Week 5 and Week 6

> 首先是提出算法实现前的一些铺垫，主要包括以下知识：
>
> *1.Top-Down and Bottom-Up*
>
> *2.Shift-Reduce Parsing*
>
> *3.一些notation为了算法实现做铺垫*

1.Top-Down and Bottom-Up

Top-Down对应先前讲的leftmost，而Bottom-Up则对应先前讲的rightmost，Week 4是从易于理解的角度讲解，而本周则是从实现的角度讲解。主要涉及到了如何shift，如何reduce，以及如何生成树，但是没有讲什么时候。

2.Shift-Reduce Parsing

基于Top-Down and Bottom-Up引出的语法解析实现的简单模型，初步引入了一些notation，如

- Split string into two substrings: $\alpha \bullet \beta$
  - where $\alpha \in (N \cup T)^*$ and $\beta \in T^*$
  - Right sub-string is not examined yet; has only terminals
  - Left sub-string has terminals and non-terminals
- The dividing point is marked by a $\bullet$
  - $\bullet$ is not a part of the string
- Initially, all input is unexamined $\bullet x_1 x_2 \ldots x_n$

主要解决了在哪reduce的问题

- Left part of the string is implemented by a stack
  - Top of the stack is left of the ●
- Shift pushes a terminal on the stack
- Reduce
  - Pops 0 or more symbols off of the stack (rhs of one rule from the CFG)
  - Pushes a non-terminal on the stack (lhs of one rule from the CFG)

期间讲了一个插曲，也就是冲突（conflict）的解决方法：

一个为：shift和reduce的冲突，可被precedence and associativity declaration修复，类似于leftmost 或者rightmost，如加法和乘法的优先级。

另一个： reduce-reduce conflict，不知道该选择哪一个规则，说明： There is ambiguity in the grammar ， Might be fixed by additional lookahead，这个类似于两个规则冲突

3.When to shift/reduce?

