Module 03: CS31003: Compilers

Syntax Analysis or Parsing

Pralay Mitra
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

pralay@cse.iitkgp.ac.in
ppd@cse.iitkgp.ac.in

August 23, 24, & 30, 2021

Module Objectives

- Understand Parsing Fundamental
- Understand LR Parsing

Module Outline

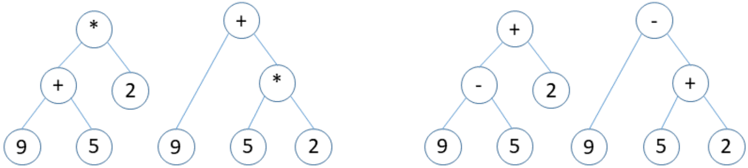
Infix $\rightarrow$ Postfix

Resolving Ambiguity by Infix $\rightarrow$ Postfix

Let us recap what we did in PDS:

```
9 + 5 * 2 =
    ((9 + 5) * 2) = 28
    (9 + (5 * 2)) = 19

9 - 5 + 2 =
    ((9 - 5) + 2) = 6
    (9 - (5 + 2)) = 2
```

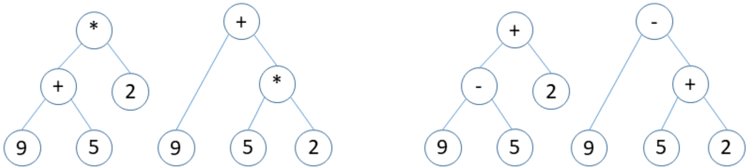


Expression Ambiguity Resolution: Infix → Postfix

```
9 + 5 * 2 = (9 + (5 * 2)) = 9 5 2 * +
            ((9 + 5) * 2) = 9 5 + 2 *

9 - 5 + 2 = (9 - (5 + 2)) = 9 5 2 + -
            ((9 - 5) + 2) = 9 5 - 2 +
```

Postfix notation is also called Reverse Polish Notation (RPN)



Associativity and Precedence

Operators

- $*, \ /$ (left)
- $+, \ -$ (left)
- $<, \ \leq, \ >, \ \geq$ (left)
- $! =, \ ==$ (left)
- $=$ (right)

Infix → Postfix: Examples

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

```
A + B * C ->
A + (B * C) ->
A (B * C) + ->
A  B  C  *  +
```

Infix → Postfix: Rules

a) Print operands as they arrive.

b) If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

c) If the incoming symbol is a left parenthesis, push it on the stack.

d) If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

e) If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

f) If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

g) If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

h) At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Operator Precedence Table

	Input						
	$\$$	$+$	$-$	$*$	$/$	$($	$)$
$\$$		$\lll$	$\lll$	$\lll$	$\lll$	$\lll$	
$+$	$\ggg$	$\ggg$	$\ggg$	$\lll$	$\lll$	$\lll$	$\ggg$
$-$	$\ggg$	$\ggg$	$\ggg$	$\lll$	$\lll$	$\lll$	$\ggg$
$*$	$\ggg$	$\ggg$	$\ggg$	$\ggg$	$\ggg$	$\lll$	$\ggg$
$/$	$\ggg$	$\ggg$	$\ggg$	$\ggg$	$\ggg$	$\lll$	$\ggg$
$($	$\lll$	$\lll$	$\lll$	$\lll$	$\lll$	$\lll$	$=$
$)$							

Infix → Postfix: Rules

- **Requires operator precedence information**
- **Operands**: Add to postfix expression.
- **Close parenthesis**: Pop stack symbols until an open parenthesis appears.
- **Operators**: Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.
- **End of input**: Pop all remaining stack symbols and add to the expression.

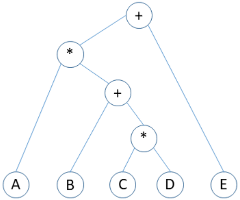
Infix → Postfix: Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +



	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Evaluating Postfix Expression

- **Create a stack to store operands (or values)**
- **Scan the given expression and do following for every scanned element**
  - If the element is a number, push it into the stack
  - If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
- **When the expression is ended, the number in the stack is the final answer**

Grammar

Grammar

$G =< T, N, S, P >$ is a (context-free) grammar where:

T	:	Set of terminal symbols
N	:	Set of non-terminal symbols
S	:	$S \in N$ is the start symbol
P	:	Set of production rules

Every production rule is of the form: $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.

Symbol convention:

$a, b, c, \cdots$	Lower case letters at the beginning of alphabet	$\in T$
$x, y, z, \cdots$	Lower case letters at the end of alphabet	$\in T^+$
$A, B, C, \cdots$	Upper case letters at the beginning of alphabet	$\in N$
$X, Y, Z, \cdots$	Upper case letters at the end of alphabet	$\in (N \cup T)$
$\alpha, \beta, \gamma, \cdots$	Greek letters	$\in (N \cup T)^*$

$G = <\{\textbf{id}, +, *, (,)\}, \{E, T, F\}, E, P>$ where P is:

$$\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}$$

Left-most Derivation of **id** + **id** * **id** $:

$$\begin{array}{llllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & \underline{T} + T\ \$ & \Rightarrow & \underline{F} + T\ \$ \\
 & \Rightarrow & \underline{\textbf{id}} + T\ \$ & \Rightarrow & \textbf{id} + \underline{T * F}\ \$ & \Rightarrow & \textbf{id} + \underline{F} * F\ \$ \\
 & \Rightarrow & \textbf{id} + \underline{\textbf{id}} * F\ \$ & \Rightarrow & \textbf{id} + \textbf{id} * \underline{\textbf{id}}\ \$ & &
\end{array}$$

Right-most Derivation of **id** + **id** * **id** $:

$$\begin{array}{llllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & E + \underline{T * F}\ \$ & \Rightarrow & E + T * \underline{\textbf{id}}\ \$ \\
 & \Rightarrow & E + \underline{F} * \textbf{id}\ \$ & \Rightarrow & E + \underline{\textbf{id}} * \textbf{id}\ \$ & \Rightarrow & \underline{T} + \textbf{id} * \textbf{id}\ \$ \\
 & \Rightarrow & \underline{F} + \textbf{id} * \textbf{id}\ \$ & \Rightarrow & \underline{\textbf{id}} + \textbf{id} * \textbf{id}\ \$ & &
\end{array}$$

