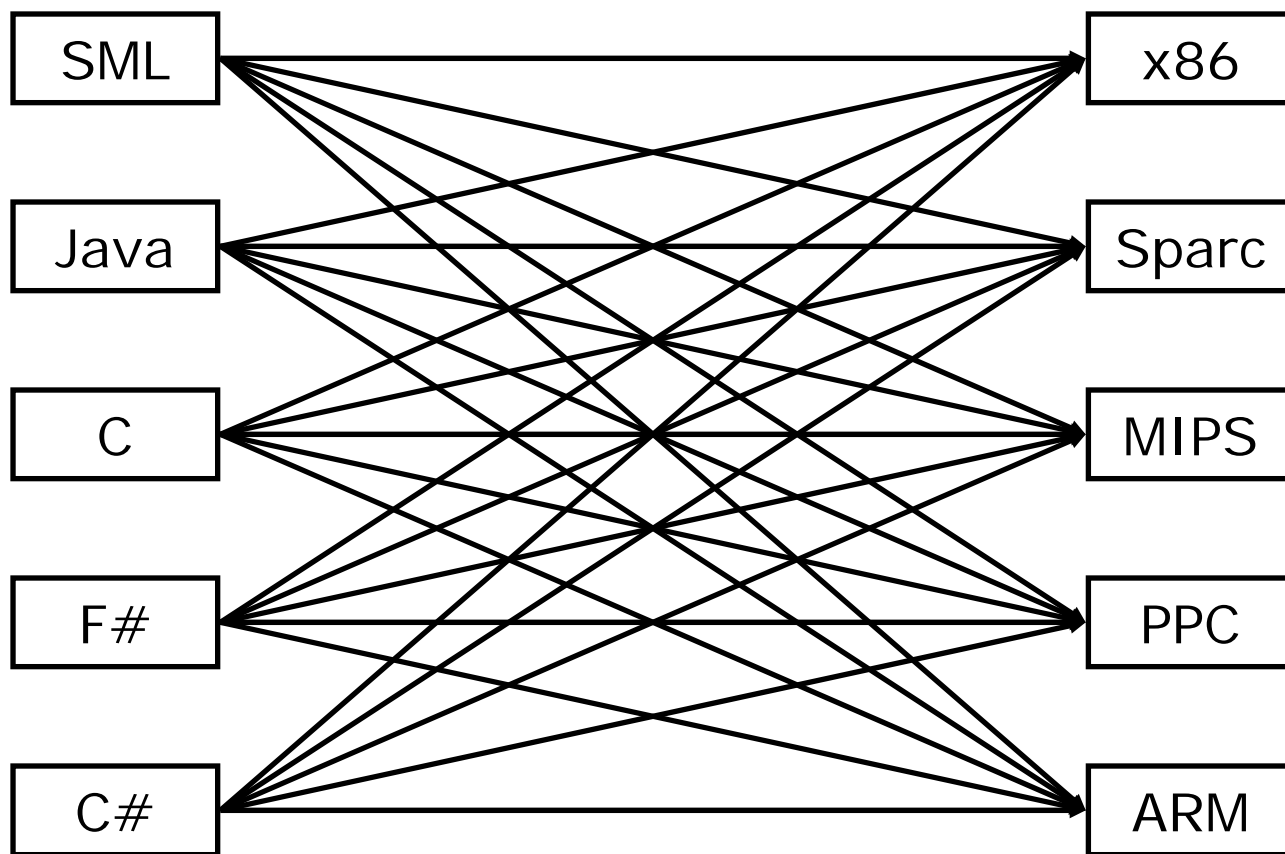




为什么要划分成不同的中间表示？

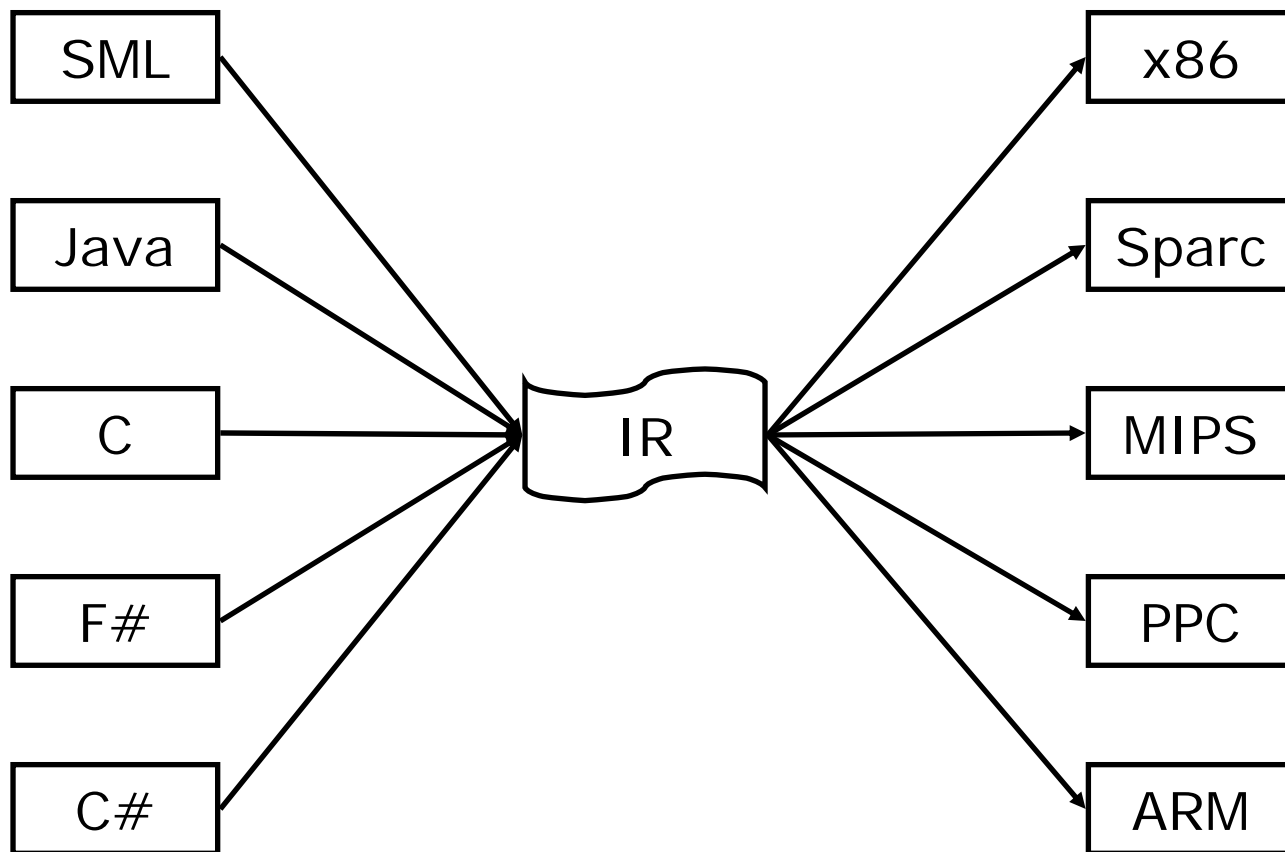
- 编译器工程上的考虑
 - 阶段划分：把整个编译过程划分成不同的阶段
 - 任务分解：每个阶段只处理翻译过程的一个步骤
 - 代码工程：代码更容易实现、除错、维护和演进
- 程序分析和代码优化的需要
 - 两者都和程序的中间表示密切相关
 - 许多优化在特定的中间表示上才可以或才容易进行
 - 课程后续我们继续深入这一话题

通用编译器语言？



$n \times m$ 个编译器

通用编译器语言？



$n+m$ 个编译器



路线图

- 全面讨论中间表示涉及的重要问题和解决方案
 - 详细介绍现代编译器中几种常用的重要中间表示
 - 三地址码
 - 控制流图
 - 静态单赋值形式
 - 详细介绍在中间表示上做程序分析的理论和技术
 - 控制流分析
 - 数据流分析
- 为下一部分讨论代码优化打下基础



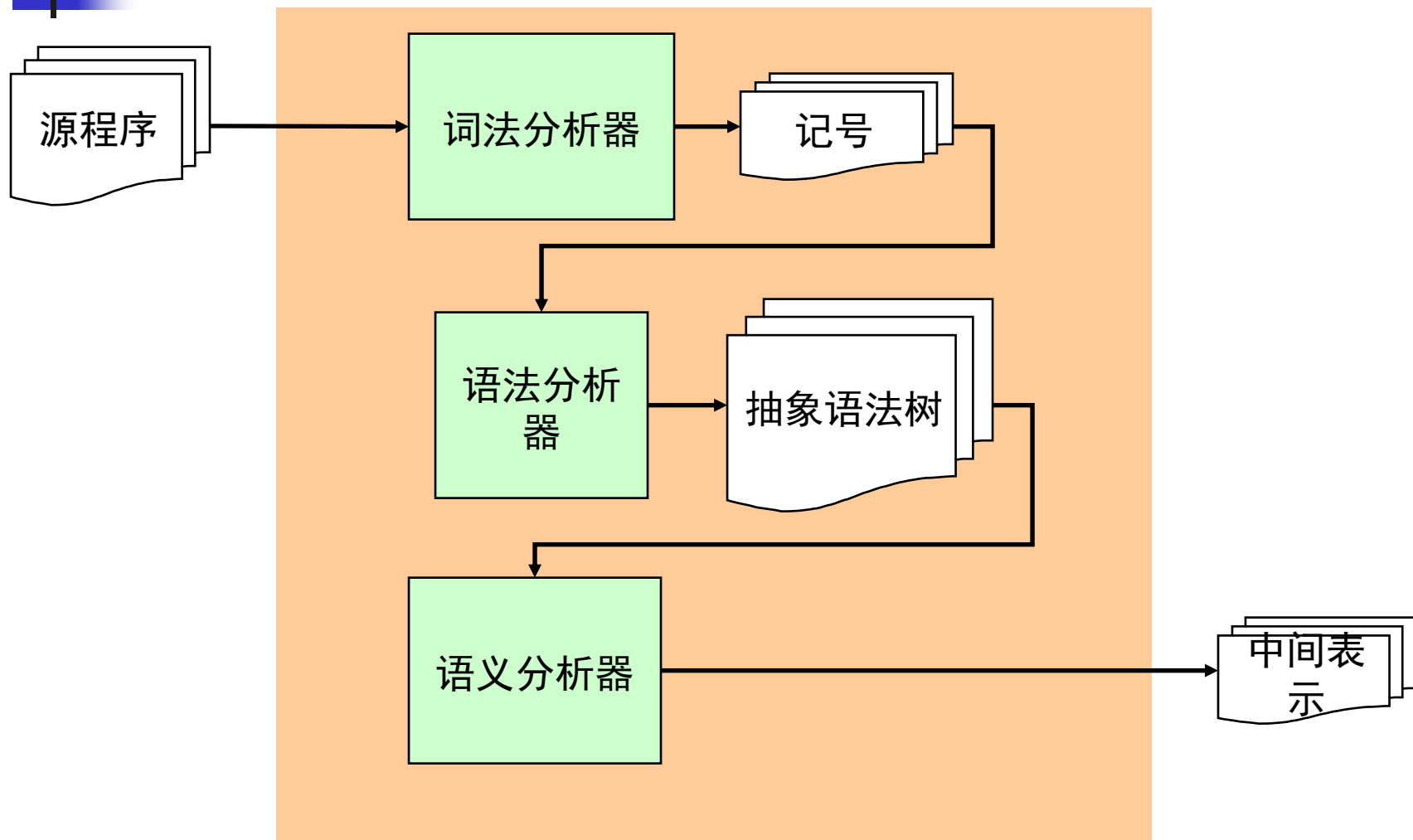
中间表示：三地址码

编译原理

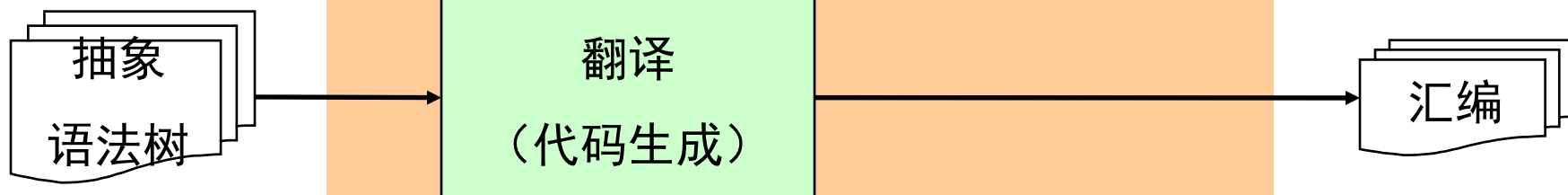
华保健

bjhua@ustc.edu.cn

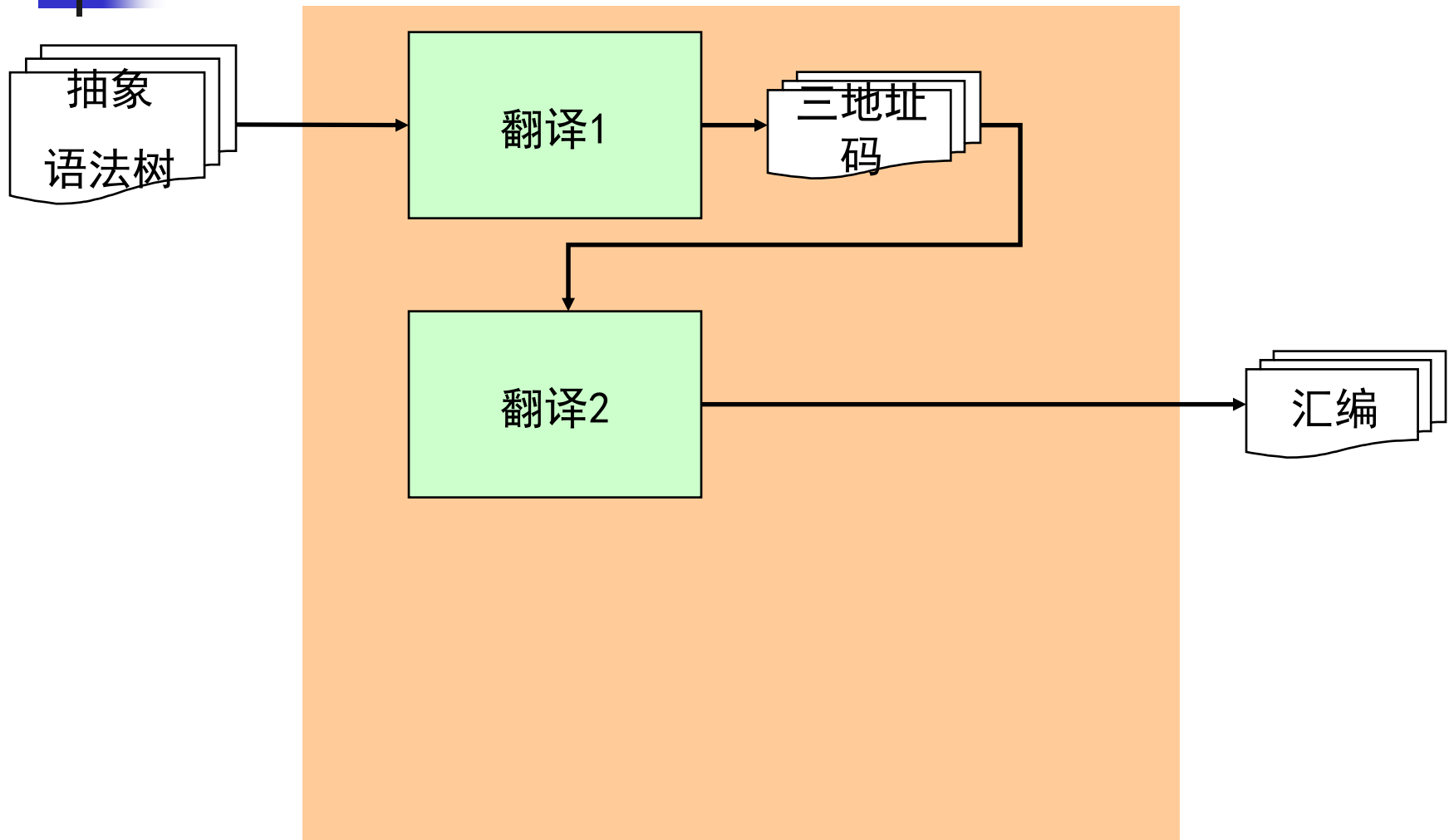
前端



最简单的结构



使用三地址码的编译器结构





三地址码的基本思想

- 给每个中间变量和计算结果命名
 - 没有复合表达式
- 只有最基本的控制流
 - 没有各种控制结构
 - 只有goto, call等
- 所以三地址码可以看成是抽象的指令集
 - 通用的RISC



示例

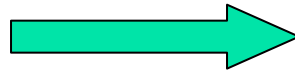
```
a = 3 + 4 * 5;
```

```
if (x < y)
```

```
    z = 6;
```

```
else
```

```
    z = 7;
```



```
x_1 = 4;
```

```
x_2 = 5;
```

```
x_3 = x_1 * x_2;
```

```
x_4 = 3;
```

```
x_5 = x_4 + x_3;
```

```
a = x_5;
```

```
Cjmp (x<y, L_1, L_2);
```

```
L_1:
```

```
    z = 6;
```

```
    jmp L_3;
```

```
L_2:
```

```
    z = 7;
```

```
    jmp L_3;
```

```
L_3:
```

```
...
```



