CS 406: Bottom-Up Parsing

Stefan D. Bruda

Winter 2016

Bottom-Up Push-Down Automata

- A different way to construct a push-down automaton equivalent to a given grammar = shift-reduce parser:
- Given $G = (N, \Sigma, S, R)$ construct the push-down automaton $M = (\{p, q\}, \Sigma, N|\Sigma, \Delta, s, \{q\})$ with Δ containing exactly all the transitions:

$$\begin{array}{rrl} \text{shift} & \forall\, a \in \Sigma : & ((p, a, \varepsilon), (p, a)) \\ \text{reduce} & \forall\, A ::= \alpha \in R : & ((p, \varepsilon, \alpha^{\mathbb{R}}), (p, A)) \\ \text{done} & & ((p, \varepsilon, S), (q, \varepsilon)) \end{array}$$

Left-to-right traversal of the input + rightmost derivation!

- Just as nondeterministic as the previous construction!

Bottom-Up Push-Down Automata

- A different way to construct a push-down automaton equivalent to a given grammar = shift-reduce parser:
- Given $G = (N, \Sigma, S, R)$ construct the push-down automaton $M = (\{p, q\}, \Sigma, N|\Sigma, \Delta, s, \{q\})$ with Δ containing exactly all the transitions:

$$\begin{array}{rrl} \text{shift} & \forall a \in \Sigma : & ((p, a, \varepsilon), (p, a)) \\ \text{reduce} & \forall A ::= \alpha \in R : & ((p, \varepsilon, \alpha^{\mathbb{R}}), (p, A)) \\ \text{done} & & ((p, \varepsilon, S), (q, \varepsilon)) \end{array}$$

Left-to-right traversal of the input + rightmost derivation!

- Just as nondeterministic as the previous construction!
  - Shift/reduce conflict: when to shift and when to reduce?
    - Establish a precedence relation (lookahead table) $P \subseteq (N|\Sigma) \times \Sigma$
    - If $(\text{stack-top}, \text{input}) \in P$ then we reduce, else we shift

- A different way to construct a push-down automaton equivalent to a given grammar = shift-reduce parser:
- Given $G = (N, \Sigma, S, R)$ construct the push-down automaton $M = (\{p, q\}, \Sigma, N|\Sigma, \Delta, s, \{q\})$ with Δ containing exactly all the transitions:

$$\begin{array}{rrl} \text{shift} & \forall\, a \in \Sigma : & ((p, a, \varepsilon), (p, a)) \\ \text{reduce} & \forall\, A ::= \alpha \in R : & ((p, \varepsilon, \alpha^{\mathbb{R}}), (p, A)) \\ \text{done} & & ((p, \varepsilon, S), (q, \varepsilon)) \end{array}$$

Left-to-right traversal of the input + rightmost derivation!

- Just as nondeterministic as the previous construction!
  - Shift/reduce conflict: when to shift and when to reduce?
    - Establish a precedence relation (lookahead table) $P \subseteq (N|\Sigma) \times \Sigma$
    - If (stack-top, input) $\in P$ then we reduce, else we shift
  - Reduce/reduce conflict: when we reduce, with what rule we reduce?
    - Use the logest rule = greedy (eat up the longest stack top)
  - We thus obtain an LR parser

Example or *LR* Parsing

$$
\begin{array}{lcl}
\langle E \rangle & ::= & \langle E \rangle + \langle T \rangle \\
\langle E \rangle & ::= & \langle T \rangle \\
\langle T \rangle & ::= & \langle T \rangle * \langle F \rangle \\
\langle T \rangle & ::= & \langle F \rangle \\
\langle F \rangle & ::= & (\langle E \rangle) \\
\langle F \rangle & ::= & y
\end{array}
$$

