Syntax Analyzer — Parser

ASU Textbook Chapter 4.2–4.5, 4.7–4.9 (w/o error handling)

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 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**
    $A \rightarrow X_1, X_2, \ldots, X_m$**, where** $A \in N$ **and** $X_i \in T \cup N$**;**
  - S**: the starting nonterminal,** $S \in N$**.**
- **Notations:**
  - **terminals : lower case English strings, e.g.,** a, b, $c \cdots$
  - **nonterminals: upper case English strings, e.g.,** A, B, $C \cdots$
  - $\alpha, \beta, \gamma \in (T \cup N)^*$
    - ▷ ***α, β, γ: alpha, beta and gamma.***
    - ▷ ***ϵ: epsilon.***
  - $$\left.\begin{array}{lcl} A & \rightarrow & X_1 \\ A & \rightarrow & X_2 \end{array}\right\} \equiv A \rightarrow X_1 \mid X_2$$

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 2nd of the 4 productions.**
  - **The second step was by choosing the first production.**
- **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

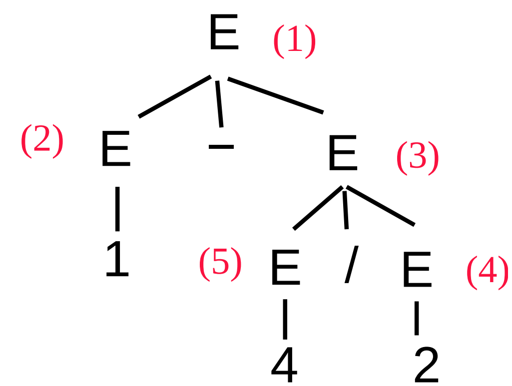
$$L(G) = \{w \mid S \stackrel{+}{\Longrightarrow} \omega\},$$

  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 different way to derive

- 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 $X \to \alpha$*
    - ▷ *symbols in α become children of X*
  - UNTIL no more leaf nonterminal left
- Need to annotate the order of derivation on the nodes.

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



Parse tree examples

- **Example:**

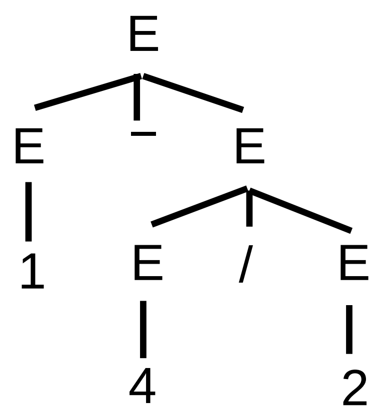
Grammar:

$E \to int$

$E \to E - E$

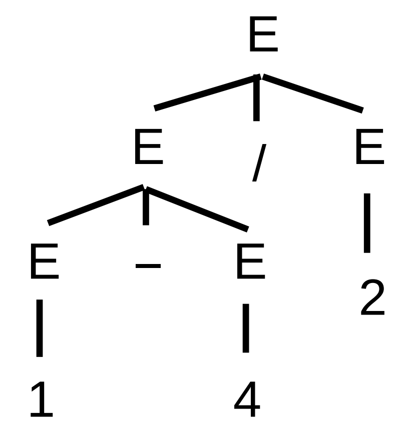
$E \to E/E$

$E \to (E)$



leftmost derivation

- **Using $1 - 4/2$ as the input, the left parse tree is derived.**
- **A string is formed by reading the lead nodes from left to right, given $1 - 4/2$.**
- **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 S, there are**
  - **more than one leftmost derivation for S, or**
  - **more than one rightmost derivation for S, or**
  - **more than one parse tree for S,**

  then G is called ambiguous .
  - **Note: the above three conditions are equivalent in that if one is true, then all three are true.**
- **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:**
    <idlist>→<idlist>, ID | ID
  - **For zero or more ID's,**
    - ▷ *<idlist>→ ϵ | ID |<idlist>, <idlist>*
      won't work due to ϵ; it can generate: ID, , ID
    - ▷ *<idlist>→ ϵ |<idlist>, ID | ID*
      won't 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.**
    - ▷ *<opt-idlist>→ ϵ |<nonEmptyIdlist>*
    - ▷ *<nonEmptyIdlist>→<nonEmptyIdlist>, ID | ID*
- **Expressions: precedence and associativity as discussed next.**

Grammar that expresses precedence correctly

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

Original grammar:

$E \rightarrow int$

$E \rightarrow E - E$

$E \rightarrow E/E$

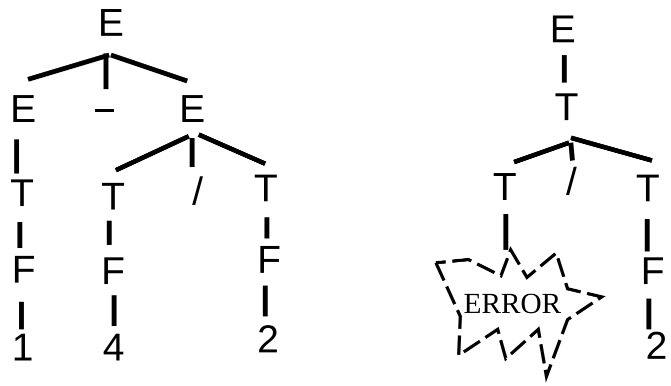
$E \rightarrow (E)$

Revised grammar:

$E \rightarrow E - E \mid T$

$T \rightarrow T/T \mid F$

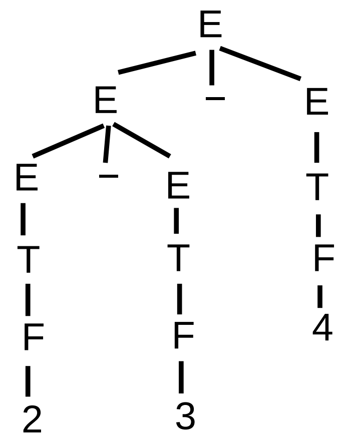
$F \rightarrow int \mid (E)$



rightmost derivation

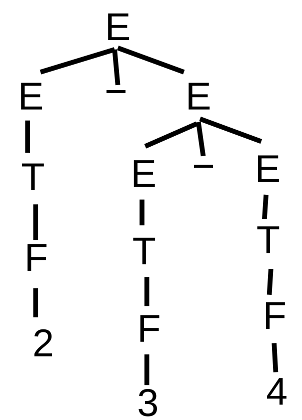
More problems with associativity

- **However, the above grammar is still ambiguous, and parse trees may not express the associative of $-$ and $/$.**
  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:	Revised grammar:	Final grammar:
$E \rightarrow int$	$E \rightarrow E - E \mid T$	$E \rightarrow E - T \mid T$
$E \rightarrow E - E$	$T \rightarrow T/T \mid F$	$T \rightarrow T/F \mid F$
$E \rightarrow E/E$	$F \rightarrow int \mid (E)$	$F \rightarrow int \mid (E)$
$E \rightarrow (E)$		

