Synthesizetreefromfragments,reconstruct rightmostderivation fromlefttoright. Automatonperformstwoactions:
- **shift**: pushnextinputsymboltostack.
- **reduce**: reducesstackfromtoptosinglestring ontheright.

Abottom-uprecognitionbyashift-reduce PDA is given by:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((())$</td>
<td>$(())$</td>
<td>reduce</td>
</tr>
<tr>
<td>$(SS)$</td>
<td>$S$</td>
<td>reduce</td>
</tr>
<tr>
<td>$(())$</td>
<td>$(())$</td>
<td>shift</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
<td>shift</td>
</tr>
<tr>
<td>$S$</td>
<td>$(S)$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Theoperationsherearestill (copyfrominputtostack) and reduce

Thebottom-uprecognizesententialforms withtwohandles, e.g.
\[
pn * p + p \quad \iff \quad pn * p \quad \iff \quad p + p \quad \iff \quad p
\]

On theotherhand, foranambiguousgrammarsuchas:

A production $A \rightarrow \epsilon$ is a handle in a sentential form if there is a rightmost derivation.

Stringsthatapearinarightmost derivation.$A$ (right)

A (right) sentential form is a mixed (terminals and non-terminals)

Handles

Technically more general than LL($k$).

- **contains $S$:**
  - if accepts when all input has been consumed and the stack
  - acceptstheemptystring
- **reduce $A \rightarrow \alpha$:**
  - reduces stack from top to bottom: if there exists a rightmost derivation of the form $\alpha \rho A \alpha' \beta \rho \epsilon$
- **shift $\beta$:** push next input symbol to stack.

Automatonpatternstwo actions:

- **from left to right:**
  - Synthesizefree from fragments, reconstruct rightmost derivation
- **from right to left:**
  - Recognizewith shift reduce PDA
Properties of LR(k) Grammars

There exist unambiguous grammars which are not LR(k).

Example:
The grammar

\[ S \rightarrow Ce \mid Dd \mid aCb \]
\[ C \rightarrow aCh \mid ab \]
\[ D \rightarrow aDb \mid abb \]

which generates the language

\[ \{ab^i \mid i \geq 0\} \cup \{a^ib^j \mid i \geq 1, j \geq 0\} \]

is not LR(k) for any \( k \geq 0 \). This is because the prefix

\[ a^ib^j \mid i \geq 1, j \geq 0 \]

does not uniquely determine whether the handle is \( C \rightarrow a \) or \( D \rightarrow abb \).

On the other hand, the language

\[ \{a^ib^j \mid i \geq 0\} \cup \{a^ib^j \mid i \geq 1, j \geq 0\} \]

has an LR(0) grammar.

Recognizing Reducible Strings

For each rightmost derivation as a reducible string.

We use a stack to hold the prefix of the analyzed sentential form up to (and including) the handle. Initially, the stack is empty.

We then follow the rules:

- If the stack contains a handle at its top, reduce the handle to the corresponding non-terminal. For an LR(k) grammar, this may require looking at the first \( k \) input characters.
- If the stack does not contain a handle at its top, but is still a prefix of a sentential form – shift the next input token to the stack top.
- If the stack contains \( S \) and the input is empty, then accept.
- Otherwise, reject.

Implementation of Shift-Reduce Parsing

It follows from the properties of rightmost derivations that we never have to look for handles deeper in the stack.

Claim 7. [Regularity of Reducible Strings] The set of reducible strings \( \alpha \beta \) is a regular language. We refer to a prefix of a reducible string as a viable prefix. In an LR parsing, we expect all stack contents to be viable prefixes.
We will show how to construct an finite-state classifier for the set of reducible strings of an LR(0) grammar.

The classifier is a DFA with several accepting states. Each accepting state identifies the production A which has been applied last when generating the reducible string. Constructing an NFAs and then determinizing it into a DFA.

The General Construction

For each production X → Y₁ ... Yₙ, construct a right-linear rule:

h Xᵢ! Y₁ Yₙ

Eliminate any productions of the form h Xᵢ! h Xᵢ.

