Syntax Analyzer — Parser

ALSU Textbook Chapter 4.1–4.7

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 $\longrightarrow$ parser $\longrightarrow$ 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 a legal program.**
- **Dealing with errors.**
  - **Syntactic errors.**
  - **Static semantic errors.**
    - ▷ ***Example: a variable is not declared or declared twice in a language where a variable must be declared before its usage.***

Error handling

- **Goals:**
  - **Report errors clearly and accurately.**
  - **Recover from errors quickly enough to detect subsequent errors.**
  - **Spend minimal overhead.**
- **Strategies:**
  - **Panic-mode recovery: skip until synchronizing tokens are found.**
    - ▷ *"";" marks the end of a C-sentence;*
    - ▷ *"}" closes a C-scope.*
  - **Phrase-level recovery: perform local correction and then continue.**
    - ▷ *Assume a un-declared variable is declared with the default type "int."*
  - **Error productions: anticipating common errors using grammars.**
    - ▷ *Example: write a grammar rule for the case when ";" is missing between two var-declarations in C.*
  - **Global correction: choose a minimal sequence of changes to obtain a globally least-cost correction.**
    - ▷ *A very difficult task!*
    - ▷ *May have more than one interpretations.*
    - ▷ *C example: In "$y = *x;$", whether an operand is missing in multiplication or the type of x should be pointer?*

Context free grammar (CFG)

- **Definitions:** $G = (T, N, P, S)$.
  - ▷ ***T: a set of terminals;***
  - ▷ ***N: a set of nonterminals;***
  - ▷ ***P: productions of the form***
    $A \to \alpha_1 \alpha_2 \cdots \alpha_m$, where $A \in N$ and $\alpha_i \in T \cup N$;
  - ▷ ***S: the starting nonterminal where $S \in N$.***
- **Notations:**
  - **terminals : strings with lower-cased English letters and printable characters.**
    - ▷ ***Examples:*** a, b, c, int ***and*** int_1.
  - **nonterminals: strings started with an upper-cased English letter.**
    - ▷ ***Examples:*** A, B, C ***and*** $Procedure$.
  - $\alpha, \beta, \gamma, \ldots \in (T \cup N)^*$
    - ▷ α, β, γ ***and*** ϵ***: alpha, beta, gamma and epsilon.***
  - $$\left.\begin{array}{lcl} A & \to & \alpha_1 \\ A & \to & \alpha_2 \end{array}\right\} \equiv A \to \alpha_1 \mid \alpha_2$$

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:

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

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

- **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";**
  - $\stackrel{+}{\Longrightarrow}$**: means "derives in one or more steps";**
  - $\stackrel{*}{\Longrightarrow}$**: means "derives in zero or more steps";**
  - **In the above example, we can write** $E \stackrel{+}{\Longrightarrow} 1 - 4/2$**.**

Language

- The **language** defined by a grammar G is

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

  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.*

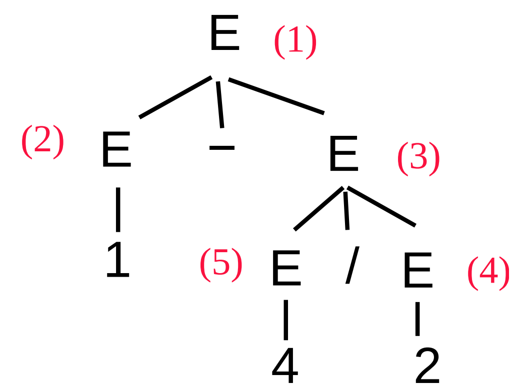
A way to describe derivations

- 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
- This is called **top-down** parsing or expanding of the parse tree.
  - Construct the parse tree starting from the root.
  - Other parsing methods, such as **bottom-up**, are known.

Top-down parsing

- **Need to annotate the order of derivation on the nodes.**

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



- **It is better to keep a systematic order in parsing for the sake of performance or ease-to-understand.**
  - **leftmost**
  - **rightmost**

Parse tree examples

- **Example:**

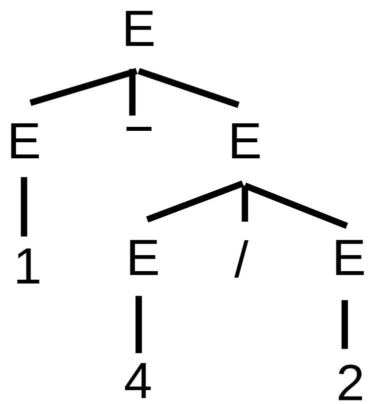
Grammar:

$E \rightarrow int$

$E \rightarrow E - E$

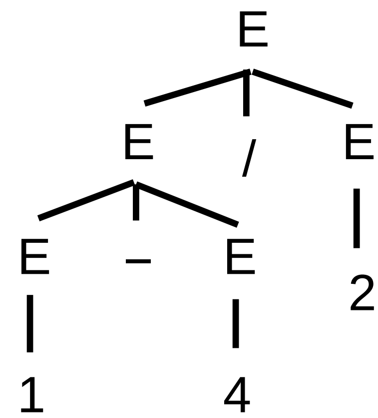
$E \rightarrow E/E$

$E \rightarrow (E)$



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 un-annotated 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 previous 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.***

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:**
    - ▷ $\textbf{\textit{Procedure}} \rightarrow \textbf{\textit{TypeDef}}\ \ id\ \ \textbf{\textit{OptParams}}\ \textbf{\textit{OptDecl}}\ \{\textbf{\textit{OptStatements}}\}$
    - ▷ $\textbf{\textit{TypeDef}} \rightarrow integer \mid char \mid float \mid \cdots$
    - ▷ $\textbf{\textit{OptParams}} \rightarrow (\ \textbf{\textit{ListParams}}\ )$
    - ▷ $\textbf{\textit{ListParams}} \rightarrow \epsilon \mid \textbf{\textit{NonEmptyParList}}$
    - ▷ $\textbf{\textit{NonEmptyParList}} \rightarrow \textbf{\textit{NonEmptyParList}}, id \mid id$
    - ▷ $\cdots$
- **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.**

Common grammar problems (CGP)

- **A grammar may have some bad "styles" or ambiguity.**
- **Some common grammar problems (CGP's) are:**
  - **Useless terms;**
  - **Dangling-else ambiguity;**
  - **Left factor;**
  - **Left recursion.**
- **Need to rewrite a grammar G_1 into another grammar G_2 so that G_2 has no CGP's and the two grammars are equivalent and G_2 contains no CGP's.**
  - **G_1 and G_2 must accept the same set of strings, that is, $L(G_1) = L(G_2)$.**
  - **The "semantic" of a given string α must stay the same using G_2.**
    - ▷ ***The "main structure" of the parse tree needs to stay unchanged.***

CGP: useless terms

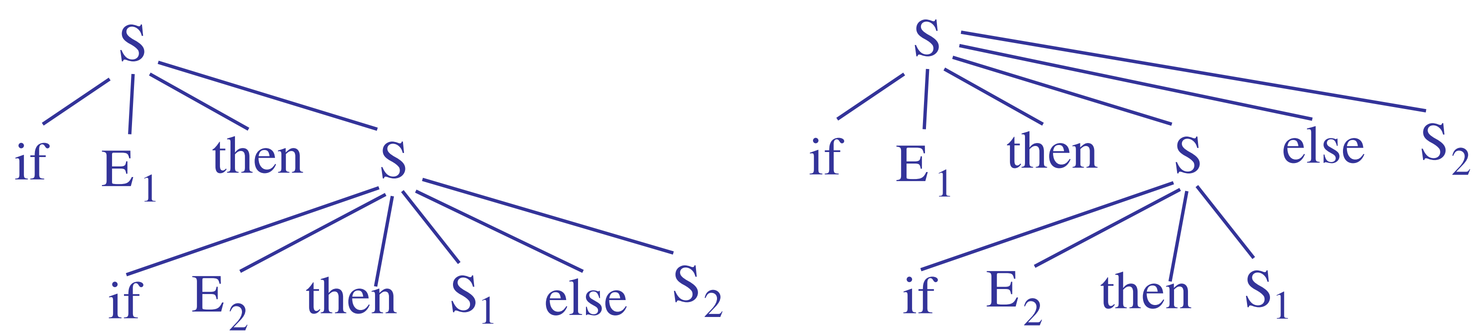
- **A nonterminal X is useless if either**
  - **a sequence includes X cannot be derived from the starting nonterminal, or**
  - **no string can be derived starting from X, where a string means ϵ or a sequence of terminals.**
- **Example 1:**
  - $S \rightarrow A\ B$
  - $A \rightarrow + \mid - \mid \epsilon$
  - $B \rightarrow digit \mid B\ digit$
  - $C \rightarrow .\ B$
- **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!**
- **From now on, we assume a grammar contains no useless nonterminals.**
- **Q: How to detect and remove indirect useless terms?**

CGP: dangling-else (1/2)

- **Example:**
  - G_1
    - ▷ $S \rightarrow if\ E\ then\ S$
    - ▷ $S \rightarrow if\ E\ then\ S\ else\ S$
    - ▷ $S \rightarrow Others$
  - **Input:** $\boxed{\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2}$
  - G_1 **is ambiguous given the above input.**
    - ▷ ***Have two parse trees.***



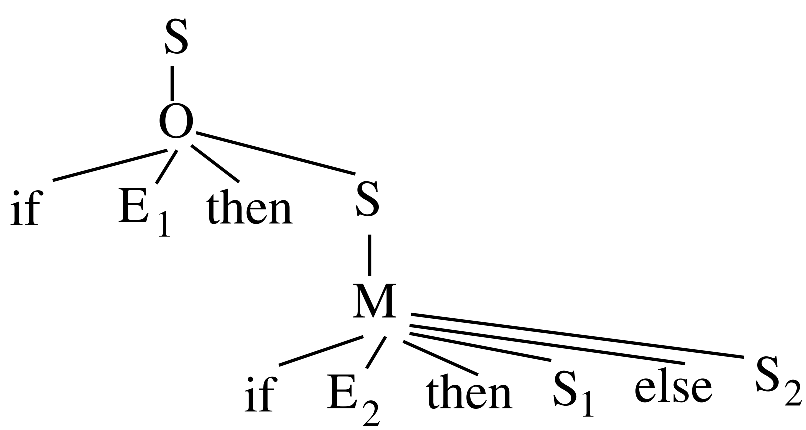
- **Dangling-else ambiguity.**
  - **General rule: Match each "else" with the closest unmatched "then."**

CGP: dangling-else (2/2)

- **Rewrite G_1 into the following:**
  - G_2
    - $S \rightarrow M \mid O$
    - $M \rightarrow if\ E\ then\ M\ else\ M \mid Others$
    - $O \rightarrow if\ E\ then\ S$
    - $O \rightarrow if\ E\ then\ M\ else\ O$
  - **Only one parse tree for the input**

    if E_1 then if E_2 then S_1 else S_2

    using grammar G_2.
  - **Intuition: "else" is matched with the nearest "then."**



CGP: left factor

- **Left factor: a grammar G has two productions whose right-hand-sides have a common prefix.**
  - ▷ *Have left-factors.*
  - ▷ *Potentially difficult to parse in a top-down fashion, but may not have ambiguity.*
- **Example: $S \rightarrow \{S\} \mid \{\}$**
  - ▷ *In this example, the common prefix is "{".*
- **This problem can be solved by using the left-factoring trick.**

• $A \rightarrow \alpha\beta_1$ • $A \rightarrow \alpha\beta_2$	**transform to**	• $A \rightarrow \alpha A'$ • $A' \rightarrow \beta_1 \mid \beta_2$

- **Example:**

• $S \rightarrow \{S\}$ • $S \rightarrow \{\}$	**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**
    - ▷ $A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$

    with
    - ▷ $A \rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m$
    - ▷ $A' \rightarrow \beta_1 \mid \cdots \mid \beta_n$
  - **repeat the above step until the current grammar has no two productions with a common prefix;**
- **Example:**
  - $S \rightarrow aaWaa \mid aaaa \mid aaTcc \mid bb$
  - **Transform to**
    - ▷ $S \rightarrow aaS' \mid bb$
    - ▷ $S' \rightarrow Waa \mid aa \mid Tcc$

CGP: left recursion

- **Definitions:**
  - **recursive grammar: a grammar is recursive if this grammar contains a nonterminal X such that**
    - $X \stackrel{+}{\Longrightarrow} \alpha X \beta$.
  - G **is immediately left-recursive if** $X \Longrightarrow X\beta$.
  - G **is left-recursive if** $X \stackrel{+}{\Longrightarrow} X\beta$.
- **Why left recursion is bad?**
  - **Potentially difficult to parse if you read input from left to right.**
  - **Difficult to know when recursion should be stopped.**
- **Remark: A left-recursived grammar cannot be parsed efficiently by a top-down parser, but may have no ambiguity.**

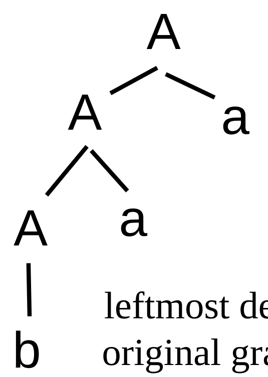
Removing immediate left-recursion (1/3)

- **Algorithm:**
  - **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,* $L(G) \equiv L(G')$.

