Syntax Analyzer — Parser

ASU Textbook Chapter 4.2–4.5, 4.7, 4.8

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

Main tasks

a program represented by a sequence of tokens → parser → if it is a legal program, then output some abstract representation of the program

- **Abstract representations of the input program:**
    - **abstract-syntax tree + symbol table**
    - **intermediate code**
    - **object code**
- **Context free grammar (CFG) is used to specify the structure of legal programs.**

Context free grammar (CFG)

- **Definitions:** $G = (T, N, P, S)$, **where**
  - T: **a set of terminals (in lower case letters);**
  - N: **a set of nonterminals (in upper case letters);**
  - P: **productions of the form** $A \rightarrow \alpha_1, \alpha_2, \ldots, \alpha_m$, **where** $A \in N$ **and** $\alpha_i \in T \cup N$;
  - S: **the starting nonterminal,** $S \in N$.
- **Notations:**
  - **terminals : lower case English strings, e.g.,** a, b, $c, \ldots$
  - **nonterminals: upper case English strings, e.g.,** A, B, $C, \ldots$
  - $\alpha, \beta, \gamma \in (T \cup N)^*$
    - ▷ *α, β, γ: alpha, beta and gamma.*
    - ▷ *ϵ: epsilon.*
  - $$\left.\begin{array}{ccc} A & \rightarrow & \alpha_1 \\ A & \rightarrow & \alpha_2 \end{array}\right\} \equiv A \rightarrow \alpha_1 \mid \alpha_2$$

How does a CFG define a language?

- **The language defined by the grammar is the set of strings (sequence of terminals) that can be "derived" from the starting nonterminal.**
- **How to "derive" something?**
  - **Start with:**
    "current sequence" $=$ the starting nonterminal.
  - **Repeat**
    - ▷ *find a nonterminal X in the current sequence*
    - ▷ *find a production in the grammar with X on the left of the form $X \rightarrow \alpha$, where α is ϵ or a sequence of terminals and/or nonterminals.*
    - ▷ *create a new "current sequence" in which α replaces X*
  - **Until "current sequence" contains no nonterminals.**
- **We derive either ϵ or a string of terminals. This is how we derive a string of the language.**

Example

Grammar:

- $E \rightarrow int$
- $E \rightarrow E - E$
- $E \rightarrow E \ / \ E$
- $E \rightarrow (\ E \ )$

$$
\begin{aligned}
& E \\
& \Longrightarrow E - E \\
& \Longrightarrow 1 - E \\
& \Longrightarrow 1 - E/E \\
& \Longrightarrow 1 - E/2 \\
& \Longrightarrow 1 - 4/2
\end{aligned}
$$

- **Details:**
  - **The first step was done by choosing the second production.**
  - **The second step was done by choosing the first production.**
  - $\cdots$
- **Conventions:**
  - $\Longrightarrow$**: means "derives in one step";**
  - $\overset{+}{\Longrightarrow}$**: means "derives in one or more steps";**
  - $\overset{*}{\Longrightarrow}$**: means "derives in zero or more steps";**
  - **In the above example, we can write** $E \overset{+}{\Longrightarrow} 1 - 4/2$**.**

Language

- The language defined by a grammar G is

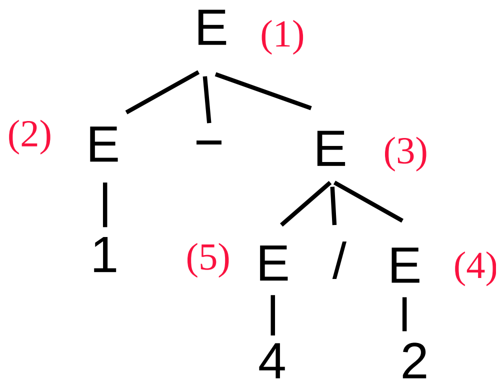
$$L(G) = \{w \mid S \overset{+}{\Longrightarrow} \omega\},$$

  where S is the starting nonterminal and ω is a sequence of terminals or ϵ.
- An element in a language is ϵ or a sequence of terminals in the set defined by the language.
- More terminology:
  - $E \Longrightarrow \cdots \Longrightarrow 1 - 4/2$ is a derivation of $1 - 4/2$ from E.
  - There are several kinds of derivations that are important:
    - *The derivation is a leftmost one if the leftmost nonterminal always gets to be chosen (if we have a choice) to be replaced.*
    - *It is a rightmost one if the rightmost nonterminal is replaced all the times.*

A way to describe derivations

- Construct a derivation or parse tree as follows:
  - start with the starting nonterminal as a single-node tree
  - REPEAT
    - ▷ *choose a leaf nonterminal* X
    - ▷ *choose a production* $X \rightarrow \alpha$
    - ▷ *symbols in* α *become the children of* X
  - UNTIL no more leaf nonterminal left
- Need to annotate the order of derivation on the nodes.

$$
\begin{array}{l}
E \\
\Longrightarrow E - E \\
\Longrightarrow 1 - E \\
\Longrightarrow 1 - E/E \\
\Longrightarrow 1 - E/2 \\
\Longrightarrow 1 - 4/2
\end{array}
$$



Parse tree examples

- **Example:**

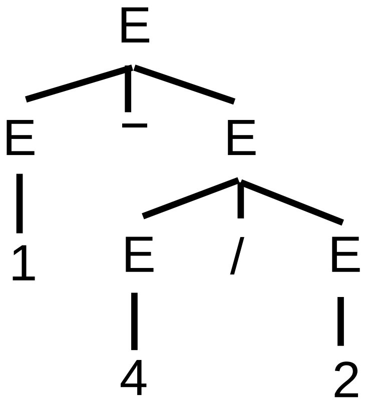
Grammar:

$E \rightarrow int$

$E \rightarrow E - E$

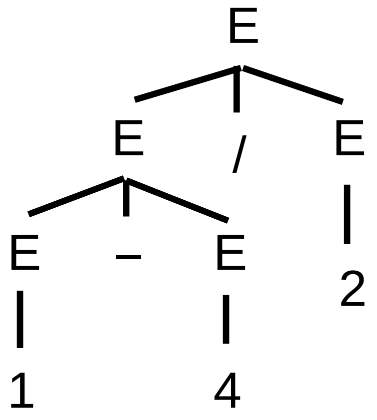
$E \rightarrow E/E$

$E \rightarrow (E)$



leftmost derivation

- Using $1 - 4/2$ as the input, the left parse tree is derived.
- A string is formed by reading the leaf nodes from left to right, which gives $1 - 4/2$.
- The string $1 - 4/2$ has another parse tree on the right.



rightmost derivation

- **Some standard notations:**
  - Given a parse tree and a fixed order (for example leftmost or rightmost) we can derive the order of derivation.
  - For the "semantic" of the parse tree, we normally "interpret" the meaning in a bottom-up fashion. That is, the one that is derived last will be "serviced" first.

Ambiguous grammar

- If for grammar G and string α, there are
  - more than one leftmost derivation for α, or
  - more than one rightmost derivation for α, or
  - more than one parse tree for α,

then G is called **ambiguous**.

  - Note: the above three conditions are equivalent in that if one is true, then all three are true.
  - Q: How to prove this?
    - ▷ *Hint: Any unannotated tree can be annotated with a leftmost numbering.*

- Problems with an ambiguous grammar:
  - Ambiguity can make parsing difficult.
  - Underlying structure is ill-defined: in the example, the precedence is not uniquely defined, e.g., the leftmost parse tree groups $4/2$ while the rightmost parse tree groups $1-4$, resulting in two different semantics.

Common grammar problems

- **Lists: that is, zero or more id's separated by commas:**
  - Note it is easy to express one or more id's:
    - ▷ $\textit{NonEmptyIdList} \rightarrow \textit{NonEmptyIdList}, id \mid id$
  - For zero or more id's,
    - ▷ $\textit{IdList}_1 \rightarrow \epsilon \mid id \mid \textit{IdList}_1, \textit{IdList}_1$
      will not work due to ϵ*; it can generate:* $id, , id$
    - ▷ $\textit{IdList}_2 \rightarrow \epsilon \mid \textit{IdList}_2, id \mid id$
      will not work either because it can generate: $, id, id$
  - We should separate out the empty list from the general list of one or more id's.
    - ▷ $\textit{OptIdList} \rightarrow \epsilon \mid \textit{NonEmptyIdList}$
    - ▷ $\textit{NonEmptyIdList} \rightarrow \textit{NonEmptyIdList}, id \mid id$
- **Expressions: precedence and associativity as discussed next.**
- **Useless terms: to be discussed.**

Grammar that expresses precedence correctly

- **Use one nonterminal for each precedence level**
- **Start with lower precedence (in our example “−”)**

Original grammar:

$E \longrightarrow int$

$E \longrightarrow E - E$

$E \longrightarrow E/E$

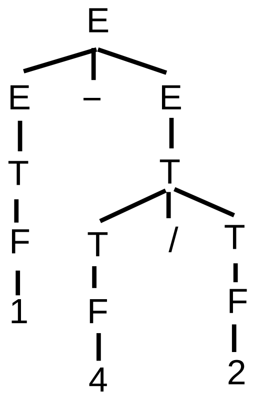
$E \longrightarrow (E)$

Revised grammar:

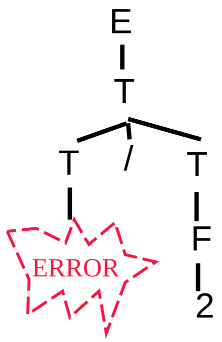
$E \longrightarrow E - E \mid T$

$T \longrightarrow T/T \mid F$

$F \longrightarrow int \mid (E)$

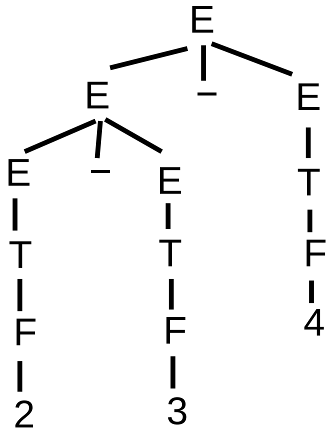


rightmost derivation



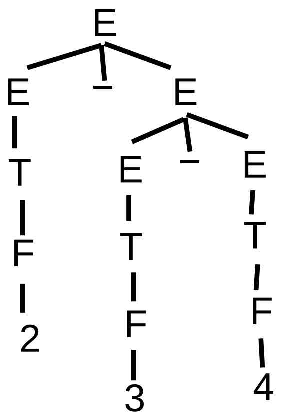
Problems with associativity

- **However, the above grammar is still ambiguous, and parse trees do not express the associative of "$-$" and "$/$" correctly. Example: $2-3-4$**



rightmost derivation

value = (2−3)−4 = −5



rightmost derivation

value = 2 − (3−4) = 3

Revised grammar:

$E \rightarrow E - E \mid T$

$T \rightarrow T/T \mid F$

$F \rightarrow int \mid (E)$

- **Problems with associativity:**
  - **The rule $E \rightarrow E - E$ has E on both sides of "$-$".**
  - **Need to make the second E to some other nonterminal parsed earlier.**
  - **Similarly for the rule $E \rightarrow E/E$.**

Grammar considering associative rules

Original grammar:

$E \to int$

$E \to E - E$

$E \to E/E$

$E \to (E)$

Revised grammar:

$E \to E - E \mid T$

$T \to T/T \mid F$

$F \to int \mid (E)$

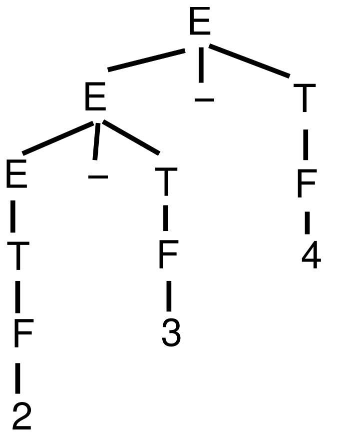
Final grammar:

$E \to E - T \mid T$

$T \to T/F \mid F$

$F \to int \mid (E)$

- **Example:** $2 - 3 - 4$



leftmost/rightmost derivation

value = (2–3)–4 = –5

Rules for associativity

- **Recursive productions:**
  - $E \rightarrow E - T$ **is called a left recursive production.**
    - ▷ $A \overset{+}{\Longrightarrow} A\alpha$.
  - $E \rightarrow T - E$ **is called a right recursive production.**
    - ▷ $A \overset{+}{\Longrightarrow} \alpha A$.
  - $E \rightarrow E - E$ **is both left and right recursive.**
- **If one wants left associativity, use left recursion.**
- **If one wants right associativity, use right recursion.**

Useless terms

- **A non-terminal X is useless if either**
  - **a sequence includes X cannot be derived from the starting nonterminal, or**
  - **no string can be derived starting from X, where a string means ϵ or a sequence of terminals.**
- **Example 1:**
  - $S \rightarrow A\ B$
  - $A \rightarrow + \mid - \mid \epsilon$
  - $B \rightarrow digit \mid B\ digit$
  - $C \rightarrow .\ B$
- **In Example 1:**
  - **C is useless and so is the last production.**
  - **Any nonterminal not in the right-hand side of any production is useless!**

More examples for useless terms

- **Example 2:**
  - $S \rightarrow X \mid Y$
  - $X \rightarrow (\ )$
  - $Y \rightarrow (\ Y\ Y\ )$
- Y **derives more and more nonterminals and is useless.**
- **Any recursively defined nonterminal without a production of deriving ϵ or a string of all terminals is useless!**
  - **Direct useless.**
  - **Indirect useless: one can only derive direct useless terms.**
- **From now on, we assume a grammar contains no useless nonterminals.**

How to use CFG

- **Breaks down the problem into pieces.**
  - Think about a C program:
    - ▷ *Declarations: typedef, struct, variables, . . .*
    - ▷ *Procedures: type-specifier, function name, parameters, function body.*
    - ▷ *function body: various statements.*
  - Example:
    - ▷ $\mathbf{Procedure} \rightarrow \mathbf{TypeDef}\ \ id\ \ \mathbf{OptParams}\ \mathbf{OptDecl}\ \{\mathbf{OptStatements}\}$
    - ▷ $\mathbf{TypeDef} \rightarrow integer \mid char \mid float \mid \cdots$
    - ▷ $\mathbf{OptParams} \rightarrow (\ \mathbf{ListParams}\ )$
    - ▷ $\mathbf{ListParams} \rightarrow \epsilon \mid \mathbf{NonEmptyParList}$
    - ▷ $\mathbf{NonEmptyParList} \rightarrow \mathbf{NonEmptyParList}, id \mid id$
    - ▷ $\cdots$
- **One of purposes to write a grammar for a language is for others to understand. It will be nice to break things up into different levels in a top-down easily understandable fashion.**

Non-context free grammars

- **Some grammar is not CFG, that is, it may be context sensitive.**
- **Expressive power of grammars (in the order of small to large):**
  - Regular expression $\equiv$ FA.
  - Context-free grammar
  - Context-sensitive grammar
  - $\cdots$
- $\{\omega c \omega \mid \omega$ **is a string of** a **and** b**'s**$\}$ **cannot be expressed by CFG.**

Top-down parsing

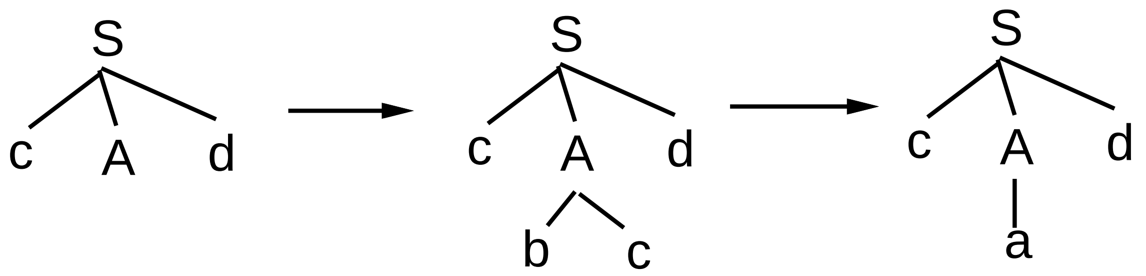
- **There are $O(n^3)$-time algorithms to parse a language defined by CFG, where n is the number of input tokens.**
- **For practical purpose, we need faster algorithms. Here we make restrictions to CFG so that we can design $O(n)$-time algorithms.**
- **Recursive-descent parsing : top-down parsing that allows backtracking.**
  - **Attempt to find a leftmost derivation for an input string.**
  - **Try out all possibilities, that is, do an exhaustive search to find a parse tree that parses the input.**

Example for recursive-descent parsing

$$S \rightarrow cAd$$

$$A \rightarrow bc \mid a$$

Input: cad



error!! backtrack

- **Problems with the above approach:**
  - **still too slow!**
  - **want to select a derivation without ever causing backtracking!**
  - **trick: use lookahead symbols!**
- **Solution: use $LL(1)$ grammars that can be parsed in $O(n)$ time.**
  - **first L: scan the input from left-to-right**
  - **second L: find a leftmost derivation**
  - **(1): allow one lookahead token!**

Predictive parser for $LL(1)$ grammars

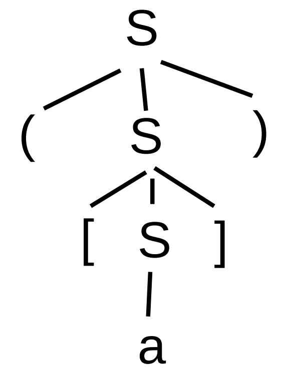
- **How a predictive parser works:**
  - start by pushing the starting nonterminal into the STACK and calling the scanner to get the first token.

  LOOP: if top-of-STACK is a nonterminal, then
    - *use the current token and the PARSING TABLE to choose a production*
    - *pop the nonterminal from the STACK*
    - *push the above production's right-hand-side to the STACK from right to left*
    - *GOTO LOOP.*
  - if top-of-STACK is a terminal and matches the current token, then
    - *pop STACK and ask scanner to provide the next token*
    - *GOTO LOOP.*
  - if STACK is empty and there is no more input, then ACCEPT!
  - If none of the above succeed, then FAIL!
    - *STACK is empty and there is input left.*
    - *top-of-STACK is a terminal, but does not match the current token*
    - *top-of-STACK is a nonterminal, but the corresponding PARSING TABLE entry is ERROR!*

Example for parsing an $LL(1)$ grammar

- **grammar:** $S \rightarrow a \mid (S) \mid [S]$ **input:** $([a])$

STACK	INPUT	ACTION
S	$([a])$	**pop, push "(S)"**
$)S($	$([a])$	**pop, match with input**
$)S$	$[a])$	**pop, push "$[S]$"**
$)]S[$	$[a])$	**pop, match with input**
$)]S$	$a])$	**pop, push "a"**
$)]a$	$a])$	**pop, match with input**
$)]$	$])$	**pop, match with input**
$)$	$)$	**pop, match with input**
		accept



leftmost derivation

- **Use the current input token to decide which production to derive from the top-of-STACK nonterminal.**

About $LL(1)$

- **It is not always possible to build a predictive parser given a CFG; It works only if the CFG is $LL(1)$!**
  - **$LL(1)$ is a subset of CFG.**
- **For example, the following grammar is not $LL(1)$, but is $LL(2)$.**
- **Grammar: $S \rightarrow (S) \mid [S] \mid () \mid [\,]$**
  Try to parse the input $()$.

STACK	INPUT	ACTION
S	$()$	pop, but use which production?

- **In this example, we need 2-token look-ahead.**
  - **If the next token is $)$, push "$()$" from right to left.**
  - **If the next token is $($, push "(S)" from right to left.**
- **Two questions:**
  - **How to tell whether a grammar G is $LL(1)$?**
  - **How to build the PARSING TABLE?**

First property for non-$LL(1)$ grammars

- **Theorem 1: G is not $LL(1)$ if a nonterminal has two productions whose right-hand-sides have a common prefix.**
  - ▷ *Have left-factors.*
  - ▷ *Q: How to prove it?*
- **Example: $S \rightarrow (S) \mid ()$**
- **In this example, the common prefix is "(".**
- **This problem can be solved by using the left-factoring trick.**
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  - **Transform to:**
    - ▷ $A \rightarrow \alpha A'$
    - ▷ $A' \rightarrow \beta_1 \mid \beta_2$
- **Example:**
  - $S \rightarrow (S) \mid ()$
  - **Transform to**
    - ▷ $S \rightarrow (S'$
    - ▷ $S' \rightarrow S) \mid)$

Algorithm for left-factoring

- **Input: context free grammar G**
- **Output: equivalent left-factored context-free grammar G'**
- **for each nonterminal A do**
  - **find the longest non-ϵ prefix α that is common to right-hand sides of two or more productions;**
  - **replace**
    - ▷ $A \longrightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$

    with
    - ▷ $A \longrightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m$
    - ▷ $A' \longrightarrow \beta_1 \mid \cdots \mid \beta_n$
  - **repeat the above process until A has no two productions with a common prefix;**

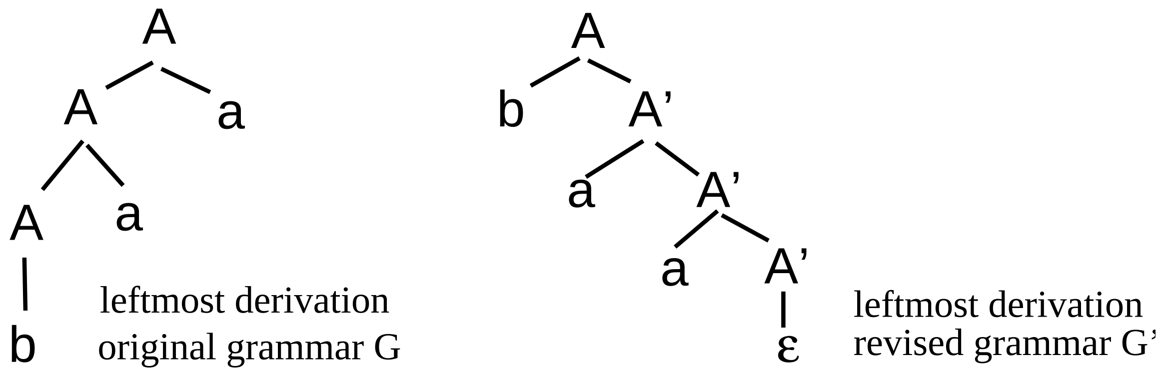
Second property for non-$LL(1)$ grammars

- **Theorem 2: A CFG grammar is not $LL(1)$ if it is left-recursive.**
  - **Q: How to prove it?**
- **Definitions:**
  - **recursive grammar: a grammar is recursive if this grammar contains a nonterminal X such that**
    $X \overset{+}{\Longrightarrow} \alpha X \beta$.
  - G **is left-recursive if** $X \overset{+}{\Longrightarrow} X\beta$.
  - G **is immediately left-recursive if** $X \Longrightarrow X\beta$.

