TDDD14 / TDDD85 – Lecture 13

LR(1) parsing

August Ernstsson, 2024

(based on lecture notes by Jonas Wallgren)

LINKÖPING UNIVERSITY

Let's start!

Problematic grammar #1: **reduce-reduce conflict**

- This is a grammar for a simple language:

  S → (A) | (B)

  A → char | integer | ident

  B → float |double | ident
- Let's try to construct the LR(0)-item automaton for the grammar. **(On board)**
  - 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 → ident or B → ident?
- *If an LR(0)-item automaton shows (a) reduce-reduce conflict(s) the grammar isn't LR(0).*

Problematic grammar #2: **shift-reduce conflict**

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

  S → L

  L → L – E

  L → E

  E → a

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

Conflicts

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

- **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.*

LR(1) items

- The 1 stands for one token lookahead.

- **Definition 3**. *An item $A \to \alpha \cdot \beta$ is valid for a viable prefix $\delta\alpha$*
  if $S \overset{*}{\Rightarrow}_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$
  - ”The next step in the parsing is to reduce αβ to A.
    We have just read as far as the α part, 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.”

LR(1) items, cont.

- 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*. ($ is seen as a token here.)
  - 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* $S \overset{*}{\Rightarrow}_{rm} \delta A a w \Rightarrow \delta\alpha\beta a w$.
  - 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.

Bulilding the automaton

- **On board**

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: **(On board)**
- The *empty positions* in the table mean error.
- 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.

A hierarchy of grammars / languages

- 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 ⊂ SLR(1) languages ⊂ LALR(1) languages ⊂ 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.

A hierarchy of grammars / languages, cont.

- 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).
- 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 languages larger.

To think about

- Recognizing CFLs needs a stack, e.g. as in a PDA.
  - Where is the corresponding stack in the recursive descent method?
- **LL(1)**: 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 Lecture 12?
  - How much memory is needed?
- **LR(1)**: 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?

Thanks for today!

LINKÖPING UNIVERSITY