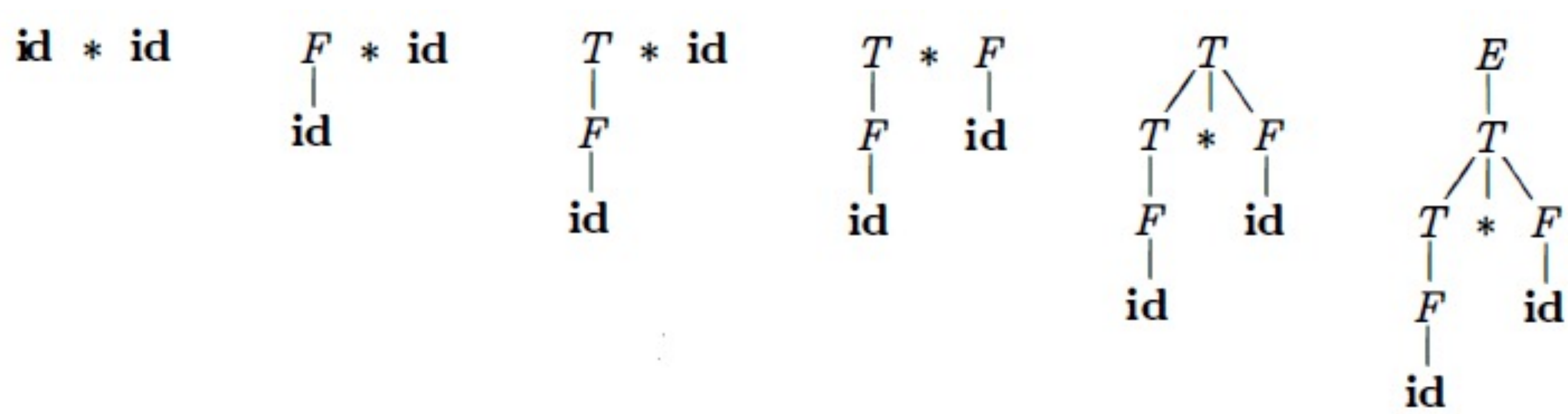
CSE 304 Compiler Design Syntax Analysis (SLR Parser)

YOUNGMIN KWON / TONY MIONE

Bottom-Up Parsing

Attempts to construct a parse tree beginning at the leaves and working up towards the root.



Bottom-up parse for id * id

Reductions

Bottom-up parsing

- Reducing a string w to the start symbol
- At each reduction step, a particular substring matching the RHS of a production is replaced by the LHS.
- Rightmost derivation is traced out in reverse.

E.g.

```
S -> aABe
A -> Abc | b
B -> d
```

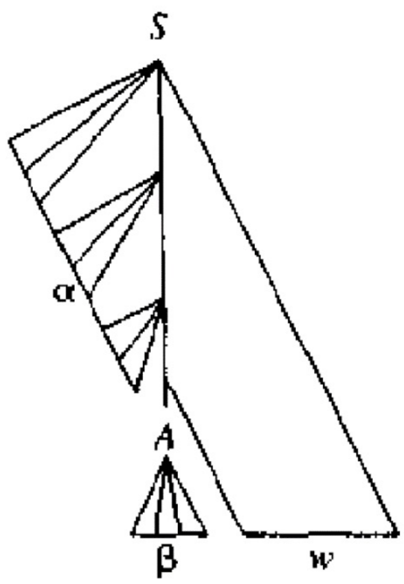
abbcde can be reduced to S

```
abbcde
aAbcde
aAde
aABe
S
```

Handle Pruning

Handle:

- A handle of a right-sentential form γ is a production A->β and a position of γ where the β may be found and replaced by A to produce the previous step of rightmost derivation.
  - If $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$, then A -> β in the position following α is a handle of α β w.
- E.g. In the previous example
  - aAbcde => abbcde, handle is A->b at position 2.
  - aAde => aAbcde, handle is A->Abc at position 2.



