Building SLR Parse Tables

The easiest technique for generating LR-based parse table is known as SLR (Simple LR). Understanding this technique should provide you with what you need to know to understand how LR parsers work in general; it is also the foundation for the more complex techniques (LR and LALR).

Remember that the idea behind LR parsing is to produce a DFA that defines the handles (string of terminals and non-terminals that indicate a reduction) of the input language. The SLR technique is based on generating sets of LR(0) items that describe the states of the DFA, as well as a transition function that maps between these states.

> **Defn**: An LR(0) item of a grammar G is a production of G with a dot (**.**) at some point on the right side.
>
> **Example:** For production E → E + T, there are four distinct LR(0) items:
>
> E → **.** E + T
>
> E → E **.** + T
>
> E → E + **.** T
>
> E →E + T **.**

A production that directly derives ε, only generates a single LR(0) item. Hence, E → ε has only one associate item : E → **.**

In order to generate these LR(0) item sets, we need to be able to compute closures across them.

> **Defn:** Closure(I) where I is a set of LR(0) items consists of
> - every item in I
> - If A → α **.** **B** β in closure(I), for all productions **B** → γ , add **B** → **.** γ to closure(I) if not already there. Keep applying this rule until no more items can be added.
>
> Initial elements (I) are often referred to as the ***kernel*** elements of closure(I).

Example: we can use the expression grammar:
E → E + T | T
T → T * F | F
F → (E) | id

Closure ({T → T **.** * F}) = {T → T **.** * F}

Closure ({T → T **.** * F, T → T * **.** F}) = {T → T **.*** F, T → T * **.** F, F → **.** (E), F → **.** id}

Closure ({F → (**.** E) }) = {F → (**.** E), E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id }

Generating LR(0) sets are the basis for constructing SLR parsers. To do this generation, we start with an augmented grammar[1]. An augmented grammar is produced by adding a new non-terminal and productions from this new non-terminal to the old start symbol. The purpose of this is to provide a single production that, when reduced, signals the end of parsing. For example, adding a symbol E' and a production E' → E produces an augmented expression grammar.

The key to producing the required sets is the Goto function that maps an item set and a grammar symbol (terminal or non-terminal) to an item set. Remember that items sets represent states in the handle DFA; that means the Goto function gives us the transitions.

> **Defn:** Goto(I,X), where I is a set of items, X is a terminal or non-terminal, is the closure(A → a X **.** b) where A → a **.** X b is in I.

Here are some examples of computing the Goto function for the expression grammar.

Goto({E'→E **.**, E → E **.** + T},+) = closure({E → E + **.** T}) =

{E → E + **.** T, T → **.** T * F, T → **.** F, F → **.** id, F → **.** (E)}

Goto({T → T * **.** F, T → **.** F},F) = closure({T → T * F **.**, T → F **.**}) =

{T → T * F **.**, T → F **.**}

Goto({E'→E **.**, E → E + **.** T},+) = closure({ }) = { }

We are now ready to look at the actual algorithm for generating the DFA.

Algorithm:

- C ={closure({S'→**.** S})}, where S' → S is the production added for augmentation
- Repeat
  - For each item I in C and grammar symbol X such that Goto(I,X) is not empty and not in already an element of C
    - Add Goto(I,X) to C

This is easiest to see by working through the augmented expression grammar example. We start with the computing

C = {closure({E' → **.** E}) } = {E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id}.

This gives us the items for the first state (state 0) of our DFA. Now we need to compute Goto functions for all of the relevant symbols in the set. In this case, we care about the symbols E, T, F, (, and id, since those are the symbols that have a **.** symbol in front of them in some item of the set C.

[1] Augmentation is only required if the grammar does not have a single production that signals the end of the parsing. However, augmentation never changes the language, so it never hurts.

- For symbol E, Goto({E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id },E) = closure ({E' → E **.**, E → E **.** + T}) = { E' → E **.**, E → E **.** + T }. We can call this state 1.
- For symbol T, Goto({E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id },T) = closure ({E → T **.**, T → T **.** * F}) = { E → T **.**, T → T **.** * F}. We can call this state 2.
- For symbol F, Goto({E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id },F) = closure ({T → F **.** }) = { T → F **.** }. We can call this state 3.
- For symbol (, Goto({E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id }, ( ) = closure ({F → (**.** E)}) = { F → (**.** E), E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id }. We can call this state 4.
- For symbol id, Goto({E' → **.** E, E → **.** E + T, E → **.** T, T → **.** T * F, T → **.** F, F → **.** (E), F → **.** id },id) = closure ({F → id **.**}) = {F → id **.**}. We can call this state 5.

