TDDD14/TDDD85 Lecture 12: LL(1) and LR(0) parsing

Jonas Wallgren

Abstract

This lecture presents the two parsing methods LL(1) and LR(0) and their theoretical background.

1 Introduction

In formal langauge theory you can use a PDA/DPDA to reason about accepting or rejecting a string in the language defined by a CFG. But in a more practical setting, in compiler theory and technology, you need something else, something whose formulation comes closer to the grammar, something that comes closer to implementation and leeds to such an implementation being at least somewhat efficient. This lecture presents two such methods—LL(1) and LR(0)—and the next lecture a third one—LR(1).

In compilers you call the process of accepting or rejecting a string *parsing* and it is not primarily seen as a method to accept or reject a string but to build its derivation tree, or parse tree as we will call it now. The methods presented in these two lectures will still only accept or reject strings, but they are constructed in such a way that implementations of them could easily be extended with code that really builds the trees.[1]

We need some terms:

Definition 1. *A* prefix *of a string is an initial part of it. E.g. the prefixes of abcd are ε, a, ab, abc, and even abcd.*

Definition 2. *A* sentential *form is a string $\gamma \in (\Sigma + N)^*$ of terminals and nonterminals that may be derived from the start symbol: $S \overset{*}{\Rightarrow} \gamma$.*
E.g. $S \overset{*}{\Rightarrow} aAbC$ in grammar G_8 in lecture 11.

Definition 3. *A* token *in compilers is what in formal languages is called a symbol (in the alphabet). The leaves of our parse trees will be called tokens.*

In e.g. section 3 and lecture 13 we will be able to explicitly recognize the end of a string. For that purpose all strings in those cases are equipped with an extra, last end-of-string symbol $\$ \notin \Sigma$.

[1] See Kozen (lecture 26) or some compiler course material.

2 Example grammar

Throughout this lecture we will use this small example grammar (start symbol S):

$S \to aBCd$
$B \to pq$
$C \to rs$

The language of this grammar contains just the string $apqrsd$, so it is regular, but it can be used to illustrate the ideas and techniques of our parsing methods.

The only leftmost derivation with the grammar is:

$S \Rightarrow_{lm} aBCd \Rightarrow_{lm} apqCd \Rightarrow_{lm} apqrsd$

and the only rightmost derivation with the grammar is:

$S \Rightarrow_{rm} aBCd \Rightarrow_{rm} aBrsd \Rightarrow_{lm} apqrsd.$

The leftmost derivation corresponds to building the parse tree as in Fig. 1 in the last page, from the root to the leaves, top-down.

The tree in Fig. 2 is built in the other direction, from the leaves to the root, bottom-up. The principle is to build a part of the tree as soon as you can. When e.g. apq is read you can build the B node without having read anything more yet. The leaves not shown in each step are yet unread. If you in each step imagine the parts built so far together with the unread tokens, e.g. with B just built it is $aBrsd$, you see that the tree building step by step corresponds to the rightmost derivation backwards. If you read a string from left to right and build nodes bottom-up as soon as possible it corresponds to deriving from the right if the tree were to be expanded top-down.

3 LL(1) parsing

The first L stands for **L**eft to right reading of the string. The second L stands for **L**eftmost derivation, The number 1 means that to decide what to do next during the parsing you are allowed to look ahead, to peek, at the next token without really using it. You talk about 1 token lookahead.

To be able to handle lookahead we need the following construction:

Definition 4. *For a nonterminal A in a grammar*
FOLLOW(A)$=\{a \in \Sigma | \exists \gamma_1, \gamma_2 : S \stackrel{*}{\Rightarrow} \gamma_1 A a \gamma_2\} \cup \{\$ | \exists \gamma_1 : S \stackrel{*}{\Rightarrow} \gamma_1 A\}$

I.e. If a sentential form can contain A immediately followed by a then a belongs to FOLLOW(A). And if a sentential form can end in A, then a special end-of-string marker, \$, belongs to FOLLOW(A).

So, in our example grammar FOLLOW(B)=$\{r\}$, FOLLOW(C)=$\{d\}$, FOLLOW(S)=$\{\$\}$.

