软件分析

程序综合：概率

熊英飞
北京大学

目录

- 程序估计问题
  - 和加速的关联
- 扩展枚举的方法解决程序估计问题
- 定义不同的展开方向
- 扩展FlashMeta解决程序估计问题

很多应用需要概率最大的程序

典型应用–自动编写重复程序


```python
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
                CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
                battlecry=Battlecry(Destroy(),
                        WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```

典型应用–缺陷修复

```java
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```

综合出新的表达式来替换掉旧的

10

程序估计Program Estimation

- 输入:
  - 一个程序空间*Prog*
  - 一条规约*Spec*
  - 概率模型P，用于计算程序的概率
- 输出:
  - 一个程序*prog*，满足
    - $prog = \operatorname{argmax}_{prog \in Prog \wedge prog \vdash spec} P(prog)$

- 如果P估计程序满足规约的概率，那么可以用来加速传统程序综合

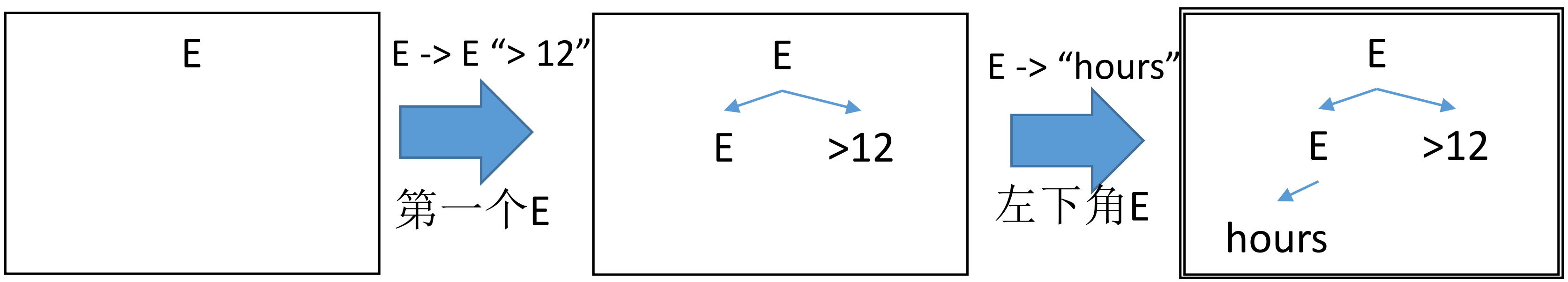
基本算法：穷举

- 用枚举的方法遍历空间中的程序
- 对每个程序计算概率
- 返回概率最大的程序

- 能否优化这个过程?

扩展枚举算法求解程序估计问题

规则展开概率模型



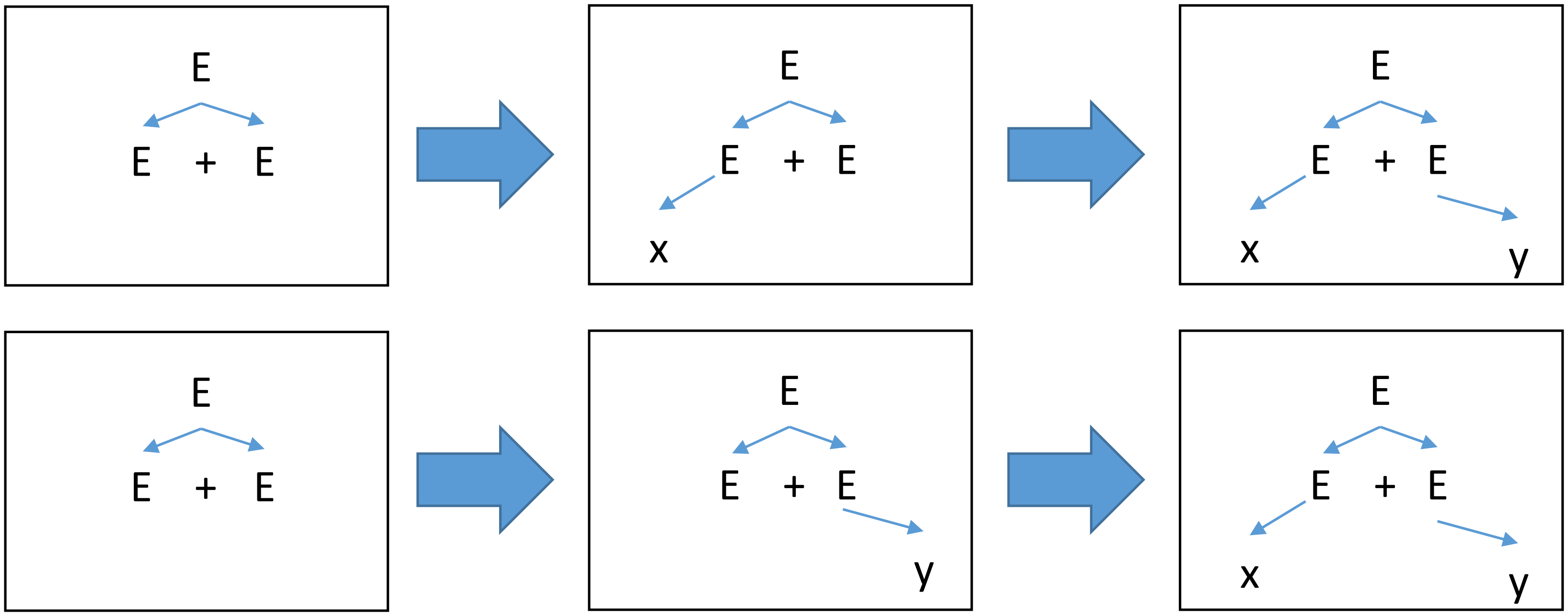
- $P(\ rule \mid prog, position\ )$
  - Prog: 当前已经展开的部分程序
  - Position： 准备展开的终结符的位置
  - Rule： 展开该终结符的概率