Example Grammar: Derivations

$G = <\{\textbf{id}, +, *, (,)\}, \{E, T, F\}, E, P>$ where P is:

1:	E	$\rightarrow$	$E + T$
2:	E	$\rightarrow$	T
3:	T	$\rightarrow$	$T * F$
4:	T	$\rightarrow$	F
5:	F	$\rightarrow$	(E)
6:	F	$\rightarrow$	$\textbf{id}$

Left-most Derivation of **id** * **id** + **id** $:

$$\begin{array}{llllllll} E\ \$ & \Rightarrow & \underline{E} + T\ \$ & \Rightarrow & \underline{T} + T\ \$ & \Rightarrow & \underline{T} * F + T\ \$ \\ & \Rightarrow & \underline{F} * F + T\ \$ & \Rightarrow & \textbf{id} * \underline{F} + T\ \$ & \Rightarrow & \textbf{id} * \underline{\textbf{id}} + T\ \$ \\ & \Rightarrow & \textbf{id} * \textbf{id} + \underline{F}\ \$ & \Rightarrow & \textbf{id} * \textbf{id} + \underline{\textbf{id}}\ \$ \end{array}$$

Right-most Derivation of **id** * **id** + **id** $:

$$\begin{array}{llllllll} E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & E + \underline{F}\ \$ & \Rightarrow & E + \underline{\textbf{id}}\ \$ \\ & \Rightarrow & \underline{T} + \textbf{id}\ \$ & \Rightarrow & \underline{T * F} + \textbf{id}\ \$ & \Rightarrow & T * \underline{\textbf{id}} + \textbf{id}\ \$ \\ & \Rightarrow & \underline{F} * \textbf{id} + \textbf{id}\ \$ & \Rightarrow & \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$ \end{array}$$

Parsing Fundamentals

Derivation	**Parsing**	**Parser**	**Remarks**
Left-most	Top-Down	Predictive: Recursive Descent, LL(1)	No Ambiguity No Left-recursion Tool: Antlr
Right-most	Bottom-Up	Shift-Reduce: SLR, LALR(1), LR(1)	Ambiguity okay Left-recursion okay Tool: YACC, Bison

RD Parsers

Recursive Descent Parser

$$\begin{array}{lcl} S & \rightarrow & c\ A\ d \\ A & \rightarrow & ab \mid a \end{array}$$

```
int main() {
    l = getchar();
    S(); // S is a start symbol

    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'b') { // Look-ahead for decision
        match('b');
    }
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad\$ ($S \Rightarrow cAd \Rightarrow cad$), cabd\$ ($S \Rightarrow cAd \Rightarrow cabd$), caad\$

Recursive Descent Parser

$$
\begin{array}{lcl}
S & \rightarrow & c\ A\ d \\
A & \rightarrow & aAb \mid a
\end{array}
$$

```
int main() {
    l = getchar();
    S(); // S is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'a') { // Look-ahead for decision
        A();
        match('b');
    }
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad$ ($S \Rightarrow cAd \Rightarrow cad$), cabd$, caabd$ ($S \Rightarrow cAd \Rightarrow caAbd \Rightarrow caabd$)

Recursive Descent Parser

$$\begin{array}{lcl} E & \rightarrow & a\,E' \\ E' & \rightarrow & +\,a\,E' \mid \epsilon \end{array}$$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.
    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$') printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E, as per the given production
    match('a');
    E'();
}
E'() { // Definition of E' as per the given production
    if (l == '+') { // Look-ahead for decision
        match('+');
        match('a');
        E'();
    }
    else return (); // epsilon production
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a$ ($E \Rightarrow aE' \Rightarrow a$), a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a$), a+a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a + aE' \Rightarrow a + a + a$)

Recursive Descent Parser

$$E \rightarrow E + E \mid a$$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E as per the given production
    if (l == 'a') { // Terminate ? -- Look-ahead does not work
        match('a');
    }

    E();            // Call ?
    match('+');
    E();
}
match(char t) { // Match function - matches and consumes
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a+a$, a+a+a$

Curse or Boon 1: Left-Recursion

A grammar is left-recursive iff there exists a non-terminal A that can derive to a sentential form with itself as the leftmost symbol. Symbolically,

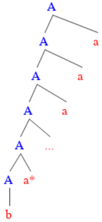
$$A \quad \Rightarrow^{+} \quad A\alpha$$

We cannot have a recursive descent or predictive parser (with left-recursion in the grammar) because we do not know how long should we recur without consuming an input

Curse or Boon 1: Left-Recursion

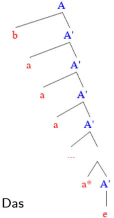
Note that, $\begin{array}{lll} A & \rightarrow & A\alpha \\ A & \rightarrow & \beta \end{array}$ leads to:

$$\begin{array}{lllllllll} A\ \$ & \Rightarrow & A\alpha\ \$ & \Rightarrow & A\alpha\alpha\ \$ & \Rightarrow & A\alpha\alpha\alpha\ \$ & \cdots \\ & \Rightarrow & A\alpha^*\ \$ & \Rightarrow & \beta\alpha^*\ \$ & & & \end{array}$$



Removing left-recursion $\begin{array}{lllll} A & \rightarrow & \beta A' & & \\ A' & \rightarrow & \alpha A' & \mid & \epsilon \end{array}$ leads to:

$$\begin{array}{lllllllll} A\ \$ & \Rightarrow & \beta A'\ \$ & \Rightarrow & \beta\alpha A'\ \$ & \Rightarrow & \beta\alpha\alpha A'\ \$ & \cdots \\ & \Rightarrow & \beta\alpha^* A'\ \$ & \Rightarrow & \beta\alpha^*\ \$ & & & \end{array}$$



Left-Recursive Example

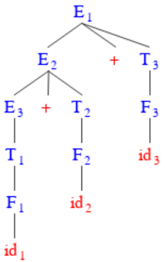
Grammar G_1 before Left-Recursion Removal