三地址码的定义

```
s -> x = n           // 常数赋值
| x = y  $\oplus$  z       // 二元运算
| x =  $\Theta$  y         // 一元运算
| x = y               // 数据移动
| x[y] = z            // 内存写
| x = y[v]            // 内存读
| x = f (x1, ..., xn) // 函数调用
| Cjmp (x1, L1, L2)   // 条件跳转
| Jmp L               // 无条件跳转
| Label L             // 标号
| Return x            // 函数返回
```

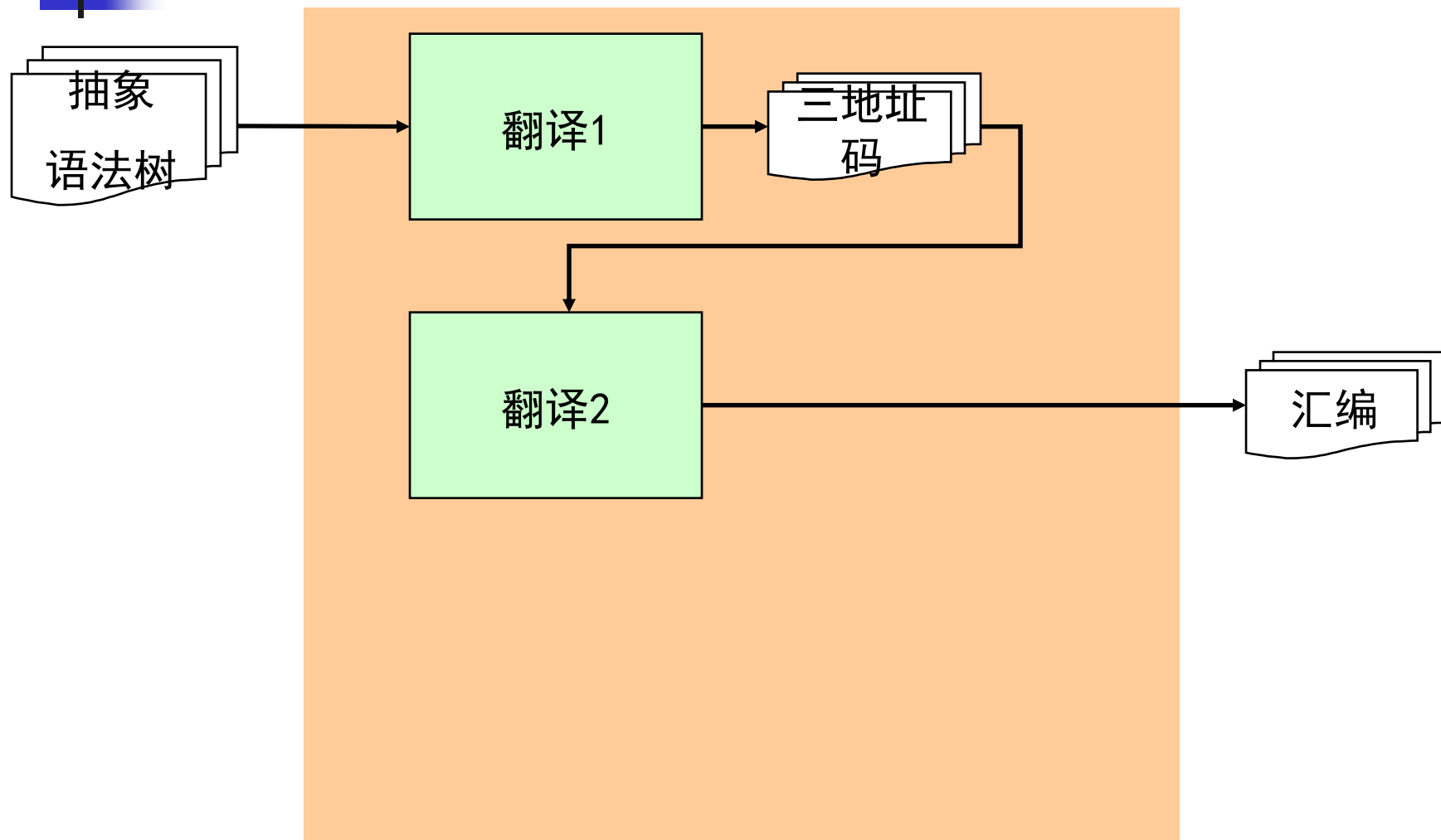


如何定义三地址码数据结构？

```
enum instr_kind {INSTR_CONST, INSTR_MOVE, ...};
struct Instr_t {enum instr_kind kind;};
struct Instr_Add {
    enum instr_kind kind;
    char *x;
    char *y;
    char *z;
};
struct Instr_Move {
    ...;
};
```

其它的编码留作练习。

如何生成三地址码？



从C--生成三地址码

```
P -> F*
F -> x ((T id,)*) { (T id;)* S* }
T -> int
    | bool
S -> x = E
    | printi (E)
    | printb (E)
    | x (E1, ..., En)
    | return E
    | if (E, S*, S*)
    | while (E, S*)
```

// 要写如下几个递归函数：

```
Gen_P(P); Gen_F(F); Gen_T(T);
Gen_S(S); Gen_E(E);
```

// 续：表达式的语法

```
E -> n
    | x
    | true
    | false
    | E + E
    | E && E
```

递归下降代码生成算法： 语句的代码生成（I）

```
Gen_S(S s)
  switch (s)
    case x=e:
      x1 = Gen_E(e);
      emit("x = x1");
      break;
    case printi(e):
      x = Gen_E(e);
      emit ("printi(x)");
      break;
    case printb(e):
      x = Gen_E(e);
      emit ("printb(x)");
      break;
```

```
S -> x = E
    | printi (E)
    | printb (E)
    | x (E1, ..., En)
    | return E
    | if (E, S*, S*)
    | while (E, S*)
```

递归下降代码生成算法： 语句的代码生成（II）

```
case x(e1, ..., en):  
    x1 = Gen_E(e1);  
    ...;  
    xn = Gen_E(en);  
    emit("x(x1, ..., xn)");  
    break;  
case return e;  
    x = Gen_E(e);  
    emit("return x");  
    break;
```

```
S -> x = E  
    | printi (E)  
    | printb (E)  
    | x (E1, ..., En)  
    | return E  
    | if (E, S*, S*)  
    | while (E, S*)
```

递归下降代码生成算法： 语句的代码生成（III）

```
case if(e, s1, s2):  
    x = Gen_E(e);  
    emit ("Cjmp(x, L1, L2)");  
    emit ("Label L1:");  
    Gen_SList(s1);  
    emit ("jmp L3");  
    emit ("Label L2:");  
    Gen_SList(s2);  
    emit ("jmp L3");  
    emit ("Label L3:");  
    break;
```

```
S -> x = E  
    | printi (E)  
    | printb (E)  
    | x (E1, ..., En)  
    | return E  
    | if (E, S*, S*)  
    | while (E, S*)
```


递归下降代码生成算法： 语句的代码生成（IIII）

```
case while(e, s):  
    emit ("Label L1:");  
    x = Gen_E(e);  
    emit ("Cjmp(x, L2, L3)");  
    emit ("Label L2:");  
    Gen_SList(s);  
    emit ("jmp L1");  
    emit ("Label L3:");  
    break;
```

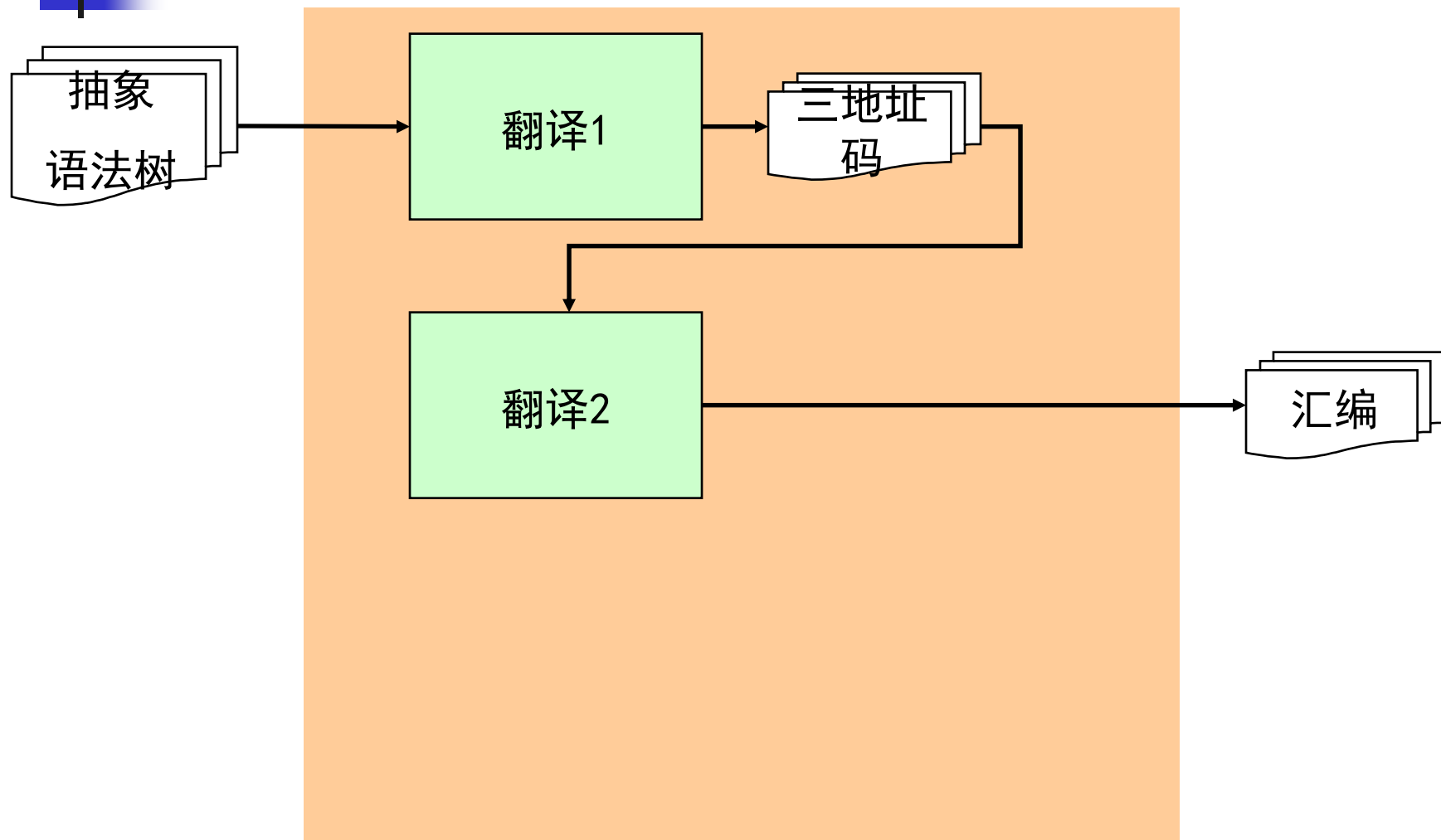
```
S -> x = E  
    | printi (E)  
    | printb (E)  
    | x (E1, ..., En)  
    | return E  
    | if (E, S*, S*)  
    | while (E, S*)
```



小结

- 三地址码的优点：
 - 所有的操作是原子的
 - 变量！没有复合结构
 - 控制流结构被简化了
 - 只有跳转
 - 是抽象的机器代码
 - 向后做代码生成更容易
- 三地址码的不足：
 - 程序的控制流信息是隐式的
 - 可以做进一步的控制流分析

