LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:
A grammar is LR(1) if, given a rightmost derivation
\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]
We can
1. isolate the handle of each right-sentential form \( \gamma_i \), and
2. determine the production by which to reduce,
by scanning \( \gamma_i \) from left-to-right, going at most 1 symbol beyond the right end of the handle of \( \gamma_i \)

Shift Reduce Parsing

- a right-sentential form is any string that may occur in a legal rightmost derivation
- a viable prefix of a right-sentential form is any prefix that does not continue past the right end of its rightmost handle

Shift-reduce parsers

- operator precedence - define precedence between operands to guide reductions
- LR(1) --- construct DFA for recognizing viable prefix, storing lookahead information in DFA
- SLR(1) --- LR(0) + FOLLOW
  > construct DFA for recognizing viable prefix, use FOLLOW to guide reductions
- LALR(1) --- construct DFA for recognizing viable prefix, propagating lookahead information in DFA
LR(1) Parsers

A table-driven LR(1) parser looks like

![Diagram showing the components of an LR(1) parser: Scanner, Table-driven Parser, ACTION & GOTO Tables, Parser Generator.]

Tables can be built by hand
It is a perfect task to automate

LR(1) Skeleton Parser

```plaintext
stack.push(INVALID); stack.push(s0); not_found = true; token = scanner.next_token(); do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A → β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s, A]);
    } else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    } else if ( ACTION[s,token] == "accept" ) then {
        not_found = false;
    } else report a syntax error and recover;
} report success;
```

The skeleton parser
- uses ACTION & GOTO tables
- does |words| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases
LR(1) Parsers (parse tables)

To make a parser for \( L(G) \), need a set of tables

The grammar

\[
\begin{align*}
1 &: \text{Goal} \rightarrow \text{SheepNoise} \\
2 &: \text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
3 &: \text{SheepNoise} \rightarrow \ baa
\end{align*}
\]

The tables

\[
\begin{array}{|c|c|c|}
\hline
\text{ACTION} & \text{GOTO} \\
\hline
\text{State} & \text{EOF} & \text{baa} & \text{State} & \text{SheepNoise} \\
\hline
0 & \_ & \_ & 0 & 1 \\
1 & \text{accept} & \text{shift 3} & 1 & 0 \\
2 & \text{reduce 3} & \text{reduce 3} & 2 & 0 \\
3 & \text{reduce 2} & \text{reduce 2} & 3 & 0 \\
\hline
\end{array}
\]

Example Parse 1

The string "baa"

\[
\begin{array}{|c|c|c|}
\hline
\text{Stack} & \text{Input} & \text{Action} \\
\hline
$ s_0 $ & \_ & \text{shift 2} \\
$ s_0 \ baa \ s_2 $ & \text{baa} \ \text{EOF} & \text{shift 2} \\
$ s_0 \ SN \ s_1 $ & \text{EOF} & \text{reduce 3} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{ACTION} & \text{GOTO} \\
\hline
\text{State} & \text{EOF} & \text{baa} & \text{State} & \text{SheepNoise} \\
\hline
0 & \_ & \_ & 0 & 1 \\
1 & \text{accept} & \text{shift 3} & 1 & 0 \\
2 & \text{reduce 3} & \text{reduce 3} & 2 & 0 \\
3 & \text{reduce 2} & \text{reduce 2} & 3 & 0 \\
\hline
\end{array}
\]
Example Parse 2

The string “baa baa”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0</td>
<td>baa baa EOF</td>
<td>shift 2</td>
</tr>
<tr>
<td>$s_0 baa s_2</td>
<td>baa EOF</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$s_0 SN s_1</td>
<td>baa EOF</td>
<td>shift 3</td>
</tr>
<tr>
<td>$s_0 SN s_1</td>
<td>EOF</td>
<td>reduce 2</td>
</tr>
<tr>
<td>$s_0 SN s_1</td>
<td>EOF</td>
<td>accept</td>
</tr>
</tbody>
</table>

