Bottom-Up Parsing
Bottom-Up Parsing

A. Pnueli

Technique more general than LL(k).

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 $A$ to $A'$ if there exists a rightmost derivation of the form $S \leftarrow \cdots \leftarrow \alpha \gamma \cdots \leftarrow \beta \xi$, where $\gamma$ is a regular string.

From left to right: Synthesize tree from fragments, reconstruct rightmost derivation from left to right.

Lecture 4: Bottom-Up Parsing

Honors Compilers, NYU, Spring, 2007
The operations here are shift (copy from input to stack) and reduce (invert a production) as a bottom-up recognition by a shift-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>shift</td>
</tr>
<tr>
<td>(SS)</td>
<td>(S)</td>
<td>shift</td>
</tr>
<tr>
<td>(S)</td>
<td>()</td>
<td>reduce</td>
</tr>
<tr>
<td>()</td>
<td>(S)</td>
<td>shift</td>
</tr>
<tr>
<td>(SS)</td>
<td>(S)</td>
<td>shift</td>
</tr>
<tr>
<td>(S)</td>
<td>()</td>
<td>reduce</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:
A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.

A production \( A \rightarrow \varepsilon \) is a handle in a (right) sentential form if it is a rightmost derivation.
A Pnueli

An unambiguous grammar is called an LR(k) grammar.

\[ LR(k) \] Grammars

This means that if there are two rightmost derivations such that their first string is a prefix of the second, then the first string is a prefix which extends at most \( k \) characters beyond the handle.

\[ P_1 = P_2, A_1 = A_2, x_1 = x_2, \text{ and } \forall A \in \Gamma, \text{ then } x_1 = x_2, \text{ and } \forall A \in \Gamma, x_1 = x_2, \text{ and } \forall A \in \Gamma, x_1 = x_2. \]
There exist unambiguous grammars which are not LR(\(k\)). For example, the grammar

\[
\{0 < ? | p \gamma q \gamma p \} \cap \{0 < ? | p \gamma q \gamma p \}
\]

which generates the language

\[
\text{\(D\)} \rightarrow \text{\(D\)} \text{\(D\)} \\
\text{\(C\)} \rightarrow \text{\(D\)} \\
\text{\(S\)} \rightarrow \text{\(C\)}
\]

has an LR(0) grammar.

On the other hand, the language does not uniquely determine whether the handle is \(C\) or \(D\).

This is because the prefix

\[
\{0 < ? | p \gamma q \gamma p \} \cap \{0 < ? | p \gamma q \gamma p \}
\]

is not LR(\(k\)) for any \(k \geq 0\).
We use a stack to hold the prefix of the analyzed sentential form up to (and including) the handle. Initially, the stack is empty.

Lecture 4: Bottom-Up Parsing

Implementation of Shift-Reduce Parsing

If the stack contains a handle at its top, reduce the handle to a sentential form — shift the next input token to the stack top.

If the stack does not contain a handle at its top, but is still a prefix of a sentential form, then accept.

Otherwise, reject.

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

Otherwise, require looking at the first $k$ input characters. For an LR$(k)$ grammar, this may require looking at the first $k$ input characters of a sentential form. For an LR(0) grammar, this may require looking at the first input character.

It follows from the properties of rightmost derivations that we never have to look for handles deeper in the stack.
LR parsing, we expect all stack contents to be viable prefixes. We refer to a prefix of a reducible string as a viable prefix. In an LR parser, we have been an infinite table. Fortunately, we have the following.

The set of reducible strings $a^r$ is a regular language.

**Claim 7.** (Regularity of Reducible Strings)

The set of reducible strings is a regular language.

Lecture 4: Bottom-up Parsing

A. Pnueli

Honors Compilers, NYU, Spring, 2007

For each rightmost derivation $S^*$ we refer to $a^r$ as a reducible string.

