11-711 Algorithms for NLP

LR Parsing

Reading:

Hopcroft and Ullman,

“Intro. to Automata Theory, Lang. and Comp.”

Section 10.6-10.7, pp. 248–256

Shift-Reduce Parsing

A class of parsers with the following principles:

- Parsing is done *Bottom-Up*, reducing the input into the grammar start symbol
- The parser builds a right-most derivation of the input *in reverse*
- Parsing algorithm simulates the operation of a PDA
- Prefix of the sentential form is kept on the stack
- Two types of operation:
  - **Shift** the next input symbol onto the stack
  - **Reduce** the stack by popping the RHS of a grammar rule, and pushing the corresponding LHS non-terminal symbol
- Parser is usually *deterministic* and with no back-tracking
- Extremely efficient, operating in linear time - $O(n)$
- But - possible to construct for only a limited class of CFGs

LR Parsing

General Principles:

- Use sets of "dotted" grammar rules to reflect the *state* of the parser:
  - What constituents have we constructed so far
  - What constituents are we predicting next
- Pre-compile the grammar into a collection of *finite* sets of "dotted" rules
- Use these sets to capture the state of the parser during parsing
- The Parser is a deterministic shift-reduce parser.
- Developed by Knuth in the late 1960s - as a framework for compiling programming languages

LR Parsing Algorithm

- Performs shift and reduce parsing actions on the stack, and changes state with each operation
- Is driven by a pre-compiled parsing table that has two parts
  - The *action* table specifies the next shift or reduce parsing operation
  - The *goto* table specifies which state to transfer to after a reduction
- The stack stores a string of the form $s_0 X_1 s_1 X_2 ... X_m s_m$ where the s_i are parser states and the X_i are grammar symbols
- At each step the parser does *one* of the following types of operations:
  - **Shift(s):** Push the current input symbol x_i on the stack followed by the new state s
  - **Reduce(i):** Reduce the stack according to rule i of the grammar
  - **Reject:** Reject the input as ungrammatical and signal an error
  - **Accept:** Accept the input as grammatical and halt

The LR Parsing Algorithm

The Algorithm in Pseudo-code:

set *ip* to point to the first symbol of *w*$;
repeat forever begin
    let *s* be the state on top of the stack and
        a the symbol pointed to by *ip*;
    if *action*[*s*, *a*] = shift *s'* **then begin**
        push *a* then *s'* on top of the stack;
        advance *ip* to the next input symbol
    end
    else if *action*[*s*, *a*] = reduce $A \rightarrow \beta$ **then begin**
        pop $2 * |\beta|$ symbols off the stack;
        let *s'* be the state now on top of the stack;
        push *A* then *goto*[*s'*, *A*] on top of the stack;
        output the production $A \rightarrow \beta$
    end
    else if *action*[*s*, *a*] = accept **then**
        return
    else *error*()
end

LR Parsing - Example

The Grammar:

$$
\begin{aligned}
(1)\ S &\rightarrow NP\, VP \\
(2)\ NP &\rightarrow art\ adj\ n \\
(3)\ NP &\rightarrow art\ n \\
(4)\ NP &\rightarrow adj\ n \\
(5)\ VP &\rightarrow aux\ VP \\
(6)\ VP &\rightarrow v\ NP
\end{aligned}
$$

The original input: “ $x =$ The large can can hold the water”
POS assigned input: “ $x =$ art adj n aux v art n”
Parser input: “ $x =$ art adj n aux v art n $”

LR Parsing - Example

Constructed Parsing Table for the Grammar:

	Reduce	Shift						Goto		
State		art	adj	n	aux	v	$	NP	VP	S
0		sh3	sh4					2		1
1							acc			
2					sh6	sh7			5	
3			sh8	sh9						
4				sh10						
5	r1									
6					sh6	sh7			11	
7		sh3	sh4					12		
8				sh13						
9	r3									
10	r4									
11	r5									
12	r6									
13	r2									

