COMPILER CONSTRUCTION (Items sets & LR(0) parser) **Fall 2019**

Lecture 19: Aug. 29,30, 2 Sep. 2019

Instructor: K.R. Chowdhary *: Professor of CS*

19.1 Simple LR Parsing

The Simple LR parsing is also called as SLR parsing. The most common type of bottom-up parser is based on the concept called $LR(k)$ parsing. The "L" stands for left-to-right scanning of the input, and the "R" stands for construction of rightmost derivation in reverse, and the k for number of lookahead symbols to make the parsing decisions. The cases of $k = 0$ and $k = 1$ are of practical interest. However, when k is dropped, the k is assumed as 1.

The LR parser are table driven, much like the LL parsers, we have already discussed. There are number of advantages of LR parsing:

- They can be used to recognize virtually all programming language constructs for which CFGs can be written. Non-LR context-free grammars exist, but they are avoided for typical programming constructs.
- The LR-parsing is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other.
- An LR parser can detect syntactic errors as soon as it is possible to do so on a left to right scan of the input.
- The LR grammars are proper superset of the grammars of predictive of LL methods. The requirement of LR(k) is far more stringent than LL(k) grammar. Hence, LR grammars can describe more languages than the LL grammars.

The typical drawback of the LR parsers is that they need too much work to construct an LR parser by hand for a typical programming-language grammar. So a specialised tool is needed to construct such parsers. Fortunately there are many tools are available, e.g., YACC. Such generators take CFG and automatically generate a parser for the grammar. If the grammar contains ambiguous constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

LR(0) is the simplest technique in the LR family. Although that makes it the easiest to learn, these parsers are too weak to be of practical use for anything but a very limited set of grammars. The fundamental limitation of LR(0) is the zero, meaning no lookahead tokens are used. It is a stifling constraint to have to make decisions using only what has already

been read, without even glancing at what comes next in the input. If we could peek at the next token and use that as part of the decision making, we will find that it allows for a much larger class of grammars to be parsed.

19.2 L(0) Automaton and Item sets

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents \$ T and next input + in Table **??** (page **??**), how does the parser know that T on the top of the stack is a handle, so that appropriate action is to reduce T to E ?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. The states represent set of "items". An $LR(0)$ item (or *item* for short) of a grammar G has a dot at some position of the body. Thus, production $A \to XYZ$ yields four items:

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

The production $A \to \varepsilon$ generates only one item $A \to .$ Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \to .XYZ$ indicates that we hope to see a string derivable from XYZ on the input. Item $A \to X.YZ$ indicates that we have just seen on input a string and that we hope next to see a string derivable from YZ. Item $A \to XYZ.$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

19.2.1 Representing Item Sets

A parser generator that produces a bottom-up parser may need to represent items and sets of items conveniently. Note that an item can be represented by a pair of integers, the first of which is the serial number of a productions of the grammar, and the second is position of the dot[1]. Sets of items can be represented by a list of these pairs. However, as we shall see, the necessary sets of items often include "closure" items, where the dot is at the beginning of the body. These can always be reconstructed from the other items in the set, and we do not have to include them in the list.

One collection of sets of $LR(0)$ items, called *canonical* $LR(0)$ collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such automation is called $LR(0)$ *automaton*. Each state of LR(0) automaton represents a set of item in the canonical LR(0) collection. The automaton for the expression grammar is shown in Fig. 19.1.

We will consider the above automaton for running example for canonical LR(0) collection. To construct the canonical LR(0) collection for a grammar, we define an augmented grammar

[1]For example, $T \to T * .F$, can be represented by (4,3)

Table 19.1: Expression Grammar

Rule No.	Rule
1.	$E \to E + T$
2.	$E \to T$
3.	$T \to T \times F$
4.	$T \to F$
5.	$F \to (E)$
6.	$F \to id$

