Formal Languages and Compilers
Lecture VI—Part 3: Syntactic Analysis

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/~artale/

Formal Languages and Compilers — BSc course

2020/21 – Second Semester

Summary of Lecture VI—Part 3

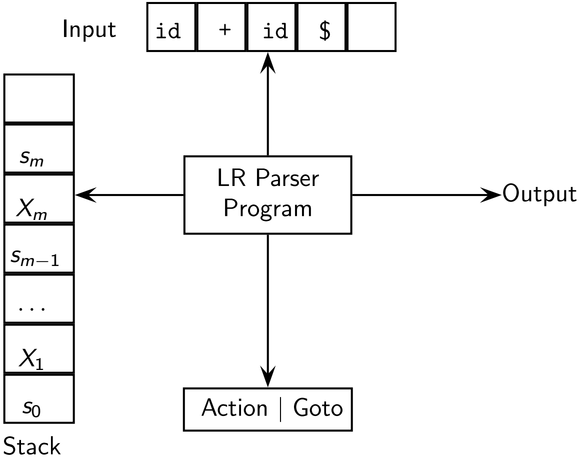
- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables

Intro to LR Parsers

- LR(k) Grammars are the most general *Non-Backtracking* Grammars that can be used in bottom-up parsers.
  - "L": Left-to-right scanning of the input;
  - "R": Rightmost derivations;
  - "k": number of lookahead symbols to take a decision.
- Predictive Grammars, i.e., LL Grammars, are a proper subset of LR Grammars (e.g., if-then-else is not LL but it is LR).
- An LR parser can detect a syntactic error as soon as possible.
- **Disadvantage.** Is difficult to build an LR parser by hand. We need specialized tools like YACC.

LR Parser Architecture

An LR parser has: An input buffer (Tokens returned from the Lexical Analyzer); A stack containing Grammar symbols and States; A parsing table with two parts, Action and Goto, implementing a DFA to decide between shift and reduce.



LR Parsing Algorithm

- The stack stores a string of the form $s_0 X_1 s_1 \ldots s_{m-1} X_m s_m$, where:
  - X_i is a Grammar Symbol;
  - s_i is a *state* summarizing the information contained in the stack below it.
- The combination $\langle$*State on top of the stack*, *Lookahead symbol*$\rangle$ is used to index the Action-Goto table.
- *Configuration of an LR Parser.* Is a pair made by the content of the stack (s_m on top) and the right-part of the input (starting with the Lookahead):

$$\langle s_0 X_1 s_1 \ldots s_{m-1} X_m s_m, a_i a_{i+1} \ldots a_n \$ \rangle$$

LR Parsing Algorithm (Cont.)

The next move of the parser is based on the pair (s_m, a_i) and on what specified in the *Action* table:

1. $action[s_m, a_i] =$ shift s_j. The parser executes a *shift* entering the configuration: $\langle s_0 X_1 s_1 \ldots X_m s_m a_i s_j,\ a_{i+1} \ldots a_n \$\rangle$.
2. $action[s_m, a_i] =$ reduce $A \to \beta$. The parser executes a *reduce* entering the configuration: $\langle s_0 X_1 s_1 \ldots X_{m-r} s_{m-r} A s,\ a_i a_{i+1} \ldots a_n \$\rangle$; where $s = goto(s_{m-r}, A)$ and $r =\mid \beta \mid$. The parser pops $2r$ symbols from the stack (r states and the r Grammar symbols β) and then pushes both A and s. The production $A \to \beta$ is in the output.
3. $action[s_m, a_i] = error$.
4. $action[s_m, \$] = accept$. The parser stops successfully.

Example: LR Parser on “id*id+id”

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			

	STACK	INPUT	ACTION
(1)	0	**id** * **id** + **id** $	shift
(2)	0 **id** 5	* **id** + **id** $	reduce by $F \to$ **id**
(3)	0 *F* 3	* **id** + **id** $	reduce by $T \to F$
(4)	0 *T* 2	* **id** + **id** $	shift
(5)	0 *T* 2 * 7	**id** + **id** $	shift
(6)	0 *T* 2 * 7 **id** 5	+ **id** $	reduce by $F \to$ **id**
(7)	0 *T* 2 * 7 *F* 10	+ **id** $	reduce by $T \to T*F$
(8)	0 *T* 2	+ **id** $	reduce by $E \to T$
(9)	0 *E* 1	+ **id** $	shift
(10)	0 *E* 1 + 6	**id** $	shift
(11)	0 *E* 1 + 6 **id** 5	$	reduce by $F \to$ **id**
(12)	0 *E* 1 + 6 *F* 3	$	reduce by $T \to F$
(13)	0 *E* 1 + 6 *T* 9	$	$E \to E+T$
(14)	0 *E* 1	$	accept