LR Parsing - Example

The input: "$x =$ art adj n aux v art n \$"

Step	Action	Stack after action
0		s_0
1	sh3	$s_0\ art\ s_3$
2	sh8	$s_0\ art\ s_3\ adj\ s_8$
3	sh13	$s_0\ art\ s_3\ adj\ s_8\ n\ s_{13}$
4	r2	$s_0\ NP\ s_2$
5	sh6	$s_0\ NP\ s_2\ aux\ s_6$
6	sh7	$s_0\ NP\ s_2\ aux\ s_6\ v\ s_7$
7	sh3	$s_0\ NP\ s_2\ aux\ s_6\ v\ s_7\ art\ s_3$
8	sh9	$s_0\ NP\ s_2\ aux\ s_6\ v\ s_7\ art\ s_3\ n\ s_9$
9	r3	$s_0\ NP\ s_2\ aux\ s_6\ v\ s_7\ NP\ s_{12}$
10	r6	$s_0\ NP\ s_2\ aux\ s_6\ VP\ s_{11}$
11	r5	$s_0\ NP\ s_2\ VP\ s_5$
12	r1	$s_0\ S\ s_1$
13	accept	$s_0\ S\ s_1$

Constructing an SLR Parsing Table

- An $LR(0)$ item is a "dotted" grammar rule $[A \rightarrow \alpha \bullet B \ \beta]$
- We construct a deterministic FSA that recognizes prefixes of rightmost sentential forms of the grammar G.
- The states of the FSA are sets of LR(0) items
- We augment the grammar with a new start rule $S' \rightarrow S$
- We define the *closure* operation on a set S of $LR(0)$ items:
  1. Every item in S is also in $closure(S)$
  2. If $[A \rightarrow \alpha \bullet B \ \beta] \in closure(S)$ and $B \rightarrow \gamma$ is a rule in G, then add $[B \rightarrow \bullet\gamma]$ to $closure(S)$
- The closure operation adds "predicted" new items to the set S (similar to Earley's *Predictor* operation)

Constructing an SLR Parsing Table

- We define the *Goto* operation for an item set S and a grammar symbol X:
  $Goto(S, X)$ is the closure of the set of all items $[A \rightarrow \alpha\ X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta] \in S$
- Example: $S_0 = \{[S \rightarrow \bullet NP\ VP]\}$

$$Goto(S_0, NP) = \left\{ \begin{array}{l} S \rightarrow NP \bullet VP \\ VP \rightarrow \bullet\ aux\ VP \\ VP \rightarrow \bullet\ v\ NP \end{array} \right\}$$

