

# Programming Language Implementation VII

In this lecture we will continue to look at **bottom-up parsing**.

- **Constructing the LR parsing table**

The material is (loosely) based on Aho et al Chapter 4.

## LR Parsers

*Reminder:* The basic idea for all LR parser methods is to execute an automaton that recognizes the viable prefixes of the grammar.

A **viable prefix** is a prefix of a right sentential form that can appear on top of the stack of an SR parser.

This is equivalent to the following definitions:

- A prefix of a right sentential form  $w$  is called a viable prefix, if it does not extend past the right hand of the rightmost handle of  $w$ .
- If  $S \xleftarrow{*rrm} uXw \xleftarrow{rrm} uvw$  then  $v$  is a handle of  $uvw$  and each prefix of  $uv$  is a viable prefix.

In the last lecture we looked at LR parsers. In this lecture we will construct the **LR parsing table**.

Among the three classes of LR methods:

- **Simple LR (SLR):** this is the simplest and uses limited lookahead in the table construction.
- **Canonical LR:** this is the most powerful but leads to large parsing tables.
- **LALR:** is slightly less powerful than Canonical LR but leads to smaller parsing tables and suffices for almost all programming language constructs.  
It is used in `yacc` and `bison`.

we will focus on SLR parsing table construction since it is the easiest to understand and forms the basis for the other two methods.

The construction of the automaton for the viable prefixes is similar to how we turn a NFA into a DFA.

## Items

Construction of the parsing table requires us to keep track of “partially evaluated” grammar rules. **Items** formalize this notion.

An **(LR(0)) item** of a grammar is a production with a dot somewhere in the RHS.

Intuitively, the marker indicates which portion of the RHS we have already read and which portion we expect to find next.

For example, production  $exp \rightarrow exp + exp$  yields 4 items:

$$exp \rightarrow \cdot exp + exp$$

$$exp \rightarrow exp \cdot + exp$$

$$exp \rightarrow exp + \cdot exp$$

$$exp \rightarrow exp + exp \cdot$$

A production  $exp \rightarrow \epsilon$  generates only the single item

$$exp \rightarrow \cdot$$

The idea is to *group the items* of a grammar into sets that form the states of a DFA that recognizes the viable prefixes of the grammar.

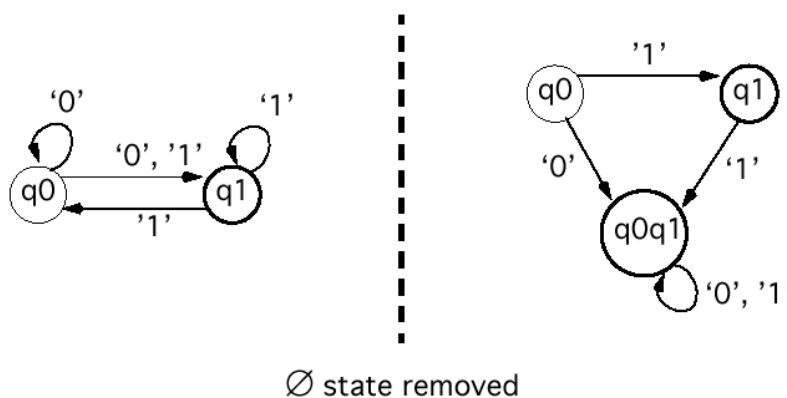
This grouping is essentially an NFA  $\rightarrow$  DFA subset conversion, and the states of the corresponding NFA would be marked by single items.

## Reminder: NFA-DFA Equivalence

A finite state automata is **deterministic** if it has no transitions via  $\epsilon$  and at most one successor state for each pair  $(q, a)$  where  $q \in Q$  and  $a \in \Sigma$ . We call such an automata a **DFA**.

Every non-deterministic state automaton (NFA) can be transformed into an equivalent deterministic state automaton (DFA) such that both automata accept the same language.

The “trick” is to model all possible state combinations (in the NFA) explicitly as separate states (in the DFA). This is called the *subset construction*.



The obvious problem is that this leads to a combinatorial explosion in the number of states.