从三地址码生成机器指令





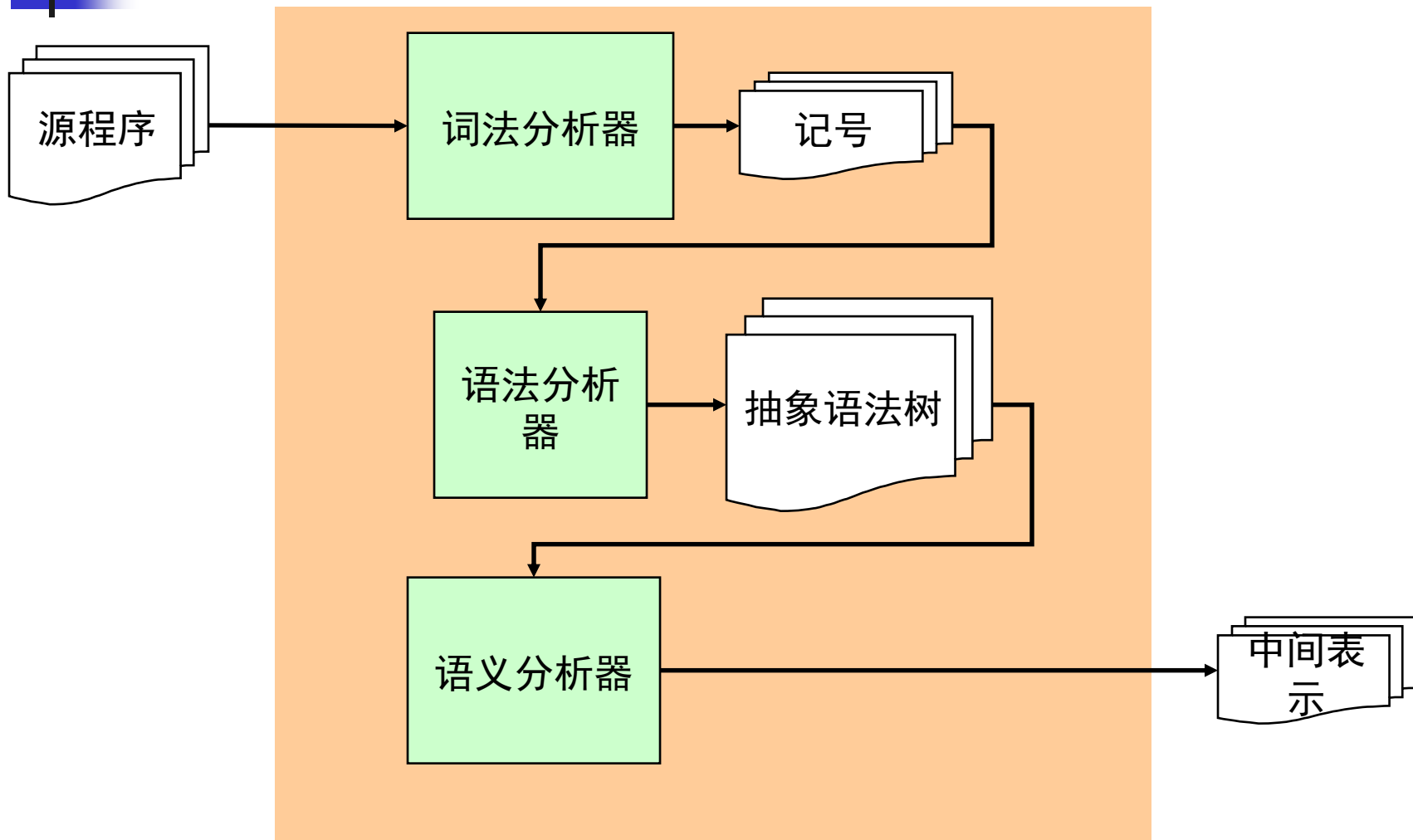
中间表示：控制流图

编译原理

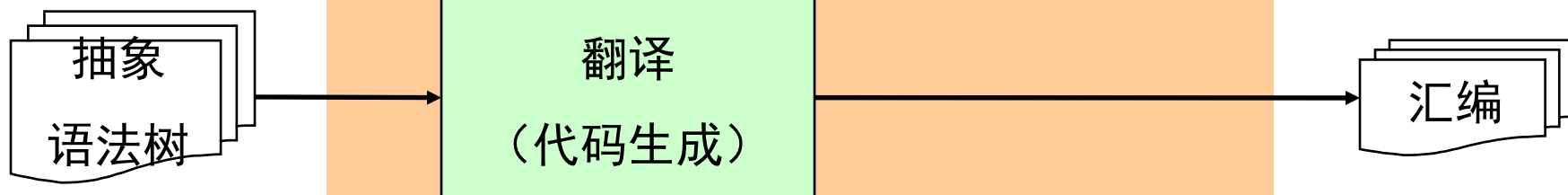
华保健

bjhua@ustc.edu.cn

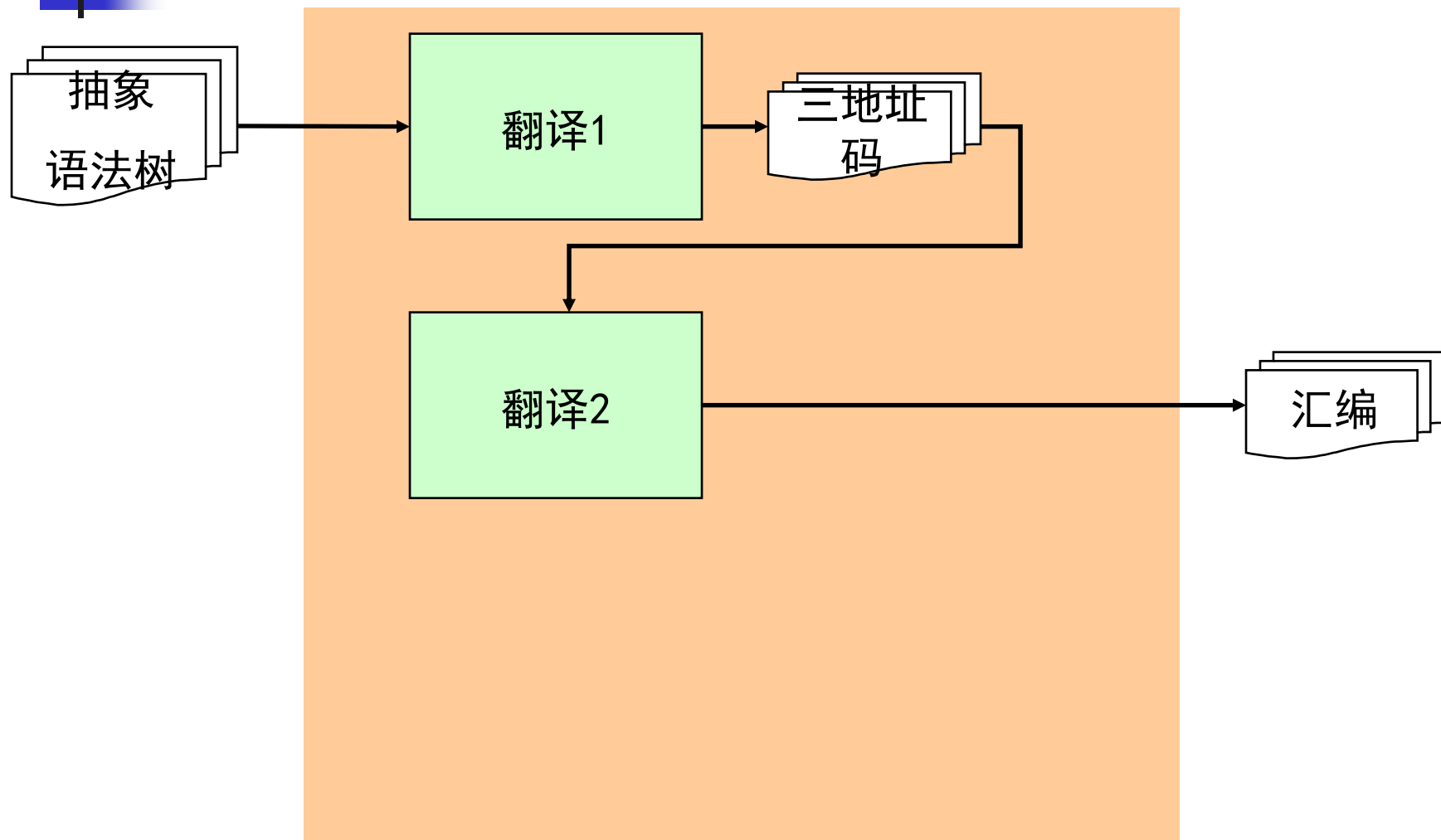
前端



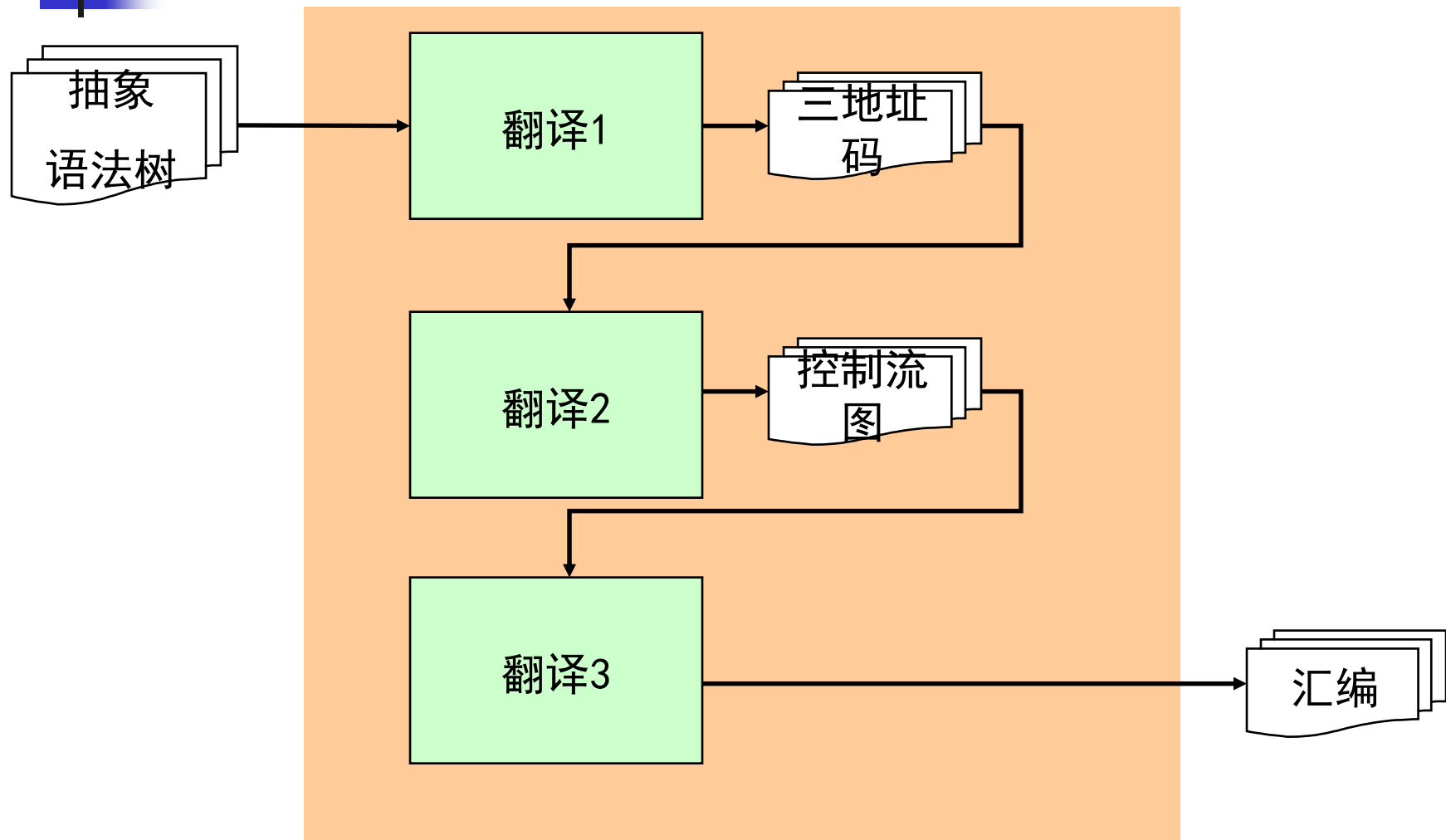
最简单的结构



使用三地址码的编译器结构



使用控制流图的编译器结构



三地址码 的不足

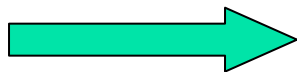
```
a = 3 + 4 * 5;
```

```
if (x < y)
```

```
    z = 6;
```

```
else
```

```
    z = 7;
```



```
x_1 = 4;
```

```
x_2 = 5;
```

```
x_3 = x_1 * x_2;
```

```
x_4 = 3;
```

```
x_5 = x_4 + x_3;
```

```
a = x_5;
```

```
Cjmp (x<y, L_1, L_2);
```

```
L_1:
```

```
    z = 6;
```

```
    jmp L_3;
```

```
L_2:
```

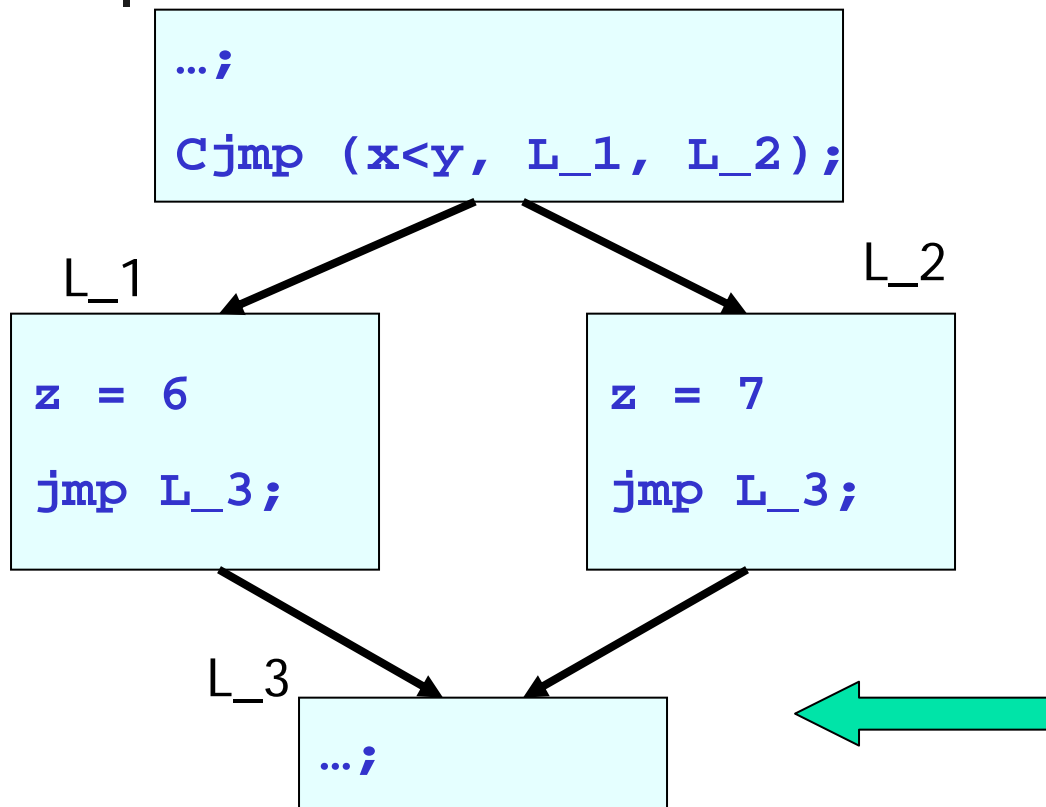
```
    z = 7;
```

```
    jmp L_3;
```

```
L_3:
```

```
...
```

控制结构



```
x_1 = 4;  
x_2 = 5;  
x_3 = x_1 * x_2;  
x_4 = 3;  
x_5 = x_4 + x_3;  
a = x_5;  
Cjmp (x<y, L_1, L_2);  
L_1:  
    z = 6;  
    jmp L_3;  
L_2:  
    z = 7;  
    jmp L_3;  
L_3:  
    ...
```



评论

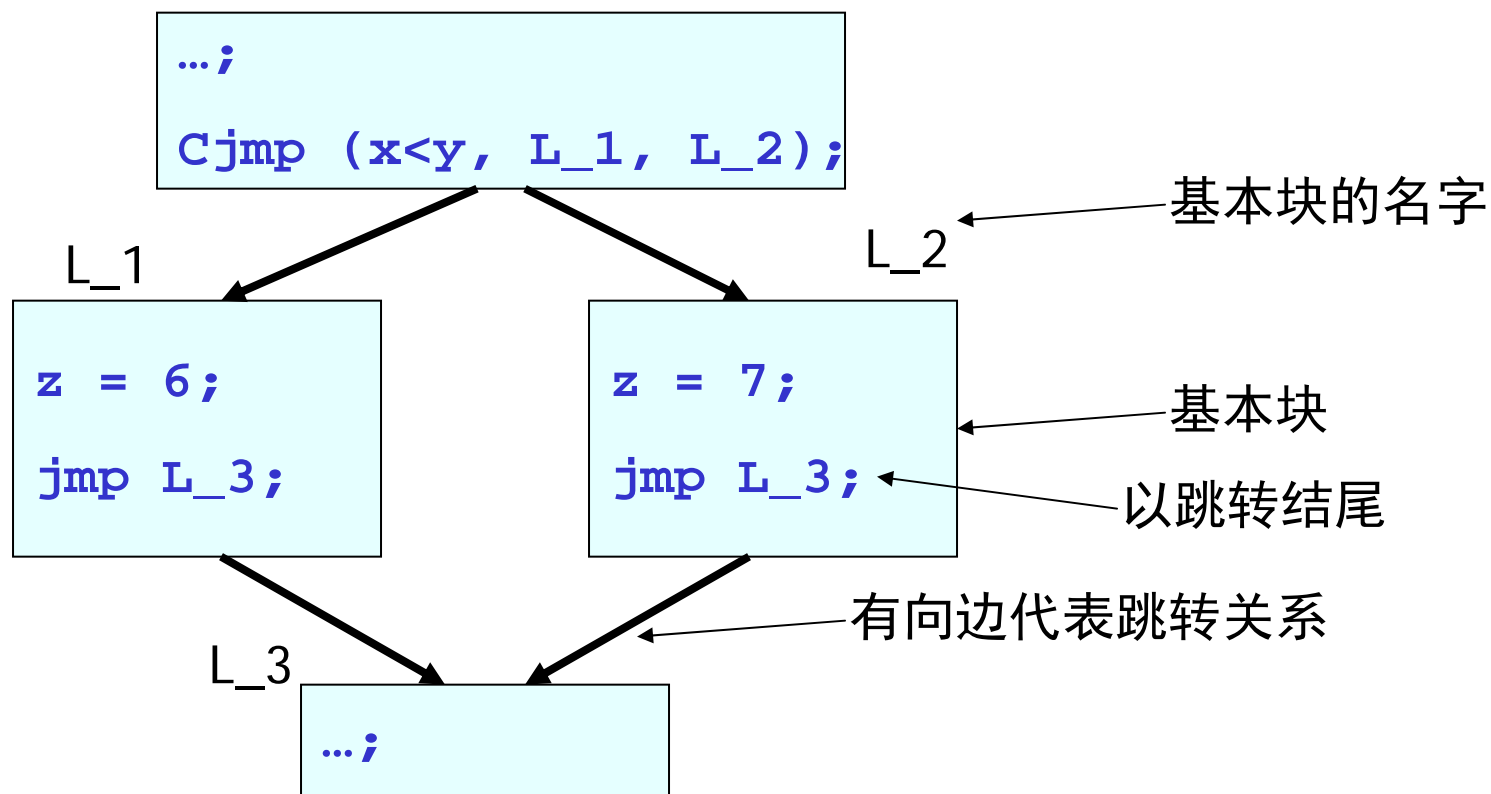
- 程序的**控制流图**表示带来很多好处:
 - 控制流分析:
 - 对很多程序分析来说, 程序的内部结构很重要
 - 典型的问题: “程序中是否存在循环?”
 - 可以进一步进行其他分析:
 - 例如**数据流分析**
 - 典型的问题: “程序第5行的变量x可能的值是什么?”
- 现代编译器的早期阶段就会倾向做控制流分析
 - 方便后续阶段的分析



基本概念

- **基本块**：是语句的一个序列，从第一条执行到最后一条
 - 不能从中间进入
 - 不能从中间退出
 - 即跳转指令只能出现在最后
- **控制流图**：控制流图是一个有向图 $G=(V, E)$
 - 节点 V ：是基本块
 - 边 E ：是基本块之间的跳转关系

示例





控制流图的定义

// 是更精细的三地址码

```
S -> x = n
    | x = y + z
    | x = y
    | x = f (x1, x2, ..., xn)
J -> jmp L
    | cjmp (x, L1, L2)
    | return x
B -> Label L;
    S1; S2; ...; Sn
    J
F -> x() { B1, ..., Bn }
P -> F1, ..., Fn
```

// 数据结构定义（以B为例）

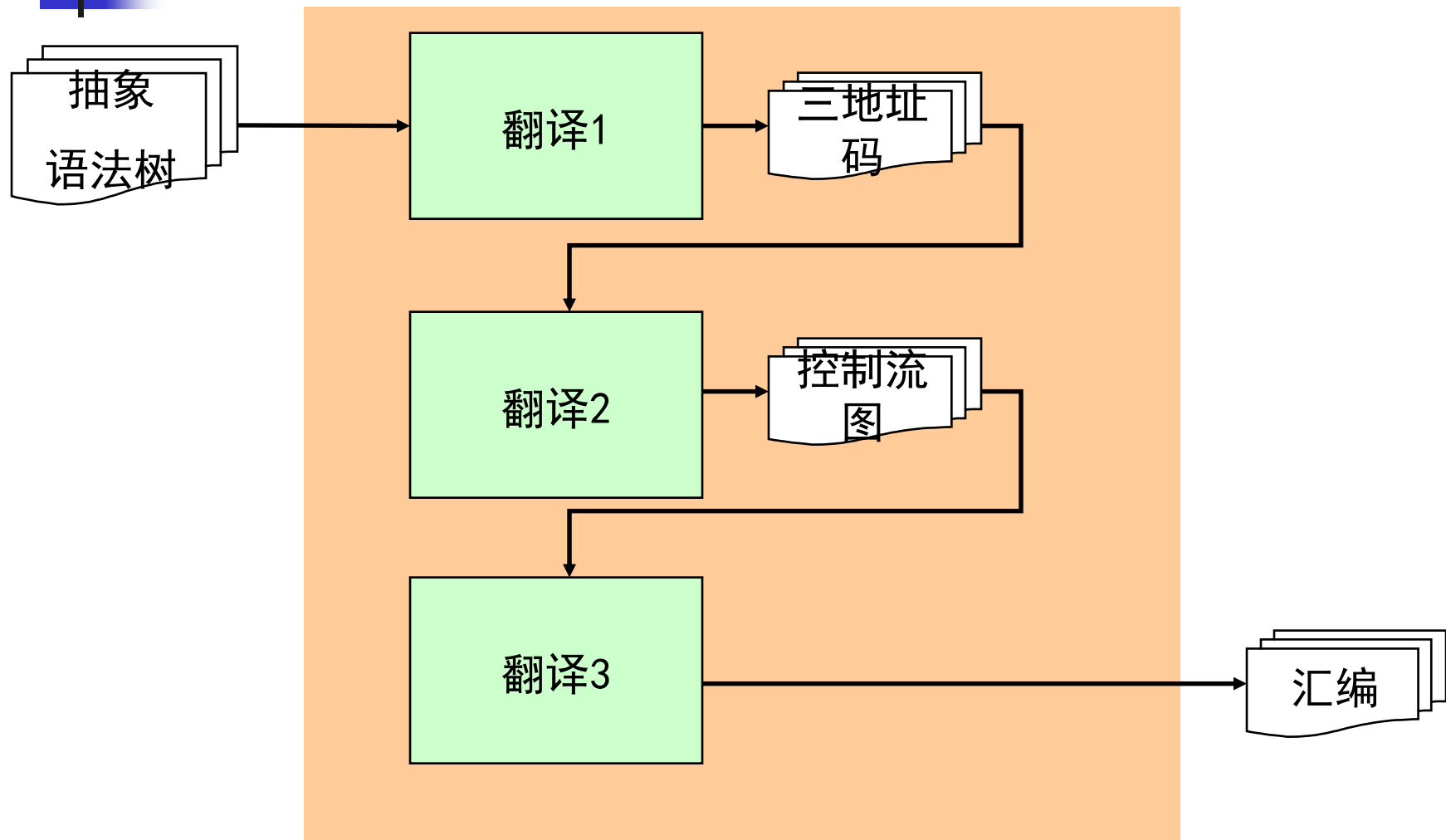
```
struct Block{
    Label_t label;
    List_t stms;
    Jump_t j;
};
```