- Similar to Earley's *Scanner* and *Completer* operations

Constructing an SLR Parsing Table

- We construct the collection of sets of $LR(0)$ items for an augmented grammar G
- We start with the item set $S_0 = \{closure(\{[S' \rightarrow \bullet S]\})\}$
- The algorithm:

```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Constructing an SLR Parsing Table - Example

Building the collection of sets of $LR(0)$ items for our simple NL Grammar:

S_0: $S' \rightarrow \bullet S$
$S \rightarrow \bullet NP\ VP$
$NP \rightarrow \bullet\ art\ adj\ n$
$NP \rightarrow \bullet\ art\ n$
$NP \rightarrow \bullet\ adj\ n$

S_1: $S' \rightarrow S \bullet$

S_2: $S \rightarrow NP \bullet VP$
$VP \rightarrow \bullet\ aux\ VP$
$VP \rightarrow \bullet\ v\ NP$

S_3: $NP \rightarrow art \bullet adj\ n$
$NP \rightarrow art \bullet n$

S_4: $NP \rightarrow adj \bullet n$

S_5: $S \rightarrow NP\ VP \bullet$

S_6: $VP \rightarrow aux \bullet VP$
$VP \rightarrow \bullet\ aux\ VP$
$VP \rightarrow \bullet\ v\ NP$

S_7: $VP \rightarrow v \bullet NP$
$NP \rightarrow \bullet\ art\ adj\ n$
$NP \rightarrow \bullet\ art\ n$
$NP \rightarrow \bullet\ adj\ n$

S_8: $NP \rightarrow art\ adj \bullet n$

S_9: $NP \rightarrow art\ n \bullet$

S_{10}: $NP \rightarrow adj\ n \bullet$

S_{11}: $VP \rightarrow aux\ VP \bullet$

S_{12}: $VP \rightarrow v\ NP \bullet$

S_{13}: $NP \rightarrow art\ adj\ n \bullet$

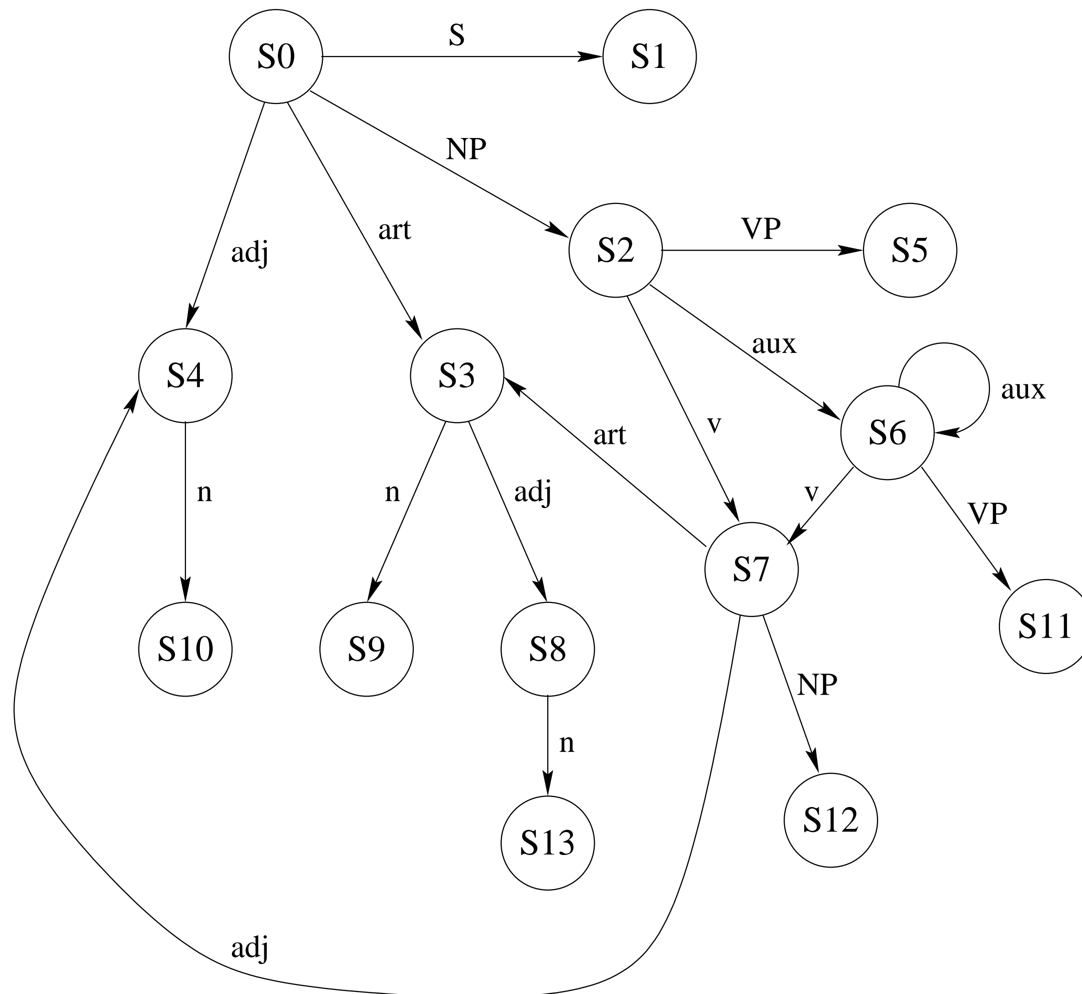
Constructing an SLR Parsing Table

Building the FSA and Parsing Table:

1. Construct the collection of sets of $LR(0)$ items from the grammar
2. State s_i corresponds to item set S_i
   (a) For any terminal symbol a, if $[A \rightarrow \alpha \bullet a\ \beta] \in S_i$ and $Goto(S_i, a) = S_j)$ then set $action(s_i, a) =$ "$shift\ s_j$ "
   (b) If $[A \rightarrow \alpha\bullet] \in S_i$ and $A \rightarrow \alpha$ is rule i then set $action(s_i, a) =$ "$reduce\ i$" for all terminal symbols a
   (c) If $[S' \rightarrow S\bullet] \in S_i$ then set $action(s_i, \$) = Accept$
