Bottom-Up Parsing

- Technique more general than LL(1).
- It accepts when all input has been consumed and the stack contains $S$.
- Automaton starts with an empty stack.
- Automaton performs two actions:
  - Shift: push next input symbol to stack.
  - Reduce: reduces stack from $\alpha\beta$ to $\alpha$ if there exists a rightmost derivation of the form $S \xrightarrow{*} \alpha x y \leftarrow \# y \alpha \beta$.

Bottom-Up Parsing
(reduce string on top of stack by inverting a production)

The operations here are **shift** (copy from input to stack) and **reduce**

<table>
<thead>
<tr>
<th>Accept</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(S)$</td>
<td>$S$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)S$</td>
<td>$S$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>$(S)$</td>
</tr>
<tr>
<td>$(S)$</td>
<td>$(S)$</td>
</tr>
</tbody>
</table>

A bottom-up recognition by a **shift-reduce PDA** is given by:

$$((())) \leftarrow ((()S) \leftarrow (SS) \leftarrow (S) \leftarrow S$$

Reversing a rightmost derivation:

**Bottom-up Parsing by a Shift-Reduce PDA**
A sentential form is a mixed (terminals and non-terminals) string that may appear in a rightmost derivation. A (right) sentential form is a mixed (terminals and non-terminals) string that may appear in a rightmost derivation.

In an unambiguous grammar, the handle is always uniquely defined.

On the other hand, for an ambiguous grammar such as

\[ E \rightarrow E + E \mid E \times E \mid E \rightarrow id \]

we may encounter sentential forms with two handles, e.g.

\[ E \rightarrow E + E \]

A production \( A \rightarrow \cdot \) is a handle in a sentential form if it is a rightmost derivation and there exist a rightmost derivation

\[ m \cdot A \leftarrow m \cdot \alpha \leftarrow S \]

A handle \( \cdot \) may appear in a rightmost derivation.

Handles
A grammar is unambiguous if the handle of every sentential form can be uniquely determined by a prefix which extends at most \( k \) characters beyond the handle. If there are two rightmost derivations such that the first string is a prefix of the second, then the first string belongs to the language of the grammar.

This means that if there are two rightmost derivations

\[
\begin{align*}
  \text{string 1} & \Rightarrow \text{string 2} \\
  \text{string 3} & \Rightarrow \text{string 4}
\end{align*}
\]

and \( \text{string 1} = \text{string 2} \), then \( \text{string 1} = \text{string 3} \).

An unambiguous grammar is called an LR(\( k \)) grammar.
A. Pnueli

Lecture 4: Bottom-Up Parsing

Properties of LR(0) Grammars

There exist unambiguous grammars which are not LR(0).

For example, the grammar:

\[ a \rightarrow b \left\{ \begin{array}{l} a \rightarrow b \mid bDbb \end{array} \right. \]

On the other hand, the language \( \{ a \in \{ a, b \}^* \mid |a| > 0 \} \) has an LR(0) grammar:

\[ \{ 0 \geq ? \mid ?q \in \{ a, b \} \} \cap \{ 0 \geq ? \mid ?q \in \{ a, b \} \} \]

which generates the language:

\[ \{ a \rightarrow b \mid bDbb \} \cap \{ a \rightarrow b \mid bDb \} \]

Hence, for any \( k \geq 0 \), this is because the prefix:

\[ \{ 0 < ? \mid p \in \{ a, b \} \} \cap \{ 0 < ? \mid p \in \{ a, b \} \} \]

is not LR(k).

Honors Compilers, NYU, Fall, 2009
Implementation of Shift-Reduce Parsing

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

If the stack does not contain a handle at its top, but is still a prefix of a right sentential form – *shift* the next input token to the stack.

Otherwise, *reduce*.

If the stack contains a handle at its top, then accept.

If the stack contains $S$ and the input is empty, then accept.

Otherwise, reject.

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

Otherwise, reject.
Recognizing Reducible Strings

The main problem in the implementation of a shift-reduce parser is to determine whether the current stack is a reducible string as a prefix of a prefix of the entire stack. In an LR parsing, we expect all stack contents to be viable prefixes.