如何生成控制流图？

- 可以直接从抽象语法树生成：
 - 如果高层语言具有特别规整控制流结构的话较容易
- 也可以先生成三地址码，然后继续生成控制流图：
 - 对于像C这样的语言更合适
 - 包含像goto这样的非结构化的控制流语句
 - 更加通用（阶段划分！）
- 接下来，我们重点讨论第二种

使用控制流图的编译器结构





由三地址码生成控制流图算法

```
List_t stms; // 三地址码中所有语句
List_t blocks = {}; // 控制流图中的所有基本块
Block_t b = Block_fresh(); // 一个初始的空的基本块
scan_stms ()
    foreach(s ∈ stms)
        if (s is "Label L") // s是标号
            b.label = L;
        else (s is some jump) // s是跳转
            b.j = s;
            blocks ∪= {b};
            b = Block_fresh ();
        else // s是普通指令
            b.stms ∪= {s};
```

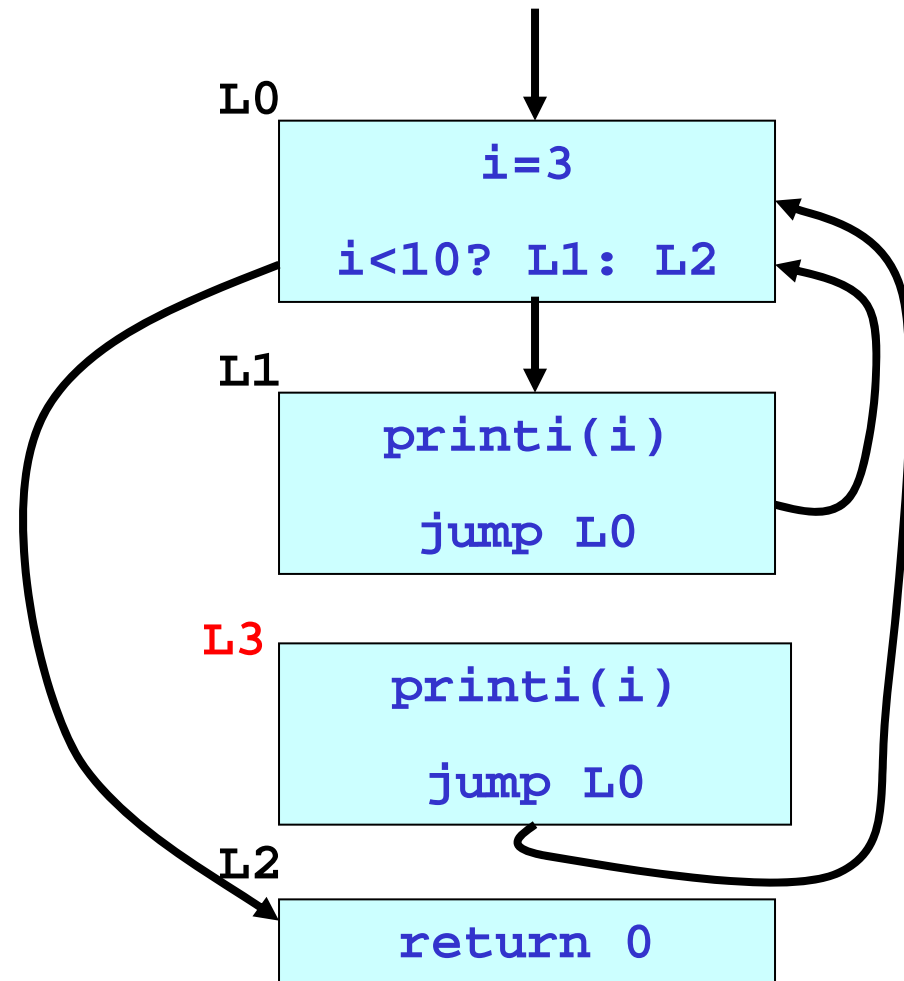


控制流图的基本操作

- 标准的图论算法都可以用在控制流图的操作上：
 - 各种遍历算法、生成树、必经节点结构、等等
- 图节点的顺序有重要的应用：
 - 拓扑序、逆拓扑序、近似拓扑序、等等
- 这里我们不打算重复算法课的内容，而是通过研究一个具体的例子来展示基本图算法的应用：
 - 死基本块删除优化

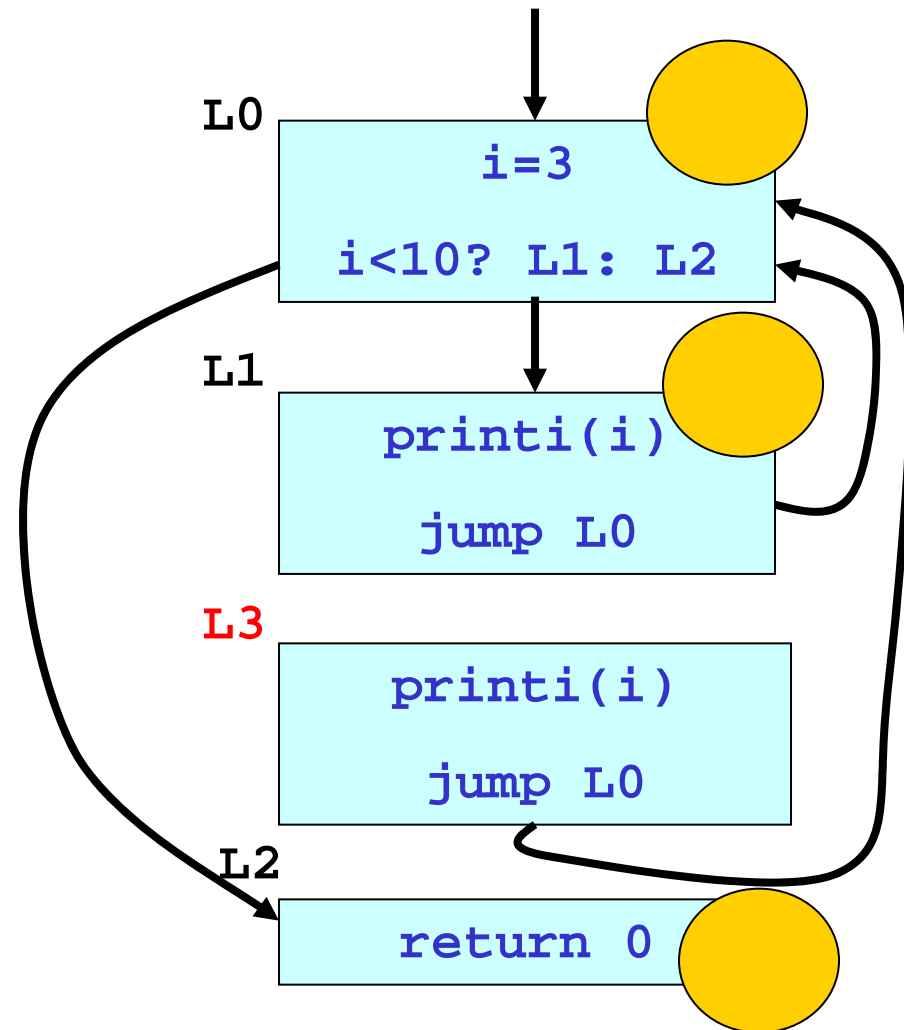
死基本块删除优化的示例

```
int f ()
{
    int i = 3;
    while (i<10){
        i = i+1;
        printi(i);
        continue;
        printi(i);
    }
    return 0;
}
```



死基本块删除的算法

```
// 输入：控制流图g
// 输出：经过死基本块删除
// 后的控制流图
dead_blocks_elim (g)
    dfs (g);
    for (each node n in g)
        if (!visited(n))
            delete (n);
```





