Chapter 4

LR Parsing

4.1 Introduction

- Bottom–up parsing technique – problem is finding the handle (right–hand–side) on the stack and replacing it with the left–hand–side.
- Called a shift–reduce parsing method – the major actions done by the parser.
- LR parsing:
  - L – scan the input from left–to–right.
  - R – construct a right–most derivation.
  - k – number of look–ahead symbols needed.
- Advantages:
  - Can be constructed to recognize virtually all programming language constructs for which context–free grammars can be written.
  - Is more general than other types of parsers (can parse everything they can and more) such as LL(k), operator and simple precedence parsers, and is as efficient.
  - Can detect syntactic errors as soon as it is possible to do so on a left–to–right scan of the input.
- Disadvantage:
  - Too much work to implement by hand, must use a parser generator.
- All the LR parser generator needs to do is construct the parse table and we are ready to parse.
- There are many different types of LR parse tables that can be constructed, each with their advantages and disadvantages. We will consider three.
  - SLR(1) – simple LR(1)

    * Easiest to implement – uses LR(0) item sets.
    * May fail to produce a table when other techniques may succeed.
  - LALR(1) – look–ahead LR
    * Of intermediate power between SLR(1) and LR(1) methods.
    * Will work on most programming language grammars.
    * With effort, can be implemented efficiently – uses LR(1) item sets compacted.
  - LR(1)
    * Most powerful.
    * Works on a large class of grammars.
    * Can be very expensive to implement – uses LR(1) item sets.

4.2 LR(k) Grammars

- Let G be some grammar with start symbol S and consider a right–most derivation of a terminal string ω in the grammar:

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \omega$$

- Now consider a typical step in the derivation as follows:

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

  where $A \rightarrow \beta$ is a production used in this step, and $\alpha\beta\gamma$ is one of the α_i or ω itself.

- We say G is LR(k) if, for every such derivation and derivation step, the production $A \rightarrow \beta$ can be inferred by scanning $\alpha\beta$ and (at most) the first k symbols of γ.

- Given that G is LR(k), there are several useful properties that make possible the development of a deterministic bottom–up parser:

  1. The parser will know when to cease scanning a given sentential form $\alpha\beta\gamma$, i.e., it can detect the boundary between β and γ.
  2. The parser is able to identify the handle β.
  3. The parser will be able to uniquely select a production $A \rightarrow \beta$ that corresponds to the handle and to this sentential form. Grammars can be LR(k) and yet have productions $A \rightarrow \beta$, $B \rightarrow \beta$ with the same right–hand side.
  4. The parser will know when to stop.

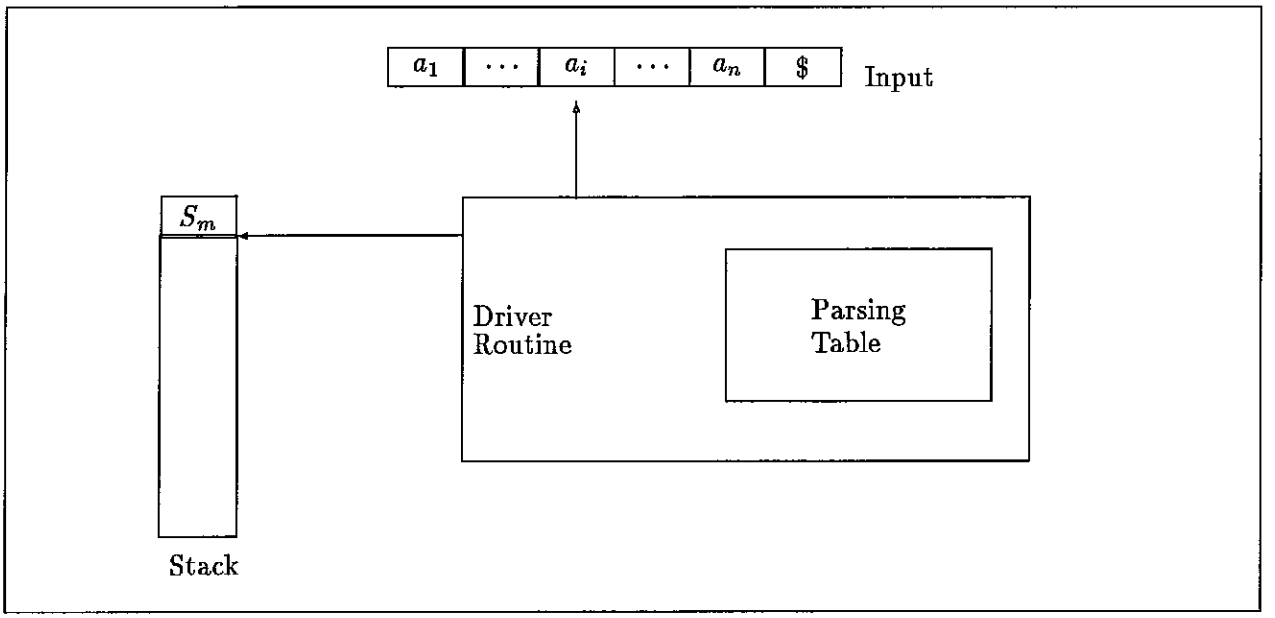


Figure 4.1: Overview of LR Parsing

4.3 Overview of an LR Parser

- Consider Figure 4.1
- The stack contains a string of the form

$$S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m$$

where

  - S_m is the top of the stack.
  - X_i is a grammar symbol.
  - S_i is a state. It essentially summarizes the information contained in the stack and is used to guide the shift–reduce decision.

