CPS 140 - Mathematical Foundations of CS
Dr. S. Rodger
Section: LR Parsing (handout)

LR PARSING

LR(k) Parser

- bottom-up parser
- shift-reduce parser
- L means: reads input left to right
- R means: produces a rightmost derivation
- k - number of lookahead symbols

LR parsing process

- convert CFG to PDA
- Use the PDA and lookahead symbols

Convert CFG to PDA

Idea: To derive a string from a CFG with a rightmost derivation, start with the start symbol and repeatedly apply productions replacing the rightmost variable at each step. In order to simulate this process with an NPDA, we will simulate this process in *reverse* by starting with the input string, using productions in reverse (replacing rhs of a production by its lhs), and deriving the start symbol. Thus, the NPDA starts by shifting the symbols of the input string onto the stack. Whenever the top symbols on the stack match the rhs of a production, pop the rhs (may be several symbols) and replace it (or push) the lhs on the stack. If replacements lead to only the start symbol on the stack, then the input string is in the language of the grammar. To see the actual rightmost derivation the NPDA simulated, start with the start symbol and apply the productions in the reverse order they were applied in the NPDA.

The constructed NPDA:

- three states: s, q, f
  start in state s, assume z on stack
- all rewrite rules in state s, backwards
  rules pop rhs, then push lhs
  (s,lhs) $\in \delta$(s,ϵ,rhs)
  Note: assuming stack can pop several symbols at once.
  This is called a reduce operation.

- additional rules in s to recognize terminals

  For each $x \in \Sigma$, $g \in \Gamma$, $(s,xg) \in \delta(s,x,g)$

  This is called a shift operation.
- pop S from stack and move into state q
- pop z from stack, move into f, accept.

Example: Construct a PDA.

S $\rightarrow$ aSb
S $\rightarrow$ b

LR Parsing Actions

1. shift

   transfer the lookahead to the stack
2. reduce

   For X $\rightarrow$ w, replace w by X on the stack
3. accept

   input string is in language
4. error

   input string is not in language

LR(1) Parse Table

- Columns:

  terminals, $ and variables

- Rows:

  state numbers: represent patterns in a derivation

LR(1) Parse Table Example

1) S → aSb
2) S → b

	a	b	$	S
0	s2	s3		1
1			acc	
2	s2	s3		4
3		r2	r2	
4		s5		
5		r1	r1	

Definition of entries:

- sN - Shift (or push) the terminal for this column onto the stack, and move to state (or row number) N.
- N - Move to state (or row number) N.
- rN - Reduce by rule number N. The rhs of this rule is on the top of the stack. Pop it and replace it by the lhs of the rule.
- acc - The input string is accepted.
- blank - Error.

LR(1) Parsing routine

"entry" is a record with four parts: state, action, rule.rhs, rule.lhs

```
state = 0
push(state)
read(symbol)                              obtain the lookahead symbol
entry = T[state,symbol]                   T is the LR parse table
while entry.action ≠ accept do
    if entry.action == shift then
        push(symbol)
        state = entry.state
        push(state)
        read(symbol)
    else if entry.action == reduce then
        do 2*size_rhs times {pop()}       pop entry.rule.rhs and states
        state := top-of-stack()           do not pop!
        push(entry.rule.lhs)
        state = T[state,entry.rule.lhs]
        push(state)
    else if entry.action == blank then
        error
    entry = T[state, symbol]
end while
if symbol ≠ $ then error
```

Example:

Trace aabbb

						5			
						b			
				3	4	4		5	
				b	S	S		b	
			2	2	2	2	4	4	
			a	a	a	a	S	S	
		2	2	2	2	2	2	2	1
		a	a	a	a	a	a	a	S
	0	0	0	0	0	0	0	0	0
S:	z	z	z	z	z	z	z	z	z
Lookahead:	a	a	b	b	b	b	b	$	$
Action:									

To construct the LR(1) parse table:

- Construct a dfa (transition diagram) to model the top of the stack whose states represent the current contents of the parsing stack.
- Using the dfa, construct an LR(1) parse table

To Construct the DFA

Idea: The states in the DFA will contain marked productions that indicate what is currently on the top of the stack, and what additional symbols need to be pushed onto the stack in order for a rhs to be on top of the stack, so a reduce operation can occur.