Now we can define LL(1):

Definition 5. *The grammar $G = \langle N, \Sigma, P, S\rangle$ is LL(1) iff when there are two rules $A \to \alpha$ and $A \to \beta$ so:*

- *If $\alpha \stackrel{*}{\Rightarrow} a\gamma_1$ and $\alpha \stackrel{*}{\Rightarrow} b\gamma_2$ then $a \neq b$.*
- *If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ then $\beta \not\stackrel{*}{\Rightarrow} \epsilon$.*
- *If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and $\beta \stackrel{*}{\Rightarrow} a\gamma$ then $a \notin FOLLOW(A)$.*

The first point says that if you look ahead on the next token you should be able to decide which rule to use.

The second point says that you shouldn't be able to derive the empty string with different rules.

The third point says that you shouldn't be able to choose between reading a in this part of the derivation and not do it.

In a PDA given the string ax and the stack $A\gamma$ there could be several possible actions. If the grammar is LL(1) there is always at most one alternative.

Some important properties of LL(1) grammars are:

- An LL(1) grammar is unambiguous.
- An LL(1) grammar can't have left-recursion. If a given grammar has left-recursion it has to be rewritten in order to possibly become LL(1). Se section 6 in lecture 8.
- An LL(1) grammar can't contain $A \to \alpha\beta|\alpha\gamma$. If a given grammar uses such a formulation it has to be rewritten to $A \to \alpha B, B \to \beta|\gamma$ for the grammar to have a possibility to become LL(1). Such rewriting is called left factoring.

3.1 Recursive descent

Recursive descent is one way of implementing an LL(1) parser. The main idea is that there is one subrogram pA for each nonterminal A. The body of a subprogram follows the right-hand side(s) of the rule(s) for the nonterminal. Some examples:

Grammar rule	$S \to aBCd$	$T \to aX\|bY$	$U \to aM\|N$

Program	procedure pS:
	`read a;`
	`call pB();`
	`call pC();`
	`read d;`
		procedure pT:
		`read first token;`
		`if a: call pX();`
		`if b: call pY();`
			procedure pU:
			`look at first token;`
			`if a: read first token; call pM();`
			`else: call pN();`

3.2 Table driven parsing

The grammar can be coded into a table. Parsing then is done by reading the table step by step while reading the string. It is like using the next-configuration function for a PDA. This will be treated in a compiler course.

4 LR(0) parsing

The L stands for **L**eft to right reading of string. The R stands for **R**ightmost derivation (in reverse). The number 0 means that we don't use any lookahead.

LR parsing works by constructing an almost-DFA. There are states and transitions as in a DFA, but we don't have any final states. We'll use the structure of the DFA to guide us through the parsing. The states of the DFA contains LR items:

Definition 6. *An LR(0) item is a grammar rule with a dot somewhere in the right-hand side.*

Examples of LR(0) items are $S \to \cdot aBCd, B \to p \cdot q,$ and $C \to rs\cdot$. The dot is a marker showing how much of a rule is used during the actual parsing.

4.1 Building the automaton

To explain the automata construction we can start with an NFA. There is one grammar rule for the start symbol. In the first state of the NFA we put the item for the start symbol with the dot at the front. See state 0 in Fig. 3. When we read the first token a we make a transition to state 1 and move the dot one step ahead. After that we should read a B, a C, and a d and finally end up in state 4 after having moved the dot one step further for each transition. But wait! We can read a and d in one step each, but B and C correspond to whole substrings. Their contents have to be read one token at a time. There must be a sequence of states that move the dot across the right-hand side of the rule for B, and also for C. So, from state 1 to state 1' you can go without reading. From state 1' you can read pq in two transitions to state 6. And from state 2, without reading, you can come to state 2' and from there to state 8 by reading rs. The item in state 6 has the dot at its end. It is called a complete item. Also the items in states 4 and 8 are complete ones. When you stand in state 1 and want to read a B you can regard the transition to state 1' as a subroutine call. The complete item then implies a return back to state 1 and a signal that the whole B substring is read. If there are several rules for a nonterminal there should be ε arrows like 1–1' to states with items for all rules for the nonterminal.