- In an actual implementation, X_i need not appear on the stack. We will leave it there for a while so you can get a feel for the information contained in the stack and what the states represent.
- The LR parse table actually consists of two tables:
  - Action table
  - Goto table
- The driver routine does the following until an accept or error is obtained:
  - Determine the state on top of the stack and the next input symbol (S_m, a_i).

0. $S' \rightarrow S\$$
1. $S \rightarrow aSbS$
2. $S \rightarrow a$

(a) Grammar

	action			*goto*
	a	b	$	S
0	S2			1
1			acc	
2	S2	R2	R2	3
3		S4		
4	S2			5
5		R1	R1	

(b) Parse Table

Stack	Input	Action
0	aaababa$	shift 2
0a2	aababa$	shift 2
0a2a2	ababa$	shift 2
0a2a2a2	baba$	reduce 2
0a2a2S3	baba$	shift 4
0a2a2S3b4	aba$	shift 2
0a2a2S3b4a2	ba$	reduce 2
0a2a2S3b4S5	ba$	reduce 1
0a2S3	ba$	shift 4
0a2S3b4	a$	shift 2
0a2S3b4a2	$	reduce 2
0a2S3b4S5	$	reduce 1
0S1	$	accept

(c) Parse of *aaababa*

Figure 4.2: Example of parsing

  - Consult the action part of the parse table under the entry (S_m, a_i) and perform the action specified.

- The goto function is used on a reduce operation. The left–hand side of the production being reduced is used with the new top of stack to put a new state on top of the stack.
- These actions will be formalized a little later.

4.3.1 Example

- Consider the grammar in Figure 4.2(a).
- The parse table in Figure 4.2(b) can be derived from the grammar in Figure 4.2(a).
- Now, let us see how we would parse the string *aaababa*. This is shown in Figure 4.2(c).
- See how this compares with the building of the parse tree (show example of this).
- The states 0–5 are generated using techniques we will develop later.
- Once we have generated the parse table, LR parsing is easily done using the following algorithm.
- Action[state, terminal] has values:
  - Shift

Algorithm 4.1 LR Parser Actions

case Action[S_m, a_i] **of**

shift S:

Enter the configuration $(S_0 S_1 \cdots S_m S, a_{i+1} \cdots a_n \$)$.

reduce p :

Assume p = $A \rightarrow \beta$ and $|\beta| = r$ (i.e., β contains r symbols, terminals or nonterminals), enter the following configuration $(S_0 S_1 \cdots S_{m-r}\ S, a_i a_{i+1} \cdots a_n \$)$ where Goto[S_{m-r}, A] = S. In other words, pop r states off the stack exposing state S_{m-r}. Using the state on top of the stack, S_{m-r} , and the left hand side of production p, A, find the state to be pushed on the stack, namely Goto[S_{m-r}, A]. Note that there is no problem in finding the handle. It is those states popped off the stack and replaced by its left hand side.

accept :

Parse is completed. The sequence of productions, if they would have been output will be the right parse.

error :

Call the error recovery routine.

end case

Figure 4.3: Actions of the parser

  - Reduce $A \rightarrow \beta$ (R 1 where 1 is the production number of $A \rightarrow \beta$)
  - Accept
  - Error

- Goto[state, nonterminal] = state

- Suppose we have the following configuration of the parser (note we have **only** states on the stack):

$$(S_0 S_1 S_2 \cdots S_m, a_i a_{i+1} \cdots a_n \$)$$

  The first component of the ordered pair is the contents of the stack and the second component is the unexpended input

- The next move of the parser is determined by reading a_i, the current input symbol, and S_m, the state on top of the stack, and then consulting the parse table entry Action[S_m, a_i]. The configuration resulting after each of the four possible types of moves are shown in Figure 4.3.

- The initial configuration for the parser is as follows:

$$(S_0, a_1 a_2 \cdots a_n \$)$$

  where S_0 is the designated initial state and $a_1 \cdots a_n$ is the string to be parsed. Moves are executed until either an error is detected or the string is accepted.

- The only difference between one LR parser and another is the information in the Action and Goto fields of the parse table, i.e., in the construction of the parse table. The parsing algorithm, given the parse table, is the same for all LR parsing methods.

4.3.2 Another example

- Consider the example in Figure 4.4(a).
- The parse table is shown in Figure 4.4(b).
- The parse for the string $(ab + a) * b$ is shown in Figure 4.4(c).

4.4 LR(0) Items

4.4.1 Introduction

- LR parsing is simple once the parse tables have been constructed. Some questions we will want to answer:
  1. What do the states represent?
  2. How do we figure the states out for a given grammar?
  3. Given the states, how is the parse table constructed?
- LR parsing concerns itself with items, for now LR(0) items. 0 indicates the number of symbols look ahead we will use in constructing the items. An item is a production with a dot. When look ahead is involved, there is more information associated with an item. If you had the production

$$A \rightarrow BaC$$

you could have the following items:

$$\begin{array}{l}[A \rightarrow \cdot BaC\ ] \\ [A \rightarrow B \cdot aC] \\ [A \rightarrow Ba \cdot C] \\ [A \rightarrow BaC \cdot\ ]\end{array}$$

- The production $A \rightarrow \lambda$ (where λ is the empty string) has only one item namely $[A \rightarrow \cdot]$.
- How many items does $A \rightarrow \beta$ have? $(|\beta| + 1)$
- The actions of the LR parser are really:
  - The stack contains, at any time, prefixes of right sentential forms.