$A \lor y \Rightarrow A \lor y \Rightarrow S$
A Reducible Strings Classifier

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

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

A.Pnueli

AReducibleStringsClassier
Out of this grammar we construct an NFA and then determinize it:

\[
\begin{align*}
[p \leftarrow \mathcal{A}] p & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] p & \quad | \\
[p \leftarrow \mathcal{A}] (\mathcal{A}) & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] (\mathcal{A}) & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] E & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] E & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] F & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] F & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] T & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] T & \quad | \\
[p \leftarrow \mathcal{A}] (\mathcal{A}) & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] (\mathcal{A}) & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] E & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] E & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] F & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] F & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] T & \quad | \\
[\mathcal{A} \leftarrow \mathcal{A}] T & \quad | \\
\end{align*}
\]

Corresponding to the grammar we construct the following right linear grammar:

\[
\begin{align*}
p & \quad | \\
p & \quad | \\
p & \quad | \\
p & \quad | \\
p & \quad | \\
p & \quad | \\
E & \quad | \\
E & \quad | \\
T & \quad | \\
T & \quad | \\
\end{align*}
\]
The General Construction

For each production
\[ \text{For each production} \quad uX \cdots \bar{Y} \leftarrow X \]
\[ uX \cdots \bar{Y} \leftarrow \langle X \rangle \]

For each non terminal \( X \) with \( \bar{Y} \), construct a right-linear rule:
\[ \langle X \rangle \leftarrow \langle X \rangle \]

Eliminate any productions of the form of the form
\[ \langle ?X \rangle \text{?} \bar{Y} \cdots \bar{Y} \leftarrow \langle X \rangle \]

Construct also the rule:
\[ \langle X \rangle \leftarrow \langle X \rangle \]

The General Construction
The Classiﬁer for LR(0) Parsing
Initially, stack is empty.

At any step, if stack contains \( \text{(start symbol)} \) and \( \notin \text{stack contains} \), shift input character into stack.

Otherwise, if the classiﬁer reaches an accepting state annotated with \( \exists \), then reduce with \( A \leftarrow \emptyset \), then reduce.

Otherwise, run the classiﬁer on the stack contents:

- Input is empty, accept.
- If the classiﬁer reaches an accepting state annotated \( \exists \), (start symbol) and

Initially, stack is empty.

LR(0) Parsing Using Classiﬁer
Note that this parse cannot be completed by a pure LR(0) parsing.

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id_1 + id_2$</td>
<td>shift</td>
<td>$id_1$</td>
</tr>
<tr>
<td>$id_3$</td>
<td>reduce by $E$</td>
<td>$E$</td>
</tr>
<tr>
<td>$id_1$</td>
<td>reduce by $id_3$</td>
<td>$id_2 + id_3$</td>
</tr>
<tr>
<td>$id_3$</td>
<td>shift</td>
<td>$id_2$</td>
</tr>
<tr>
<td>$id_1$</td>
<td>reduce by $id_3$</td>
<td>$id_0$</td>
</tr>
<tr>
<td>$id_1$</td>
<td>shift</td>
<td>$id_0$</td>
</tr>
</tbody>
</table>

Example of a Parse
SLR(0) Parsing Using LR(0) Classifier

Initially, stack is empty.

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

Otherwise, let $\alpha$ be the next incoming input character.

Run the classifier on the stack contents:

- If the classifier reaches an accepting state $I_j$, such that $I_j$ has no $\alpha$-annotated successor and $\alpha \in \text{Follow}(A)$, then reduce.

- Otherwise, shift input character into stack.
Replace Symbol Stack by State Stack.

Note that the symbols of the original stack can be reconstructed from the state stack.

Initially, stack contains state $S_0$.

At any step, if stack contains state $S_0$ and input is empty, accept.

Initially, stack contains state $S_0$.

At any step, if stack contains state $S_0$ and input is empty, accept.