- Handle pruning:
  - A->β in α β w is a handle.
  - Reducing β to A can be thought as pruning the handle (removing the children of A from the parse tree).
- A Rightmost derivation in reverse can be obtained by handle pruning

Shift-Reduce Parsing

Shift-Reduce parsing

- A bottom-up parsing where a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- While scanning the input from left to right, the parser shifts 0+ input symbols onto the stack
- If it is ready to reduce the RHS of a production, pop the RHS from the stack and push the LHS to the stack.
- Handles always appear at the top of the stack

4 Actions if Shift-Reduce Parsing

- **Shift**: push the next input symbol to the stack
- **Reduce**: pop the RHS of a production and push the LHS.
- **Accept**: announce the success
- **Error**: found an error

Shift-Reduce Parsing

Why the handle is always on top of the stack?

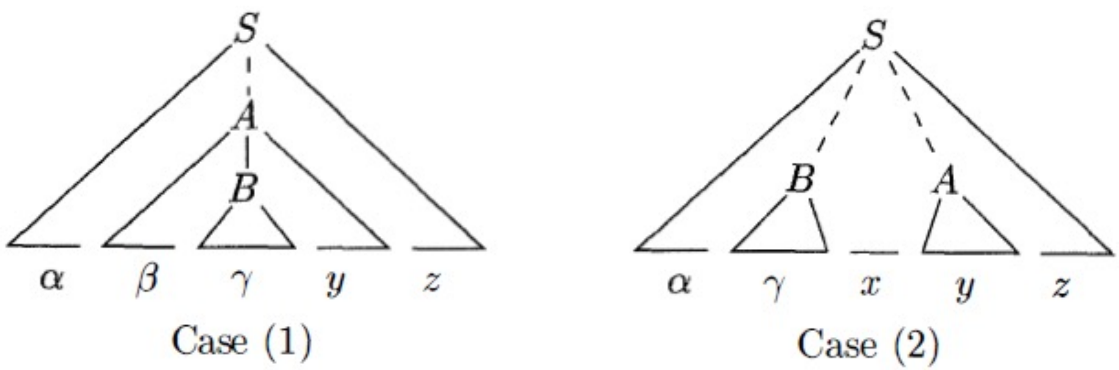
Two possible cases of two successive steps of rightmost derivation

(1) $S \Rightarrow^{*} \alpha\ A\ z \Rightarrow \alpha\ \beta\ B\ y\ z \Rightarrow \alpha\ \beta\ \gamma\ y\ z$

- A is replaced by β B y (has a nonterminal B), then B is replaced.

(2) $S \Rightarrow^{*} \alpha\ B\ x\ A\ z \Rightarrow \alpha\ B\ x\ y\ z \Rightarrow \alpha\ \gamma\ x\ y\ z$

- A is replaced by y (terminals only), then B is replaced.



Shift-Reduce Parsing

- Case 1: $S \Rightarrow^* \alpha\ A\ z \Rightarrow \alpha\ \beta\ B\ y\ z \Rightarrow \alpha\ \beta\ \gamma\ y\ z$
  - ($ α β γ | y z $): the parser reached this configuration. γ is the handle and it is reduced to B.
  - ($ α β B | y z $): since B is the rightmost nonterminal in α β B y z, the handle cannot be inside the stack.
  - ($ α β B y | z $): the parser shifted y. β B y is the handle and it gets reduced to A.
- Case 2: $S \Rightarrow^* \alpha\ B\ x\ A\ z \Rightarrow \alpha\ B\ x\ y\ z \Rightarrow \alpha\ \gamma\ x\ y\ z$
  - ($ α γ | x y z $): the parser reached this configuration. γ is the handle and it is reduced to B
  - ($ α B x y | z $): after shifting x y, get the next handle y on top of the stack and reduce it to A
  - ($ α B x A | z $): configuration after the reduction.

Shift-Reduce Parsing

Viable Prefixes

- The set of prefixes of right-sentential forms that can appear on the stack of shift-reduce parser.
- A prefix of a right-sentential form that does not continue past the right end of the rightmost handle.