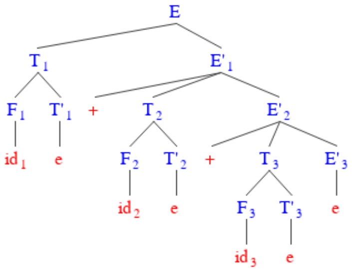
1:	E	$\rightarrow$	$E + T$
2:	E	$\rightarrow$	T
3:	T	$\rightarrow$	$T * F$
4:	T	$\rightarrow$	F
5:	F	$\rightarrow$	(E)
6:	F	$\rightarrow$	**id**

Grammar G_2 after Left-Recursion Removal

1:	E	$\rightarrow$	$T\ E'$		
2:	E'	$\rightarrow$	$+\ T\ E'$	$\mid$	ϵ
3:	T	$\rightarrow$	$F\ T'$		
4:	T'	$\rightarrow$	$*\ F\ T'$	$\mid$	ϵ
5:	F	$\rightarrow$	(E)		
6:	F	$\rightarrow$	**id**		

- These are syntactically equivalent. But what happens semantically?
- Can left recursion be effectively removed?
- What happens to Associativity?





Curse or Boon 2: Ambiguous Grammar

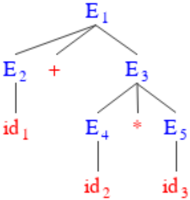
$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \textbf{id}
\end{array}
$$

- Ambiguity simplifies. But, ...
  - Associativity is lost
  - Precedence is lost
- Can *Operator Precedence (infix → postfix)* give us a clue?

Ambiguous Derivation of **id + id * id**

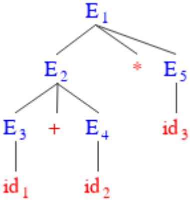
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{E * E}\ \$ \\
&\Rightarrow E + E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

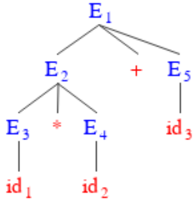
$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E + E} * \mathbf{id}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Ambiguous Derivation of **id * id + id**

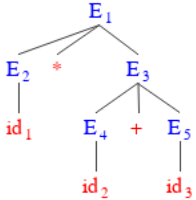
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E * E} + \mathbf{id}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{E + E}\ \$ \\
&\Rightarrow E * E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$



Remove: Ambiguity and Left-Recursion

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \textbf{id}
\end{array}
$$

Removing ambiguity:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Removing left-recursion:

$$
\begin{array}{rlll}
1: & E & \rightarrow & T\ E' \\
2|3: & E' & \rightarrow & +\ T\ E' \mid \epsilon \\
4: & T & \rightarrow & F\ T' \\
5|6: & T' & \rightarrow & *\ F\ T' \mid \epsilon \\
7: & F & \rightarrow & (E) \\
8: & F & \rightarrow & \textbf{id}
\end{array}
$$

LR Parsers

Shift-Reduce Parser: Example: Grammar

Sample grammar G_1:

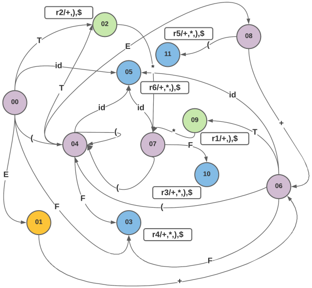
$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Shift-Reduce Parser: Example: Parse Table

State	Action						GO TO		
	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			

Shift-Reduce Parser: Example: Parsing **id** $*$ **id** $+$ **id**

Step	Stack	Symbols	Input	Act.
(1)	0		**id** * **id** + **id** $	s5
(2)	0 5	**id**	* **id** + **id** $	r6
(3)	0 3	F	* **id** + **id** $	r4
(4)	0 2	T	* **id** + **id** $	s7
(5)	0 2 7	T *	**id** + **id** $	s5
(6)	0 2 7 5	T * **id**	+ **id** $	r6
(7)	0 2 7 10	$T * F$	+ **id** $	r3
(8)	0 2	T	+ **id** $	r2
(9)	0 1	E	+ **id** $	s6
(10)	0 1 6	E +	**id** $	s5
(11)	0 1 6 5	E + **id**	$	r6
(12)	0 1 6 3	$E + F$	$	r4
(13)	0 1 6 9	$E + T$	$	r1
(14)	0 1	E	$	acc



1: $E \rightarrow E + T$
2: $E \rightarrow T$
3: $T \rightarrow T * F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow$ **id**

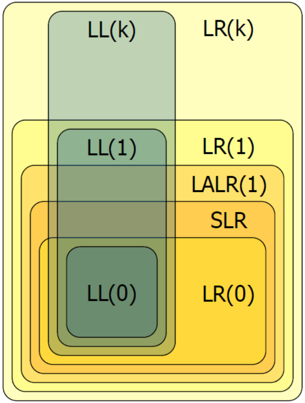
$$
\begin{aligned}
E\,\$ &\Rightarrow \underline{E + T}\,\$ \\
&\Rightarrow E + \underline{F}\,\$ \\
&\Rightarrow E + \underline{\textbf{id}}\,\$ \\
&\Rightarrow \underline{T} + \textbf{id}\,\$ \\
&\Rightarrow \underline{T * F} + \textbf{id}\,\$ \\
&\Rightarrow T * \underline{\textbf{id}} + \textbf{id}\,\$ \\
&\Rightarrow \underline{F} * \textbf{id} + \textbf{id}\,\$ \\
&\Rightarrow \underline{\textbf{id}} * \textbf{id} + \textbf{id}\,\$
\end{aligned}
$$

State	Action						GO TO		
	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			

LR Parsers

LR Fundamentals

LR Parsing: CFG Classes



- **LL(k), Top-Down, Predictive**: LL parser (Left-to-right, Leftmost derivation) with k look-ahead
- **LR(k), Bottom-Up, Shift-Reduce**: LR parser (Left-to-right, Rightmost derivation) with k look-ahead