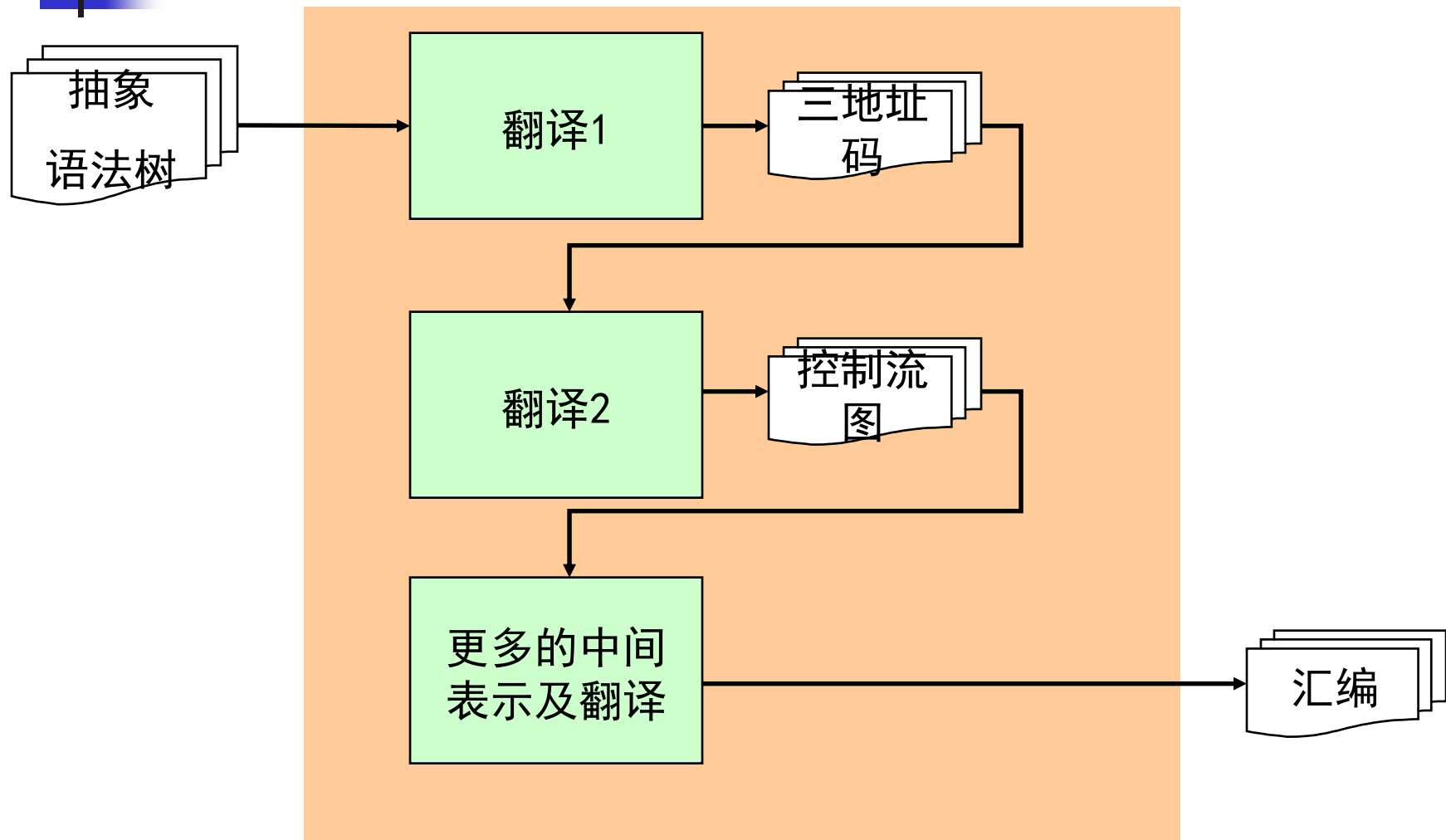
中间表示：数据流分析

编译原理

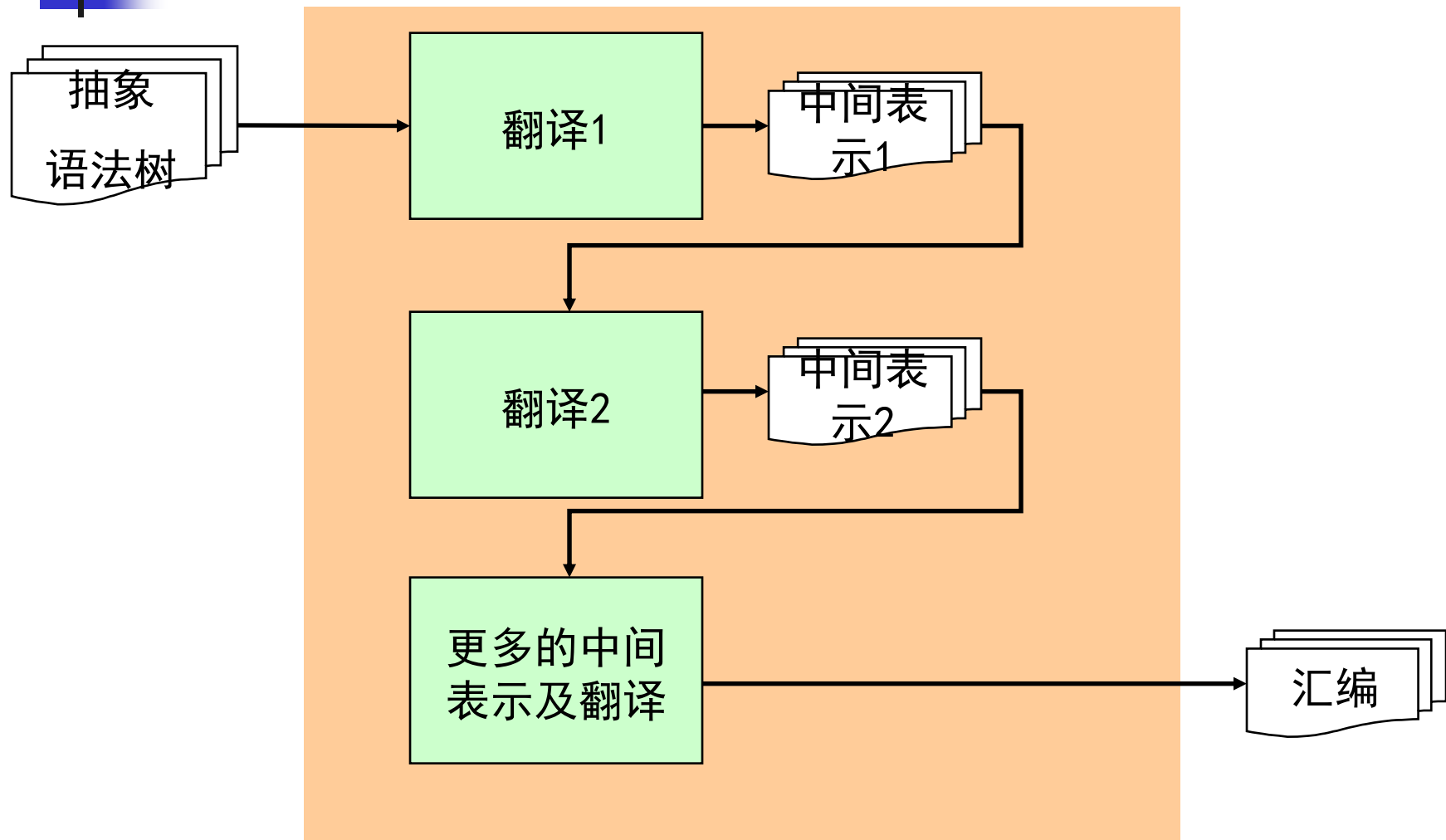
华保健

bjhua@ustc.edu.cn

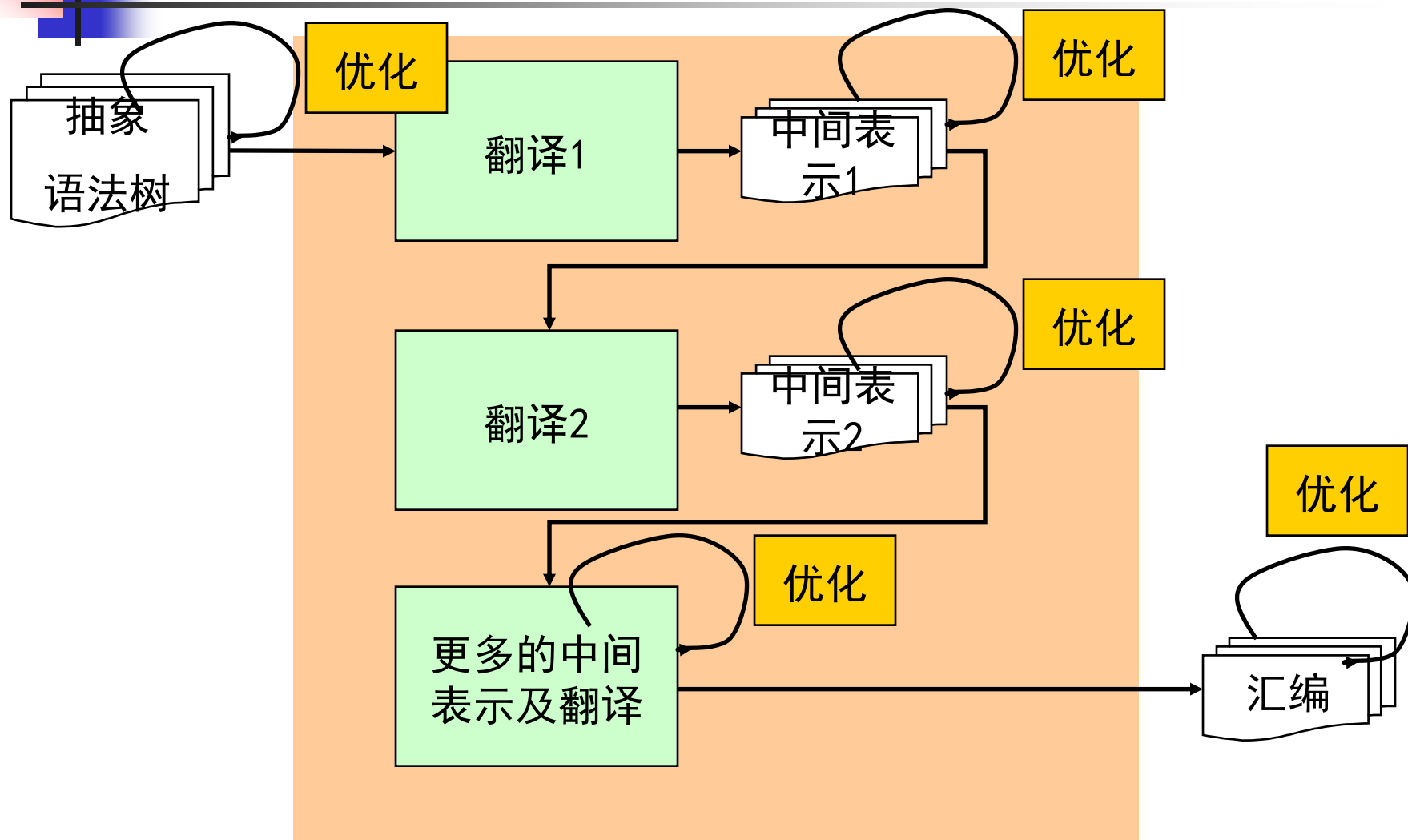
控制流图



一般结构



优化



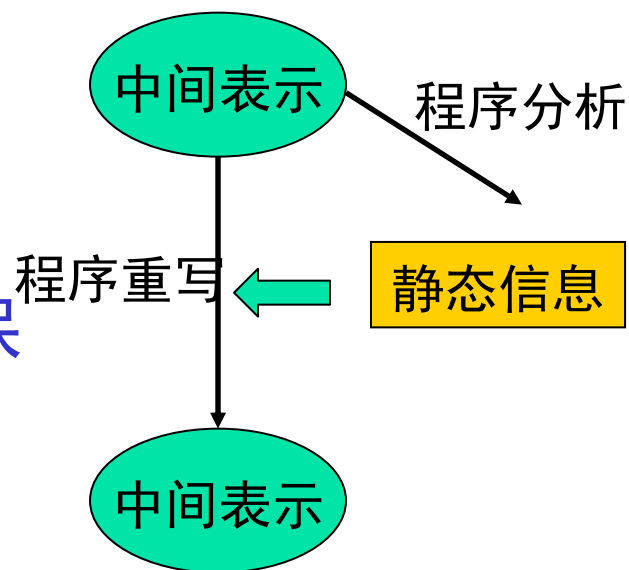
优化的一般模式

■ 程序分析

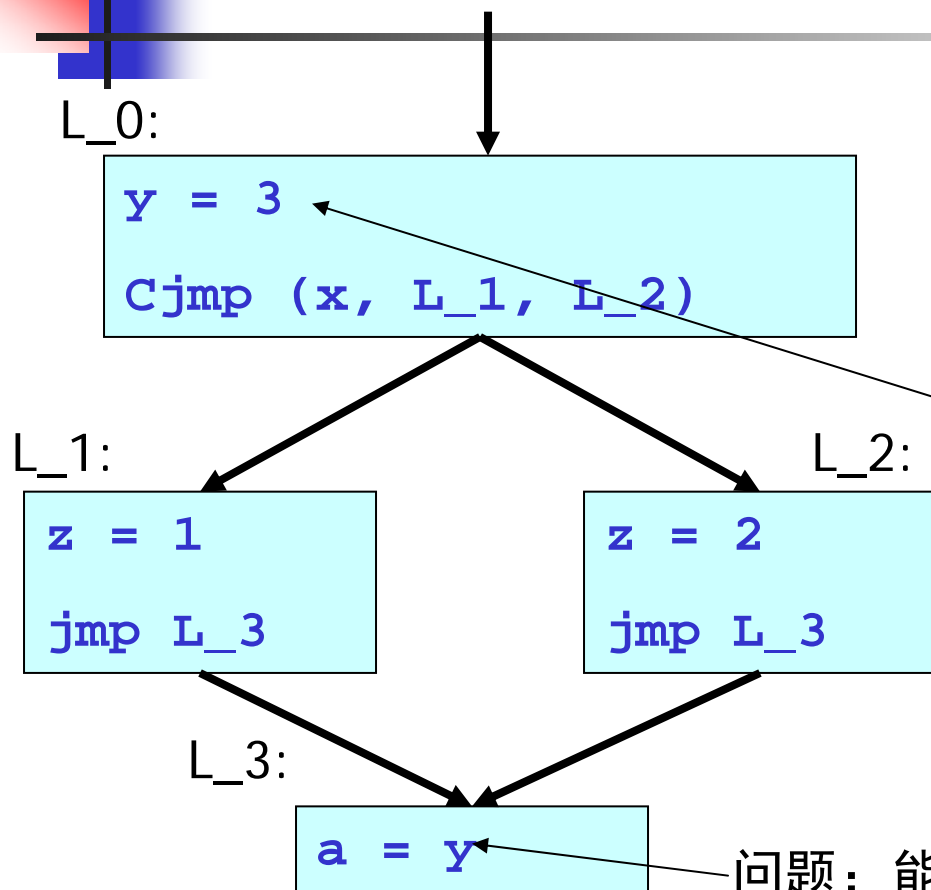
- 控制流分析，数据流分析，依赖分析，...
- 得到被优化程序的静态保守信息
 - 是对动态运行行为的近似

■ 程序重写

- 以上一步得到的信息制导对程序的重写



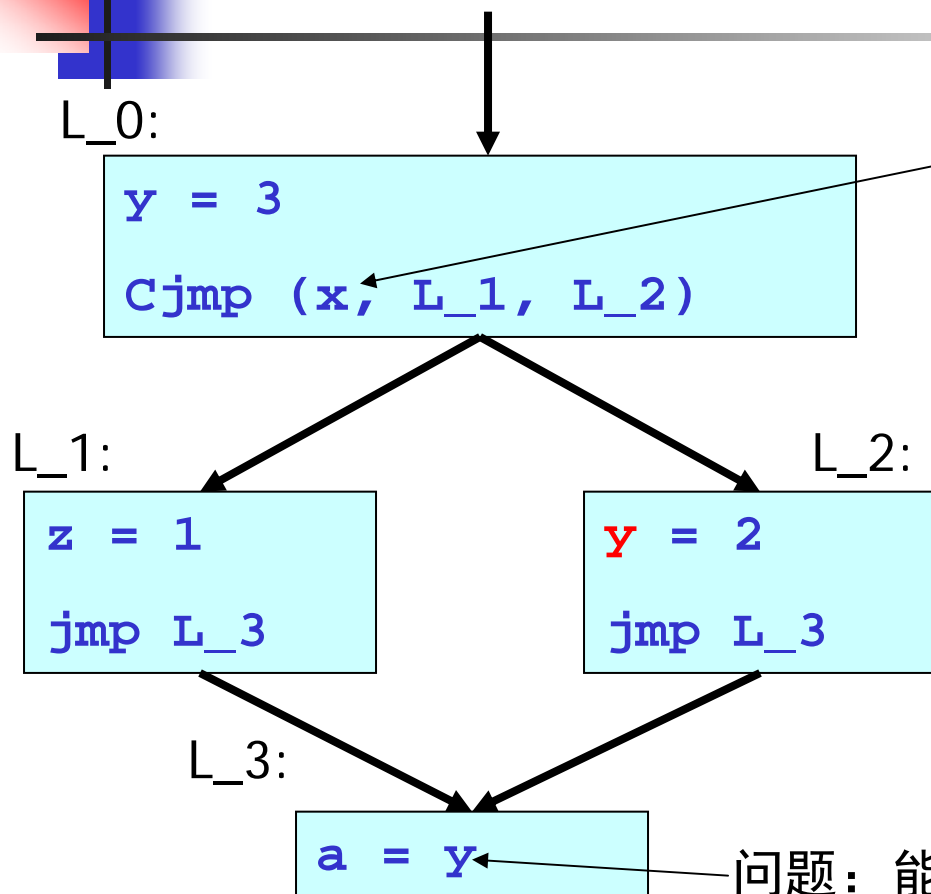
静态保守



对程序进行分析的结果表明：
只有L_0块中的y=3能到达L_3。
这种分析称为“**到达定义**”分析。

问题：能把y替换成3么？如果能，
这称为“**常量传播**”优化。

静态保守

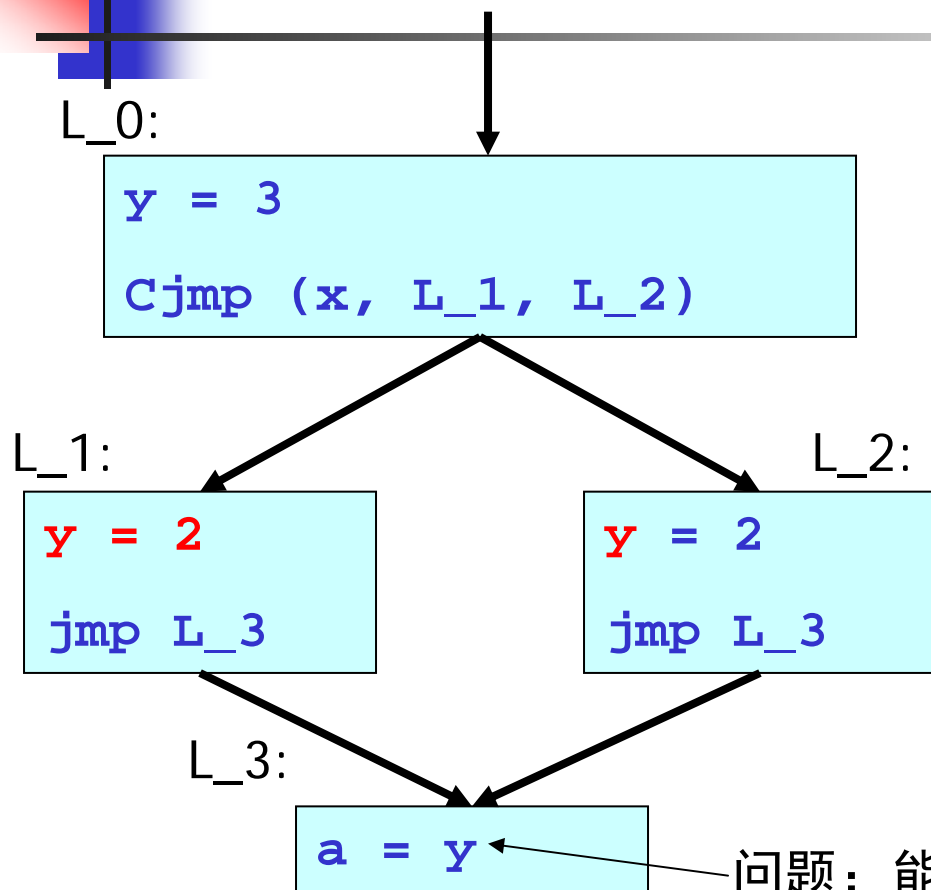


编译器需要证明: $x == 1$?

但若 x 是程序输入的话, 运行时才能知道值。所以编译器只能采用静态能够获得的信息对程序做保守估计: “L_2 可能会执行”。

问题: 能把 y 替换成 3 么? 如果能, 这称为 “**常量传播**” 优化。

静态保守



编译器能够证明可能有L_1或L_2中的对y的赋值能够到达L_3。这同样只依赖于程序的静态结构得到的保守信息，实际只会有一个块能到达。

问题：能把y替换成3么？如果能，这称为“**常量传播**”优化。



数据流分析

- 通过对程序代码进行静态分析，得到关于程序数据相关的保守信息
 - 必须保证程序分析的结果是安全的
- 根据优化的目标不同，需要进行的数据流分析也不同
 - 接下来我们将研究两种具体的数据流分析
 - 到达定义分析
 - 活性分析