P	(	)	*y*	+	$*$	$
(						
)		✓		✓	✓	✓
y		✓		✓	✓	✓
+						
$*$						
⟨E⟩						
⟨T⟩		✓		✓		✓
⟨F⟩		✓		✓	✓	✓

$$
\begin{array}{lrll}
(& (p, a, \varepsilon), & (p, a) &), a \in \{+. * .(,), y\} \\
(& (p, \varepsilon, \langle T \rangle + \langle E \rangle), & (p, \langle E \rangle) &) \\
(& (p, \varepsilon, \langle T \rangle), & (p, \langle E \rangle) &) \\
(& (p, \varepsilon, \langle F \rangle * \langle T \rangle), & (p, \langle T \rangle) &) \\
(& (p, \varepsilon, \langle F \rangle), & (p, \langle T \rangle) &) \\
(& (p, \varepsilon,)\langle E \rangle(), & (p, \langle F \rangle) &) \\
(& (p, \varepsilon, y), & (p, \langle F \rangle) &) \\
(& (p, \varepsilon, \langle E \rangle) & (q, \varepsilon) &)
\end{array}
$$

	Input	Stack
	y + y * y $	$
shift	+ y * y $	y $
red	+ y * y $	⟨F⟩ $
red	+ y * y $	⟨T⟩ $
red	+ y * y $	⟨E⟩ $
shift	y * y $	+ ⟨E⟩ $
shift	* y $	y + ⟨E⟩ $
red	* y $	⟨F⟩ + ⟨E⟩ $
red	* y $	⟨T⟩ + ⟨E⟩ $
shift	y $	* ⟨T⟩ + ⟨E⟩ $
shift	$	y * ⟨T⟩ + ⟨E⟩ $
red	$	⟨F⟩ * ⟨T⟩ + ⟨E⟩ $
g-red	$	⟨T⟩ + ⟨E⟩ $
g-red	$	⟨E⟩ $
done	$	$

red = reduce (unambiguous)
g-red = greedy reduce (longest rule)

A First *LR* Parsing Algorithm

```
function LRPARSER(G = (N, Σ, S, R), PrecTable:(N|Σ) × Σ):
    PUSH(ADVANCE())                                  (1)
    accepted ← False
    while not accepted do
        if TOP()TOP() = S$ and PEEK() = $ then
            accepted ← True                          (3)
        else
            action ← PrecTable[TOP()][PEEK()]
            if action = shift then
                PUSH(ADVANCE())                      (1)
            else if action = reduce A ::= x_1 ... x_m then
                for i = m down to 1 do
                    POP(x_i)
                PUSH(A)                              (2)
            else
                ERROR("Syntax error")
                accepted ← True
```

$$
\begin{array}{lll}
\Delta & = & \{((p,a,\varepsilon),(p,a)) : a \in \Sigma\} \quad (1) \\
 & | & \{((p,\varepsilon,\alpha^{\mathbb{R}}),(p,A)) : A ::= \alpha \in R\} \quad (2) \\
 & | & \{((p,\varepsilon,S),(q,\varepsilon))\} \quad (3)
\end{array}
$$

- Stack operations: PUSH(), POP(), TOP()
- Operations on the input stream: ADVANCE() (returns the next token and consume it), PEEK() (returns the next token but does not consume it)

A PRACTICAL *LR* PARSING ALGORITHM

```
function LRPARSER(G = (N, Σ, S, R), LRTable:(N|Σ) × Σ):
    PUSH(0)
    accepted ← False
    while not accepted do
        action ← LRTable[TOP()][PEEK()]
        if action = shift s then
            PUSH(s)
            if s is accepting then  accepted ← True
            else  ADVANCE()
        else if action = reduce ⟨A⟩ ::= w then
            POP(|w|)
            PREPEND(⟨A⟩)
        else ERROR("Syntax error")
```

- PREPEND() pushes one symbol at the beginning of the input stream
- Each time $\langle A \rangle ::= w$ is used the prefix w of the current input string is replaced by $\langle A \rangle$
  - Handle = a sequence of symbols that will be next replaced by a reduction
  - The tokens are shifted on the stack until a handle appears
  - When a handle appears, it is reduced