- Add a new production S' → S to the grammar, where S' is the new start symbol.
- place a marker "_" on the rhs of the production to indicate status of parsing process.

  S' → _S

  The items in the rhs to the left of the marker are the items we have parsed (they are on the top of the stack), and the items to the right of the marker are the items we have not seen yet (still need to be pushed onto the stack).

  Example: A → a _ Ab indicates that "a" is on top of the stack and we need to push "A" and "b" on the stack before we can reduce "aAb" to "A".

- Compute the set of productions closure(S' → _S).

  Definition of closure:

  1. closure(A → v_xy) = {A → v_xy} if x is a terminal.
  2. closure(A → v_xy) = {A → v_xy} ∪ (closure(x → _w) for all w (where w is the right hand side of a production in which x is the left hand side)) if x is a variable.

- The closure(S' → _S) is designated as state 0 and marked as "unprocessed".
- Repeat until all states have been processed
  - unproc = any unprocessed state
  - For each x that appears in A→u_xv (where the A production is from the state "unproc") do
    * Add a transition labeled "x" from state "unproc" to a new state with production A→ux_v (Note: If there is more than one production in state "unproc" that has a marker before the x, then one new state is created and all of these productions are placed into the new state, with the marker moved to the right of the x)
    * The set of productions for the new state are: closure(A→ux_v) (Note: If there was more than one production put in from the previous step, then closure is applied to all of those productions).
    * If the new state is identical (has same productions and marker positions) to another state, then combine the two states into one state. Otherwise, mark the new state as "unprocessed"

- Identify final states. Any state that has at least one production with "_" at the end of the rhs is a final state.

Example: Construct DFA

(0) S' $\rightarrow$ S
(1) S $\rightarrow$ aSb
(2) S $\rightarrow$ b

Backtracking through the DFA

Short Version:

Consider aabbb

- Start in state 0.
- Shift "a" and move to state 2.
- Shift "a" and move to state 2.
- Shift "b" and move to state 3.
  Reduce by "S $\rightarrow$ b"
  Pop "b" and Backtrack to state 2.
  Shift "S" and move to state 4.
- Shift "b" and move to state 5.
  Reduce by "S $\rightarrow$ aSb"
  Pop "aSb" and Backtrack to state 2.
  Shift "S" and move to state 4.

- Shift “b” and move to state 5.

  Reduce by “S $\rightarrow$ aSb”

  Pop “aSb” and Backtrack to state 0.

  Shift “S” and move to state 1.

- Accept. aabbb is in the language.

A More detailed explanation of the Backtracking:

A state in the DFA represents what is currently on “top” of the stack. A state is a final state if it represents the fact that a right hand side is on top of the stack.

Consider the string “aabbb”. We will trace the string through the DFA.

Start in state 0, the start state. We have not recognized any part of the string yet.

We recognize the first “a” in the string (shift the “a” onto the stack) and move into state 2. State 2 represents the fact that “aa*” is on top of the stack. In this case, “a” is on top of the stack.

We recognize the second “a” in the string (shift it onto the stack) and remain in state 2. The stack now contains “aa” which is in the form “aa*”.

We recognize the first “b”, shift it onto the stack, and move into state 3. State 3 represents the fact that “aa*b” or “b” is on top of the stack. In this case, “aab” is on the stack (with “b” on top). We now have the right hand side of a production rule on top of the stack. This is why state 3 is a final state! Final states indicate that a reduction is possible. We apply the reduction “S $\rightarrow$ b”. We will pop “b” from the stack and backtrack in the DFA back to state 2, since the current contents on the stack is now “aa” (which state 2 represents). We will push “S” onto the stack and move from state 2 to state 4, since state 4 represents “aa*S”, and “aaS” is now the contents of the stack.

We recognize the second “b” in the string, shift it onto the stack and move into state 5, which represents that the current stack contents are in the form “aa*Sb”, in this case they are “aaSb”. State 5 is a final state, which means that the right hand side of the production “S $\rightarrow$ aSb”is on top of the stack. We can reduce by this production. We will pop “aSb” from the stack, and backtrack in the DFA from state 5 to state 4 to state 2 to state 2. The current contents of the stack is now “a” which is represented by state 2. We push “S” onto the stack and move into state 4. The current stack contents are “aS”.

