Shift-reduce Parsing (Bottom-up Parsing)

2013-10-06 01:10:13 Partha Biswas

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of “reducing” (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is **LR** (scanning from **L**eft to right and using **R**ight-most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc.

Here are the definitions of some commonly used terminologies in this context.

Handles

A “handle” of a string is a substring that matches the RHS of a production and whose reduction to the non-terminal (on the LHS of the production) represents one step along the reverse of a rightmost derivation toward reducing to the start symbol.

If $S \rightarrow^* \alpha Aw \rightarrow^* \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$.

In such a case, it is suffice to say that the substring β is a handle of $\alpha\beta w$, if the position of β and the corresponding production are clear.

Consider the following grammar:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

and a right-most derivation is as follows:

$E \rightarrow \underline{E + E} \rightarrow E+ \underline{E * E} \rightarrow E + E * \underline{id3} \rightarrow E + \underline{id2} * id3 \rightarrow \underline{id1} + id2 * id3$

The id’s are subscripted for notational convenience.

Note that the reduction is in the opposite direction from id1 + id2 * id3 back to E, where the handle at every step is underlined.

Implementation of Shift-Reduce Parsing

A convenient way to implement a shift-reduce parser is to use a **stack** to hold grammar symbols and an **input buffer** to hold the string w to be parsed. The symbol $ is used to mark the bottom of the stack and also the right-end of the input.

Notationally, the top of the stack is identified through a separator symbol |, and the input string to be parsed appears on the right side of |. The stack content appears on the left of |.

For example, an intermediate stage of parsing can be shown as follows:

$id1 | + id2 * id3$ …. (1)

Here “$id1” is in the stack, while the input yet to be seen is “+ id2 * id3$*

In shift-reduce parser, there are two fundamental operations: shift and reduce.

Shift operation: The next input symbol is shifted onto the top of the stack.

After shifting + into the stack, the above state captured in (1) would change into:

$id1 + | id2 * id3$

Reduce operation: Replaces a set of grammar symbols on the top of the stack with the LHS of a production rule.

After reducing id1 using $E \rightarrow id$, the state (1) would change into:

$E | + id2 * id3$

Viable Prefixes

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. It is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.

Item

An item $X \rightarrow \beta.\gamma$ is valid for a viable prefix $\alpha\beta$ if

$S' \rightarrow^* \alpha X w \rightarrow \alpha\beta\gamma w$

is obtainable by a rightmost derivation. After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items.

Notations for Possible Actions in LR Parsing Table

- sn: Shift into state n
- gn: Goto state n
- rk: Reduce by rule k
- a: Accept
- <Nothing> : Error

Understanding Parsing using Examples

In every example, we introduce a new start symbol (S’), and define a new production from this new start symbol to the original start symbol of the grammar.

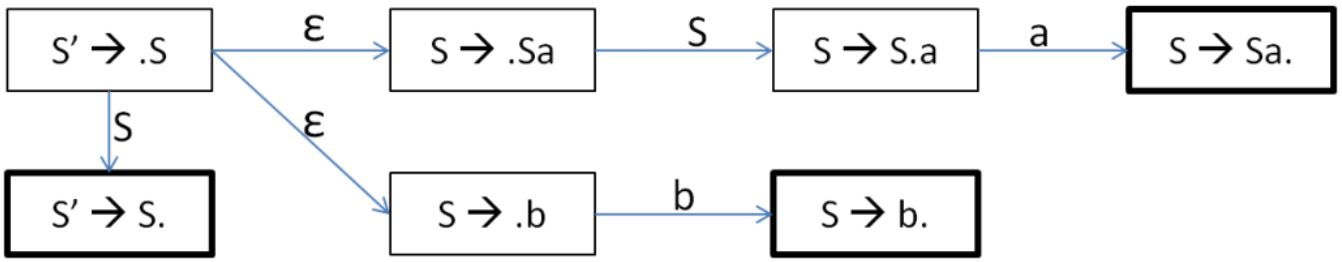
Consider the following grammar (putting an explicit end-marker $ at the end of the first production):