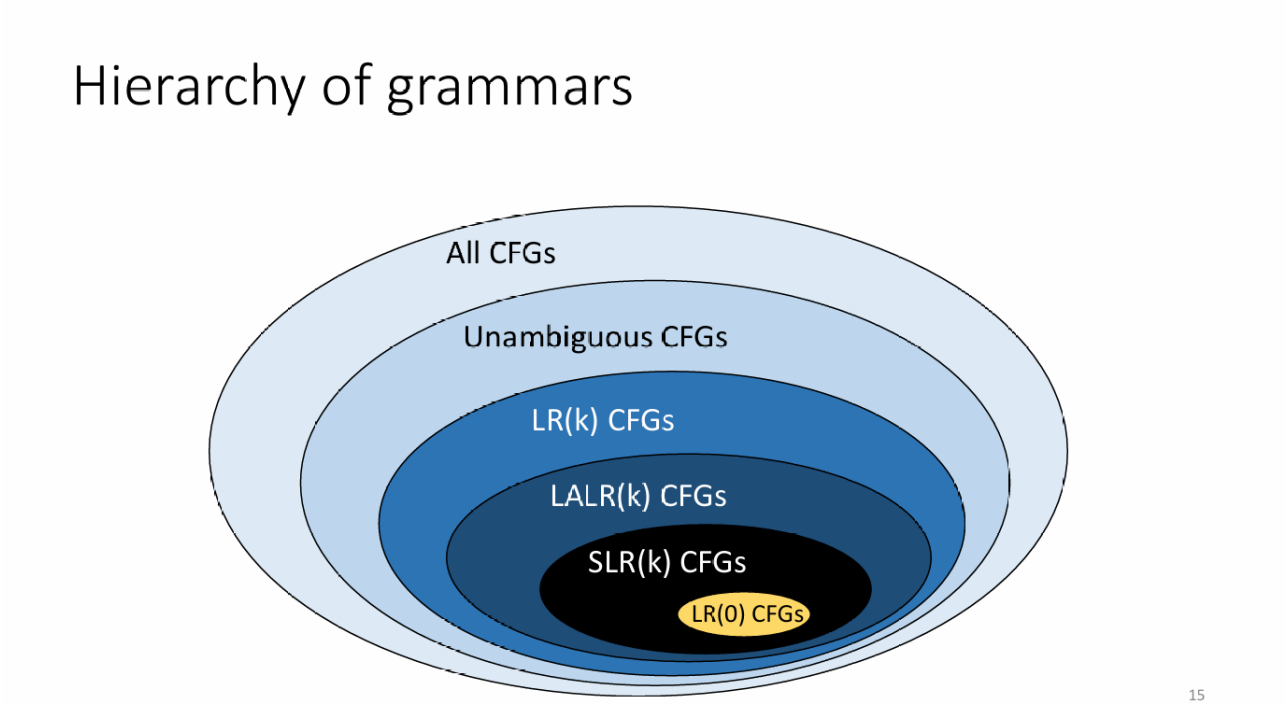
接着讲什么时候解决冲突，主要引入了prefix and handle

Viable Prefix and Handle

- Intuition: reduce only if we can eventually reach the start symbol
- Assume a rightmost derivation
  - $S \Rightarrow^* \alpha X \beta \Rightarrow \alpha w \beta$

    ⟵ reduction
- Then αw is a viable prefix of $\alpha w \beta$
  - A handle w is valid if we can reduce w to X
  - We only reduce a handle
- A handle *always* appears on top of the stack, never inside

然后将了语法规范（specific）和parser的关系，核心在这张图：基本观点是，规范越简单解析器越简单，解析器无法准确识别所用的上下文无关语法（CFG），解析器的实现也需要效率和功能的trade-off



3.When to shift/reduce?realize

algorithm 1（LR0）

> 关键在于构造*action*和*goto*两张表格
>
> 构造上述两张表格需要知道*Configuration set*
>
> 构造*configuration set*需要知道如何计算*Closure property*

- action和goto表格的状态转移可看下表：

  只需要注意一点的就是其状态转移只和stack头的元素有关，当弹出元素之后的状态转移也是如此

Trace “(id)*id”

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)

0 is the start state

	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Stack	Input	Action/Goto
0	(id) * id $	Shift S5
0 5	id) * id $	Shift S8
0 5 8	) * id $	Reduce 3 F→id, pop 8, goto [5,F]=1
0 5 1	) * id $	Reduce 1 T→ F, pop 1, goto [5,T]=6
0 5 6	) * id $	Shift S7
0 5 6 7	* id $	Reduce 4 F→ (T), pop 7 6 5, goto [0,F]=1
0 1	* id $	Reduce 1 T → F pop 1, goto [0,T]=2

10

- closure的构造

  把所有的非终结符递归的转换即可

Example: I = closure(S’ → • T)

S’ → • T

T → • T * F

T → • F

F → • id

F → • (T)

S’ → T
T → F | T * F
F → id | (T)

- Successor(I, X)的构造

> 类似于状态转移后的状态的构造，状态转移需要移动圆点

Successor Example

I = {S' → • T,
T → • F,
T → • T * F,
F → • id,
F → • (T) }

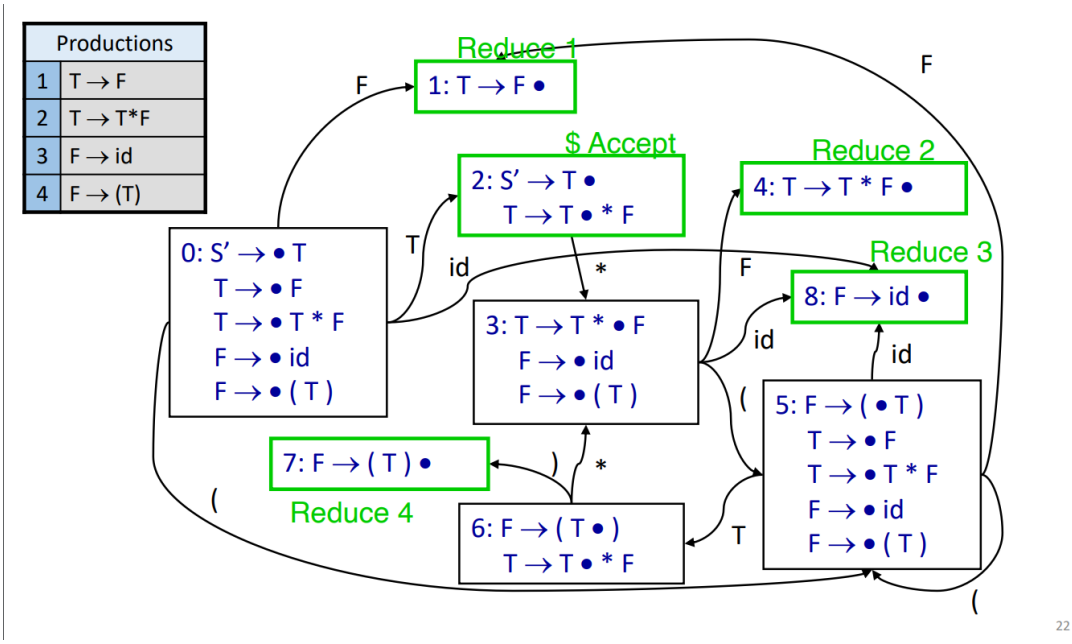
S' → T
T → F | T * F
F → id | (T)

