Shift-Reduce Parsing

Prof. James L. Frankel
Harvard University

Version of 6:50 PM 4-Oct-2016

Bottom-Up Parsing

- Unlike our predictive parsers, a bottom-up parser builds the parse tree from the leaves to the root
- LR(*k*) parser
  - L for left-to-right scanning of the input
  - R for constructing a rightmost derivation in reverse
  - *k* for the number of tokens of lookahead required (*k*=1 is the default)
- LR grammars exist for all programming languages
  - More languages can be described by LR than LL grammars
- Efficient implementations are straightforward
- Syntax errors are tagged as soon as possible

LR Grammar

- (G_1) E → E + T
- (G_2) E → T
- (G_3) T → T * F
- (G_4) T → F
- (G_5) F → (E)
- (G_6) F → id

LR Parsing Table

State	ACTION						GOTO		
	Id	+	*	(	)	$	E	T	F
0	S(5)			S(4)			1	2	3
1		S(6)				Accept			
2		$R(G_2)$	S(7)		$R(G_2)$	$R(G_2)$			
3		$R(G_4)$	$R(G_4)$		$R(G_4)$	$R(G_4)$			
4	S(5)			S(4)			8	2	3
5		$R(G_6)$	$R(G_6)$		$R(G_6)$	$R(G_6)$			
6	S(5)			S(4)				9	3
7	S(5)			S(4)					10
8		S(6)			S(11)				
9		$R(G_1)$	S(7)		$R(G_1)$	$R(G_1)$			
10		$R(G_3)$	$R(G_3)$		$R(G_3)$	$R(G_3)$			
11		$R(G_5)$	$R(G_5)$		$R(G_5)$	$R(G_5)$			

LR-Parsing Algorithm Initial State

- Start with s_0 on the stack, where s_0 is the initial state
- Start with $\omega\$$ in the input buffer

LR-Parsing Algorithm

```
Let a be the first symbol of ω$;
while(1) {
  Let s be the state on top of the stack;
  if(ACTION[s, a] = shift t) {
    push t onto the stack;
    Let a be the next input symbol;
  } else if(ACTION[s, a] = reduce A → β) {
    pop | β | symbols off the stack;
    Let state t now be on top of the stack;
    push GOTO[t, A] onto the stack;
    output the production A → β;
  } else if(ACTION[s, a] = accept)
    break;
  else {
    call error-recovery routine;
  }
}
```

Example of Shift-Reduce Parsing

	Stack	Symbols	Input	Action
1	0		id + id * id $	Shift 5
2	0 5	id	+ id * id $	Reduce G_6 (F → id)
3	0 3	F	+ id * id $	Reduce G_4 (T → F)
4	0 2	T	+ id * id $	Reduce G_2 (E → T)
5	0 1	E	+ id * id $	Shift 6
6	0 1 6	E +	id * id $	Shift 5
7	0 1 6 5	E + id	* id $	Reduce G_6 (F → id)
8	0 1 6 3	E + F	* id $	Reduce G_4 (T → F)
9	0 1 6 9	E + T	* id $	Shift 7
10	0 1 6 9 7	E + T *	id $	Shift 5
11	0 1 6 9 7 5	E + T * id	$	Reduce G_6 (F → id)
12	0 1 6 9 7 10	E + T * F	$	Reduce G_3 (T → T * F)
13	0 1 6 9	E + T	$	Reduce G_1 (E → E + T)
14	0 1	E	$	Accept