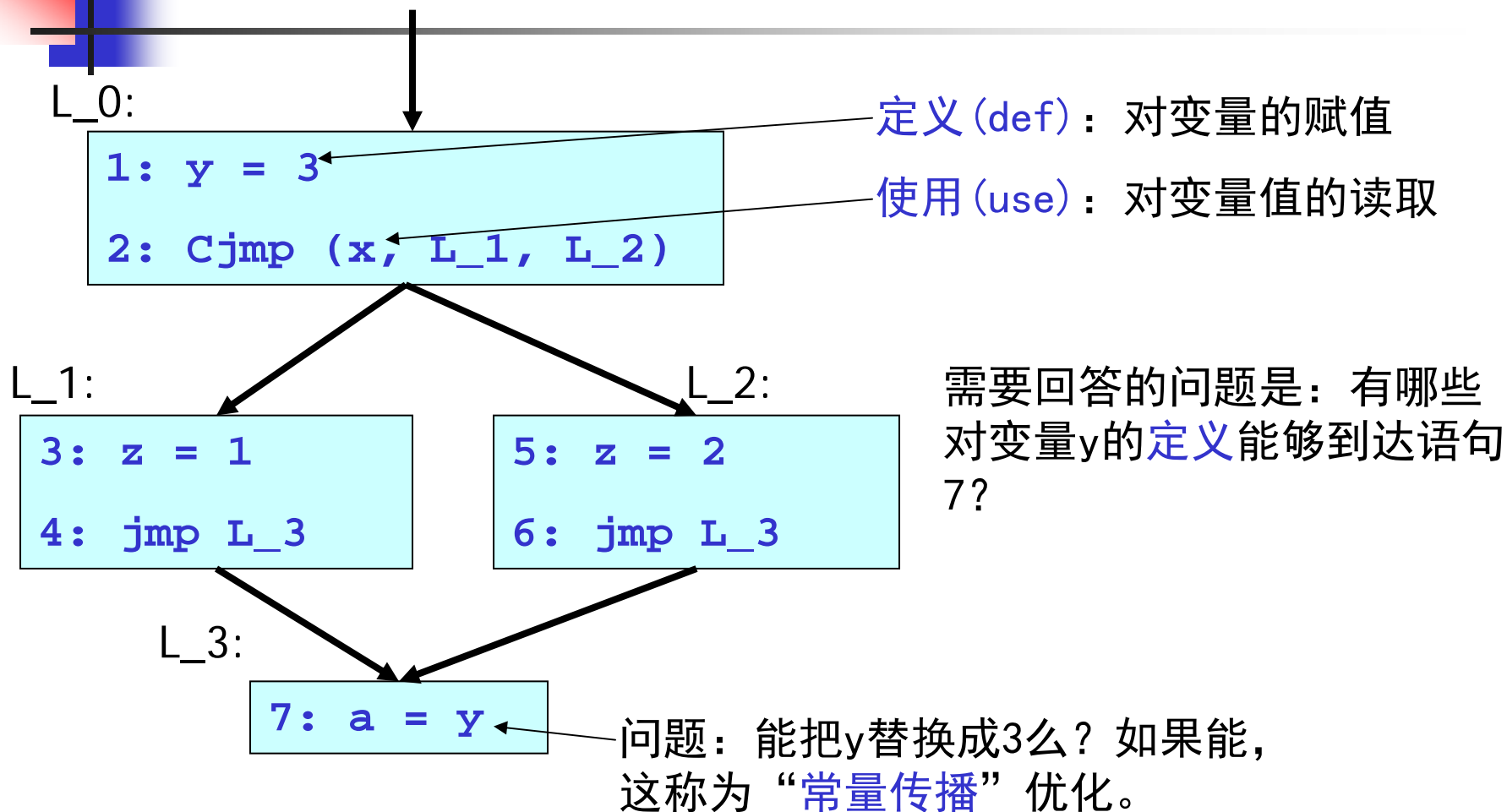
中间表示：到达定义分析

编译原理

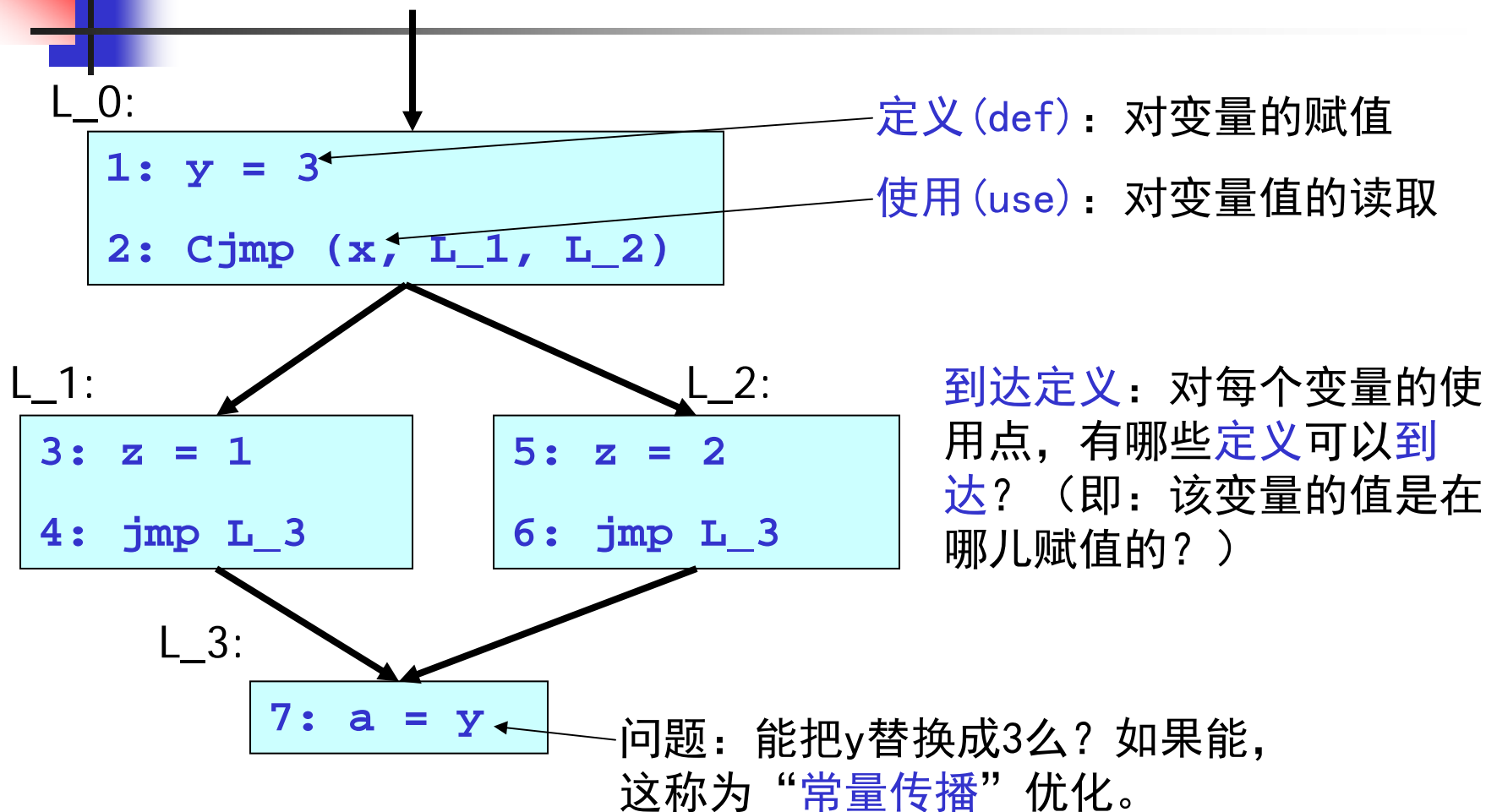
华保健

bjhua@ustc.edu.cn

定义、使用



到达定义





数据流方程

对任何一条定义：

`[d: x = ...]`

给出两个集合：

`gen[d: x = ...] = {d}`

`kill[d: x = ...] = defs[x] - {d}`

1: y = 3

2: z = 4

3: x = 5

4: y = 6

5: y = 7

6: z = 8

7: a = y



数据流方程

对任何一条定义：

$[d: x = \dots]$

给出两个集合：

$\text{gen}[d: x = \dots] = \{d\}$

$\text{kill}[d: x = \dots] = \text{defs}[x] - \{d\}$

数据流方程：

$\text{in}[s_i] = \text{out}[s_{i-1}]$

$\text{out}[s_i] = \text{gen}[s_i] \cup (\text{in}[s_i] - \text{kill}[s_i])$

1: $y = 3$

2: $z = 4$

3: $x = 5$

4: $y = 6$

5: $y = 7$

6: $z = 8$

7: $a = y$



从数据流方程到算法

// 算法：对一个基本块的到达定义算法

// 输入：基本块中所有的语句

// 输出：对每个语句计算in和out两个集合

List_t stms; // 一个基本块中的所有语句

set = {}; // 临时变量，记录当前语句s的in集合

reaching_definition ()

foreach (s ∈ stms)

in[s] = set;

out[s] = gen[s] ∪ (in[s]-kill[s])

set = out[s]



示例

	1	2	3	4	5	6	7
in	{}	{1}	{1,2}				
out	{1}	{1,2}	{1,2,3}				

$\text{in}[s_i] = \text{out}[s_{i-1}]$

$\text{out}[s_i] = \text{gen}[s_i] \cup (\text{in}[s_i] - \text{kill}[s_i])$

1: $y = 3$

2: $z = 4$

3: $x = 5$

4: $y = 6$

5: $y = 7$

6: $z = 8$

7: $a = y$

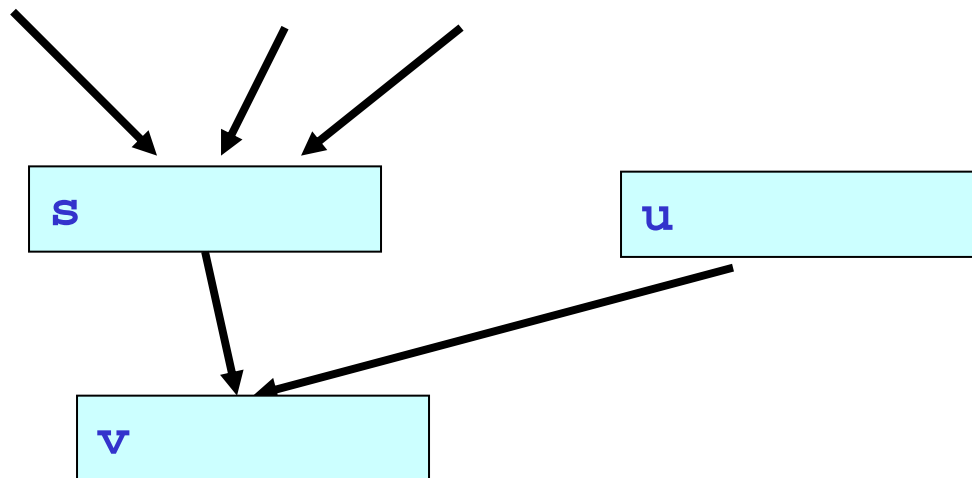
语句	1	2	3	4	5	6	7
gen	{1}	{2}	{3}	{4}	{5}	{6}	{7}
kill	{4,5}	{6}	{}	{1,5}	{1,4}	{2}	{}

对于一般的控制流图

- 前向数据流方程:

$$\text{in}[s] = \bigcup_{p \in \text{pred}(s)} \text{out}[p]$$

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$





从数据流方程到不动点算法

```
// 算法：对所有基本块的到达定义算法
// 输入：基本块中所有的语句
// 输出：对每个语句计算in和out两个集合
List_t stms;           // 所有基本块中的所有语句
set = {};              // 临时变量，记录当前语句s的in集合
reaching_definition ()
    while (some set in[] or out[] is still changing)
        foreach (s ∈ stms)
            foreach (predecessor p of s)
                set U= out[p];
            in[s] = set;
            out[s] = gen[s] U (in[s]-kill[s]);
```

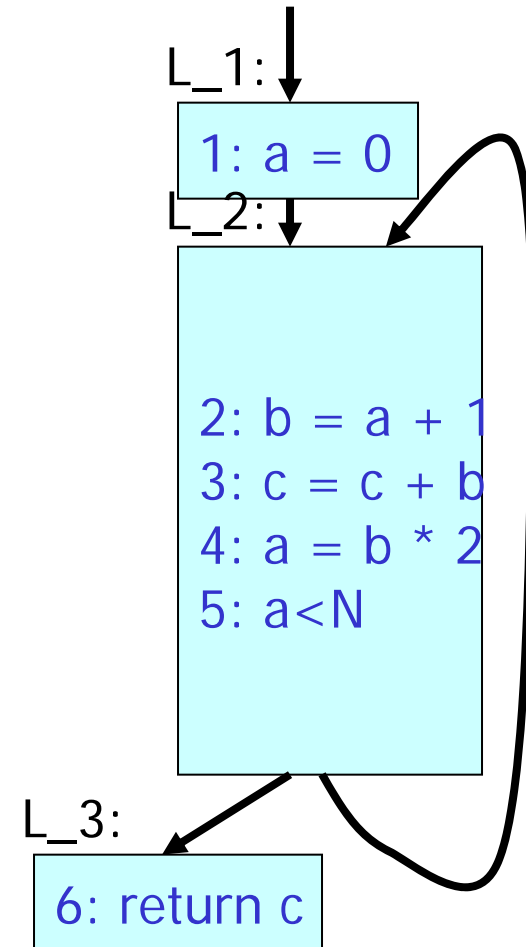
示例

$$\text{in}[s] = \bigcup_{p \in \text{pred}(s)} \text{out}[p]$$

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$

	in/out	in/out	in/out	in/out
1	{ } { }	{ } {1}		
2	{ } { }	{1} {1,2}		
3	{ } { }	{1,2} {1,2,3}		
4	{ } { }	{1,2,3} {2,3,4}		
5	{ } { }	{2,3,4} {2,3,4}		
6	{ } { }	{2,3,4} {2,3,4}		

语句	1	2	3	4	5	6	7
gen	{1}						
kill	{4}						





中间表示：活性分析

编译原理

华保健

bjhua@ustc.edu.cn



进行活性分析的动机

- 在代码生成的讨论中，我们曾假设目标机器有无限多个（虚拟）寄存器可用
 - 简化了代码生成的算法
 - 对物理机器是个坏消息
 - 机器只有有限多个寄存器
 - 必须把无限多个虚拟寄存器分配到有限个寄存器中
- 这是寄存器分配优化的任务
 - 需要进行活性分析



示例

考虑这段三地址码：

```
a = 1
```

```
b = a + 2
```

```
c = b + 3
```

```
return c
```

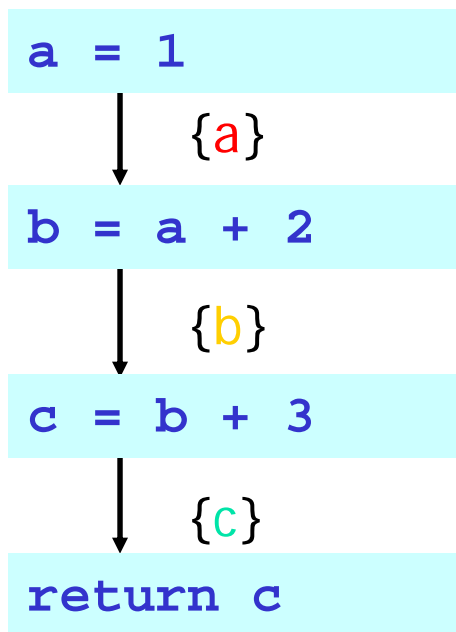
有三个变量 a , b , c .

假设目标机器上只有一个物理寄存器： r .

是否可能把三个变量 a , b , c 同时放到寄存器 r 中？

示例

考虑这段三地址码：



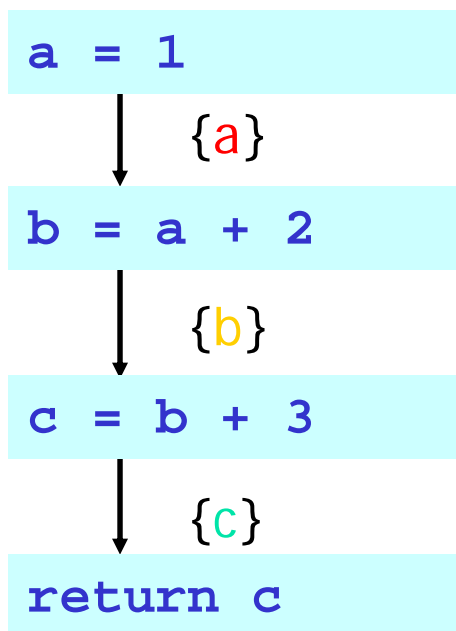
计算在给定的程序点，哪些变量是“活跃”的

活跃信息给出了活跃区间的概念.

活跃区间互不相交，所以三个变量可交替使用同一个寄存器。

示例

考虑这段三地址码：



寄存器分配：

`a` => `r`

`b` => `r`

`c` => `r`

代码重写：

`r = 1`

`r = r + 2`

`r = r + 3`

`return r`



数据流方程

对任何一条语句：

$[d: s]$

给出两个集合：

$gen[d: s] = \{x \mid \text{变量}x\text{在语句}s\text{中被使用}\}$

$kill[d: s] = \{x \mid \text{变量}x\text{在语句}s\text{中被定义}\}$

1: $x = y + z$

2: $z = z + x$

数据流方程

■ 基本块内的后向数据流方程：

$$\text{out}[s_i] = \text{in}[s_{i+1}]$$

$$\text{in}[s] = \text{gen}[s] \cup (\text{out}[s] - \text{kill}[s])$$

// 示例1:

a = 1

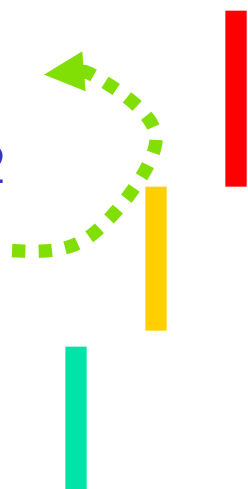
↓ int

b = a + 2

↓ out

c = b + 3

return c



// 示例2:

a = 1

b = a + 2

c = b + 3

return a + c



一般的数据流方程

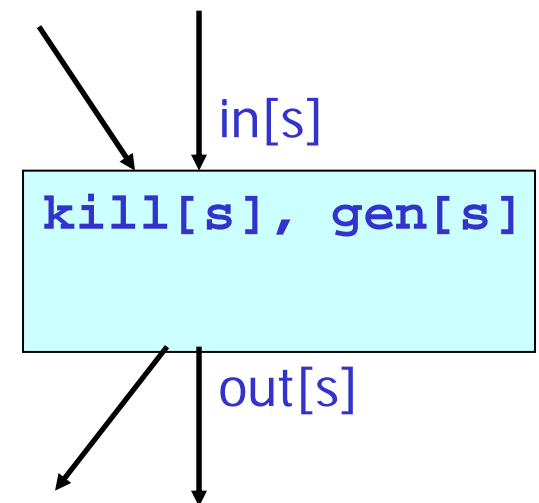
- 方程:

$$\text{out}[s] = \bigcup_{p \in \text{succ}[s]} \text{in}[p]$$

$$\text{in}[s] = \text{gen}[s] \cup (\text{out}[s] - \text{kill}[s])$$

- 同样可给出不动点算法

- 从初始的空集 $\{\}$ 出发
- 循环到没有集合变化为止



$$\begin{aligned} \text{out}[s] &= \bigcup_{p \in \text{succ}[s]} \text{in}[p] \\ \text{in}[s] &= \text{gen}[s] \cup (\text{out}[s] - \text{kill}[s]) \end{aligned}$$

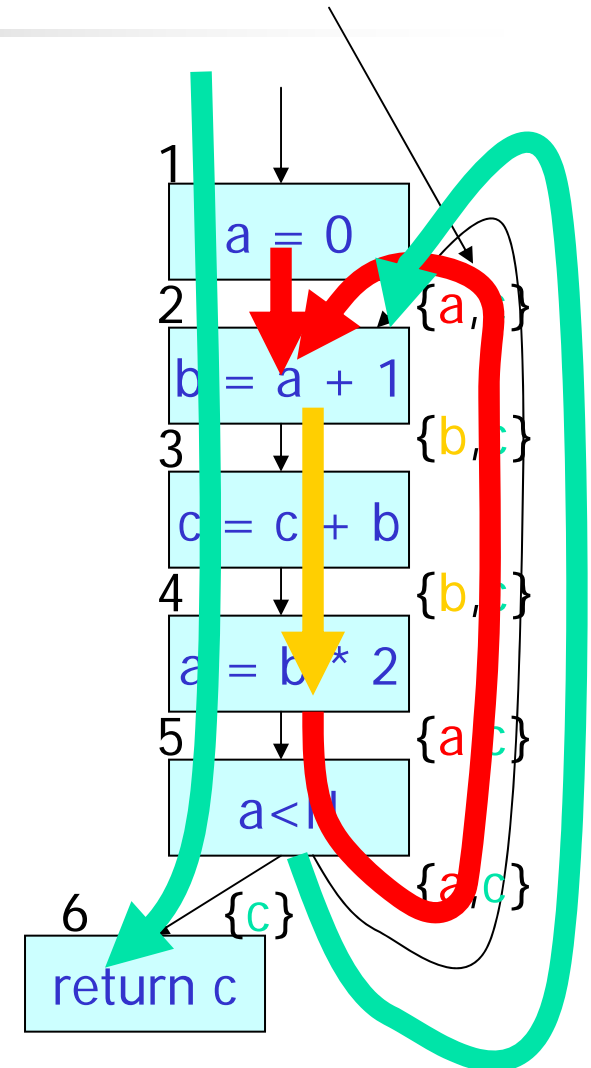
示例

	in/out	in/out	in/out	in/out	in/out
1	{ } { }	{ } { }	{ } {a}	...	
2	{ } { }	{a} { }	{a} {b,c}	...	
3	{ } { }	{b,c} { }	{b,c}{b}	...	
4	{ } { }	{b} { }	{b}{a,c}	...	
5	{ } { }	{a} {a}	{a}{a,c}	...	
6	{ } { }	{c} { }	{c} { }	...	