LR Parsers

- LR parser (Left-to-right, Rightmost derivation in reverse)
- Reads input text from left to right without backing up
- Produces a rightmost derivation in reverse
- Performs bottom-up parse
- To avoid backtracking or guessing, an LR(k) parser peeks ahead at k look-ahead symbols before deciding how to parse earlier symbols. Typically k is 1.
- LR parsers are deterministic – produces a single correct parse without guesswork or backtracking
- Works in linear time
- Variants of LR parsers and generators:
  - LP(0) Parsers
  - SLR Parsers
  - LALR Parsers – Generator: Yacc (AT & T), Byacc (Berkeley Yacc)
  - Canonical LR(1) or CLR Parsers – Generator: Bison (GNU)
  - Minimal LR(1) Parsers – Generator: Hyacc (Hawaii Yacc)
  - GLR Parsers – Generator: Bison (GNU) with `%glr-parser` declaration
- Minimal LR and GLR parsers have better memory performance CLR Parsers and address reduce/reduce conflicts more effectively

Handles

- If $S \Rightarrow_{rm}^{+} \alpha$ then α is called a **right sentential form**
- A **handle** of a right sentential form is:
  - A substring β that matches the RHS of a production $A \rightarrow \beta$
  - The reduction of β to A is a step along the reverse of a rightmost derivation
- If $S \Rightarrow_{rm}^{+} \gamma A w \Rightarrow_{rm} \gamma\beta w$ where w is a sequence of tokens then
  - The substring β of $\gamma\beta w$ and the production $A \rightarrow \beta$ make the handle
- Consider the reduction of **id** * **id** + **id** to the start symbol E:

	Sentential Form	Production
	$\underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$$	$F \rightarrow \textbf{id}$
$\Rightarrow$	$\underline{F} * \textbf{id} + \textbf{id}\ \$$	$T \rightarrow F$
$\Rightarrow$	$T * \underline{\textbf{id}} + \textbf{id}\ \$$	$F \rightarrow \textbf{id}$
$\Rightarrow$	$\underline{T * F} + \textbf{id}\ \$$	$T \rightarrow T * F$
$\Rightarrow$	$\underline{T} + \textbf{id}\ \$$	$E \rightarrow T$
$\Rightarrow$	$E + \underline{\textbf{id}}\ \$$	$F \rightarrow \textbf{id}$
$\Rightarrow$	$E + \underline{F}\ \$$	$T \rightarrow F$
$\Rightarrow$	$\underline{E + T}\ \$$	$E \rightarrow E + T$
$\Rightarrow$	$E\ \$$	

- LR Parsing is about *Handle Pruning* – Start with the sentence, identify handle, reduce – till the start-symbol is reached

LR Parsers

- An LR parser is a DPDA having:
  - An Input Buffer
  - A Stack of Symbols – terminals as well as non-terminals
  - A DFA that has four types of actions:
    - **Shift** – Target state on input symbol
    - **Reduce** – Production rule and Target state on non-terminal on reduction (GOTO actions)
    - **Accept** – Successful termination of parsing
    - **Reject** – Failure termination of parsing
- The parser operates by:
  - Shifting tokens onto the stack
  - When a handle β is on top of stack, parser reduces β to LHS of production
  - Parsing continues until an error is detected or input is reduced to start symbol
- Designing an LR Parser is all about designing its DFA and actions

LR Parsers

LR(0) Parser

FIRST and FOLLOW

- $FIRST(\alpha)$, where α is any string of grammar symbols, is defined to be the set of terminals that begin strings derived from α. If $\alpha \Rightarrow^* \epsilon$, then ϵ is also in $FIRST(\alpha)$. *Examples*:
  - Given $S \rightarrow 0|A$, $A \rightarrow AB|1$, $B \rightarrow 2$;
    $FIRST(B) = \{2\}$, $FIRST(A) = \{1\}$, $FIRST(S) = \{0, 1\}$
  - Given $E \rightarrow E + E|E * E|(E)|\text{id}$;
    $FIRST(E) = \{\text{id}, (\}$
  - Given $B \rightarrow A, A \rightarrow Ac|Aad|bd|\epsilon$;
    $FIRST(B) = FIRST(A) = \{\epsilon, a, b, c\}$
- $FOLLOW(A)$, for non-terminal A, is defined to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$, for some α and β. \$ can also be in the $FOLLOW(A)$. *Examples*:
  - Given $E \rightarrow E + E|E * E|(E)|\text{id}$;
    $FOLLOW(E) = \{+, *,), \$\}$
  - Given $B \rightarrow A, A \rightarrow Ac|Aad|bd|\epsilon$;
    $FOLLOW(B) = \{\$\}$, $FOLLOW(A) = \{a, c, \$\}$

LR(0) Parser Construction

- LR(0) grammars can be parsed looking only at the stack
- Making shift/reduce decisions without any look-ahead token
- Based on the idea of an item or a configuration
- An LR(0) item consists of a production and a dot •

$$A \rightarrow X_1 \cdots X_i \bullet X_{i+1} \cdots X_n$$

- The dot symbol • may appear anywhere on the right-hand side
  - Marks how much of a production has already been seen
  - $X_1 \cdots X_i$ appear on top of the stack
  - $X_{i+1} \cdots X_n$ are still expected to appear
- An LR(0) state is a set of LR(0) items
  - It is the set of all items that apply at a given point in parse

LR(0) Parser Construction

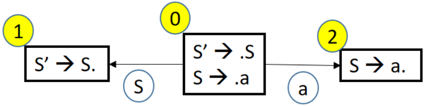
Sample Grammar, G_6				**Augmented Grammar, G_6**			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	x	1:	S	$\rightarrow$	x
2:	S	$\rightarrow$	(L)	2:	S	$\rightarrow$	(L)
3:	L	$\rightarrow$	S	3:	L	$\rightarrow$	S
4:	L	$\rightarrow$	L, S	4:	L	$\rightarrow$	L, S

- **LR(0) Item**: An LR (0) item is a production in G with dot at some position on the right side of the production. *Examples*: $S \rightarrow .(L)$, $S \rightarrow (.L)$, $S \rightarrow (L.)$, $S \rightarrow (L).$
- **Closure**: Add all items arising from the productions from the non-terminal after the period in an item. Closure is computed transitively. *Examples*:
  - Closure($S \rightarrow .(L)$) = $\{S \rightarrow .(L)\}$
  - Closure($S \rightarrow (.L)$) = $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **State**: Collection of LR(0) items and their closures. *Examples*:
  - $\{S' \rightarrow .S, S \rightarrow .x, S \rightarrow .(L)\}$
  - $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **Actions**: Shift (s#), Reduce (r#), Accept (acc), Reject (' '), GOTO (#):
  - Shift on input symbol to state# (dot precedes the terminal to shift)
  - Reduction on all input symbols by production# (dot at the end of a production)
  - Accept on reduction by the augmented production $S' \rightarrow S$
  - Reject for blank entries – cannot be reached for a valid string
  - GOTO on transition of non-terminal after reduction (dot precedes the non-terminal to reduce to)