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

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:**

<procedure>→<type-def> ID <opt-params><opt-decl> {<opt-statements>}

- ▷ *<opt-params>→ (<list-params>)*
- ▷ *<list-params>→* ϵ *|<nonEmptyParlist>*
- ▷ *<nonEmptyParlist>→<nonEmptyIdlist>, ID | ID*

- **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.**

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: Y is useless.**
  - $S \rightarrow X \mid Y$
  - $X \rightarrow (\ )$
  - $Y \rightarrow (\ Y\ Y\ )$
- **Y derives more and more nonterminals and is useless.**
- **Any recursively defined nonterminal without a production of deriving ϵ 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.**

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 expressions $\equiv$ FA.**
  - **Context-free grammar**
  - **Context-sensitive**
  - $\cdots$
- $\{\omega c \omega \mid \omega$ **is a string of** a **and** b**'s**$\}$ **cannot be expressed by CFG.**

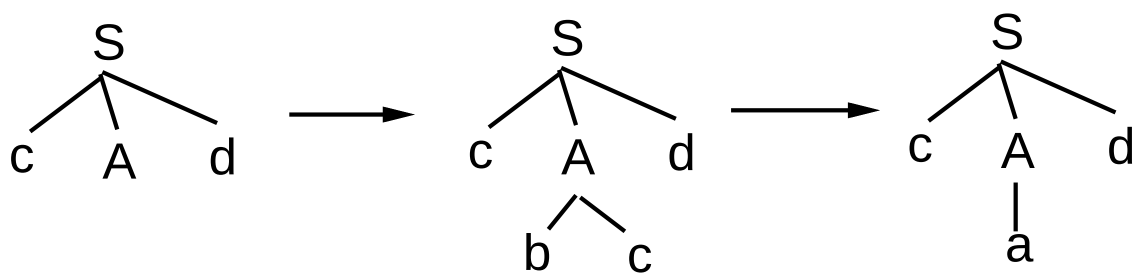
Top-down parsing

- 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!**

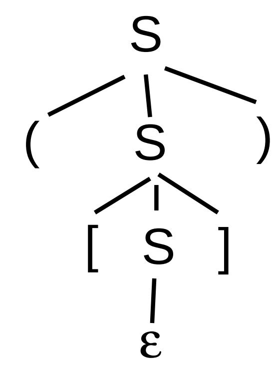
Predictive parser for $LL(1)$ grammars

- **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 and push the above production's right-hand-side*
    - ▷ *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:** $S \to \epsilon \mid (S) \mid [S]$ **input:** $([\ ])$



leftmost derivation

input	stack	action
(	S	pop, push "(S)"
(	(S)	pop, match with input
([	S)	pop, push "[S]"
([	[S])	pop, match with input
([ ]	S])	pop, push ϵ
([ ]	])	pop, match with input
([ ])	)	pop, match with input
([ ])		accept

- **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 $LL(1)$!
- 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 $()$.

input	stack	action
(	S	pop, but use which production?

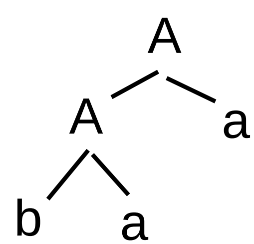
- In this example, we need 2-token look-ahead.
  - If the next token is $)$, push $()$.
  - If the next token is $($, push (S).
- Two questions:
  - How to tell whether a grammar G is $LL(1)$?
  - How to build the PARSING TABLE?

Properties of non-$LL(1)$ grammars

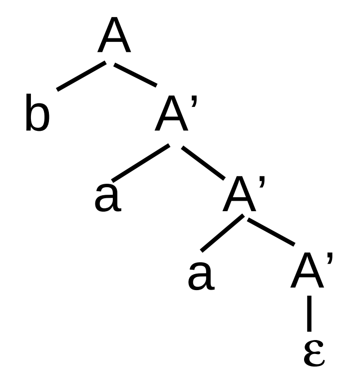
- **Theorem 1: A CFG grammar is not $LL(1)$ if it is left-recursive.**
- **Definitions:**
  - **recursive grammar: a grammar is recursive if the following is true for a nonterminal X in G:**
    $X \stackrel{+}{\Longrightarrow} \alpha X \beta$.
  - G **is left-recursive if** $X \stackrel{+}{\Longrightarrow} X\beta$.
  - 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 $L(G) \equiv L(G')$.**



leftmost derivation
original grammar G



leftmost derivation
revised grammar G'

- **Example:**

$$\boxed{\begin{array}{l}\textbf{input } baa \\ \beta \equiv b \\ \alpha \equiv a\end{array}}$$

Rule for removing immediate left-recursion

- Both grammar recognize the same string, but G' is not left-recursive.
- However, G is clear and intuitive.
- General rule for removing immediately left-recursion:
  - Replace $A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \beta_n$
  - with
    - $A \longrightarrow \beta_1 A' \mid \cdots \mid \beta_n A'$
    - $A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$
- 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.**
  - **Algorithm 4.1 works only if the grammar has no cycles or ϵ-productions.**
    - ▷ ***Cycle:*** $A \stackrel{+}{\Longrightarrow} A$
    - ▷ ***ϵ-production:*** $A \rightarrow \epsilon$
  - **It is possible to remove cycles and ϵ-productions 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*** $A_i \rightarrow A_j\gamma$
    - ▷ ***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}

Intuition for Algorithm 4.1

- **Intuition: if $A_{i_1} \stackrel{+}{\Longrightarrow} \alpha_2 A_{i_2} \beta_2 \stackrel{+}{\Longrightarrow} \alpha_3 A_{i_3} \beta_3 \stackrel{+}{\Longrightarrow} \cdots$ and $i_1 < i_2 < i_3 < \cdots$, then it is not possible to have recursion.**
- **Trace Algorithm 4.1**
  - **After each i-loop, only productions of the form $A_i \rightarrow A_k \gamma$, $i < k$ are left.**
  - $i = 1$
    - ▷ ***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*** $A_2 \rightarrow A_k \alpha \gamma$, ***where*** $A_1 \rightarrow A_k \alpha \gamma$ ***and*** $k > 1$
    - ▷ ***remove immediate left-recursion for*** A_2
  - $i = 3$
    - ▷ $j = 1$: ***replace*** $A_3 \rightarrow A_1 \delta_1$
    - ▷ $j = 2$: ***replace*** $A_3 \rightarrow A_2 \delta_2$
    - ▷ ***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:**
    - ▷ $A \rightarrow bdA' \mid eA'$
    - ▷ $A' \rightarrow cA' \mid adA' \mid \epsilon$

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

- **Theorem 2: G is not $LL(1)$ if a nonterminal has two productions whose right-hand-sides have a common prefix.**
  - ▷ ***Have left-factors.***
- **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**
  - $A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$ **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 process until A has no two productions with a common prefix.**

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 are left-factored may still not be $LL(1)$.