⟨st⟩	::=	⟨S⟩ $	(1)
⟨S⟩	::=	⟨A⟩ ⟨C⟩	(2)
⟨C⟩	::=	*c*	(3)
	|	ε	(4)
⟨A⟩	::=	*a* ⟨B⟩ ⟨C⟩ *d*	(5)
	|	⟨B⟩ ⟨Q⟩	(6)
⟨B⟩	::=	*b* ⟨B⟩	(7)
	|	ε	(8)
⟨Q⟩	::=	*q*	(9)
	|	ε	(10)

State	*a*	*b*	*c*	*d*	*q*	$	⟨st⟩	⟨S⟩	⟨A⟩	⟨B⟩	⟨C⟩	⟨Q⟩
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

LR Parsing Example

Action	Input	Stack
	abbdc\$	0
shift 3	*bbdc*\$	3,0
shift 2	*bdc*\$	2,3,0
shift 2	*dc*\$	2,2,3,0
reduce 8	⟨B⟩*dc*\$	2,2,3,0
shift 13	*dc*\$	13,2,2,3,0
reduce 7	⟨B⟩*dc*\$	2,3,0
shift 13	*dc*\$	13,2,3,0
reduce 7	⟨B⟩*dc*\$	3,0
shift 9	*dc*\$	9,3,0
reduce 4	⟨C⟩*dc*\$	9,3,0
shift 10	*dc*\$	10,9,3,0
shift 12	*c*\$	12,10,9,3,0
reduce 5	⟨A⟩*c*\$	0
shift 1	*c*\$	1,0
shift 11	\$	11,1,0
reduce 3	⟨C⟩\$	1,0
shift 14	\$	14,1,0
reduce 2	⟨S⟩\$	0
shift 4	\$	4,0
shift 8	\$	8,4,0
reduce 1	⟨st⟩\$	0
accept		

⟨st⟩	::=	⟨S⟩ \$	(1)
⟨S⟩	::=	⟨A⟩ ⟨C⟩	(2)
⟨C⟩	::=	*c*	(3)
	|	ε	(4)
⟨A⟩	::=	*a* ⟨B⟩ ⟨C⟩ *d*	(5)
	|	⟨B⟩ ⟨Q⟩	(6)
⟨B⟩	::=	*b* ⟨B⟩	(7)
	|	ε	(8)
⟨Q⟩	::=	*q*	(9)
	|	ε	(10)

- Some shift/reduce conflicts can be resolved by assigning precedence and associativity to tokens

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid id$$

Conflict Resolution Basics

- Some shift/reduce conflicts can be resolved by assigning precedence and associativity to tokens

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid \textit{id}$$

  - Suppose that a *LR* parser reaches the following configuration:

Input	Stack	Prefix
$*$ *id* \$	7,4,1,0	$\langle \text{exp} \rangle + \langle \text{exp} \rangle$

  - If $*$ takes precedence over $+$ then we must shift $*$
  - If $+$ takes precedence over $*$ then we must reduce $\langle \text{exp} \rangle + \langle \text{exp} \rangle$ to $\langle \text{exp} \rangle$

Conflict Resolution Basics

- Some shift/reduce conflicts can be resolved by assigning precedence and associativity to tokens

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid id$$

  - Suppose that a *LR* parser reaches the following configuration:

Input	Stack	Prefix
$*\ id\ \$$	7,4,1,0	$\langle \text{exp} \rangle + \langle \text{exp} \rangle$

  - If $*$ takes precedence over $+$ then we must shift $*$
  - If $+$ takes precedence over $*$ then we must reduce $\langle \text{exp} \rangle + \langle \text{exp} \rangle$ to $\langle \text{exp} \rangle$

  - Suppose that a *LR* parser reaches the following configuration:

Input	Stack	Prefix
$+\ id\ \$$	7,4,1,0	$\langle \text{exp} \rangle + \langle \text{exp} \rangle$