Example of removing immediate left-recursion

- **Need to remove left-recursion to come out an $LL(1)$ grammar. Example:**
  - **Grammar G: $A \rightarrow A\alpha \mid \beta$, where β does not start with A**
  - **Revised grammar G':**
    - ▷ $A \rightarrow \beta A'$
    - ▷ $A' \rightarrow \alpha A' \mid \epsilon$
  - **The above two grammars are equivalent. That is $L(G) \equiv L(G')$.**
- **Example:**

input baa
$\beta \equiv b$
$\alpha \equiv a$



leftmost derivation
original grammar G

leftmost derivation
revised grammar G'

Rule for removing immediate left-recursion

- **Both grammars recognize the same string, but G' is not left-recursive.**
- **However, G is clear and intuitive.**
- **General rule for removing immediately left-recursion:**
  - **Replace** $A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$
  - **with**
    - ▷ $A \longrightarrow \beta_1 A' \mid \cdots \mid \beta_n A'$
    - ▷ $A' \longrightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$
- **This rule does not work if $\alpha_i = \epsilon$ for some i.**
  - **This is called a *direct cycle* in a grammar.**
- **May need to worry about whether the semantics are equivalent between the original grammar and the transformed grammar.**

Algorithm 4.1

- **Algorithm 4.1 systematically eliminates left recursion and works only if the input grammar has no cycles or ϵ-productions.**
  - ▷ *Cycle:* $A \xRightarrow{+} A$
  - ▷ *ϵ-production:* $A \rightarrow \epsilon$
  - ▷ *It is possible to remove cycles and all but one ϵ-production using other algorithms.*

```
Input: grammar $G$ without cycles and $\epsilon$-productions.
Output: An equivalent grammar without left recursion.
Number the nonterminals in some order $A_1, A_2, \ldots, A_n$
for $i = 1$ to $n$ do
    for $j = 1$ to $i - 1$ do
        replace $A_i \rightarrow A_j\gamma$
        with $A_i \rightarrow \delta_1\gamma \mid \cdots \mid \delta_k\gamma$
        where $A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k$ are all the current $A_j$-productions.
    Eliminate immediate left-recursion for $A_i$
        New nonterminals generated above are numbered $A_{i+n}$
```

Algorithm 4.1 — Discussions

- **Intuition:**
  - Consider only the productions where the leftmost item on the right hand side are nonterminals.
  - If it is always the case that
    - ▷ $A_i \overset{+}{\Longrightarrow} A_j\alpha$ *implies* $i < j$, *then*

    it is not possible to have left-recursion.
- **Why cycles are not allowed?**
  - For the procedure of removing immediate left-recursion.
- **Why ϵ-productions are not allowed?**
  - Inside the loop, when $A_j \to \epsilon$, that is some $\delta_g = \epsilon$, and the prefix of γ is some A_k where $k < i$, it generates $A_i \to A_k$, $k < i$.
- **Time and space complexities:**
  - Size of the resulting grammar can be $O(w^3)$, where w is the original size.
  - $O(n^2w^3)$ time, where n is the number of nonterminals in the input grammar.

Trace an instance of Algorithm 4.1

- **After each i-loop, only productions of the form $A_i \rightarrow A_k\gamma$, $i < k$ remain.**
- $i = 1$
  - **allow** $A_1 \rightarrow A_k\alpha$, $\forall k$ **before removing immediate left-recursion**
  - **remove immediate left-recursion for** A_1
- $i = 2$
  - $j = 1$: **replace** $A_2 \rightarrow A_1\gamma$ **by**
    $A_2 \rightarrow (A_{k_1}\alpha_1 \mid \cdots \mid A_{k_p}\alpha_p)\gamma$, **where**
    $A_1 \rightarrow (A_{k_1}\alpha_1 \mid \cdots \mid A_{k_p}\alpha_p)$ **and** $k_j > 1$ $\forall k_j$
  - **remove immediate left-recursion for** A_2
- $i = 3$
  - $j = 1$: **replace** $A_3 \rightarrow A_1\gamma_1$
  - $j = 2$: **replace** $A_3 \rightarrow A_2\gamma_2$
  - **remove immediate left-recursion for** A_3
- $\cdots$

Example

- **Original Grammar:**
  - **(1)** $S \rightarrow Aa \mid b$
  - **(2)** $A \rightarrow Ac \mid Sd \mid e$
- **Ordering of nonterminals:** $S \equiv A_1$ **and** $A \equiv A_2$**.**
- $i = 1$
  - **do nothing as there is no immediate left-recursion for** S
- $i = 2$
  - **replace** $A \rightarrow Sd$ **by** $A \rightarrow Aad \mid bd$
  - **hence (2) becomes** $A \rightarrow Ac \mid Aad \mid bd \mid e$
  - **after removing immediate left-recursion:**
    - ▷ $A \rightarrow bdA' \mid eA'$
    - ▷ $A' \rightarrow cA' \mid adA' \mid \epsilon$
- **Resulting grammar:**
  - ▷ $S \rightarrow Aa \mid b$
  - ▷ $A \rightarrow bdA' \mid eA'$
  - ▷ $A' \rightarrow cA' \mid adA' \mid \epsilon$

Left-factoring and left-recursion removal

- **Original grammar:**
  $S \rightarrow (S) \mid SS \mid ()$
- **To remove immediate left-recursion, we have**
  - $S \rightarrow (S)S' \mid ()S'$
  - $S' \rightarrow SS' \mid \epsilon$
- **To do left-factoring, we have**
  - $S \rightarrow (S''$
  - $S'' \rightarrow S)S' \mid)S'$
  - $S' \rightarrow SS' \mid \epsilon$
- **A grammar is not $LL(1)$ if it**
  - **is left recursive or**
  - **has left-factors.**
- **However, grammars that are not left recursive and have no left-factors may still not be $LL(1)$.**
  - **Q: Any examples?**

Definition of $LL(1)$ grammars

- **To see if a grammar is $LL(1)$, we need to compute its FIRST and FOLLOW sets, which are used to build its parsing table.**
- **FIRST sets:**
  - **Definition: let α be a sequence of terminals and/or nonterminals or ϵ**
    - ▷ ***FIRST**(α) **is the set of terminals that begin the strings derivable from** α*
    - ▷ ***if** α **can derive** ϵ**, then** $\epsilon \in$ **FIRST**(α)*
  - **FIRST**$(\alpha) = \{t \mid$ **(**t **is a terminal and** $\alpha \overset{*}{\Longrightarrow} t\beta$**) or (** $t = \epsilon$ **and** $\alpha \overset{*}{\Longrightarrow} \epsilon$**)**$\}$

How to compute FIRST(X)? (1/2)

- X is a terminal:
  - FIRST$(X) = \{X\}$
- X is ϵ:
  - FIRST$(X) = \{\epsilon\}$
- X is a nonterminal: must check all productions with X on the left-hand side. That is,

  for all $\boxed{X \rightarrow Y_1 Y_2 \cdots Y_k}$ perform the following steps:
  - put FIRST$(Y_1) - \{\epsilon\}$ into FIRST(X)
  - if $\epsilon \in$ FIRST(Y_1), then put FIRST$(Y_2) - \{\epsilon\}$ into FIRST(X)
  - $\cdots$
  - if $\epsilon \in$ FIRST(Y_{k-1}), then put FIRST$(Y_k) - \{\epsilon\}$ into FIRST(X)
  - if $\epsilon \in$ FIRST(Y_i) for each $1 \leq i \leq k$, then put ϵ into FIRST(X)

How to compute FIRST(X)? (2/2)

- **Algorithm to compute FIRST's for all non-terminals.**
  - compute FIRST's for ϵ and all terminals;
  - initialize FIRST's for all non-terminals to $\emptyset$;
  - Repeat

    for all nonterminals X do
    - ▷ *apply the steps to compute* $FIRST(X)$
  - Until no items can be added to any FIRST set;
- **What to do when recursive calls are encountered?**
  - direct recursive calls
  - indirect recursive calls
  - actions: do not go further
    - ▷ *why?*
- **The time complexity of this algorithm.**
  - at least one item, terminal or ϵ, is added to some FIRST set in an iteration;
  - total number of items in all FIRST sets are $(|T| + 1) \cdot |N|$, where T is the set of terminals and N is the set of nonterminals.
  - $O(|N|^2 \cdot |T|)$.

Example for computing FIRST(X)

- **Start with computing FIRST for the last production and walk your way up.**

Grammar

$E \rightarrow E'T$

$E' \rightarrow -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow /\ FT' \mid \epsilon$

$F \rightarrow int \mid (E)$

$H \rightarrow E'T$

FIRST$(F) = \{int, (\}$

FIRST$(T') = \{/, \epsilon\}$

FIRST$(T) = \{int, (\}$**,**
since $\epsilon \notin$ **FIRST**(F)**, that's all.**

FIRST$(E') = \{-, \epsilon\}$

FIRST$(H) = \{-, int, (\}$

FIRST$(E) = \{-, int, (\}$**,**
since $\epsilon \in$ **FIRST**(E')**.**

How to compute FIRST(α)?

- To build a parsing table, we need FIRST(α) for all α such that $X \rightarrow \alpha$ is a production in the grammar.
  - Need to compute FIRST(X) for each nonterminal X.
- Let $\alpha = X_1 X_2 \cdots X_n$. Perform the following steps in sequence:
  - put FIRST(X_1) $- \{\epsilon\}$ into FIRST(α)
  - if $\epsilon \in$ FIRST(X_1), then put FIRST(X_2) $- \{\epsilon\}$ into FIRST(α)
  - $\cdots$
  - if $\epsilon \in$ FIRST(X_{n-1}), then put FIRST(X_n) $- \{\epsilon\}$ into FIRST(α)
  - if $\epsilon \in$ FIRST(X_i) for each $1 \leq i \leq n$, then put $\{\epsilon\}$ into FIRST(α).
- What to do when recursive calls are encountered?

Example for computing FIRST(α)

Grammar

$E \rightarrow E'T$

$E' \rightarrow -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow /FT' \mid \epsilon$

$F \rightarrow int \mid (E)$

$\boldsymbol{FIRST}(F) = \{int, (\}$

$\boldsymbol{FIRST}(T') = \{/, \epsilon\}$

$\boldsymbol{FIRST}(T) = \{int, (\}$

$\boldsymbol{FIRST}(E') = \{-, \epsilon\}$

$\boldsymbol{FIRST}(E) = \{-, int, (\}$

$\boldsymbol{FIRST}(E'T) = \{-, int, (\}$

$\boldsymbol{FIRST}(-TE') = \{-\}$

$\boldsymbol{FIRST}(\epsilon) = \{\epsilon\}$

$\boldsymbol{FIRST}(FT') = \{int, (\}$

$\boldsymbol{FIRST}(/FT') = \{/\}$

$\boldsymbol{FIRST}(\epsilon) = \{\epsilon\}$

$\boldsymbol{FIRST}(int) = \{int\}$

$\boldsymbol{FIRST}((E)) = \{(\}$

- **FIRST**$(T'E') =$
  - ▷ $(\boldsymbol{FIRST}(T') - \{\epsilon\}) \cup$
  - ▷ $(\boldsymbol{FIRST}(E') - \{\epsilon\}) \cup$
  - ▷ $\{\epsilon\}$

Why do we need FIRST(α)?

- **During parsing, suppose top-of-STACK is a nonterminal A and there are several choices**
  - $A \rightarrow \alpha_1$
  - $A \rightarrow \alpha_2$
  - $\cdots$
  - $A \rightarrow \alpha_k$

  for derivation, and the current lookahead token is a
- **If $a \in$ FIRST(α_i), then pick $A \rightarrow \alpha_i$ for derivation, pop, and then push α_i.**
- **If a is in several FIRST(α_i)'s, then the grammar is not $LL(1)$.**
- **Question: if a is not in any FIRST(α_i), does this mean the input stream cannot be accepted?**
  - **Maybe not!**
  - **What happen if ϵ is in some FIRST(α_i)?**

FOLLOW sets

- **Assume there is a special EOF symbol "\$" ends every input.**
- **Add a new terminal "\$".**
- **Definition: for a nonterminal X, FOLLOW(X) is the set of terminals that can appear immediately to the right of X in some partial derivation.**

  That is, $S \overset{+}{\Longrightarrow} \alpha_1 X t \alpha_2$, where t is a terminal.
- **If X can be the rightmost symbol in a derivation, then \$ is in FOLLOW(X).**
- **FOLLOW$(X) =$**

  $\{t \mid$ (t is a terminal and $S \overset{+}{\Longrightarrow} \alpha_1 X t \alpha_2$) or (t is \$ and $S \overset{+}{\Longrightarrow} \alpha X$)$\}$.

How to compute FOLLOW(X)?

- If X is the starting nonterminal, put \$ into FOLLOW(X).
- Find the productions with X on the right-hand-side.
  - for each production of the form $Y \rightarrow \alpha X \beta$, put FIRST$(\beta) - \{\epsilon\}$ into FOLLOW(X).
  - if $\epsilon \in$ FIRST(β), then put FOLLOW(Y) into FOLLOW(X).
  - for each production of the form $Y \rightarrow \alpha X$, put FOLLOW(Y) into FOLLOW(X).
- Repeat the above process for all nonterminals until nothing can be added to any FOLLOW set.
  - What to do when recursive calls are encountered?
  - Q: time and space complexities
- To see if a given grammar is $LL(1)$, or to to build its parsing table:
  - compute FIRST(α) for every α such that $X \rightarrow \alpha$ is a production
  - compute FOLLOW(X) for all nonterminals X
    - ▷ *need to compute FIRST(α) for every α such that $Y \rightarrow \beta X \alpha$ is a production*

A complete example

- **Grammar**
  - $S \rightarrow Bc \mid DB$
  - $B \rightarrow ab \mid cS$
  - $D \rightarrow d \mid \epsilon$

α	**FIRST**(α)	**FOLLOW**(α)
D	$\{d, \epsilon\}$	$\{a, c\}$
B	$\{a, c\}$	$\{c, \$\}$
S	$\{a, c, d\}$	$\{c, \$\}$
Bc	$\{a, c\}$	
DB	$\{d, a, c\}$	
ab	$\{a\}$	
cS	$\{c\}$	
d	$\{d\}$	
ϵ	$\{\epsilon\}$	

Why do we need FOLLOW sets?