程序概率的计算

- 定理：给定任意的规则展开序列，我们有
  - $P(prog) = \prod_i P(\ rule_i \mid prog_i, position_i\ )$

  - $prog_i$: 第i步已经生成的程序
  - $position_i$: 第i步准备展开的非终结符的位置
  - rule:第i步采用的产生式
  - $prog$: 完整程序

程序概率计算的收敛性



- 以上定理表明，任意展开序列都有相同概率

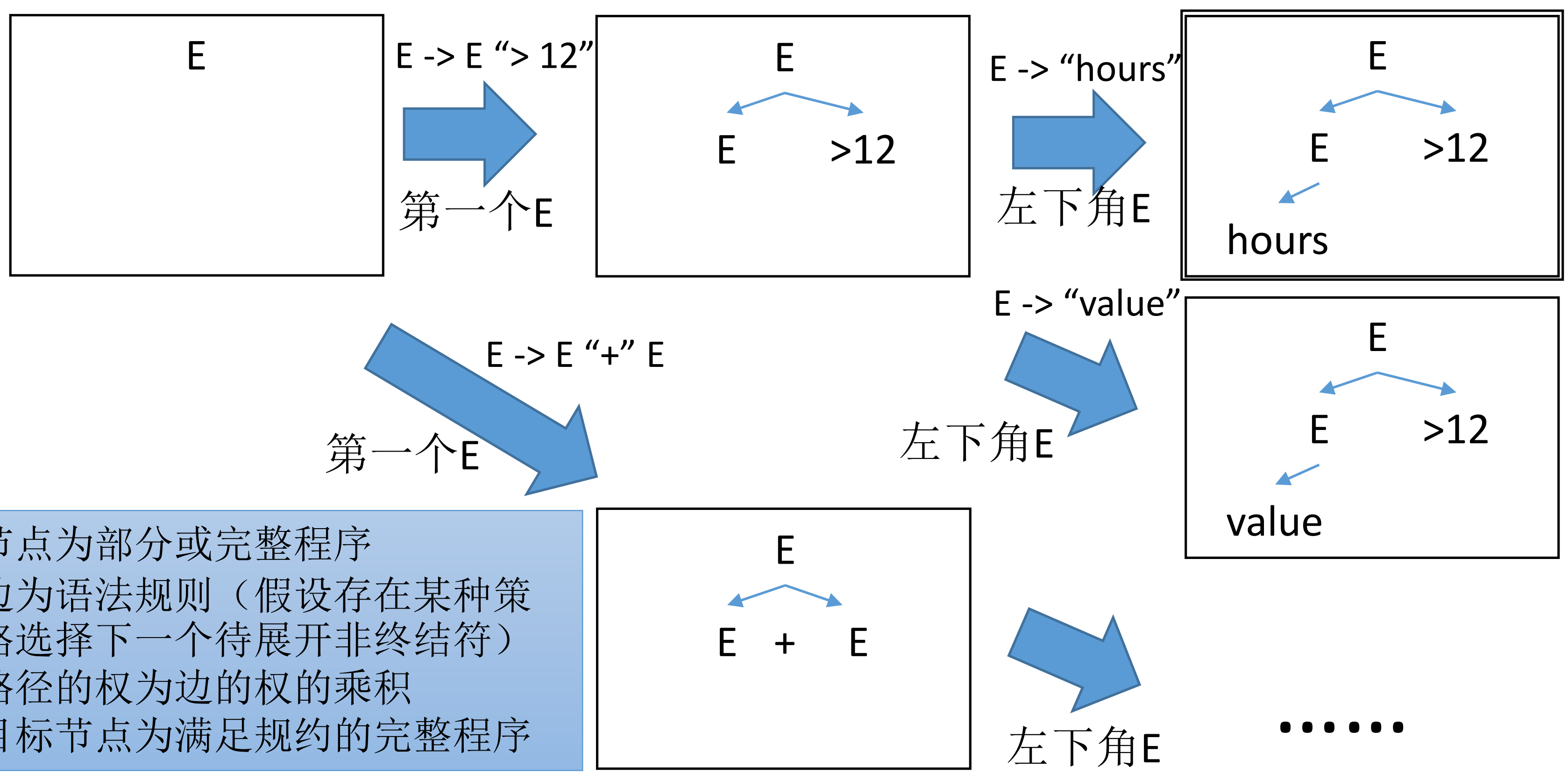
证明

- 假设存在一个policy，决定一个不完整程序中哪个节点先被展开，那么policy的选择和prog的概率是独立的
  - $Pr(prog)$
  - $= \ \mathrm{Pr}(prog \mid policy)$ //独立性
  - $= Pr((\langle prog_i, pos_i, rule_i \rangle)_{i=1}^{n} \mid policy)$
  - $= Pr(\, prog_1 \mid \ policy\,)\, Pr(\, pos_1 \mid \ policy, prog_1\,)$
    $Pr(\, rule_1 \mid \ policy, prog_1, pos_1\,)$
    $Pr(\, eprog_2 \mid \ policy, prog_1, pos_1, rule_1\,) \ldots$
    $Pr(\, eprog_{n+1} \mid \ policy, (eprog_i)_{i=1}^{n}, (pos_i)_{i=1}^{n}, (rule_i)_{i=1}^{n}\,)$
  - $= \prod_i Pr\left(rule_i \mid policy, \left(rule_j\right)_{j=1}^{i-1}, pos_i\right)$ //删除概率为1的项
  - $= \prod_i \mathrm{Pr}(rule_i \mid \ policy, prog_i, pos_i)$
  - $= \prod_i \mathrm{Pr}(rule_i \mid \ prog_i, pos_i)$ //独立性