1  Goal  ➔  Sheep Noise
2  Sheep Noise  ➔  Sheep Noise baa
3  | b a a

ACTION

<table>
<thead>
<tr>
<th>State</th>
<th>EOF</th>
<th>baa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>shift 2</td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
<td>shift 3</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2</td>
<td>reduce 2</td>
</tr>
</tbody>
</table>

GOTO

<table>
<thead>
<tr>
<th>State</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

LR(0) items

- An LR(0) item is a string $\alpha$, where $\alpha$ is a production from $G$ with a
  - at some position in the rhs
    - The • indicates how much of an item we have seen at a given state in the parse.
- $[A ::= • X Y Z]$ indicates that the parser is looking for a string that can be derived from $X Y Z$
- $[A ::= X Y • Z]$ indicates that the parser has seen a string derived from $X Y$ and is looking for one derivable from $Z$

LR(0) Items (no lookahead)

- The production $A ::= X Y Z$ generates 4 LR(0) items.
  - $[A ::= • X Y Z]$
  - $[A ::= X • Y Z]$
  - $[A ::= X Y • Z]$
  - $[A ::= X Y Z •]$
**LR(0) machine**

**Definitions**
- **closure** of \([A ::= \alpha \cdot B \,\beta]\) contains itself and any items of form \([B ::= \omega]\), repeat for new items.
- **goto(X)** of \([A ::= \alpha \cdot X \,\beta]\) contains the closure of \([A ::= \alpha \cdot X \,\beta]\)

**LR(0) DFA construction**
1. begin with closure of start symbol \([S ::= \alpha]\)
2. for each state, calculate **goto(X)** for all grammar symbols X, generating states
3. repeat step 2 for all newly generated states

**Properties**
- states in the DFA are sets of LR(0) items
- states represent viable prefixes of productions
- to recognize viable prefixes of language, save state of current production on stack when reducing new nonterminal

---

**LR(0) items**

**The Grammar**
- P1: \(E ::= T + E\)
- P2: \(|\ T \)
- P3: \(T ::= \text{id}\)

**The Augmented Grammar**
- P0: \(S' ::= E\)
- P1: \(E ::= T + E\)
- P2: \(|\ T \)
- P3: \(T ::= \text{id}\)

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>([S' ::= * E])</td>
<td>goto(S1)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T + E])</td>
<td>goto(S2)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T])</td>
<td>goto(S2)</td>
</tr>
<tr>
<td></td>
<td>([T ::= * \text{id}])</td>
<td>goto(S3)</td>
</tr>
<tr>
<td>S1</td>
<td>([S' ::= * E])</td>
<td>Reduce (and accept)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T + E])</td>
<td>goto(S4)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T])</td>
<td>Reduce</td>
</tr>
<tr>
<td>S2</td>
<td>([T ::= * \text{id}])</td>
<td>Reduce</td>
</tr>
<tr>
<td>S3</td>
<td>([E ::= T + * E])</td>
<td>goto(S5)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T + E])</td>
<td>goto(S2)</td>
</tr>
<tr>
<td></td>
<td>([E ::= * T])</td>
<td>goto(S2)</td>
</tr>
<tr>
<td></td>
<td>([T ::= * \text{id}])</td>
<td>goto(S3)</td>
</tr>
<tr>
<td>S4</td>
<td>([E ::= T + E * E])</td>
<td>Reduce</td>
</tr>
<tr>
<td>S5</td>
<td>([E ::= T + E * E])</td>
<td>Reduce</td>
</tr>
</tbody>
</table>

---

Oops - Shift-reduce Conflict. What do you do here?
LR(0) acceptance

- A grammar is LR(0) if the associated item lists contain no shift reduce conflicts (or reduce-reduce conflicts).

- Using the tables for this grammar, try to parse \(a+b\).
  - For what states do you shift?
  - For what states do you reduce?
  - What happens?