Intuitive LR Parser Construction

- $G_3 = \{S \rightarrow a\}$

LR(0) Parser:
S' → S
S → a
L(G) = {a}



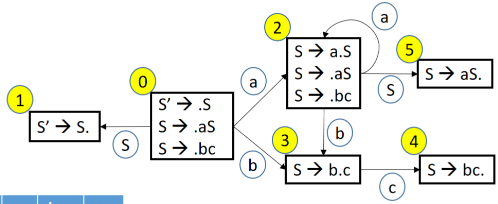
State	a	$	S
0	s2		1
1		Acc	
2	r1	r1	

S' → S → a$

Intuitive LR Parser Construction

- $G_4 = \{S \rightarrow aS|bc\}$

LR(0) Parser:
S' → S
S → aS | bc
L(G) = {a*bc}

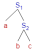
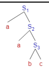
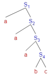
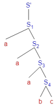


State	a	b	c	$	S
0	s2	s3			1
1				Acc	
2	s2	s3			5
3			s4		
4	r2	r2	r2	r2	
5	r1	r1	r1	r1	

S → bc$
S → aS$ → abc$
S → aS$ → aaS$ → aabc$

Intuitive LR Parser Construction

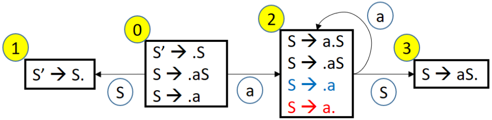
$G_4 = \{S \rightarrow \mathbf{a}\ S \mid \mathbf{b\ c}\}$. $S'\ \$ \Rightarrow S\ \$ \Rightarrow \mathbf{a}\ S\ \$ \Rightarrow \mathbf{a\ a}\ S\ \$ \Rightarrow \mathbf{a\ a\ a}\ S\ \$ \Rightarrow \mathbf{a\ a\ a\ b\ c}\ \$$

Step	Stack	Symbols	Input	Action	Parse Tree
(1)	0		**a a a b c $**	shift	
(2)	0 2	**a**	**a a b c $**	shift	
(3)	0 2 2	**a a**	**a b c $**	shift	
(4)	0 2 2 2	**a a a** S	**b c $**	shift	
(5)	0 2 2 2 3	**a a a b**	**c $**	shift	
(6)	0 2 2 2 3 4	**a a a** $\underline{\mathbf{b\ c}}$	**$**	reduce by $S \rightarrow \mathbf{b\ c}$	
(7)	0 2 2 2 5	**a a** $\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(8)	0 2 2 5	**a** $\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(9)	0 2 5	$\underline{\mathbf{a}\ S}$	**$**	reduce by $S \rightarrow \mathbf{a}\ S$	
(10)	1	$\underline{S}$	**$**	accept	

Intuitive LR Parser Construction

- $G_5 = \{S \rightarrow aS|a\}$

SLR(1) Parser:
S' → S
S → aS | a
L(G) = {a+}



State	a	$	S
0	s2		1
1		Acc	
2	s2/r2	r2	3
3	r1	r1	

S → a$
S → aS$ → aa$
S → aS$ → aaS$ → aaa$

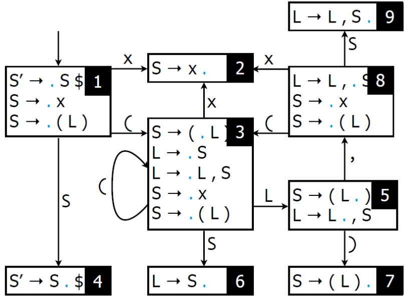
LR(0) Parser Construction

Sample Grammar, G_6				**Augmented Grammar, G_6**			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	x	1:	S	$\rightarrow$	x
2:	S	$\rightarrow$	(L)	2:	S	$\rightarrow$	(L)
3:	L	$\rightarrow$	S	3:	L	$\rightarrow$	S
4:	L	$\rightarrow$	L, S	4:	L	$\rightarrow$	L, S

- **LR(0) Item**: An LR (0) item is a production in G with dot at some position on the right side of the production. *Examples*: $S \rightarrow .(L)$, $S \rightarrow (.L)$, $S \rightarrow (L.)$, $S \rightarrow (L).$
- **Closure**: Add all items arising from the productions from the non-terminal after the period in an item. Closure is computed transitively. *Examples*:
  - Closure($S \rightarrow .(L)$) = $\{S \rightarrow .(L)\}$
  - Closure($S \rightarrow (.L)$) = $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **State**: Collection of LR(0) items and their closures. *Examples*:
  - $\{S' \rightarrow .S, S \rightarrow .x, S \rightarrow .(L)\}$
  - $\{S \rightarrow (.L), L \rightarrow .S, L \rightarrow .L, S, S \rightarrow .x, S \rightarrow .(L)\}$
- **Actions**: Shift (s#), Reduce (r#), Accept (acc), Reject (' '), GOTO (#):
  - Shift on input symbol to state# (dot precedes the terminal to shift)
  - Reduction on all input symbols by production# (dot at the end of a production)
  - Accept on reduction by the augmented production $S' \rightarrow S$
  - Reject for blank entries – cannot be reached for a valid string
  - GOTO on transition of non-terminal after reduction (dot precedes the non-terminal to reduce to)

LR(0) Parser Example

$$
\begin{array}{llcl}
0: & S' & \rightarrow & S \\
1: & S & \rightarrow & x \\
2: & S & \rightarrow & (L) \\
3: & L & \rightarrow & S \\
4: & L & \rightarrow & L, S
\end{array}
$$

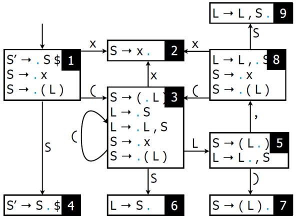


	(	)	x	,	$	S	L
1	s 3		s 2			g 4	
2	r 1	r 1	r 1	r 1	r 1		
3	s 3		s 2			g 6	g 5
4					a		
5		s 7		s 8			
6	r 3	r 3	r 3	r 3	r 3		
7	r 2	r 2	r 2	r 2	r 2		
8	s 3		s 2			g 9	
9	r 4	r 4	r 4	r 4	r 4		