0. $E' \rightarrow E\$$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow TF$
4. $T \rightarrow F$
5. $F \rightarrow F*$
6. $F \rightarrow (E)$
7. $F \rightarrow a$
8. $F \rightarrow b$
9. $F \rightarrow \epsilon$

(a) Grammar

	action								*goto*		
	a	b	ϵ	(	)	+	*	$	E	T	F
0	S5	S6	S7	S4					1	2	3
1						S8		acc			
2	S5	S6	S7	S4	R2	R2		R2			9
3	R4	R4	R4	R4	R4	R4	S10	R4			
4	S5	S6	S7	S4					11	2	3
5	R7	R7	R7	R7	R7	R7	R7	R7			
6	R8	R8	R8	R8	R8	R8	R8	R8			
7	R9	R9	R9	R9	R9	R9	R9	R9			
8	S5	S6	S7	S4						12	3
9	R3	R3	R3	R3	R3	R3	S10	R3			
10	R5	R5	R5	R5	R5	R5	R5	R5			
11					S13	S8					
12	S5	S6	S7	S4	R1	R1		R1			9
13	R6	R6	R6	R6	R6	R6	R6	R6			

(b) Parse Table

Stack	Input	Action	Stack	Input	Action
0	(ab+a)*b$	S4	04 11 8 12	)*b$	R1
04	ab+a)*b$	S5	04 11	)*b$	S13
045	b+a)*b$	R7	04 11 13	*b$	R6
043	b+a)*b$	R4	03	*b$	S10
042	b+a)*b$	S6	03 10	b$	R5
0426	+a)*b$	R8	03	b$	R4
0429	+a)*b$	R3	02	b$	S6
042	+a)*b$	R2	026	$	R8
04 11	+a)*b$	S8	029	$	R3
04 11 8	a)*b$	S5	02	$	R2
04 11 85	)*b$	R7	01	$	acc
04 11 83	)*b$	R4			

(c) Parse of $(ab + a) * b$

Figure 4.4: Another example of parsing

Start operation: If S is the start symbol of the grammar, and $S \rightarrow \alpha$ is some production, then item $[S \rightarrow \cdot\ \alpha]$ is associated with the start state.

Completion operation: If $[A \rightarrow \alpha \cdot B\gamma]$, where $B \in V_n$, is an item in some state I, then every item of the form $[B \rightarrow \cdot\beta]$ must be included in state I. This rule is repeated until no more new items can be added to state I. This operation is also called *closure.*

Read operation: Let $[A \rightarrow \alpha \cdot X\gamma]$, where $X \in (V_n \cup V_t)$, be an item associated with some state I. Then $[A \rightarrow \alpha X \cdot \gamma]$ is associated with a state J (possibly the same as I), and a transition from I to J on symbol X exists.

Figure 4.5: Algorithm for item set construction

  - Each time a decision needs to be made concerning the next action to take, the parser runs through a finite state automaton from the bottom of the stack to the top of the stack.
  - The finite state automaton essentially "tells us" what action to take.

- Suppose our grammar is augmented with a new start symbol S'. This is a must in LR parsing. It gives us a unique start state in the finite state automaton.
- Consider the following grammar:

0. $S' \rightarrow S$
1. $S \rightarrow aSbS$
2. $S \rightarrow a$

- A dotted production $A \rightarrow\ \beta_1 \cdot \beta_2$ effectively says: we are looking for a handle $\beta_1\beta_2$ for the production $A \rightarrow \beta_1\beta_2$, and we have seen β_1 thus far.
- So item $S' \rightarrow \cdot S$ says we have seen nothing thus far and we expect a match with S before we can reduce by $S' \rightarrow S$. But then $S \rightarrow aSbS$ and $S \rightarrow a$. So clearly $S \rightarrow \cdot aSbS$ and $S \rightarrow \cdot a$ have to be grouped with item $S' \rightarrow \cdot S$.

4.4.2 Item Set Construction

- The algorithm for item set construction is shown in Figure 4.5.

4.4.3 Construction of Finite State System

1. Give the start state a number, and use the *start operation* to put one item into it. Use the *completion operation* to get more items into this state. Eventually the *completion operation* has to end.

2. Use the *read operation* to start one or more new states, based on the present state. It is possible for this new state to be equivalent to some previous state. If so, these states are merged.

3. Complete the new state started in step 2 by applying the *completion operation.*

4. Repeat steps 2 and 3 until no new states are obtained.

4.4.4 Example

- Consider the grammar shown in Figure 4.6(a).
- The item sets are constructed as shown in Figure 4.6(b) (show the FSA).
- The transition table is created from the item sets in Figure 4.6(b) and is shown in Figure 4.6(c).
- Consider a possible stack:

$$aaSbS$$

  We would be in state 5. State five says we have found the handle and need to reduce

- If instead we had the stack:

$$aaS$$

  We would be in state 3. State three says that if the next input symbol is a b, then we shift into state 4 else it is an error.