- But there are LR(0) grammars:
  - Consider earlier grammar:
    \[
    \text{Goal} ::= \text{SheepNoise}
    \]
    \[
    \text{SheepNoise} ::= \text{SheepNoise} \text{ baa} | \text{baa}
    \]

- Build LR(0) item lists

<table>
<thead>
<tr>
<th>State</th>
<th>Item List</th>
<th>Action</th>
</tr>
</thead>
</table>
| S0    | [Goal ::= • SheepNoise] got to (S1) 
[SheepNoise ::= • SheepNoise baa] got to (S2) 
[SheepNoise ::= • baa] got to (S3) |        |
| S1    | [Goal ::= SheepNoise •] Accept |        |
| S2    | [SheepNoise ::= SheepNoise • baa] got to (S4) |        |
| S3    | [SheepNoise ::= baa •] Reduce |        |
| S4    | [SheepNoise ::= SheepNoise baa •] Reduce |        |

No conflicts in table, so grammar is LR(0) and any string can be parsed.

---

SLR(1)

- Perhaps all is not lost. Consider LR(0) conflict previously in parsing \(a+b\). If at the point of conflict we can then look one symbol ahead, perhaps we can resolve the problem.

- That is, state S2 was:

  \[
  S2
  \]
  \[
  \begin{align*}
  E ::= T & \ast E \quad \text{goto}(S4) \\
  E ::= T & \ast \quad \text{Reduce}
  \end{align*}
  \]

- For what inputs do we shift? For what inputs do we reduce?
  - If you look at grammar you should see that shift is the desired action only when the next symbol is the \(\ast\) symbol. So a one character lookahead may be all that is needed to resolve the problem.

- Define an inadequate state as a state containing LR(0) items, which have either a shift-reduce or a shift-shift conflict.

- A grammar is SLR(1) if for each inadequate state S:
  - If \([X ::= \alpha \ast \beta]\) and \([Y ::= \omega \ast]\) are in S then First(\(j\)) \(\cap\) Follow(\(Y\)) = \(\emptyset\), and
  - If \([X ::= \alpha \ast\)] and \([Y ::= \omega \ast]\) are in S then Follow(\(X\)) \(\cap\) Follow(\(Y\)) = \(\emptyset\)
Redo Expression grammar – Grammar now SLR(1)

• The Grammar

\[
P_1 \quad E ::= T + E
\]

\[
P_2 \quad | T
\]

\[
P_3 \quad T ::= id
\]

• The Augmented Grammar

\[
P_0 \quad S' ::= E
\]

\[
P_1 \quad E ::= T + E
\]

\[
P_2 \quad | T
\]

\[
P_3 \quad T ::= id
\]

\[
S_0 \quad [S' ::= \text{• } E] \rightarrow \text{goto}(S1)
\]

\[
S_1 \quad [S' ::= \text{• } E \text{•}] \rightarrow \text{Reduce (and accept)}
\]

\[
S_2 \quad [E ::= \text{• } T + E \text{•}] \rightarrow \text{First(\text{•})= +, goto(S4)}
\]

\[
S_3 \quad [E ::= \text{• } T \text{•}] \rightarrow \text{Follow(E) = \text{eof}, Reduce}
\]

\[
S_4 \quad [T ::= \text{• } id \text{•}] \rightarrow \text{Reduce}
\]

\[
S_5 \quad [E ::= \text{• } T + E \text{•}] \rightarrow \text{Reduce}
\]

Computing FIRST Sets

Define FIRST as

• If \( \alpha \Rightarrow^* a \beta \), \( a \in T \), \( \beta \in (T \cup NT)^* \), then \( a \in \text{FIRST(\alpha)} \)

• If \( \alpha \Rightarrow^* \varepsilon \), then \( \varepsilon \in \text{FIRST(\alpha)} \)

Note: if \( \alpha = X \beta \), \( \text{FIRST(\alpha)} = \text{FIRST(X)} \)

To compute FIRST

• Use a fixed-point method

\( \text{FIRST(A)} \in 2^{(T \cup \varepsilon)} \)

• Loop is monotonic

\( \Rightarrow \) Algorithm halts

For SheepNoise:

\( \text{FIRST(Goal)} = \{ \text{baa} \} \)

\( \text{FIRST(SN)} = \{ \text{baa} \} \)

\( \text{FIRST(baa)} = \{ \text{baa} \} \)
**LR(1) items**

We can get more powerful parsers by keeping track of lookahead information in the states of the LR parser.