规则展开概率模型的实现

- 通常计算$P(\ rule_i \mid prog_i, position_i, context\ )$
  - 其中context根据需要可以为程序规约、补全的上下文等
- 可以用任意统计模型或机器学习模型实现

程序估计问题作为路径查找问题



- 节点为部分或完整程序
- 边为语法规则（假设存在某种策略选择下一个待展开非终结符）
- 路径的权为边的权的乘积
- 目标节点为满足规约的完整程序

如何求解概率最大的程序？

- 采用求解路径查找问题的标准算法
- 迪杰斯特拉算法
- 定向搜索（Beam Search）
- A*算法

迪杰斯特拉算法

- 定义节点的权为到达该节点的路径的最大权
- 维护一个可达节点列表，并记录每个节点的权
- 选择(1)权最大的节点, (2)从该节点出发的未探索的权最大的边，将新到达节点加入列表
- 如果某个节点已经没有未探索出边，则从列表中删除
- 反复上一步直到找到目标节点

注：在本问题中只能被一条路径到达，而在一般路径查找问题中，每个节点可以被多条路径达到，所以通用算法还需到达了旧节点时更新最大权。

迪杰斯特拉算法求解的例子

- <E,1>
- <E+E,0.5>, <E-E, 0.4>, ~~<x, 0.05>~~, ~~<y, 0.05>~~
- <E-E, 0.4>, <x+E, 0.3>, <(E+E)+E, 0.1>, <y+E, 0.1>
- <x+E, 0.3>, <x-E, 0.2>, <y-E, 0.1>, <(E+E)+E, 0.1>, <y+E, 0.1>, <(E+E)-E, 0.05>, <(E-E)-E, 0.05>
- ……

定向搜索（Beam Search）

- 在迪杰斯特拉算法中不保留所有节点，只保留概率最大的k个
- 近似算法，不保证最优，也不保证找到结果

A*算法

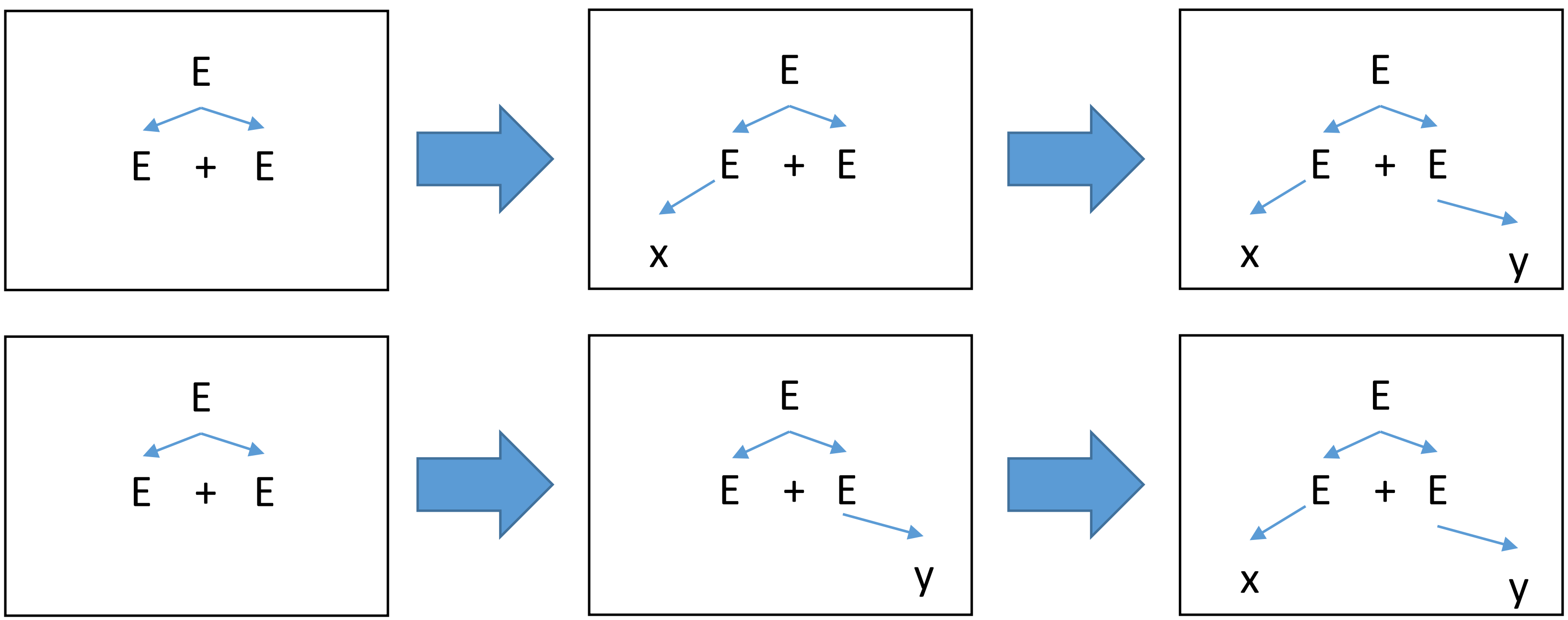
- 节点n的权=到达该节点的权*h(n)
  - h(n)=剩余路径权的上界
- 其他同迪杰斯特拉算法
- 如何知道剩余路径权的上界?
  - 假设存在函数$\hat{P}(rule)$，满足
    - $\forall prog, position: \hat{P}(rule) \leq P(\ rule \mid \ prog, position\ )$
  - 在语法展开式上做静态分析，分析出每个非终结符的概率上界
    - 从 E->E+E | x | y |…
    - 得到方程 $\hat{P}(E) = \max(\hat{P}(E \to E + E)\hat{P}(E)\hat{P}(E), \hat{P}(E \to x), \hat{P}(E \to y), \ldots)$
  - 剩余路径权的上界为所有未展开非终结符概率上界的积