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**
    - ▷ ***if* α *can derive* ϵ*, then* $\epsilon \in$ *FIRST*(α)**
  - **FIRST$(\alpha) = \{t \mid$ (t is a terminal and $\alpha \stackrel{*}{\Longrightarrow} t\beta$) or ($t = \epsilon$ and $\alpha \stackrel{*}{\Longrightarrow} \epsilon$)$\}$**

How to compute FIRST(X)?

- 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,

  $$X \rightarrow Y_1 Y_2 \cdots Y_k$$

  Perforam the following step in sequence:
  - 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)
- Repeat the above process for all nonterminals until nothing can be added to any FIRST set.

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')**.**

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

How to compute FIRST(α)?

- **Given FIRST(X) for each terminal and nonterminal X, compute FIRST(α) for α being a sequence of terminals and/or nonterminals**
- **To build a parsing table, we need FIRST(α) for all α such that $X \rightarrow \alpha$ is a production in the grammar.**
- **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).**

Example for computing FIRST(α)

Grammar

$E \rightarrow E'T$

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

$T \rightarrow FT'$

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

$F \rightarrow int \mid (E)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **FIRST**$(T'E') =$
  - (***FIRST***$(T') - \{\epsilon\})\cup$
  - (***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, then \$ is in FOLLOW(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)

- 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.
- To see if a given grammar is $LL(1)$ and also to build its parsing table:
  - compute FIRST(α) for every production $X \rightarrow \alpha$
  - compute FOLLOW(X) for all nonterminals X
- Note that FIRST and FOLLOW sets are always sets of terminals, plus, perhaps, ϵ for some FIRST sets.

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_{g_i}) for only one g_i, then we use that derivation.**
  - **If $a \in$ FIRST(α_i) for two i, 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 some $\alpha_{g_i} \overset{*}{\Longrightarrow} \epsilon$ and $a \in$ FOLLOW(X), then we can can use the derivation $X \rightarrow \alpha_{g_i}$.**

Grammars that are not $LL(1)$

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

  $$A \rightarrow \alpha \mid \beta$$

  and any one of the followings is true:
  - **FIRST**$(\alpha) \cap$ **FIRST**$(\beta) \neq \emptyset$.
  - $\epsilon \in$ **FIRST**(α) **and FIRST**$(\beta) \cap$ **FOLLOW**$(A) \neq \emptyset$.
  - $\epsilon \in$ **FIRST**(α) **and** $\epsilon \in$ **FIRST**(β).
- **If a grammar is not $LL(1)$, then**
  - **you cannot write a linear-time predictive parser as described above;**
  - **we do not know to use the production $A \rightarrow \alpha$ or the production $A \rightarrow \beta$ when the lookadead symbol is a and, respectively,**
    - ▷ $a \in$ ***FIRST***$(\alpha) \cap$ ***FIRST***(β)***;***
    - ▷ $a \in$ ***FIRST***$(\beta) \cap$ ***FOLLOW***(A)***;***
    - ▷ $a \in$ ***FOLLOW***(A)***.***

A complete example (1/2)

- **Grammar:**
  - **<prog_head>→ PROG ID <file_list> SEMICOLON**
  - **<file_list>→ ϵ | L_PAREN <file_list> SEMICOLON**
- **FIRST and FOLLOW sets:**

α	FIRST(α)	FOLLOW(α)
ϵ	$\{\epsilon\}$	
<prog_head>	{PROG}	{$}
<file_list>	$\{\epsilon,$ L_PAREN$\}$	{SEMICOLON}
PROG ID <file_list> SEMICOLON	{PROG}	
L_PAREN <file_list> SEMICOLON	{LPAREN}	

A complete example (2/2)

Input: PROG ID SEMICOLON

Input	stack	action
	<prog_head> \$	
PROG	<prog_head> \$	pop, push
PROG	PROG ID <file_list> SEMICOLON \$	match input
ID	ID <file_list> SEMICOLON \$	match input
SEMICOLON	<file_list> SEMICOLON \$	WHAT TO DO?

- **Last actions:**
  - **Two choices:**
    - ▷ *<file_list>* $\rightarrow \epsilon$ | *L_PAREN <file_list> SEMICOLON*
  - **SEMICOLON** $\notin$ **FIRST**(ϵ) **and** **SEMICOLON** $\notin$ **FIRST**(**L_PAREN** <**file_list**> **SEMICOLON**)
  - <**file_list**> $\stackrel{*}{\Longrightarrow} \epsilon$
  - **SEMICOLON** $\in$ **FOLLOW**(<**file_list**>)
  - **Hence we use the derivation** <**file_list**> $\rightarrow \epsilon$

$LL(1)$ 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 leaves to the root.**
- **Example:**

  Grammar:

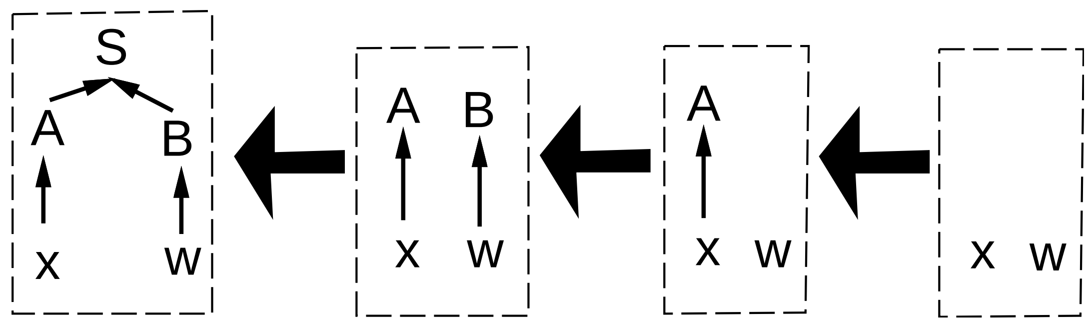
  $S \rightarrow AB$

  $A \rightarrow x \mid Y$

  $B \rightarrow w \mid Z$

  $Y \rightarrow xb$

  $Z \rightarrow wp$



- **Input xw.** $S \underset{rm}{\Longrightarrow} AB \underset{rm}{\Longrightarrow} Aw \underset{rm}{\Longrightarrow} xw$.
- **This grammar is not $LL(1)$.**

Definitions (1/2)

- **Rightmost derivation:**
  - $S \underset{rm}{\Longrightarrow} \alpha$: **the rightmost nonterminal is replaced.**
  - $S \overset{+}{\underset{rm}{\Longrightarrow}} \alpha$: α **is derived from** S **using one or more rightmost derivations.**
    - ▷ ***α is called a right-sentential form .***
- **Define similarly leftmost derivations.**
- **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 in** γ **where** β **can be found.**