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR(0) Parser Example: Parsing (x , x) $

	(	)	x	,	$	S	L
1	s 3		s 2			g 4	
2	r 1	r 1	r 1	r 1	r 1		
3	s 3		s 2			g 6	g 5
4					a		
5		s 7		s 8			
6	r 3	r 3	r 3	r 3	r 3		
7	r 2	r 2	r 2	r 2	r 2		
8	s 3		s 2			g 9	
9	r 4	r 4	r 4	r 4	r 4		



Step	Stack	Symbols	Input	Action
(1)	1		(x , x) $	shift
(2)	1 3	(	x , x) $	shift
(3)	1 3 2	(x	, x) $	reduce by $S \rightarrow$ x
(4)	1 3 6	(S	, x) $	reduce by $L \rightarrow S$
(5)	1 3 5	(L	, x) $	shift
(6)	1 3 5 8	(L ,	x) $	shift
(7)	1 3 5 8 2	(L , x	) $	reduce by $S \rightarrow$ x
(8)	1 3 5 8 9	(L , S	) $	reduce by $L \rightarrow L$, S
(9)	1 3 5	(L	) $	shift
(10)	1 3 5 7	(L)	$	reduce by $S \rightarrow$ (L)
(11)	1 4	S	$	accept

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR(0) Parser: Practice Example

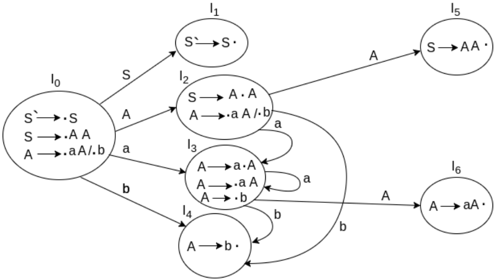
Construct an LR(0) parser for G_7:

$$
\begin{array}{llll}
1: & S & \rightarrow & A\ A \\
2: & A & \rightarrow & a\ A \\
3: & A & \rightarrow & b
\end{array}
$$

LR(0) Parser: Practice Example: Solution

Construct an LR(0) parser for G_7:

$$\begin{array}{llll} 1: & S & \rightarrow & A\ A \\ 2: & A & \rightarrow & a\ A \\ 3: & A & \rightarrow & b \end{array}$$



State	Action			GO TO	
	a	b	$	A	S
0	s3	s4		2	1
1			acc		
2	s3	s4		5	
3	s3	s4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

Source: https://www.javatpoint.com/canonical-collection-of-lr-0-items

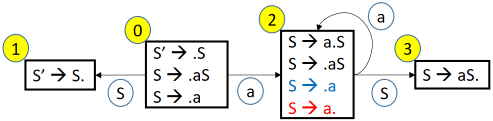
LR Parsers

SLR(1) Parser

LR(0) Parser: Shift-Reduce Conflict

- $G_5 = \{S \rightarrow aS|a\}$

SLR(1) Parser:
S' → S
S → aS | a
L(G) = {a+}

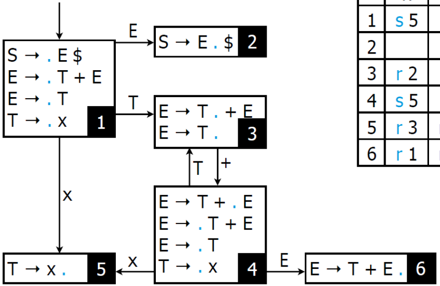


State	a	$	S
0	s2		1
1		Acc	
2	s2/r2	r2	3
3	r1	r1	

S → a$
S → aS$ → aa$
S → aS$ → aaS$ → aaa$

- Consider State 2.
  - By $S \rightarrow .a$, we should shift on a and remain in state 2
  - By $S \rightarrow a.$, we should reduce by production 2
- We have a Shift-Reduce Conflict
- As $FOLLOW(S) = \{\$\}$, we decide in favor of shift. Why?

LR(0) Parser: Shift-Reduce Conflict



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

E → T + E
E → T
T → x

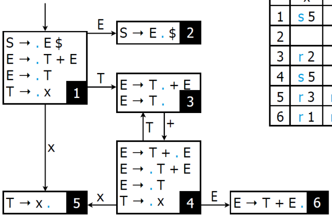
- Consider State 3.
  - By $E \to T. + E$, we should shift on $+$ and move to state 4
  - By $E \to T.$, we should reduce by production 2
- We have a Shift-Reduce Conflict
- To resolve, we build SLR(1) Parser

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

SLR(1) Parser Construction

- **LR(0) Item**: Canonical collection of LR(0) Items used in SLR(1) as well
- **Closure**: Same way as LR(0)
- **State**: Collection of LR(0) items and their closures.
- **Actions**: Shift (s#), Reduce (r#), Accept (acc), Reject (<space>), GOTO (#):
  - Shift on input symbol to state#
  - **Reduction by production# only on the input symbols that belong to the FOLLOW of the left-hand side**
  - Accept on reduction by the augmented production
  - GOTO on transition of non-terminal after reduction

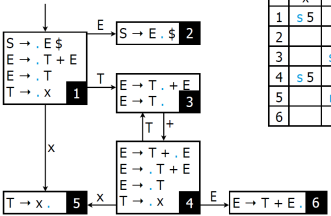
SLR Parse Table: Shift-Reduce Conflict on LR(0)



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

E → T + E
E → T
T → x

Reduce a production $S \rightarrow \ldots$ on symbols $k \in T$ if $k \in Follow(S)$



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

SLR(1) Parser: Practice Example

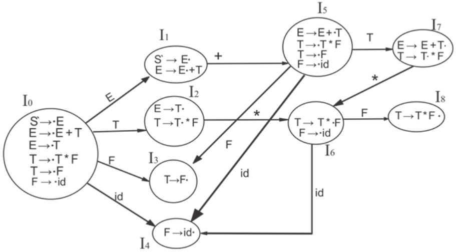
Construct an SLR(1) parser for G_8:

$$
\begin{array}{llcl}
1: & S & \rightarrow & E \\
2: & E & \rightarrow & E + T \\
3: & E & \rightarrow & T \\
4: & T & \rightarrow & T * F \\
5: & T & \rightarrow & F \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