  - If $+$ is left-associative then we reduce, else we shift

- Reduce/reduce conflicts become essentially shift/reduce conflicts in an *LR* parser

  $$\langle \text{stmt} \rangle \; ::= \; \text{if e then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \mid \text{if e then } \langle \text{stmt} \rangle \mid \text{other}$$

  - Suppose that a *LR* parser reaches the following configuration:

Input	Stack	Prefix
else other $	9,4,8,5,3,1,0	if e if e then other

  - If we shift then the else branch will belong to the inner if
  - If we reduce then the else branch will belong to the outer if

Conflict Resolution Basics (cont'd)

- Reduce/reduce conflicts become essentially shift/reduce conflicts in an *LR* parser

$$\langle \text{stmt} \rangle \;::=\; \text{if e then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \mid \text{if e then } \langle \text{stmt} \rangle \mid \text{other}$$

  - Suppose that a *LR* parser reaches the following configuration:

Input	Stack	Prefix
else other $	9,4,8,5,3,1,0	if e if e then other

  - If we shift then the else branch will belong to the inner if
  - If we reduce then the else branch will belong to the outer if
  - Usual strategy is greedy (reduce with the longest rule) → the shift/reduce conflict is resolved in favor of shifting

$LR(k)$ Parsing Definitions and Notations

- An $LR(k)$ parser can look ahead at the next k tokens in the input (plus the top of the stack)
- At any given time it can either reduce the current handler on the stack (reduce) or add to the handler (shift)
  - The decision is based on the symbols already shifted (left context) and the next k lookahead symbols (right context)
  - Driven by an LR algorithm + parse (lookahead) table
    - Every entry in the parse table can accommodate at most one item $\rightarrow$ an LR parser is deterministic
  - Confusing notation: $LR(0)$ and $LR(1)$ parsers both look ahead at the next input token
    - The 0 in $LR(0)$ refers to the lookahead used in constructing the parse table
- $LR(k)$ parsers for $k \geq 2$ have huge parse tables and so are not in wide use

- Notation: $\text{FIRST}_k(w) = \{p \in \Sigma^* : w \Rightarrow^* pu, |p| = k, u \in (N|\Sigma)^*\}$
- A grammar is $LR(k)$ iff it is possible to construct a $LR(k)$ table for that grammar
- Formally, a grammar $(N, \Sigma, \langle S \rangle, R)$ is $LR(k)$ iff the following conditions imply $\alpha\langle A \rangle z = \gamma\langle B \rangle x$:
  1. $\langle S \rangle \overset{R}{\Rightarrow}{}^* \alpha\langle A \rangle z \overset{R}{\Rightarrow} \alpha\beta w$
  2. $\langle S \rangle \overset{R}{\Rightarrow}{}^* \gamma\langle B \rangle x \overset{R}{\Rightarrow} \alpha\beta y$
  3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

$LR(k)$ Grammars

- Notation: $\text{FIRST}_k(w) = \{p \in \Sigma^* : w \Rightarrow^* pu, |p| = k, u \in (N|\Sigma)^*\}$
- A grammar is $LR(k)$ iff it is possible to construct a $LR(k)$ table for that grammar
- Formally, a grammar $(N, \Sigma, \langle S \rangle, R)$ is $LR(k)$ iff the following conditions imply $\alpha\langle A \rangle z = \gamma\langle B \rangle x$:
  1. $\langle S \rangle \overset{R}{\Rightarrow}{}^* \alpha\langle A \rangle z \overset{R}{\Rightarrow} \alpha\beta w$
  2. $\langle S \rangle \overset{R}{\Rightarrow}{}^* \gamma\langle B \rangle x \overset{R}{\Rightarrow} \alpha\beta y$
  3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$
- Suppose we already have $\alpha\beta$ as the current handle and w as remaining input; should we reduce using $\langle A \rangle ::= \beta$?
  - We can decide by looking at $\text{FIRST}_k(w)$
  - In $LR(k)$ parsing we can thus always determine the correct reduction by looking at the left context and the next k tokens in the input