- Let's go back and consider the parse as shown in Figure 4.6(d) using the finite state machine shown in Figure 4.6(c).
- Therefore the push down automaton is "keeping track" of what has been done and indicating what is left that needs to be done

4.4.5 Inadequate or Inconsistent States

- Definition:
  - Any state containing both a completed item $[A \rightarrow \alpha\cdot]$ and any other item is said to be *inadequate* or *inconsistent.*
  - Such a state represents a conflict in a parsing decision.
- If the LR states contain no inadequate states, the grammar is said to be LR(0). We are also sure the grammar is unambiguous.
- Resolution of inadequate states (three different resolutions):

0. $S' \rightarrow S$
1. $S \rightarrow aSbS$
2. $S \rightarrow a$

(a) Grammar

0. $[S' \rightarrow \cdot S]$
   $[S \rightarrow \cdot aSbS]$
   $[S \rightarrow \cdot a]$
1. $[S' \rightarrow S \cdot]$
2. $[S \rightarrow a \cdot SbS]$
   $[S \rightarrow a \cdot]$
   $[S \rightarrow \cdot aSbS]$
   $[S \rightarrow \cdot a]$
3. $[S \rightarrow aS \cdot bS]$
4. $[S \rightarrow aSb \cdot S]$
   $[S \rightarrow \cdot aSbS]$
   $[S \rightarrow \cdot a]$
5. $[S \rightarrow aSbS \cdot]$

(b) LR(0) items

	a	b	S
0	2		1
1			
2	2		3
3		4	
4	2		5
5			

(c) Transition table

Stack	Input	Action
0	aaababa$	S2
02	aababa$	S2
022	ababa$	S2
0222	baba$	R2
0223	baba$	S4
02234	aba$	S2
022342	ba$	R2
022345	ba$	R1
023	ba$	S4
0234	a$	S2
02342	$	R2
02345	$	R1
01	$	acc

(d) Parse of *aaababa*$

Figure 4.6: Example of LR(0) items

1. Look ahead may be determined by computing the *Follow* sets for the nonterminal on the left hand side of the production in the inadequate state. This leads to a SLR(1) parser.
2. Discard the LR(0) item construction and use a larger k (i.e., $1, 2, \cdots$). Note that the size of each state set and the number of states increases exponentially with k. This construction is practical only for small k or for a small grammar.
3. A full LR(1) item construction can be compromised in the interest of reducing the number of states and the state–set size by merging certain states. This will lead to a look ahead set for each inadequate state. This leads to a LALR(1) parser.

4.4.6 First and Follow Sets

First Sets

- The set of terminal symbols that can appear at the far left of any parse tree derives from a particular nonterminal is that nonterminal's *First* set.
- Note that ϵ can appear in a *First* set.
- Consider the algorithm shown in Figure 4.7.
- One final note: the *First* notation is a little confusing because you see it used in four ways:

  $First(a)$ (where $a \in V_t$) is a.

  $First(\epsilon)$ is $\{\epsilon\}$.

  $First(A)$ (where $A \in V_n$) is that nonterminal's *First* set, described above.

  $First(\alpha)$ (where $\alpha \in (V_n \cup V_t)^+$) is the *First* set computed using the procedure in Rule 3 above: $First(\alpha)$ always includes the *First* set of the leftmost symbol of α. If that symbol is nullable, then it is the union of the *First* sets of the first two symbols, if both of these symbols are nullable, then it is the union of the first three symbols, and so forth. If all the symbols in α are nullable, then $First(\alpha)$ includes ϵ.

Follow Sets

- A terminal symbol is in a nonterminal's *Follow* set if it can follow that nonterminal in some derivation.
- To find the *Follow* set for a nonterminal, consider the algorithm in Figure 4.8.
- Note that ϵ cannot appear in a *Follow* set.

4.4.7 Construction of SLR(1) Parse Table

- The algorithm for the construction of an SLR(1) parse table is shown in Figure 4.9.

1. *First*(a), where $a \in V_t$, is $\{a\}$. If a is ϵ, then ϵ is put into the *First* set.
2. Given a production of the form $A \rightarrow a\alpha$, where $a \in V_t$, and $\alpha \in (V_n \cup V_t)^*$, a is a member of *First*(A).
3. Given a production of the form $A \rightarrow \alpha X \beta$, where α is a collection of zero or more *nullable* nonterminals, $X \in (V_n \cup V_t)$, and $\beta \in (V_n \cup V_t)^*$, the *First*($A$) includes *First*($X$) $\cup$ *First*(α). A nonterminal is nullable if it can go to ϵ by some derivation. ϵ is always a member of a nullable nonterminal's *First* set.

Figure 4.7: Algorithm for *First* set

1. If S is the start symbol, then \$ is in *Follow*(S).
2. Given a production of the form $A \rightarrow \ldots X \alpha B \ldots$, where α is a collection of zero or more nullable nonterminals and $X \in (V_n \cup V_t)$. *Follow*(X) includes *First*(α) $\cup$ *First*(B).
3. Given a production of the form $A \rightarrow \ldots B\alpha$, were $B \in V_n$, and α is a collection of zero or more nullable nonterminals, everything in *Follow*(A) is also in *Follow*(B).

Figure 4.8: Algorithm for *Follow* set

Algorithm: SLR parse table.

Input: An augmented grammar G'.

Output: The SLR parsing table functions *action* and *goto* for G'.