Parse the input:

- If $S_j$ is an accepting state annotated with the production $A \rightarrow \gamma$, then remove the top $|\epsilon|$ states from the top of the stack.
  - If $S_j$ is an accepting state annotated with the production $A \rightarrow \gamma$, then remove the top $|\epsilon|$ states from the top of the stack.
  - If $S_j$ is an accepting state annotated with the production $A \rightarrow \gamma$, then remove the top $|\epsilon|$ states from the top of the stack.
  - If $S_j$ is an accepting state annotated with the production $A \rightarrow \gamma$, then remove the top $|\epsilon|$ states from the top of the stack.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Otherwise, let $a$ be the next incoming input character, and be $S_j$.
A. Pnueli

A Parse with a State Stack

<table>
<thead>
<tr>
<th>States</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0I</td>
<td>F</td>
<td>id1</td>
<td>+</td>
</tr>
<tr>
<td>1I</td>
<td>E</td>
<td>id2</td>
<td>+</td>
</tr>
<tr>
<td>2I</td>
<td>T</td>
<td>id3</td>
<td>+</td>
</tr>
<tr>
<td>3I</td>
<td>E</td>
<td>id1</td>
<td>+</td>
</tr>
<tr>
<td>4I</td>
<td>T</td>
<td>id2</td>
<td>+</td>
</tr>
<tr>
<td>5I</td>
<td>F</td>
<td>id3</td>
<td>+</td>
</tr>
<tr>
<td>6I</td>
<td>T</td>
<td>id1</td>
<td>+</td>
</tr>
<tr>
<td>7I</td>
<td>T</td>
<td>id2</td>
<td>+</td>
</tr>
<tr>
<td>8I</td>
<td>T</td>
<td>id3</td>
<td>+</td>
</tr>
</tbody>
</table>

A Parse with a State Stack

Lecture 4: Bottom-Up Parsing

Honors Compilers, NYU, Spring, 2007
These consist of two parts: a parsing-action function \( \text{ACTION} \) and a goto function \( \text{GOTO} \). For a non-terminal \( A \),

\[
\text{ACTION} \quad \text{GOTO} \quad [A,?I] \phi = [A,?I] \theta
\]

Error — The parser discovers an error in the input.

Accept — The parser accepts the input and stops.

Reduce — The stack.

Shift — A non-terminal \( A \) is the number of a production.

Consume input and push \( I \) to the stack.

\( \phi \leftarrow A \)

\( \text{ACTION} \quad \text{GOTO} \quad [A,?I] \phi = [A,?I] \theta \)

\( \text{ACTION} \quad \text{GOTO} \quad [A,?a] \phi = [A,?a] \theta \)

The value of \( \phi \) can have one of four values:

- \( \text{Shift} \) — written \( \downarrow I \)
- \( \text{Reduce} \) — written \( \phi \leftarrow A \)
- \( \text{Accept} \)
- \( \text{Error} \)

### LR Parsing Tables

- A. Pnueli

Honors Compilers, NYU, Spring, 2007

125
<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td></td>
</tr>
</tbody>
</table>

LR Table for Arithmetical Expressions

A. Pnueli
For every non-terminal $A$ such that $I = (A, I)$, set $\text{GOTO}\ [A, I]$ to $\forall A, I = (A, I)$ such that $\forall A, I = (A, I)$.

For every terminal $a$, if $I$ is an accepting state annotated by $A$, set $\text{ACTION}\ [I, a] = \text{reduce } A$.

If $I$ is an accepting state annotated by $S$, set $\text{ACTION}\ [I, ] = \text{accept}$.

If $I$ is an accepting state annotated by $S = S_0$, then set $S \leftarrow S_0$.

If $I$ is an accepting state annotated by $A!$, for each $a \in \text{Follow}(A)$, set $B \leftarrow A$.

If $I$ is an accepting state annotated by $A$, for each terminal $a$, if $I = (a, I)$, then set $\text{ACTION}\ [I, a] = \text{shift}$.

For every non-terminal $A$, if $I = (A, I)$, then set $\text{ACTION}\ [I, a] = (A, I)$ such that $\forall A, I = (A, I)$.

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

Construct the $LR(0)$ classifier for the grammar.