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 **in** γ **with** A**.**
- **A right-most derivation in reverse can be obtained by handle reducing.**

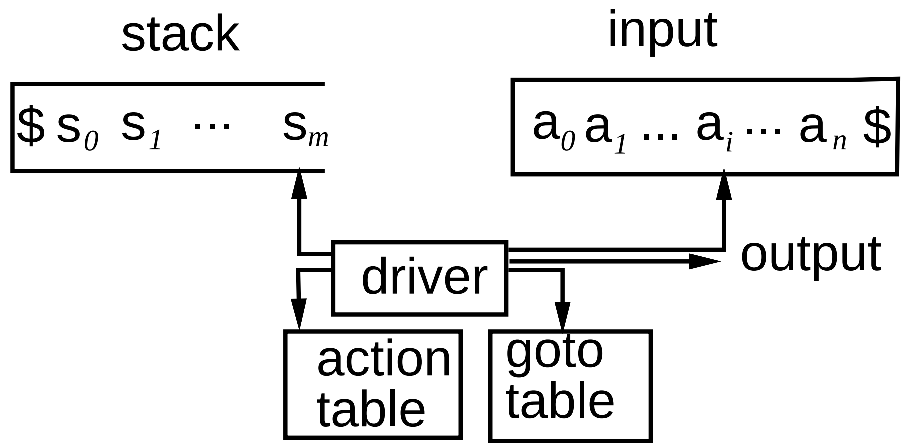
STACK implementation

- **Four possible actions:**
  - **shift: shift the input to STACK.**
  - **reduce: perform a reversed rightmost derivation.**
  - **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

- **viable prefix: the set of prefixes of right sentential forms that can appear on the stack.**

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.**

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.**

$$x_1, x_2, \ldots, \underbrace{\boxed{x_i, x_{i+1}, \ldots, x_{i+j},}}_{\text{a handle}} \underbrace{\boxed{x_{i+j+1}, \ldots, x_{i+j+k-1},}}_{\text{lookahead tokens}} \ldots$$

- **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

- Construct a FSA to recognize all possible viable prefixes.
- An $LR(0)$ item (item for short) is a production, with a dot at some position in the RHS (right-hand side). For example:
  - $A \rightarrow XYZ$
    - $A \rightarrow \cdot XYZ$
    - $A \rightarrow X \cdot YZ$
    - $A \rightarrow XY \cdot Z$
    - $A \rightarrow XYZ\cdot$
  - $A \rightarrow \epsilon$
    - $A \rightarrow \cdot$

  The dot indicates the place of a handle.
- Assume G is a grammar with the starting symbol S. Augmented grammar G' is to add a new starting symbol S' and a new production $S' \rightarrow S$ to G. We assume working on the augmented grammar from now on.

Closure

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

Example for the closure function

- **Example:**
  - $E' \rightarrow E$
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * F \mid F$
  - $F \rightarrow (E) \mid id$

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

  - $\{E' \rightarrow \cdot E$,
  - $E \rightarrow \cdot E + T$,
  - $E \rightarrow \cdot T$,
  - $T \rightarrow \cdot T * F$,
  - $T \rightarrow \cdot F$,
  - $F \rightarrow \cdot (E)$,
  - $F \rightarrow \cdot id\}$

GOTO table

- $GOTO(I, X)$, **where** I **is a set of items and** X **is a legal symbol is defined as**
  - **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 group 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: 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 the state can be derived.**