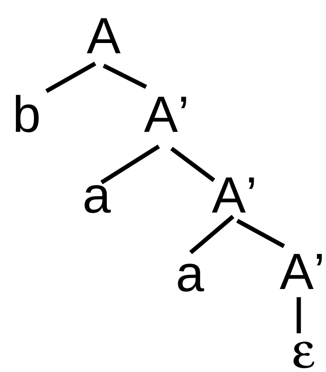
Removing immediate left-recursion (2/3)

- **Example:**
  - **Grammar G:**
    - ▷ $A \rightarrow Aa \mid b$
  - **Revised grammar G':**
    - ▷ $A \rightarrow bA'$
    - ▷ $A' \rightarrow aA' \mid \epsilon$
  - **The above two grammars are equivalent.**
    - ▷ ***That is,*** $L(G) \equiv L(G')$.
- **Parsing example:**

input $\boxed{baa}$



leftmost derivation
original grammar G



leftmost derivation
revised grammar G’

Removing immediate left-recursion (3/3)

- **Both grammars recognize the same string, but G' is not left-recursive.**
- **However, G is clear and intuitive.**
- **General algorithm for removing immediately left-recursion:**
  - **Replace** $A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$
  - **with**
    - ▷ $A \rightarrow \beta_1 A' \mid \cdots \mid \beta_m A'$
    - ▷ $A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \epsilon$
- **This rule does not work if $\alpha_i = \epsilon$ for some i.**
  - **This is called a *direct cycle* in a grammar.**
    - ▷ ***A direct cycle:*** $X \Longrightarrow X$.
    - ▷ ***A cycle:*** $X \stackrel{+}{\Longrightarrow} X$.
  - **Q: why do you need to define direct cycles or cycles?**
- **May need to worry about whether the semantics are equivalent between the original grammar and the transformed grammar.**

Removing left recursion: Algorithm 4.19

- **Algorithm 4.19 systematically eliminates left recursion and works when the input grammar has no cycles or ϵ-productions.**
  - ▷ *Cycle:* $A \stackrel{+}{\Longrightarrow} A$
  - ▷ *ϵ-production:* $A \rightarrow \epsilon$
  - ▷ *Can remove cycles and all but one ϵ-production using other algorithms.*

- **Input: grammar G without cycles and ϵ-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* $\boxed{A_i \rightarrow A_j\gamma}$ *with* $\boxed{A_i \rightarrow \delta_1\gamma \mid \cdots \mid \delta_k\gamma}$*, 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 4.19 — 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?**
  - The algorithm of removing immediate left-recursion cannot handle direct cycles.
  - A cycle becomes a direct cycle during the process of substituting nonterminals.
- **Why ϵ-productions are not allowed?**
  - Inside the loop, when $A_j \rightarrow \epsilon$,
    - ▷ ***that is some*** $\delta_g = \epsilon$***,***
    - ▷ ***and the prefix of*** γ ***is some*** A_k ***where*** $k < i$***,***
    - ▷ ***it generates*** $A_i \rightarrow A_k$***, and*** $i > k$***.***
- **Time and space complexities:**
  - The size may be blowed up exponentially.
  - Works well in real cases.

Trace an instance of Algorithm 4.19

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

Example

- **Original Grammar:**
  - **(1)** $S \to Aa \mid b$
  - **(2)** $A \to Ac \mid Sd \mid e$
- **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** $A \to Sd$ **by** $A \to Aad \mid bd$
  - **hence (2) becomes** $A \to Ac \mid Aad \mid bd \mid e$
  - **after removing immediate left-recursion:**
    - ▷ $A \to bdA' \mid eA'$
    - ▷ $A' \to cA' \mid adA' \mid \epsilon$
- **Resulting grammar:**
  - ▷ $S \to Aa \mid b$
  - ▷ $A \to bdA' \mid eA'$
  - ▷ $A' \to cA' \mid adA' \mid \epsilon$

Left-factoring and left-recursion removal

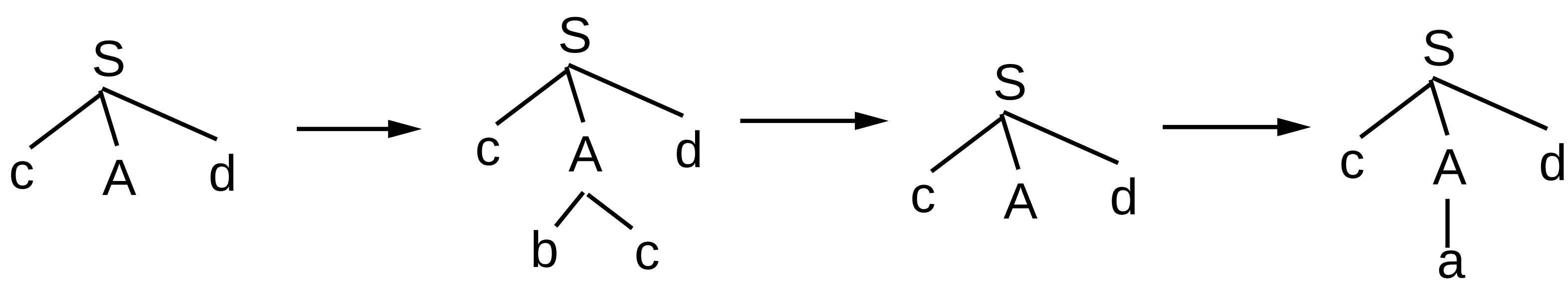
- **Original grammar:**
  - $S \to (S) \mid SS \mid ()$
- **To remove immediate left-recursion, we have**
  - $S \to (S)S' \mid ()S'$
  - $S' \to SS' \mid \epsilon$
- **To do left-factoring, we have**
  - $S \to (S''$
  - $S'' \to S)S' \mid)S'$
  - $S' \to SS' \mid \epsilon$

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.
  - Top-down parsing naturally corresponds to leftmost derivation.
  - 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.

Recursive-descent parsing: example

- **Grammar:** $\boxed{\begin{array}{l} S \rightarrow cAd \\ A \rightarrow bc \mid a \end{array}}$ **Input:** $\boxed{cad}$



error!! backtrack

- **Problems with the above approach:**
  - **Still too slow!**
  - **Need to be able to select a derivation without ever causing backtracking!**
    - ▷ ***Predictive parser*** *: a recursive-descent parser needing no backtracking.*

Predictive parser

- **Goal: Find a rich class of grammars that can be parsed using predictive parsers.**
- **The class of $LL(1)$ grammars [Lewis & Stearns 1968] can be parsed by a predictive parser in $O(n)$ time.**
  - **First "L": scan the input from left-to-right.**
  - **Second "L": find a leftmost derivation.**
  - **Last "(1)": allow one lookahead token!**
- **Based on the current lookahead symbol, pick a derivation when there are multiple choices.**
  - **Using a STACK during implementation to avoid recursion.**
  - **Build a PARSING TABLE T, using the symbol X on the top of STACK and the lookahead symbol s as indexes, to decide the production to be used.**
    - ▷ ***If X is a terminal, then $X = s$ and input s is matched.***
    - ▷ ***If X is a nonterminal, then $T(X, s)$ tells you the production to be used in the next derivation.***

Predictive parser: Algorithm

- **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 REJECT!**

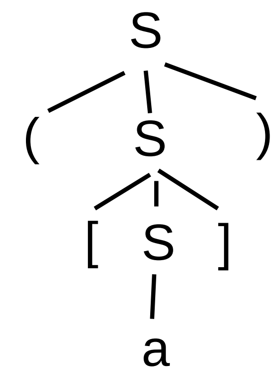
When does the parser reject an input?

- **STACK is empty and there is some input left;**
  - **A proper prefix of the input is accepted.**
- **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;**

Parsing an $LL(1)$ grammar: example

- **Grammar:** $\boxed{S \rightarrow a \mid (S) \mid [S]}$ **Input:** $\boxed{([a])}$

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 $LL(1)$ — (1/2)

- **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 proper subset of CFG.**
- **For example, the following grammar is not $LL(1)$, but is $LL(2)$.**
  - **Grammar:** $\boxed{S \rightarrow (S) \mid [S] \mid () \mid [\,]}$ **Input:** $\boxed{()}$

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.***

About $LL(1)$ — (2/2)

- **A grammar is not $LL(1)$ if it**
  - **is ambiguous,**
    - ▷ ***Q: Why?***
  - **is left-recursive, or**
    - ▷ ***Q: Why?***
  - **has left-factors.**
    - ▷ ***Q: Why?***
- **However, grammars that are not ambiguous, are not left-recursive and have no left-factor may still not be $LL(1)$.**
  - **Q: Any examples?**
- **Two questions:**
  - **How to tell whether a grammar G is $LL(1)$?**
  - **How to build the PARSING TABLE if it is $LL(1)$?**

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*;***
    - ▷ $\epsilon \in$ ***FIRST*(α) *if and only if* $\alpha \stackrel{+}{\Longrightarrow} \epsilon$*.***
- **FIRST$(\alpha) =$**
  $\{t \mid$ **(**t **is a terminal and** $\alpha \stackrel{*}{\Longrightarrow} t\beta$**) or (** $t = \epsilon$ **and** $\alpha \stackrel{*}{\Longrightarrow} \epsilon$**)**$\}$
- **Why do we need FIRST SETS?**
  - **When there are many choices $A \Longrightarrow \alpha_1 | \cdots | \alpha_k$,**
  - **and the lookahead symbol is s,**
  - **we use $A \Longrightarrow \alpha_i$ if $s \in$ FIRST(α_i).**

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:
  - FIRST(X) = FIRST$(Y_1) - \{\epsilon\}$;
  - if $\epsilon \in$ FIRST(Y_1), then
    - ▷ *put FIRST$(Y_2) - \{\epsilon\}$ into FIRST(X);*
  - if $\epsilon \in$ FIRST$(Y_1) \cap$ FIRST(Y_2), then
    - ▷ *put FIRST$(Y_3) - \{\epsilon\}$ into FIRST(X);*
  - $\cdots$
  - if $\epsilon \in \cap_{i=1}^{k-1}$FIRST$(Y_i)$, then
    - ▷ *put FIRST$(Y_k) - \{\epsilon\}$ into FIRST(X);*
  - if $\epsilon \in \cap_{i=1}^{k}$FIRST$(Y_i)$, then
    - ▷ *put ϵ into FIRST(X).*

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

- **Algorithm to compute FIRST's for all nonterminals.**
  - compute FIRST's for ϵ and all terminals;
  - initialize FIRST's for all nonterminals 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?**
  - Types of recursive calls: direct or 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;
    - ▷ *maximum 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.*
  - Each iteration takes $O(|N| + |T|)$ time.
  - $O(|N| \cdot |T| \cdot (|N| + |T|))$.

Example for computing FIRST(X)

- **A heuristic ordering to compute FIRST for all nonterminal.**
  - **First process nonterminal X such that $X \Longrightarrow \alpha_1 | \cdots | \alpha_k$, and $\alpha_i = \epsilon$ or a prefix of α_i is a terminal.**
  - **Then find nonterminals that only depends on nonterminals whose FIRST are computed.**

Grammar

$E \rightarrow E'T$

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

$T \rightarrow FT'$

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

$F \rightarrow int \mid (E)$

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

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

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

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

FIRST$(E) = \{-, int, (\}$,
since $\epsilon \in$ FIRST(E').
Note $\epsilon \notin$ FIRST$(E') \cap$ FIRST(T).