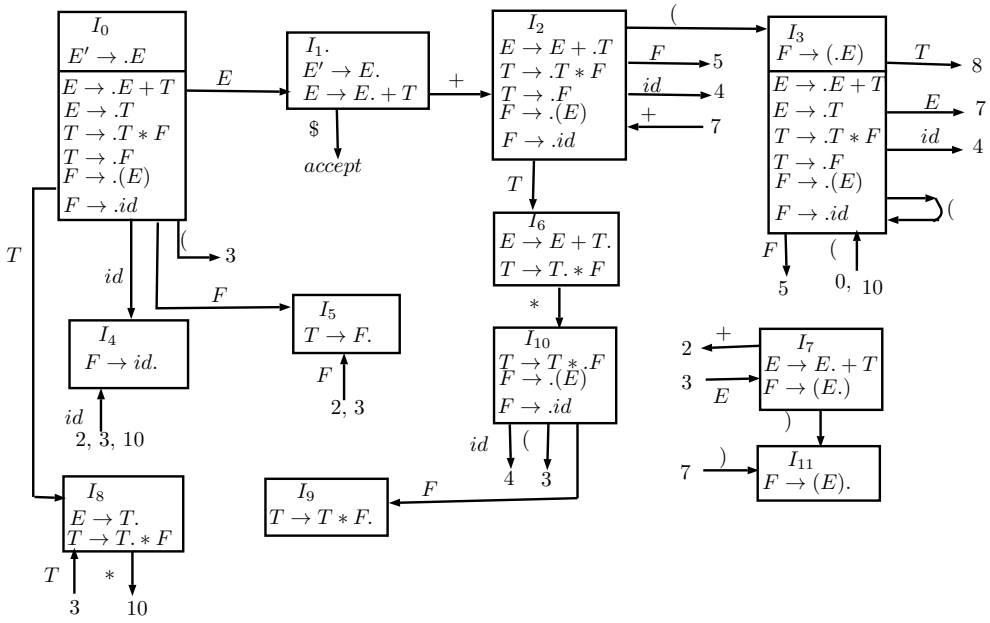

Figure 19.1: LR(0) automaton for expression grammar given in Table 19.1

and two functions CLOSURE and GOTO. If G is a grammar with S as start symbol, then G' – the augmented grammar for G, has a new start symbol S', and a new production $S' \to S$. The purpose of this new starting symbol is to indicate to the parser when it should stop the parsing and announce acceptance of the input. The acceptance occurs only when parser is about to reduce by $S' \to S$. This is obvious because in $E \to E + T$, when we have finally reduced to E, but that is both start symbol and a symbol in the body of a production.

Closure of Item Sets If I is set of items for grammar G, then CLOSURE(I) is the set of items constructed from I by two rules:

1. Initially, add every item I to CLOSURE(I), which we may call as I_0;
2. IF $A \to \alpha.B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to CLOSURE(I) if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Note that, Fig. 19.1 is CLOSURE(I) if it includes all transitions, and all required items in each item-set.

Intuitively, $A \to \alpha.B\beta$ in CLOSURE(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. The substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B-productions. We therefore, add items for all the B-productions; that is, if $B \to \gamma$ is a production, we also include $B \to .\gamma$ in CLOSURE(I).

Example 19.1 *Construct an LR(0) automaton for an augmented grammar.*

Consider the augmented grammar given in Table 19.2.

Table 19.2: Augmented Grammar

Rule No.	Rule
(1)	$E' \to E$
(2)	$E \to E + T$
(3)	$E \to T$
(4)	$T \to T \times F$
(5)	$T \to F$
(6)	$F \to (E)$
(7)	$F \to id$

If I is the set of one item $\{[E' \to .E]\}$, then CLOSURE(I) contains the set of item I_0 as in Fig. 19.1. Try to analyse the closure is computed, first $E' \to .E$ is added in CLOSURE(I) due to (1). Since there is an E immediately to the right of a .(dot), we add the E productions with dots at the left ends: $E \to .E + T$ and $E \to .T$. Now there is a T immediately to the right of a dot in the latter item, so we add $T \to .T * F$ and $T \to .F$. Next, the F to the right of a dot forces us to add $F \to .(E)$ and $F \to .id$, but no other items need to be added. □

The closure can be computed as shown in Algorithm 1. The CLOSURE(I) function can be implemented by keeping a boolean array *added*, indexed by the nonterminals of G, such that *added*$[B]$ is set to *true* if and when we add the item $B \to .\gamma$ for each B-production $B \to \gamma$.

