CSL302: Compiler Design

Bottom Up Parsing

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

Acknowledgement

- Today's slides are modified from that of *Stanford University:*
  - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

A Deterministic Automaton

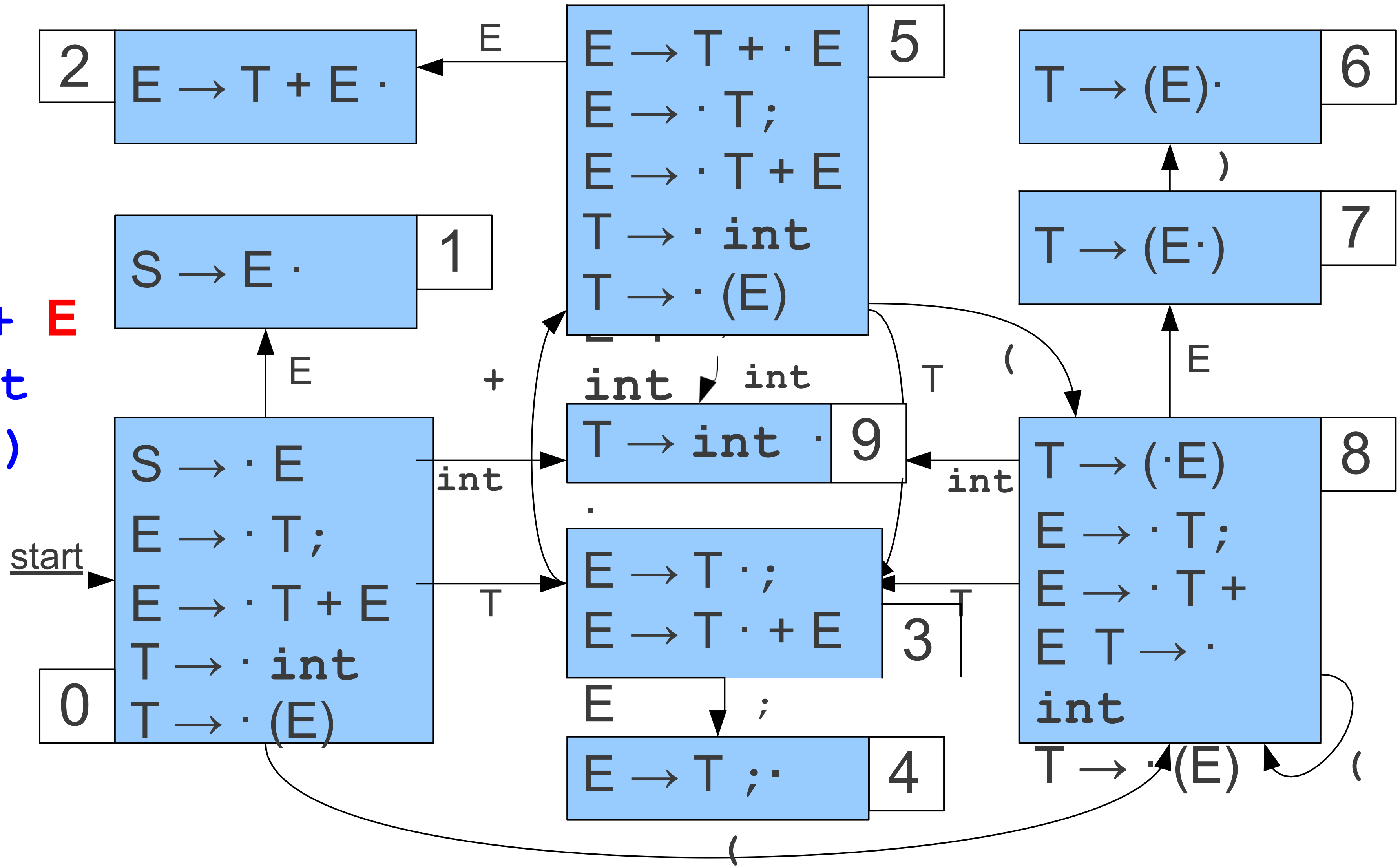
S → E

E → T;