Method:

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \rightarrow \alpha \cdot a\beta] \in I_i$, *goto*$(I_i, a) = I_j$, and $a \in V_t$, then set *action*$[i, a]$ = *shift* j.
   (b) If $[A \rightarrow \alpha \cdot] \in I_i$, then *action*$[i, a]$ = *reduce* $A \rightarrow \alpha \; \forall \; a \in$ *Follow*(A). Note that the *Follow*(A) is giving the SLR(1) table a small amount of look ahead. A may not be S'.
   (c) If $[S' \rightarrow S \cdot] \in I_i$, then *action*$[i, \$]$ = *accept*.

   If any conflicting actions are generated by the above rules, we say the gammar is not SLR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If *goto*$(I_i, A) = I_j$, then *goto*$[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$.

Figure 4.9: Algorithm for SLR(1) parse table construction

4.4.8 Examples

Example 1

- Consider the augmented grammar for arithmetic expressions shown in Figure 4.10(b).
- The LR(0) items are shown in Figure 4.10(a).
- The transition table is shown in Figure 4.10(c).
- We just need to add the needed reduces and shifts.
- The *Follow* sets are shown in Figure 4.11(a).
- The SLR(1) parse table is shown in Figure 4.11(b).

Example 2

- Consider the grammar shown in Figure 4.12(a).
- Item sets are shown in Figure 4.12(b).
- The *Follow* sets are shown in Figure 4.13(a).
- The SLR(1) parse table is shown in Figure 4.13(b).

4.5 LR(1) Items

4.5.1 Example

- Consider the augmented grammar shown in Figure 4.14(a).
- The LR(0) item sets are shown in Figure 4.14(b).
- The *Follow* sets are shown in Figure 4.14(c).
- SLR(1) table is shown in Figure 4.14(d).
- Consider the previous grammar and look at the following string:

Stack	Input	Action
0	*cb*\$	S4
0 4	*b*\$	

- Look how we got to state 4. We can never have a b following an A (under the assumption we reduce).

Ab is not a *viable prefix*

- A *viable prefix* is so called because it is always possible to add terminal symbols to the end of a viable prefix to obtain a right sentential form.

I_0 : $[E' \to \cdot E]$
$[E \to \cdot E + T]$
$[E \to \cdot T]$
$[T \to \cdot T * F]$
$[T \to \cdot F]$
$[F \to \cdot (E)]$
$[F \to \cdot a]$

I_1 : $[E' \to E \cdot]$
$[E \to E \cdot + T]$

I_2 : $[E \to T \cdot]$
$[T \to T \cdot * F]$

I_3 : $[T \to F \cdot]$

I_4 : $[F \to (\cdot E)]$
$[E \to \cdot E + T]$
$[E \to \cdot T]$
$[T \to \cdot T * F]$
$[T \to \cdot F]$
$[F \to \cdot (E)]$
$[F \to \cdot a]$

I_5 : $[F \to a \cdot]$

I_6 : $[E \to E + \cdot T]$
$[T \to \cdot T * F]$
$[T \to \cdot F]$
$[F \to \cdot (E)]$
$[F \to \cdot a]$

I_7 : $[T \to T * \cdot F]$
$[F \to \cdot (E)]$
$[F \to \cdot a]$

I_8 : $[F \to (E \cdot)]$
$[E \to E \cdot + T]$

I_9 : $[E \to E + T \cdot]$
$[T \to T \cdot * F]$

I_{10} : $[T \to T * F \cdot]$

I_{11} : $[F \to (E) \cdot]$

(a) LR(0) items

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to a$

(b) Grammar

	a	+	*	(	)	E	T	F
I_0	I_5			I_4		I_1	I_2	I_3
I_1		I_6						
I_2			I_7					
I_3								
I_4	I_5			I_4		I_8	I_2	I_3
I_5								
I_6	I_5			I_4			I_9	I_3
I_7	I_5			I_4				I_{10}
I_8		I_6			I_{11}			
I_9			I_7					
I_{10}								
I_{11}								

(c) Transition table

Figure 4.10: Example of LR(0) items

$Follow(E) = \{+,), \$\}$
$Follow(T) = \{+,), \$, *\}$
$Follow(F) = \{+,), \$, *\}$

(a) *Follow* set

	action						*goto*		
	a	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

(b) Parse table

Figure 4.11: Example of parse table construction

4.5.2 Construction of LR(1) Parse Table

- In the SLR(1) method, state i calls for reduction by $A \rightarrow \alpha$ if set of items I_i contains item $[A \rightarrow \alpha\cdot]$ and $a \in Follow(A)$. In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack, is such the βA cannot be followed by a in a right sentential form. Thus the reduction by $A \rightarrow \alpha$ would be invalid on input a.

- It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \rightarrow \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbol can follow a handle α for which there is a possible reduction to A.

- The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of the item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or \$. This is called an LR(1) item. The 1 refers to the length of the second component (called the look ahead of the item).

- The look ahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where $\beta \neq \lambda$, but an item of the form $[A \rightarrow \alpha\cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.

- Construction of the LR(1) items is shown in Figure 4.15.