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 first.**
- **Let $\alpha = X_1 X_2 \cdots X_n$. Perform the following steps in sequence:**
  - **FIRST(α) = FIRST$(X_1) - \{\epsilon\}$;**
  - **if $\epsilon \in$ FIRST(X_1), then**
    - ▷ ***put FIRST$(X_2) - \{\epsilon\}$ into FIRST(α);***
  - **if $\epsilon \in$ FIRST$(X_1) \cap$ FIRST(X_2), then**
    - ▷ ***put FIRST$(X_3) - \{\epsilon\}$ into FIRST(α);***
  - $\cdots$
  - **if $\epsilon \in \cap_{i=1}^{n-1}$FIRST$(X_i)$, then**
    - ▷ ***put FIRST$(X_n) - \{\epsilon\}$ into FIRST(α);***
  - **if $\epsilon \in \cap_{i=1}^{n}$FIRST$(X_i)$, then**
    - ▷ ***put $\{\epsilon\}$ into FIRST(α).***
- **What to do when recursive calls are encountered?**
- **What are the time and space complexities?**

Example for computing FIRST(α)

Grammar

$E \rightarrow E'T$

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

$T \rightarrow FT'$

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

$F \rightarrow int \mid (E)$

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

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

$\textbf{\textit{FIRST}}(T) = \{int, (\}$

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

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

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

$\textbf{\textit{FIRST}}(-TE') = \{-\}$

$\textbf{\textit{FIRST}}(\epsilon) = \{\epsilon\}$

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

$\textbf{\textit{FIRST}}(/FT') = \{/\}$

$\textbf{\textit{FIRST}}(\epsilon) = \{\epsilon\}$

$\textbf{\textit{FIRST}}(int) = \{int\}$

$\textbf{\textit{FIRST}}((E)) = \{(\}$

- **FIRST**$(T'E') =$
  - $(\textbf{\textit{FIRST}}(T') - \{\epsilon\}) \cup$
  - $(\textbf{\textit{FIRST}}(E') - \{\epsilon\}) \cup$
  - $\{\epsilon\}$

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

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

How to compute FOLLOW(X)?

- **Initialization:**
  - If X is the starting nonterminal, initial value of FOLLOW(X) is $\{\$\}$.
  - If X is not the starting nonterminal, initial value of FOLLOW(X) is $\emptyset$.
- **Repeat**

  for all nonterminals X do
  - Find the productions with X on the right-hand-side.
  - for each production of the form $Y \to \alpha X \beta$, 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 $Y \to \alpha X$, put FOLLOW(Y) into FOLLOW(X).

  until nothing can be added to any FOLLOW set.
- **Questions:**
  - What to do when recursive calls are encountered?
  - What are the time and space complexities?

Examples for FIRST's and FOLLOW's

- **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:**
    - $X \rightarrow \alpha_1$
    - $\cdots$
    - $X \rightarrow \alpha_k$
  - **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 $\alpha_i \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in$ FOLLOW(X), then we can use the derivation $X \rightarrow \alpha_i$.**
  - **$\alpha_i \stackrel{*}{\Longrightarrow} \epsilon$ if and only if $\epsilon \in$ FIRST(α_i).**

Whether a grammar is $LL(1)$? (1/2)

- To see whether 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;
    - ▷ *Need to first compute FIRST(X) for every nonterminal X.*
  - 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.*
- Note that FIRST and FOLLOW sets are always sets of terminals, plus, perhaps, ϵ for some FIRST sets.
- A grammar is not $LL(1)$ if there exists productions

$$X \rightarrow \alpha \mid \beta$$

  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$(X) \neq \emptyset$.

Whether a grammar is $LL(1)$? (2/2)

- **If a grammar is not $LL(1)$, then**
  - **you cannot write a linear-time predictive parser as described previously.**
- **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:**
    - ▷ $\textbf{Parameter} \rightarrow \epsilon \mid id \mid l\_paren\ \textbf{Parameter}\ r\_paren$
  - $semicolon \notin$ **FIRST**(ϵ) **and** $semicolon \notin$ **FIRST**(id) **and** $semicolon \notin$ **FIRST**$(l\_paren$ **Parameter** $r\_paren)$
  - **Parameter** $\stackrel{*}{\Longrightarrow} \epsilon$ **and** $semicolon \in$ **FOLLOW**(**Parameter**)
  - **Hence we use the derivation**
    Parameter $\rightarrow \epsilon$

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

Grammar:

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

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

- **Check for possible conflicts in $X \to a \mid \epsilon$.**
  - **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 $C \to a \mid \epsilon$.**
  - **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	$C \rightarrow a$	$C \rightarrow \epsilon$

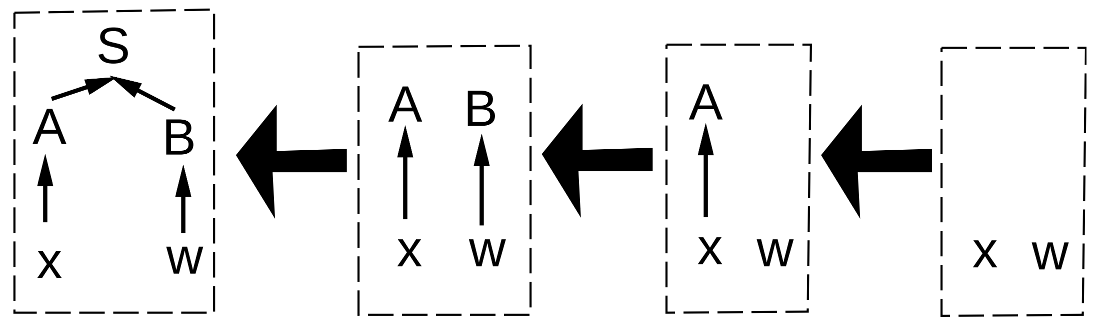
Bottom-up parsing (Shift-reduce parsers)

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

Input: $\boxed{xw}$

- **Grammar:**

$$
\begin{aligned}
&S \to AB \\
&A \to x \mid Y \\
&B \to w \mid Z \\
&Y \to xb \\
&Z \to wp
\end{aligned}
$$



- **This grammar is not $LL(1)$.**
  - **Why?**
  - **It can be rewritten into an $LL(1)$ grammar, though.**

Right-sentential form

- **Rightmost derivation:**
  - $S \underset{rm}{\Longrightarrow} \alpha$: **the rightmost nonterminal is replaced.**
  - $S \overset{+}{\underset{rm}{\Longrightarrow}} \alpha$: α **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

- **Handle** : a handle for a right-sentential form $\gamma = \alpha\beta\eta$
  - 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*
  - such that $\gamma' = \alpha A \eta$ is also a right-sentential form and
  - η contains only terminals or is ϵ.
- **Properties of a handle.**
  - γ' is obtained by replacing β at the position w with A in γ.
  - $\gamma = \alpha\beta\eta$ and is a right-sentential form.
  - $\gamma' = \alpha A \eta$ and is also a right-sentential form.
  - $\gamma' \underset{rm}{\Longrightarrow} \gamma$ and since η contains no nonterminals.

Handle: example

- **Grammar:** $\boxed{\begin{array}{l} S \to aABe \\ A \to Abc \mid b \\ B \to d \end{array}}$ **Input:** $\boxed{abbcde}$
- $S \underset{rm}{\Longrightarrow} aABe \underset{rm}{\Longrightarrow} aAde \underset{rm}{\Longrightarrow} aAbcde \underset{rm}{\Longrightarrow} abbcde$
- $\gamma \equiv aAbcde$ **is a right-sentential form**
- $A \to Abc$ **and position 2 in** γ **is a handle for** γ
- $\gamma' \equiv aAde$ **is also a right-sentential form**

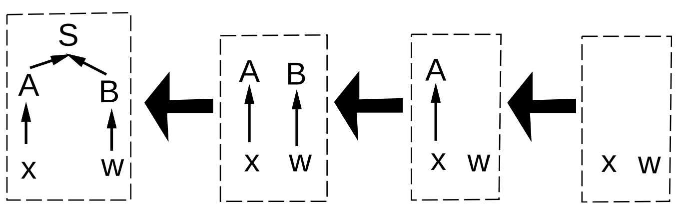
Handle reducing

- **Reduce**: replace a handle in a right-sentential form with its left-hand-side at the location specified in the handle.
- 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 to find handles?
  - What to do when there are two possible handles?
    - *Have a common prefix or suffix.*
    - *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**
- **Make sure handles are always on the top of STACK.**

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 STACK.**
  - **Some suffix of a viable prefix is a prefix of a handle.**
    - ▷ ***push the current input token to STACK***
    - ▷ ***shift***
  - **Some suffix of a viable prefix is a handle.**
    - ▷ ***perform a handle reduction***
    - ▷ ***reduce***

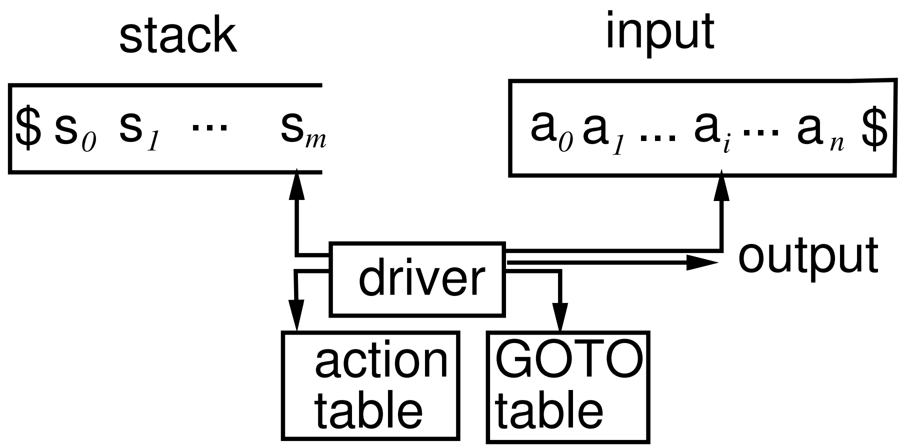
Properties of viable prefixes

- **Some prefix of a right-sentential form cannot appear on the top of STACK during parsing.**
  - **Grammar:**
    - $S \to AB$
    - $A \to x \mid Y$
    - $B \to w \mid Z$
    - $Y \to xb$
    - $Z \to wp$
  - **Input:** $\boxed{xw}$
    - ***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.***
- **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 STACK.**
  - **That is, some substring of the input is the first handle.**

Using viable prefixes

- **Strategy:**
  - **Try to recognize all possible viable prefixes.**
    - *Can recognize them incrementally.*
  - **Shift is allowed if after shifting, the top of STACK is still a viable prefix.**
  - **Reduce is allowed if after a handle is found on the top of STACK and after reducing, the top of STACK is still a viable prefix.**
- **Questions:**
  - *How to recognize a viable prefix efficiently?*
  - *What to do when multiple actions are allowed?*

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.
  - Last (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.

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

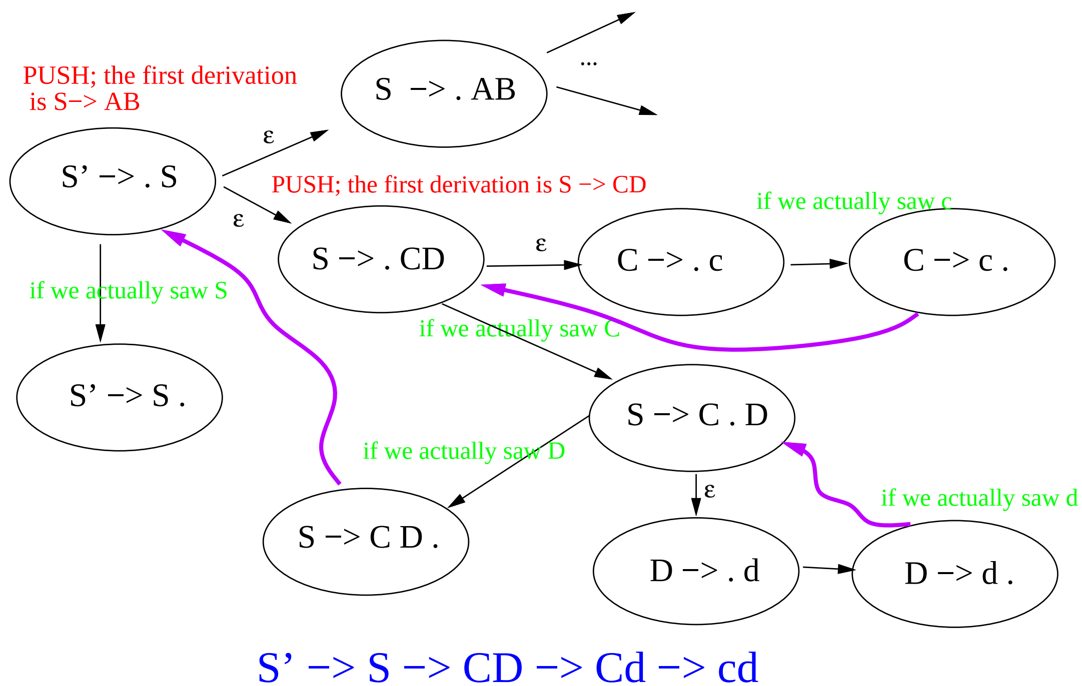
- Top-down parsing for $LL(k)$ grammars: be able to choose a production by seeing only the first k symbols that will be derived from that production.