LR(0) Table Construction

- An *LR*(0) table is constructed based on exploring the state space of the parser
  - The state space is finite so the algorithms takes finite time
  - May or may not succeed in constructing a table (with one entry per cell)
  - If the construction does not succeed then inadequate states (which lack sufficient information to have unique entries) are identified
- States represent sets of *LR*(0) items (or just items)
- An item for a Grammar *G* is a rule of *G* with a marker (or bookmark) at some position in the right hand side.
  - The rule $\langle \text{A} \rangle ::= XYZ$ yields the following four items:

    $\langle \text{A} \rangle ::= \bullet XYZ \quad \langle \text{A} \rangle ::= X \bullet YZ \quad \langle \text{A} \rangle ::= XY \bullet Z \quad \langle \text{A} \rangle ::= XYZ\bullet$
  - The rule $\langle \text{A} \rangle ::= \varepsilon$ generates a single item: $\langle \text{A} \rangle ::= \bullet$
  - Intuitively, an item indicates how much of the rule has been seen so far in the input
- Canonical *LR*(0) collections are sets of items and provide the basis for the construction of the *LR*(0) finite automaton

$LR(0)$ Automaton: Essential Algorithms

function Closure(I: set of items) **returns** set of items**:**

```
ans ← I
repeat
    prev ← ans
    foreach rule A ::= α • Bγ do
        foreach rule B ::= w do
            ans ← ans ∪ {B ::= •w}
until ans = prev:
return ans
```

function GoTo(I: set of items, $X \in N|\Sigma$) **returns** set of items**:**

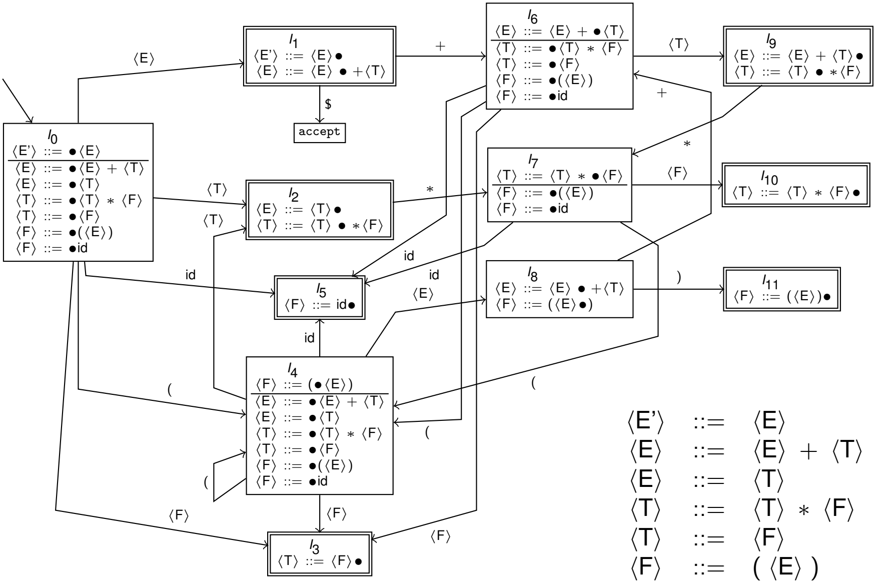
```
ans ← ∅
foreach rule A ::= α • Xγ do
    ans ← ans ∪ {A ::= αX • γ}
return Closure(ans)
```

- $A ::= \alpha \bullet B\gamma$ being in Closure(I) means that at some point during parsing we might see next a substring derivable from $B\gamma$
- If so, then this substring will have a prefix derivable from B
- GoTo is then used to define the transitions of the $LR(0)$ automaton

CONSTRUCTING THE $LR(0)$ AUTOMATON

```
function LR0AUTOMATON(G = (N, Σ, S, R)) returns finite automaton:
    start ← CLOSURE({S ::= •w ∈ R})
    states ← {start}
    transitions ← ∅
    repeat
        grow ← False
        foreach I ∈ states do
            foreach X ∈ N|Σ do
                next ← GOTO(I, X)
                if next ≠ ∅ then
                    transitions ← transitions ∪ {I --X--> next}
                    if next ∉ states then
                        states ← states ∪ {next}
                        grow ← True
    until grow:
    accepting ← {X ∈ states : A ::= u• ∈ X}
    return finite automaton with initial state start, states states,
           transitions transitions, and accepting states accepting
```

