TDDD14/TDDD85 Lecture 13: LR(1) parsing

Jonas Wallgren

Abstract

This lecture closes the context free part of the course by presenting LR(1) parsing.

1 Introduction

Last lecture presented LR(0) parsing as a method to parse strings in a languge defined by a CFG. That language was unproblematic to parse. In this lecture we will continue our exploration of the LR parsing world.

2 Two problematic grammars

We will show two grammars having different problems.

2.1 One problematic grammar

This is a grammar for a simple language:

$S \rightarrow (A)|(B)$
$A \rightarrow char|integer|ident$
$B \rightarrow float|double|ident$

Let's try to construct the LR(0)-item automaton for the grammar. See Fig. 1 at the last page. The first state contains the items with the dot first for the two rules for the start symbol. The only thing to do is to read a left paranthesis. That takes us to the next state. The dots are moved over the parantheses. Now since there is a dot in front of A and B the other six items are added to the state. One possible way to continue to a new state is to read *ident* and go to the third state. Now, there is a problem: Which rule to use for reduction—$A \rightarrow ident$ or $B \rightarrow ident$?

Definition 1. *A* reduce-reduce conflict *is a situation where there are (at least) two complete items in a state in the item automaton.*

If an LR(0)-item automaton shows (a) reduce-reduce conflict(s) the grammar isn't LR(0).

2.2 Another problematic grammar

This is a grammar for a language not quite that simple:

$S \to L$
$L \to L - E$
$L \to E$
$E \to a$
$E \to b$

The language of this grammar contains strings like $a - b - a - b$.

An initial attemp to construct the LR(0)-item automaton for the grammar is found in Fig. 2. The first item for the first rule is found at top in the first state. Since there is a dot in front of L the next two items are added. Since there now is a dot in font of E the last two items are added. One of the possible things to do is, as a result of a reduce action, to move the dot over L and go to the next state. What to do in that state? There is a possibility to read a $-$ and there is a possibility to reduce.

Definition 2. *A* shift-reduce conflict *is a situation where ther are (at least) one complete item and one not-complete item in a state in the item automaton.*

If an LR(0)-item automaton shows (a) shift-reduce conflict(s) the grammar isn't LR(0).

3 LR(1) items

The 1 stands for one token lookahead.

Recall this passage from the last lecture:

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

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."

Now, in LR(1) we can take into account what follows A.

Definition 4. *An LR(1) item is an LR(0) item together with a lookahead token in a set. (Also \$ is seen as a token here*[1]*.)*

An example of an LR(1) token is $L \to L \cdot -E\{\$\}$.

Definition 5. *Let* $a \in \Sigma$.
An item $A \to \alpha \cdot \beta\{a\}$ *is valid for a viable prefix* $\delta\alpha$ *if* $S \overset{*}{\Rightarrow}_{rm} \delta Aaw \Rightarrow \delta\alpha\beta aw$.

The situation is like in LR(0) with the added constraint that a follows what A is derived to.

Definition 6.
An item $A \to \alpha \cdot \beta\{\$\}$ *is valid for a viable prefix* $\delta\alpha$ *if* $S \overset{*}{\Rightarrow}_{rm} \delta A \Rightarrow \delta\alpha\beta$.

If a state of items in the automata contains several LR(1) items built from the same LR(0) item they could be collapsed into one LR(1) item with several elements in its lookahead set.
E.g. $A \to b \cdot c\{x\}$ and $A \to b \cdot c\{y\}$ gives $A \to b \cdot c\{x, y\}$.

Elements in the lookahed set are elements in the FOLLOW set for the nonterminal in the left-hand side.

4 Building the automaton

This is how the automaton in Fig. 3 is constructed. Start with the top item in state 0. It's the only rule for the start symbol. The dot is in the beginning. The lookahead set contains what could follow S, i.e. end of string. There is a dot before L, so there are the two new items $L \to \cdot L - E\{\$\}$ and $L \to \cdot E\{\$\}$. Since L is at the end of the first item what could follow L is what could follow S so the lookahead set is the same. Now there is a dot before L in another situation—it's followed by a $-$. The new items are $L \to \cdot L - E\{-\}$ and $L \to \cdot E\{-\}$. As I said before these items could be written as the two items in state 0.

It's only when a dot is in front of a nonterminal we have to bother about constructing a new lookahead set. If the nonterminal is at the end just copy the lookahead set of the rule where the dot was found. If the nonterminal is not at the end then the new lookahead set should contain those tokens that may follow directly after the nonterminal.