LR Parsers

LR(k) Parsing:

- L: left-to-right scanning of the input.
- R: constructing the rightmost derivation in reverse.
- k: number of input symbols of lookahead.

SLR (Simple LR): easiest to implement, least powerful.

Canonical LR: most powerful, most expensive.

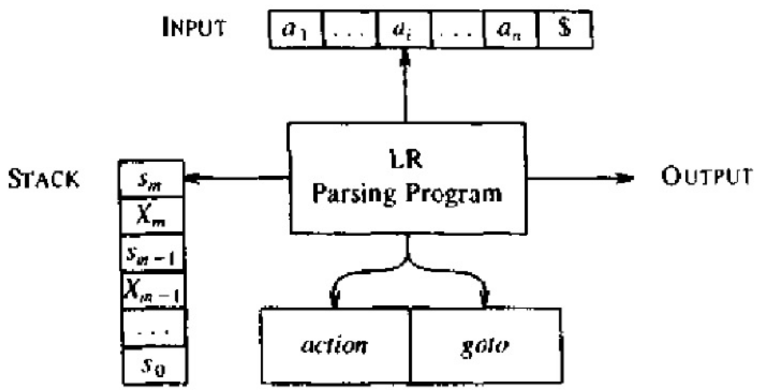
LALR (look-ahead LR): intermediate in power and cost. Work with most programming language grammars.

LR Parsing Algorithm

- Configuration
  – (s_1, X_1, s_2, X_2 … s_n | a_1, a_2, …), where s_i is a state, X_i is a symbol, a_i is a token.

4 Actions of LR parser

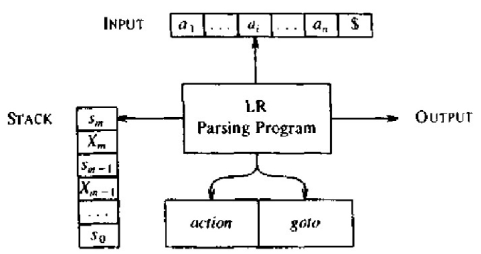
- Shift and go to state s
  - (… s_1 | a_1 a_2…) -> (… s_1 a_1 s | a_2 …)
- Reduce X -> X_1 … X_n
  - (… s_0 X_1 s_1 … X_n s_n | a_1 …) -> (… s_0 X s | a_1 …), where s is the goto target of s_0 for symbol X.
- Accept: finish with success
- Error: found an error



LR Parsing Example

Parse id * id + id

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



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			

LR Parsing Example

	Stack	Symbols	Input	Action
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

Constructing SLR Parsing Table

States of an SLR parser represent sets of items.

LR(0) items of a grammar G is a production of G with a dot at some positions of the RHS.

- E.g. `A -> XYZ: A->.XYZ, A->X.YZ,`
  `A->XY.Z, A->XYZ.`
  `A -> ϵ: A->.`
- **An item represents how much of a production we have seen**
  - `X->X.YZ` means, we've just seen a string derivable from X and expect to see a string derivable from YZ.

Augmented grammar

- Add a new start symbol S' and add a production S' -> S
- To indicate when to stop.

Constructing SLR Parsing Table

The central idea of SLR parsing is to construct a DFA recognizing the viable prefixes.

- Imagine an NFA:
  - States are the items
  - Add a transition from $A \rightarrow \alpha.X\beta$ to $A \rightarrow \alpha X.\beta$ labeled X.
  - Add a transition from $A \rightarrow \alpha.B\beta$ to $B \rightarrow .\gamma$ labeled ϵ
- Construct a DFA using the subset construction algorithm.

Canonical LR(0) items

- Give basis for the DFA states
- CLOSURE and GOTO functions can find the canonical LR(0) items.