   (d) For any non-terminal symbol B, if $[A \rightarrow \alpha \bullet B\ \beta] \in S_i$ and $Goto(S_i, B) = S_j)$ then set $goto(s_i, B) = s_j$
   (e) All table entries not defined in (a)-(d) are set as "error"

Constructing an SLR Parsing Table - Example

The constructed FSA for our example grammar:



LR(k) Parsing

How to handle conflicts in the SLR table:

- A table *conflict*: more than one action is specified in $action(s_i, a)$
- Conflicts can be either *shift-reduce* or *reduce-reduce*
- Parser will not be able to parse deterministically
- A Grammar for which this happens is *not* SLR
- More powerful techniques for building item sets can sometimes resolve the problem, by making use of lookaheads into the input
- Known techniques: Canonical LR(k), LALR(k)
- A lookahead of *one* is sufficient (and optimal) in many cases
- Another option - extending the LR Parsing algorithm: **GLR Parsing**

Parsing with an LR Parser

- The pointers that form the parse tree can be created while performing reduce actions
- A *parse node* is created for each constituent that is pushed onto the stack
- When we do a reduce - we create a new parse node for the LHS non-terminal and link it to the parse-nodes of the popped RHS constituents
- At the end - the S constituent on the stack points to the root of the parse tree

LR Parsing - Example

The input: “$x =$ art adj n aux v art n $”

Step	Action	Stack after action	Parse Node
0		s_0	
1	sh3	$s_0\ art_1\ s_3$	1 art
2	sh8	$s_0\ art_1\ s_3\ adj_2\ s_8$	2 adj
3	sh13	$s_0\ art_1\ s_3\ adj_2\ s_8\ n_3\ s_{13}$	3 n
4	r2	$s_0\ NP_4\ s_2$	4 NP (1 2 3)
5	sh6	$s_0\ NP_4\ s_2\ aux_5\ s_6$	5 aux
6	sh7	$s_0\ NP_4\ s_2\ aux_5\ s_6\ v_6\ s_7$	6 v
7	sh3	$s_0\ NP_4\ s_2\ aux_5\ s_6\ v_6\ s_7\ art_7\ s_3$	7 art
8	sh9	$s_0\ NP_4\ s_2\ aux_5\ s_6\ v_6\ s_7\ art_7\ s_3\ n_8\ s_9$	8 n
9	r3	$s_0\ NP_4\ s_2\ aux_5\ s_6\ v_6\ s_7\ NP_9\ s_{12}$	9 NP (7 8)
10	r6	$s_0\ NP_4\ s_2\ aux_5\ s_6\ VP_{10}\ s_{11}$	10 VP (6 9)
11	r5	$s_0\ NP_4\ s_2\ VP_{11}\ s_5$	11 VP (5 10)
12	r1	$s_0\ S_{12}\ s_1$	12 S (4 11)
13	accept	$s_0\ S\ s_1$	