Recognizing viable prefixes

- Use an **$LR(0)$ item** (**item** for short) to record all possible extensions of the current viable prefix.
  - It 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 original 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 current viable prefix.*
  - ▷ *Use NFA to record information about the next possible handle.*
  - ▷ *push down automata = FA + stack.*
  - ▷ *Need to use DFA for simplicity.*



Closure

- **The closure operation $closure(I)$, where I is a set of some $LR(0)$ 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 some** $LR(0)$ **items and** X **is a legal symbol, means**
  - **If** $A \rightarrow \alpha \cdot X\beta$ **is in** I**, then**
  - $closure(\{A \rightarrow \alpha X \cdot \beta\}) \subseteq GOTO(I, X)$
- **Informal meanings:**
  - **currently we have seen** $A \rightarrow \alpha \cdot X\beta$
  - **expect to see** X
  - **if we see** X**,**
  - **then we should be in the state** $closure(\{A \rightarrow \alpha X \cdot \beta\})$**.**
- **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 some $LR(0)$ 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.

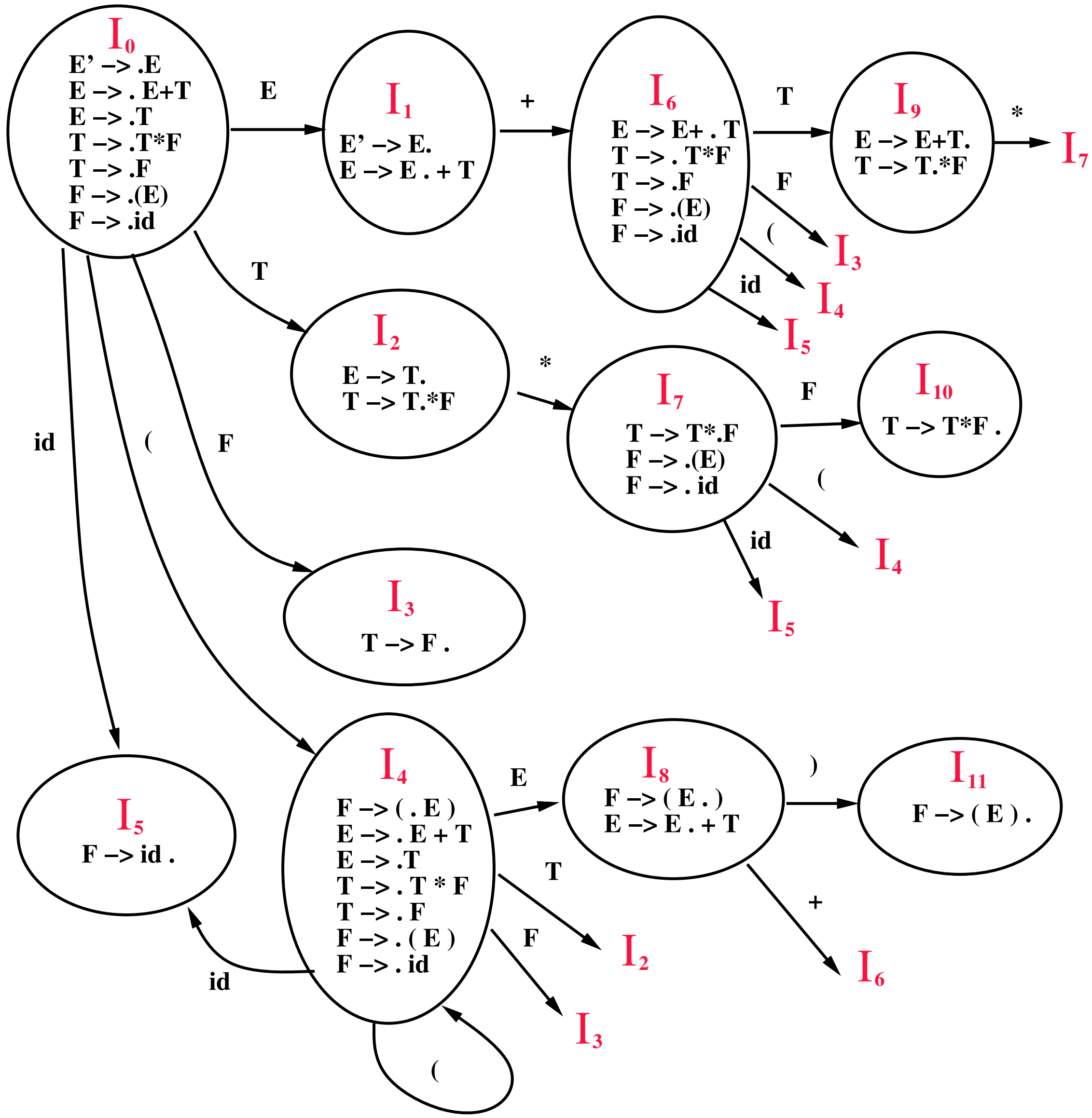
Example of sets of $LR(0)$ items

- **Grammar:**
$$E' \to E$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

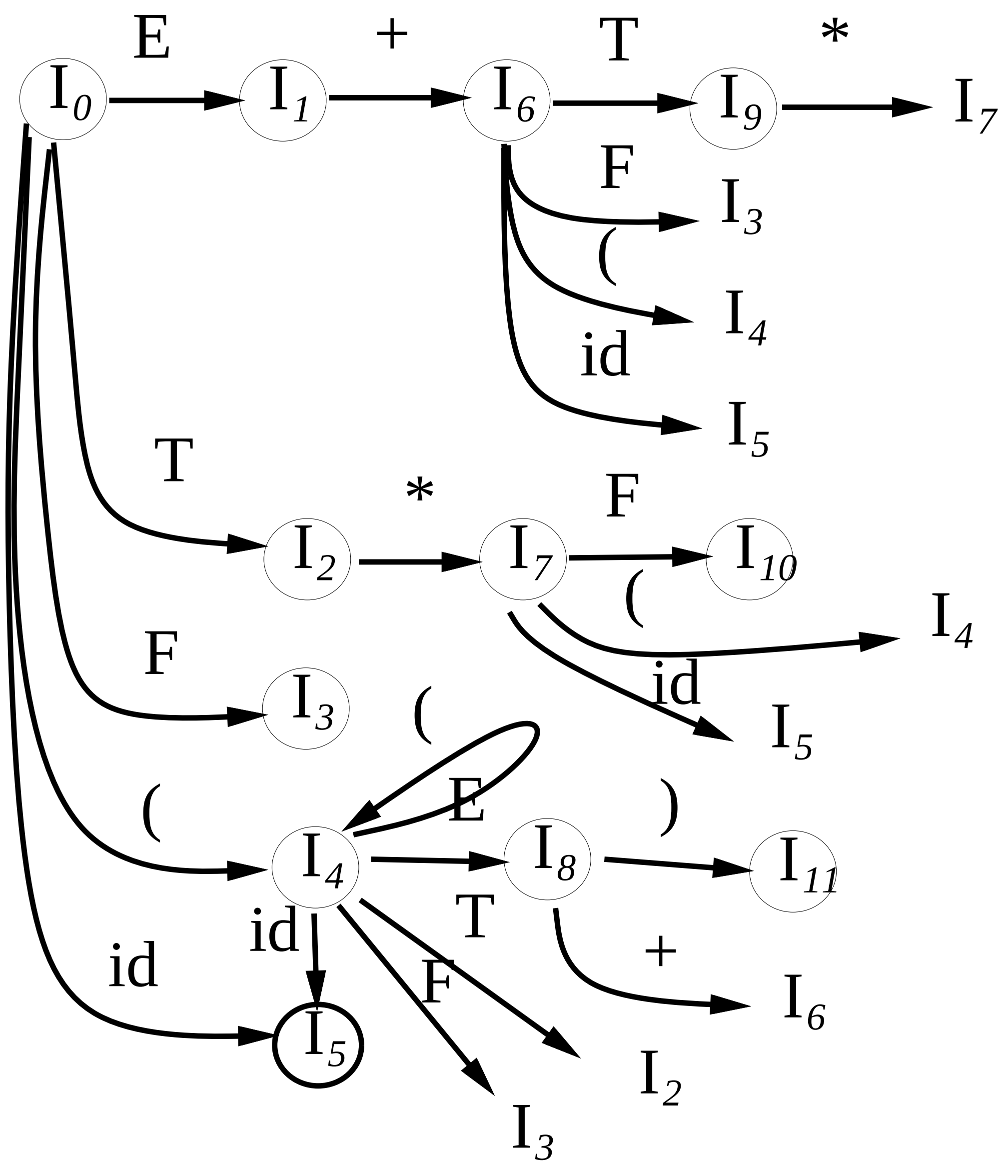
$$I_0 = closure(\{E' \to \cdot E\}) =$$
$$\{E' \to \cdot E,$$
$$E \to \cdot E + T,$$
$$E \to \cdot T,$$
$$T \to \cdot T * F,$$
$$T \to \cdot F,$$
$$F \to \cdot (E),$$
$$F \to \cdot id\}$$

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

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 **=**
  - $\{T \to T * \cdot F,$
  - $F \to \cdot(E),$
  - $F \to \cdot id\}$
- **We expect to follow one of the following three possible derivations:**

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

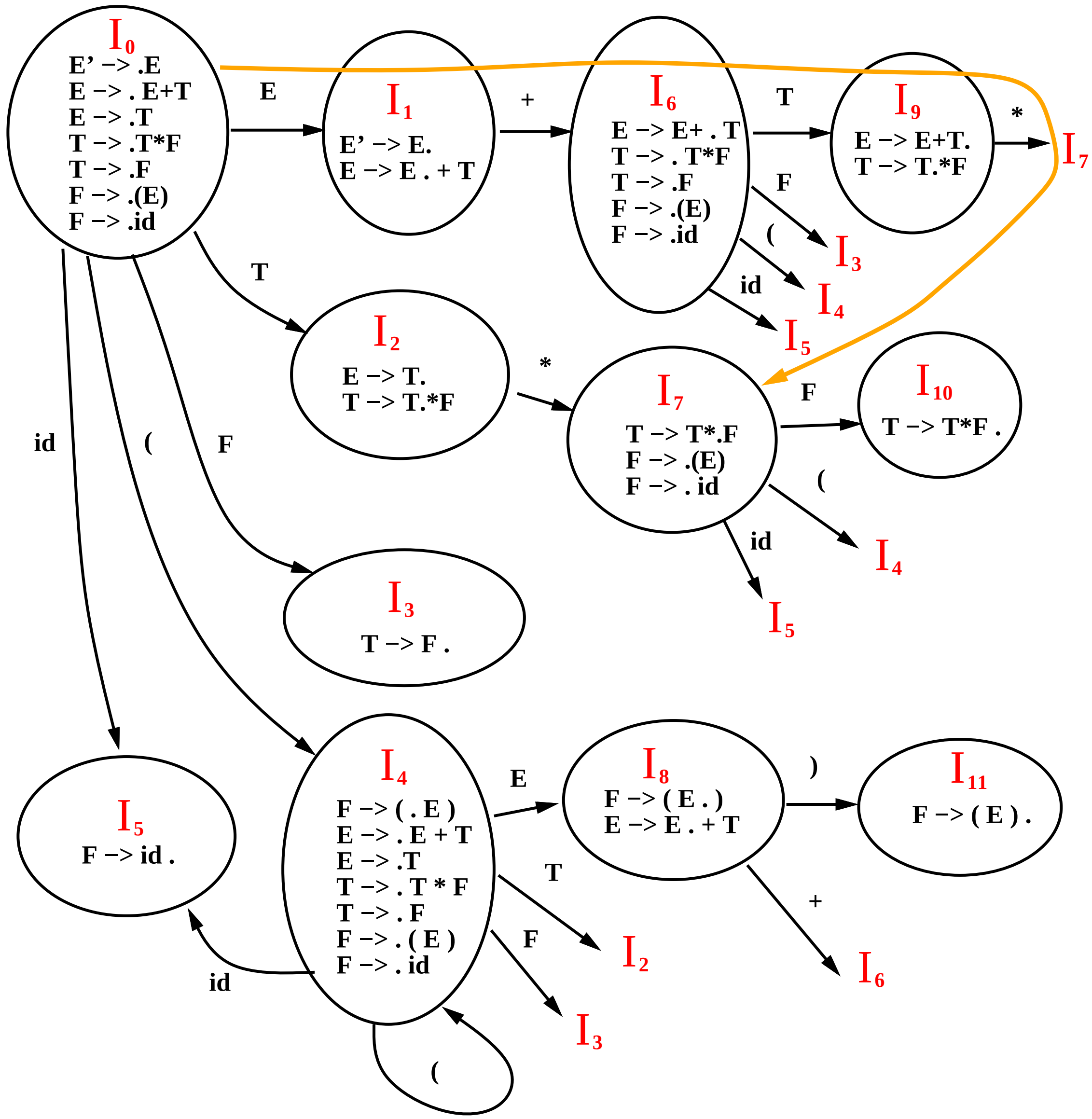
$$
\begin{aligned}
E' &\underset{rm}{\Longrightarrow} E \\
&\underset{rm}{\Longrightarrow} E + T \\
&\underset{rm}{\Longrightarrow} E + T * F \\
&\underset{rm}{\Longrightarrow} \underline{E + T\ *}\ (E) \\
&\cdots
\end{aligned}
$$

