



# 抽象语法树

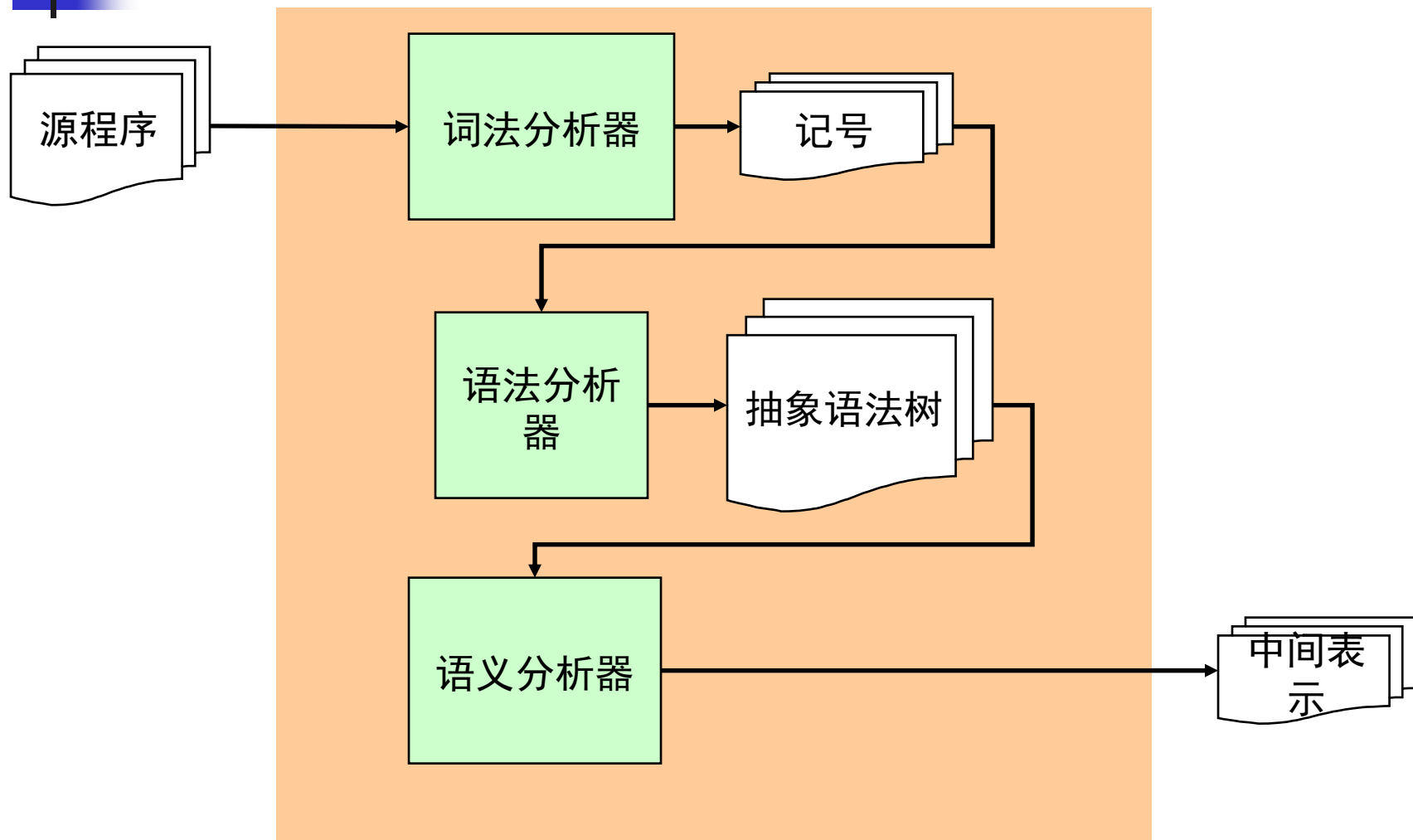
---

编译原理

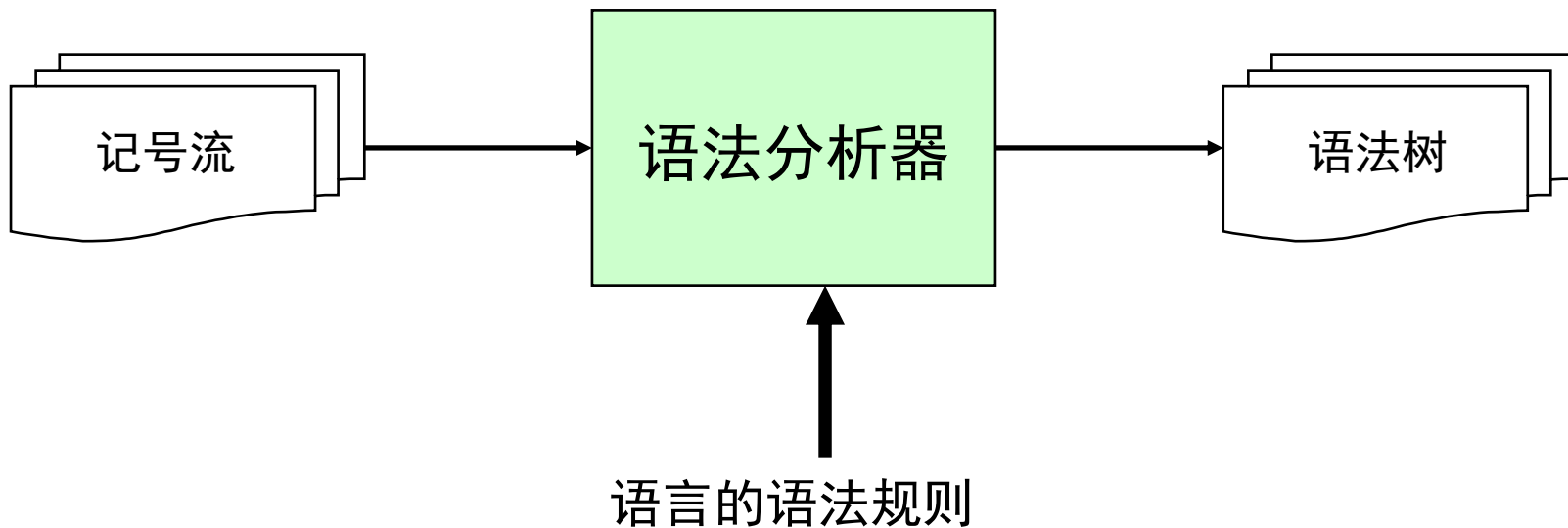
华保健

[bjhua@ustc.edu.cn](mailto:bjhua@ustc.edu.cn)