Compute Successor(I, '(')

{ F → (• T), T → • F, T → • T * F,
F → • id, F → • (T) }

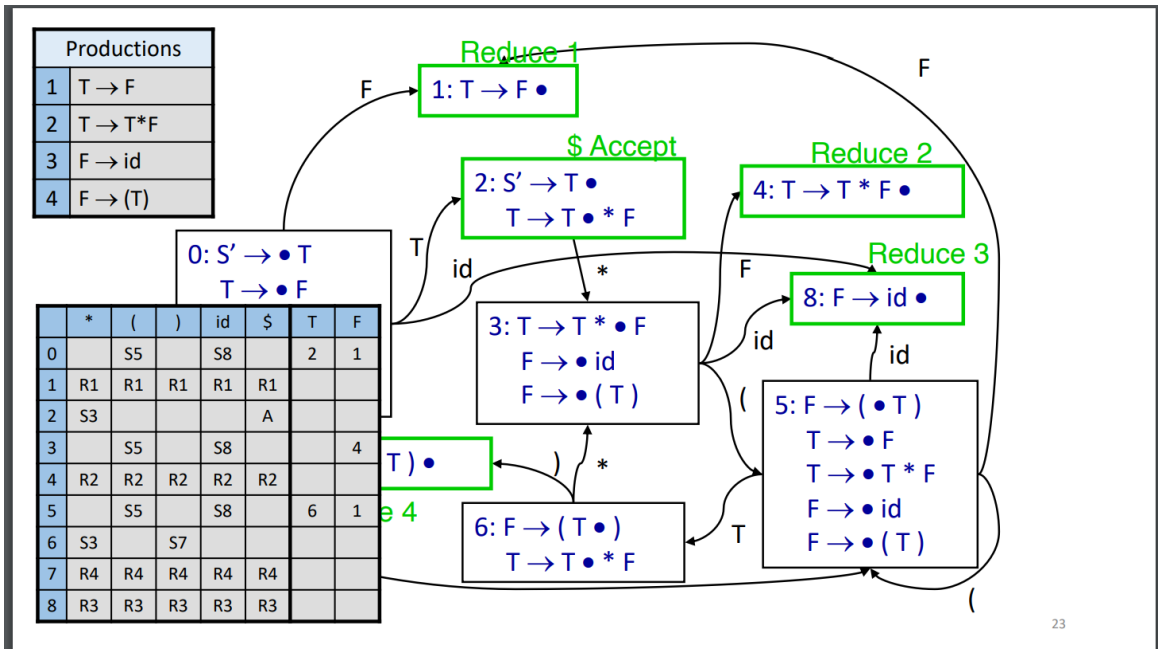
20

- configuration set的构造



22

- action和goto的表格获得



	*	(	)	id	$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

23

- 需要注意的地方

> *1.*状态不要重复，状态也不能少
>
> *2.*若出现冲突说明*grammar*定义的不好

如此例子：则出现了reduce和shift冲突和reduce和reduce冲突，注意有的并非冲突，可以一个状态多次操作