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

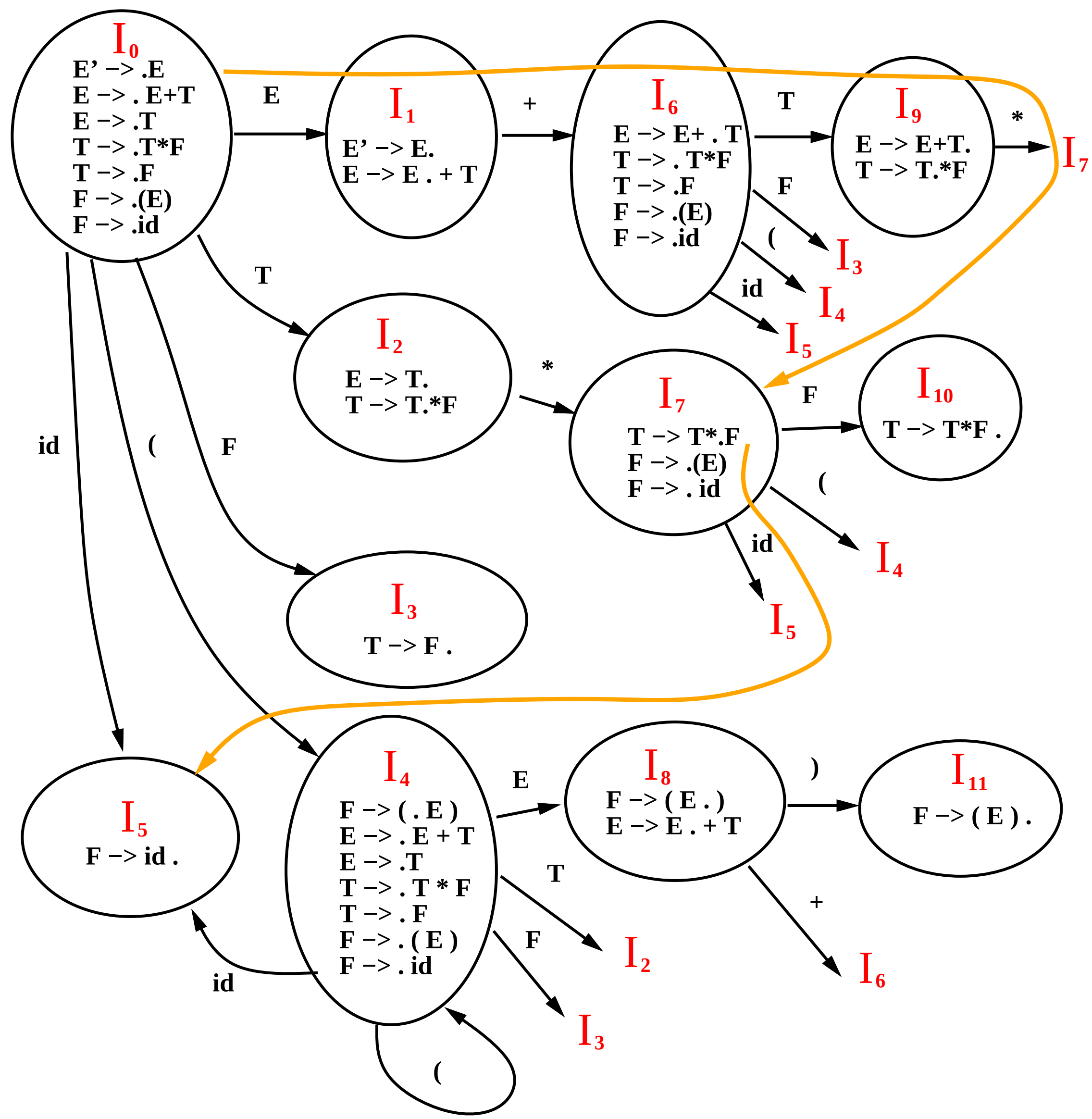
High-level ideas of parsing

- **Viable prefix: saved in the STACK to record the path it comes from.**
  - **All possible viable prefixes are compactly recorded in the transition diagram.**
- **Top of STACK: the current state it is in.**
- **Shift: we can extend the current viable prefix.**
  - **PUSH and change state.**
- **Reduce: we can perform a handle reduction.**
  - **POP and backtrack to the state we were last in.**

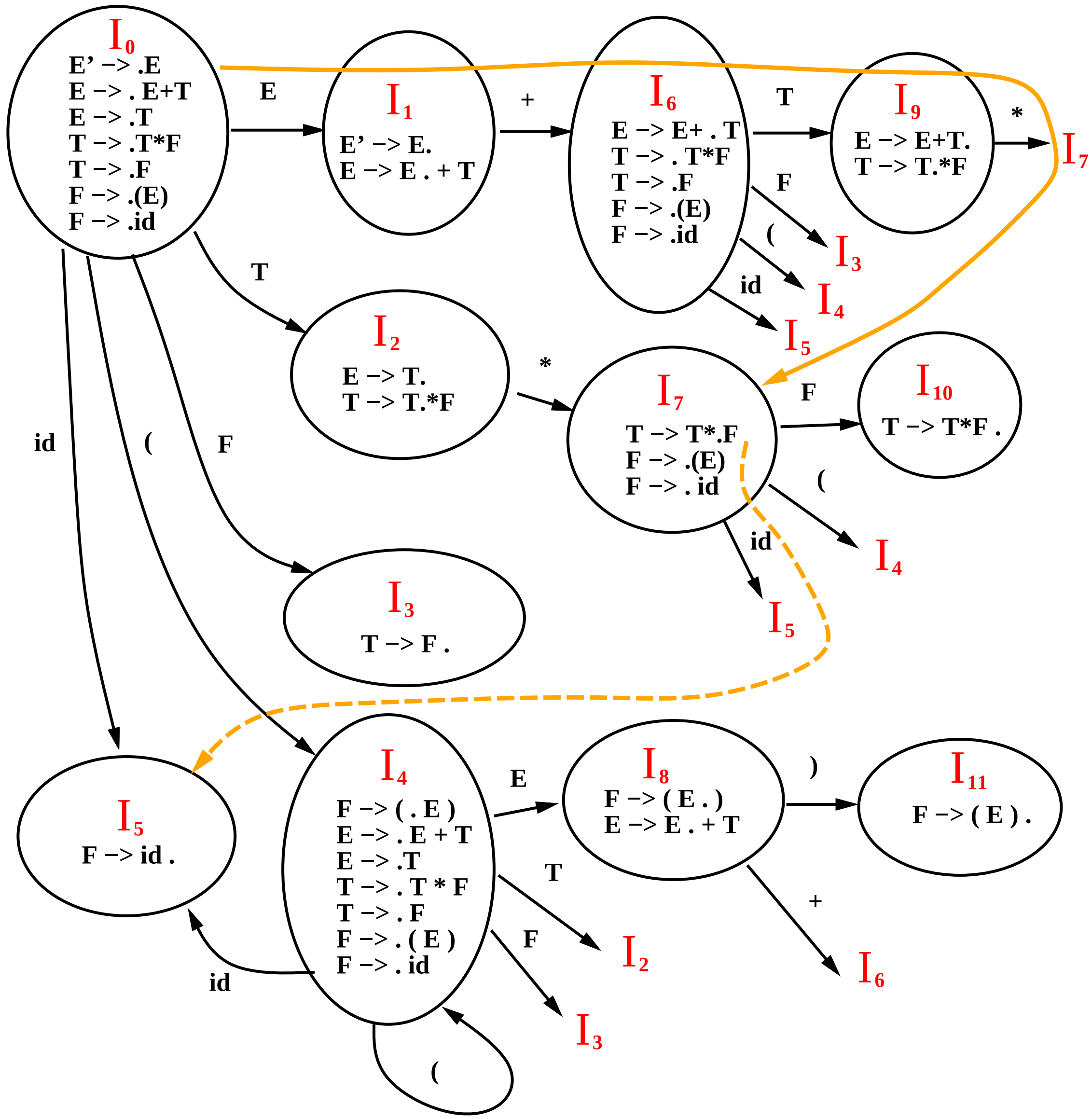
Parsing example



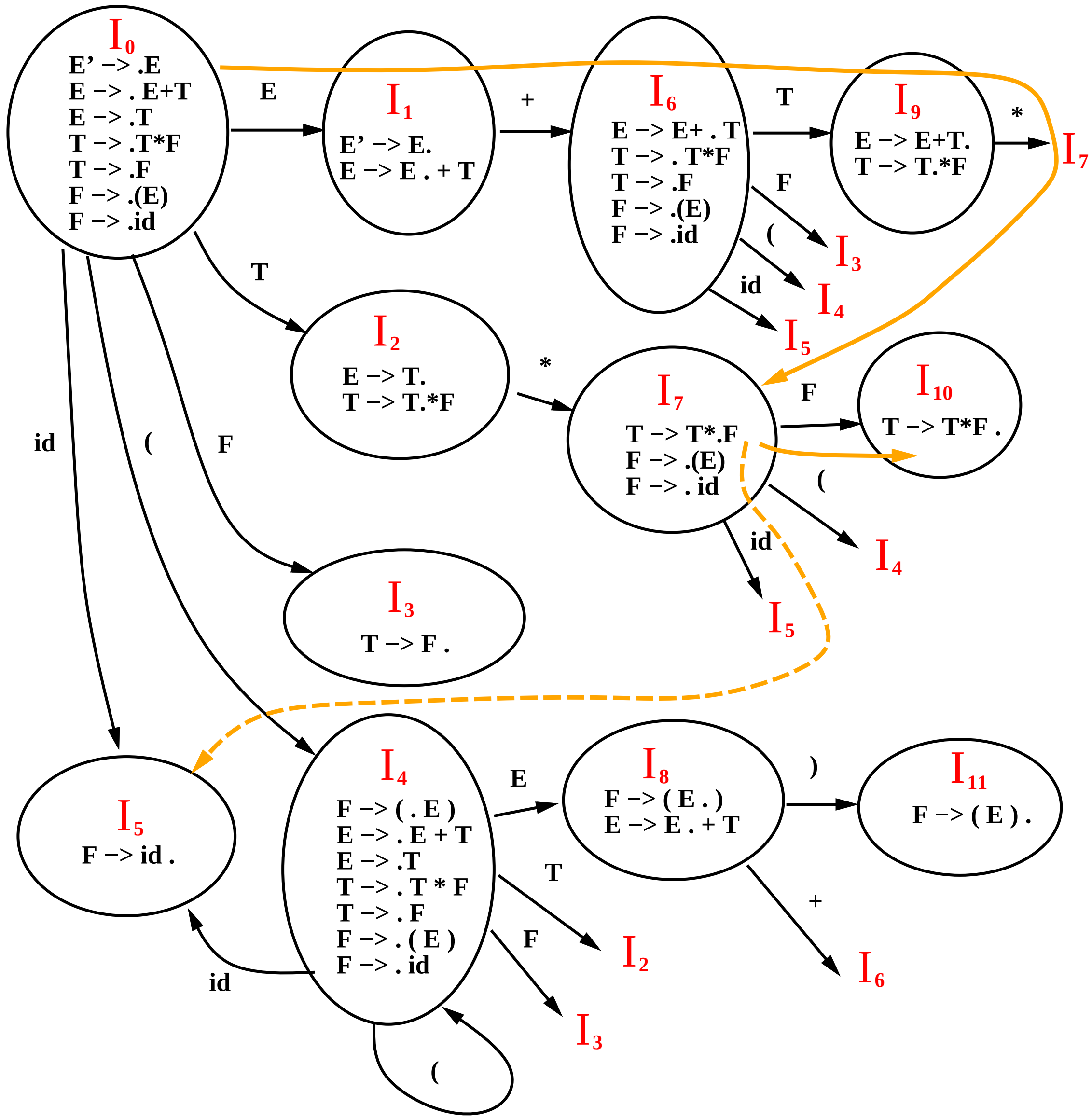
$$E + T * F \underset{rm}{\Longrightarrow} E + T \ * \ id$$