Grammar

r1. $E \to E + T$

r2. $E \to T$

r3. $T \to T * F$

r4. $T \to F$

r5. $F \to (E)$

r6. $F \to \texttt{id}$

Example: LR Parser on “id*id+id”

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			

	STACK	INPUT	ACTION
(1)	0	**id** $*$ **id** + **id** \$	shift
(2)	0 **id** 5	$*$ **id** + **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 F 3	$*$ **id** + **id** \$	reduce by $T \rightarrow F$
(4)	0 T 2	$*$ **id** + **id** \$	shift
(5)	0 T 2 $*$ 7	**id** + **id** \$	shift
(6)	0 T 2 $*$ 7 **id** 5	+ **id** \$	reduce by $F \rightarrow$ **id**
(7)	0 T 2 $*$ 7 F 10	+ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 T 2	+ **id** \$	reduce by $E \rightarrow T$
(9)	0 E 1	+ **id** \$	shift
(10)	0 E 1 + 6	**id** \$	shift
(11)	0 E 1 + 6 **id** 5	\$	reduce by $F \rightarrow$ **id**
(12)	0 E 1 + 6 F 3	\$	reduce by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	\$	$E \rightarrow E+T$
(14)	0 E 1	\$	accept

Summary

- Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables

Automata and LR Parsing

- **Definition 1.** Right-Sentential Form: A string α derived from the scope of the language, $S \Rightarrow^{*}_{rm} \alpha$, by means of right-most derivations.
- **Definition 2.** Handle: Substring of a right-sentential form that matches a right hand side of a production.
- The Handle will always appear on top of the stack, never inside.

Automata and LR Parsing (Cont.)

- The *Action* and *Goto* tables define the transition function of an Automaton that recognizes handles on top of the stack.
- The automaton does not need to read the stack every time: The state on top of the stack is the state the automaton would be after reading the symbols of the stack.
- This is why an LR parser has full control on the content of the stack just knowing the state on top of the stack.

Summary

- Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables

SLR Parsers

- Simple LR (SLR) is the simplest LR parsing Grammar.
- **Definition**. An LR(0) Item of a Grammar, `G`, is a production with a *dot* at some position in the right side.
- **Example.** The production $A \rightarrow XYZ$ gives rise to four items:

$$A \rightarrow . XYZ$$
$$A \rightarrow X . YZ$$
$$A \rightarrow XY . Z$$
$$A \rightarrow XYZ .$$

  The production $A \rightarrow \epsilon$ generates the item $A \rightarrow .$
- **Intuition.** An item indicates how much of a production we have seen in the parsing process, and can be represented by a pair of integers:

$$\langle \text{ Number of Production, Dot Position } \rangle$$

Constructing SLR Parsing Tables

- Items are useful to build the transition function of the Automaton recognizing handles.
- Items representing the same situation are grouped together into sets.
- Each of these sets represents a state of the DFA recognizing handles.
- The Canonical Collection of LR(0) Items provides the basis to construct the SLR parsing tables (implementing the DFA).
- The canonical collection is defined in terms of two operations, Closure and Goto, and an Augmented Grammar, i.e., a Grammar with a new scope S' and a new production $S' \to S$.
  - The production $S' \to S$ indicates acceptance, i.e., the parser accepts iff it is about to reduce by $S' \to S$.

Summary

- Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables

The Closure Operation

- **Algorithm.** *Closure(I)*.
  If I is a set of items for an augmented Grammar $\mathtt{G}'$, then *closure(I)* is the set of items such that:
  1. Initially every item in I is added to *closure(I)*;
  2. If $A \rightarrow \alpha . B\beta \in$ *closure(I)* and $B \rightarrow \gamma$, then we add the item $B \rightarrow . \gamma$ to *closure(I)*. Go to step 1 until no more items can be added to *closure(I)*.
- **Intuition.** $A \rightarrow \alpha . B\beta \in$ *closure(I)* indicates that:
  1. We expect to see something derivable from A, and
  2. α is already on top of the stack, thus
  3. we expect to see something derivable from $B\beta$, and then
  4. if $B \rightarrow \gamma$ we could also expect something derivable from γ.