剪枝

- 之前描述的剪枝过程仍然可以用于求解程序估计问题
- 判断出一个部分程序无法满足规约时，从列表中移除对应节点

定义程序展开的顺序

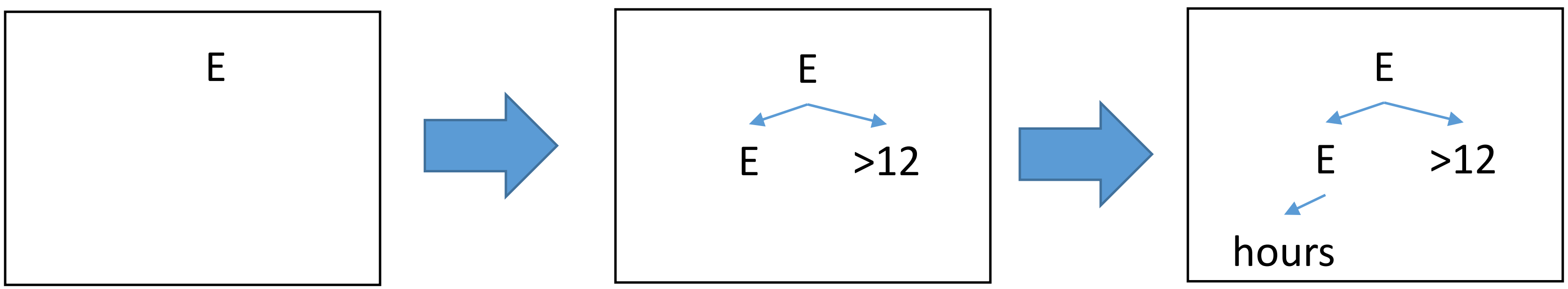
展开的顺序



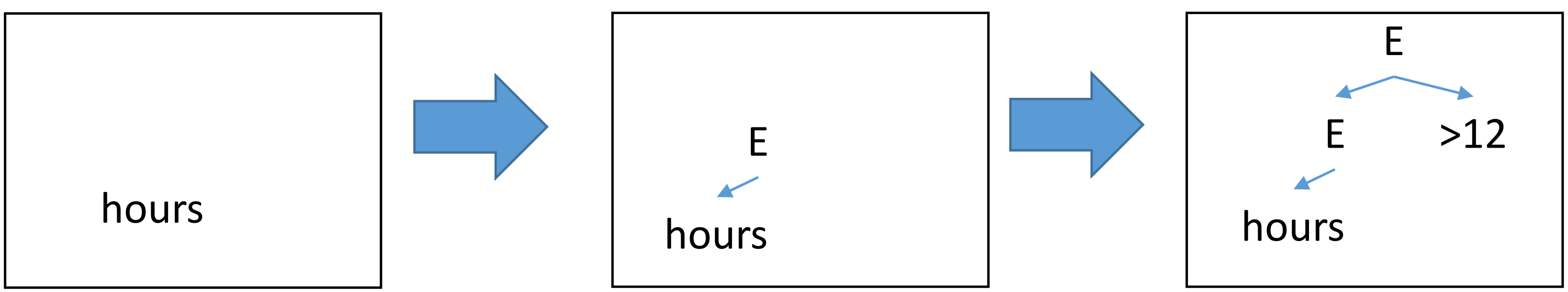
- 如果左下采用E->x的概率极大，而右下采用E->y的概率较低，则上面的顺序能显著减少搜索时间
- 需要根据应用特点定义非终结选择策略

超越上下文无关文法的顺序?

- 自顶向下



- 自底向上



扩展规则

- 允许描述不同方向的语法扩展
- 由本课题组提出
- 通过采用合适的扩展规则，求解效率可提高一倍以上

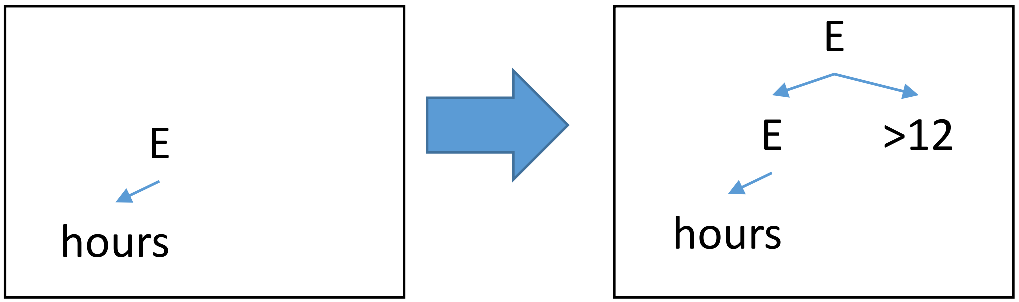
从上下文无关文法到扩展规则

$T \rightarrow E$

$E \rightarrow E\text{ “} > 12\text{”} \mid E\text{ “} > 0\text{”} \mid E\text{ “} + \text{” } E \mid \text{“}hours\text{”} \mid \text{“}value\text{”} \mid \ldots$