- Note FOLLOW(S) always includes \$.
- Situation:
  - During parsing, the top-of-STACK is a nonterminal X and the lookahead symbol is a.
  - Assume there are several choices for the nest derivation:
    - ▷ $X \longrightarrow \alpha_1$
    - ▷ $\cdots$
    - ▷ $X \longrightarrow \alpha_k$
  - If $a \in$ FIRST(α_i) for exactly one i, then we use that derivation.
  - If $a \in$ FIRST(α_i), $a \in$ FIRST(α_j), and $i \neq j$, then this grammar is not $LL(1)$.
  - If $a \notin$ FIRST(α_i) for all i, then this grammar can still be $LL(1)$!
- If there exists some i such that $\alpha_i \overset{*}{\Longrightarrow} \epsilon$ and $a \in$ FOLLOW(X), then we can use the derivation $X \longrightarrow \alpha_i$.
  - $\alpha_i \overset{*}{\Longrightarrow} \epsilon$ if and only if $\epsilon \in$ FIRST(α_i).

Grammars that are not $LL(1)$

- **A grammar is not $LL(1)$ if there exists productions**

$$\boxed{X \rightarrow \alpha \mid \beta}$$

and any one of the followings is true:

  - FIRST$(\alpha) \cap$ FIRST$(\beta) \neq \emptyset$.
    - *It may be the case that $\epsilon \in$ FIRST(α) and $\epsilon \in$ FIRST(β).*
  - $\epsilon \in$ FIRST(α), and FIRST$(\beta) \cap$ FOLLOW$(X) \neq \emptyset$.
- **If a grammar is not $LL(1)$, then**
  - you cannot write a linear-time predictive parser as described above.
- **If a grammar is not $LL(1)$, then**
we do not know to use the production $X \rightarrow \alpha$ or the production $X \rightarrow \beta$ when the lookahead symbol is a in any of the following cases:
  - $a \in$ FIRST$(\alpha) \cap$ FIRST(β);
  - $\epsilon \in$ FIRST(α) and $\epsilon \in$ FIRST(β);
  - $\epsilon \in$ FIRST(α), and $a \in$ FIRST$(\beta) \cap$ FOLLOW(X).

A complete example (1/2)

- **Grammar:**
  - **ProgHead** $\rightarrow$ $prog\ id$ **Parameter** $semicolon$
  - **Parameter** $\rightarrow$ $\epsilon \mid id \mid l\_paren$ **Parameter** r_paren
- **FIRST and FOLLOW sets:**

α	FIRST(α)	FOLLOW(α)
ProgHead	$\{prog\}$	$\{\$\}$
Parameter	$\{\epsilon, id, l\_paren\}$	$\{semicolon, r\_paren\}$
$prog\ id$ Parameter $semicolon$	$\{prog\}$	
l_paren Parameter r_paren	$\{l\_paren\}$	

A complete example (2/2)

Input: prog id semicolon

STACK	INPUT	ACTION
\$ ProgHead	*prog id semicolon* \$	pop, push
\$ *semicolon* Parameter *id prog*	*prog id semicolon* \$	match with input
\$ *semicolon* Parameter *id*	*id semicolon* \$	match with input
\$ *semicolon* Parameter	*semicolon* \$	WHAT TO DO?

- **Last actions:**
  - **Three choices:**
    - ▷ ***Parameter*** $\rightarrow \epsilon \mid id \mid l\_paren$ ***Parameter*** r_paren
  - $semicolon \notin$ **FIRST**(ϵ) **and** $semicolon \notin$ **FIRST**(id) **and** $semicolon \notin$ **FIRST**$(l\_paren$ **Parameter** $r\_paren)$
  - **Parameter** $\overset{*}{\Longrightarrow} \epsilon$ **and** $semicolon \in$ **FOLLOW**(**Parameter**)
  - **Hence we use the derivation**
    Parameter $\rightarrow \epsilon$

$LL(1)$ parsing table (1/2)

Grammar:

- $S \rightarrow XC$
- $X \rightarrow a \mid \epsilon$
- $C \rightarrow a \mid \epsilon$

α	FIRST(α)	FOLLOW(α)
S	$\{a, \epsilon\}$	$\{\$\}$
X	$\{a, \epsilon\}$	$\{a, \$\}$
C	$\{a, \epsilon\}$	$\{\$\}$
ϵ	$\{\epsilon\}$	
a	$\{a\}$	
XC	$\{a, \epsilon\}$	

- **Check for possible conflicts in $X \rightarrow a \mid \epsilon$.**
  - **FIRST**$(a) \cap$ **FIRST**$(\epsilon) = \emptyset$
  - $\epsilon \in$ **FIRST**(ϵ) **and FOLLOW**$(X) \cap$ **FIRST**$(a) = \{a\}$

    Conflict!!
  - $\epsilon \notin$ **FIRST**(a)
- **Check for possible conflicts in $C \rightarrow a \mid \epsilon$.**
  - **FIRST**$(a) \cap$ **FIRST**$(\epsilon) = \emptyset$
  - $\epsilon \in$ **FIRST**(ϵ) **and FOLLOW**$(C) \cap$ **FIRST**$(a) = \emptyset$
  - $\epsilon \notin$ **FIRST**(a)

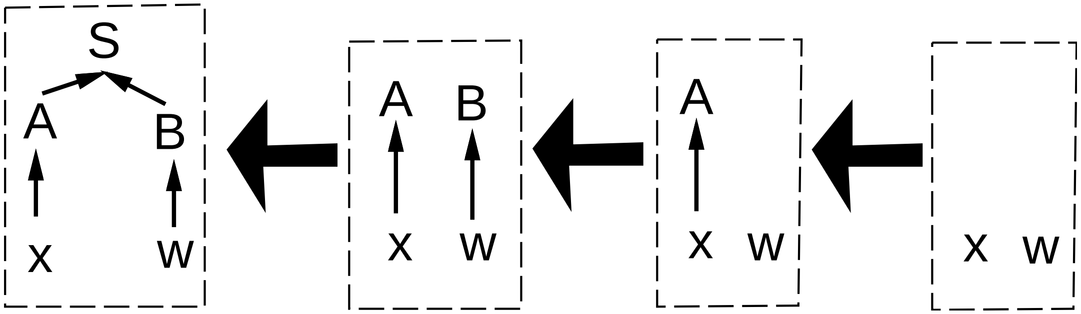
$LL(1)$ parsing table (2/2)

- **Parsing table:**

	a	**$**
S	$S \rightarrow XC$	$S \rightarrow XC$
X	**conflict**	$X \rightarrow \epsilon$
C	$C \rightarrow a$	$C \rightarrow \epsilon$

Bottom-up parsing (Shift-reduce parsers)

- **Intuition: construct the parse tree from the leaves to the root.**



- **Example:**

  Grammar:

  $S \rightarrow AB$

  $A \rightarrow x \mid Y$

  $B \rightarrow w \mid Z$

  $Y \rightarrow xb$

  $Z \rightarrow wp$

- **Input xw.**
- **This grammar is not $LL(1)$.**

Definitions (1/2)

- **Rightmost derivation:**
  - $S \underset{rm}{\Longrightarrow} \alpha$: **the rightmost nonterminal is replaced.**
  - $S \overset{+}{\underset{rm}{\Longrightarrow}} \alpha$: **$\alpha$ is derived from S using one or more rightmost derivations.**
    - ▷ ***α is called a right-sentential form.***
  - **In the previous example:** $S \underset{rm}{\Longrightarrow} AB \underset{rm}{\Longrightarrow} Aw \underset{rm}{\Longrightarrow} xw$.
- **Define similarly for leftmost derivation and left-sentential form.**
- **handle: a handle for a right-sentential form γ**
  - **is the combining of the following two information:**
    - ▷ ***a production rule $A \rightarrow \beta$ and***
    - ▷ ***a position w in γ where β can be found.***
  - **Let γ' be obtained by replacing β at the position w with A in γ. It is required that γ' is also a right-sentential form.**

Definitions (2/2)

- **Example:**

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

input: abbcde

$\gamma \equiv aAbcde$ is a right-sentential form

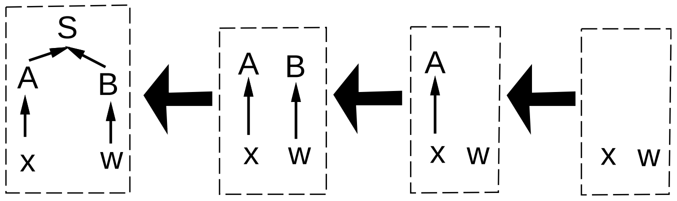
$A \rightarrow Abc$ and position 2 in γ is a handle for γ

- **reduce : replace a handle in a right-sentential form with its left-hand-side. In the above example, replace Abc starting at position 2 in γ with A.**
- **A right-most derivation in reverse can be obtained by handle reducing.**
- **Problems:**
  - **How handles can be found?**
  - **What to do when there are two possible handles?**
    - ▷ ***Ends at the same position.***
    - ▷ ***Have overlaps.***

STACK implementation

- **Four possible actions:**
  - **shift: shift the input to STACK.**
  - **reduce: perform a reversed rightmost derivation.**
    - *The first item popped is the rightmost item in the right hand side of the reduced production.*
  - **accept**
  - **error**

STACK	INPUT	ACTION
\$	xw\$	shift
\$$x$	w\$	reduce by $A \rightarrow x$
\$$A$	w\$	shift
\$$Aw$	\$	reduce by $B \rightarrow w$
\$$AB$	\$	reduce by $S \rightarrow AB$
\$$S$	\$	accept

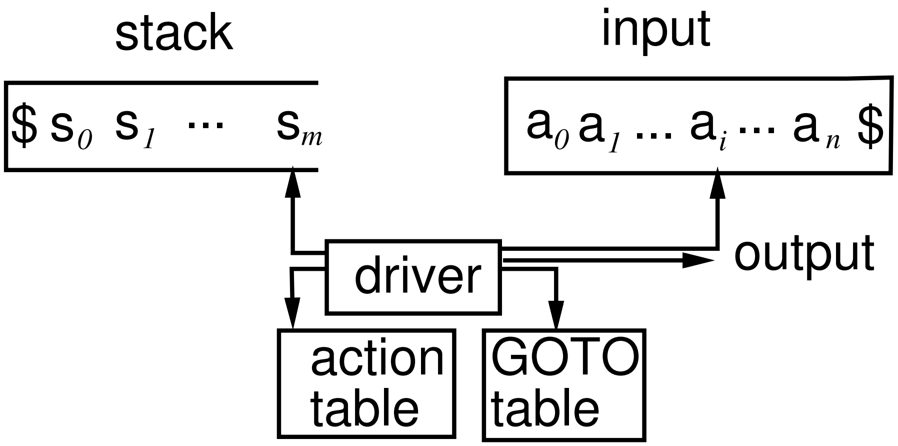


$$S \underset{rm}{\Longrightarrow} AB \underset{rm}{\Longrightarrow} Aw \underset{rm}{\Longrightarrow} xw.$$

Viable prefix

- Definition: the set of prefixes of right-sentential forms that can appear on the top of the stack.
  - Some suffix of a viable prefix is a prefix of a handle.
  - Some suffix of a viable prefix may be a handle.
- Some prefix of a right-sentential form cannot appear on the top of the stack during parsing.
  - xw is a right-sentential form.
  - The prefix xw is not a viable prefix.
  - You cannot have the situation that some suffix of xw is a handle.
- Note: when doing bottom-up parsing, that is reversed rightmost derivation,
  - it cannot be the case a handle on the right is reduced before a handle on the left in a right-sentential form;
  - the handle of the first reduction consists of all terminals and can be found on the top of the stack;
    - ▷ *That is, some substring of the input is the first handle.*