## Reminder: Subset Construction

$$\text{NFA } N = (S, V, T, S_0, F)$$

$$\text{DFA } D = (S', V, T', S_0', F')$$

$$S' = 2^S$$

$$S_0' = \{S_0\}$$

$$F' = \{s \in S' \mid \exists x \in s : (x \in F)\}$$

$$((q, x) \rightarrow p) \in T' \text{ with } q = \{q_1, q_2, \dots, q_n\}$$

if and only if

$$((q_i, x) \rightarrow s) \in T \text{ and } p = \bigcup_i s$$

Construction of  $D$  for the previous example:

$$N = (\{q_0, q_1, q_2\}, \{0, 1\}, \{(q_0, 0) \rightarrow \{q_0, q_1\}, (q_0, 1) \rightarrow \{q_1\}, (q_1, 1) \rightarrow \{q_0, q_1\}\}, q_0, q_1)$$

yields

$$D = (\{\{q_0, q_1\}, \{q_0, q_1\}\}, \{0, 1\}, \{((q_0, 0) \rightarrow \{q_0, q_1\}), (q_0, 1) \rightarrow \{q_1\}, (q_1, 1) \rightarrow \{q_0, q_1\}, (\{q_0, q_1\}, 0) \rightarrow \{\{q_0, q_1\}\}, (\{q_0, q_1\}, 1) \rightarrow \{\{q_0, q_1\}\}\}, \{\{q_0\}, \{q_1\}, \{q_0, q_1\}\})$$

## LR construction

To construct the viable prefix recognizing automaton we need the so-called *canonical LR(0) collection*. which will be the set of states of this automaton.

We will need two functions to perform this:

- *closure* (on items) and
- *goto* (the transition function of this automaton)

We start with an augmented grammar  $G'$  which is  $G$  with a new start symbol  $S'$  and the new production  $S' \rightarrow S$ , where  $S$  is the start symbol of the original grammar.

We need this extra production to recognize the reduction that leads to acceptance.

## Closure

First we need to define the closure operation on items.

The **closure** of a set of items  $I$ , written  $closure(I)$ , is computed by:

- Start with  $I$ .
- If

$$A \rightarrow \alpha \cdot B\beta$$

is in the set and

$$B \rightarrow \gamma$$

is a production, add

$$B \rightarrow \cdot \gamma$$

Continue doing this until nothing can be added.

The intuition is that if  $A \rightarrow \alpha \cdot B\beta$  is in  $closure(I)$  then we might expect to next see a substring derivable from  $B\beta$ .

Since  $B \rightarrow \gamma$  is a production we might therefore expect to see a string derivable from  $\gamma$ .

Consider the augmented grammar

$$\begin{aligned} exp' &\rightarrow exp \\ exp &\rightarrow exp + term \\ exp &\rightarrow term \\ term &\rightarrow term * factor \\ term &\rightarrow factor \\ factor &\rightarrow (exp) \\ factor &\rightarrow \mathbf{int} \end{aligned}$$

If  $I$  is  $\{exp' \rightarrow \cdot exp\}$  then what is  $closure(I)$ ?

## Valid Items

Next we have to construct the *goto* function. This function (together with the item collection) embodies the idea of a *valid item*.

An item  $X \rightarrow Y.Z$  is *valid* for a viable prefix  $vY$  if there is a rightmost derivation  $S \xleftarrow{*}_{rrm} vXw \xleftarrow{rrm} vYZw$ .

Informally this means: If we find  $vY$  on top of the stack and  $X \rightarrow Y.Z$  is a valid item then if  $Z$  is not empty we have not yet shifted the handle on the stack so we must shift (to move  $YZ$  on the stack). If  $Z$  is empty then  $Y$  must be the handle so we have to reduce by  $X \rightarrow YZ$ .

(This holds provided that no conflicts occur).

$goto(I, X)$  where  $I$  is the set of all valid items for a viable prefix  $w$  is the set of all valid items for the viable prefix  $wX$ .

This will be the state transition function of the automaton.

## Goto

Let  $I$  be a set of items and  $X$  a grammar symbol.

The set  $goto(I, X)$  is the set of items we arrive at by processing the symbol  $X$ .