When moving the dot over a symbol the lookahead set is not changed. Moving the dot takes place in one item having the same nonterminal in its left-hand side, so the lookahead situation doesn't change.

[1] I.e. the new special end-of-string symbol

5 Using the automaton

Shift actions work as in the LR(0) case. Lookahed sets are involved in reduce actions. Look at state 3. Now the shift-reduce conflict can be solved. If we see a $-$ it is a token that cannot follow S, so it's OK to shift. If we see \$ it cannot be shifted (\$ never can.) but it can follow S so it is OK to reduce.

The automaton usually is described/represented by this table:

State	a	b	$-$	\$
0	shift	shift		
1			reduce $E \to a$	reduce $E \to a$
2			reduce $E \to b$	reduce $E \to b$
3			shift	accept
4	shift	shift		
5			reduce $L \to L - E$	reduce $L \to L - E$
6			reduce $L \to L - E$	reduce $L \to L - E$

The empty positions in the table mean error. Some positions you come to if you try to parse a string not in the language of the grammar. Some other positions you end up in only if you handle the method wrong.

So, parsing using an LR(1) table follows this steps:

- Start in the start state with the start state on the stack.
- Look at the next token without really reading it.
- Look up the next action in the table using the state and the token as indices.
- Perform the action as for LR(0).
- Iterate this procedure until an Accept action or an error is found in the table.

6 A hierarchy of grammars and a hierarchy of lamguages

We have just seen an example of a grammar that is not LR(0) but it is LR(1). The following relations hold between some grammar formalisms:

$$LR(0)\ languages \subset SLR(1)\ languages \subset LALR(1)\ languages \subset LR(1)\ languages.$$

The two in the middle thus have a power in between LR(0) and LR(1). They will be presented in some compiler course. Often LR(1) automata/tables become very large in "real" applications. Those two formalisms try to solve that problem to some extent while still having the capability to express normal programming language constructions.

If we continue beyond LR(1) this is a relation between grammars:

$$LR(0)\ grammars \subset LR(1)\ grammars \subset LR(2)\ grammars \subset LR(3)\ grammars \subset \cdots$$

i.e. you can have a grammar that needs 3 tokens lookahead, it is LR(3), and gets a conflict when handled with LR(2). This is the corresponding relation between languages:

$$LR(0)\ languages \subset LR(1)\ languages = LR(2)\ languages = LR(3)\ languages = \cdots$$

i.e. if a language has an LR(3) grammar that grammar can be rewritten to an LR(2) one and even an LR(1) one. It maybe will be much bigger, but it can be done. It is only the step from LR(0) to LR(1) that makes the set of definable languages larger.

7 More to think about

- Parse the string $a{-}b{-}a$ according to the table above and/or the automaton in Fig. 3.
- Can every context-free grammar be converted into an equivalent LR(k) grammar, for some k?

Fig.1

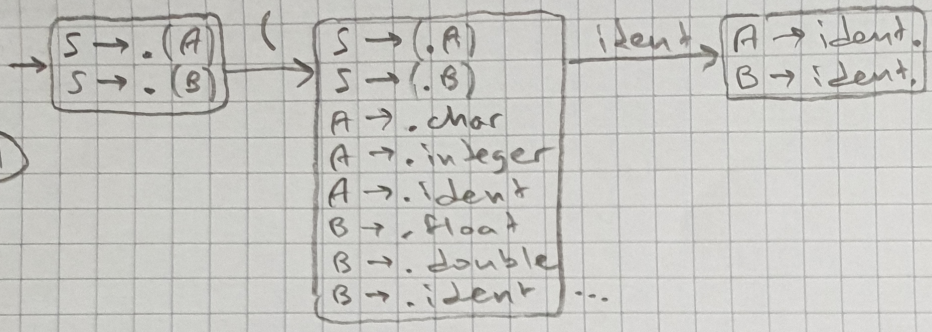

Fig.2

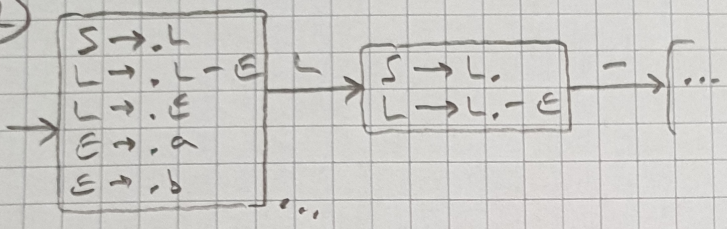

Fig.3