0. $E' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow TF$
4. $T \rightarrow F$
5. $F \rightarrow F*$
6. $F \rightarrow (E)$
7. $F \rightarrow a$
8. $F \rightarrow b$
9. $F \rightarrow e$

(a) Grammar

I_0 :
$[E' \rightarrow \cdot E]$
$[E \rightarrow \cdot E + T]$
$[E \rightarrow \cdot T]$
$[T \rightarrow \cdot TF]$
$[T \rightarrow \cdot F]$
$[F \rightarrow \cdot F*]$
$[F \rightarrow \cdot (E)]$
$[F \rightarrow \cdot a]$
$[F \rightarrow \cdot b]$
$[F \rightarrow \cdot e]$

I_1 :
$[E' \rightarrow E\cdot]$
$[E \rightarrow E \cdot + T]$

I_2 :
$[E \rightarrow T\cdot]$
$[T \rightarrow T \cdot F]$
$[F \rightarrow \cdot F*]$
$[F \rightarrow \cdot (E)]$
$[F \rightarrow \cdot a]$
$[F \rightarrow \cdot b]$
$[F \rightarrow \cdot e]$

I_3 :
$[T \rightarrow F\cdot]$
$[F \rightarrow F \cdot *]$

I_4 :
$[F \rightarrow (\cdot E)]$
$[E \rightarrow \cdot E + T]$
$[E \rightarrow \cdot T]$
$[T \rightarrow \cdot TF]$
$[T \rightarrow \cdot F]$
$[F \rightarrow \cdot F*]$
$[F \rightarrow \cdot (E)]$
$[F \rightarrow \cdot a]$
$[F \rightarrow \cdot b]$
$[F \rightarrow \cdot e]$

I_5 :
$[F \rightarrow a\cdot]$

I_6 :
$[F \rightarrow b\cdot]$

I_7 :
$[F \rightarrow e\cdot]$

I_8 :
$[E \rightarrow E + \cdot T]$
$[T \rightarrow \cdot TF]$
$[T \rightarrow \cdot F]$
$[F \rightarrow \cdot F*]$
$[F \rightarrow \cdot (E)]$
$[F \rightarrow \cdot a]$
$[F \rightarrow \cdot b]$
$[F \rightarrow \cdot e]$

I_9 :
$[T \rightarrow TF\cdot]$
$[F \rightarrow F \cdot *]$

I_{10} :
$[F \rightarrow F * \cdot]$

I_{11} :
$[F \rightarrow (E\cdot)]$
$[E \rightarrow E \cdot + T]$

I_{12} :
$[E \rightarrow E + T\cdot]$
$[F \rightarrow T \cdot F]$
$[F \rightarrow \cdot F*]$
$[F \rightarrow \cdot (E)]$
$[F \rightarrow \cdot a]$
$[F \rightarrow \cdot b]$
$[F \rightarrow \cdot e]$

I_{13} :
$[F \rightarrow (E)\cdot]$

(b) LR(0) items

Figure 4.12: Grammar and LR(0) items for Example 2

$$Follow(E) = \{+,), \$\}$$
$$Follow(T) = \{+,), \$, (, a, b, e\}$$
$$Follow(F) = \{+,), \$, *, (, a, b, e\}$$

(a) *Follow* sets

	action								*goto*		
	a	b	e	(	)	+	*	$	E	T	F
0	S5	S6	S7	S4					1	2	3
1						S8		acc			
2	S5	S6	S7	S4	R2	R2		R2			9
3	R4	R4	R4	R4	R4	R4	S10	R4			
4	S5	S6	S7	S4					11	2	3
5	R7	R7	R7	R7	R7	R7	R7	R7			
6	R8	R8	R8	R8	R8	R8	R8	R8			
7	R9	R9	R9	R9	R9	R9	R9	R9			
8	S5	S6	S7	S4						12	3
9	R3	R3	R3	R3	R3	R3	S	R3			
10	R5	R5	R5	R5	R5	R5	R5	R5			
11					S13	S8					
12	S5	S6	S7	S4	R1	R1		R1			9
13	R6	R6	R6	R6	R6	R6	R6	R6			

(b) Parse table

Figure 4.13: *Follow* sets and parse table for Example 2

0. $S' \rightarrow S$
1. $S \rightarrow Aa$
2. $S \rightarrow dAb$
3. $S \rightarrow dca$
4. $S \rightarrow cb$
5. $A \rightarrow c$

(a) Grammar

0. $[S' \rightarrow \cdot S]$
   $[S \rightarrow \cdot Aa]$
   $[S \rightarrow \cdot dAb]$
   $[S \rightarrow \cdot dca]$
   $[S \rightarrow \cdot cb]$
   $[A \rightarrow \cdot c]$
1. $[S' \rightarrow S \cdot]$
2. $[S \rightarrow A \cdot a]$
3. $[S \rightarrow d \cdot Ab]$
   $[S \rightarrow d \cdot ca]$
   $[A \rightarrow \cdot c]$
4. $[S \rightarrow c \cdot b]$
   $[A \rightarrow c \cdot]$
5. $[S \rightarrow Aa \cdot]$
6. $[S \rightarrow dA \cdot b]$
7. $[S \rightarrow dc \cdot a]$
   $[A \rightarrow c \cdot]$
8. $[S \rightarrow cb \cdot]$
9. $[S \rightarrow dAb \cdot]$
10. $[S \rightarrow dca \cdot]$

(b) LR(0) items

$Follow(S) = \{\$\}$
$Follow(A) = \{a, b\}$

(c) *Follow* sets

	action					*goto*	
	a	b	c	d	\$	S	A
0			S4	S3		1	2
1					acc		
2	S5						
3			S7				6
4	R5	S8/R5					
5					R1		
6		S9					
7	R5/S10	R5					
8					R4		
9					R2		
10					R3		