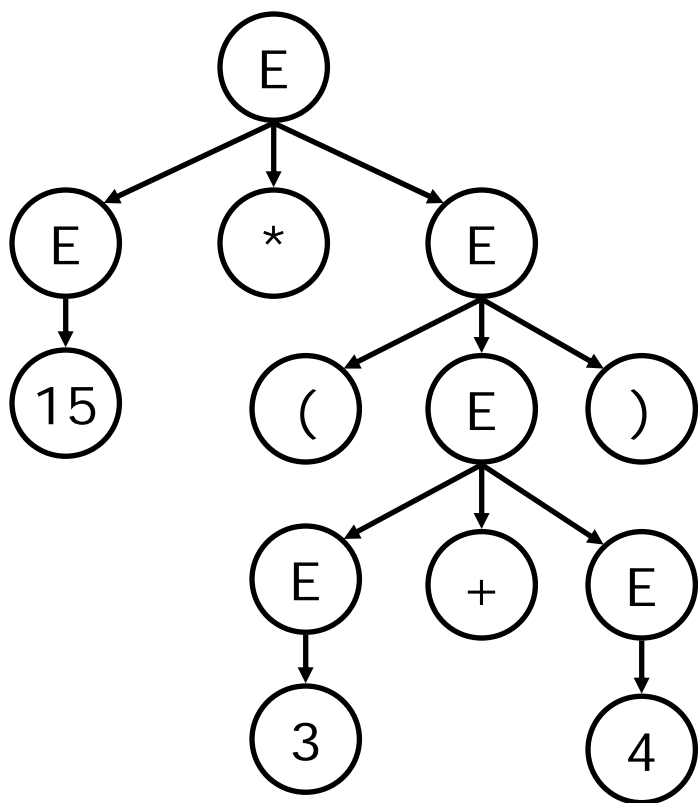
# 前端



# 语法分析器的任务



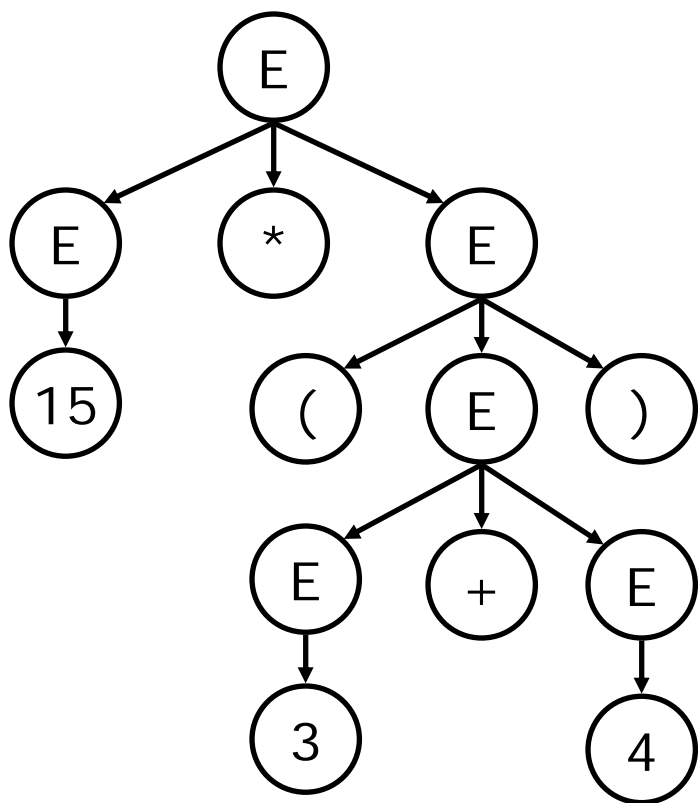
# 分析树