We refer to a prefix of a reducible string as a reducible prefix. In an LR parsing, we have been an infinite table. Fortunately, we have the following reduction:

\[ S \rightarrow \alpha \xi \beta, \text{ where } \alpha \xi \beta \text{ is reducible.} \]

Claim 7. (Regularity of Reducible Strings)

The set of reducible strings is a regular language. Thus, we should have a table of all possible stack contents and the entire stack.

In this determination, we are allowed to consult the next input character, and to which non-terminal \( A \) should be reduced.

For each rightmost derivation, we refer to a reducible string as a reducible prefix.

\[ \alpha \xi \beta \leftrightarrow \gamma \alpha \xi \beta \leftrightarrow \gamma A \alpha \xi \beta \]

A. Pnueli

Honors Compilers, NYU. Fall, 2009
We will show how to construct a finite-state classifier for the set of reducible strings of an LR(0) grammar.

A Reducible Strings Classifier

This is a DFA with several accepting states. Each accepting state identifies the production which has been applied last when generating the reducible string by a rightmost derivation.

A.Pnueli
AReducibleStringsClassifier

Honors Compilers, NYU, Fall, 2009
Out of this grammar we construct an NFA and then determinize it.

\[
\begin{align*}
[p \leftarrow E] & \mid \quad [(E) \leftarrow A] (E) & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
[\langle F \rangle] & \mid \quad \langle E \rangle & \leftarrow \langle F \rangle \\
\end{align*}
\]

We construct the following right linear grammar:

\[
\begin{align*}
p & \mid \quad (E) & \leftarrow & \langle F \rangle \\
A & \mid \quad A * L & \leftarrow & \langle L \rangle \\
L & \mid \quad L + E & \leftarrow & \langle E \rangle \\
E & \leftarrow & \langle E \rangle \\
\end{align*}
\]
\[
[u_X \cdots I_X \leftarrow X]^{u_X \cdots I_X} \leftarrow \langle X \rangle
\]

Construct also the rule

\[\langle X \rangle \leftarrow \langle X \rangle\]

Eliminate any productions of the form

\[\langle ? X \rangle^{u_{X}} \cdots I_X \leftarrow \langle X \rangle\]

Linear rule

For each non terminal \(X \in \{1, \ldots, n\}^u\), construct a right-

\[u_X \cdots I_X \leftarrow X\]

The General Construction
The Classiﬁer for LR(0) Parsing

A. Pnueli

Honors Compilers, NYU, Fall, 2009
Initially, stack is empty.

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

otherwise, shift input character into stack.

Otherwise, run the classifier on the stack contents:

- If the classifier reaches an accepting state annotated with \( \mathcal{A} \), then reduce.
- If the classifier gets stuck before scanning the entire stack, then reject.
- If the stack does not contain a viable prefix, then reject.

Otherwise, shift input character into stack.

LR(0) Parsing Using Classifier

A. Pnueli

Honors Compilers, NYU, Fall, 2009
This parse cannot be completed by pure LR(0) parsing. To resolve shift-reduce conflicts, examine whether the next incoming character can continue the classification.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Action</th>
<th>Input →</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>accept</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>reduce by $F$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon + \epsilon$</td>
<td>reduce by $F$</td>
<td>$\epsilon + \epsilon$</td>
</tr>
<tr>
<td>$\epsilon * \epsilon$</td>
<td>reduce by $F$</td>
<td>$\epsilon * \epsilon$</td>
</tr>
<tr>
<td>$\epsilon ? \epsilon$</td>
<td>reduce by $F$</td>
<td>$\epsilon ? \epsilon$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>shift</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon * \epsilon$</td>
<td>shift</td>
<td>$\epsilon * \epsilon$</td>
</tr>
<tr>
<td>$\epsilon ? \epsilon$</td>
<td>shift</td>
<td>$\epsilon ? \epsilon$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>reduce by $T$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon + \epsilon$</td>
<td>reduce by $T$</td>
<td>$\epsilon + \epsilon$</td>
</tr>
<tr>
<td>$\epsilon * \epsilon$</td>
<td>reduce by $T$</td>
<td>$\epsilon * \epsilon$</td>
</tr>
<tr>
<td>$\epsilon ? \epsilon$</td>
<td>reduce by $T$</td>
<td>$\epsilon ? \epsilon$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>shift</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon * \epsilon$</td>
<td>shift</td>
<td>$\epsilon * \epsilon$</td>
</tr>
<tr>
<td>$\epsilon ? \epsilon$</td>
<td>shift</td>
<td>$\epsilon ? \epsilon$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>reduce by $E$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$\epsilon + \epsilon$</td>
<td>reduce by $E$</td>
<td>$\epsilon + \epsilon$</td>
</tr>
<tr>
<td>$\epsilon * \epsilon$</td>
<td>reduce by $E$</td>
<td>$\epsilon * \epsilon$</td>
</tr>
<tr>
<td>$\epsilon ? \epsilon$</td>
<td>reduce by $E$</td>
<td>$\epsilon ? \epsilon$</td>
</tr>
</tbody>
</table>