We recognize the third “b” in the string, shift it onto the stack and move into state 5. Current stack is “aSb”. We reduce, popping “aSb” from the stack and backtrack from state 5 to state 4 to state 2 to state 0. The current contents of the stack are empty. We push “S” onto the stack and move into state 1, which represents that the stack contents are “S”, our goal. The string is accepted.

Note the productions identified in order are:

S $\rightarrow$ b
S $\rightarrow$ aSb
S $\rightarrow$ aSb
S’ $\rightarrow$ S

In reverse order the productions and the corresponding derivation is:

S’ $\rightarrow$ S	S’ $\Rightarrow$ S
S $\rightarrow$ aSb	$\Rightarrow$ aSb
S $\rightarrow$ aSb	$\Rightarrow$ aaSbb
S $\rightarrow$ b	$\Rightarrow$ aabbb

To construct LR(1) table from diagram:

1. If there is an arc from state1 to state2
   (a) arc labeled x is terminal or $
   T[state1, x] = sh state2
   (b) arc labeled X is nonterminal
   T[state1, X] = state2
2. If state1 is a final state with $X \rightarrow w\_$
   For all a in FOLLOW(X), T[state1,a] = reduce by $X \rightarrow w$
   (or T[state1,a] = rN where N is the number of the production $X \rightarrow w$)
3. If state1 is a final state with $S' \rightarrow S\_$
   T[state1,$] = accept
4. All other entries are error

Example: LR(1) Parse Table

$$
\begin{array}{l}
(0)\ S' \rightarrow S \\
(1)\ S \rightarrow aSb \\
(2)\ S \rightarrow b
\end{array}
$$

Here is the LR(1) Parse Table with extra information about the stack contents of each state.

Stack contents	State number	Terminals			Variables
		a	b	$	S
(empty)	0				
	1				
	2				
	3				
	4				
	5				

Actions for entries in LR(1) Parse table T[state,symbol]

Let entry = T[state,symbol].

- If symbol is a terminal or \$
  - If entry is "shift state$i$"
    push lookahead and state$i$ on the stack
  - If entry is "reduce by rule X $\rightarrow$ w"
    pop w and k states (k is the size of w) from the stack. Let state$i$ be the state currently on top of the stack. Push X onto the stack. Push the state$j$ onto the stack, where state$j$ = T[state$i$, X].
  - If entry is "accept"
    Halt. The string is in the language.
  - If entry is "error"
    Halt. The string is not in the language.
- If symbol is nonterminal

  We have just reduced the rhs of a production X $\rightarrow$ w to a symbol. The entry is a state number, call it state$i$. Push T[state$i$, X] on the stack.

Constructing Parse Tables for CFG's with ϵ-rules

A $\rightarrow \epsilon$ written as A $\rightarrow \epsilon\_$

A ϵ-rule is recognized as being reducible right away. Any state that has a ϵ-rule is a final state that can apply the ϵ-rule as a reduction.

It doesn't make sense to push ϵ onto the stack, so there won't be any arcs with ϵ. (Besides, allowing ϵ's in our DFA would turn our DFA into an NFA!). For the rule "A $\rightarrow \epsilon$", we enter it into the table for any lookahead that is in the FOLLOW(A).

Example

$$\begin{array}{l} S \rightarrow ddX \\ X \rightarrow aX \\ X \rightarrow \epsilon \end{array}$$

Add a new start symbol and number the rules:

(0)	S' $\rightarrow$ S
(1)	S $\rightarrow$ ddX
(2)	X $\rightarrow$ aX
(3)	X $\rightarrow$ ϵ

Construct the DFA:

Construct the LR(1) Parse Table

	a	d	$	S	X
0					
1					
2					
3					
4					
5					
6					

Possible Conflicts:

1. Shift/Reduce Conflict

   Example:

   $$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow abcd \end{aligned}$$

   In the DFA:

   $$\begin{aligned} A &\rightarrow ab\_ \\ A &\rightarrow ab\_\ cd \end{aligned}$$

2. Reduce/Reduce Conflict

   Example:

   $$\begin{aligned} A &\rightarrow ab \\ B &\rightarrow ab \end{aligned}$$

   In the DFA:

   $$\begin{aligned} A &\rightarrow ab\_ \\ B &\rightarrow ab\_ \end{aligned}$$

3. Shift/Shift Conflict