(1) $S' \rightarrow S\$$

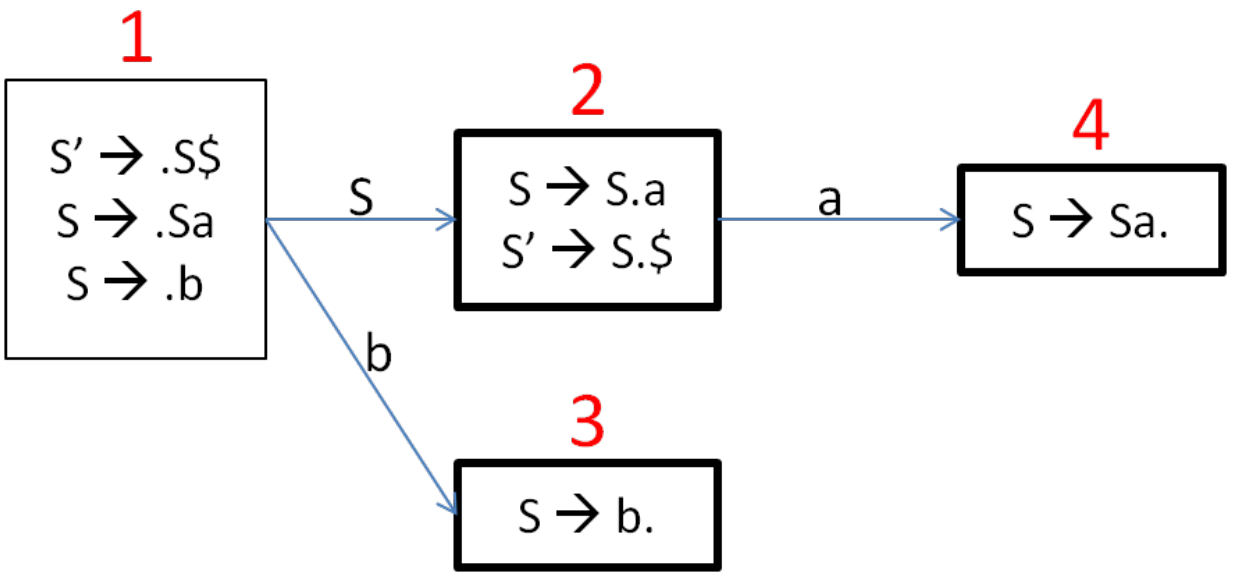
(2) S → Sa

(3) S → b

For this example, the NFA for the stack can be shown as follows:



After doing ε-closure, the resulting DFA is as follows:



The states of DFA are also called “Canonical Collection of Items”. Using the above notation, the ACTION-GOTO table can be shown as follows:

State	a	b	$	S
1		s3		g2
2	s4, r1	r1	r1, a	
3	r3	r3	r3	
4	r2	r2	r2	

Notice that there are two entries for state 2 on input ‘a’. So, this cannot be LR(0) grammar. So, the next enhancement is to look into the FOLLOW set of S’ (which is {$}) and it is different from the shift symbol, this shift-reduce conflict can be resolved easily by reducing on input if it belongs to the FOLLOW set for S’. This enhancement to parsing is called SLR(1) parsing. The corresponding table can be written as follows:

State	a	b	$	S
1		s3		g2
2	s4		a	
3	r3	r3	r3	
4	r2	r2	r2	

Since there is no duplicate entry in the ACTION-GOTO table, this grammar is an SLR(1) grammar.

Now, let's look into progressive parsing of input string baa$.

First, let's consider the case when the stack only contains the grammar symbols. The DFA at every step (shown in each entry of the following table) would start from the bottom of the stack (identified with a '$').

Stack | Input	DFA halt state	Action
$\|baa$	1	Shift
$b\|aa$	3	Reduce
$S\|aa$	2	Shift
$Sa\|a$	4	Reduce
$S\|a$	2	Shift
$Sa\|$	4	Reduce
$S\|$	2	Accept

For the DFA to process from the bottom of the stack at every step is quite wasteful. So, it makes sense to save the grammar symbol along with the current state into the stack. With this change, each stack entry (i..e, the LHS of |) is a pair of grammar symbol and state.

Stack (<entry,state> pair) | Input	Next symbol	Action
$1\|baa$	b	s3
$1b3\|aa$	a	r3
$1S2\|aa$	a	s4
$1S2a4\|a$	a	r2
$1S2\|a$	a	s4
$1S2a4\|$	$	r2
$1S2\|$	$	a

One step further than SLR(1) parsing is LR(1) parsing where the look-ahead is built into each item in the DFA. A slightly less powerful than LR(1), but more space-optimized than LR(1) is LALR(1), which essentially combines the redundant states of LR(1) DFA into one state.

Exercises

Draw the DFA and ACTION-GOTO tables for the following grammars, which require progressively more powerful parsing.

LR(0)

(1) S' → S$

(2) S → (L)

(3) S → x

(4) L → S

(5) L → L , S

SLR(1)

(1) S' → E

(2) E → T + E

(3) E → T

(4) T → int * T

(5) T → int

(6) T → (E)

LR(1) and LALR(1)

(1) S' → S

(2) S → V = E

(3) S → E

(4) E → V

(5) V → x

(6) V → * E