This is an example of a parse.
Initially, stack is empty.

- Otherwise, shift input character into stack.

- If classifier gets stuck, then reject.

- If classifier reaches an accepting state $I_j$ annotated with $\epsilon 
\in \text{Follow}(A)$, then reduce.

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

Run the classifier on the stack contents:

- If the classifier reaches an accepting state.

- Otherwise, shift input character into stack.

- If input is empty, accept.

- At any step, if stack contains $\text{F}'$ (start symbol) and

  - Initially, stack is empty.

  - Otherwise, shift input using LR(0) classifier.

  - SLR(0) Parsing Using LR(0) Classifier

  - A. Pnueli
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, and \( I_j \) be the (classifier) state at the top of the stack.
  - If \( \delta(I_j, a) = I_k \), then push \( I_k \) to the stack (shift).
  - Otherwise, if \( I_j \) is an accepting state annotated with the production \( A \rightarrow \beta \), then remove the top \( |\beta| \) states from the top of the stack (reduce). If \( I_k \) is the state exposed by this removal, then push \( \delta(I_k, A) \) to the stack.

Note that the symbols of the original stack can be reconstructed from the state stack.
### Lecture 4: Bottom-Up Parsing

**APnPueli**

**AParse with a State Stack**

<table>
<thead>
<tr>
<th>Symbols</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>*</td>
<td>8</td>
</tr>
<tr>
<td>*</td>
<td>9</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
</tr>
</tbody>
</table>

**A Parse with a State Stack**

- **Action**
  - Shift
  - Reduce by
  - Accept

- **Input** →
- **Symbols** ←
- **States** ←
These consist of two parts: a parsing-action function \( \text{ACTION} \) and a goto function \( \text{GOTO} \). A. Pnueli

\[
\forall \xi, I \in \Sigma^* \ (\exists \xi', I') \ (\text{ACTION}[\xi, I]) = [\xi', I']
\]

For a non-terminal \( A \).

\( \text{GOTO} \) function.

- Error — The parser discovers an error in the input.
- Accept — The parser accepts the input and stops.

The stack.

- Reduce \( A \) \( \rightarrow \gamma \), written \( Rk \), where \( k \) is the number of a production.
- Shift \( I \), written \( Sj \), written \( Sj \), consume input and push \( I \) to the stack.

Honors Compilers, NYU, Fall, 2009
<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>GOTO</td>
</tr>
</tbody>
</table>

LR Table for Arithmetical Expressions

A. Pnueli

Honors Compilers, NYU, Fall 2009

Lecture 4: Bottom-Up Parsing
Construct the LR(0) classifier for the grammar.

State $i$ corresponds to $I_i$. The parsing actions are determined as follows:

- For every terminal $a$, if $\delta(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to shift $j$.
- If $I_i$ is a terminal $a$, then set $\text{ACTION}[i, a]$ to reduce $A \rightarrow \beta$, where $\beta$ is a sequence of terminals in $\text{Follow}(A)$. Here $A \neq S'$.
- If $I_i$ is an accepting state annotated by $S_0 \rightarrow$, then set $\text{ACTION}[i, \$]$ to accept.
- All entries not set by the above rules are set to error.

For every non-terminal $A$ such that $\delta(I_i, A) = I_j$, set $GOTO[i, A]$ to $j$. 