For each  $A \rightarrow \alpha \cdot X\beta$  in  $I$  we produce

$$A \rightarrow \alpha X \cdot \beta$$

Now we take the closure of these.

Consider the grammar

$$\begin{aligned} exp' &\rightarrow exp \\ exp &\rightarrow exp + term \\ exp &\rightarrow term \\ term &\rightarrow term * factor \\ term &\rightarrow factor \\ factor &\rightarrow (exp) \\ factor &\rightarrow \mathbf{int} \end{aligned}$$

If  $I$  is

$$\{exp' \rightarrow exp \cdot, exp \rightarrow exp \cdot + term\}$$

then  $goto(I, +)$  is what?

## The Canonical Collection of LR(0) items

To build the parsing table we must first compute the **collection of sets of LR(0) items**,  $C$ , for the augmented grammar.

- Initialize  $C$  to  $\{closure(\{S' \rightarrow \cdot S\})\}$ .
- **repeat**
  - for** each set  $I \in C$  and each grammar symbol  $X$  **do**
  - if**  $goto(I, X)$  is not empty **then**
    - $C := C \cup \{goto(I, X)\}$
- until**  $C$  is unchanged

For the grammar

$$\begin{aligned} exp' &\rightarrow exp \\ exp &\rightarrow exp + term \\ exp &\rightarrow term \\ term &\rightarrow term * factor \\ term &\rightarrow factor \\ factor &\rightarrow (exp) \\ factor &\rightarrow \mathbf{int} \end{aligned}$$

we have the collection of sets of items:

$I_0 : \text{exp}' \rightarrow \cdot \text{exp}$   
 $\text{exp} \rightarrow \cdot \text{exp} + \text{term}$   
 $\text{exp} \rightarrow \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_5 : \text{factor} \rightarrow \mathbf{int} \cdot$

$I_6 : \text{exp} \rightarrow \text{exp} + \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_1 : \text{exp}' \rightarrow \text{exp} \cdot$   
 $\text{exp} \rightarrow \text{exp} \cdot + \text{term}$

$I_7 : \text{term} \rightarrow \text{term} * \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_2 : \text{exp} \rightarrow \text{term} \cdot$   
 $\text{term} \rightarrow \text{term} \cdot * \text{factor}$

$I_8 : \text{factor} \rightarrow (\text{exp} \cdot)$   
 $\text{exp} \rightarrow \text{exp} \cdot + \text{term}$

$I_3 : \text{term} \rightarrow \text{factor} \cdot$

$I_9 : \text{exp} \rightarrow \text{exp} + \text{term} \cdot$   
 $\text{term} \rightarrow \text{term} \cdot * \text{factor}$