Algorithm 1 Computing the $CLOSURE(I)$

```
1: Set-of-items CLOSURE(I)
2: J = I;
3: repeat
4:    for each item A → α.Bβ ∈ J do
5:       for each production B → γ ∈ G do
6:          if (B → .γ ∉ J) then
7:             add B → .γ to J
8:          end if
9:       end for
10:   end for
11: until no more items are added to J on one round;
12: return J
```

Note that if B-production is added to the closure of I with the dot at the left end of production body, then all B-productions will be similarly added to the closure. Hence, it is not necessary in some circumstances actually to list the items $B \to .\gamma$ added to I by closure. A list of terminals B whose productions were so added will suffice. We divide all the sets of item of interest into two classes:

1. *Kernel items:* The initial items, $S' \to .S$, and all items whose dots are not at the left end.
2. *Nonkernel items:* All item with their dots at the left end, except for $S' \to .S$.

In Fig. 19.1, the kernel and non-kernel items are separated by a horizontal line.

Moreover, each set of items of interest is formed by taking the closure of a set of kernel items; the items added in the closure can never be closure items, of closure. Thus, we can represent the sets of items we are really interested in with very little storage if we through away all non-kernel items, knowing that they could be regenerated by the closure process.

Function GOTO The second useful function is $GOTO(I, X)$ function, where I is set of items, and X is a grammar symbol, and GOTO(I, X) provides a transition, from state I, with edge X, to other states. The $GOTO(I, X)$ is defined to be the closure of the set of all items $[A \to \alpha X.\beta]$ such that $[A \to \alpha.X\beta]$ is in I. Intuitively, the $GOTO$ function is used to define the transitions in the $LR(0)$ automaton for a grammar. The states of the automaton corresponds to sets of items, and $GOTO(I, X)$ specifies the transitions from the states for I under input X.

Example 19.2 *Demonstrating computations of $GOTO(I, X)$ function.*

If I is the set of two items $\{[E' \to E.], [E \to E.+T]\}$, then $GOTO(I, +)$ contains the items:

$$
\begin{aligned}
&E \to E + .T \\
&T \to .T * F \\
&T \to .F \\
&F \to .(E) \\
&F \to .id
\end{aligned}
$$

We have computed $GOTO(I, +)$ by examining I for items with + immediately to the right of the dot. The $E' \to E.$ is not such an item, but $E \to E.+T$ is. We moved the dot over the + to get $E \to E+.T$ and then took the closure of this singleton set comprising $E \to E+.T$, which gives remaining for items in above item set. □

We are now ready for the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'.

19.2.2 Use of LR(0) automaton

The main idea of "Simple LR" or SLR, parsing is the construction from the expression grammar, a LR(0) automaton. The states of automaton are the sets of item from the canonical LR(0) collections, and the transitions are given by GOTO function. The LR(0) automaton for the expression grammar shown in Table 19.1 was given in Fig. 19.1.

The start state of the LR(0) automaton is $CLOSURE(\{[S' \to .S]\})$, where S' is the start symbol of the augmented grammar. All the states are accepting states. We say, "state j" to refer to the state corresponding to the set of item I_j.

Algorithm 2 Computation of canonical collection of sets of LR(0) items

```
1: items(G'){
2: C = CLOSURE({[S' → .S]});
3: repeat
4:    for (each set of items I in C) do
5:       for (each grammar symbol S) do
6:          if (GOTO(I, X) is not empty and not in C) then
7:             add GOTO(I, X) to C
8:          end if
9:       end for
10:   end for
11: until no new set of items are added to C on a round
12: }
```

How can a LR(0) automata help with shift-reduce decisions? Shift-reduce decisions can be as follows. Suppose that the string[2] γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next input symbol a if state j has a transition on a. Otherwise, we choose to reduce; the items in state j will tell us which production to use. The LR parsing algorithm we are going to discuss shortly, uses stack to keep track of states as well as the grammar symbols; in fact, the grammar symbol can be recovered from the state, so the stack holds states. The next example give a preview of how an LR(0) automaton and a stack of states can be used to make shift-reduce parsing decisions.