Honors Compilers, NYU, Fall, 2009
Characterization of SLR(0) Grammars

A grammar is SLR(0) if it is LR(0) classification satisfies the following requirements:

For every accepting state \( I \) annotated by production \( \gamma \), and terminal \( a \) such that \( \gamma \) is defined, \( (\gamma, I)(a) \) is defined, \( a \in \text{Follow}(\gamma) \) and \( \text{Follow}(\gamma) \cap \text{Follow}(\gamma') = \emptyset \), then \( \text{Follow}(\gamma) \neq \emptyset \).

That is, the classifier has no shift-reduce conflict and all reduce-reduce conflicts are resolved by the function \( \text{Follow} \).

\[ \emptyset = (\text{Follow}(A_1 \cup \text{Follow}(A_2)) = \emptyset, \text{Follow}(A_1) \neq \text{Follow}(A_2) \]
Examples of YACC: A Calculator

```
#include "ctype.h"

%%
 line
 expr
 term
 factor

%%

expr : \n | expr \+ term
     | term  /* factor */ DIGIT

term  : term \* factor
     | factor

factor : ( expr )
    | DIGIT

%%

{ printf("%d\n", $1); }

{$$ = $1 + $3; }

{$$ = $1 * $3; }

{$$ = $2; }
```

Honors Compilers, NYU, Fall, 2009
Sample yylex

```c
int c;
    c = getchar();
    if (isdigit(c)) {
        yyval = c - '0';
        return DIGIT;
    }
    return c;
```

Honors Compilers, NYU, Fall, 2009
%%

```c
#include <stdio.h>

%token NUMBER

%%

lines: |
  \n  printf("\n", $2);
| lines

expr: expr'+'expr
    | expr'-'expr
    | expr'*'expr
    | expr'/'expr
    | '栝(expr)
    | UMINUS expr
    | NUMBER;

expr'+'expr
  | expr'-'expr
  | expr'*'expr
  | expr'/'expr
  | '栝(expr)
  | UMINUS expr
  | NUMBER;
  
%%

Use of Ambiguous Grammars

A. Pnueli

Lecture 4: Bottom-Up Parsing

Honors Compilers, NYU, Fall, 2009
```
When SLR(0) Parsing is Inadequate

There are LR grammars which are not SLR(0). Consider the following grammar:

\[ S \rightarrow aA \mid bB \]
\[ A \rightarrow c \]
\[ B \rightarrow c \]

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

Let us construct an LR(0) classifier for this grammar. As a first step, we construct a right-linear grammar for all rightmost sentential forms.

The right-linear grammar produces:

\[ [\epsilon \rightarrow B]c \leftarrow \langle B \rangle \]
\[ [\epsilon \rightarrow A]c \leftarrow \langle A \rangle \]
\[ [\epsilon \rightarrow S] \rightarrow \langle A \rangle \]
\[ [pB] \leftarrow \langle B \rangle q \]
\[ [pA] \leftarrow \langle A \rangle q \]
\[ [qB] \leftarrow \langle B \rangle q \]
\[ [qA] \leftarrow \langle A \rangle q \]
\[ [qS] \leftarrow \langle S \rangle \]

Honors Compilers, NYU, Fall 2009
Note that we have a conflict at state $I^2$ which cannot be resolved by the $\text{Follow}$ function since $\text{Follow}(A) = \text{Follow}(B) = \{\epsilon, d, e, q\}$. The following set $\{\epsilon, \epsilon\}$ is defined:

$$\{\epsilon, \epsilon\} = (\text{Follow} AB = \text{Follow} BA = \{\epsilon\})$$
To resolve conflicts such as the one detected above, we can extend the analysis of sentential forms to full consideration of the terminal immediately following the non-terminal. The non-terminals of the induced right-linear grammar \( G_{LR(1)} \) have the form \( A \rightarrow a \) for a non-terminal \( A \) and terminal \( a \). In order to construct an \( LR(1) \) classifier we generate again an induced right-linear grammar \( G_{LR(1)} \). The non-terminals of the induced right-linear grammar \( G_{LR(1)} \) have the form \( h \rightarrow Aa \) for a non-terminal \( A \) and terminal \( a \) of \( G \). We define a \( LR(1) \) classifier to be a DFA which accepts the string \( \alpha \) under the mode \( \langle A, \beta, \epsilon \rangle \) if \( \alpha \) is a rightmost derivation of \( \beta \).