- Note in passing that the whole construction is similar to the one that constructs a deterministic finite automaton out of a nondeterministic one



$$
\begin{array}{llll}
\langle E' \rangle & ::= & \langle E \rangle & (1) \\
\langle E \rangle & ::= & \langle E \rangle + \langle T \rangle & (2) \\
\langle E \rangle & ::= & \langle T \rangle & (3) \\
\langle T \rangle & ::= & \langle T \rangle * \langle F \rangle & (4) \\
\langle T \rangle & ::= & \langle F \rangle & (5) \\
\langle F \rangle & ::= & (\langle E \rangle) & (6) \\
\langle F \rangle & ::= & \text{id} & (7)
\end{array}
$$

USING THE $LR(0)$ AUTOMATON: SHIFT ACTIONS

- Suppose that the string γ takes the automaton from state 0 to state *j*
- When the next input symbol is *a* we shift iff state *j* has an outgoing transition labeled *a*
  - Example: the previous $LR(0)$ automaton generates the following table:

State	$+$	$*$	(	)	id	\$	⟨E'⟩	⟨E⟩	⟨T⟩	⟨F⟩
0			4		5			1	2	3
1	6					accept				
2		7								
3										
4			4		5			8	2	
5										
6			4		5				9	3
7			4		5					10
8	6			11						
9		7								
10										
11										

- Suppose that the string γ takes the automaton from state 0 to state *j*
- When the next input symbol is *a* we shift iff state *j* has an outgoing transition labeled *a*

USING THE $LR(0)$ AUTOMATON: REDUCE ACTIONS

- Suppose that the string γ takes the automaton from state 0 to state j
- When the next input symbol is a we shift iff state j has an outgoing transition labeled a
- Otherwise we reduce
  - The items in state j tell us what rules to use for this purpose
  - Reductions can only happen in the final states, which contain reducible items that is, items of form $A ::= w\bullet$
  - For each reducible item we reduce with the corresponding rule for the whole state line in the table
  - Can result in shift/reduce conflicts whenever some cells on a line already contain shift entries
  - Can result in reduce/reduce conflicts whenever some state contains more than one reducible item

LR(0) Parse Table Example

State	+	*	(	)	id	$	⟨E'⟩	⟨E⟩	⟨T⟩	⟨F⟩
0			4		5			1	2	3
1	1, 6	1	1	1	1	1, accept	1	1	1	1
2	3	3, 7	3	3	3	3	3	3	3	3
3	5	5	5	5	5	5	5	5	5	5
4			4		5			8	2	
5	7	7	7	7	7	7	7	7	7	7
6			4		5				9	3
7			4		5					10
8	6			11						
9	2	2, 7	2	2	2	2	2	2	2	2
10	4	4	4	4	4	4	4	4	4	4
11	6	6	6	6	6	6	6	6	6	6

State	+	*	(	)	id	$	⟨E'⟩	⟨E⟩	⟨T⟩	⟨F⟩
0			[4]		[5]			[1]	[2]	[3]
1	1, [6]	1	1	1	1	1, accept	1	1	1	1
2	3	3, [7]	3	3	3	3	3	3	3	3
3	5	5	5	5	5	5	5	5	5	5
4			[4]		[5]			[8]	[2]	
5	7	7	7	7	7	7	7	7	7	7
6			[4]		[5]				[9]	[3]
7			[4]		[5]					[10]
8	[6]			[11]						
9	2	2, [7]	2	2	2	2	2	2	2	2
10	4	4	4	4	4	4	4	4	4	4
11	6	6	6	6	6	6	6	6	6	6

Three shift/reduce conflicts but no reduce/reduce conflict