TDDD14 / TDDD85 – Lecture 12

LL(1) and LR(0) parsing

August Ernstsson, 2024 (based on lecture notes by Jonas Wallgren)

LINKÖPING UNIVERSITY

From previous lectures

Closure under union

- **Theorem 1**. CFLs are closed under union.
- **Proof**. Construct a new grammar:
  - $G_3 = \langle N_1 \cup N_2 \cup \{ S_3 \}, \quad \Sigma_1 \cup \Sigma_2, \quad P_1 \cup P_2 \cup \{ S_3 \rightarrow S_1 \mid S_2 \}, \quad S_3 \rangle$.
  - (If N_1 and N_2 are not disjoint sets, rename the nonterminals in N_2)
  - Now, starting from S_3 all derivations starting with either S_1 or S_2 can be performed
    - All strings in $L(G_1)$ and $L(G_2)$ can be derived
    - $L(G_3) = L(G_1) \cup L(G_2)$

Closure under concatenation

- **Theorem 2**. CFLs are closed under concatenation.
- **Proof**. Construct a new grammar:
  - $G_4 = \langle N_1 \cup N_2 \cup \{ S_4 \}, \quad \Sigma_1 \cup \Sigma_2, \quad P_1 \cup P_2 \cup \{ S_4 \to S_1 S_2 \}, \quad S_4 \rangle$.
  - (If N_1 and N_2 are not disjoint sets, rename the nonterminals in N_2)
  - Now, starting from S_4 all derivations consist of one part derived from S_1 in $L(G_1)$ followed by one part derived from S_2 in $L(G_2)$,
    - $L(G_4) = L(G_1)L(G_2)$.

Let's start!

Today's topic

- 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 that
  - whose formulation comes closer to the grammar,
  - comes closer to implementation,
    - while being at least somewhat 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 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.

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: $S \overset{*}{\Rightarrow} \gamma$.
  - Example: $S \overset{*}{\Rightarrow}$ aAbC (in grammar G8 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.

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.
- One possible leftmost derivation: $S \Rightarrow_{lm} aBCd \Rightarrow_{lm} apqCd \Rightarrow_{lm} apqrsd$
  - **(On whiteboard)**
- One possible rightmost derivation: $S \Rightarrow_{rm} aBCd \Rightarrow_{rm} aBrsd \Rightarrow_{lm} apqrsd$
  - **(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).

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

Follow sets

- To be able to handle lookahead we need the following construction:
- **Definition 4**. For a nonterminal A in a grammar
  - $\text{FOLLOW}(A) = \{ a \in \Sigma \mid \exists \gamma_1, \gamma_2 : S \overset{*}{\Rightarrow} \gamma_1 A a \gamma_2 \} \cup \{ \$ \mid \exists \gamma_1 : S \overset{*}{\Rightarrow} \gamma_1 A \}$

- 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,
  - Then $ belongs to FOLLOW(A).

- So, in our example grammar, FOLLOW(B) = {r}, FOLLOW(C) = {d}, 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 $\alpha \stackrel{*}{\Rightarrow} a\gamma_1$ and $\beta \stackrel{*}{\Rightarrow} b\gamma_2$ then $a \neq b$ (Error in lecture notes)
  2. If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then <u>not</u> $\beta \stackrel{*}{\Rightarrow} \varepsilon$.
  3. If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ and $\beta \stackrel{*}{\Rightarrow} a\gamma$ then $a \notin$ FOLLOW(A).

- <u>Interpretation</u>:
  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 ax and the stack Aγ, 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 *can't 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 subrogram `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 → aBCd	T → aX|bY	U → aM |N
Procedure/function	**procedure pS()**	**procedure pT()**	**procedure pU()**

Program	`read a;`
	`call pB();`
	`call pC();`
	`read d;`
		`read first token;`
		`if a:`
		`   call pX();`
		`elsif b:`
		`   call pY();`
			`look at first token;`
			`if a:`
			`   read first token;`
			`   call pM();`
			`elsif:`
			`   call pN();`

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.

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 is used during the actual parsing.

Handles and viable prefixes

- If in a rightmost derivation there is a step aBCd ⇒ aBrsd we will in parsing look at aBrsd ⇐rm aBCd from left to right: we will from the parts rs construct C.
  - 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* <u>*valid*</u> *for a* <u>*viable prefix*</u> *$\delta\alpha$ if* $S \ {}_{*}\!\!\Rightarrow_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$
- I.e. the next step in the parsing is to reduce αβ to A. We have just read as far as the α part of that, so an appropriate item is A → α · β. The whole prefix up to this point is δα, 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.

Building the LR(0) automaton

- On the whiteboard.

Parsing: Using the automaton

- 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.
  - If the parse tree was built 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 → pq
0a1B2	rsd	Shift
0a1B2r7	sd	Shift
0a1B27r7s8	d	Reduce C → rs
0a1B2C3	d	Shift
0a1B2C3d4		Reduce S → 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:
  - **Monday**: LL(1) and LR(0) parsing. **Done!**
  - **Friday**: LR(1) parsing

Thanks for today!

LINKÖPING UNIVERSITY