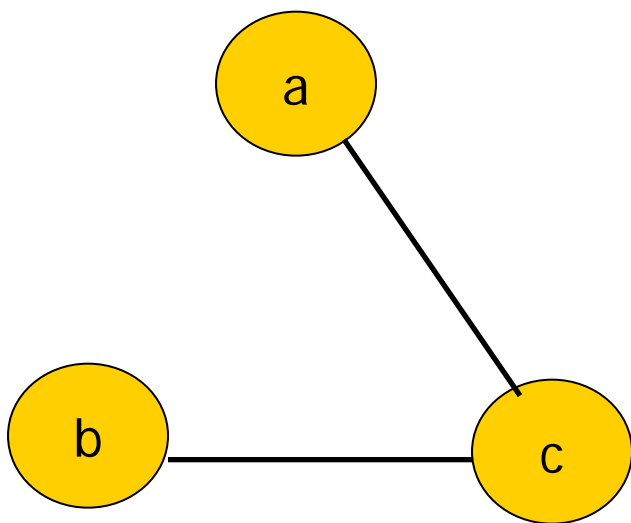
语句的遍历顺序: 1, 2, 3, 4, 5, 6

node	1	2	3	4	5	6
def	{a}	{b}	{c}	{a}	{ }	{ }
use	{ }	{a}	{b, c}	{b}	{a, N}	{c}

Final live_out



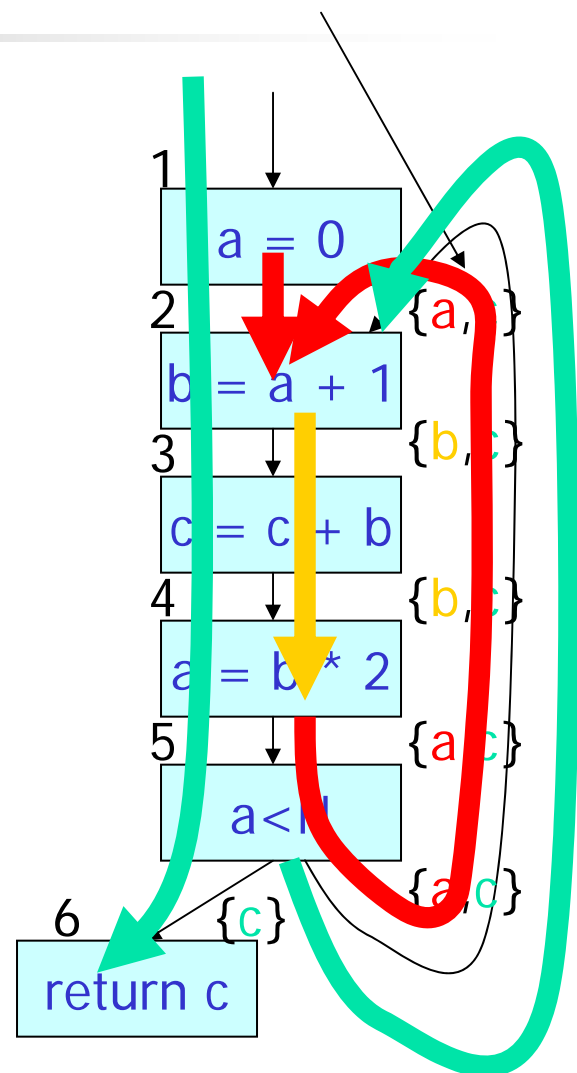
干扰图



干扰图是一个无向图 $G=(V, E)$ ：

1. 对每个变量构造无向图 G 中一个节点；
2. 若变量 x, y 同时活跃，在 x, y 间连一条无向边。

Final live_out





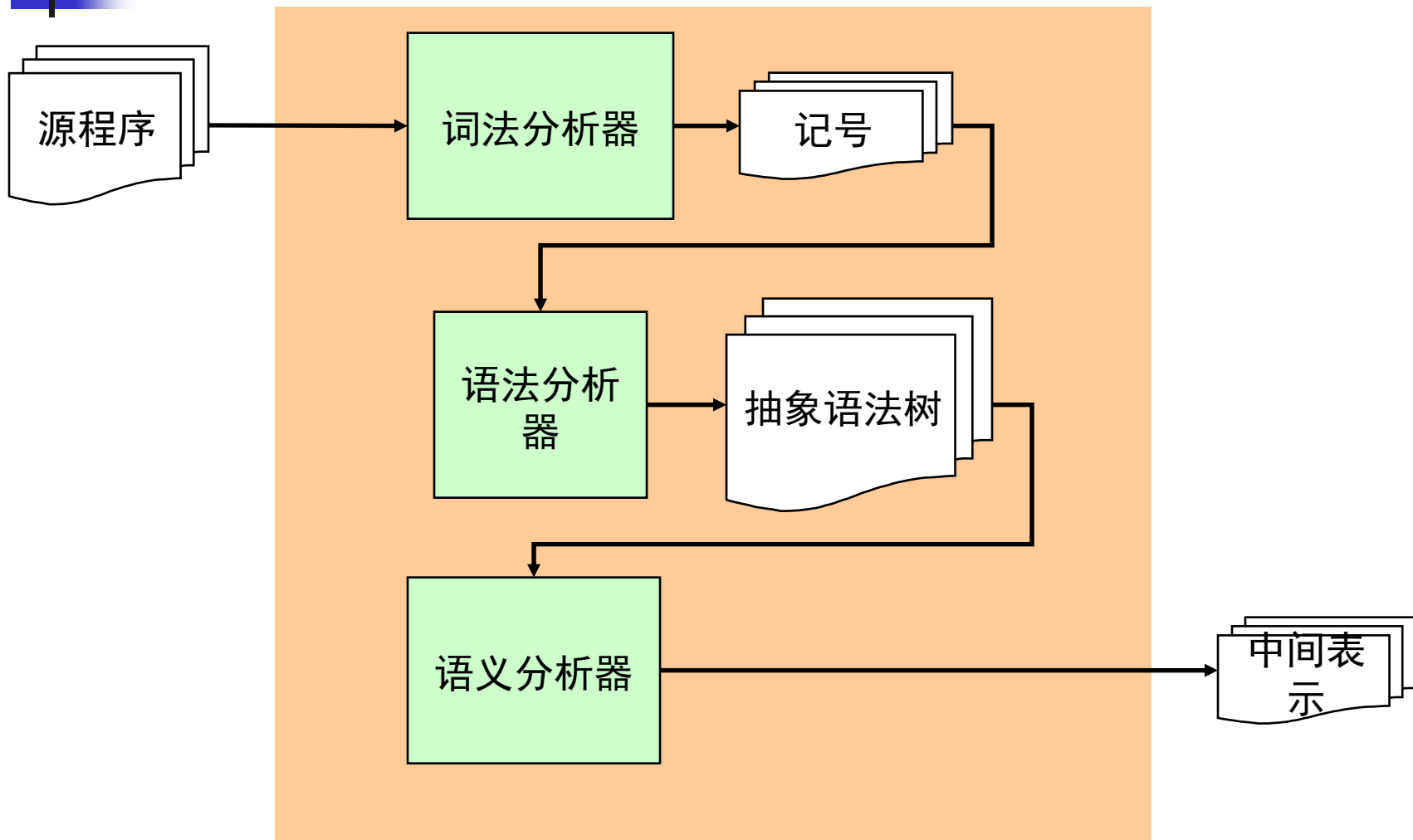
代码优化：基本概念

编译原理

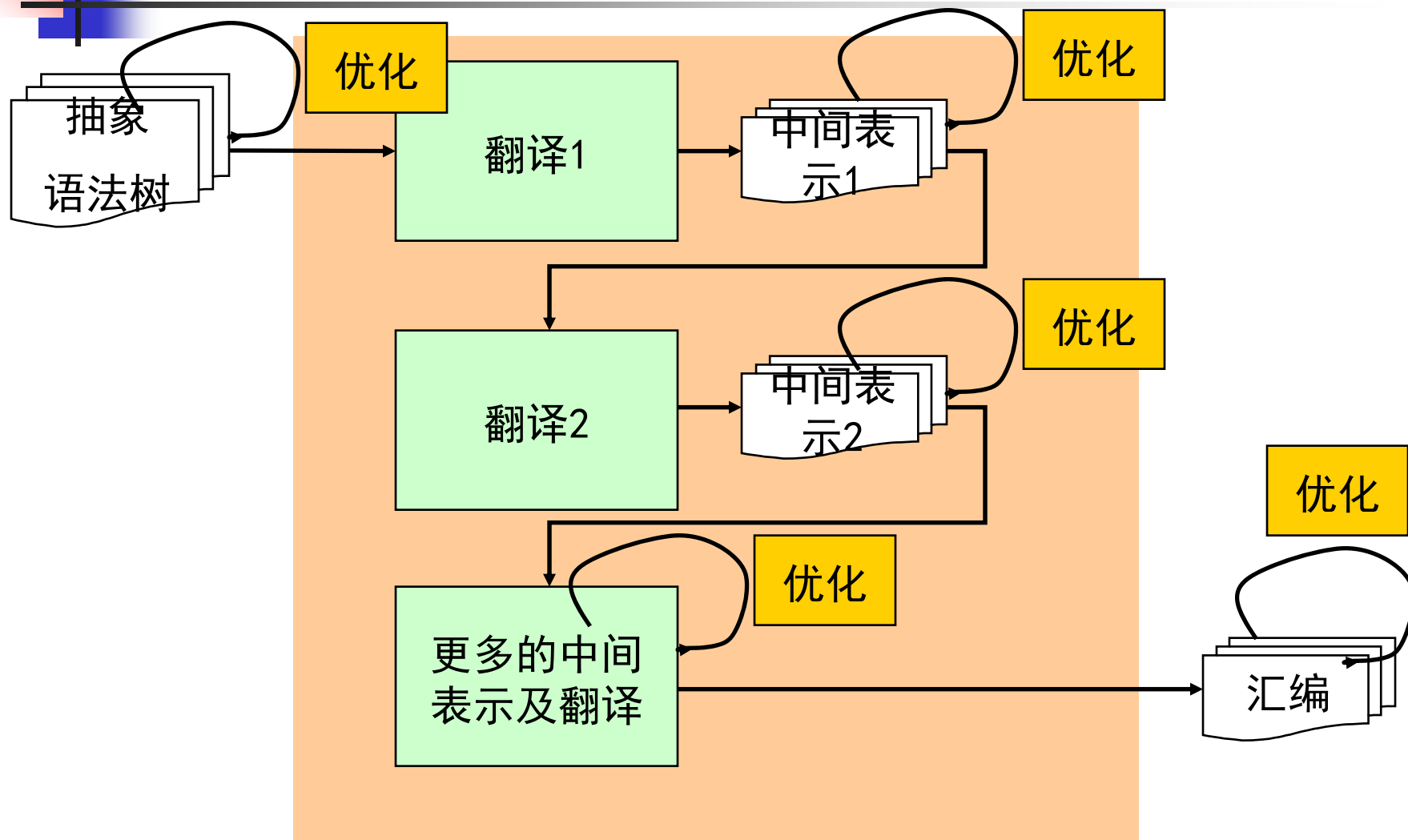
华保健

bjhua@ustc.edu.cn

前端



优化的位置





什么是“代码优化”？

- 代码优化是对被优化的程序进行的一种语义保持的变换
- 语义保持：
 - 程序的可观察行为不能改变
- 变换的目的是让程序能够比变换前：
 - 更小
 - 更快
 - cache行为更好
 - 更节能
 - 等等



不存在“完全优化”

- 等价于停机问题：
 - 给定程序 p ，把 $\text{Opt}(p)$ 和下面的程序比较：
L:

 jmp L
- 这称为“*编译器从业者永不失业定理*”



代码优化很困难

- 不能保证优化总能产生“好”的结果
- 优化的顺序和组合很关键
- 很多优化问题是非确定的
- 优化的正确性论证很微妙



正确的观点

- “把该做对的做对”
 - 不是任何程序都会同概率出现
 - 所以能处理大部分常见情况的优化就可以接受
- “不期待完美编译器”
 - 如果一个编译器有足够多的优化，则就是一个好的编译器



路线图

- 前端优化
 - 局部的、流不敏感的
 - 常量折叠、代数优化、死代码删除等
- 中期优化
 - 全局的、流敏感的
 - 常量传播、拷贝传播、死代码删除、公共子表达式删除等
- 后端优化
 - 在后端（汇编代码级）进行
 - 寄存器分配、指令调度、窥孔优化等



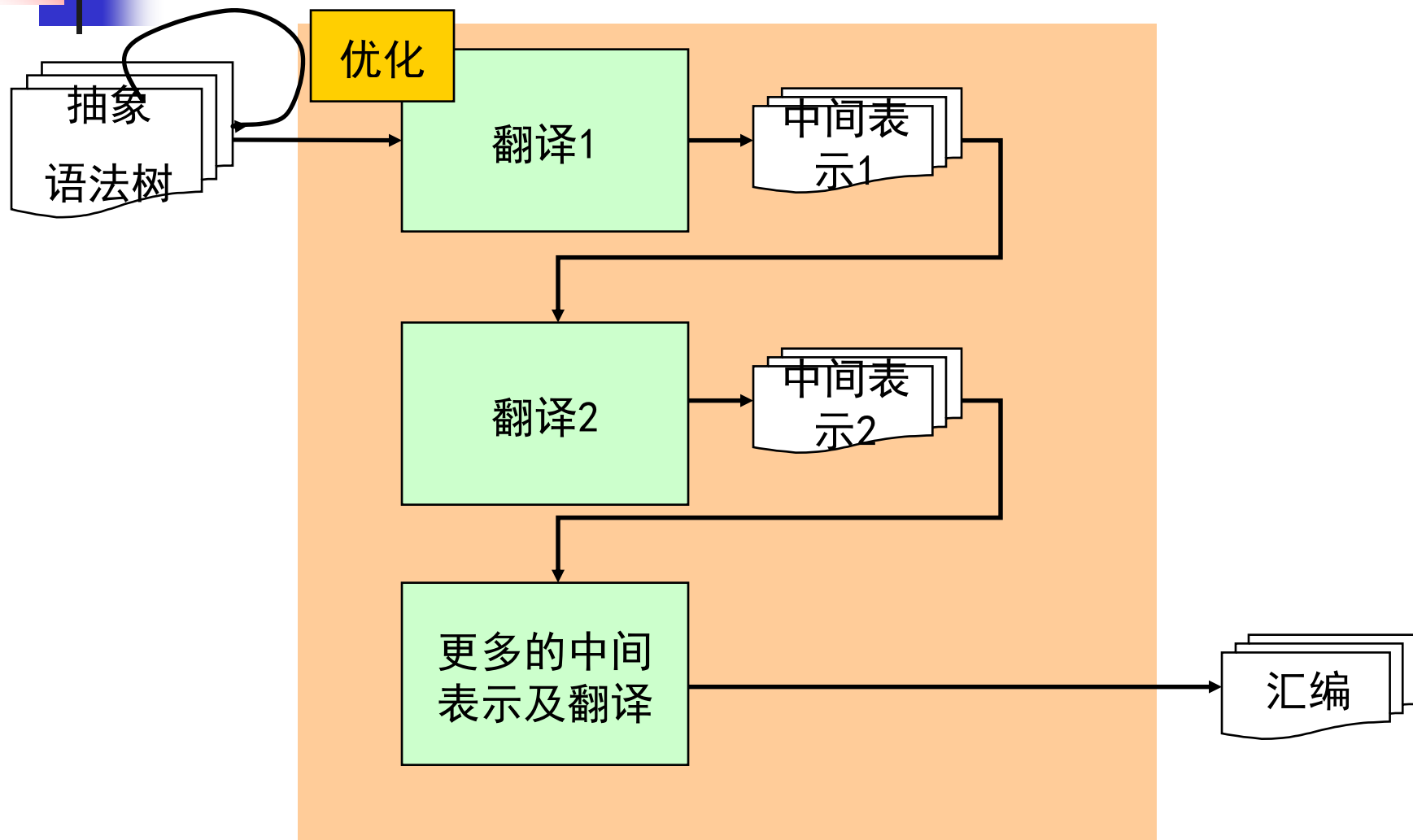
代码优化：前端优化

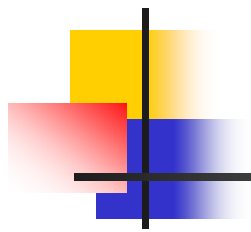
编译原理

华保健

bjhua@ustc.edu.cn

前端优化的地位





常量折叠



常量折叠

- 基本思想：
 - 在编译期计算表达式的值
 - 例如： $a = 3 + 5 \implies a = 8$
 - 例如： `if (true && false) ...` \implies `if (false)`
- 可以在整型、布尔型、浮点型等数据类型上进行



算法

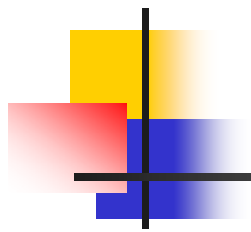
// 语法制导的常量折叠算法

```
const_fold(Exp_t e)
    while (e is still shrinking)
        switch (e->kind)
            case EXP_ADD:
                Exp_t l = e->left;
                Exp_t r = e->right;
                if (l->kind==EXP_NUM && r->kind==EXP_NUM)
                    e = Exp_Num_new (l->value + r->value);
                break;
            default:
                break;
```



小结

- 容易实现、可以在语法树或者中间表示上进行
- 通常被实现成公共子函数被其它优化调用
- 必须要很小心遵守语言的语义
 - 例如：考虑溢出或异常
 - 例子： $0xffffffff + 1 ==> 0$ (???)