(d) Parse table

Figure 4.14: Example of conflicts

1. *Initial state construction*
   (a) If $S \rightarrow \alpha$ is a production, then add $[S \rightarrow \cdot\alpha, \$]$ to the initial state. S is the start symbol.
   (b) If $[A \rightarrow \cdot B\alpha, \omega]$ is in the initial state, and $B \rightarrow \beta$ is a production, then add all the items of the form $[B \rightarrow \cdot\beta, z]$ to the initial state, where $z \in \mathit{First}(\alpha\omega)$. Repeat this step until no more items can be added to the initial state.
2. *Starting a new state*
   Let $[A \rightarrow \alpha \cdot X\gamma, \omega]$ be an item in some state I, where $X \in (V_t \cup V_n)$. We locate every item in state I such the X follows the "$\cdot$", then start a new state J with these items, but move the mark past X. The set of items so used to start state J is called the *core set* of J. It may be equivalent to some other core set, and may be merged.
3. *Closing a new state*
   For every item of the form $[A \rightarrow \alpha \cdot B\gamma, \omega]$ in a state J, where $B \in V_n$, and for every production $B \rightarrow \beta$, we add to J all the items of the form $[B \rightarrow \cdot\beta, u]$, where $u \in \mathit{First}(\gamma w)$. This step is repeated until no more items can be added to J.
4. Steps 2 and 3 are repeated alternately, in that order, until no more new states can be constructed or closed.

Figure 4.15: Construction of LR(1) items

4.5.3 An Example

- Consider the following grammar shown in Figure 4.16(a).
- The LR(1) items are shown in Figure 4.16(b).
- LR(1) parse table is shown in Figure 4.16(c)
- Typically, this will increase the number of states greatly. For example, a grammar like Pascal would have several hundred states for SLR (and LALR) but typically would have several thousand states in a LR(1) table.

4.5.4 Another example

- Consider the grammar shown in Figure 4.17(a).
- The LR(1) items are shown in Figure 4.17(b).
- The canonical LR(1) table is shown in Figure 4.17(c).

4.5.5 Constructing LALR Parsing Tables

- The parse tables constructed for the canonical LR parser can be too large, therefore we would like to cut the size down but not lose too much flexibility.

0. $S' \rightarrow S$
1. $S \rightarrow Aa$
2. $S \rightarrow dAb$
3. $S \rightarrow dca$
4. $S \rightarrow cb$
5. $A \rightarrow c$

(a) Grammar

0. $[S' \rightarrow \cdot S, \$]$
   $[S \rightarrow \cdot Aa, \$]$
   $[S \rightarrow \cdot dAb, \$]$
   $[S \rightarrow \cdot dca, \$]$
   $[S \rightarrow \cdot cb, \$]$
   $[A \rightarrow \cdot c, a]$

1. $[S' \rightarrow S\cdot, \$]$

2. $[S \rightarrow A \cdot a, \$]$

3. $[S \rightarrow d \cdot Ab, \$]$
   $[S \rightarrow d \cdot ca, \$]$
   $[A \rightarrow \cdot c, b]$

4. $[S \rightarrow c \cdot b, \$]$
   $[A \rightarrow c\cdot, a]$

5. $[S \rightarrow Aa\cdot, \$]$

6. $[S \rightarrow dA \cdot b, \$]$

7. $[S \rightarrow dc \cdot a, \$]$
   $[A \rightarrow c\cdot, b]$

8. $[S \rightarrow cb\cdot, \$]$

9. $[S \rightarrow dAb\cdot, \$]$

10. $[S \rightarrow dca\cdot, \$]$

(b) LR(1) items

	action					*goto*	
	a	*b*	*c*	*d*	$	*S*	*A*
0			S4	S3		1	2
1					acc		
2	S5						
3			S7				6
4	R5	S8					
5					R1		
6		S9					
7	S10	R5					
8					R4		
9					R2		
10					R3		

(c) Parse table

Figure 4.16: LR(1) parse table

0. $S' \rightarrow S$
1. $S \rightarrow CC$
2. $C \rightarrow eC$
3. $C \rightarrow d$

(a) Grammar

0. $[S' \rightarrow \cdot S, \$]$
   $[S \rightarrow \cdot CC, \$]$
   $[C \rightarrow \cdot eC, e/d]$
   $[C \rightarrow \cdot d, e/d]$

1. $[S' \rightarrow S\cdot, \$]$

2. $[S \rightarrow C \cdot C, \$]$
   $[C \rightarrow \cdot eC, \$]$
   $[C \rightarrow \cdot d, \$]$

3. $[C \rightarrow e \cdot C, e/d]$
   $[C \rightarrow \cdot eC, e/d]$
   $[C \rightarrow \cdot d, e/d]$

4. $[C \rightarrow d\cdot, e/d]$

5. $[S \rightarrow CC\cdot, \$]$

6. $[C \rightarrow e \cdot C, \$]$
   $[C \rightarrow \cdot eC, \$]$
   $[C \rightarrow \cdot d, \$]$

7. $[C \rightarrow d\cdot, \$]$

8. $[C \rightarrow eC\cdot, e/d]$

9. $[C \rightarrow eC\cdot, \$]$

(b) LR(1) items

	action			*goto*	
	e	d	\$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

(c) Parse table

Figure 4.17: Another example of an LR(1) parse table

Algorithm: LALR table construction (there are other techniques which consume less space)

Input: A grammar G augmented by production $S' \rightarrow S$.