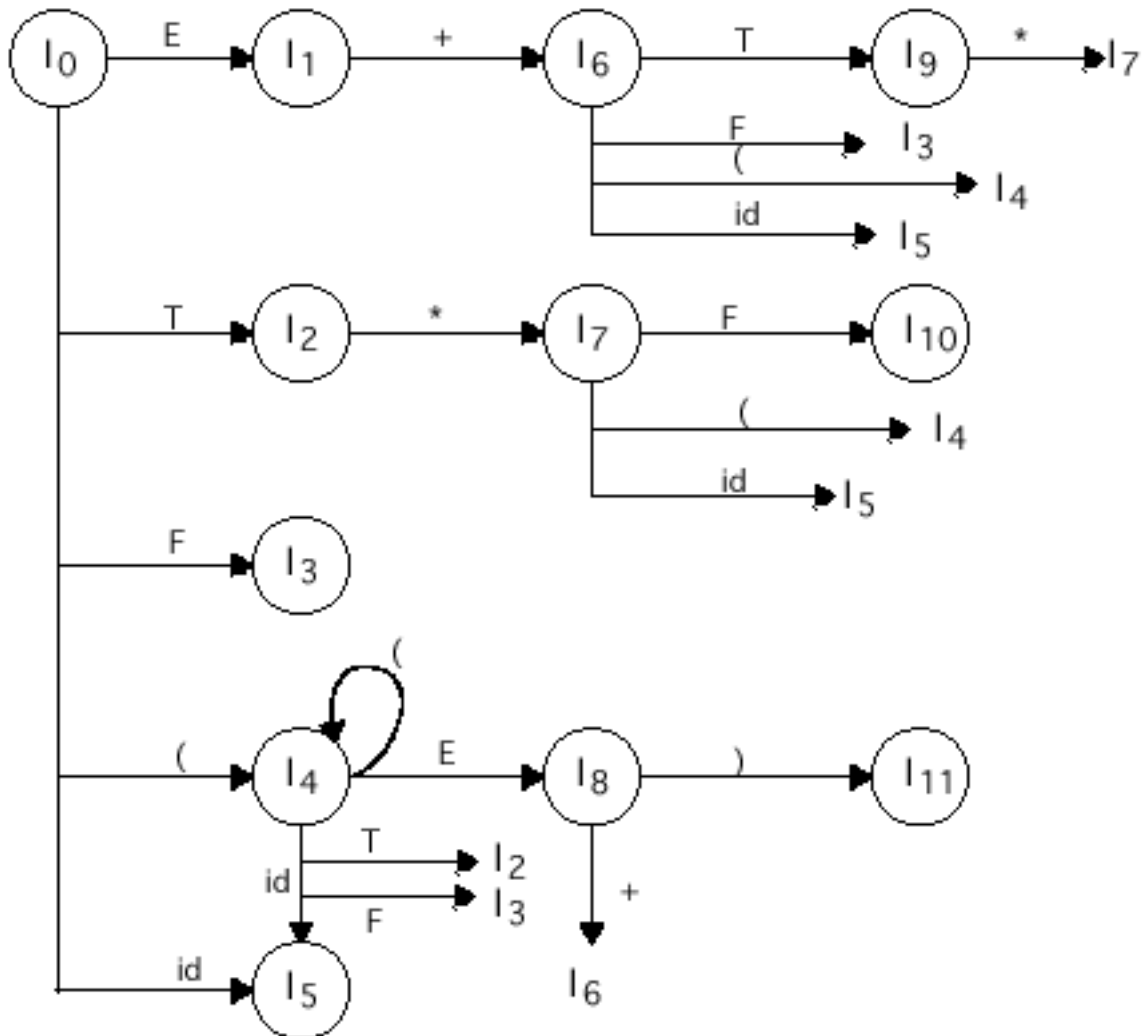
$I_4 : \text{factor} \rightarrow (\cdot \text{exp})$   
 $\text{exp} \rightarrow \cdot \text{exp} + \text{term}$   
 $\text{exp} \rightarrow \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_{10} : \text{term} \rightarrow \text{term} * \text{factor} \cdot$

$I_{11} : \text{factor} \rightarrow (\text{exp}) \cdot$

## The Characteristic Automaton

The previous grammar corresponds to the characteristic automaton below. Its states represent the canonical LR(0) item collection and the state transition function is derived from the *goto* function from above.



Abbreviations used:  $E$  is *Exp*,  $T$  is *term*,  $F$  is *factor*.

Each state of  $D$  is an end state and the state  $I_0$  (which contains the item  $S' \rightarrow \cdot S$ ) is its initial state.

If this automaton (starting in state  $I_0$ ) processes some viable prefix  $v$  it reaches a state  $I_n$  that represents exactly the valid items for  $v$ .

## Constructing the SLR Parsing Table

- Compute  $C = \{I_0, I_1, \dots, I_n\}$  the collection of sets of LR(0) items for the augmented grammar.
- Make a state  $s_i$  for each set  $I_i$  of items in  $C$ :
  1. If  $A \rightarrow \alpha \cdot a \beta$  is in  $I_i$  and  $goto(I_i, a) = I_j$  then set  $action[s_i, a]$  to *shift*  $s_j$ . Note that  $a$  must be a terminal.
  2. If  $A \rightarrow \alpha \cdot$  is in  $I_i$  then set  $action[s_i, a]$  to *reduce*  $A \rightarrow \alpha$  for all  $a \in FOLLOW(A)$ . Note that  $A$  may not be  $S'$ .
  3. If  $S' \rightarrow S \cdot$  is in  $I_i$  then set  $action[s_i, \$]$  to *accept*.
  4. For each nonterminal  $A$ , if  $goto(I_i, A) = I_j$  then set  $goto[s_i, A]$  to  $j$ .
- All other entries are set to *error*.