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？

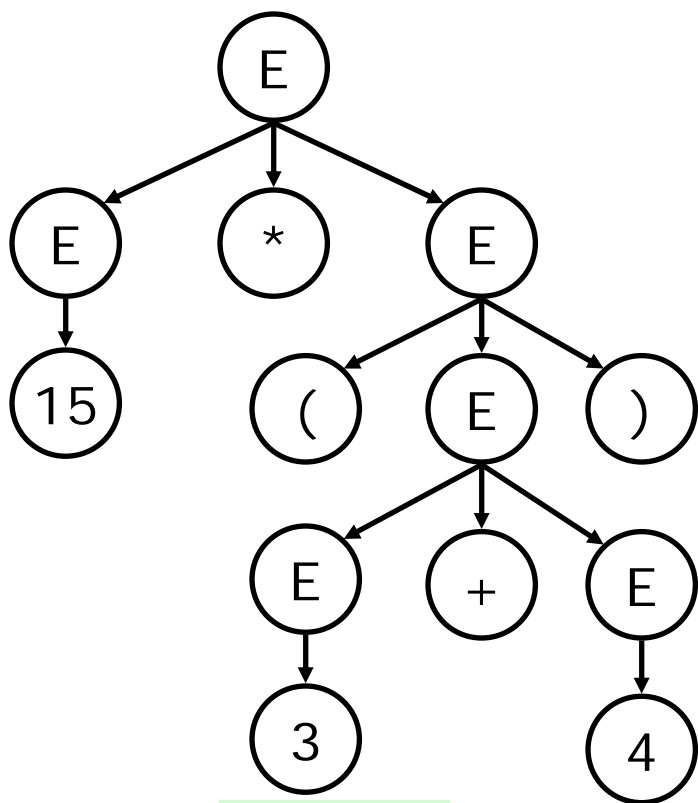
# 分析树



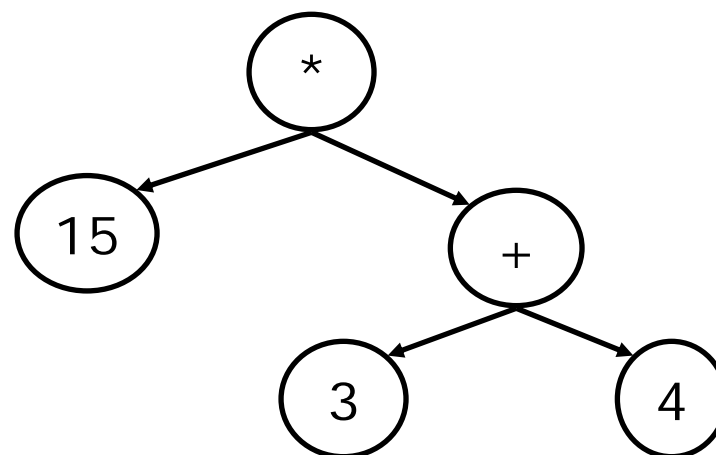
15 \* ( 3+4 )