Since there may be several rules for the start symbol the real start state is q_0 with epsilon moves to the first states for all rules for the start symol.

We don't want ε arrows. We want a DFA. The result of the transformation (the subset construction) is shown in Fig. 4. The automaton in Fig. 3 is constructed to show the background for Fig. 4 but that automaton can be constructed directly from the grammar:

In the start state there are the items for the start symbol with the dot in the beginning.

For each state:

- If it contains an item with the dot in front of a nonterminal then add to the state all items for that nonterminal with the dot in the beginning.
- For each non-complete item make a transition to a state with the same item with the dot moved one step ahead and the same symbol on the transition.

E.g. State 1 is constructed from state 0 by moving the dot over a in the item and there is an a on the transition. Now, in state 1 there is an item with the dot in front of B. Then the item for B with the dot in the beginning is added to the state. That results in two items in the state. They give rise to the two transitions to states 2 and 5 handling corresponding items with the dot moved and the symbols on the transitions.

4.2 Principal treatment

If in a rightmost derivation there is a step $aBCd \Rightarrow aBrsd$ we will in parsing look at $aBrsd \Leftarrow_{rm} aBCd$ from left to right, i.e. we will from the parts rs construct C, since that corresponds to the backwards reading of a grammar rule. The rs part is called a handle. A handle is what is to be replaced by a nonterminal in a backwards derivation step, a reduction step. We want to find the handles to know where to reduce. Starting from state 0 reading aBrs we end up in state 8 with a complete item. There a handle is found. All the prefixes up to that point—a, aB, aBr, and $aBrs$—are called viable prefixes.

Definition 7. *An item $A \to \alpha \cdot \beta$ is* valid *for a viable prefix $\delta\alpha$ if* $S \stackrel{*}{\Rightarrow}_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$

I.e. the next step in the parsing is to reduce $\alpha\beta$ to A. We have just read as far as the α part of that, so an appropriate item is $A \to \alpha \cdot \beta$. The whole prefix up to this point is $\delta\alpha$, it is viable since we are about to read a handle. Thus the item is valid for this prefix.

The states of the automaton contain valid items.

4.3 Using the automaton

Now, we will describe how to use the automaton in Fig, 4.

Parsing a string consists of two different actions:

- *Shift*: One token is read and a transition step is taken in the automaton. E.g. in state 1 p is read and the new state is 5.
- *Reduce*: One new derivation step is found. If the parse tree was built a new part of the parse tree could be built now. E.g. in state 6 the B node is built. That causes the control to go back to state 1 and continue to state 2 since the dot now can be moved over the B in the item in state 1.

Like for PDA configurations parsing actions handle a stack and the input string. For every shift action both the symbol read and the new state are pushed onto the stack. For every reduce action the right-hand side of the current rule are popped from the stack together with the corresponding states. The left-hand side of the grammar rule is pushed instead together with the resulting new state. It works like this:

Stack	Remaining string	Action
0	$apqrsd$	
		Shift
$0a1$	$pqrsd$	
		Shift
$0a1p5$	$qrsd$	
		Shift
$0a1p5q6$	rsd	
		Reduce $B \to pq$
$0a1B2$	rsd	
		Shift
$0a1B2r7$	sd	
		Shift
$0a1B2r7s8$	d	
		Reduce $C \to rs$
$0a1B2C3$	d	
		Shift
$0a1B2C3d4$		
		Reduce $S \to aBCd$
$0S$		
		Accept

There, at the end, with only the start state and the start symbol on the stack there is a third action, *Accept*, which means that the string belongs to the language.

Definition 8. *A grammar is LR(0) if it is accepted by an LR(0) parser.*

Definition 9. *A language is LR(0) if it has an LR(0) grammar.*

5 More to think about

1. Recognizing CFLs needs a stack, e.g. as in a PDA, Where is the corresponding stack in the recursive descent method?
2. Implement a recursive descent parser for the example grammar used during this lecture in your favourite programming language.
3. What is the complexity of the parsing methods proposed during this lecture? How much memory is needed?