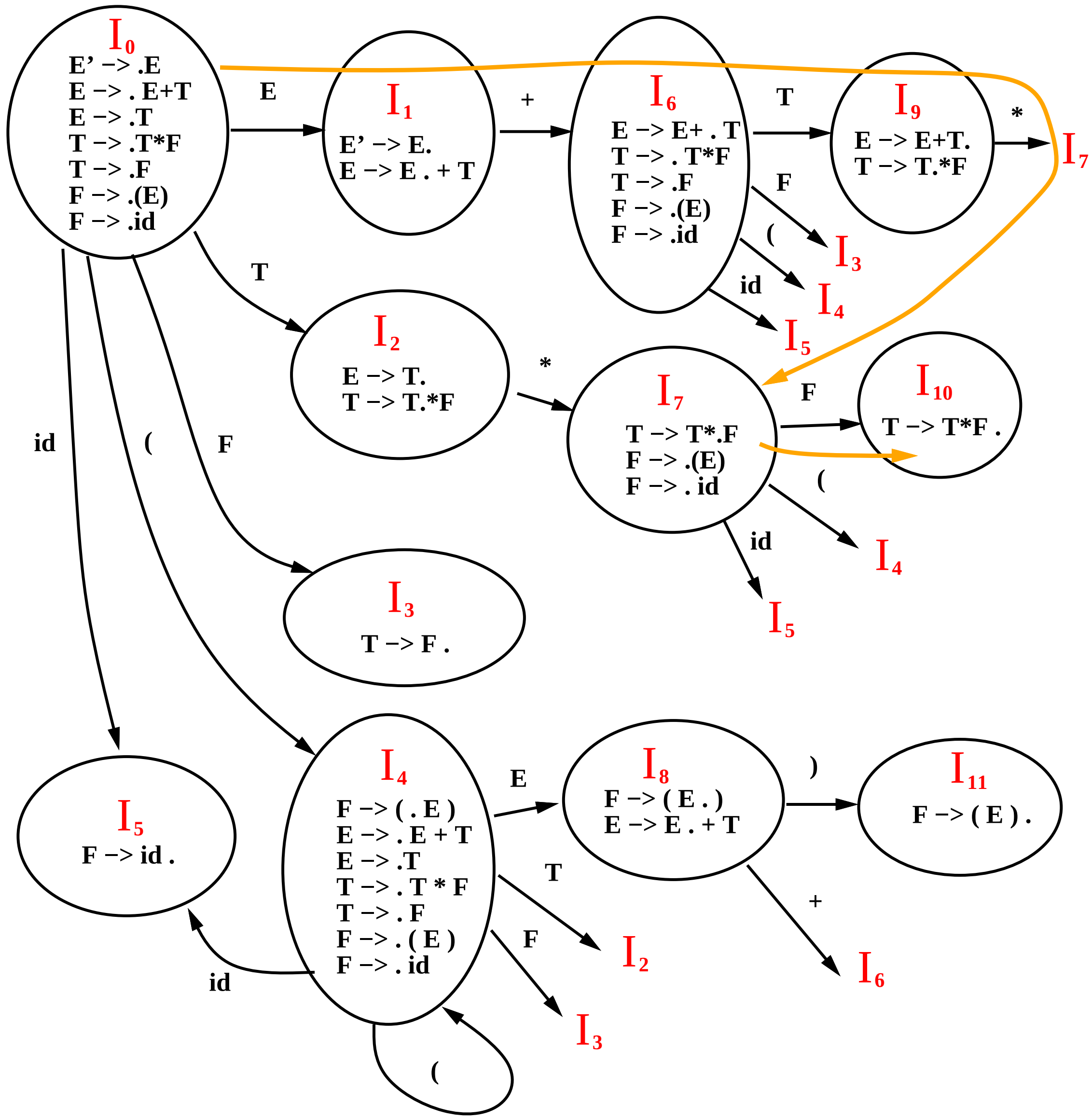
$$E + T * F \underset{rm}{\Longrightarrow} E + T \ * \ id$$



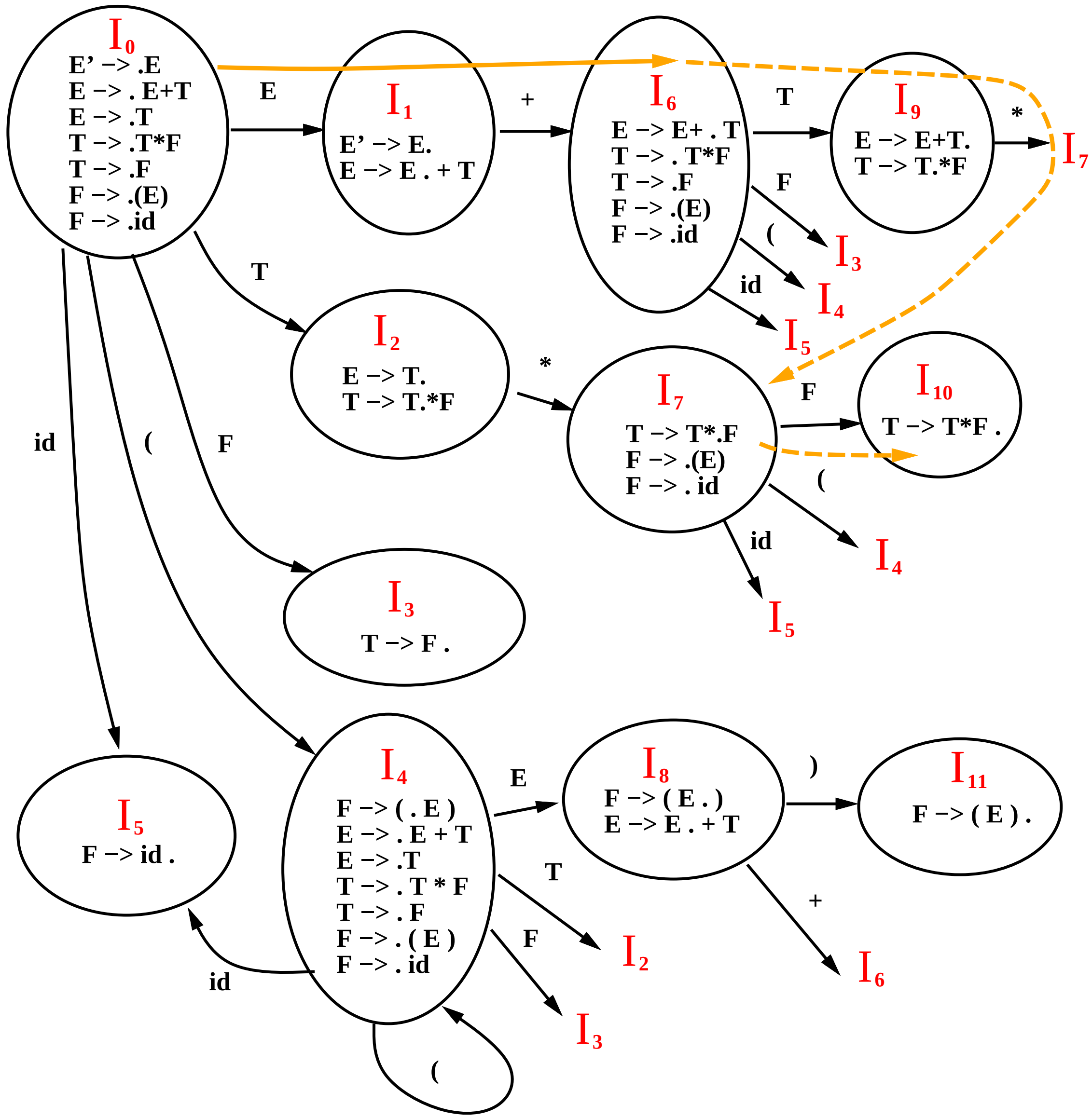
$$E + T * F \underset{rm}{\Longrightarrow} E + T \;\; * \;\; id$$



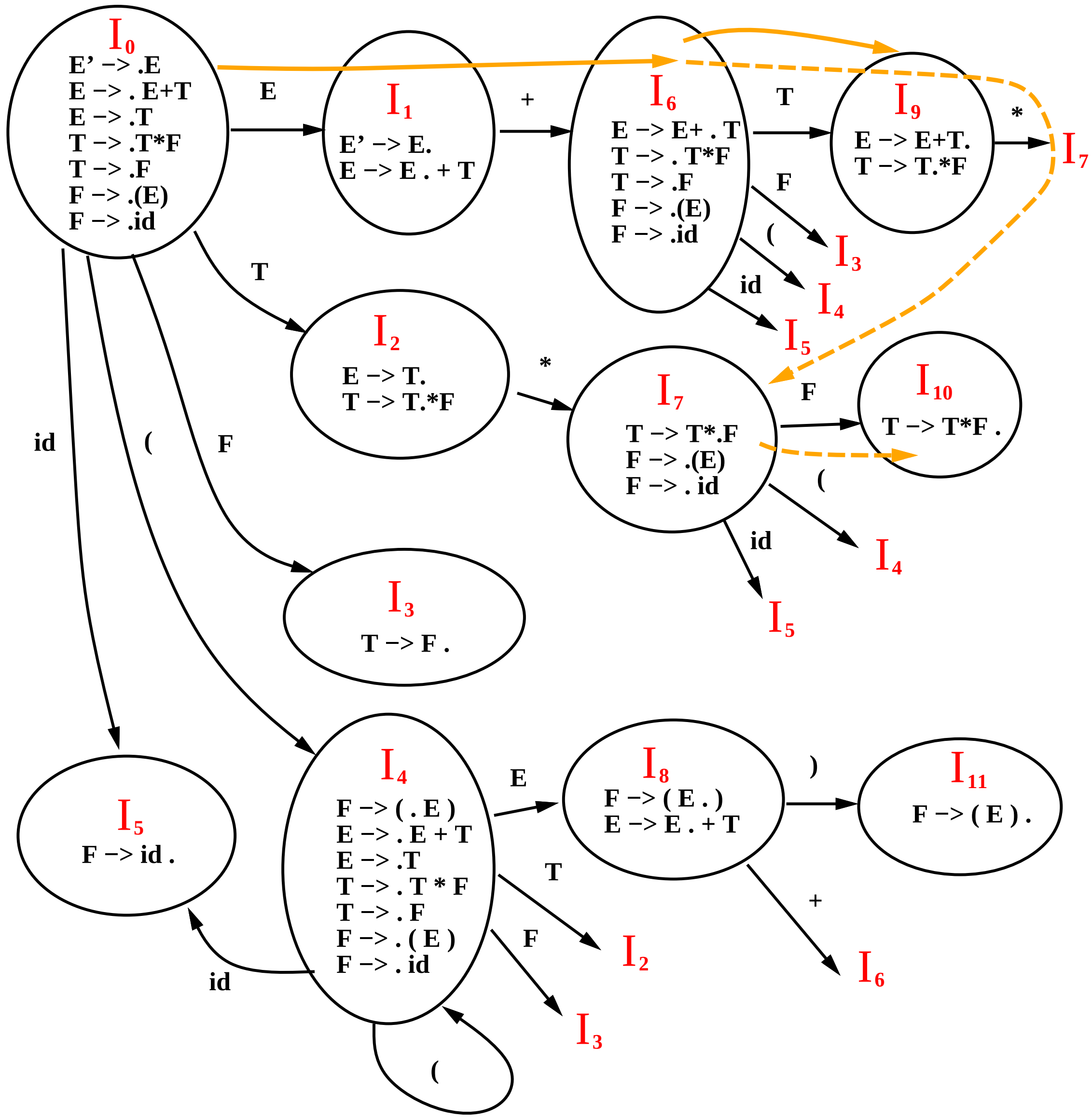
$$E + T * F \underset{rm}{\Longrightarrow} E + T \;\; * \;\; id$$