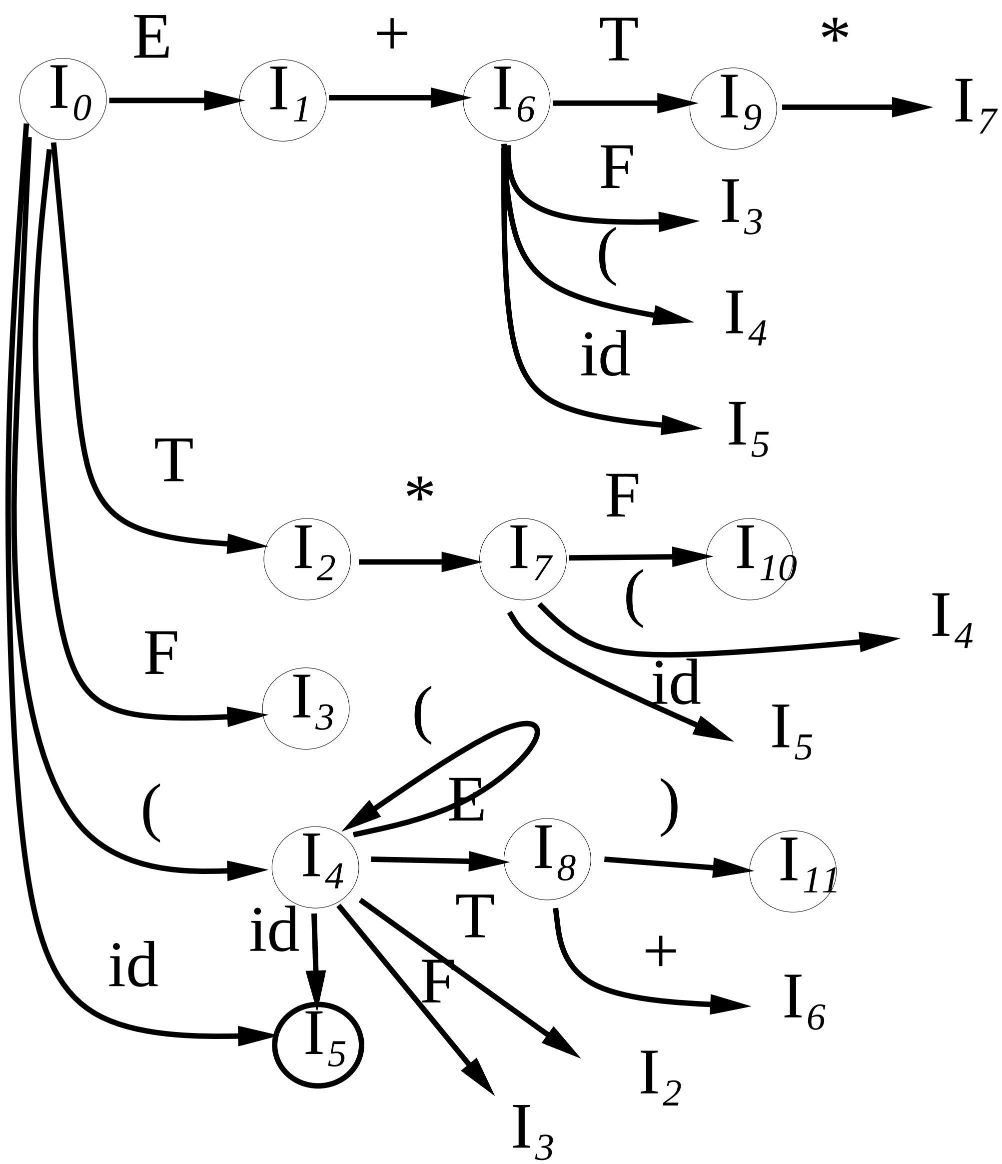
Example of sets of $LR(0)$ items

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

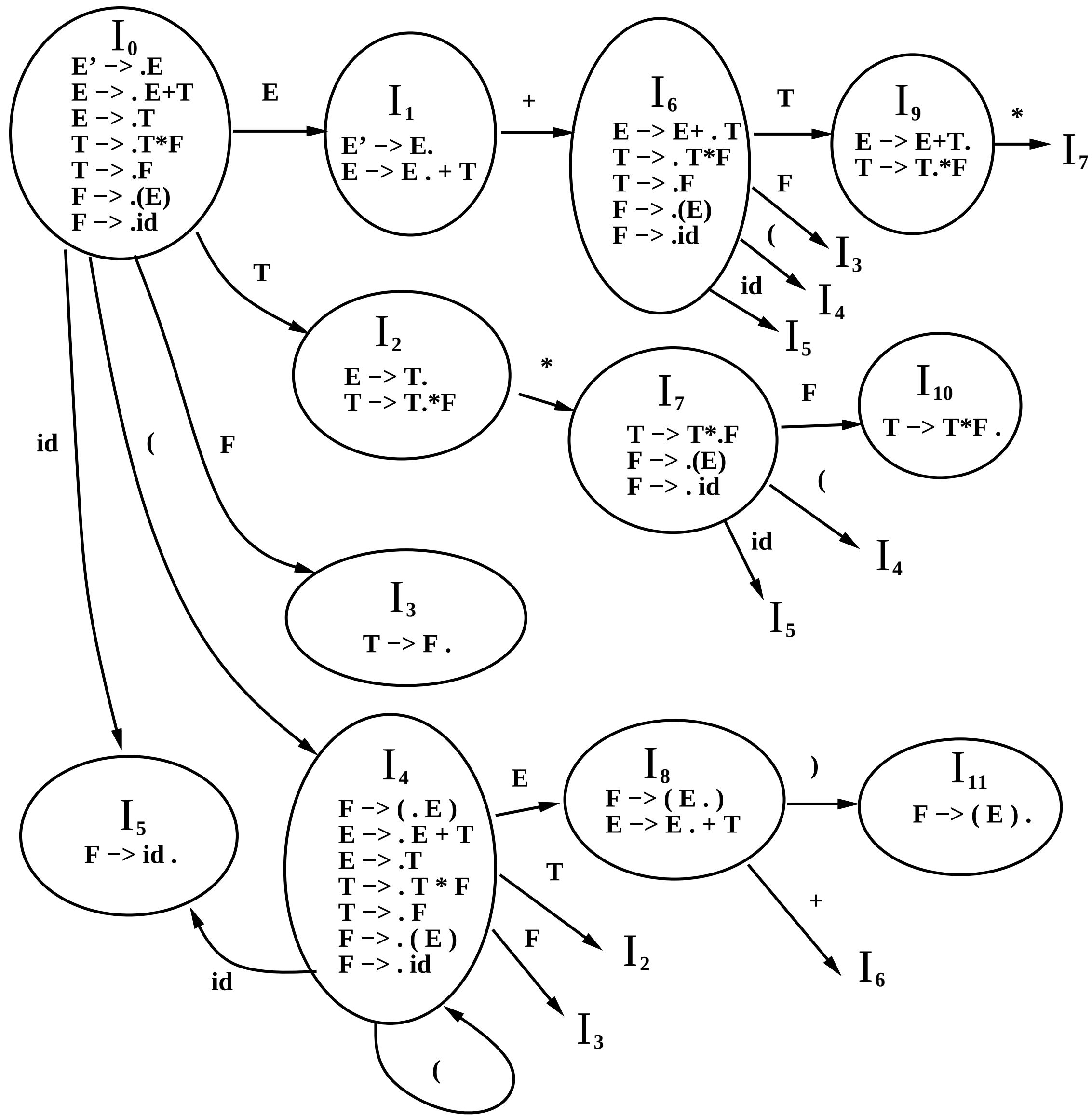
$I_0 = closure(\{E' \rightarrow \cdot E\}) =$
$\{E' \rightarrow \cdot E,$
$E \rightarrow \cdot E + T,$
$E \rightarrow \cdot T,$
$T \rightarrow \cdot T * F,$
$T \rightarrow \cdot F,$
$F \rightarrow \cdot(E),$
$F \rightarrow \cdot id\}$

- **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:

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

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

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

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

- $closure(I)$: a state/configuration during parsing recording all possible things that we are expecting.
- 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

- **Input: id * id + id**

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

$LR(0)$ parsing

- LR parsing without lookahead symbols.
- Constructed from DFA for recognizing 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 goto the state $GOTO(I, A)$ where I is the state on the top of the stack after removing β
- Conflicts:
  - 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

state	action						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- **ri means reduce by production numbered i.**
- **si means shift and then go to state I_i.**
- **Use FOLLOW(A) to resolve some conflicts.**

Discussion (1/3)

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

Discussion (2/3)