Fig. 1

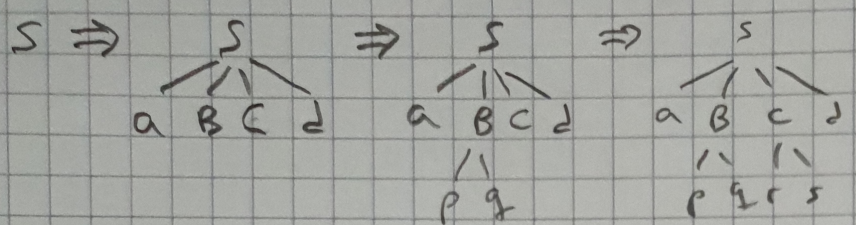

Fig. 2

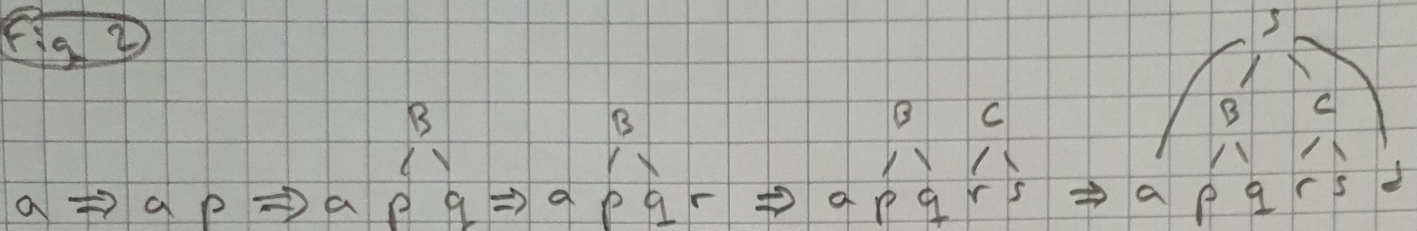

Fig. 3

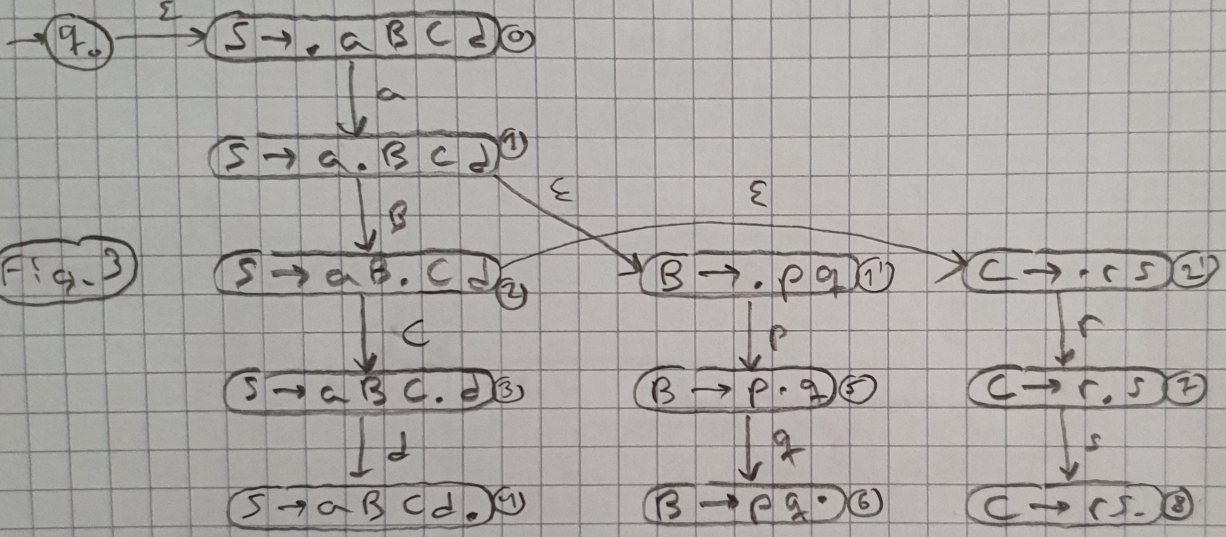

Fig. 4