- 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

$E \rightarrow E + T$

|  $T$

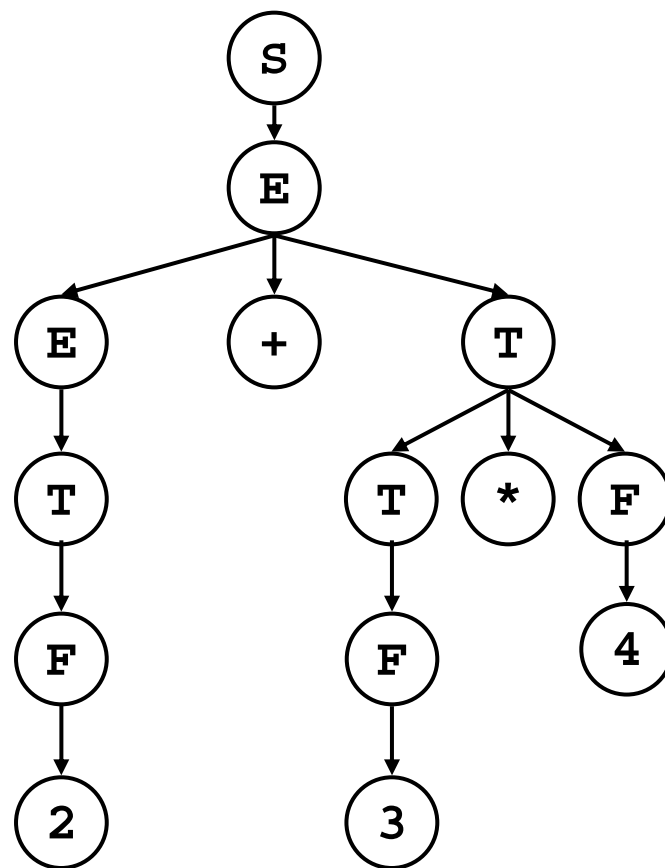
$T \rightarrow T * F$

|  $F$

$F \rightarrow n$

|  $(E)$

2 + 3 \* 4





# 具体语法和抽象语法

2 + 3 \* 4

$E \rightarrow E + T$

|  $T$

$T \rightarrow T * F$

|  $F$

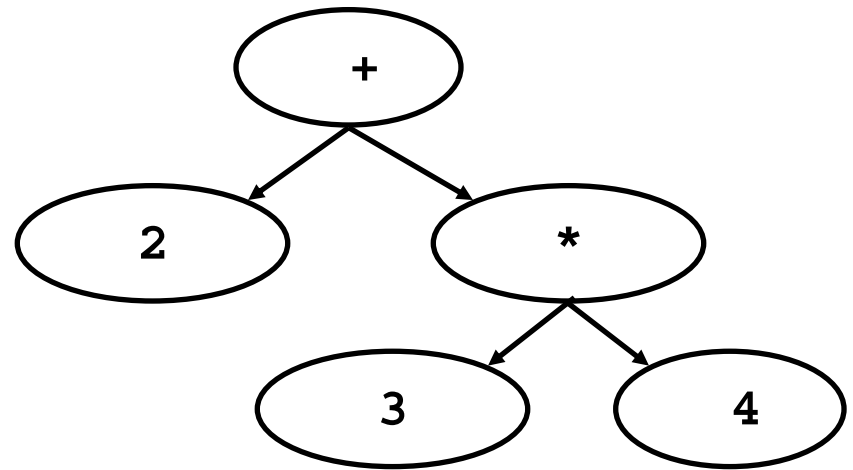
$F \rightarrow n$

|  $(E)$

$E \rightarrow n$

|  $E + E$

|  $E * E$

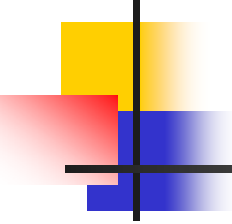




# 抽象语法树数据结构

---

- 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");  
        }  
    }  
}
```