SLR(1) Parser: Practice Example: Solution

Construct an SLR(1) parser for G_8:

1:	$S \rightarrow E$		4:	$T \rightarrow T * F$
2:	$E \rightarrow E + T$		5:	$T \rightarrow F$
3:	$E \rightarrow T$		6:	$F \rightarrow$ **id**



States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Source: https://www.javatpoint.com/slr-1-parsing

LR Parsers

LR(1) Parser

SLR(1) Parser: Shift-Reduce Conflict

Grammar G_9

1:	S	$\rightarrow$	$L = R$
2:	S	$\rightarrow$	R
3:	L	$\rightarrow$	$*R$
4:	L	$\rightarrow$	id
5:	R	$\rightarrow$	L

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

- $= \in FOLLOW(R)$ as $S \Rightarrow L = R \Rightarrow *R = R$
- So in State#2 we have a shift/reduce Conflict on $=$
- The grammar is not ambiguous. Yet we have the shift/reduce conflict as SLR is not powerful enough to remember enough left context to decide what action the parser should take on input $=$, having seen a string reducible to L.
- To resolve, we build LR(1) Parser

Source: Dragon Book

LR(1) Parser Construction

Sample Grammar G_7				**Augmented Grammar** G_7			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	CC	1:	S	$\rightarrow$	CC
2:	C	$\rightarrow$	cC	2:	C	$\rightarrow$	cC
3:	C	$\rightarrow$	d	3:	C	$\rightarrow$	d

- **LR(1) Item**: An LR(1) item has the form $[A \rightarrow \alpha.\beta, a]$ where $A \rightarrow \alpha\beta$ is a production and a is the look-ahead symbol which is a terminal or \$. As the dot moves through the right-hand side of the production, token a remains attached to it. LR(1) item $[A \rightarrow \alpha., a]$ calls for a reduce action when the look-ahead is a. *Examples*: $[S \rightarrow .CC, \$]$, $[S \rightarrow C.C, \$]$, $[S \rightarrow CC., \$]$
- **Closure(S)**:

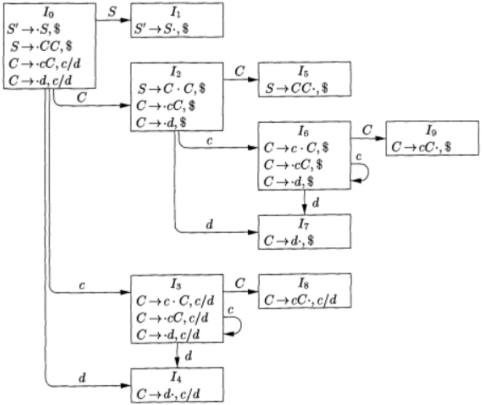
  For each item $[A \rightarrow \alpha.B\beta, t] \in S$,
  For each production $B \rightarrow \gamma \in G$,
  For each token $b \in FIRST(\beta t)$,
  Add $[B \rightarrow .\gamma, b]$ *to* S

  Closure is computed transitively. *Examples*:
  - Closure($[S \rightarrow C.C, \$]$) = $\{[S \rightarrow C.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$
  - Closure($[C \rightarrow c.C, c/d]$) = $\{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$
- **State**: Collection of LR(1) items and their closures. *Examples*:
  - $\{[S \rightarrow C.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$
  - $\{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$

LR(1) Parser: Example

Construct an LR(1) parser for G_7:

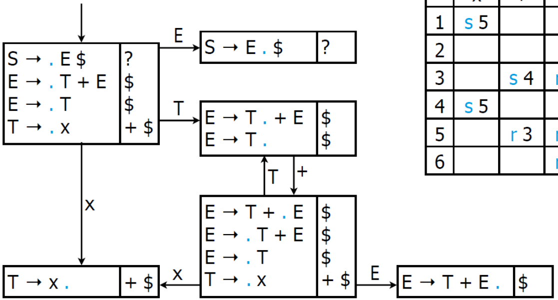
$$
\begin{array}{llll}
1: & S & \rightarrow & CC \\
2: & C & \rightarrow & cC \\
3: & C & \rightarrow & d
\end{array}
$$



STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Source: Dragon Book

LR(1) Parser: Example



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

Source: https://www.slideshare.net/eelcovisser/lr-parsing-71059803?from_action=save

LR Parsers

LALR(1) Parser

LALR(1) Parser Construction

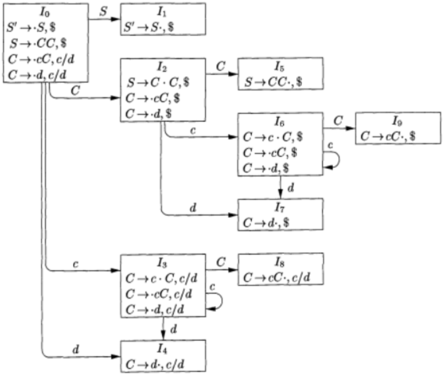
Sample Grammar G_7				**Augmented Grammar** G_7			
				0:	S'	$\rightarrow$	S
1:	S	$\rightarrow$	CC	1:	S	$\rightarrow$	CC
2:	C	$\rightarrow$	cC	2:	C	$\rightarrow$	cC
3:	C	$\rightarrow$	d	3:	C	$\rightarrow$	d

- **LR(1) States**: Construct the Canonical LR(1) parse table.
- **LALR(1) States**: Two or more LR(1) states having the same set of core LR(0) items may be merged into one by combining the look-ahead symbols for every item. Transitions to and from these merged states may also be merged accordingly. All other states and transitions are retained. *Examples*:
  - Merge
    State#3 = $\{[C \rightarrow c.C, c/d], [C \rightarrow .cC, c/d], [C \rightarrow .d, c/d]\}$ with
    State#6 = $\{[C \rightarrow c.C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]\}$ to get
    State#36 = $\{[C \rightarrow c.C, c/d/\$], [C \rightarrow .cC, c/d/\$], [C \rightarrow .d, c/d/\$]\}$
  - Merge
    State#4 = $\{[C \rightarrow d., c/d]\}$ with
    State#7 = $\{[C \rightarrow d., \$]\}$ to get
    State#47 = $\{[C \rightarrow d., c/d/\$]\}$
- **Reduce/Reduce Conflict**: LR(1) to LALR(1) transformation cannot introduce any new shift/reduce conflict. But it may introduce reduce/reduce conflict.