The Closure Operation: An Example

- **Example.** Consider the augmented grammar on the left, then, $closure(\{E' \rightarrow .E\})$ contains the items shown on the right:

`Augmented Grammar`

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & (E) \\ F & \rightarrow & \mathtt{id} \end{array}$$

$closure(\{E' \rightarrow .E\})$

$$\begin{array}{lcl} E' & \rightarrow & .E \\ E & \rightarrow & .E + T \\ E & \rightarrow & .T \\ T & \rightarrow & .T * F \\ T & \rightarrow & .F \\ F & \rightarrow & .(E) \\ F & \rightarrow & .\mathtt{id} \end{array}$$

The Goto Operation

- **Definition.** If I is a set of items and $X \in \mathrm{V_N} \cup \mathrm{V_T}$, then, $goto(I, X)$ is the *closure* of the set of all items $A \to \alpha X.\beta$ such that $A \to \alpha.X\beta$ is in I.
- **Intuition 1.** $goto(I, X)$ represents the transition of the automaton from state I and input X.
- **Intuition 2.** If I is a set of items valid for a prefix α of a right-sentential form, then, $goto(I, X)$ is valid for the prefix αX.

The Goto Operation: An example

Example. If $I = \{E' \to E. \ , \ E \to E.+T\}$, then:
$goto(I, +) = closure(\{E \to E+.T\})$, is the set:

$$\begin{array}{lcl} E & \to & E+.T \\ T & \to & .T * F \\ T & \to & .F \\ F & \to & .(E) \\ F & \to & .\mathtt{id} \end{array}$$

Canonical Collection

Algorithm. *Canonical Collection for an Augmented Grammar* $\mathtt{G}'$

1. Initially, $C = \{closure(\{S' \rightarrow .S\})\}$;
2. For each set of items I in C and each Grammar symbol X
   If $goto(I, X) \neq \emptyset$ and $goto(I, X) \notin C$, then
   add $goto(I, X)$ to C;
3. Go to step 2 if new items have been added, otherwise stop.

Example: Canonical Collection for Arithmetic Expressions

$$
\begin{array}{llcl}
I_0: & E' & \rightarrow & .E \\
& E & \rightarrow & .E + T \\
& E & \rightarrow & .T \\
& T & \rightarrow & .T * F \\
& T & \rightarrow & .F \\
& F & \rightarrow & .(E) \\
& F & \rightarrow & .\texttt{id}
\end{array}
$$

$$
\begin{array}{llcl}
I_1: & E' & \rightarrow & E. \\
& E & \rightarrow & E.+T
\end{array}
$$

$$
\begin{array}{llcl}
I_2: & E & \rightarrow & T. \\
& T & \rightarrow & T.*F
\end{array}
$$

$$
\begin{array}{llcl}
I_3: & T & \rightarrow & F.
\end{array}
$$

$$
\begin{array}{llcl}
I_4: & F & \rightarrow & (.E) \\
& E & \rightarrow & .E + T \\
& E & \rightarrow & .T \\
& T & \rightarrow & .T * F \\
& T & \rightarrow & .F \\
& F & \rightarrow & .(E) \\
& F & \rightarrow & .\texttt{id}
\end{array}
$$

$$
\begin{array}{llcl}
I_5: & F & \rightarrow & \texttt{id}.
\end{array}
$$

$$
\begin{array}{llcl}
I_6: & E & \rightarrow & E+.T \\
& T & \rightarrow & .T * F \\
& T & \rightarrow & .F \\
& F & \rightarrow & .(E) \\
& F & \rightarrow & .\texttt{id}
\end{array}
$$

$$
\begin{array}{llcl}
I_7: & T & \rightarrow & T*.F \\
& F & \rightarrow & .(E) \\
& F & \rightarrow & .\texttt{id}
\end{array}
$$

$$
\begin{array}{llcl}
I_8: & F & \rightarrow & (E.) \\
& E & \rightarrow & E.+T
\end{array}
$$

$$
\begin{array}{llcl}
I_9: & E & \rightarrow & E + T. \\
& T & \rightarrow & T.*F
\end{array}
$$

$$
\begin{array}{llcl}
I_{10}: & T & \rightarrow & T * F.
\end{array}
$$

$$
\begin{array}{llcl}
I_{11}: & F & \rightarrow & (E).
\end{array}
$$

Example: Canonical Collection for Arithmetic Expressions

Augmented Grammar

$$
\begin{array}{rlcl}
\text{r0.} & E' & \rightarrow & E \\
\text{r1.} & E & \rightarrow & E + T \\
\text{r2.} & E & \rightarrow & T \\
\text{r3.} & T & \rightarrow & T * F \\
\text{r4.} & T & \rightarrow & F \\
\text{r5.} & F & \rightarrow & (E) \\
\text{r6.} & F & \rightarrow & \texttt{id}
\end{array}
$$