Output: The LALR parsing tables Action and Goto

Method:

1. Construct $C = \{I_0, I_1, ..., I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the sets of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, J_1, ..., J_m\}$ be the resulting sets of LR(1) items. The parsing actions are created as before.
4. The Goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, i.e., $J = I_1 \cup I_2 \cup ... \cup I_m$, then the cores of Goto(I_1, X), Goto(I_2, X), ..., Goto(I_k, X) are the same. Goto(J, X) = $\bigcup_{i=1}^{k}$ Goto(I_i, X).

Figure 4.18: LALR(1) parse table construction

- Look at the item sets for states 4 and 7. Each of these states has only items with the first component $C \rightarrow d\cdot$, but the look ahead symbol is different.
- How are states 4 and 7 used. Language recognizes strings c^*dc^*d. State 4 is where the first d is recognized and state 7 is where the second d is recognized. If only one d is in the string, an error is recognized in state 4. If some d's follow the second d, an error is recognized in state 7.
- Suppose we merge states 4 and 7. What effect would that have on the parser? It behaves essentially like the original, although it might reduce d to C in circumstances where the original would declare error, e.g., *ccd* or *cdcdc*. The error will eventually be caught; in fact, it will be caught before any more input symbols are shifted.
- We look to merge states of LR(1) items with the same core. Thus we would merge states 4 and 7, 3 and 6, and 8 and 9. The Goto's of these states may be also be merged.
- Consider the algorithm shown in Figure 4.18 that shows how LALR parse tables are constructed.

4.5.6 Previous Example

- Merged states are shown in Figure 4.19(a)
- New LALR(1) table is shown in Figure 4.19(b).
- Consider the parse of the string *ccd*\$ as shown in Figure 4.19(c) and 4.19(d).
- We recognize the error later, went through more work, but we did not consume any more input

36. $[C \rightarrow e \cdot C, e/d/\$]$
$[C \rightarrow \cdot eC, e/d/\$]$
$[C \rightarrow \cdot d, e/d/\$]$

47. $[C \rightarrow d\cdot, e/d/\$]$

89. $[C \rightarrow eC\cdot, e/d/\$]$

(a) Merged states

	action			*goto*	
	e	*d*	$	*S*	*C*
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

(b) LALR(1) parse table

Canonical LR Stack	Input	Action
0	*eed*$	S3
0 3	*ed*$	S3
0 3 3	*d*$	S4
0 3 3 4	$	error

(c) Parse with LR(1) parse table

LALR Stack	Input	Action
0	*eed*$	S36
0 36	*ed*$	S36
0 36 36	*d*$	S47
0 36 36 47	$	R3
0 36 36 89	$	R2
0 36 89	$	R2
0 2	$	error

(d) Parse with LALR(1) parse table

Figure 4.19: An example of an LALR(1) parse table

4.6 Ambiguous Grammars

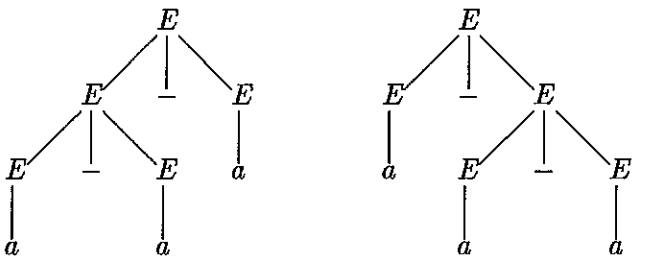
- Every ambiguous grammar fails to be LR.
- Consider the grammar in Figure 4.20(a).
- If we parse the string $a - a - a$, we get the parse trees shown in Figure 4.20(b).
- The first tree is the one preferred. Why? Left to right evaluation of operands. Let us construct an LR parser which prefers such a parse. The LR(0) items are shown in Figure 4.20(c).
- The *Follow* set is shown in Figure 4.20(d).
- The SLR(1) parse table is shown in Figure 4.20(e).
- We have a conflict in state four on a minus sign. The conflict occurs because the grammar is ambiguous. How can the conflict be resolved. In this case, we want to get the first parse tree as shown before. If we choose to reduce rather than shift, we obtain the proper tree.
- Why would you ever want to use an ambiguous grammar such as

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

  Consider the grammar that enforces precedence rules and left associativity. Much of the time of the parser is spent reducing productions of the form $E \rightarrow T$ or $T \rightarrow F$. The parser for the above grammar will not waste time reducing by single productions.
- Tools such as yacc allow you to specify precedence and associativity in an ambiguous grammar. This sometimes allows the parser generator to resolve the ambiguity.

1. $E \rightarrow E - E$
2. $E \rightarrow a$

(a) Grammar



(b) Parse trees

0. $[E' \rightarrow \cdot E]$
   $[E \rightarrow \cdot E - E]$
   $[E \rightarrow \cdot a]$

1. $[E' \rightarrow E\cdot]$
   $[E \rightarrow E \cdot - E]$

2. $[E \rightarrow a\cdot]$

3. $[E \rightarrow E - \cdot E]$
   $[E \rightarrow \cdot E - E]$
   $[E \rightarrow \cdot a]$

4. $[E \rightarrow E - E\cdot]$
   $[E \rightarrow E \cdot - E]$

(c) LR(0) items

$Follow(E) = \{\$, -\}$

(d) *Follow* sets

	action			*goto*
	a	–	$	E
0	S2			4
1		S3	acc	
2		R2	R2	
3	S2			4
4		S3/R1	R1	

(e) SLR(1) parse table

Figure 4.20: Ambiguous grammars