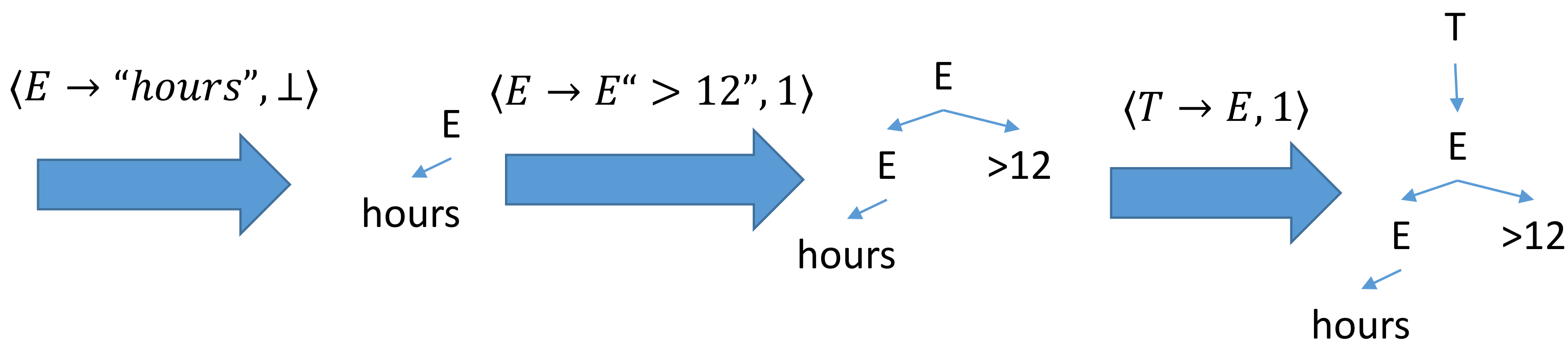
$\langle E \rightarrow \text{“}hours\text{”}, \quad \bot\rangle$
$\langle E \rightarrow \text{“}value\text{”}, \quad \bot\rangle$
$\langle E \rightarrow E\text{ “} > 12\text{”}, \quad 1\rangle$
$\langle E \rightarrow E\text{ “} + \text{” } E, \quad 1\rangle$
$\langle T \rightarrow E, \quad 1\rangle$
$\langle E \rightarrow E\text{ “} > 12\text{”}, \quad 0\rangle$
$\langle E \rightarrow E\text{ “} + \text{” } E, \quad 0\rangle$
$\langle E \rightarrow \text{“}hours\text{”}, \quad 0\rangle$
$\langle E \rightarrow \text{“}value\text{”}, \quad 0\rangle$

自底向上规则：$\langle E \rightarrow E\text{“} > 12\text{”}, 1\rangle$
如果第i个子节点已经产生，产生整棵子树



自顶向下规则：$\langle E \rightarrow E\text{“} > 12\text{”}, 0\rangle$
如果根节点已经产生，产生整颗子树
创建规则：$\langle E \rightarrow \text{“}hours\text{”}, \bot\rangle$
从零产生一颗子树

基于扩展规则的程序生成过程



扩展规则树Expansion Tree

- 抽象语法树在扩展规则上的对应，记录扩展规则如何被应用的

<table>
<tr><th>hours>12</th><th>hours+value</th></tr>
<tr><td>

$$\begin{tikzcd}
(T \to E, 1) \\
(E \to E \text{“} > 12\text{”}, 1) \arrow[u] \\
(E \to \text{“hours”}, \perp) \arrow[u]
\end{tikzcd}$$

</td><td>

$$\begin{tikzcd}
 & (T \to E, 1) & \\
 & (E \to E \text{“}+\text{”}\ E, 1) \arrow[u] \arrow[dr] & \\
(E \to \text{“hours”}, \perp) \arrow[ur] & & (E \to \text{“value”}, 0)
\end{tikzcd}$$

</td></tr>
</table>

抽象语法树 -> 扩展规则树

- **扩展规则的性质**
  - **完整性**: 对任意AST, 至少有一个扩展规则树
  - **唯一性**: 对任意AST, 最多有一个扩展规则树

- 是否总是存在完整和唯一的扩展规则集合?

唯一和完整集合的充分条件

$\text{T} \rightarrow E$

$E \rightarrow E \text{ “} > 12\text{”} \mid E \text{ “} > 0\text{”} \mid E \text{ “} + \text{” } E \mid \text{“}hours\text{”} \mid \text{“}value\text{”} \mid \ldots$

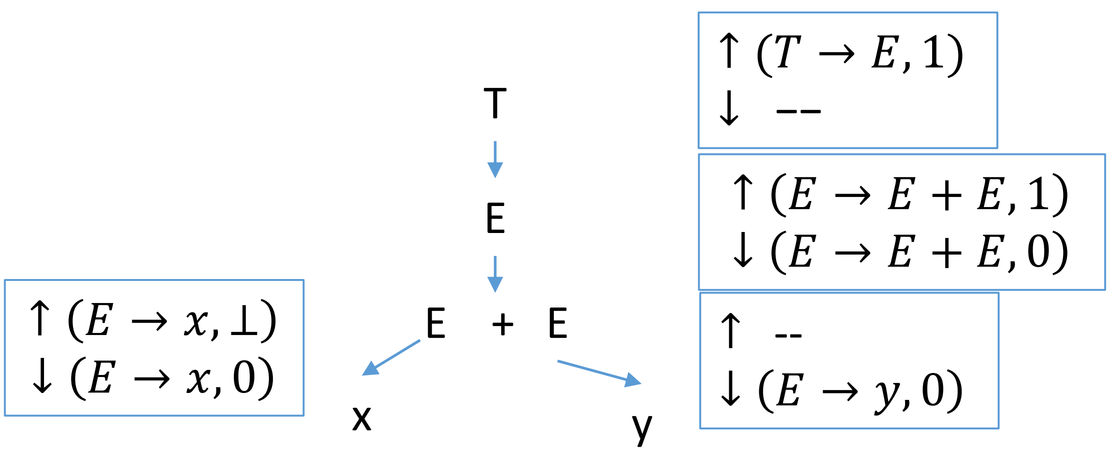
$$
\begin{array}{ll}
\langle E \rightarrow \text{“}hours\text{”}, & \perp\rangle \\
\langle E \rightarrow \text{“}value\text{”}, & \perp\rangle \\
\langle E \rightarrow E \text{ “} > 12\text{”}, & 1\rangle \\
\langle E \rightarrow E \text{ “} + \text{” } E, & 1\rangle \\
\langle T \rightarrow E, & 1\rangle \\
\langle E \rightarrow E \text{ “} > 12\text{”}, & 0\rangle \\
\langle E \rightarrow E \text{ “} + \text{” } E, & 0\rangle \\
\langle E \rightarrow \text{“}hours\text{”}, & 0\rangle \\
\langle E \rightarrow \text{“}value\text{”}, & 0\rangle
\end{array}
$$

