

TDDD14 / TDDD85 – Lecture 12

LL(1) and LR(0) parsing

August Ernstsson, 2026 (based on lecture notes by Jonas Wallgren)

Let's start!

Today's topic

- In formal language 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
 - whose formulation comes closer to the grammar,
 - that comes closer to implementation, while being reasonably efficient.
- This lecture presents two such methods: **LL(1)** and **LR(0)** parsing.
- In the next lecture a third one: **LR(1)**.

Parsing

- In compilers you call the process of accepting or rejecting a string **parsing**
 - it is not (primarily) seen as a method to accept or reject a string,
 - but rather to build its derivation tree, or **parse tree**.
- The methods presented in these two lectures will still only accept or reject strings,
 - but they are constructed in such a way that they could be extended with code that builds the trees (see Kozen, chapter 26).

Definitions

- **Definition 1.** A *prefix* of a string is an initial part of it.
 - Example: 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: $S \xRightarrow{*} \gamma$.
 - Example: $S \xRightarrow{*} aAbC$ (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 input string is a sequence of tokens.
 - The leaves of our parse trees will be tokens.

End-of-string marker

- We want to be able to *explicitly* recognize the end of a string.
- Therefore, all strings in those cases are equipped with an extra, last end-of-string symbol $\$ \notin \Sigma$.

Working grammar

- We will use this small example grammar (start symbol S):
 - $S \rightarrow aBCd$
 - $B \rightarrow pq$
 - $C \rightarrow rs$
- 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.
- Only possible leftmost derivation: $S \Rightarrow_{lm} aBCd \Rightarrow_{lm} apqCd \Rightarrow_{lm} apqrsd$
 - **(On whiteboard)**
- Only possible rightmost derivation: $S \Rightarrow_{rm} aBCd \Rightarrow_{rm} aBrsd \Rightarrow_{rm} apqrsd$
 - **(On whiteboard)**

LL(1) parsing

- **L**: *Left to right* reading of the string.
- **L**: *Leftmost* derivation.
- **(1)** 1 token *lookahead*.
 - To decide what to do during the parsing, you are allowed to peek at the next token (without really using it).
 - For this purpose, we need to define *follow sets*.

Follow sets

- To be able to handle lookahead we need the following construction:
- **Definition 4.** For a nonterminal A in a grammar
 - $\text{FOLLOW}(A) = \{ a \in \Sigma \mid \exists \gamma_1, \gamma_2 : S \xRightarrow{*} \gamma_1 A a \gamma_2 \} \cup \{ \$ \mid \exists \gamma_1 : S \xRightarrow{*} \gamma_1 A \}$
- If a sentential form can contain A immediately followed by a ,
 - then a belongs to $\text{FOLLOW}(A)$.
- And if a sentential form can end in A , then a special end-of-string marker, $\$$ belongs to $\text{FOLLOW}(A)$.
- So, in our example grammar, $\text{FOLLOW}(B) = \{r\}$, $\text{FOLLOW}(C) = \{d\}$, $\text{FOLLOW}(S) = \{\$\}$.

Definition: LL(1)

- **Definition 5.** A grammar is LL(1) iff whenever there are two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$, the following holds:
 1. If $\alpha \xRightarrow{*} a\gamma_1$ and $\beta \xRightarrow{*} b\gamma_2$ then $a \neq b$ (Error in lecture notes)
 2. If $\alpha \xRightarrow{*} \varepsilon$ then not $\beta \xRightarrow{*} \varepsilon$.
 3. If $\alpha \xRightarrow{*} \varepsilon$ and $\beta \xRightarrow{*} a\gamma$ then $a \notin \text{FOLLOW}(A)$.
- Interpretation:
 1. *If you look ahead on the next token, you should be able to decide which rule to use.*
 2. *You shouldn't be able to derive the empty string with different rules.*
 3. *You shouldn't be able to choose between reading "a" and not do it.*

Some LL(1) properties

- In a PDA, given the string \mathbf{ax} and the stack \mathbf{Ay} , there could be several possible actions. If the grammar is LL(1), there is always *at most one* alternative.
- LL(1) grammars are *unambiguous*.
- LL(1) grammars *can't have left-recursion*.
 - If a given grammar has left-recursion it has to be rewritten in order to possibly become LL(1). See lecture 8.
- LL(1) grammars *cannot contain* $A \rightarrow \alpha\beta \mid \alpha\gamma$.
 - If present, it has to be rewritten to $A \rightarrow \alpha B$, $B \rightarrow \beta \mid \gamma$ for the grammar to have a possibility to become LL(1).
 - Such rewriting is called *left factoring*.

Recursive descent parsing

- *Recursive descent* is one way of implementing an LL(1) parser.
- There is one subprogram pA for each nonterminal A .
- The body of a subprogram follows the right-hand side(s) of the rule(s) for the nonterminal.

Grammar rule	$S \rightarrow aBCd$	$T \rightarrow aX bY$	$U \rightarrow aM N$
Procedure/function	procedure pS()	procedure pT()	procedure pU()
Program	<pre>read a; call pB(); call pC(); read d;</pre>	<pre>read first token; if a: call pX(); elseif b: call pY();</pre>	<pre>look at first token; if a: read first token; call pM(); elseif: call pN();</pre>

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.
- Similar to using the *next-configuration function* for a PDA!
- This will be treated in a compiler course.

LR(0) parsing

- **L**: Left to right reading of string.
- **R**: Rightmost derivation (in reverse).
- **(0)** means that we don't use any lookahead.
- LR parsing works by constructing an almost-DFA.
 - There are states and transitions, but we don't have any final states.
 - 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 $S \rightarrow \cdot aBCd$ $B \rightarrow p \cdot q$ $C \rightarrow rs \cdot$
 - The dot is a marker showing how much of a rule has been used during the actual parsing.

Building the LR(0) automaton

- On the whiteboard.

Handles and viable prefixes

- If in a rm-derivation there is a step $\mathbf{aBCd} \Rightarrow \mathbf{aBrsd}$, we will in parsing view $\mathbf{aBrsd} \leftarrow_{\text{rm}} \mathbf{aBCd}$ from left to right: we will "from the parts \mathbf{rs} construct \mathbf{C} ".
 - The \mathbf{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 \mathbf{aBrs} , we end up in state 8 with a complete item. There, a handle is found.
- All the prefixes up to that point— \mathbf{a} , \mathbf{aB} , \mathbf{aBr} , and \mathbf{aBrs} —are called *viable prefixes*.

Handles and viable prefixes

- **Definition 7.** An item $A \rightarrow \alpha \cdot \beta$ is valid for a viable prefix $\delta\alpha$ if $S \xRightarrow{*} \delta A w \Rightarrow_{\text{rm}} \delta\alpha\beta w$
 - That means, 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 $A \rightarrow \alpha \cdot \beta$.
 - 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 LR(0) automaton contain *valid items*.

Parsing: Using the automaton

- LR-parsing a string consists of two different actions: **shift** and **reduce**.
- **Shift**: One token is read and a transition step is taken in the automaton.
 - Example: In state 1: "p" is read, and the new state is 5.
- **Reduce**: One new derivation step is found.
 - A new part of the parse tree could be built now.
 - Example: 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.

Parsing: Using the automaton

- Like 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, together with the resulting *new state*.

Using the automaton

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

Definitions: LR(0)

- **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.*

To think about

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

Coming up soon ...

- This week:
 - **Tuesday:** LL(1) and LR(0) parsing. **Done!**
 - **Wednesday:** LR(1) parsing

Thanks for today!