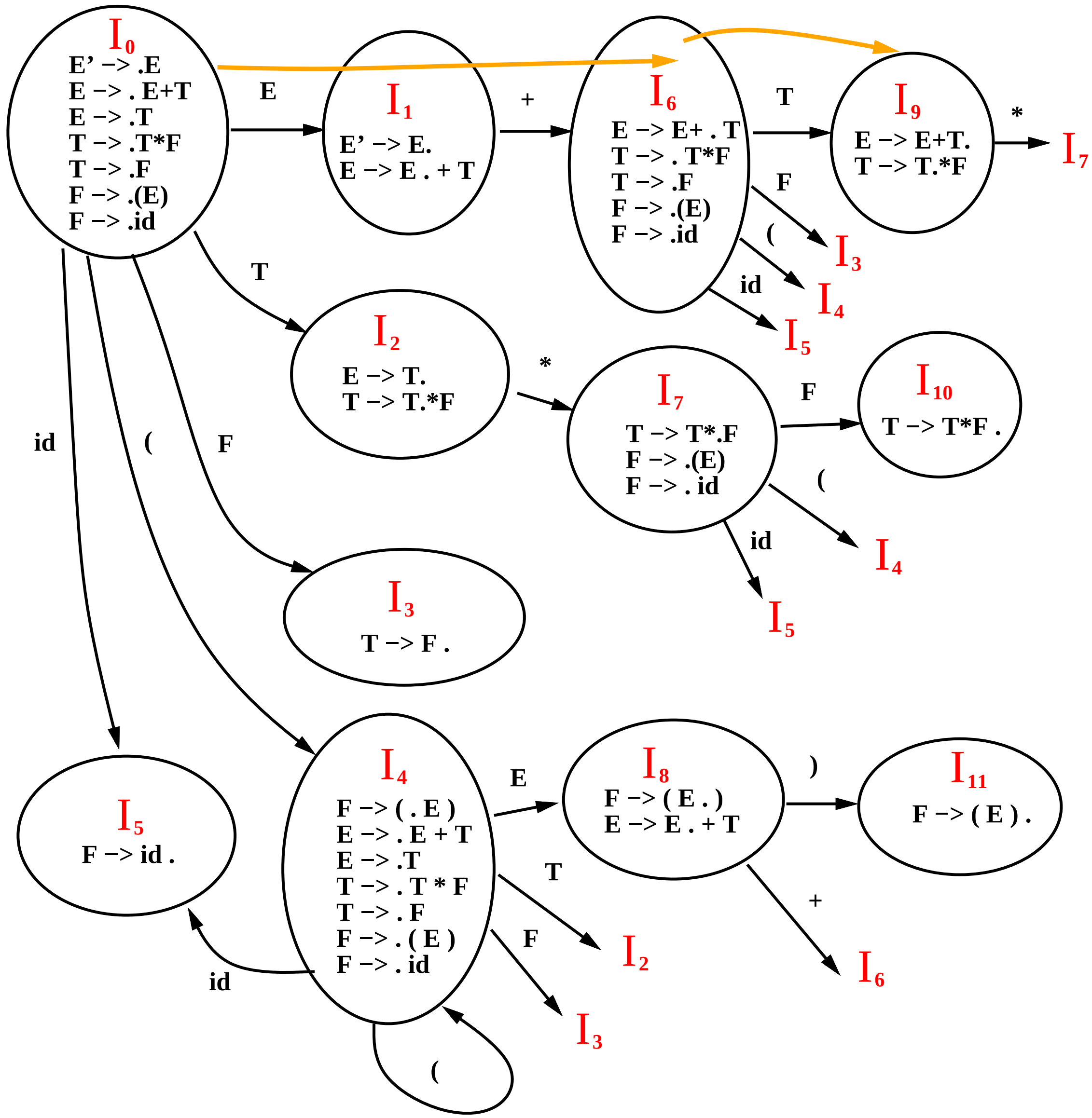
$$E + T \underset{rm}{\Longrightarrow} E + T * F$$



$$E + T \underset{rm}{\Longrightarrow} E + T * F$$



$$E + T \underset{rm}{\Longrightarrow} E + T * F$$



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 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 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 $closure(\{A \rightarrow \alpha X \cdot \beta\}) \subseteq GOTO(I, X)$.

$LR(0)$ parsing

- LR parsing without lookahead symbols.
- Initially,
  - Push I_0 into the stack.
  - Begin to scan the input from left to right.
- In state I_i
  - if $\{A \rightarrow \alpha \cdot a\beta\} \subseteq I_i$ then perform "shift i" while seeing the terminal a in the input, and then go to the state $I_j = closure(\{A \rightarrow \alpha a \cdot \beta\})$.
    - ▷ *Push a into the STACK first.*
    - ▷ *Then push I_j into the STACK.*
  - if $\{A \rightarrow \beta \cdot\} \subseteq I_i$, then perform "reduce by $A \rightarrow \beta$" and then go to the state $I_j = GOTO(I, A)$ where I is the state on the top of STACK after removing β
    - ▷ *Pop β and all intermediate states from the STACK.*
    - ▷ *Push A into the STACK.*
    - ▷ *Then push I_j into the STACK.*
  - Reject if none of the above can be done.
  - Report "conflicts" if more than one can be done.
- Accept an input if EOF is seen at I_0.

Parsing example (1/2)

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 $F \rightarrow id$
\$ I_0 E I_1 + I_6 F I_3	\$	in I_6, saw F, goto I_3
...	...	...

Parsing example (2/2)

STACK	input	action
\$ I_0	id + id * id\$	shift 5
\$ I_0 id I_5	+ id * id\$	reduce by $F \to id$
\$ I_0 F	+ id * id\$	in I_0, saw F, goto I_3
\$ I_0 F I_3	+ id * id\$	reduce by $T \to F$
\$ I_0 T	+ id * id\$	in I_0, saw T, goto I_2
\$ I_0 T I_2	+ id * id\$	reduce by $E \to T$
\$ I_0 E	+ id * id\$	in I_0, saw E, goto I_1
\$ I_0 E I_1	+ id * id\$	shift 6
\$ I_0 E I_1 + I_6	id * id\$	shift 5
\$ I_0 E I_1 + I_6 id I_5	* id\$	reduce by $F \to id$
\$ I_0 E I_1 + I_6 F	* id\$	in I_6, saw F, goto I_3
\$ I_0 E I_1 + I_6 F I_3	* id\$	reduce by $T \to F$
\$ I_0 E I_1 + I_6 T I_9	* id\$	in I_6, saw T, goto I_9
\$ I_0 E I_1 + I_6 T I_9 * I_7	id\$	shift 7
\$ I_0 E I_1 + I_6 T I_9 * I_7 id I_5	\$	shift 5
\$ I_0 E I_1 + I_6 T I_9 * I_7 F	\$	reduce by $F \to id$
\$ I_0 E I_1 + I_6 T I_9 * I_7 F I_{10}	\$	in I_7, saw F, goto I_{10}
\$ I_0 E I_1 + I_6 T	\$	reduce by $T \to T * F$
\$ I_0 E I_1 + I_6 T I_9	\$	in I_6, saw T, goto I_9
...	...	...

Problems of $LR(0)$ parsing

- **Conflicts: handles have overlaps, thus multiple actions are allowed at the same time.**
  - **shift/reduce conflict**
  - **reduce/reduce conflict**
- **Very few grammars are $LR(0)$. For example:**
  - **In I_2 of our example, 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."***
- **Idea: use FOLLOW(E) as look ahead information to resolve some conflicts.**

$SLR(1)$ parsing algorithm

- **Using FOLLOW sets to resolve conflicts in constructing $SLR(1)$ [DeRemer 1971] 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 $A \to \alpha \cdot a\beta$ 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 $A \to \alpha \cdot$ is in I_i, then**
    - ▷ ***$action(I_i, a)$ is "reduce by $A \to \alpha$" for all terminal $a \in$ FOLLOW(A); here $A \neq S'$.***
  - **If $S' \to S \cdot$ 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: $L \rightarrow id \cdot$

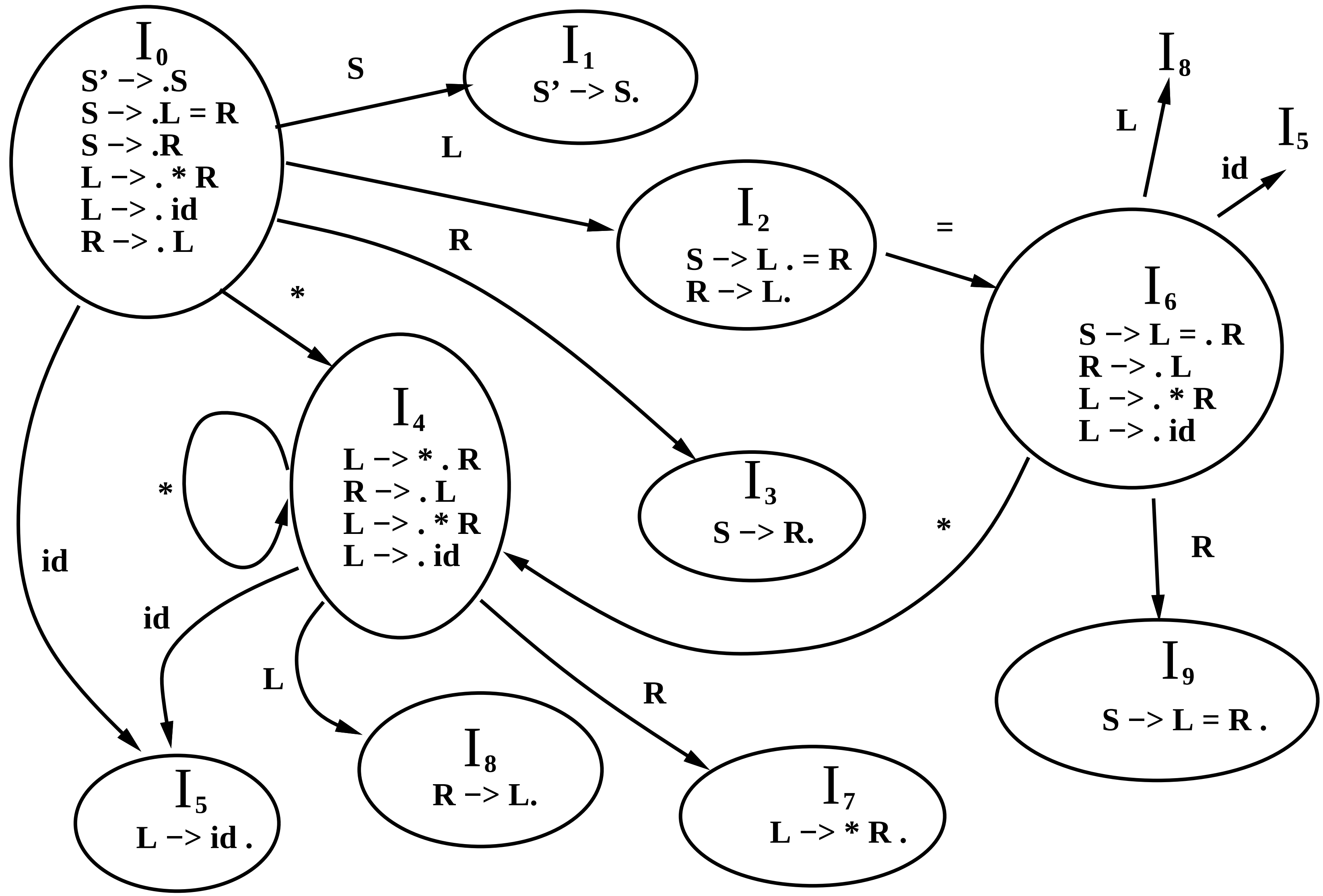
I_6:
- ▷ $S \rightarrow L = \cdot R$
- ▷ $R \rightarrow \cdot L$
- ▷ $L \rightarrow \cdot * R$
- ▷ $L \rightarrow \cdot id$

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 "$\$\ I_0\ L\ I_2$" 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 the viable prefix $\beta\alpha$ on the top of 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.**
  - **Construct FOLLOW(ω) on the fly.**
  - **Assume $\omega = \omega' X$ and FOLLOW(ω') is known.**
  - **Can FOLLOW$(\omega' X)$ be computed efficiently?**

$LR(1)$ 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.**
  - **Instead of maintaining FOLLOW sets of viable prefixes, we maintain FIRST sets of possible future extensions of the current viable prefix.**
- **Formally: $[A \rightarrow \alpha \cdot \beta, a]$ is valid (or reachable) for a viable prefix γ if there exists a derivation**

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

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

Examples of $LR(1)$ items

- **Grammar:**
  - $S \rightarrow BB$
  - $B \rightarrow aB \mid b$
- $$S \underset{rm}{\overset{*}{\Longrightarrow}} aaBab \underset{rm}{\Longrightarrow} aaaBab$$

  viable prefix aaa **can reach** $[B \rightarrow a \cdot B, a]$
- $$S \underset{rm}{\overset{*}{\Longrightarrow}} BaB \underset{rm}{\Longrightarrow} BaaB$$

  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,**

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

    ▷ *ω is ϵ or a sequence of terminals.*

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

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

    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**.