- An LR(k) item is a pair [α, β] where
  - α is a production from G with a • at some position in the rhs
  - β is a lookahead string containing k symbols (terminals or eof)

LR(1) items

- example: [A ::= X • Y Z, a]
- several LR(1) items may have the same core {i.e., LR(0) item lists}
  - [A ::= X • Y Z, a]
  - [A ::= X • Y Z, b]
- we represent this as [A ::= X • Y Z, ab]

**LR(1) Lookahead**

What’s the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has • at right end.
  - in [A ::= X • Y Z, a], a has no direct use
  - in [A ::= X Y Z •, a], a is useful
- allows use of (non-invertible) grammars where productions have the same rhs

The point

- For [A ::= α •, a] and [B ::= α •, b],
- we can decide between reducing to A and to B by looking at limited right context!
**LR(1) machine**

**Definitions**
- **closure** of \([A ::= \alpha \cdot B, a]\) contains itself and any items of form \([B ::= \cdot \omega, \text{First(} \beta \cdot a)\)], repeat for new items.
- **goto(X)** of \([A ::= \alpha \cdot X, a]\) contains the closure of \([A ::= \alpha \cdot X, a]\).

**LR(1) DFA construction**
- begin with closure of start symbol \([S ::= \cdot \alpha, \text{eof}\])
- for each state, calculate **goto}(X) for all grammar symbols X, generating states
- repeat step 2 for all newly generated states

**Properties**
- \([A ::= X \cdot YZ, a] \Rightarrow \text{have recognized } X & YZ \text{ would be valid}
- \([A ::= X \cdot YZ, a] \Rightarrow [Y ::= \cdot \beta, \omega] \text{ & } [Y ::= \cdot \delta, \gamma] \text{ are also valid, where } \omega, \gamma \in \text{FIRST(Z } \alpha) \text{ recognizing Y takes parser to } [A ::= XY \cdot Z, a]\)

---

**Example: LR(1) states**

<table>
<thead>
<tr>
<th>State</th>
<th>Grammar</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S0</strong></td>
<td>([ S' ::= \cdot E, $ ])</td>
<td>goto(S1)</td>
</tr>
<tr>
<td></td>
<td>([ E ::= \cdot T + E, $ ])</td>
<td>goto(S2) FIRST((\epsilon \cdot $)) = (\epsilon \cdot $)</td>
</tr>
<tr>
<td></td>
<td>([ E ::= \cdot T, $ ])</td>
<td>goto(S2) FIRST((\epsilon \cdot $)) = (\epsilon \cdot $)</td>
</tr>
<tr>
<td></td>
<td>([ T ::= \cdot \text{id}, * ])</td>
<td>goto(S3) FIRST(+ E $) = +</td>
</tr>
<tr>
<td></td>
<td>([ T ::= \cdot \text{id}, $ ])</td>
<td>goto(S3) FIRST((\epsilon \cdot $)) = (\epsilon \cdot $)</td>
</tr>
<tr>
<td><strong>S1</strong></td>
<td>([ S' ::= \cdot E, $ ])</td>
<td>Reduce</td>
</tr>
<tr>
<td><strong>S2</strong></td>
<td>([ E ::= \cdot T + E, $ ])</td>
<td>If next(=) goto(S4)</td>
</tr>
<tr>
<td></td>
<td>([ E ::= \cdot T, $ ])</td>
<td>if next(=$) Reduce</td>
</tr>
<tr>
<td><strong>S3</strong></td>
<td>([ T ::= \cdot \text{id}, * ])</td>
<td>Reduce</td>
</tr>
<tr>
<td></td>
<td>([ T ::= \cdot \text{id}, $ ])</td>
<td>Reduce</td>
</tr>
<tr>
<td><strong>S4</strong></td>
<td>([ E ::= \cdot T + E, $ ])</td>
<td>goto(S5)</td>
</tr>
<tr>
<td></td>
<td>([ E ::= \cdot T, $ ])</td>
<td>goto(S2) FIRST((\epsilon \cdot $)) = (\epsilon \cdot $)</td>
</tr>
<tr>
<td></td>
<td>([ E ::= \cdot \text{id}, * ])</td>
<td>goto(S3) FIRST(+ E $) = +</td>
</tr>
<tr>
<td></td>
<td>([ T ::= \cdot \text{id}, $ ])</td>
<td>goto(S3) FIRST((\epsilon \cdot $)) = (\epsilon \cdot $)</td>
</tr>
<tr>
<td><strong>S5</strong></td>
<td>([ E ::= \cdot T + E, $ ])</td>
<td>Reduce</td>
</tr>
</tbody>
</table>
**LALR(1) parsers**