Model of a shift-reduce parser



- **Push-down automata!**
  - **Current state S_m encodes the symbols that has been shifted and the handles that are currently being matched.**
  - $\$S_0S_1 \cdots S_m a_i a_{i+1} \cdots a_n\$$ **represents a right-sentential form.**
  - **GOTO table:**
    - ▷ ***when a "reduce" action is taken, which handle to replace;***
  - **Action table:**
    - ▷ ***when a "shift" action is taken, which state currently in, that is, how to group symbols into handles.***
- **The power of context free grammars is equivalent to nondeterministic push down automata.**
  - ▷ ***Not equal to deterministic push down automata.***

LR parsers

- **By Don Knuth at 1965.**
- $LR(k)$**: see all of what can be derived from the right side with** k **input tokens lookahead.**
    - **first** L**: scan the input from left to right**
    - **second** R**: reverse rightmost derivation**
    - (k)**: with** k **lookahead tokens.**
- **Be able to decide the whereabout of a handle after seeing all of what have been derived so far plus** k **input tokens lookahead.**

$$X_1, X_2, \ldots, \underbrace{\boxed{X_i, X_{i+1}, \ldots, X_{i+j},}}_{\text{a handle}} \underbrace{\boxed{X_{i+j+1}, \ldots, X_{i+j+k},}}_{\text{lookahead tokens}} \ldots$$

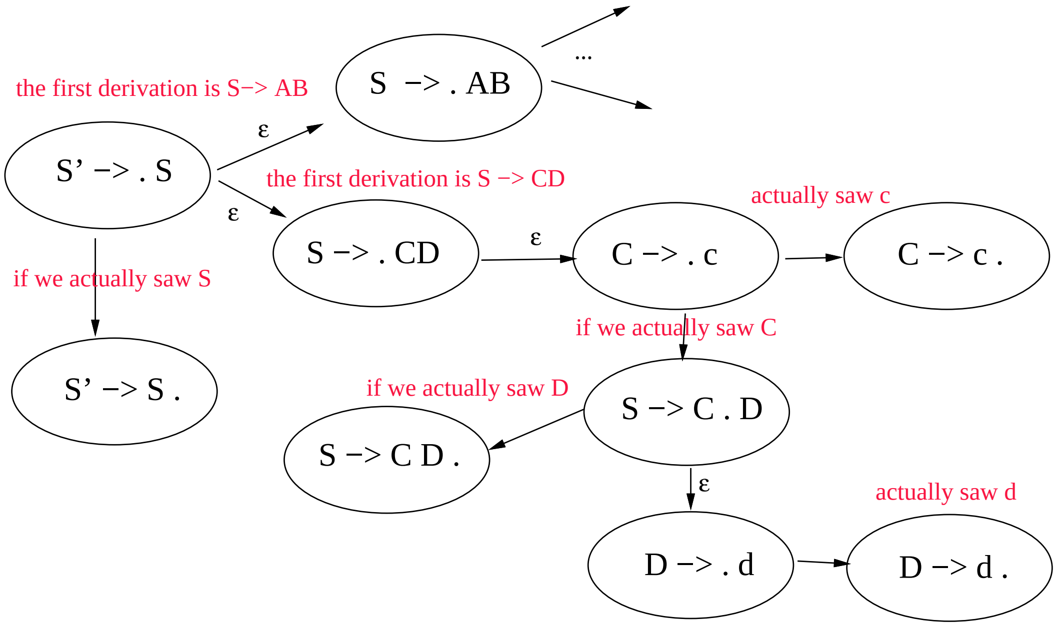
- **Top-down parsing for** $LL(k)$ **grammars: be able to choose a production by seeing only the first** k **symbols that will be derived from that production.**

$LR(0)$ parsing

- **Use a push down automata to recognize viable prefixes.**
- **An $LR(0)$ item (item for short) is a production, with a dot at some position in the RHS (right-hand side).**
  - **The production is the handle.**
  - **The dot indicates the prefix of the handle that has seen so far.**
- **Example:**
  - $A \rightarrow XY$
    - ▷ $A \rightarrow \cdot XY$
    - ▷ $A \rightarrow X \cdot Y$
    - ▷ $A \rightarrow XY \cdot$
  - $A \rightarrow \epsilon$
    - ▷ $A \rightarrow \cdot$
- **Augmented grammar G' is to add a new starting symbol S' and a new production $S' \rightarrow S$ to a grammar G with the starting symbol S.**
  - **We assume working on the augmented grammar from now on.**

High-level ideas for $LR(0)$ parsing

- **Grammar:**
  - $S' \rightarrow S$
  - $S \rightarrow AB \mid CD$
  - $A \rightarrow a$
  - $B \rightarrow b$
  - $C \rightarrow c$
  - $D \rightarrow d$
- **Approach:**
  - *Use a stack to record the history of all partial handles.*
  - *Use NFA to record information about the current handle.*
  - *push down automata = FA + stack.*
  - *Need to use DFA for simplicity.*



S’ −> S −> CD −> Cd −> cd

Closure

- **The closure operation $closure(I)$, where I is a set of items, is defined by the following algorithm:**
  - **If $A \rightarrow \alpha \cdot B\beta$ is in $closure(I)$, then**
    - ▷ *at some point in parsing, we might see a substring derivable from $B\beta$ as input;*
    - ▷ *if $B \rightarrow \gamma$ is a production, we also see a substring derivable from γ at this point.*
    - ▷ *Thus $B \rightarrow \cdot\gamma$ should also be in $closure(I)$.*
- **What does $closure(I)$ mean informally?**
  - **When $A \rightarrow \alpha \cdot B\beta$ is encountered during parsing, then this means we have seen α so far, and expect to see $B\beta$ later before reducing to A.**
  - **At this point if $B \rightarrow \gamma$ is a production, then we may also want to see $B \rightarrow \cdot\gamma$ in order to reduce to B, and then advance to $A \rightarrow \alpha B \cdot \beta$.**
- **Using $closure(I)$ to record all possible things about the next handle that we have seen in the past and expect to see in the future.**

Example for the closure function

- **Example: E' is the new starting symbol, and E is the original starting symbol.**
  - $E' \rightarrow E$
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * F \mid F$
  - $F \rightarrow (E) \mid id$
- $closure(\{E' \rightarrow \cdot E\}) =$
  - $\{E' \rightarrow \cdot E$,
  - $E \rightarrow \cdot E + T$,
  - $E \rightarrow \cdot T$,
  - $T \rightarrow \cdot T * F$,
  - $T \rightarrow \cdot F$,
  - $F \rightarrow \cdot(E)$,
  - $F \rightarrow \cdot id\}$

GOTO table

- $GOTO(I, X)$, **where** I **is a set of items and** X **is a legal symbol, means**
  - **If** $A \to \alpha \cdot X\beta$ **is in** I**, then**
  - $closure(\{A \to \alpha X \cdot \beta\}) \subseteq GOTO(I, X)$
- **Informal meanings:**
  - **currently we have seen** $A \to \alpha \cdot X\beta$
  - **expect to see** X
  - **if we see** X**,**
  - **then we should be in the state** $closure(\{A \to \alpha X \cdot \beta\})$**.**
- **Use the GOTO table to denote the state to go to once we are in** I **and have seen** X**.**

Sets-of-items construction

- **Canonical $LR(0)$ items** : the set of all possible DFA states, where each state is a set of $LR(0)$ items.
- Algorithm for constructing $LR(0)$ parsing table.
  - $C \leftarrow \{closure(\{S' \rightarrow \cdot S\})\}$
  - repeat
    - ▷ *for each set of items I in C and each grammar symbol X such that $GOTO(I, X) \neq \emptyset$ and not in C do*
    - ▷ *add $GOTO(I, X)$ to C*
  - until no more sets can be added to C
- **Kernel** of a state:
  - Definitions: items
    - ▷ *not of the form $X \rightarrow \cdot\beta$ or*
    - ▷ *of the form $S' \rightarrow \cdot S$*
  - Given the kernel of a state, all items in this state can be derived.