代数化简



代数化简

- 基本思想:

- 利用代数系统的性质对程序进行化简

- 示例:

- $a = 0 + b \implies a = b$

- $a = 1 * b \implies a = b$

- $2 * a \implies a + a$ (强度消弱)

- $2 * a \implies a < < 1$ (强度消弱)

- 同样必须非常仔细的处理语义

- 示例: $(i - j) + (i - j) \implies i + i - j - j$



算法

// 代数化简的算法

```
alg_simp(Exp_t e)
```

```
    while (e is still shrinking)
```

```
        switch (e->kind)
```

```
            case EXP_ADD:
```

```
                Exp_t l = e->left;
```

```
                Exp_t r = e->right;
```

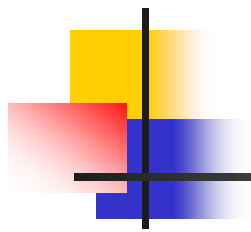
```
                if (l->kind==EXP_NUM && l->value==0)
```

```
                    e = r;
```

```
                break;
```

```
            case ...: ...;
```

// 类似



死代码删除



死代码（不可达代码）删除

- 基本思想：
 - 静态移除程序中不可执行的代码
 - 示例：
 - `if (false)`
 `s1;`
 `else s2;` \impl `s2;`
- 在控制流图上也可以进行这些优化，但在早期做这些优化可以简化中后端



算法

// 不可达代码删除算法

```
deadcode(Stm_t s)
```

```
    while(s is still shrinking)
```

```
        switch (s->kind)
```

```
            case STM_IF:
```

```
                Exp_t e = s->condition;
```

```
                if (e->kind==EXP_FALSE)
```

```
                    s = s->elsee;
```

```
                break;
```

```
            case ...: ...;
```

// 类似



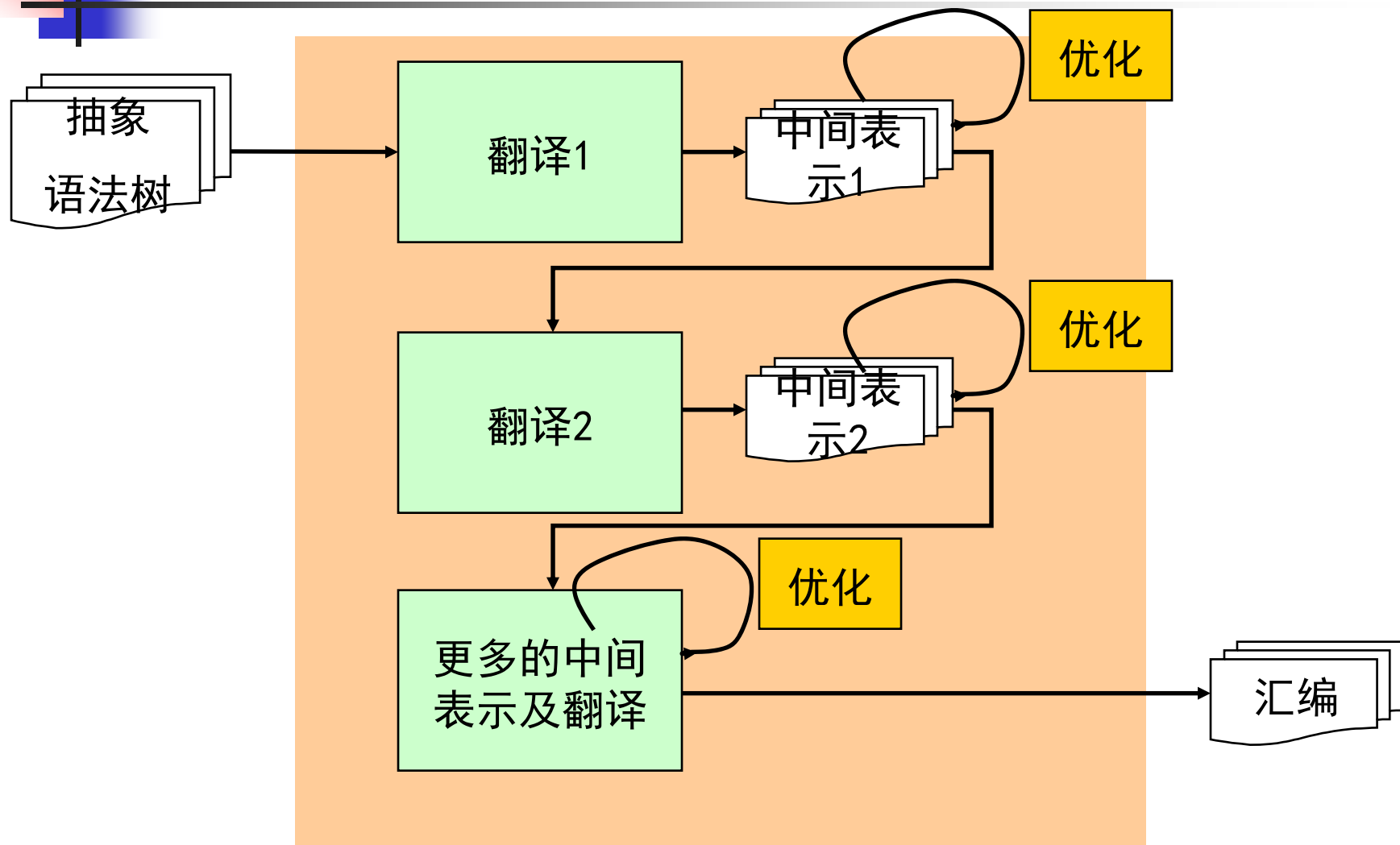
代码优化：中间表示上的优化

编译原理

华保健

bjhua@ustc.edu.cn

中间表示上优化的地位





中间表示上的代码优化

- 依赖于具体所使用的中间表示：
 - 控制流图（CFG）、控制依赖图（CDG）、静态单赋值形式（SSA）、后续传递风格（CPS）等
- 共同的特点是需要进行程序分析
 - 优化是全局进行的，而不是局部
 - 通用的模式是：程序分析→程序重写
- 在这部分中，我们将基于控制流图进行讨论
 - 但类似的技术可以用于其它类型的中间表示

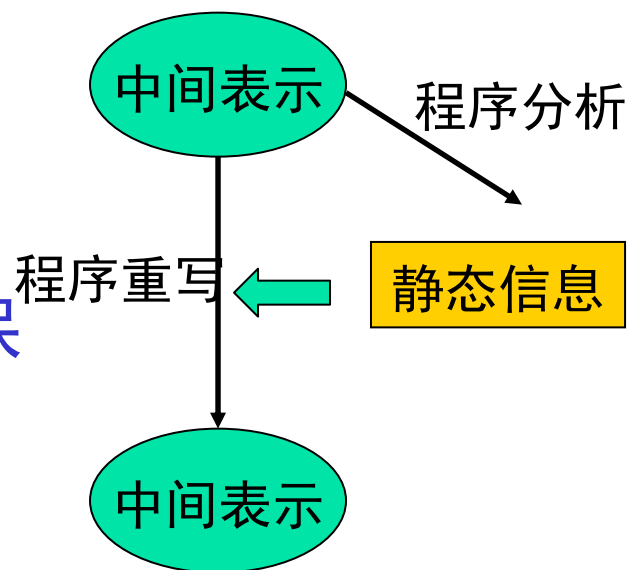
优化的一般模式

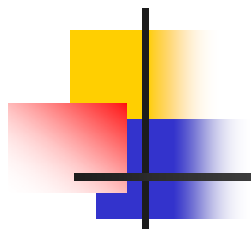
■ 程序分析

- 控制流分析，数据流分析，依赖分析，...
- 得到被优化程序的静态保守信息
 - 是对动态运行行为的近似

■ 程序重写

- 以上一步得到的信息制导对程序的重写

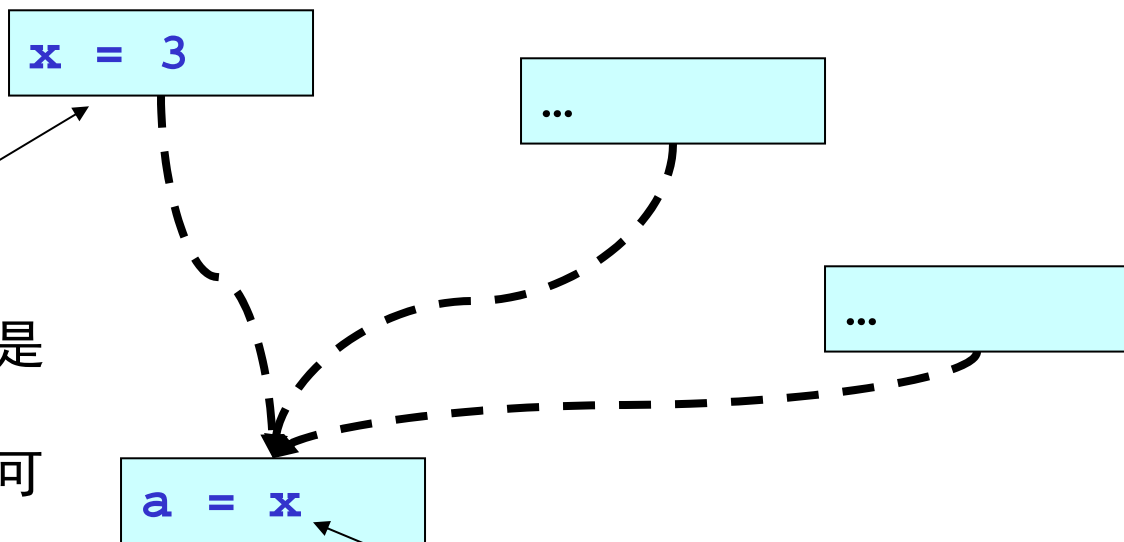




常量传播

常量传播

先进行到达定义分析，
如果这个定义“ $x=3$ ”是
唯一能够到达使用
“ $a=x$ ”的定义，那么可
以进行这个替换！



是否可以把 x 的使用替换
成 x 的定义3？



算法

// 常量传播算法

`const_prop(Prog_t p)`

// 第一步：程序分析

`reaching_definition(p);`

// 第二步：程序改写

`foreach (stm s in p: y = x1, ..., xn)`

`foreach (use of xi in s)`

`if(the reaching def of xi is unique: xi = n)`

`y = x1, ..., xi-1, n, xi+1, ..., xn`



讨论

// 对示例程序

```
x = 1;  
y = 2;  
z = x + y;  
a = z + 5;  
print (a);
```

// 第一步：常量传播优化

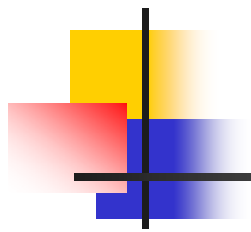
```
x = 1;  
y = 2;  
z = 1 + 2;  
a = z + 5;  
print (a);
```

// 第二步：常量折叠优化

```
x = 1;  
y = 2;  
z = 3;  
a = z + 5;  
print (a);
```

// 第三步：常量传播优化

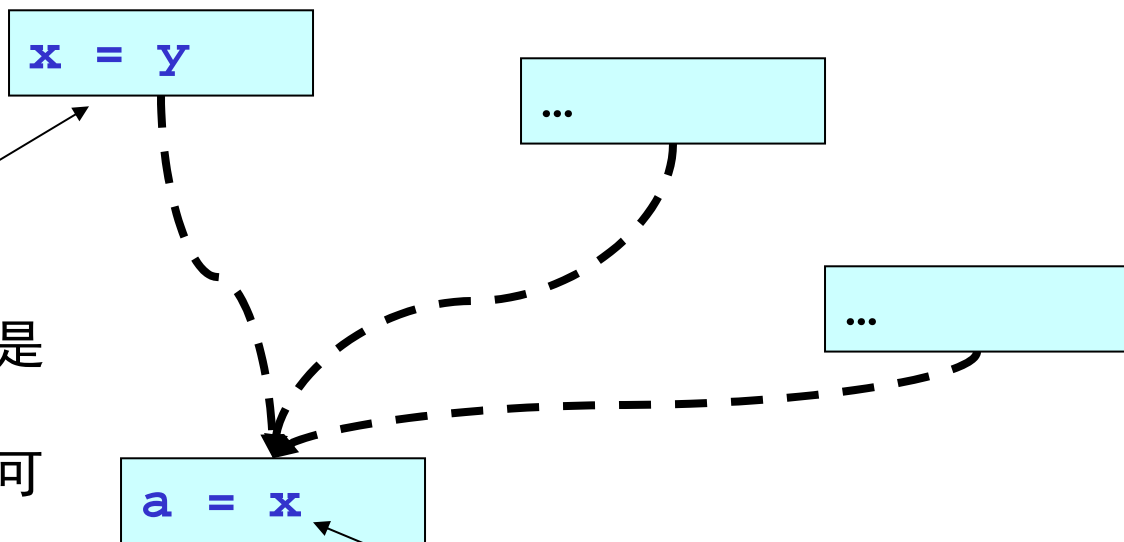
```
x = 1;  
y = 2;  
z = 3;  
a = 3 + 5;  
print (a);
```



拷贝传播

拷贝传播

先进行到达定义分析，
如果这个定义“ $x=y$ ”是
唯一能够到达使用
“ $a=x$ ”的定义，那么可
以进行这个替换！



是否可以把 x 的使用替换
成 x 的定义 y ？



算法

// 拷贝传播算法

`copy_prop(Prog_t p)`

// 第一步：程序分析

`reaching_definition(p);`

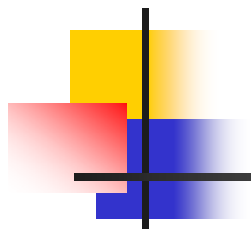
// 第二步：程序改写

`foreach (stm s in p: y = x1, ..., xn)`

`foreach (use of xi in s)`

`if(the reaching def of xi is unique: xi = z)`

`y = x1, ..., xi-1, z, xi+1, ..., xn`

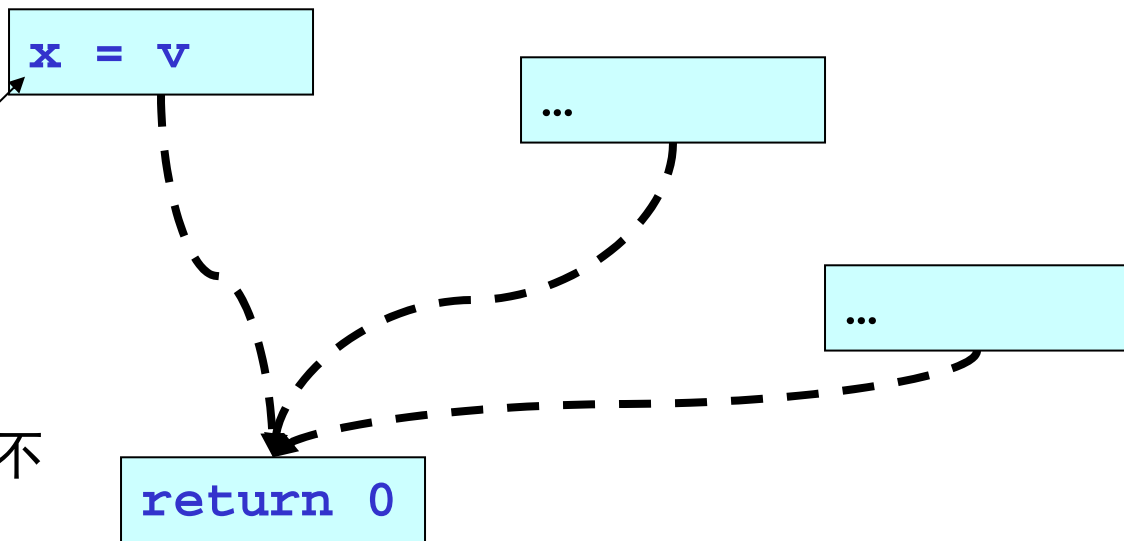


死代码删除

死代码删除

能把这个语句移除么？

进行活性分析，如果x不是该语句的live_out，则可以将该语句移除。





算法

// 死代码删除算法

`dead_code(Prog_t p)`

// 第一步：程序分析

`liveness_analysis(p);`

// 第二步：程序改写

`foreach (stm s in p: y = ...)`

`if (y is NOT in live_out[s])`

`remove (s);`



讨论

// 对示例程序

```
x = 1;  
y = 2;  
z = x + y;  
a = z + 5;  
print (a);
```

// 第一步：常量传播优化

```
x = 1;  
y = 2;  
z = 1 + 2;  
a = z + 5;  
print (a);
```

// 第二步：常量折叠优化

```
x = 1;  
y = 2;  
z = 3;  
a = z + 5;  
print (a);
```

// 第三步：常量传播优化

```
x = 1;  
y = 2;  
z = 3;  
a = 3 + 5;  
print (a);
```