Valid items

- Item $A \rightarrow \beta_1 . \beta_2$ is valid for a viable prefix $\alpha\ \beta_1$ if there is a derivation $S' \Rightarrow^* \alpha\ A\ w \Rightarrow \alpha\ \beta_1\ \beta_2\ w$

CLOSURE and GOTO functions

CLOSURE(I)

- If I is a set of items, CLOSURE(I) is a set of items built by the two rules
  - Add every item in I to CLOSURE(I)
  - If $A \rightarrow \alpha.B\beta\gamma$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, add $B \rightarrow .\gamma$ to CLOSURE(I). Apply this rule until no more new items are added to CLOSURE(I).
- $A \rightarrow \alpha.B\beta$ in CLOSURE(I) means, we might next see a substring derivable from $B\beta$. Hence we add $B \rightarrow .\gamma$ to CLOSURE(I).

GOTO(I,X)

- GOTO(I,X) is the closure of the set of all items $A \rightarrow \alpha X.\beta$ such that $A \rightarrow \alpha.X\beta$ is in I.
- The closures of items are the states of DFA and GOTO(I,X) specifies the transition from the state I under input X.

CLOSURE and GOTO functions

Given the augmented grammar

```
E'  -> E
E   -> E + T | T
T   -> T * F | F
F   -> (E) | id
```

CLOSURE({ E'->.E }) is

{ E'->.E, E->.E+T, E->.T, T->.T*F, T->.F,
F->.(E), F->.id }

GOTO({ E'->E., E->E.+T }, +) is

{ E->E+.T, T->.T*F, T->.F, F->.(E),
F->.id }

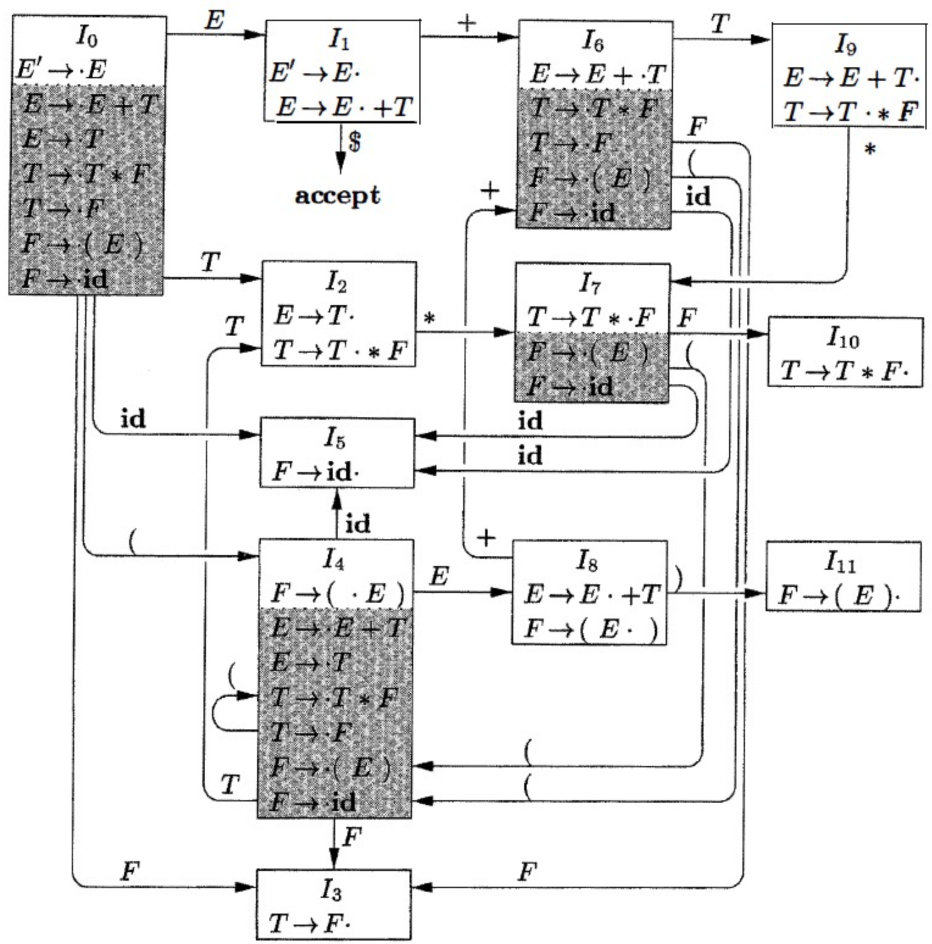
Canonical LR(0) items

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}

void items(G′) {
    C = CLOSURE({[S′ → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