Example of sets of $LR(0)$ items

- **Grammar:**

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

$I_0 = closure(\{E' \rightarrow \cdot E\}) =$

$\{E' \rightarrow \cdot E,$

$E \rightarrow \cdot E + T,$

$E \rightarrow \cdot T,$

$T \rightarrow \cdot T * F,$

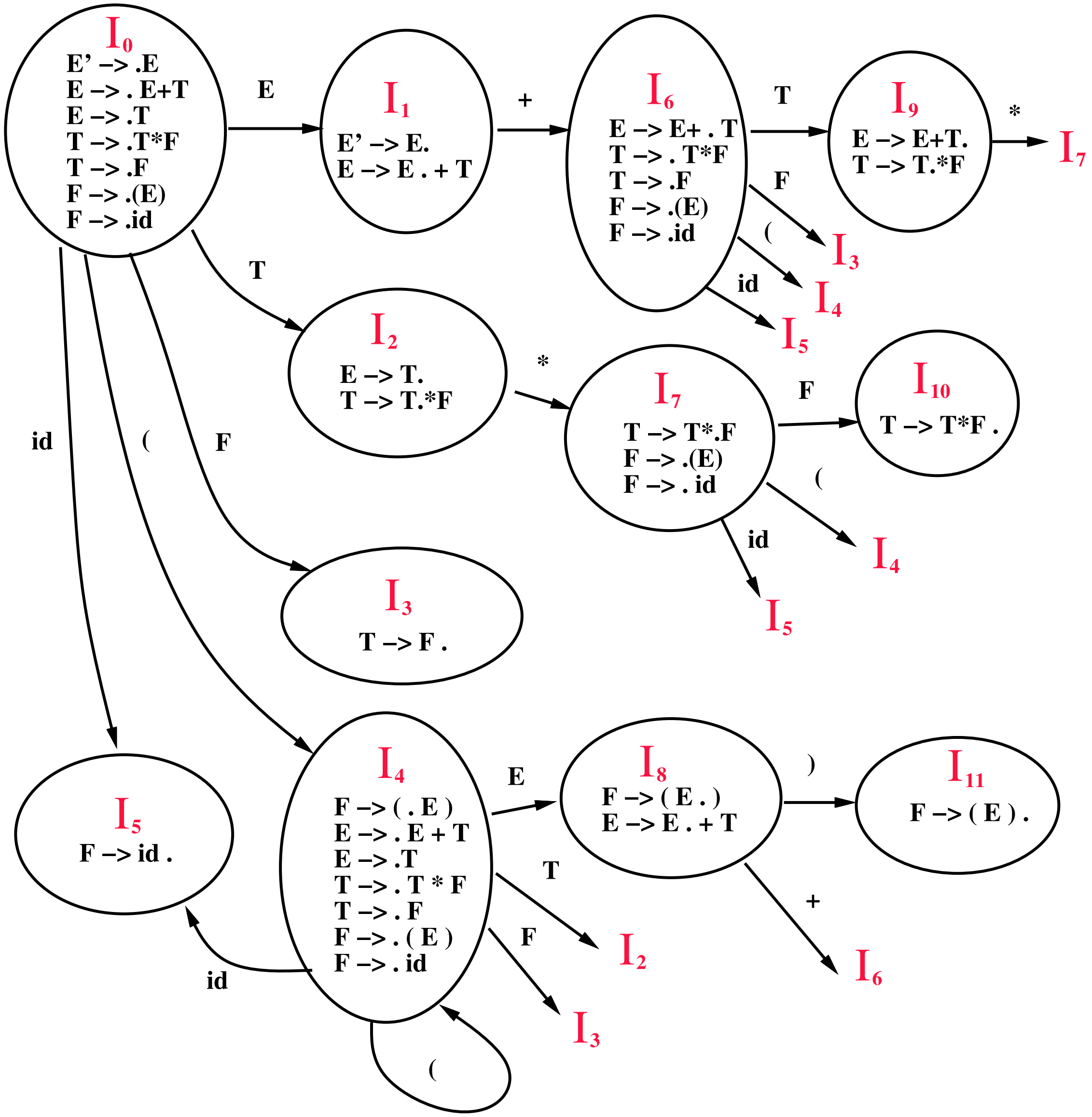
$T \rightarrow \cdot F,$

$F \rightarrow \cdot (E),$

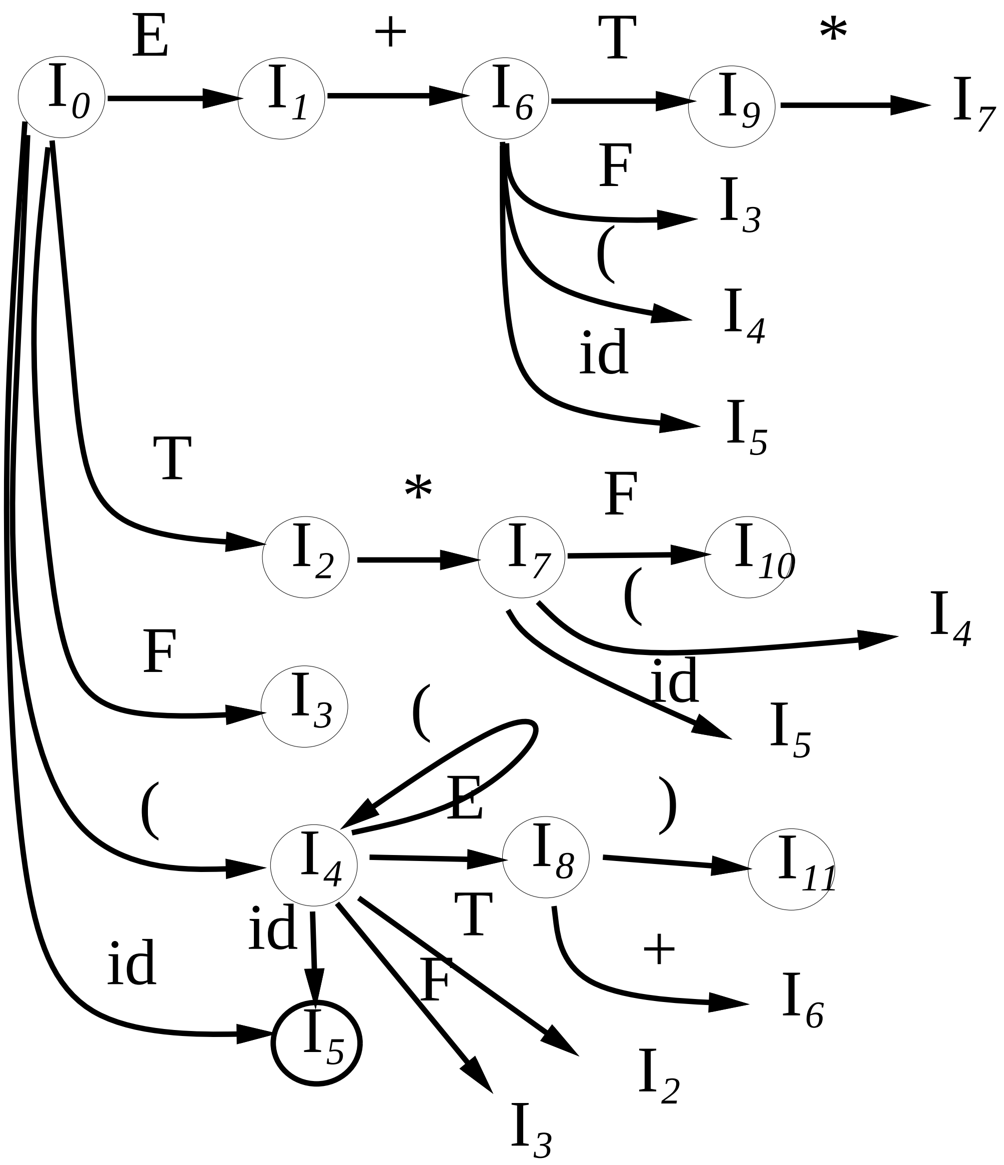
$F \rightarrow \cdot id\}$

- **Canonical $LR(0)$ items:**
  - $I_1 = GOTO(I_0, E) =$
    - ▷ $\{E' \rightarrow E\cdot,$
    - ▷ $E \rightarrow E \cdot +T\}$
  - $I_2 = GOTO(I_0, T) =$
    - ▷ $\{E \rightarrow T\cdot,$
    - ▷ $T \rightarrow T \cdot *F\}$

Transition diagram (1/2)



Transition diagram (2/2)



Meaning of $LR(0)$ transition diagram

- $E + T*$ is a viable prefix that can happen on the top of the stack while doing parsing.
- after seeing $E + T*$, we are in state I_7. I_7 =
  - $\{T \rightarrow T * \cdot F,$
  - $F \rightarrow \cdot(E),$
  - $F \rightarrow \cdot id\}$
- We expect to follow one of the following three possible derivations:

$$\begin{aligned} E' &\underset{rm}{\Longrightarrow} E \\ &\underset{rm}{\Longrightarrow} E + T \\ &\underset{rm}{\Longrightarrow} E + T * F \\ &\underset{rm}{\Longrightarrow} E + T * id \\ &\underset{rm}{\Longrightarrow} \underline{E + \underline{T*}}F * id \\ &\cdots \end{aligned}$$

$$\begin{aligned} E' &\underset{rm}{\Longrightarrow} E \\ &\underset{rm}{\Longrightarrow} E + T \\ &\underset{rm}{\Longrightarrow} E + T * F \\ &\underset{rm}{\Longrightarrow} \underline{E + T*}(E) \\ &\cdots \end{aligned}$$

$$\begin{aligned} E' &\underset{rm}{\Longrightarrow} E \\ &\underset{rm}{\Longrightarrow} E + T \\ &\underset{rm}{\Longrightarrow} E + T * F \\ &\underset{rm}{\Longrightarrow} \underline{E + T*}id \\ &\cdots \end{aligned}$$

Meanings of $closure(I)$ and $GOTO(I, X)$

- $closure(I)$: **a state/configuration during parsing recording all possible information about the next handle.**
  - **If $A \rightarrow \alpha \cdot B\beta \in I$, then it means**
    - ▷ ***in the middle of parsing, α is on the top of the stack;***
    - ▷ ***at this point, we are expecting to see $B\beta$;***
    - ▷ ***after we saw $B\beta$, we will reduce $\alpha B\beta$ to A and make A top of stack.***
  - **To achieve the goal of seeing $B\beta$, we expect to perform some operations below:**
    - ▷ ***We expect to see B on the top of the stack first.***
    - ▷ ***If $B \rightarrow \gamma$ is a production, then it might be the case that we shall see γ on the top of the stack.***
    - ▷ ***If it does, we reduce γ to B.***
    - ▷ ***Hence we need to include $B \rightarrow \cdot\gamma$ into $closure(I)$.***
- $GOTO(I, X)$: **when we are in the state described by I, and then a new symbol X is pushed into the stack,**
  - **If $A \rightarrow \alpha \cdot X\beta$ is in I, then $closure(\{A \rightarrow \alpha X \cdot \beta\}) \subseteq GOTO(I, X)$.**

Parsing example

STACK	input	action
\$ I_0	id*id+id\$	shift 5
\$ I_0 id I_5	* id + id\$	reduce by $F \rightarrow id$
\$ I_0 F	* id + id\$	in I_0, saw F, goto I_3
\$ I_0 F I_3	* id + id\$	reduce by $T \rightarrow F$
\$ I_0 T	* id + id\$	in I_0, saw T, goto I_2
\$ I_0 T I_2	* id + id\$	shift 7
\$ I_0 T I_2 * I_7	id + id\$	shift 5
\$ I_0 T I_2 * I_7 id I_5	+ id\$	reduce by $F \rightarrow id$
\$ I_0 T I_2 * I_7 F	+ id\$	in I_7, saw F, goto I_{10}
\$ I_0 T I_2 * I_7 F I_{10}	+ id\$	reduce by $T \rightarrow T * F$
\$ I_0 T	+ id\$	in I_0, saw T, goto I_2
\$ I_0 T I_2	+ id\$	reduce by $E \rightarrow T$
\$ I_0 E	+ id\$	in I_0, saw E, goto I_1
\$ I_0 E I_1	+ id\$	shift 6
\$ I_0 E I_1 + I_6	id\$	shift 5
\$ I_0 E I_1 + I_6 F	\$	reduce by $F \rightarrow id$
...	...	...

$LR(0)$ parsing

- LR parsing without lookahead symbols.
- Constructed a DPDA to recognize viable prefixes.
- In state I_i
  - if $A \rightarrow \alpha \cdot a\beta$ is in I_i then perform "shift" while seeing the terminal a in the input, and then go to the state $closure(\{A \rightarrow \alpha a \cdot \beta\})$
  - if $A \rightarrow \beta\cdot$ is in I_i, then perform "reduce by $A \rightarrow \beta$" and then go to the state $GOTO(I, A)$ where I is the state on the top of the stack after removing β
- Conflicts: handles have overlap
  - shift/reduce conflict
  - reduce/reduce conflict
- Very few grammars are $LR(0)$. For example:
  - In I_2, you can either perform a reduce or a shift when seeing "*" in the input
  - However, it is not possible to have E followed by "*". Thus we should not perform "reduce".
  - Use FOLLOW(E) as look ahead information to resolve some conflicts.

$SLR(1)$ parsing algorithm

- **Using FOLLOW sets to resolve conflicts in constructing $SLR(1)$ parsing table, where the first "S" stands for "Simple".**
  - **Input: an augmented grammar G'**
  - **Output: the $SLR(1)$ parsing table**
- **Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of $LR(0)$ items for G'.**
- **The parsing table for state I_i is determined as follows:**
  - **If $A \rightarrow \alpha \cdot a\beta$ is in I_i and $GOTO(I_i, a) = I_j$, then $action(I_i, a)$ is "shift j" for a being a terminal.**
  - **If $A \rightarrow \alpha\cdot$ is in I_i, then $action(I_i, a)$ is "reduce by $A \rightarrow \alpha$" for all terminal $a \in$ FOLLOW(A); here $A \neq S'$**
  - **If $S' \rightarrow S\cdot$ is in I_i, then $action(I_i, \$)$ is "accept".**
- **If any conflicts are generated by the above algorithm, we say the grammar is not $SLR(1)$.**

$SLR(1)$ parsing table

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

	action						GOTO		
state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r5	r5		r5	r5			
4	s5			s4			8	2	3
5		r7	r7		r7	r7			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r2	s7		r2	r2			
10		r4	r4		r4	r4			
11		r6	r6		r6	r6			

- **ri means reduce by the ith production.**
- **si means shift and then go to state I_i.**
- **Use FOLLOW sets to resolve some conflicts.**

Discussion (1/3)

- **Every $SLR(1)$ grammar is unambiguous, but there are many unambiguous grammars that are not $SLR(1)$.**
- **Grammar:**
  - $S \rightarrow L = R \mid R$
  - $L \rightarrow *R \mid id$
  - $R \rightarrow L$
- **States:**

I_0:
- ▷ $S' \rightarrow \cdot S$
- ▷ $S \rightarrow \cdot L = R$
- ▷ $S \rightarrow \cdot R$
- ▷ $L \rightarrow \cdot * R$
- ▷ $L \rightarrow \cdot id$
- ▷ $R \rightarrow \cdot L$

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:
- ▷ $L \rightarrow * \cdot R$
- ▷ $R \rightarrow \cdot L$
- ▷ $L \rightarrow \cdot * R$
- ▷ $L \rightarrow \cdot id$