Let the initial parser state be the state derived from the item set  $I_i$  that contains  $S' \rightarrow S$ .

*If this method produces conflicts, the grammar is not SLR(1).*

## SLR(1) Table Example

Our example expression grammar

$$exp \rightarrow exp + term \quad (1)$$

$$exp \rightarrow term \quad (2)$$

$$term \rightarrow term * factor \quad (3)$$

$$term \rightarrow factor \quad (4)$$

$$factor \rightarrow (exp) \quad (5)$$

$$factor \rightarrow \mathbf{int} \quad (6)$$

State	Action	Goto
	id + * ( ) \$	E T F
0	s5        s4	1 2 3
1	s6            acc	
2	r2 s7        r2 r2	
3	r4 r4        r4 r4	
4	s5        s4	8 2 3
5	r6 r6        r6 r6	
6	s5        s4	9 3
7	s5        s4	10
8	s6            s11	
9	r1 s7        r1 r1	
10	r3 r3        r3 r3	
11	r5 r5        r5 r5	

We have seen how the canonical item collection is constructed.

$I_0$  generates  $action[s_0, (] = shift(s_4)$  and  $action[s_0, id] = shift(s_5)$ .

$I_1$  generates  $action[s_1, \$] = accept$  and  $action[1, +] = shift(s_6)$ .

$I_2$  generates  $action[s_2, \$] = action[s_2, +] = action[s_2, )] = reduce(E \rightarrow T)$  since  $follow(E) = \{\$, +, )\}$  and  $action[2, *] = shift(s_7)$ .

etc.

## SLR(1) Grammars and Conflicts

Steps (1)–(3) may lead to conflicting actions in which case the grammar is not SLR(1).

The following grammar, which models a C assignment, is an example of an unambiguous grammar which is **not** SLR(1).

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow *R \mid \mathbf{id} \\ R &\rightarrow L \end{aligned}$$

The LR(0) collection is

$$\begin{array}{ll} I_0 : S' \rightarrow \cdot S & I_4 : L \rightarrow * \cdot R \\ S \rightarrow \cdot L = R & R \rightarrow \cdot L \\ S \rightarrow \cdot R & L \rightarrow \cdot * R \\ L \rightarrow \cdot * R & L \rightarrow \cdot \mathbf{id} \\ L \rightarrow \cdot \mathbf{id} & \\ R \rightarrow \cdot L & I_5 : L \rightarrow \mathbf{id} \cdot \\ \\ I_1 : S' \rightarrow S \cdot & I_6 : S \rightarrow L = \cdot R \\ & R \rightarrow \cdot L \\ I_2 : S \rightarrow L \cdot = R & L \rightarrow \cdot * R \\ R \rightarrow L \cdot & L \rightarrow \cdot \mathbf{id} \\ \\ I_3 : S \rightarrow R \cdot & I_7 : L \rightarrow * R \cdot \\ \\ I_8 : R \rightarrow L \cdot & I_9 : S \rightarrow L = R \cdot \end{array}$$

The first item in  $I_2$  sets  $action[s_2, =]$  to *shift*. The second item in  $I_2$  sets  $action[s_2, =]$  to *reduce*, since  $=$  is in  $follow(R)$  should be.

This is an example of a *shift/reduce conflict*. Note that the grammar is not ambiguous.

The problem is that SLR(1) grammars are not powerful enough, performing a reduce whenever the next symbol is in the *FOLLOW* set even though contextual information might rule this out.

LALR(1) grammars work much like SLR but **remember** more about context. When setting up the parse table the LALR construction does not use the *FOLLOW* set but rather a subset of this. The grammar above is LALR(1).

## Summary

We have continued looked at bottom-up parsing:

- Constructing the LR parsing table.

## Homework

- Read Chapter 4 of Aho et al.