A Short Introduction to the LR(0) Parser Method

José de Oliveira Guimarães
jose@dc.ufscar.br

May 21, 2004

The LR(k) methods for parsing scan the tokens from **L**eft to right (as usual) and create a **R**ightmost derivation for the start non-terminal symbol. k is the number of lookahead tokens needed before deciding what to do during the analysis (this will become clear later).

We will only study LR(0) parsers. They are a good introduction to other LR(k) parsers and are not too complex. We will use the grammar

```
Z ::= E$
E ::= T | E + T
T ::= i | ( E )
```

An LR(0) item is a production with a point on it, such as

```
E ::= E• + T
T ::= •i
T ::= i•
T ::= ( E • )
```

An item `N ::=` $\alpha \bullet \beta$ means that α has already been matched with the input and that we hope to find something derived from β in the tokens ahead. For example, suppose we are using the item `E ::= E• + T` and the input "`i• + i`". The point in the input shows where is the next token, which is `+` in this case (the next token appears after the point). Therefore, part of the input have already been parsed and we believe: a) the production `E ::= E + T` matches the input, b) `E` matches `i` (before the point in `E ::= E • + T` matches before the point in "`i • + i`"), and c) the tokens following the point, "`+ i`" will be matched by "`+ T`".

Since the `+` in the input matches the `+` in the production, we end up with the item `E ::= E +• T` and input "`i +• i`". We hope `T` will derive `i` because `T` appears after the point in the item.

At some point, we will have the item "`E ::= E + T•`" and input "`i + i•`". Since the point is at the end of the item, it can be reduced to `E`. We call "`E + T`" a handle, which is a sequence of symbols that matches the input and can be reduced.

Things are not so simple as explained above. We are not always sure of which item to use. Given the input "`•i$`" and the above grammar, we start with non-terminal `E`. But should we use item "`E ::= •T`" or "`E ::= •E + T`"?. We have not yet started the parsing (the point appears before the input). Even when we give a look at the first token, `i`, we cannot decide which item to use. It can be both. Then we use both. Later on, the wrong one, "`E ::= •E + T`", will be discarded. In general, we use a *set of items* that possibly are the correct ones.

An LR(0) item `N ::=` $\alpha \bullet \beta$ is used to mean that:

- α has already been recognized in the input; that is, some tokens that appear before the current token match α. This does not mean all the input before the current token were recognized by α;
- we hope the next tokens will match β and therefore $\alpha\beta$ is a handle that will be reduced to `N`. If it really is, it will be reduced to `N` when the point reach the end of the production: "`N ::=` $\alpha\beta\bullet$".

An item `N ::=` $\alpha \bullet \beta$ is called a shift item. An item `N ::=` $\alpha\beta\bullet$ is called a reduce item.

LR(0) parsing is made using an automaton with the item sets as shown in Figure 2.89 (see book xerox). We will describe the steps of the algorithm through an example which walks through the automaton recording the path from S_0, the first state, to the current state. Then the algorithm may walk from S_0 to S_1 to S_3 through the arrows labelled `E`, `+`, and `T`. We will use the input "`i + i $`" in the example. This input is changed by the algorithm resulting in what will be called "modified input". For example, `T` may appear in the input: "`T` $\bullet$ `+ i $`". The point indicates the next token, which is `+` in this case. `T` has already been recognized.