I_5: $L \rightarrow id\cdot$

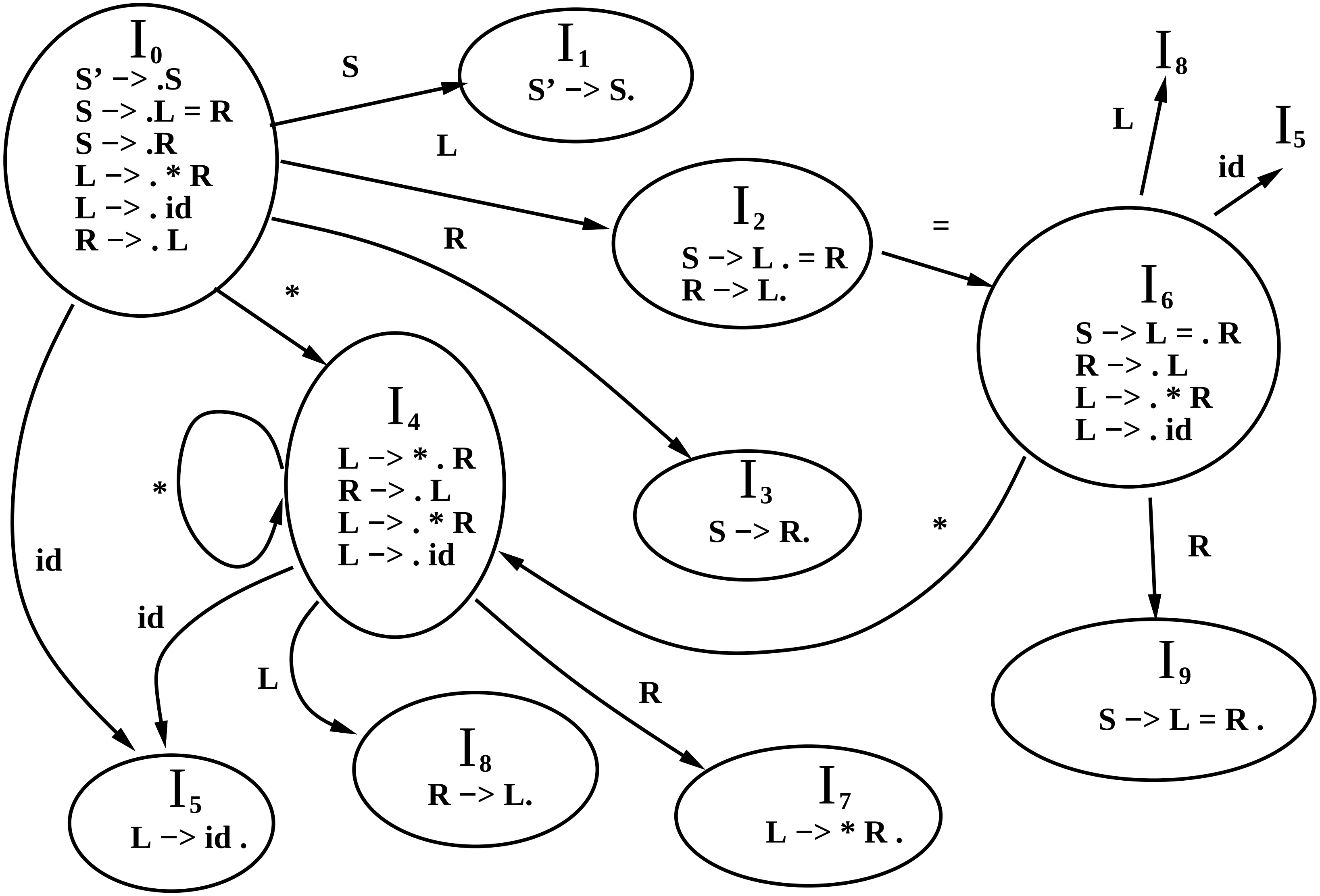
I_6:
- ▷ $S \rightarrow L = \cdot R$
- ▷ $R \rightarrow \cdot L$
- ▷ $L \rightarrow \cdot * R$
- ▷ $L \rightarrow \cdot id$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

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

Discussion (2/3)



Discussion (3/3)

- **Suppose the stack has $\$I_0LI_2$ and the input is "=". We can either**
  - **shift 6, or**
  - **reduce by $R \rightarrow L$, since $= \in$ FOLLOW(R).**
- **This grammar is ambiguous for $SLR(1)$ parsing.**
- **However, we should not perform a $R \rightarrow L$ reduction.**
  - **After performing the reduction, the viable prefix is $\$R$;**
  - **$= \notin$ FOLLOW$(\$R)$;**
  - **$= \in$ FOLLOW$(*R)$;**
  - **That is to say, we cannot find a right-sentential form with the prefix $R = \cdots$.**
  - **We can find a right-sentential form with $\cdots * R = \cdots$**

Canonical LR — $LR(1)$

- In $SLR(1)$ parsing, if $A \rightarrow \alpha\cdot$ is in state I_i, and $a \in$ FOLLOW(A), then we perform the reduction $A \rightarrow \alpha$.
- However, it is possible that when state I_i is on the top of the stack, we have viable prefix $\beta\alpha$ on the top of the stack, and βA cannot be followed by a.
  - In this case, we cannot perform the reduction $A \rightarrow \alpha$.
- It looks difficult to find the FOLLOW sets for every viable prefix.
- We can solve the problem by knowing more left context using the technique of lookahead propagation .

$LR(1)$ items

- **An $LR(1)$ item is in the form of $[A \rightarrow \alpha \cdot \beta, a]$, where the first field is an $LR(0)$ item and the second field a is a terminal belonging to a subset of FOLLOW(A).**
- **Intuition: perform a reduction based on an $LR(1)$ item $[A \rightarrow \alpha \cdot, a]$ only when the next symbol is a.**
- **Formally: $[A \rightarrow \alpha \cdot \beta, a]$ is valid (or reachable) for a viable prefix γ if there exists a derivation**

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \delta A \omega \underset{rm}{\Longrightarrow} \underbrace{\delta \ \ \alpha}_{\gamma} \beta \ \ \omega,$$

where

- **either $a \in$ FIRST(ω) or**
- **$\omega = \epsilon$ and $a = \$$.**

Examples of $LR(1)$ items

- **Grammar:**
  - $S \rightarrow BB$
  - $B \rightarrow aB \mid b$
-
$$S \overset{*}{\underset{rm}{\Longrightarrow}} aaBab \underset{rm}{\Longrightarrow} aaaBab$$
viable prefix aaa **can reach** $[B \rightarrow a \cdot B, a]$
-
$$S \overset{*}{\underset{rm}{\Longrightarrow}} BaB \underset{rm}{\Longrightarrow} BaaB$$
viable prefix Baa **can reach** $[B \rightarrow a \cdot B, \$]$

Finding all $LR(1)$ items

- **Ideas: redefine the closure function.**
  - Suppose $[A \rightarrow \alpha \cdot B\beta, a]$ is valid for a viable prefix $\gamma \equiv \delta\alpha$.
  - In other words,

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \delta A a \omega \underset{rm}{\Longrightarrow} \delta\alpha B\beta a\omega.$$

  - Then for each production $B \rightarrow \eta$, assume $\beta a\omega$ derives the sequence of terminals bc.

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \delta\alpha B \boxed{\beta a\omega} \overset{*}{\underset{rm}{\Longrightarrow}} \delta\alpha B \boxed{bc} \overset{*}{\underset{rm}{\Longrightarrow}} \delta\alpha \boxed{\eta} bc$$

  Thus $[B \rightarrow \cdot\eta, b]$ is also valid for γ for each $b \in$ FIRST(βa).
  Note a is a terminal. So FIRST$(\beta a) =$ FIRST$(\beta a\omega)$.

- **Lookahead propagation** .

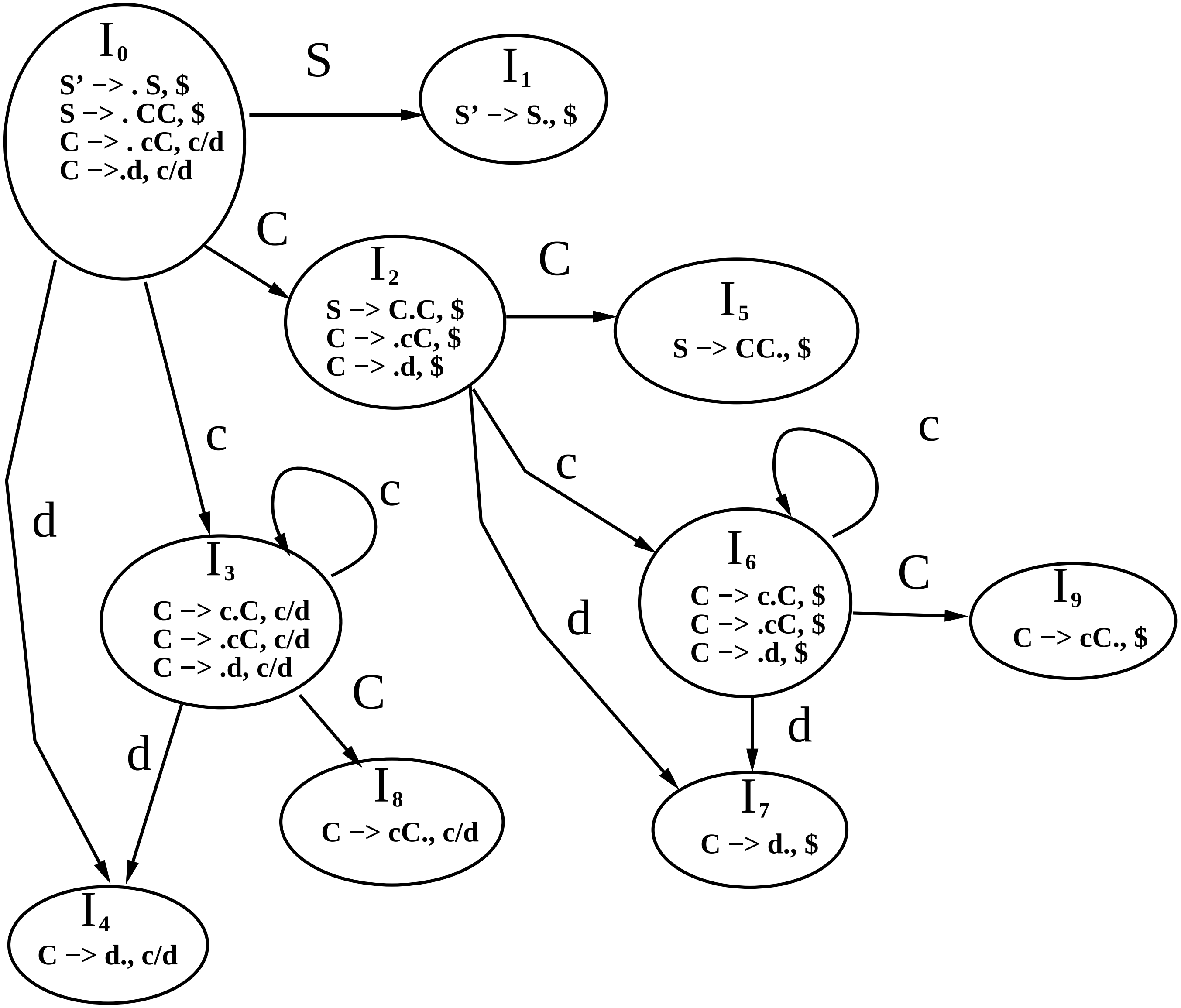
Algorithm for $LR(1)$ parsers

- $closure_1(I)$
  - **repeat**
    ```
    ▷ for each item $[A \rightarrow \alpha \cdot B\beta, a]$ in $I$ do
    ▷         if $B \rightarrow \cdot\eta$ is in $G'$
    ▷         then add $[B \rightarrow \cdot\eta, b]$ to $I$ for each $b \in$ FIRST$(\beta a)$
    ```
  - **until no more items can be added to** I
  - **return** I
- $GOTO_1(I, X)$
  - **let** $J = \{[A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X\beta, a] \in I\}$**;**
  - **return** $closure_1(J)$
- $items(G')$
  - $C \leftarrow \{closure_1(\{[S' \rightarrow \cdot S, \$]\})\}$
  - **repeat**
    ```
    ▷ for each set of items $I \in C$ and each grammar symbol $X$ such that
      $GOTO_1(I, X) \neq \emptyset$ and $GOTO_1(I, X) \notin C$ do
    ▷         add $GOTO_1(I, X)$ to $C$
    ```
  - **until no more sets of items can be added to** C

Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \rightarrow S$
  - $S \rightarrow CC$
  - $C \rightarrow cC \mid d$
- $closure_1(\{[S' \rightarrow \cdot S, \$]\})$ =
  - $\{[S' \rightarrow \cdot S, \$],$
  - $[S \rightarrow \cdot CC, \$],$
  - $[C \rightarrow \cdot cC, c/d],$
  - $[C \rightarrow \cdot d, c/d]\}$
- **Note:**
  - **FIRST**$(\epsilon\$) = \{\$\}$
  - **FIRST**$(C\$) = \{c, d\}$
  - $[C \rightarrow \cdot cC, c/d]$ **means**
    - ▷ $[C \rightarrow \cdot cC, c]$ ***and***
    - ▷ $[C \rightarrow \cdot cC, d]$**.**

$LR(1)$ transition diagram



$LR(1)$ parsing example

- **Input** $cdccd$

STACK	INPUT	ACTION
\$ I_0	cdccd\$	
\$ I_0 c I_3	dccd\$	shift 3
\$ I_0 c I_3 d I_4	ccd\$	shift 4
\$ I_0 c I_3 C I_8	ccd\$	reduce by $C \rightarrow d$
\$ I_0 C I_2	ccd\$	reduce by $C \rightarrow cC$
\$ I_0 C I_2 c I_6	cd\$	shift 6
\$ I_0 C I_2 c I_6 c I_6	d\$	shift 6
\$ I_0 C I_2 c I_6 c I_6	d\$	shift 6
\$ I_0 C I_2 c I_6 c I_6 d I_7	\$	shift 7
\$ I_0 C I_2 c I_6 c I_6 C I_9	\$	reduce by $C \rightarrow cC$
\$ I_0 C I_2 c I_6 C I_9	\$	reduce by $C \rightarrow cC$
\$ I_0 C I_2 C I_5	\$	reduce by $S \rightarrow CC$
\$ I_0 S I_1	\$	reduce by $S' \rightarrow S$
\$ I_0 S'	\$	accept

Generating $LR(1)$ parsing table

- **Construction of canonical $LR(1)$ parsing tables.**
  - **Input: an augmented grammar G'**
  - **Output: the canonical $LR(1)$ parsing table, i.e., the $ACTION_1$ table**
- **Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of $LR(1)$ items form G'.**
- **Action table is constructed as follows:**
  - **if $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ and $GOTO_1(I_i, a) = I_j$, then $action_1[I_i, a] =$ "shift j" for a is a terminal.**
  - **if $[A \rightarrow \alpha\cdot, a] \in I_i$ and $A \neq S'$, then $action_1[I_i, a] =$ "reduce by $A \rightarrow \alpha$"**
  - **if $[S' \rightarrow S\cdot, \$] \in I_i$, then $action_1[I_i, \$] =$ "accept."**
- **If conflicts result from the above rules, then the grammar is not $LR(1)$.**
- **The initial state of the parser is the one constructed from the set containing the item $[S' \rightarrow \cdot S, \$]$.**

Example of an $LR(1)$ parsing table

	action_1			GOTO_1	
state	c	d	\$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- **Canonical $LR(1)$ parser:**
  - **Most powerful!**
  - **Has too many states and thus occupy too much space.**

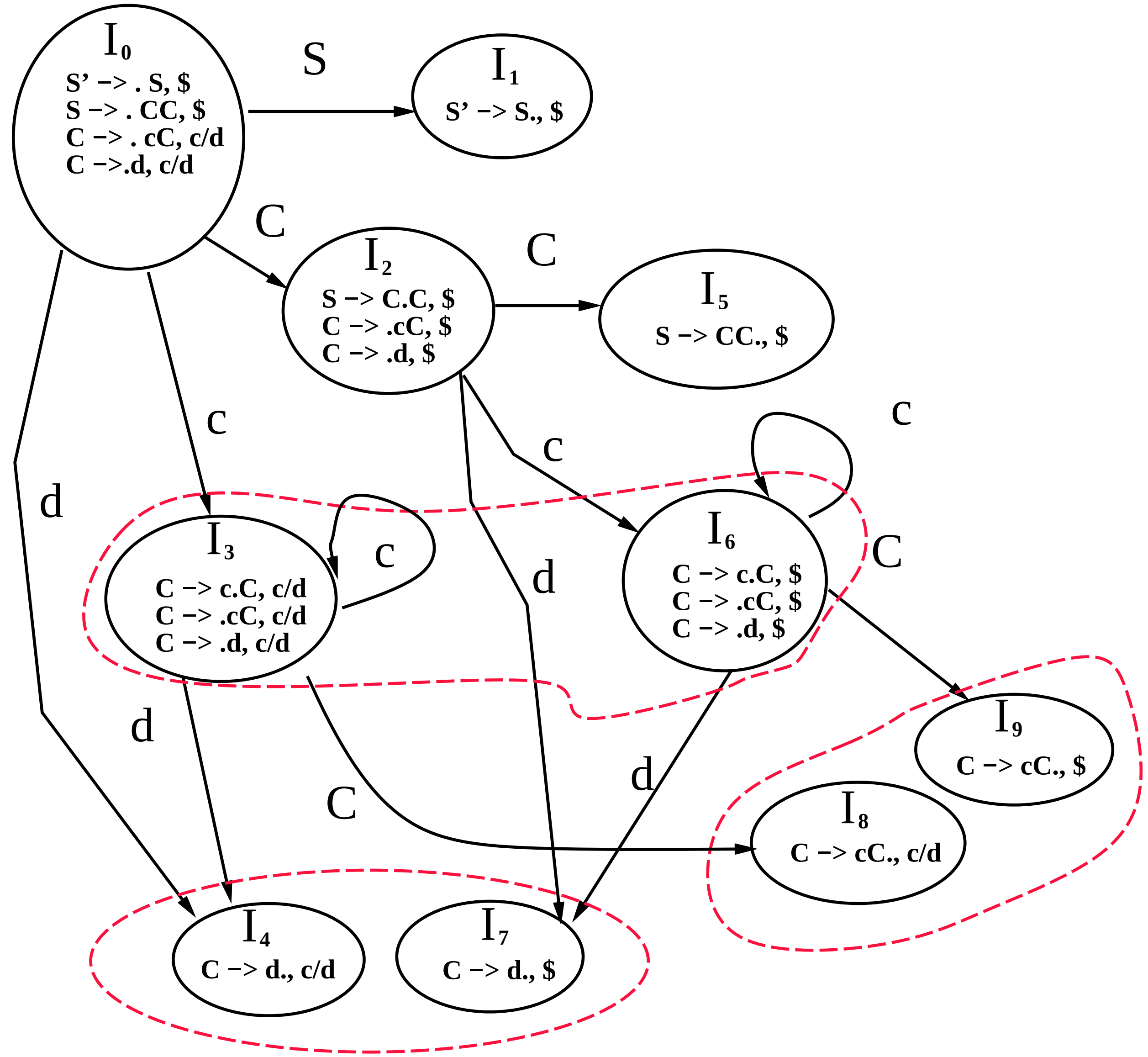
$LALR(1)$ parser — Lookahead LR

- The method that is often used in practice.
- Most common syntactic constructs of programming languages can be expressed conveniently by an $LALR(1)$ grammar.
- $SLR(1)$ and $LALR(1)$ always have the same number of states.
- Number of states is about 1/10 of that of $LR(1)$.
- Simple observation:
  - an $LR(1)$ item is of the form $[A \rightarrow \alpha \cdot \beta, c]$
- We call $A \rightarrow \alpha \cdot \beta$ the first component.
- Definition: in an $LR(1)$ state, set of first components is called its core.

Intuition for $LALR(1)$ grammars

- **In an $LR(1)$ parser, it is a common thing that several states only differ in lookahead symbols, but have the same core.**
- **To reduce the number of states, we might want to merge states with the same core.**
    - **If I_4 and I_7 are merged, then the new state is called $I_{4,7}$**
- **After merging the states, revise the $GOTO_1$ table accordingly.**
- **Merging of states can never produce a shift-reduce conflict that was not present in one of the original states.**
    - $I_1 = \{[A \rightarrow \alpha \cdot, a], \ldots\}$
    - $I_2 = \{[B \rightarrow \beta \cdot a\gamma, b], \ldots\}$
    - **For I_1, we perform a reduce on a.**
    - **For I_2, we perform a shift on a.**
    - **Merging I_1 and I_2, the new state $I_{1,2}$ has shift-reduce conflicts.**
    - **This is impossible!**
    - **In the original table, I_1 and I_2 have the same core.**
    - $[A \rightarrow \alpha \cdot, c] \in I_2$ **and** $[B \rightarrow \beta \cdot a\gamma, d] \in I_1$**.**
    - **The shift-reduce conflict already occurs in I_1 and I_2.**

$LALR(1)$ transition diagram



Possible new conflicts from $LALR(1)$

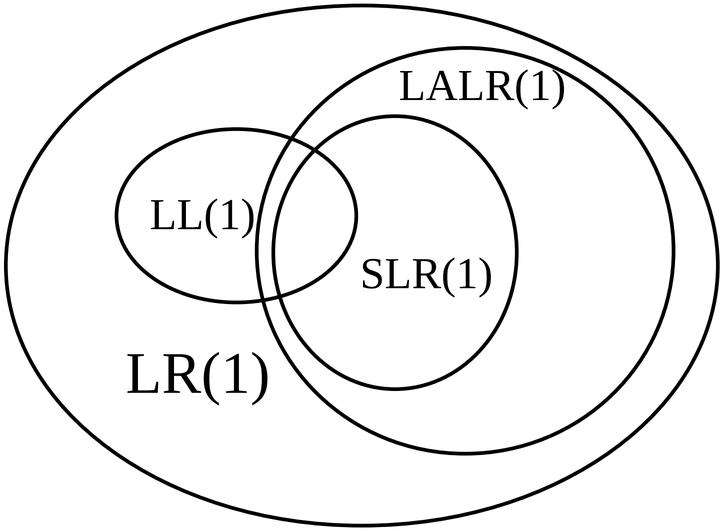
- May produce a new reduce-reduce conflict.
- For example (textbook page 238), grammar:
  - $S' \rightarrow S$
  - $S \rightarrow aAd \mid bBf \mid aBe \mid bAe$
  - $A \rightarrow c$
  - $B \rightarrow c$
- The language recognized by this grammar is $\{acd, ace, bcd, bce\}$.
- You may check that this grammar is $LR(1)$ by constructing the sets of items.
- You will find the set of items $\{[A \rightarrow c\cdot, d], [B \rightarrow c\cdot, e]\}$ is valid for the viable prefix ac, and $\{[A \rightarrow c\cdot, e], [B \rightarrow c\cdot, d]\}$ is valid for the viable prefix bc.
- Neither of these sets generates a conflict, and their cores are the same. However, their union, which is
  - $\{[A \rightarrow c\cdot, d/e],$
  - $[B \rightarrow c\cdot, d/e]\}$

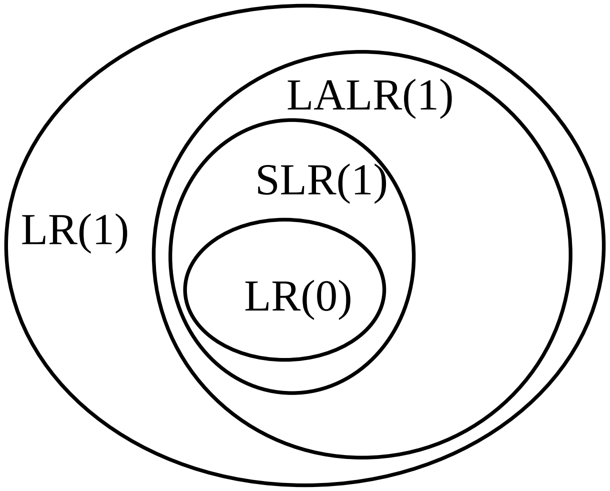
  generates a reduce-reduce conflict, since reductions by both $A \rightarrow c$ and $B \rightarrow c$ are called for on inputs d and e.

How to construct $LALR(1)$ parsing table

- **Naive approach:**
  - **Construct $LR(1)$ parsing table, which takes lots of intermediate spaces.**
  - **Merging states.**
- **Space efficient methods to construct an $LALR(1)$ parsing table are known.**
  - **Constructing and merging on the fly.**

Summary





- $LR(1)$ **and** $LALR(1)$ **can almost handle all programming languages, but** $LALR(1)$ **is easier to write and uses much less space.**
- $LL(1)$ **is easier to understand, but cannot handle several important common-language features.**