LALR(1) Parser: Example

Construct an LALR(1) parser for G_7:

$$\begin{array}{llll} 1: & S & \rightarrow & CC \\ 2: & C & \rightarrow & cC \\ 3: & C & \rightarrow & d \end{array}$$



STATE	ACTION c	ACTION d	ACTION $	GOTO S	GOTO C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

STATE	ACTION c	ACTION d	ACTION $	GOTO S	GOTO C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Source: Dragon Book

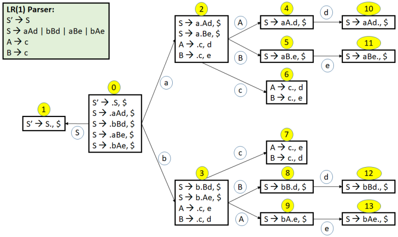
LALR(1) Parser: Reduce-Reduce Conflict

Consider $G_{10} = < \{a, b, c, d, e\}, \{S, A, B\}, S, P >$ where $P =$

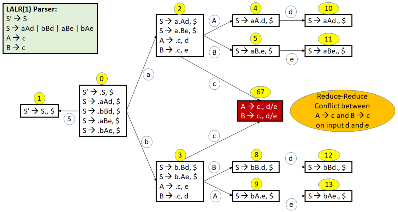
$$\begin{array}{llll} 0: & S' & \rightarrow & S \\ 1: & S & \rightarrow & aAd \\ 2: & S & \rightarrow & bBd \\ 3: & S & \rightarrow & aBe \\ 4: & S & \rightarrow & bAe \\ 5: & A & \rightarrow & c \\ 6: & B & \rightarrow & c \end{array}$$

Clearly, $L(G) = \{acd, bcd, ace, bce\}$

LR(1) Parser



LALR(1) Parser



LR Parsers: Practice Examples

Determine the LR Class (LR(0), SLR(1), LR(1) or LALR(1)) for the following grammars:

- G: $S \rightarrow aSb \mid b$
- G: $S \rightarrow Sa \mid b$
- G: $S \rightarrow (S) \mid SS \mid \epsilon$
- G: $S \rightarrow (S) \mid SS \mid ()$
- G: $S \rightarrow ddX \mid aX \mid \epsilon$
- G: $S \rightarrow E$; $E \rightarrow T + E \mid T$; $T \rightarrow int * T \mid int \mid (E)$
- G: $S \rightarrow V = E \mid E$; $E \rightarrow V$; $V \rightarrow x \mid *E$
- G: $S \rightarrow AB$; $A \rightarrow aAb \mid a$; $B \rightarrow d$

LR Parsers

LR(k) Parser

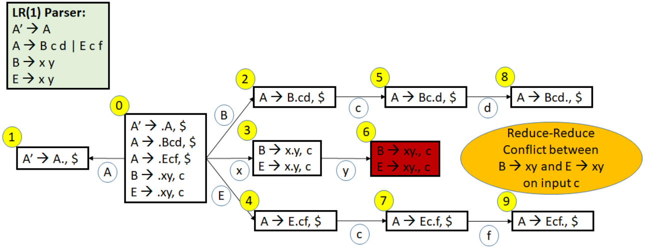
This section is not included in the examination but may be crucially important for understanding

LR(1) Parser: Shift-Reduce Conflict

Grammar G_{11}

1:	A	$\rightarrow$	$B\ c\ d$
2:	A	$\rightarrow$	$E\ c\ f$
3:	B	$\rightarrow$	$x\ y$
4:	E	$\rightarrow$	$x\ y$

- For this grammar, an example input that starts with xyc is enough to confuse an LR(1) parser, as it has to decide whether xy matches B or E after only seeing 1 symbol further (i.e. c).



- An LL(1) parser would also be confused, but at the x - should it expand A to $B\ c\ d$ or to $E\ c\ f$, as both can start with x. An LL(2) or LL(3) parser would have similar problems at the y or c respectively.
- An LR(2) parser would be able to also see the d or f that followed the c and so make the correct choice between B and E.
- An LL(4) parser would also be able to look far enough ahead to see the d or f that followed the c and so make the correct choice between expanding A to $B\ c\ d$ or to $E\ c\ f$.

Source: http://www.cs.man.ac.uk/~pjj/cs212/ho/node19.html#sec:BorE

LR(k) Parser: Shift-Reduce Conflict

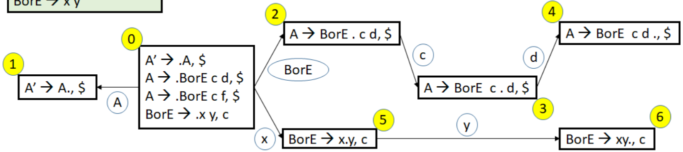
Grammar G_{12}

$$
\begin{array}{llcl}
1: & A & \rightarrow & B\ C\ d \\
2: & A & \rightarrow & E\ C\ f \\
3: & B & \rightarrow & x\ y \\
4: & E & \rightarrow & x\ y \\
5: & C & \rightarrow & C\ c \\
6: & C & \rightarrow & c
\end{array}
$$

- The grammar would confuse any LR(k) or LL(k) parser with a fixed amount of look-ahead
- To workaround, rewrite

$$
\begin{array}{llcl}
1: & A & \rightarrow & B\ C\ d \\
2: & A & \rightarrow & E\ C\ f \\
3: & B & \rightarrow & x\ y \\
4: & E & \rightarrow & x\ y
\end{array}
\quad \text{as} \quad
\begin{array}{llcl}
1: & A & \rightarrow & BorE\ c\ d \\
2: & A & \rightarrow & BorE\ c\ f \\
3: & BorE & \rightarrow & x\ y
\end{array}
$$

LR(1) Parser:
A' → A
A → BorE c d | BorE c f
BorE → x y



Source: http://www.cs.man.ac.uk/~pjj/cs212/ho/node19.html#sec:BorE