For each non-terminal Yᵢ ∈ [1...n], construct a right-linear rule:

h Xᵢ! Y₁ Yₙ

For each production X → Y₁ ... Yₙ, construct a classifier via right-linear grammar.

A classifier for LR(0) parsing

The classifier also the rule

⟨X⟩ ← ⟨X⟩

A Reducible Strings Classifier

Corresponding to the grammar:

p₁ | (A) ← A
p₁ | A * L ← L
p₁ | L + E ← E
p₁ | E ← E
Initially, stack is empty.

At any step, if stack contains $E_0$ (start symbol) and input is empty, accept.

Otherwise, let $a$ be the next incoming input character.

Run the classifier on the stack contents:
- If the classifier reaches an accepting state annotated with $A!$, such that $A \notin \text{Follow}(A)$, then reduce with $A \rightarrow A'$ such that $A \notin \text{Follow}(A)$.
- Otherwise, if $I_j$ is the state exposed by this removal, then push $(I_j; A)$ to the stack.

If any step, if stack contains $E$ (start symbol) and input is empty, accept.

Replace Symbol Stack by State Stack

Instead of running the LR(0) classifier on the stack at each step, we can keep a stack of the classifier states. This leads to the following parsing algorithm:

Initially, stack contains state $I_0$.

At any step, if stack contains $I_0$ and input is empty, accept.

Otherwise, let $a$ be the next incoming input character.

If $(I_j; a) = I_k$, then push $I_k$ to the stack (shift).

Otherwise, if $I_j$ is an accepting state annotated with the production $A \rightarrow \gamma$, then remove the top of the stack (reduce).

NotethatthisparsecannotbecompletedbyapureLR(0) parsing.

Example of a Parse

 чувствует, что он не может быть восстановлен из исходного стека.

LR(0) Parsing Using Classifier

Notethatthesymbolsoftheoriginalstackcanbereconstructedfromthestatestack.
For every non-terminal $A$ such that $\delta(I, I) = I$, set $\textsc{goto}[I, I] = \textsc{error}$.

1. All entries not set by the above rules are set to $\textsc{error}$.
2. If $I$ is an accepting state annotated by $S$, then set $S \leftarrow S$.
3. If $I$ is in an accepting state annotated by $S$, then set $S \neq V$.

For every terminal $a$, $\delta(I, a)$ is set to $I$.

If $I$ is in an accepting state annotated by $a$, then set $S \leftarrow S$.

If $I$ is in an accepting state annotated by $a$, then set $\textsc{error}$.

State $S$ corresponds to $I$.

Construct the $\textsc{lr}(0)$ classifier for the grammar.

### LR Parsing Tables

#### Converting SLR(0) Parsing Tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>SYMBOLS</th>
<th>STATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\textsc{accept}$</td>
<td>$A$</td>
<td>$A + A$</td>
</tr>
<tr>
<td>$\textsc{reduce}$ by $A$</td>
<td>$A + A$</td>
<td>$A + A$</td>
</tr>
<tr>
<td>$\textsc{reduce}$ by $a$</td>
<td>$A + A$</td>
<td>$A + A$</td>
</tr>
<tr>
<td>$\textsc{reduce}$ by $B$</td>
<td>$A + A$</td>
<td>$A + A$</td>
</tr>
</tbody>
</table>

The value of $\delta(I, a)$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

For each state $S$, the $\textsc{reduce}$ function takes an argument of state $S$. The value of $\delta(I, a)$ is a terminal $a$ or a terminal $a$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

The value of $\delta(I, a)$ is a terminal $a$ or a terminal $a$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.