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: $L \rightarrow id\cdot$

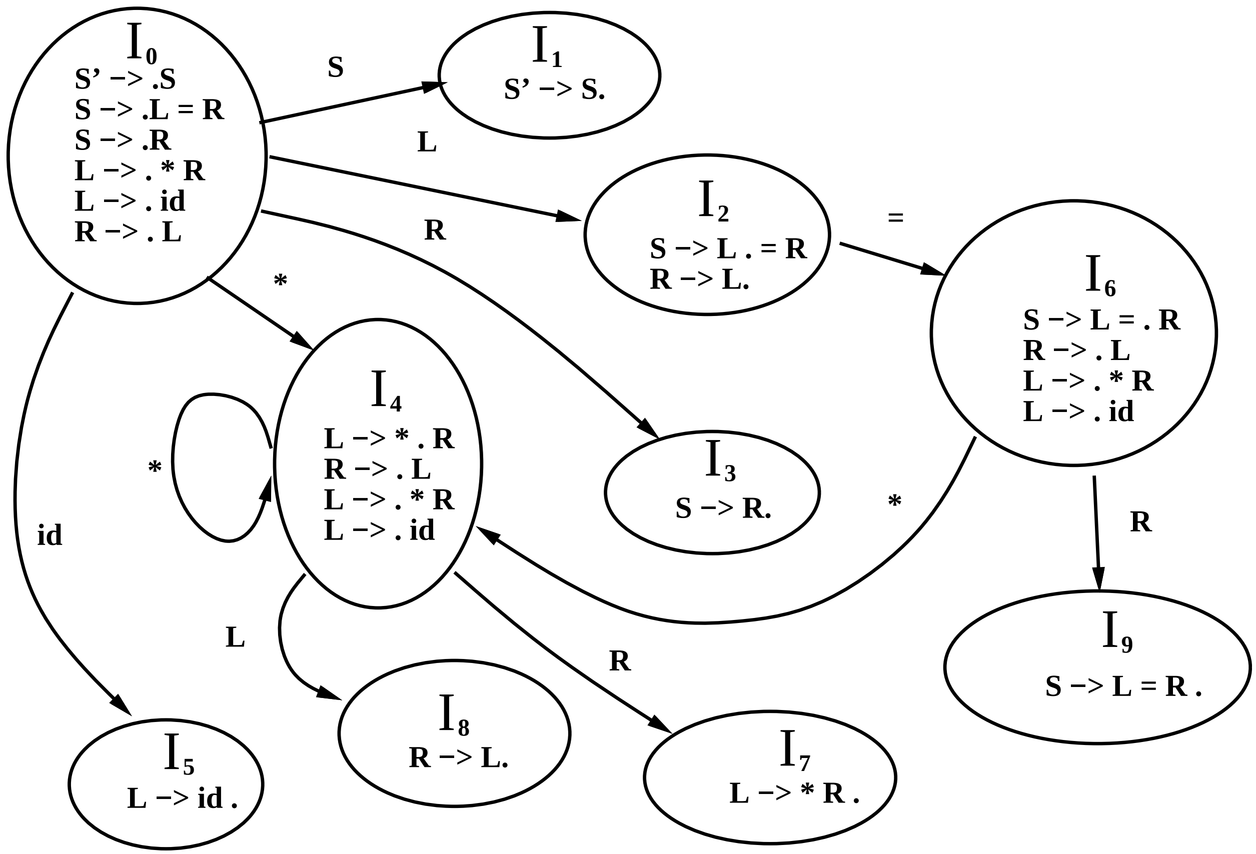
I_6:

- $S \rightarrow L = \cdot R$
- $R \rightarrow \cdot L$
- $L \rightarrow \cdot * R$
- $L \rightarrow \cdot id$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

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



Discussion (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, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a.**
- **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

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

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

$LR(1)$ parsing example

- **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** $[A \to \alpha \cdot B\beta, a]$ **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** $B \to \eta$ **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 $[B \to \cdot\eta, b]$ **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 .**

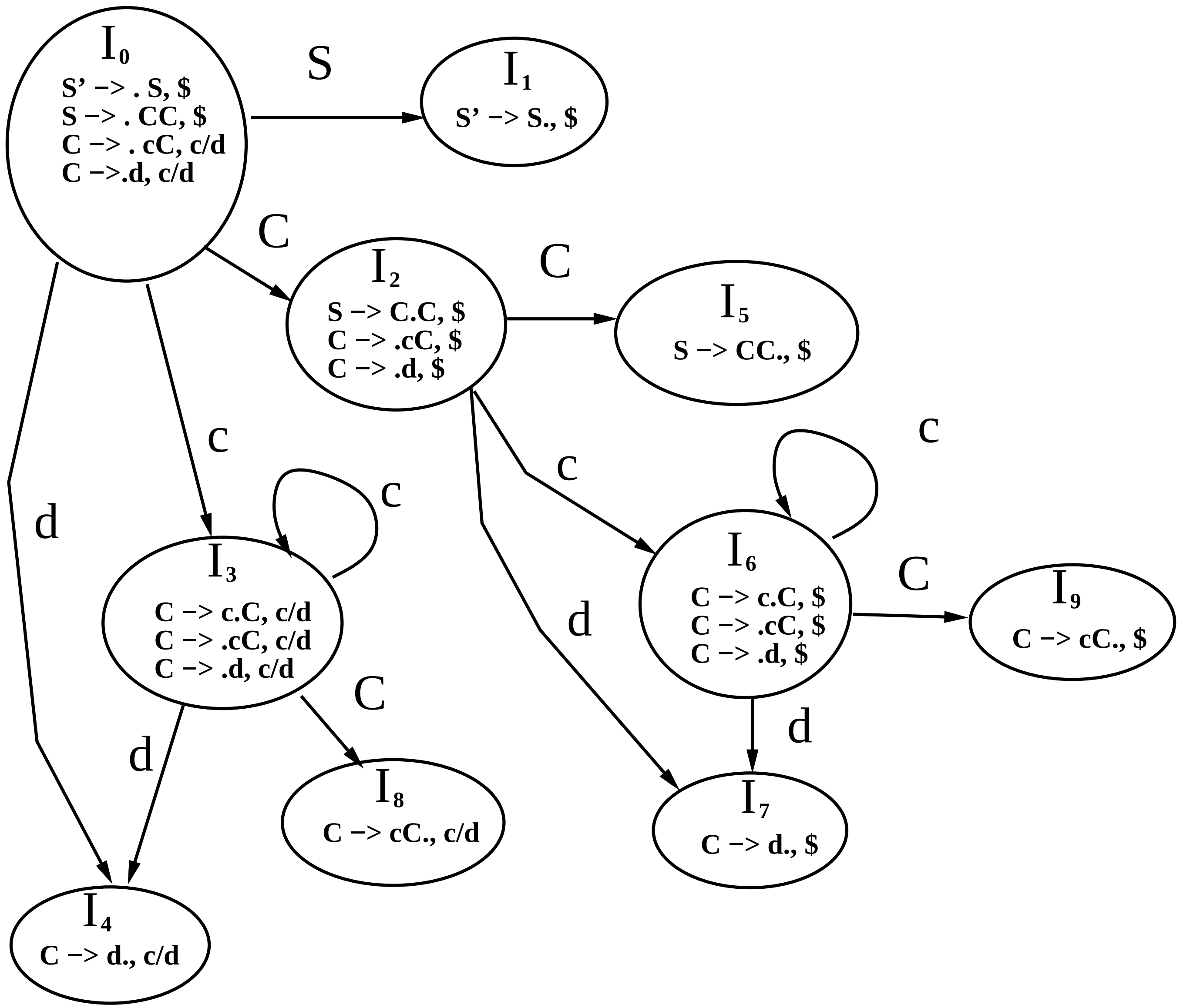
Algorithm for $LR(1)$ parsing functions

- $closure(I)$
  - **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
- $GOTO(I, X)$
  - **let** $J = \{[A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X\beta, a] \in I\}$.
  - **return** $closure(J)$
- $items(G')$
  - $C \leftarrow \{closure(\{[S' \rightarrow \cdot S, \$]\})\}$
  - **repeat**
    - ▷ ***for each set of items*** $I \in C$ ***and each grammar symbol*** X ***such that*** $GOTO(I, X) \neq \emptyset$ ***and*** $GOTO(I, X) \notin C$ ***do***
    - ▷ ***add*** $GOTO(I, X)$ ***to*** C
  - **until no more sets of items can be added to** C