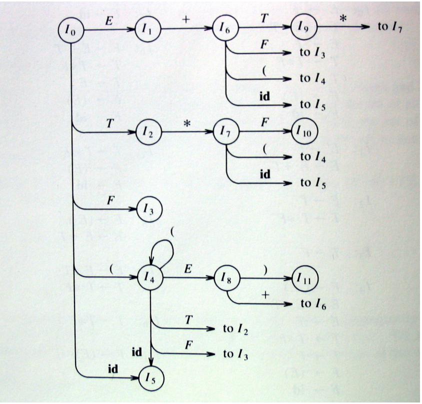
Example: Canonical Collection for Arithmetic Expressions

Example: Canonical Collection for Arithmetic Expressions

- $$\begin{array}{llll}
I_0: & E' & \rightarrow & .E \\
 & E & \rightarrow & .E + T \\
 & E & \rightarrow & .T \\
 & T & \rightarrow & .T * F \\
 & T & \rightarrow & .F \\
 & F & \rightarrow & .(E) \\
 & F & \rightarrow & .\texttt{id} \\
 & & & \\
I_1: & E' & \rightarrow & E. \\
 & E & \rightarrow & E.+T \\
 & & & \\
I_2: & E & \rightarrow & T. \\
 & T & \rightarrow & T.*F \\
 & & & \\
I_3: & T & \rightarrow & F. \\
 & & & \\
I_4: & F & \rightarrow & (.E) \\
 & E & \rightarrow & .E + T \\
 & E & \rightarrow & .T \\
 & T & \rightarrow & .T * F \\
 & T & \rightarrow & .F \\
 & F & \rightarrow & .(E) \\
 & F & \rightarrow & .\texttt{id}
\end{array}$$

$$\begin{array}{llll}
I_5: & F & \rightarrow & \texttt{id}. \\
 & & & \\
I_6: & E & \rightarrow & E+.T \\
 & T & \rightarrow & .T * F \\
 & T & \rightarrow & .F \\
 & F & \rightarrow & .(E) \\
 & F & \rightarrow & .\texttt{id} \\
 & & & \\
I_7: & T & \rightarrow & T*.F \\
 & F & \rightarrow & .(E) \\
 & F & \rightarrow & .\texttt{id} \\
 & & & \\
I_8: & F & \rightarrow & (E.) \\
 & E & \rightarrow & E.+T \\
 & & & \\
I_9: & E & \rightarrow & E + T. \\
 & T & \rightarrow & T.*F \\
 & & & \\
I_{10}: & T & \rightarrow & T * F. \\
 & & & \\
I_{11}: & F & \rightarrow & (E).
\end{array}$$

Example: Goto Function for Arithmetic Expressions



- The above figure represents the transition function of the DFA recognizing *viable prefixes* of the Grammar for Arithmetic Expressions.
- *Viable Prefix*: Prefix of right-sentential form that could be on top of the stack of an SLR Parser.

Summary

- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables

SLR Parsing Tables

Algorithm. SLR Parsing Tables *Action* and *Goto*.

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the canonical collection for the augmented grammar G'.
2. To each item set I_k we create a new state s_k. Then the *action* table is:
   - $action[s_k, a]$ = "shift s_j", if $A \to \alpha.a\beta \in I_k$, and $goto(I_k, a) = I_j$.
   - $action[s_k, a]$ = "reduce $A \to \alpha$", for all $A \to \alpha. \in I_k$, and for all a in FOLLOW(A). Here $A \neq S'$.
   - $action[s_k, \$]$ = "accept", if $S' \to S. \in I_k$.
3. $goto[s_k, A] = s_j$, if $goto(I_k, A) = I_j$.
4. All the entries not defined by rules (2) and (3) are made "error".
5. The initial state, I_0, is the one constructed from the closure of $S' \to .S$

Note. The Parsing table does not contains multiple entries if and only if the Grammar is SLR.

Summary of Lecture VI—Part 3

- LR Parsing Algorithm: An Intro
- Automata and Bottom-up Parsing
- SLR Parsing
  - Closure and Goto Operations, Canonical Collection;
  - SLR Parsing Tables