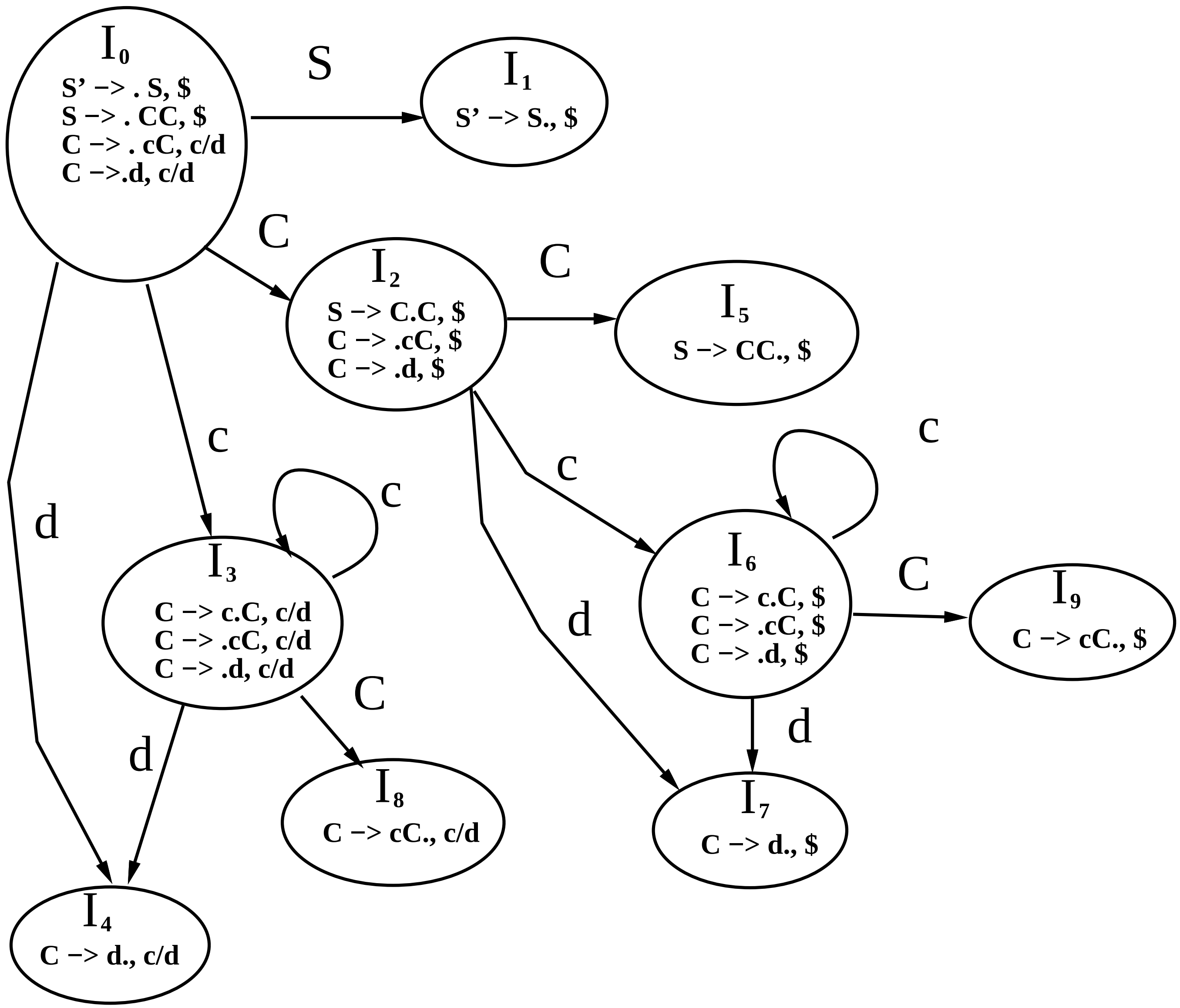
Algorithm for $LR(1)$ parsers

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

$LR(1)$ 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 occupies too much space.**

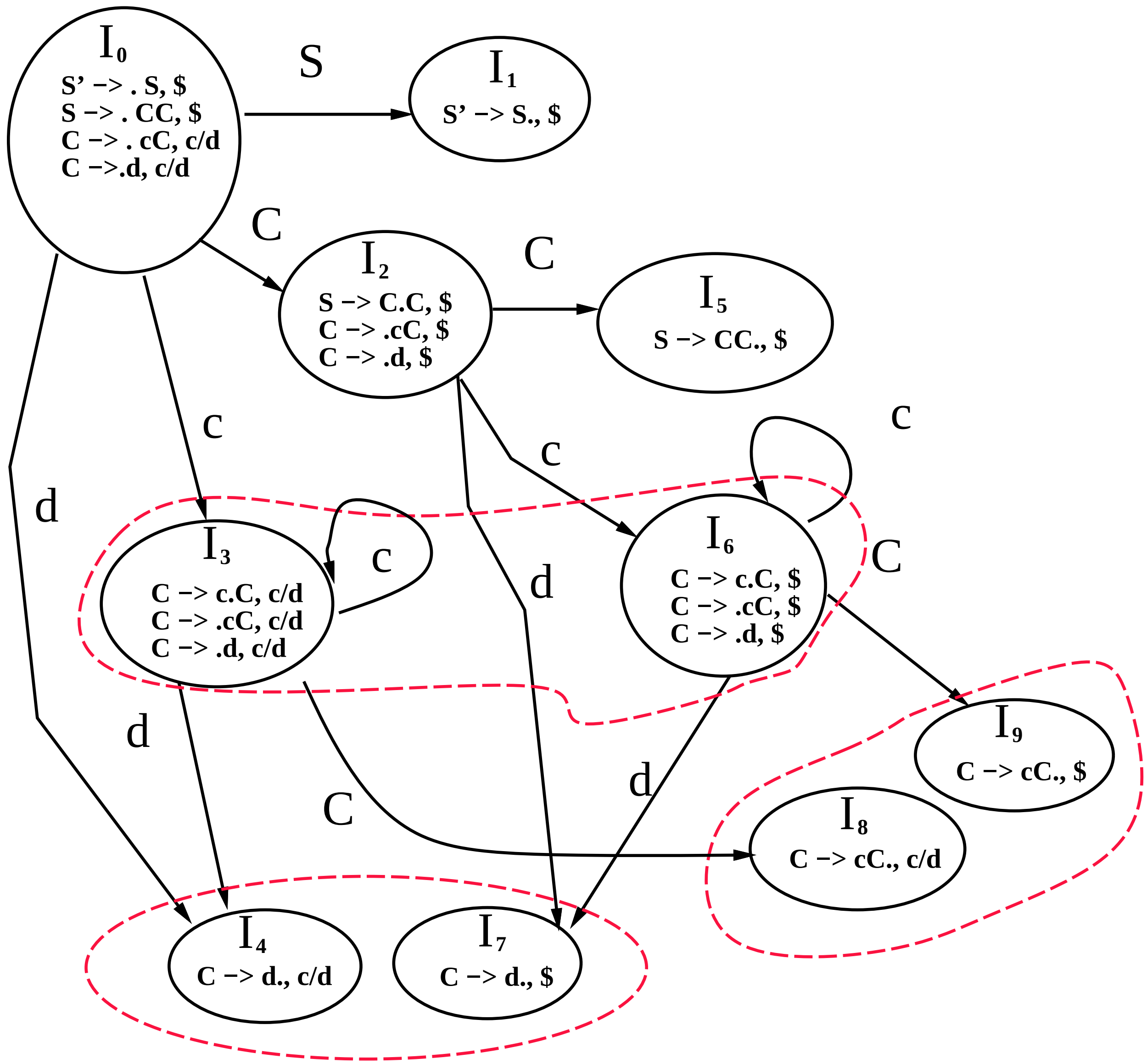
$LALR(1)$ parser — Lookahead LR

- The method that is often used in practice.
- Most common syntactic constructs of programming languages can be expressed conveniently by an $LALR(1)$ grammar [DeRemer 1969].
- $SLR(1)$ 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.**
  - $I_1 = \{[A \rightarrow \alpha\cdot, a], \ldots\}$
    - ▷ ***For I_1, one of the actions is to perform a reduce when the lookahead symbol is "a".***
  - $I_2 = \{[B \rightarrow \beta \cdot a\gamma, b], \ldots\}$
    - ▷ ***For I_2, one of the actions is to perform a shift on input "a".***
  - **Merging I_1 and I_2, the new state $I_{1,2}$ has shift-reduce conflicts.**
  - **However, we merge I_1 and I_2 because they have the same core.**
    - ▷ ***That is, $[A \rightarrow \alpha\cdot, c] \in I_2$ and $[B \rightarrow \beta \cdot a\gamma, d] \in I_1$.***
    - ▷ ***The shift-reduce conflict already occurs in I_1 and I_2.***
- **Merging of states can produce a new reduce-reduce conflict.**

$LALR(1)$ transition diagram



Possible new conflicts from $LALR(1)$

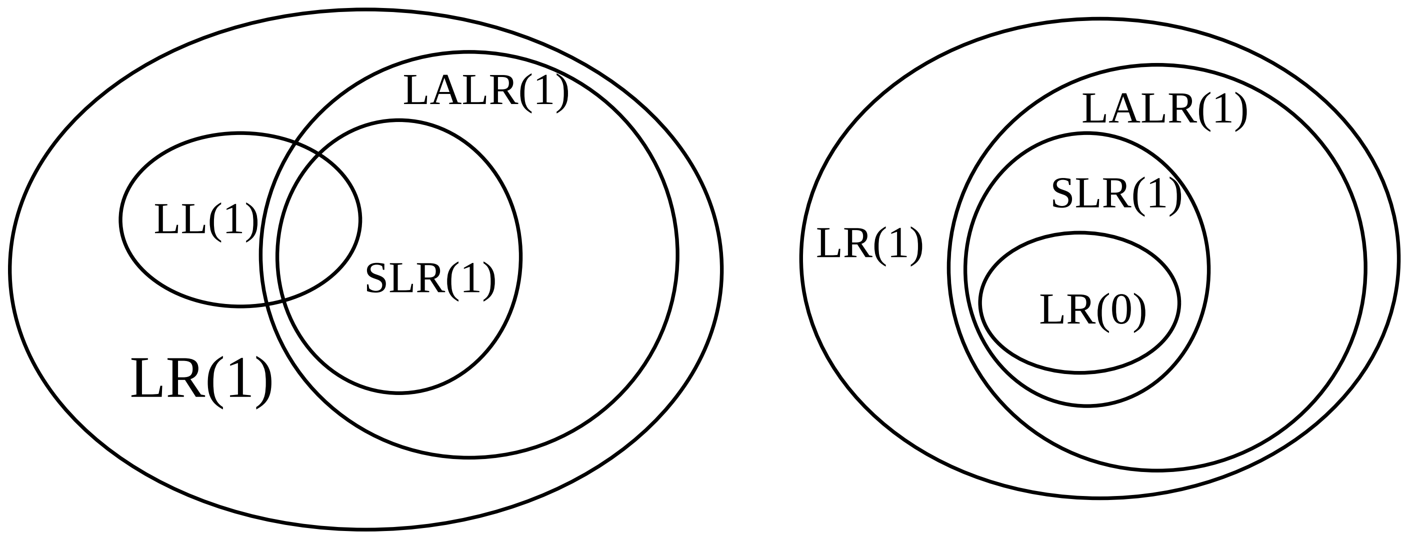
- **May produce a new reduce-reduce conflict.**
- **For example (textbook page 267, Example 4.58), grammar:**
  - $S' \to S$
  - $S \to aAd \mid bBf \mid aBe \mid bAe$
  - $A \to c$
  - $B \to c$
- **The language recognized by this grammar is $\{acd, ace, bcd, bce\}$.**
- **You may check that this grammar is $LR(1)$ by constructing the sets of items.**
- **You will find the set of items $\{[A \to c\cdot, d], [B \to c\cdot, e]\}$ is valid for the viable prefix ac, and $\{[A \to c\cdot, e], [B \to c\cdot, d]\}$ 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**
  - $\{[A \to c\cdot, d/e],$
  - $[B \to c\cdot, d/e]\}$**,**

  generates a reduce-reduce conflict, since reductions by both $A \to c$ and $B \to c$ 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 and/or time efficient methods to construct an $LALR(1)$ parsing table are known.**
  - **Constructing and merging on the fly.**
  - $\cdots$

Summary



- $LR(1)$ **and** $LALR(1)$ **can almost express all important programming languages issues, but** $LALR(1)$ **is easier to write and uses much less space.**
- $LL(1)$ **is easier to understand and uses much less space, but cannot express some important common-language features.**
  - **May try to use it first for your own applications.**
  - **If it does not succeed, then use more powerful ones.**