Summary: LR(0) Parsing

1 Basic Concepts

1.1 Derivations

A derivation is a sequence of mappings between the START symbol and the input tokens. At each step, one nonterminal is replaced by the RHS of a production rule which has the nonterminal on it LHS.

The Leftmost Derivation is obtained by starting with the START symbol, and always expanding the *leftmost* nonterminal first. E.g., (the nonterminal in red is the one about to be expanded):

E

E * E

(E) * E

(E + E) * E

(id + E) * E

(id + id) * E

(id + id) * id

The RightMost Derivation is obtained by starting with the START symbol, and always expanding the *rightmost* nonterminal first:

E

E * E

E * id

(E) * id

(E + E) * id

(E + id) * id

(id + id) * id

LR parsing produces a rightmost derivation, but since it works from the bottom up, it *reduces* sequence of tokens on the *left side* first. But when read from the top, the derivation sequence appears to expand rightmost nonterminals first.

1.2 Items & Closure

A **dotted item**, or simply, **item**, is a production rule with a dot in it to indicate how much of the RHS has so far been recognised.

The **closure** of an item is used to see which production rules could be used to extend the current structure. It is calculated as follows:

a) Add the item itself to the closure
b) For any item in the closure, A :- α . β γ where the next symbol after the dot is a nonterminal, add the production rules of that symbol where the dot is before the first item (e.g., for each rule β :- γ, add items: β :- . γ
c) Repeat (b, c) for any new items added under (b).

Example: Given the grammar:

1. E':- E $	Closure of	E':- . E $:	
2. E :- T			
3. E :- E + T		E':- . E $	(by rule a)
3. T :- id		E :- . T	(by rule b)
4. T :- (E)		E :- . E + T	(by rule b)
		T :- . id	(by rule c, b)
		T :- . (E)	(by rule c, b)

2 LR(0) Parsing

For this discussion, assume the following grammar:

1. E :- T
2. E :- E + T
3. T :- id
4. T :- (E)

2.1 Building the DFA

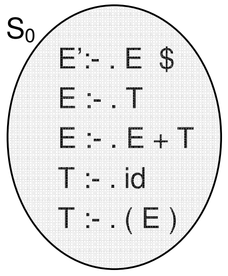
1) Augment the grammar with a new state

    0. E':- E $

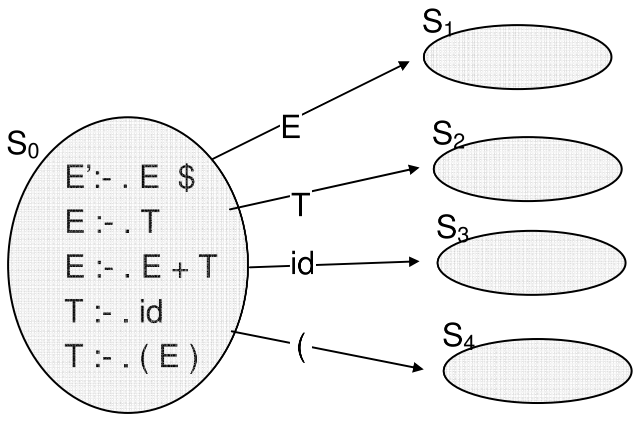
2) Build the node for State $S_{0:}$
    a. Create an item from the rule introduced in (1), with the dot before the first item.
    b. Build the closure for this rule:

        E':- . E $
        E :- . T
        E :- . E + T
        T :- . id
        T :- . (E)

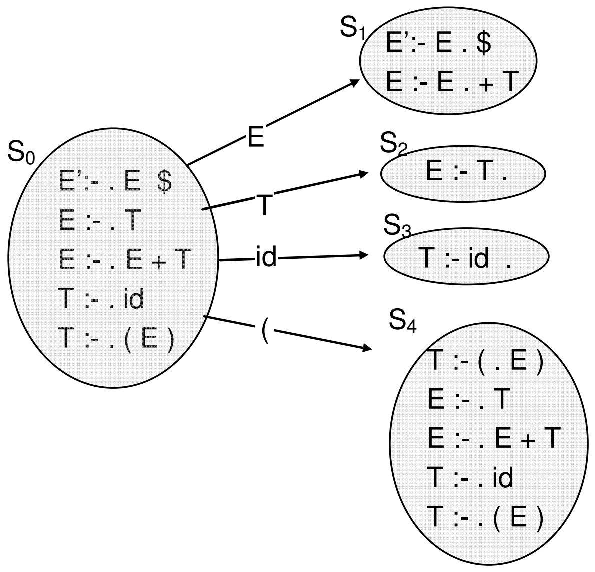
    c. Create a node for S_0, labelled with this closure:



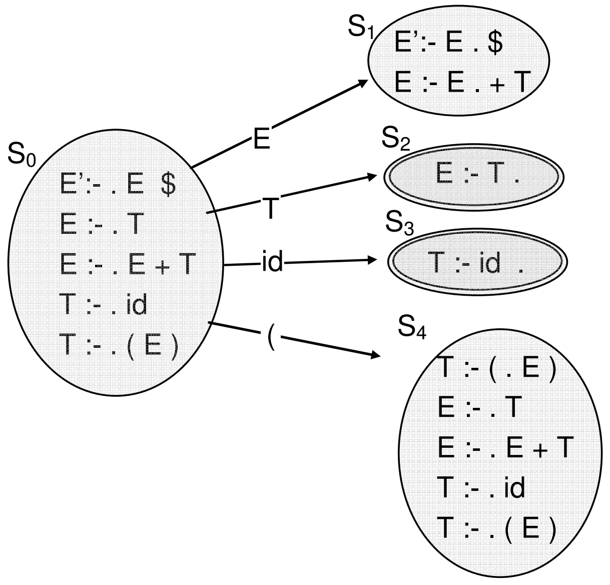
3) For each unique symbol which follows a dot in the closure(S0):
    a. Create a new state representing the recognition of the symbol
    b. Link the original state to the new state with an arc labelled with the recognised symbol
    c. Add a DFA node for this state.



d. The closure of the new states is calculated as follows:
   i) It contains only those items from the previous state where the recognised symbol was the symbol after the dot.
   ii) For each of these items, move the dot AFTER he recognised symbols.
   iii) Where the new next symbol is a nonterminal, add the closure of these items to the closure (note in the state S_4, we started with just T -> (. E) but as the next symbol was E, we closure of this item added 3 other rules:



e. Where the closure includes an item with dot at end, place a double circle around the state (some actions from this state cause a REDUCE)



4) Repeat (3) for each of the new states,

Except: where a new state which has the same closure as an existing one, link to that state instead.

2.2 *Constructing the Parse Table*

1) Create a Goto table with one row for each state, and one column for each terminal and nonterminal in the grammar (except for the augmented grammar symbol E').
2) Add a column for the Action associated with each state
3) For each transition arc between Si and Sj, with label X: enter j in the cell (Si, X) (i.e. enter the end state in the row of the start state under the arc label)
4) For each state with outgoing arcs, enter 'shift' in the action column of the state.
5) For each state with double circle (a reduce state), enter 'reduce' in the state's row of the action column, followed by the number of the rule which is completed in this state.
6) In place of the action reduce for the topmost rule (here E' :- E $), put the action 'accept'.

	Action	**id**	**+**	**(**	**)**	**$**	**E**	**T**
s0	shift	3		4			1	2
s1	shift		6			5		
s2	r1							
s3	r3							

2.3 *Using the Parse Table*

Initially we have:

1. The input vector: a vector of the input tokens, terminated by token $
2. Next token pointer: pointer to the next input token to be processed, initially pointing at the first token.
3. The Stack: initially we place S_0 on the stack.

Recursively:

1. **Apply action**: Apply the action given in the action column for the current state (the topmost state in the stack).

    a. If shift, move the next token onto the top of the stack, and move the pointer to the next token.
    b. If reduce, look up the rule given in the action, and remove n*2 items from the stack (where n is the number of symbols on the RHS of the rule). Then place the LHS of the production on the stack.
    c. If accept, then finish parsing, we have a finished analysis.

2. **Goto State**: The top of the stack now contains a state and a symbol (terminal or nonterminal). In the Goto table, look up the row for the state, and the column symbol.
    a. If there is a number there, put this number on the top of the stack (it is the new state number).
    b. If there is no Goto, then return ERROR (the input cannot be parsed by the grammar).

2.4 Grammar Limitations

The grammar is LR(0) if:

1. there are no shift-reduce conflicts: i.e., no state in the DFA has an outgoing arc and a reduce circle.
2. There are no reduce-reduce conflicts: i.e. no state in the DFA contains two items which have the dot at the end of the RHS.