Example for constructing $LR(1)$ closures

- **Grammar:**
  - $S' \rightarrow S$
  - $S \rightarrow CC$
  - $C \rightarrow cC \mid d$
- $closure(\{[S' \rightarrow \cdot S, \$]\})$ **=**
  - $\{[S' \rightarrow \cdot S, \$],$
  - $[S \rightarrow \cdot CC, \$],$
  - $[C \rightarrow \cdot cC, c/d],$
  - $[C \rightarrow \cdot d, c/d]\}$
- **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

Algorithm for $LR(1)$ 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 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 $GOTO(I_i, a) = I_j$, then**
    $action[I_i, a] =$ "shift j" for a is a terminal.
  - **if $[A \rightarrow \alpha\cdot, a] \in I_i$ and $A \neq S'$, then**
    $action[I_i, a] =$ "reduce by $A \rightarrow \alpha$"
  - **if $[S' \rightarrow S., \$] \in I_i$, then**
    $action[I_i, \$] =$ "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 $[S' \rightarrow \cdot S, \$]$.**

An example of an $LR(1)$ parsing table

	action			GOTO	
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**
  - **too many states and thus occupy too much space**
  - **most powerful**

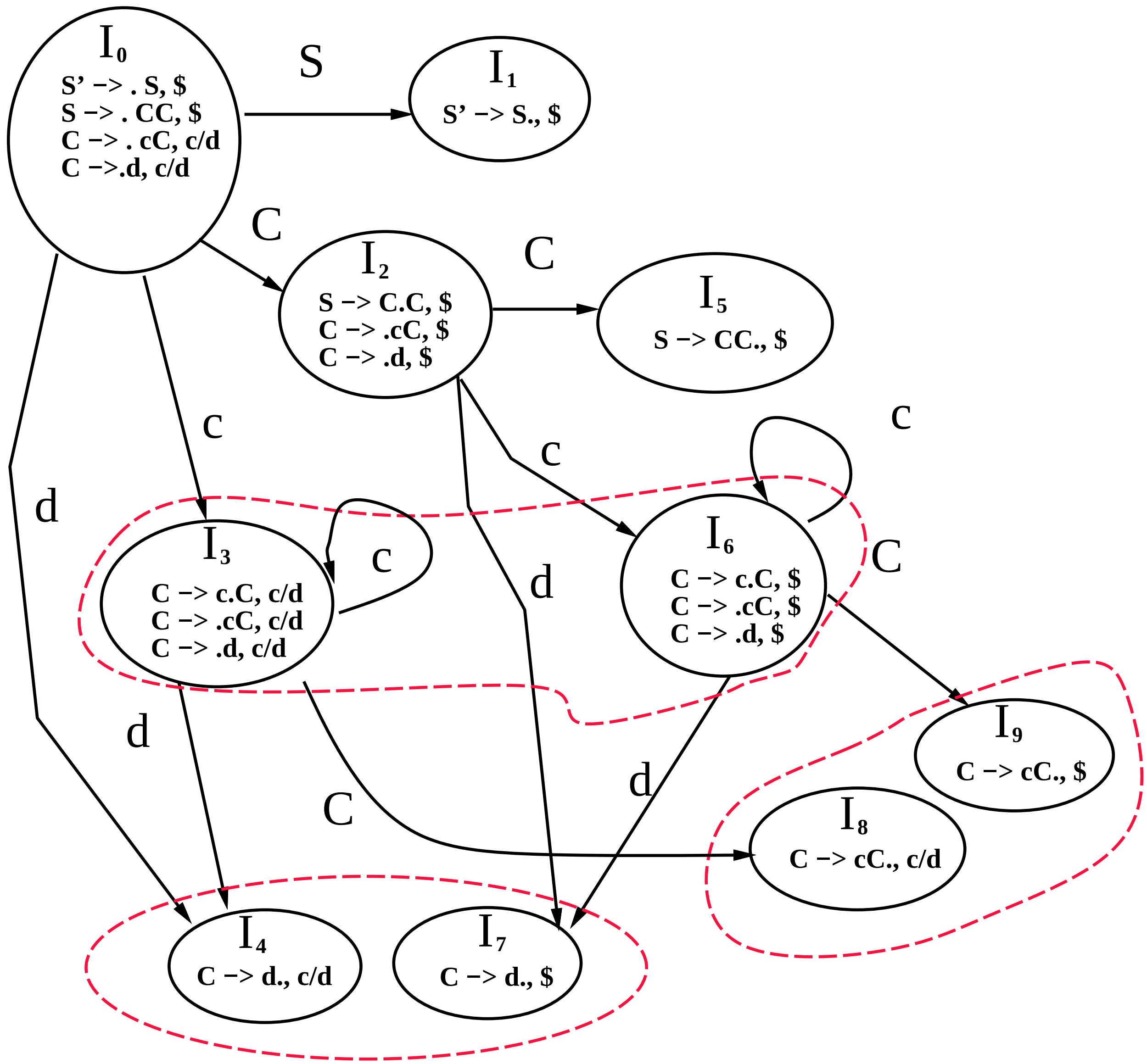
$LALR(1)$ parser — Lookahead LR

- 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 in the form of $[A \rightarrow \alpha \cdot \beta, c]$
- 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 $LR(1)$ parser, it is a common thing that several states only differ in lookahead symbol, 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 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 since I_1 and I_2 have the same core.
  - So $[A \rightarrow \alpha\cdot, c] \in I_2$ 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



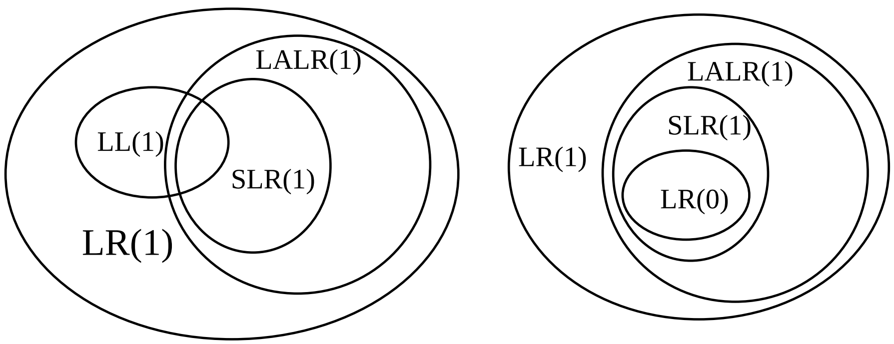
Possible new conflicts from $LALR(1)$

- **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.