$\textsc{reduce}$ can have one of four values: $\textsc{error}$, a terminal $a$, $\textsc{reduce}^*$, or $\textsc{reduce}^+$.
A grammar is SLR(0) if its LR(0) classifier satisfies the following requirements:

1. For every accepting state \( I \) annotated by production \( A \rightarrow \gamma \) and terminal \( a \) such that \((I, a)\) is defined,
   \[ \text{Follow}(A) \cap \text{Follow}(\gamma) \neq \emptyset \]

2. For every accepting state \( I \) annotated by production \( A \rightarrow \gamma \) and terminal \( a' \), there exists a production \( A \rightarrow \alpha \) such that
   \[ \text{Follow}(A) \cap \text{Follow}(\alpha) = \emptyset \]

3. The classifier has no shift-reduce conflicts, and

   \[ \text{Follow}(A) \cap \text{Follow}(\gamma) = \emptyset \]

Examples of YACC: A Calculator

```yacc
%{
#include "ctype.h"
}

%token DIGIT

%%
line: | expr'
     { printf("%d\n", $1); } |
     { printf("%g\n", $2); } |
l
expr:  | expr' + expr    { $1 + $3; } |
     | expr' - expr    { $1 - $3; } |
     | expr' * expr    { $1 * $3; } |
     | expr' / expr    { $1 / $3; } |
     | '(' expr ')'    { $2; } |
     | $1; |

UMINUS | term
       { $2 - $3; } |
       { $2 + $3; } |

factor: | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
```

Sample yylex

```c
#include "stdio.h"

%token NUMBER
%left '+', '-'
%left '*', '/'
%right UMINUS
%%
lines: | lines expr
      { printf("%g\n", $2); } |
      { printf("%g\n", $2); } |
      ...

expr:  | expr' + expr
      { $1 + $3; } |
     | expr' - expr
      { $1 - $3; } |
     | expr' * expr
      { $1 * $3; } |
     | expr' / expr
      { $1 / $3; } |
     | '(' expr ')' | $2; |
     | $1; |

UMINUS | term
       { $2 - $3; } |
       { $2 + $3; } |

factor: | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
       | factor'*' factor
         { $1 * $3; } |
```

Charaterization of SLR(0) Grammars
A terminal $a$ of $G$ and terminal $a'$ of $\text{LR}(1)$ have the form $\langle a \rangle$ for a non-terminal $G$.

In order to construct an $\text{LR}(1)$ classifier we generate

$$[a', a] \leftarrow [a, a'] \leftarrow \langle G \rangle$$

accepts the string $a'b'$ under the mode $a'$. Other classes are handled similarly following the analysis of sentential forms to full resolve conflicts such as the one detected above.

To resolve conflicts such as the one detected above,

**Moving to an $\text{LR}(1)$ Classifier**

This right-linear grammar produces the classifier presented in the next slide.

$$[a \rightarrow b] \leftarrow [b \rightarrow a]$$

Precedes

We now construct a right-linear grammar for all rightmost sentential forms.

Let us construct an $\text{LR}(0)$ classifier for this grammar. As a first step,

$$\langle a \rangle \leftarrow \langle b \rangle$$

Consider the following grammar:

$$\langle a \rangle \leftarrow \langle b \rangle$$

These are $\text{LR}$ grammars which are not $\text{SLR}(0)$. Consider the

An $\text{LR}(0)$ classifier for the problem grammar

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$

$\langle a \rangle \leftarrow \langle b \rangle$
Reconsider the previously considered problem grammar:

```
S → aAd
A → c
B → c
```

Constructing the induced grammar $G_{LR(1)}$, we obtain:

```
S → aAd
A → c
B → c
```

The Resulting $LR(1)$ Classifier

The $LR(1)$ classifier for the grammar:

```
S → aAd
A → c
B → c
```

From $LR(1)$ to $LALR$

Consider the following grammar:

```
S → S
C → cC
```

Constructing the induced grammar $G_{LR(1)}$, we obtain:

```
S → S
C → cC
```

For every terminal $a$, if $(I^a, I^a) = I^j$, then set $ACTION[i; a] = \text{reduce } A$. Here $A = S, C$.

For every non-terminal $A$ such that $(I^a, I^a) = I^j$, set $ACTION[i; a] = \text{shift } j$.

If any conflicting actions result from the above rules, then the grammar is not $LR(1)$.

All entries not set by the above rules are set to $\text{error}$.
Note that we can merge together states $I_4$ with $I_7$ and state $I_8$ with $I_9$ without creating conflicts (but giving up some error-detection options). This leads to an LALR classifier.