1. 除了初始符号开头的规则，所有语法规则都有对应的自顶向下展开规则
2. 所有语法规则最多只有一条自底向上的展开规则
3. 对于所有从初始符号（延自底向上展开规则）反向可达的非终结符，其所有语法规则都有一条自底向上展开规则或创建规则

从初始符号开始选择创建/自底向上规则即可

抽象语法树 -> 扩展规则树

- 利用一个动态规划算法，AST可以在O(n)时间内转成Expansion Tree
  - 后根次序依次判断每个AST结点是否可以被自底向上和自顶向下的方式生成，如果可以，记录下采用的规则
  - 先根次序恢复出Expansion Tree



求解程序估计问题

- 给定某种结点选择策略，可以从扩展规则树得到展开序列
- 同样看做路径查找问题求解

扩展FlashMeta求解程序估计问题

FlashMeta vs 程序估计问题

- 能否直接采用FlashMeta求解程序估计问题?
- 之前的方法无法使用
  - 子问题无法支持基于部分模型的概率模型
  - 分解子问题后难以看做路径查找问题
- MaxFlash
  - 2020年由北京大学吉如一等人提出
  - 效率超过FlashMeta达400-2000倍

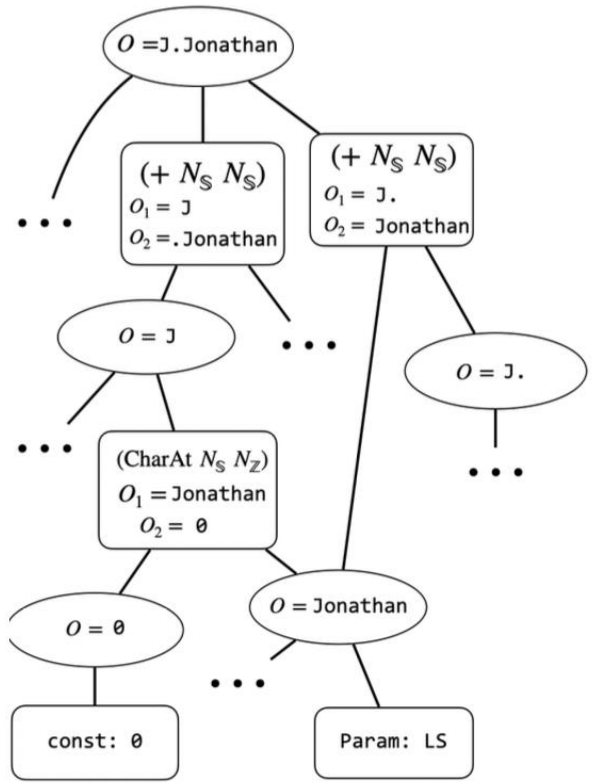
例：字符串处理程序合成

Start symbol	S	$\rightarrow$	$N_{\mathbb{S}} \mid N_{\mathbb{Z}}$
String expr	$N_{\mathbb{S}}$	$\rightarrow$	Parameters $\mid$ (+ $N_{\mathbb{S}}$ $N_{\mathbb{S}}$)
		$\mid$	(CharAt $N_{\mathbb{S}}$ $N_{\mathbb{Z}}$) $\mid$ '.'
Integer value	$N_{\mathbb{Z}}$	$\rightarrow$	0 $\mid$ 1

('John','Jonathan') $\rightarrow$ 'J.Jonathan'

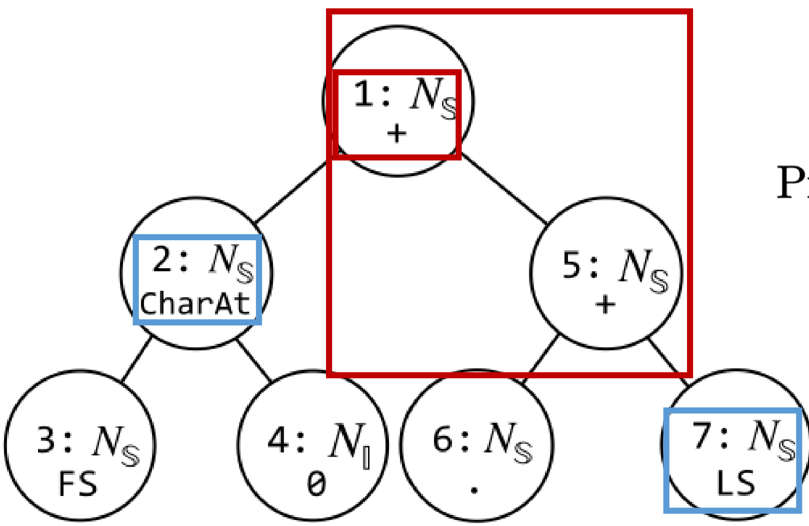
(+ (CharAt *FS* 0) (+ '.' *LS*))