Example 19.3 *Shift-reduce parsing using LR(0) Grammar.*

The table 19.3 illustrates the actions of shift-reduce parser on input $id * id$, using the LR(0) automaton shown in Fig. 19.1. The stack is used to hold the states; for clarity, the grammar symbols corresponding to states on the stack appear in the column Symbols. At line (1), the stack holds the start state 0 of the automaton; the corresponding symbol is the bottom-of-stack marker \$.

The next input symbol is id and state 0 has a transition on id to state 3. We therefore shift. At line (2), state 4 (symbol id) has been pushed on to the stack. There is no transition on input $*$, so we reduce. From item $[F \to id.]$ in state 4, the reduction is by production $F \to id$. With symbols, a reduction is implemented by popping the body of the production from the stack (on line 2), i.e., the the body is id and pushing the head of the production (in this case F). With states, we pop state 4 for symbol id, which brings state 0 to top of the stack, and look for a transition on F, the head of the production. The state 0 has a transition on F to state 5, so we push state 5 with corresponding symbol F, see line (3), and so on. Finally, we get the sequence of states pushed on and popped from stack as shown in Table 19.3.

[2]We take γ as a string because that (i.e., shifting) will assemble a string of one or more symbols on top of the stack, which is to be reduced

Table 19.3: Parsing of expression: $id * id$

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	$id * id$ \$	shift to 4
(2)	0 4	\$ id	$* id$ \$	reduce by $F \to id$
(3)	0 5	\$ F	$* id$ \$	reduce by $T \to F$
(4)	0 8	\$ T	$* id$ \$	shift to 10
(5)	0 8 10	\$ $T *$	id \$	shift to 4
(6)	0 8 10 4	\$ $T * id$	\$	reduce by $F \to id$
(7)	0 8 10 9	\$ $T * F$	\$	reduce by $T \to T * F$
(8)	0 8	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

19.3 Review Questions

1. In shift-reduce parsing, what do you mean by "derivation in reverse order"? Explain.
2. What is "handle pruning" in such reduce parsing? Explain.
3. What do you understand by "right-mots derivation" and "in reverse" in a shift-reduce parsing? What is the physical position of a handle, when it is about to be reduced?
4. At any moment, what is preferred operation out of "shift" and "reduce"? Justify your answer.
5. What are shift-reduce and reduce-reduce conflicts?
6. What are the functions performed by LR(0) automton?
7. Why there is need of augmented grammar?
8. What is *Closure* in LR(0) automata?
9. How is canonical LR parsing different from LALR parsing?
10. Explain the difference between kernel and non-kernel items.
11. Can you generate all the non-kernel items using kernel items?
12. What is fundamental difference between shift-reduce and LR parser?

19.4 Exercises

1. Show that for the grammar $S \to aSa \mid aa$, which generate all the strings of even length, a recursive descent parser recognizes strings $aa, aaaa, aaaaaaaa$, but not $aaaaaa$.
2. Consider the following grammar where S is the start symbol:

$$S \to ictSeS \mid ictS \mid a$$

   (a) Compute FOLLOW for each non-terminal of the above grammar.
   (b) Construct the canonical collection of LR(0) items for the grammar.

(c) Is the grammar SLR(1)? Why?

3. (a) Construct the set of LR(0) items for the grammar given below and construct the SLR parsing table for it. Is the grammar LR(0) ?

```
T -> B | { L }
L -> T L | B
B -> a | b
```

   (b) Depict the parsing action (stack content, remaining input, action) sequence of the above parser for the input string $\{a\{ba\}\}$

4. Consider the grammar-

```
S -> ict{S}eS |ictS | a = aPa
P -> + | *
```

   (a) List all the LR(0) item sets for the grammar.

   (b) Construct the SLR(1) parsing table.

   (c) Is the grammar SLR(1)? Is the grammar unambiguous?

5. How is canonical LR parsing method different from SLR parsing? What would you say about their respective language recognition power?

6. Explain the construction process of closure of an Item set, in your own words.

7. Find the closure of following item sets, as well justify the construction.

   (a) $\{[F \rightarrow (.E)]\}$

   (b) $\{[F \rightarrow (E.)]\}$

   (c) $\{[T \rightarrow T. * F]\}$

   (d) $\{[T \rightarrow T * .F]\}$

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.