The algorithm starts in state S_0, which has five items. Since we do not know which item is adequate for the input, we put them all in S_0. As the input is "`i + i$`", the algorithm moves to state S_5, "`i`$\bullet$ `+ i`" and `i` is reduced to `T`. The result is "$\bullet$`T + i`" and the current state is S_0 again. State S_5 is removed from the path $S_0 \longrightarrow S_5$ because the arrow labelled "`i`" was reduced to `T`. Now the algorithm moves to state S_6 since the current symbol is `T` and the current state is S_0: "`T`$\bullet$ `+ i`". In S_6, `T` is reduced to `E`, resulting in "$\bullet$`E + i`". The current state is S_0 again and we move to state S_1: "`E`$\bullet$ `+ i`". Since the current symbol is `+`, the algorithm moves to state S_3: "`E +`$\bullet$ `i`". The current symbol is `i` and the current state becomes S_5: "`E + i`$\bullet$". Symbol `i` is reduced to `T`: "`E + `$\bullet$ `T`". The current state is S_3 again. The current symbol is `T` and we move to S_4: "`E + T`$\bullet$". There is a reduction to `E`: "$\bullet$`E`". Note that in the path $S_0 \longrightarrow S_1 \longrightarrow S_3 \longrightarrow S_4$, the arrows are labelled with the symbols `E`, `+`, and `T`. This means the parser recognized the symbols `E + T` in the input. There were several alternatives in S_0, but the input directed the parser to S_4 via S_1 and S_3. Since S_4 reduces `E + T` to `E`, we return to state S_0 — the states S_1, S_3, and S_4 were associated to `E`, `+`, and `T` and are removed because of the reduction `E ::= E + T`. Now the current state is S_0 and the modified input is just "$\bullet$`E$`". We move to S_1: "`E`$\bullet$`$`". There is only one option: move to S_2, resulting in "`E$`$\bullet$". States S_1 and S_2 are removed, "`E$`" is reduced to `Z`, and the current state is S_0. The parsing ends successfully.

Let us show now how LR(0) parsing can be made in a more systematic form. We will use a stack with states and symbols:

$\mathtt{s}_0\mathtt{A}_0\mathtt{s}_1\mathtt{A}_1 \ldots \mathtt{s}_t\mathtt{A}_t$

The rightmost state is the top of the stack. The path from the start state (S_0 in the example) to the current state is represented in the stack. For example, in parsing "`i + i`", when we reach S_5 the stack is

$\mathtt{S}_0\mathtt{E}\mathtt{S}_1\mathtt{+}\mathtt{S}_3\mathtt{i}\mathtt{S}_5$

State S_5 orders a reduction `T ::= i`. All states and symbols associated to the reduction are removed from the stack, resulting in

$\mathtt{S}_0\mathtt{E}\mathtt{S}_1\mathtt{+}\mathtt{S}_3\mathtt{T}$

Now with S_3 and `T` the algorithm moves to state S_4:

$\mathtt{S}_0\mathtt{E}\mathtt{S}_1\mathtt{+}\mathtt{S}_3\mathtt{T}\mathtt{S}_4$

State S_4 orders a reduction `E ::= E + T`. Again, all states from the path $S_0 \longrightarrow S_1 \longrightarrow S_3 \longrightarrow S_4$ but S_0 are removed from the stack and `E + T` is replaced by `E`:

$\mathtt{S}_0\mathtt{E}$

The complete LR(0) parsing of the input "`i + i$`" is shown below.

Stack	Input	Action
S_0	`i + i$`	shift `i`
S_0`i`S_5	`+ i $`	reduce by `T ::= i`
S_0`T`	`+ i $`	
S_0`T`S_6	`+ i $`	reduce by `E ::= T`
S_0`E`	`+ i $`	
S_0`E`S_1	`+ i $`	shift `+`
S_0`E`S_1`+`S_3	`i $`	shift `i`
S_0`E`S_1`+`S_3`i`S_5	`$`	reduce by `T ::= i`
S_0`E`S_1`+`S_3`T`	`$`	
S_0`E`S_1`+`S_3`T`S_4	`$`	reduce by `E ::= E + T`
S_0`E`	`$`	
S_0`E`S_1	`$`	shift $
S_0`E`S_1`$`S_2		reduce by `Z ::= E$`
S_0`Z`		accept

The LR(0) method decides to shift or reduce an item without consulting any token of the input. This is why this method uses 0: it uses 0 tokens when deciding which action to take.