I typically write the above information in a table format where each item of each state is annotated with the appropriate state 1-5. The kernel of each state (the items we started with before computing the closure) is in boldface.

State	Item	Goto	State	Item	Goto
0:	**E' → . E**	1	1:	**E' → E .**	
	E → . E + T	1		**E → E . + T**	6
	E → . T	2			
	T → . T * F	2			
	T → . F	3			
	F → . (E)	4			
	F → . id	5			
2:	**E → T .**		3:	**T → F .**	
	T → T . * F	7			
4:	**F → (. E)**	8	5:	**F → id .**	
	E → . E + T	8			
	E → . T	2			
	T → . T * F	2			
	T → . F	3			
	F → . (E)	4			
	F → . id	5			

We continue by computing Goto in each of the newly created states. When the **.** occurs at the end of a production this is going to correspond to a reduction in the parsing algorithm so we won't have a goto value there. We will deal with this situation once we have completed the DFA.

State	Item	Goto	State	Item	Goto
0:	**E' → . E** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	1 1 2 2 3 4 5	1:	**E' → E .** **E → E . + T**	 6
2:	**E → T .** **T → T . * F**	 7	3:	**T → F .**	
4:	**F → (. E)** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	8 8 2 2 3 4 5	5:	**F → id .**	
6:	**E → E + . T** T → . T * F T → . F F → . (E) F → . id		7:	**T → T * . F** F → . (E) F → . id	
8:	**F → (E .)** **E → E . + T**				

Notice in the table that we don't have to create new states when the goto function can use a state that already exists. In state 4, we have transitions to states 2, 3 and 4 because any newly created state (using our rules) would look like these states. However, we can only re-use a state if it is *exactly* the same. For example, we must create a state 8 even though its kernel is similar to that of state 2.

The table on the next page has the complete table for the grammar. Before using this table to find the SLR action/goto tables, we need to compute the follow sets for all of the non-terminals in the grammar and we need to number the productions.

0: E' → E
1: E → E + T
2: E → T
3: T →T * F
4: T → F
5: F → (E)
6: F→ id

Follow(E') = {$}
Follow(E) = {$,),+}
Follow(T) = {$,),+,*}
Follow(F) = {$,),+,*}

State	Item	Goto	State	Item	Goto
0:	**E' → . E** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	1 1 2 2 3 4 5	1:	**E' → E .** **E → E . + T**	 6
2:	**E → T .** **T → T . * F**	 7	3:	**T → F .**	
4:	**F → (. E)** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	8 8 2 2 3 4 5	5:	**F → id .**	
6:	**E → E + . T** T → . T * F T → . F F → . (E) F → . id	9 9 3 4 5	7:	**T → T * . F** F → . (E) F → . id	10 4 5
8:	**F → (E .)** **E → E . + T**	11 6	9:	**E → E + T .** **T → T . * F**	 7
10:	**T → T * F .**		11:	**F → (E) .**	

The action/goto tables are extracted directly from the above information in the following manner. First, each state in the above table will be a row in the action/goto tables. Filling in the entries for row I is done as follows:

- Action[I,c] = shift to state goto(items in state I, c) for terminal c
- Goto[I,c] = goto(items in state I, c) for non-terminal c
- For production number n: A → α in state I where **.** occurs at the end, Action[I,a] = reduce production n, for all elements a in Follow(A)
- For the added production(augmented production) with the **.** at the end in state I, Action[I,$] = accept.

Consider state 6. Action[6,(] = shift 4, Action[6,id] = shift 5, Goto[6,T] = 9, and Goto[6,F] = 3. For state 5, Action[5,*] = Action[5,)] = Action[5,+] = Action[5,$] = reduce 6. The derived action/goto tables are given on the next page.

State	+	*	(	)	id	$		E	T	F
0			s4		s5			1	2	3
1	s6					accept				
2	r2	s7		r2		r2				
3	r4	r4		r4		r4				
4			s4		s5			8	2	3
5	r6	r6		r6		r6				
6			s4		s5				9	3
7			s4		s5					10
8	s6			s11						
9	r1	s7		r1	r1					
10	r3	r3		r3	r3					
11	r5	r5		r5	r5					