DFA for viable prefixes

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$



Constructing SLR Parsing Tables

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a \beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

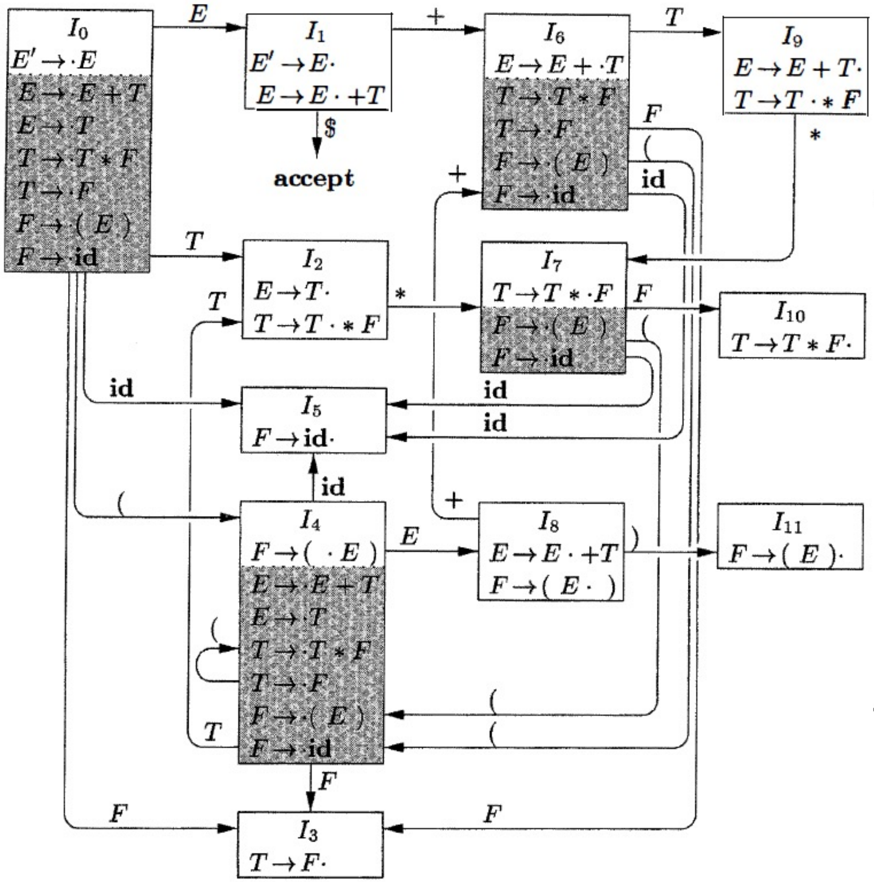
   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.



STATE	id	+	*	(	)	$	E	T	F
	ACTION						**GOTO**		
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			

$E' \to E$

(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

(6) $F \to \mathbf{id}$

```
FIRST(E′): (, id
FIRST(E) : (, id
FIRST(T) : (, id
FIRST(F) : (, id
FOLLOW(E′): $
FOLLOW(E) : +, ), $
FOLLOW(T) : +, *, ), $
FOLLOW(F) : +, *, ), $
```

Constructing SLR Parsing Tables

Example: build an SLR Parsing Table for the grammar below.

```
E -> E + id
E -> id
```

Items

I_0: E'->.E, E->.E+id, E->.id

I_1: E'->E., E->E.+id

I_2: E->id.

I_3: E->E+.id

I_4: E->E+id.

FIRST/FOLLOW

FIRST(E') = FIRST(E) = {id}

FOLLOW(E') = {$}

FOLLOW(E)= {+,$}

	+	id	$	E
0		s2		1
1	s3		acc	
2	r2			
3		s4		
4	r1		r1	