**Problem**
- LR(1) parsers are powerful, but have many more states than LR(0) (approximately x 10 for Pascal)
- Larger state tables longer to construct, run

**LALR(1) parsers**
- Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.
  - Example of two sets of LR(1) items with same core:
    - \([A := \alpha \cdot \beta, a], [A := \alpha \cdot \beta, b]\), and \([A := \alpha \cdot \beta, c], [A := \alpha \cdot \beta, d]\)
- If two sets of LR(1) items, I1 and I2 have the same core, we can merge the states that represent them in the ACTION and GOTO tables
- Almost as powerful as LR(1), same size as LR(0)

---

**LR(1) Parsers – Formal definitions – Repeat of previous slides**

How does this LR(1) stuff work?
- Unambiguous grammar \(\Rightarrow\) unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - Build a handle-recognizing DFA
  - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA & leave old DFA’s state on stack
- Final state in DFA \(\Rightarrow\) a reduce action
  - New state is GOTO[state at TOS (after pop), lhs]
  - For SN, this takes the DFA to s₁

Control DFA for SN
Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?
• Use the grammar to build a model of the DFA
• Use the model to build ACTION & GOTO tables
• If construction succeeds, the grammar is LR(1)

The Big Picture
• Model the state of the parser
• Use two functions goto( s, X ) and closure( s )
  > goto() is analogous to move() in the subset construction
  > closure() adds information to round out a state
• Build up the states and transition functions of the DFA
• Use this information to fill in the ACTION and GOTO tables

LR(k) items

An LR(k) item is a pair \([P, \delta]\), where

- \(P\) is a production \(A \rightarrow \beta\) with a • at some position in the rhs
- \(\delta\) is a lookahead string of length \(\leq k\) (words or EOF)

The • in an item indicates the position of the top of the stack

- \([A \rightarrow \beta\gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta\gamma\) immediately after the symbol on top of the stack
- \([A \rightarrow \beta\gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta\gamma\) at this point in the parse, and that the parser has already recognized \(\beta\).
- \([A \rightarrow \beta\gamma, a]\) means that the parser has seen \(\beta\gamma\), and that a lookahead symbol of \(a\) is consistent with reducing to \(A\).

The table construction algorithm uses items to represent valid configurations of an LR(1) parser
**LR(1) Items**

The production $A \rightarrow \beta$, where $\beta = B_1B_1B_1$ with lookahead $a$, can give rise to 4 items:

$$[A \rightarrow B_1B_1B_1, a], [A \rightarrow B_1B_1B_1 \cdot, a], [A \rightarrow B_1\cdot B_1B_1, a], [A \rightarrow B_1B_1\cdot B_1, a], [A \rightarrow B_1B_1B_1, a]$$

The set of LR(1) items for a grammar is finite.