Constructing $SLR(0)$ Parsing Tables

A. Pnueli

Honors Compilers, NYU, Spring, 2007
A grammar is SLR(0) if its LR(0) classifier satisfies the following requirements:

- For every accepting state \( I \) annotated by production \( \alpha \) and terminal \( a \) such that \( I \) is defined, \( (a, I) \) is not an element of \( \text{Follow}(\alpha) \).
- \( \text{Follow}(\alpha_1) \cap \text{Follow}(\alpha_2) = \emptyset \) and \( \alpha_1 \neq \alpha_2 \), then \( \text{Follow}(\alpha_1) \neq \text{Follow}(\alpha_2) \).

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

#include <ctype.h>

%%

DIGIT: \d{2};

expression: expression + term;

factor: (expression);

term: factor * factor;

expression: DIGIT

%%

Honors Compilers, NYU, Spring, 2007
```c
{ 
    return c;
    
    { 
        return DIGIT;
        yyval = c - '0';
    } if (isdigit(c))
    c = getchar();
    int c;
    } yylex()

Sample yylex
```
Use of Ambiguous Grammars

A. Pnueli

Honors Compilers, NYU, Spring, 2007

#include<stdio.h>

%token NUMBER

%left '+', '-'
%left '*', '/'
%right 'UMINUS'

lines:linesexpr'

\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$
\$

print\\n\%, "", ', $2);':

expr:expr'+'expr

f $$=$1+$3;

 expr'-'expr

f $$=$1

expr'*'expr

f $$=$1

expr'/'expr

f $$=$1

'('expr')'

f $$=$2;

UMINUSexpr

f $$=$=

NUMBER;

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

\[
S \rightarrow aAd \\
A \rightarrow c \\
B \rightarrow c \\
\]

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

\[
\begin{align*}
&[\text{← } B] \ c \ \leftarrow \ \langle B \rangle \\
&[\text{← } A] \ c \ \leftarrow \ \langle A \rangle \\
&[\text{← } aAb] \ a \ \leftarrow \ \langle S \rangle \\
&[\text{← } aAb] \ Bb \ \leftarrow \ \langle S \rangle \\
&[\text{← } aAb] \ Bd \ \leftarrow \ \langle S \rangle \\
&\langle B \rangle a \ | \ \langle B \rangle q \\
\end{align*}
\]

This right-linear grammar produces the classifier for this grammar. As a first step, we construct a right-linear classifier for all rightmost sentential forms. Let us construct an LR(0) classifier for this grammar.
Note that we have a conflict at state $I_2$ which cannot be resolved by the $\text{Follow}$ function since 

$$\{ \epsilon, p \} = (B) \text{Follow} = (A) \text{Follow} = \{ f, d, e, q \}$$
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 at hand. We define a LR(1) classifter to be a DFA which accepts the string \( a \) under the mode \( \langle a \rangle \) for a non-terminal \( a \). To construct an LR(1) classifter we generate again an induced right-linear grammar \( G_{LR}(1) \). The non-terminals of \( G_{LR}(1) \) have the form \( \langle a \rangle G \) for a non-terminal \( G \) and terminal \( a \) of \( G \). The LR(1) classifter we generate in order to construct an LR(1) classifter we generate again an induced right-linear grammar again an induced right-linear grammar.
Generating the Right-Linear Grammar

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

1. The start symbol of $G_{LR(1)}$ is generated as follows:
   
   - If $B \notin \text{non-terminal}$, add to $G_{LR(1)}$ the production $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow \langle \alpha \rangle$

   - If $B \in \text{non-terminal}$, add to $G_{LR(1)}$ the production $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow \langle \alpha \rangle$

2. Assume a context-free grammar $G$. The right-linear grammar $G_{LR(1)}$ is generated as follows:
   
   - The start symbol of $G_{LR(1)}$ is $S$, where $S^\text{\$}$ is the start symbol of $G$. For every production
     
     $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A$

     add to $G_{LR(1)}$ the productions:

     1. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

     2. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

     3. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

     4. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

     5. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

     6. $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$

3. For every production $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$, add to $G_{LR(1)}$ the following productions:

   - For every non-terminal $B_i$ and terminal $b$, such that $B_i$ is a non-terminal and $b \in \text{FIRST}(B_i + 1)$, add to $G_{LR(1)}$ the production $\langle \alpha \rangle B_1 \ldots B_i b \leftarrow \langle \alpha \rangle$

   - For every production $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$, such that $B_k$ has already been added to $G_{LR(1)}$, add to $G_{LR(1)}$ the production $\langle \alpha \rangle B_1 \ldots B_\ell \leftarrow A \langle \alpha \rangle$.
Applying to the Problem Grammar

Reconsider the previously considered problem grammar:

\[
S \rightarrow \alpha \beta \gamma
\]

\[
A \rightarrow \alpha \beta \gamma
\]

\[
B \rightarrow \alpha \beta \gamma
\]

\[
G_{\text{LR}}(1)
\]

We obtain:

\[
[S] \rightarrow \alpha \beta \gamma
\]

\[
[A] \rightarrow \alpha \beta \gamma
\]

\[
[B] \rightarrow \alpha \beta \gamma
\]

\[
\langle \alpha \beta \rangle q \mid \langle \alpha \beta \rangle q
\]

\[
\langle \alpha \beta \rangle q \mid \langle \alpha \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]

\[
\langle p \beta \rangle q \mid \langle p \beta \rangle q
\]
The Resulting LR(1) Classifier
• \( I = (\forall, I) \) set \( \text{GOTO} \) for every non-terminal \( A \) such that \( \forall \) \( I \).
• All entries not set by the above rules are set to \( \text{ERROR} \).
• If any conflicting actions result from the above rules, then the grammar is not LR(1).
• If \( I \) is an accepting state annotated by \( [A, \gamma] \), then set \( \text{ACTION} \) [\( I, \gamma \)] to \( \text{ACCEPT} \).
• If \( I \) is an accepting state annotated by \( [S, \gamma] \), then set \( \text{ACTION} \) [\( I, \gamma \)] to \( \text{SHIFT} \).
• For every terminal \( \alpha \), if \( I = (\forall, I) \) then set \( \text{ACTION} \) [\( I, \alpha \)] to \( \text{SHIFT} \).
• State \( I \) corresponds to \( I' \). The parsing actions are determined as follows:

**Constructing LR(1) Parsing Tables**
Consider the following grammar:

\[
S \\rightarrow S \mid C C \\
C \rightarrow c C \mid c \\
S \rightarrow S \mid S \]

Constructing the induced grammar \(GLR(1)\), we obtain:

\[
[p, p \leftarrow C] p \mid [p, p \leftarrow C] C C \mid \langle pC \rangle C \leftarrow \langle pS \rangle \\
[p, p \leftarrow C] p \mid [p, p \leftarrow C] C C \mid \langle pC \rangle C \leftarrow \langle pS \rangle \\
[S, p \leftarrow C] p \mid [S, p \leftarrow C] C C \mid \langle pC \rangle C \leftarrow \langle pS \rangle \\
[S, s \leftarrow S] S \mid \langle pS \rangle \leftarrow \langle pS \rangle \\
[S, s \leftarrow S] S \mid \langle pS \rangle \leftarrow \langle pS \rangle \\
[S, s \leftarrow S] S \mid \langle pS \rangle \leftarrow \langle pS \rangle \\
\]

Consider the following grammar:

\[
p \mid C C \leftarrow C \\
C C \leftarrow S \\
S \leftarrow S \\
\]

**From LR(1) to LALR**
Lecture 4: Bottom-Up Parsing

A. Pnueli

Honors Compilers, NYU, Spring, 2007

Note that we can merge together states with $I_6$ and state with $I_8$ without creating conflicts (but giving up some error-detection options). This leads to an LALR classifier.

The LR(1) Classifier
The LALR(1) Classifier