FlashMeta求解过程



概率模型：自顶向下预测模型 Topdown Prediction Model

- **TPM**: 节点展开规则概率只取决于其祖先，即兄弟节点的展开规则相互独立
- **示例:**



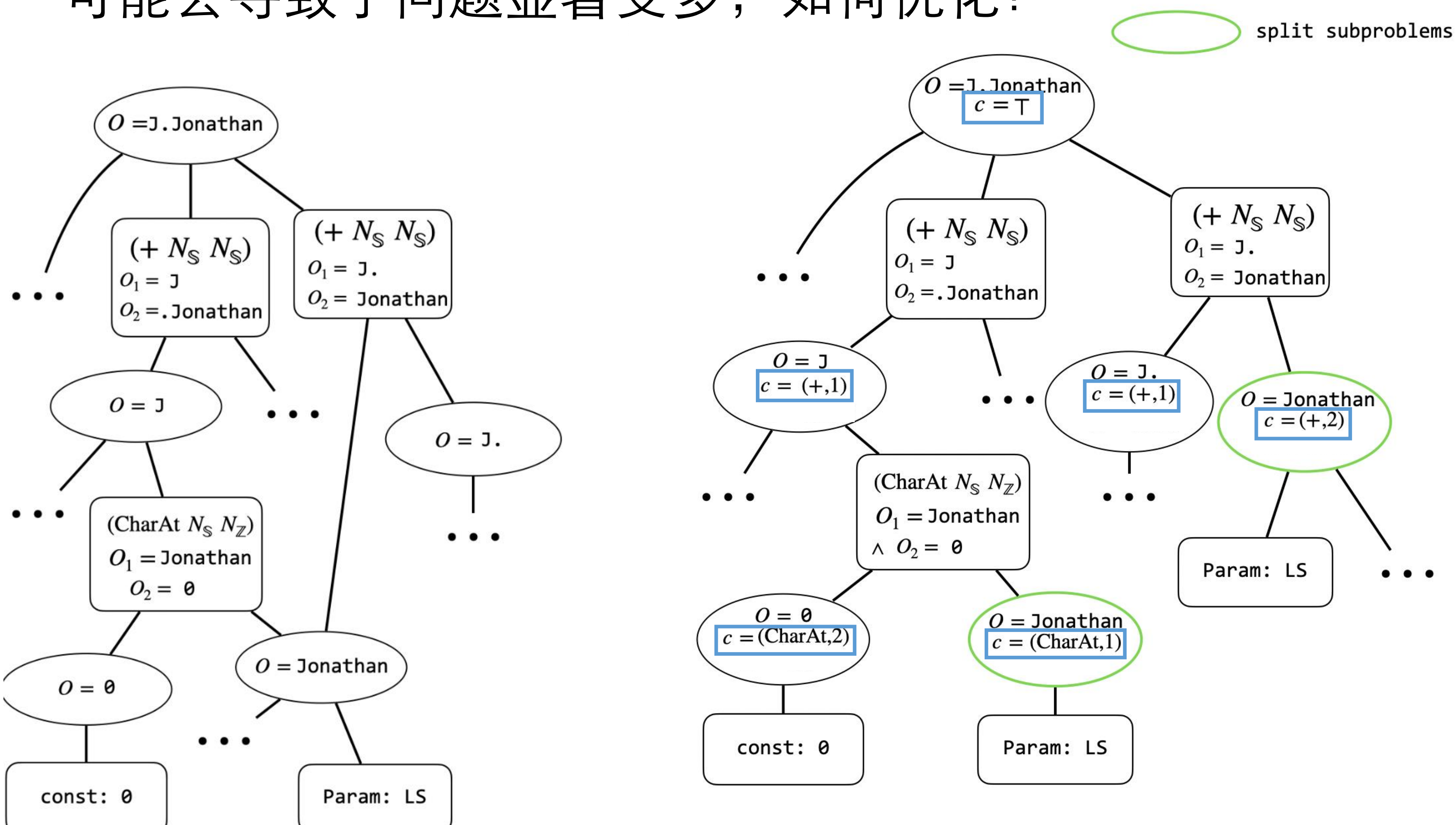
$$\Pr[\texttt{CharAt} \text{ on vertex } 2] = \Pr[\texttt{CharAt} \mid (+, 1)]$$

$$\Pr[\texttt{LS} \text{ on vertex } 7] = \Pr[\texttt{LS} \mid (+, 2), (+, 1)]$$

通常定义为依赖最近k层祖先节点

在FlashMeta中引入概率

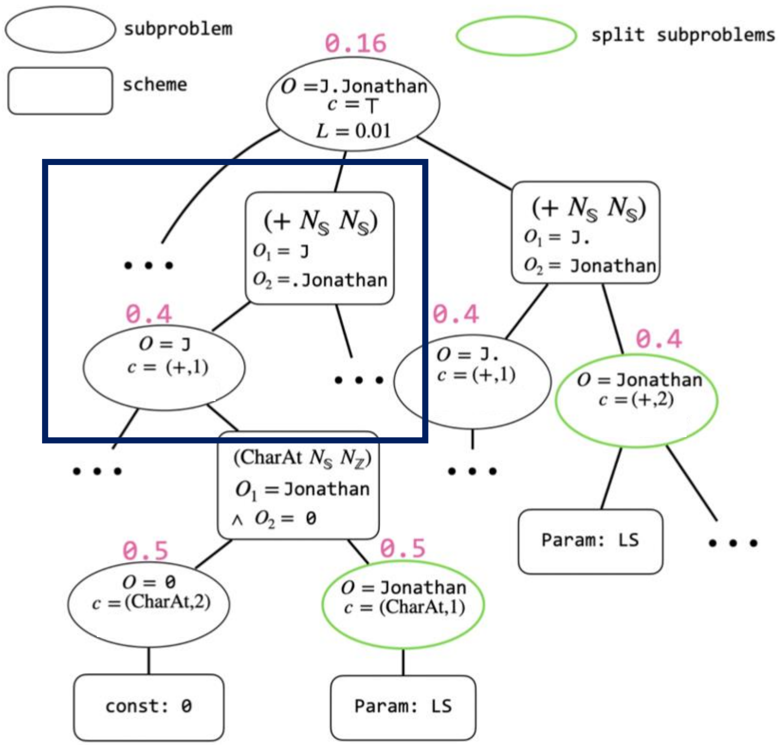
- 为子问题添加祖先节点的上下文信息
- 可能会导致子问题显著变多，如何优化?