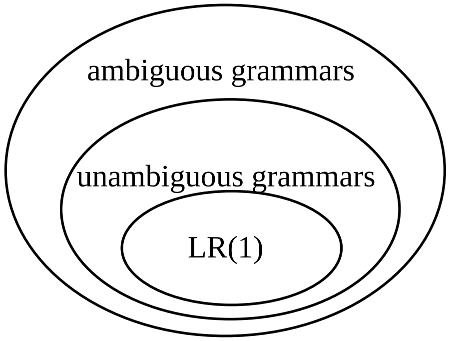
How to construct $LALR(1)$ parsing table

- **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.**
  - **Construction and merging on the fly.**



- **Summary:**
- **$LR(1)$ and $LALR(1)$ can almost handle all programming languages, but $LALR(1)$ is easier to write.**
- **$LL(1)$ is easier to understand.**

Using ambiguous grammars



- **Ambiguous grammars provides a shorter, more natural specification than any equivalent unambiguous grammars.**
- **Sometimes need ambiguous grammars to specify important language constructs.**
- **For example: declare a variable before its usage.**

```
var xyz : integer
begin
    ...
    xyz := 3;
    ...
```

Ambiguity from precedence and associativity

- **Use precedence and associativity to resolve conflicts.**
- **Example:**
  - G_1:
    - ▷ $E \rightarrow E + E \mid E * E \mid (E) \mid id$
    - ▷ ***ambiguous, but easy to understand!***
  - G_2:
    - ▷ $E \rightarrow E + T \mid T$
    - ▷ $E \rightarrow T * F \mid F$
    - ▷ $F \rightarrow (E) \mid id$
    - ▷ ***unambiguous, but it is difficult to change the precedence;***
    - ▷ ***parse tree is much larger for G_2, and thus takes more time to parse.***
- **When parsing the following input for G_1: $id + id * id$.**
  - **Assume the input parsed so far is $id + id$.**
  - **We now see "*".**
  - **We can either shift or perform "reduce by $E \rightarrow E + E$".**
  - **When there is a conflict, say in $SLR(1)$ parsing, we use precedence and associativity information to resolve conflicts.**

Dangling-else ambiguity

- **Grammar:**
  - $S \rightarrow a \mid if$ <**condition**> $then$ <**statement**>
    $\mid if$ <**condition**> $then$ <**statement**> $else$ <**statement**>
- **When seeing**

  <u>if c then S</u> else S
  - **shift or reduce conflict;**
  - **always favor a shift.**
  - **Intuition: favor a longer match.**

Special cases

- **Ambiguity from special-case productions:**
  - **Sometime a very rare happened special case causes ambiguity.**
  - **It's too costly to revise the grammar. We can resolve the conflicts by using special rules.**
  - **Example:**
    - ▷ $E \rightarrow E\ sub\ E\ sup\ E$
    - ▷ $E \rightarrow E\ sub\ E$
    - ▷ $E \rightarrow E\ sup\ E$
    - ▷ $E \rightarrow \{E\} \mid character$
  - **Meanings:**
    - ▷ $W\ sub\ U$: W_U.
    - ▷ $W\ sup\ U$: W^U.
    - ▷ $W\ sub\ U\ sup\ V$ is W_U^V, not ${W_U}^V$
  - **Resolve by semantic and special rules.**
  - **Pick the right one when there is a reduce/reduce conflict.**
    - ▷ *Reduce the production listed earlier.*

YACC (1/2)

- **Yet Another Compiler Compiler:**
  - **A UNIX utility for generating $LALR(1)$ parsing tables.**
  - **Convert your YACC code into C programs.**
  - **file.y $\longrightarrow$ $\boxed{\text{yacc file.y}}$ $\longrightarrow$ y.tab.c**
  - **y.tab.c $\longrightarrow$ $\boxed{\text{cc -ly -ll y.tab.c}}$ $\longrightarrow$ a.out**
- **Format:**
  - **declarations**
  - **%%**
  - **translation rules**
    - ▷ ***<left side>: <production>***
    - ▷ **{ *semantic rules* }**
  - **%%**
  - **supporting C-routines.**

YACC (2/2)

- **Assume the Lexical analyzer routine is $yylex()$.**
- **When there are ambiguities:**
  - **reduce/reduce conflict: favor the one listed first.**
  - **shift/reduce conflict: favor shift. (longer match!)**
- **Error handling:**
  - **Use special error handling productions.**
  - **Example:**
    ```
    lines: error '\n' {...}
    ```
  - **when there is an error, skip until newline.**
  - $error$**: special token.**
  - $yyerror(string)$**: pre-defined routine for printing error messages.**
  - $yyerrok()$**: reset error flags.**

YACC code example (1/2)

```
%{
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#define YYSTYPE int /* integer type for YACC stack */

%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%left UMINUS

%%
```

YACC code example (2/2)

```
lines   :  lines expr '\n'        {printf("%d\n", $2);}
        |  lines '\n'
        |  /* empty, i.e., epsilon */
        |  lines error '\n' { yyerror("Please reenter:"); yyerrok; }
        ;
expr    :  expr '+' expr    { $$ = $1 + $3; }
        |  expr '-' expr    { $$ = $1 - $3; }
        |  expr '*' expr    { $$ = $1 * $3; }
        |  expr '/' expr    { $$ = $1 / $3; }
        |  '(' expr ')'     { $$ = $2; }
        |  '-' expr  %prec UMINUS  { $$ = - $2; }
        |  NUMBER      { $$ = atoi(yytext);}
        ;

%%
#include "lex.yy.c"
```

Included Lex program

```
%{
%}
Digit         [0-9]
IntLit        {Digit}+
%%
[ \t] {/* skip white spaces */}
[\n] {return('\n');}
{IntLit}                         {return(NUMBER);}
"+"                              {return('+');}
"-"                              {return('-');}
"*"                              {return('*');}
"/"                              {return('/');}
.         {printf("error token <%s>\n",yytext); return(ERROR);}
%%
```

YACC rules

- **Can assign associativity and precedence.**
  - **in increasing precedence**
  - **left/right or non-associativity**
    - ▷ *Dot products of vectors has no associativity.*
- **Semantic rules: every item in the production is associated with a value.**
  - **YYSTYPE: the type for return values.**
  - **$$: the return value if the production is reduced.**
  - **$$i$: the return value of the ith item in the production.**
- **Actions can be inserted in the moddle of a production, each such action is treated as a nonterminal.**
  - **Example:**

```
expr    :  expr { $$ = 32;}  '+' expr    { $$ = $1 + $2 + $4; };

is equivalent to

expr  :  expr $ACT '+' expr {$$ = $1 + $2 + $4;};
$ACT : {$$ = 32;};
```

YACC programming styles

- **Avoid in-production actions.**
  - **Replace them by markers.**
- **Keep the right hand side of a production short.**
  - **Better to be less than 4 symbols.**
- **Avoid using C-language reserved words.**
  - **Watch out C-language rules.**
- **Try to find some unique symbols for each production.**
  - **array $\rightarrow$ ID [elist]**
  -
    - $\boldsymbol{arrary \rightarrow aelist\ ]}$
    - $\boldsymbol{aelist \rightarrow aelist,\ ID \mid ahead}$
    - $\boldsymbol{ahead \rightarrow ID\ [\ ID}$