E → T + E

T → int

T → (E)



LR(0) Tables

(1) S → E
(2) E → T;
(3) E → T + E
(4) T → int
(5) T → (E)

	Action					Goto	
	int	+	;	(	)	E	T
0	S9			S8		S1	S3
1	r1	r1	r1	r1	r1		
2	r3	r3	r3	r3	r3		
3		S5	S4				
4	r2	r2	r2	r2	r2		
5	S9			S8		S2	S3
6	r5	r5	r5	r5	r5		
7					s6		
8	S9			S8		S7	S3
9	r4	r4	r4	r4	r4		

Representing the Automaton

- LR(0) parsers are usually represented via two tables: an **action** table and a **goto** table.
- The **action** table maps each state to an action:
  - **shift**, which shifts the next terminal, and
  - **reduce** **A** $\rightarrow$ ***ω***, which performs reduction **A** $\rightarrow$ ***ω***.
  - Any state of the form **A** $\rightarrow$ ***ω*** **·** does that reduction; everything else shifts.
- The **goto** table maps state to a next state.
  - This is just the transition table for the automaton.

Why This Matters

- Our initial goal was to find handles.
- When running this automaton, if we ever end up in a state with a rule of the form

$$A \rightarrow \omega \cdot$$

- Then we might be looking at a handle.
- This automaton can be used to discover possible handle locations!

Our First Algorithm: LR(0)

- Bottom-up predictive parsing with:
  - **L**: **L**eft-to-right scan of the input.
  - **R**: **R**ightmost derivation.
  - (**0**): Zero tokens of lookahead.
- Use the handle-finding automaton, without any lookahead, to predict where handles are.

Examples

$S \rightarrow E$
$E \rightarrow T$ `;`
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

int;

Examples

$S \longrightarrow E$

$E \longrightarrow T$ `;`

$E \longrightarrow T$ `+` E

$T \longrightarrow$ `int`

$T \longrightarrow$ `(`E`)`

int+int;

LR(0) Tables

(1) $S \rightarrow E$

(2) $E \rightarrow T$ `;`

(3) $E \rightarrow T$ `+` E

(4) $T \rightarrow$ `int`

(5) $T \rightarrow$ `(E)`

	Action					Goto	
	`int`	`+`	`;`	`(`	`)`	E	T
0	S9			S8		S1	S3
1	Acc	Acc	Acc	Acc	Acc		
2	r3	r3	r3	r3	r3		
3		S5	S4				
4	r2	r2	r2	r2	r2		
5	S9			S8		S2	S3
6	r5	r5	r5	r5	r5		
7					s6		
8	S9			S8		S7	S3
9	r4	r4	r4	r4	r4		

LR(0) Tables

(1) S → E
(2) E → T;
(3) E → T + E
(4) T → int
(5) T → (E)

	Action					Goto	
	int	+	;	(	)	E	T
0	S9			S8		S1	S3
1	Acc	Acc	Acc	Acc	Acc		
2	r3	r3	r3	r3	r3		
3		S5	S4				
4	r2	r2	r2	r2	r2		
5	S9			S8		S2	S3
6	r5	r5	r5	r5	r5		
7					s6		
8	S9			S8		S7	S3
9	r4	r4	r4	r4	r4		

The LR(0) Algorithm

- Maintain a stack of (symbol, state) pairs, which is initially (`?`, 1) for some dummy symbol `?`.
- While the stack is not empty:
  - Let **state** be the top state.
  - If **action[state]** is **shift**:
    - Let `t` be the next symbol in the input.
    - Push (`t`, **goto[state, `t`]**) atop the stack.
  - If **action[state]** is **reduce A** $\to \omega$:
    - Remove $|\omega|$ symbols from the top of the stack.
    - Let **top-state** be the state on top of the stack.
    - Push (**A**, **goto[top-state, A]**) atop the stack.
  - Otherwise, report an error.

Exercise: Construct Parser Table

$E \longrightarrow T$

$T \longrightarrow T * F$

$T \longrightarrow F$

$F \longrightarrow \texttt{id}$