搜索方法

- 分支限界 (Branch and Bound)
  - 在搜索时对每一个子问题带上一个概率下界l，表示只有在当前子程序的概率大于l的时候，才有可能形成一个可能的程序
- 迭代加深 (Iteratively Deepening)
  - 设置全局的概率下界，如果找不到解则放宽全局的概率下界
- 估价函数 (Heuristic Function)
  - 类似之前A*算法的估价函数
  - 估计每一个子问题的最优解上界
    - 如：每个非终结符的概率上界
    - 如：每个<上下文-非终结符>的概率上界

示例



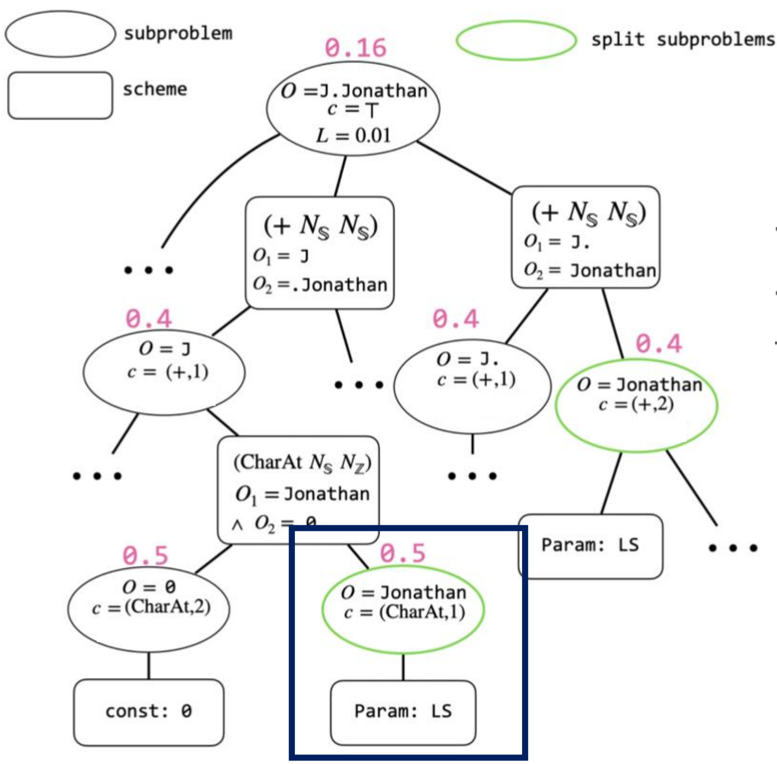
	CharAt	+	FS	LS	'.'	0	1
Start	0	1	0	0	0	0	0
(+, 1)	0.5	0.01	0.09	0	0.4	0	0
(+, 2)	0.05	0.5	0.05	0.4	0	0	0
(CharAt, 1)	0.05	0.05	0.5	0.4	0	0	0
(CharAt, 2)	0	0	0	0	0	0.5	0.5

$$\Pr[+ \mid \text{empty}] = 1$$

$$\boxed{(\underbrace{+}_{1}\ \ P_1\ \ \underbrace{P_2}_{\leqslant 0.4})}\ \geq 0.01$$

P_1: ≥0.025

示例



	CharAt	+	FS	LS	’.’	0	1
(CharAt, 1)	0.05	0.05	0.5	0.4	0	0	0

跳过

	CharAt	+	FS	LS	’.’	0	1
Start	0	1	0	0	0	0	0
(+, 1)	0.5	0.01	0.09	0	0.4	0	0
(+, 2)	0.05	0.5	0.05	0.4	0	0	0
(CharAt, 1)	0.05	0.05	0.5	0.4	0	0	0
(CharAt, 2)	0	0	0	0	0	0.5	0.5

如何重用子问题

- 重复子问题的结果复用是动态规划的核心
  - FlashMeta 的子问题：当前的非终结符 S，输入输出用例 A
  - 目前的子问题：非终结符 S，输入输出用例 A，**概率下界 L**，上下文 c
- 需要复用概率下界不同的子问题
  - 考虑两个除了概率下界不同以外，其他都一样的子问题 (P, 0.2), (P, 0.1)
  - Case 1: (P, 0.2) 先于 (P, 0.1)
    - 有解，则同样是 (P, 0.1) 的解；
    - 无解，则可以更新 P 的估价函数
  - Case 2: (P, 0.1) 先于 (P, 0.2)
    - 有解，则同样是 (P, 0.2) 的解（因为总是搜索概率最大的结果）
    - 无解，(P, 0.2) 同样无解

MaxFlash总结

- 分支限界和启发式函数：给定概率下界的时候，只搜索部分满足下界的程序空间
- 迭代加深：逐步放宽概率下界，避免一次探索过大空间
- 子问题重用：概率下界不同的子问题也能复用，使得不同迭代之间的计算不会被浪费

参考文献

- Yingfei Xiong, Bo Wang, Guirong Fu, Linfei Zang. Learning to Synthesize. GI'18: Genetic Improvment Workshop, May 2018.
- Ruyi Ji, Yican Sun, Yingfei Xiong, Zhenjiang Hu. Guiding Dynamic Programing via Structural Probability for Accelerating Programming by Example. OOPSLA'20: Object-Oriented Programming, Systems, Languages, and Applications 2020, November 2020.