\[
\begin{align*}
\alpha & \leftarrow \epsilon \\
\text{for every rightmost derivation,} & \\
\text{we consider the terminal imme} & \\
\text{diately following the } & \\
\text{non-terminal.}
\end{align*}
\]

\[ S \]

Moving to an \( LR(1) \) classifier.
Generating the Right-Linear Grammar $G_{LR(1)}$

Assume a context-free grammar $G$. The right-linear grammar $G_{LR(1)}$ is generated as follows:

1. The start symbol of $G$ is $S$.
2. For every production $A \rightarrow B_1 \cdots B_k$, add to $G_{LR(1)}$ the productions:
   - $A \rightarrow B_1 \cdots B_k a$.
   - If $B_i$ is non-terminal, add to $G_{LR(1)}$ the production $A \rightarrow B_1 \cdots B_j a$.
3. For every terminal $a$ and every terminal $b$ such that $b \in \text{First}(B_i + 1)$, add to $G_{LR(1)}$ the production $A \rightarrow B_1 \cdots B_i \cdots b$.
4. If $B_k$ is non-terminal, add to $G_{LR(1)}$ the production $A \rightarrow B_1 \cdots B_k a$.
5. The start symbol of $G_{LR(1)}$ is $S$, where $S$ is the start symbol of $G$. The right-linear grammar $G_{LR(1)}$ is generated as follows:
Reconsider the previously considered problem grammar:

\[
\begin{align*}
S & \rightarrow \text{aAd} \\
& \rightarrow \text{aBe} \\
& \rightarrow \text{bBd} \\
& \rightarrow \text{bAe} \\
\end{align*}
\]

Constructing the induced grammar, we obtain:

\[
\begin{align*}
\langle A \rangle q & \mid \langle B \rangle a \\
\langle B \rangle q & \mid \langle A \rangle a \\
\langle B \rangle q & \mid \langle A \rangle a \\
\langle B \rangle q & \mid \langle A \rangle a \\
\end{align*}
\]

Applying to the Problem Grammar:

\[
\begin{align*}
\text{c} & \leftarrow \text{B} \\
\text{c} & \leftarrow \text{A} \\
\text{q} & \leftarrow \text{B} \text{e} \\
\text{q} & \leftarrow \text{B} \text{d} \\
\text{q} & \leftarrow \text{A} \text{d} \\
\text{q} & \leftarrow \text{A} \text{e} \\
\text{q} & \leftarrow \text{B} \text{d} \\
\text{q} & \leftarrow \text{A} \text{d} \\
\text{S} & \rightarrow \text{q} \\
\end{align*}
\]
The Resulting LR(1) Classifier
Constructing LR(1) Parsing Tables

For every non-terminal \( A \), set \( GOTO(i,A) = \{ j \mid \forall a, (i,a) \in \delta \} \)

All entries not set by the above rules are set to error.

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

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

For every terminal \( a \), if \( (i,a) \in \delta \), then set 

\[ ACTION[i,a] = \text{shift} \]

If \( i \) is an accepting state annotated by \( A \), then set 

\[ ACTION[i,a] = \text{reduce} A \]

If \( i \) is an accepting state annotated by \( S \), then set 

\[ ACTION[i,a] = \text{accept} \]

For every terminal \( a \), if \( (i,a) \in \delta \) and \( \exists j \), then set 

\[ ACTION[i,a] = (\exists j) \]

State \( i \) corresponds to \( \text{LR(1)} \). The parsing actions are determined as follows:

Construct the \( \text{LR(1)} \) classiﬁer for the grammar.
Consider the following grammar

\[
S \rightarrow SC \\
S \rightarrow C \\
C \rightarrow cC \\
C \rightarrow \epsilon
\]

Converting the induced grammar \(\text{GLR}^{(1)}\), we obtain:

\[
[S \leftarrow S] S \rightarrow \epsilon \\
[S \leftarrow S] S \\
[S \leftarrow S] S \rightarrow S
\]

Consider the following grammar

\[
P \rightarrow S \\
P \rightarrow \epsilon
\]
LR classifer: This leads to an LALR classifer. Note that we can merge together states without creating conflicts (but giving up some error-detection options).
The LALR(1) Classifier

A. Pnueli