What’s the point of all these lookahead symbols?

- Carry them along to choose correct reduction (if a choice occurs)
- Lookaheads are bookkeeping, unless item has $\cdot$ at right end
  - Has no direct use in $[A \rightarrow \beta \cdot, a]$
  - In $[A \rightarrow \beta \cdot, a]$, a lookahead of $a$ implies a reduction by $A \rightarrow \beta$
  - For $[A \rightarrow \beta \cdot, a], [B \rightarrow \gamma \delta, b]$, $a \Rightarrow$ reduce to $A$; $\text{FIRST}(\delta) \Rightarrow$ shift

⇒ Limited right context is enough to pick the actions.

---

**LR(1) Table Construction**

High-level overview

1. Build the canonical collection of sets of LR(1) Items, $I$
   - **a** Begin in an appropriate state, $s_0$
     - $\rightarrow [S \rightarrow S, \text{EOF}]$, along with any equivalent items
     - Derive equivalent items as $\text{closure}(I_0)$
   - **b** Repeatedly compute, for each $s_n$ and each $X$, $\text{goto}(s_n, X)$
     - If the set is not already in the collection, add it
     - Record all the transitions created by $\text{goto}()$
     - This eventually reaches a fixed point

2. Fill in the table from the collection of sets of LR(1) items

   The canonical collection completely encodes the transition diagram for the handle-finding DFA
Back to Finding Handles

Revisiting an issue from last class

Parser in a state where the stack (the fringe) was

\[ \textit{Expr} \rightarrow \textit{Term} \]

With lookahead of *.

How did it choose to expand \textit{Term} rather than reduce to \textit{Expr}?

- \textit{Lookahead} symbol is the key
- With lookahead of + or –, parser should reduce to \textit{Expr}
- With lookahead of * or /, parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism
Computing Closures

Closure(s) adds all the items implied by items already in s

- Any item \([A \rightarrow \beta \cdot B \delta, a]\) implies \([B \rightarrow \cdot \tau, x]\) for each production with \(B\) on the lhs, and each \(x \in \text{FIRST}(\delta a)\)
- Since \(\beta B\delta\) is valid, any way to derive \(\beta B\delta\) is valid, too

The algorithm

```
Closure(s)
while (s is still changing)
  \forall\ \text{items } [A \rightarrow \beta \cdot B \delta, a] \in s
  \forall \text{ productions } B \rightarrow \tau \in P
  \forall b \in \text{FIRST}(\delta a) \text{// } \delta \text{ might be } \varepsilon
  \text{ if } [B \rightarrow \cdot \tau, b] \notin s
  \text{ then add } [B \rightarrow \cdot \tau, b] \text{ to } s
```

- Classic fixed-point algorithm
- Halts because \(s \subset \text{ITEMS}\)
- Worklist version is faster

Closure “fills out” a state

Example From SheepNoise

Initial step builds the item \([\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}]\) and takes its closure()

\[
\text{Closure}( [\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}] )
\]

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}])</td>
<td>Original item</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa, EOF}])</td>
<td>1, (\delta a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \cdot \text{baa, EOF}])</td>
<td>1, (\delta a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \cdot \text{baa, baa}])</td>
<td>2, (\delta a) is (\text{baa EOF})</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \cdot \text{baa baa}])</td>
<td>2, (\delta a) is (\text{baa EOF})</td>
</tr>
</tbody>
</table>

So, \(S_0\) is

\[
\{ [\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa, EOF}], [\text{SheepNoise} \rightarrow \cdot \text{baa, EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa, baa}], [\text{SheepNoise} \rightarrow \cdot \text{baa baa}] \}
\]
**Computing Gotos**

*Goto*(s,x) computes the state that the parser would reach if it recognized an x while in state s

- *Goto*([A→β•Xδ,a]) produces [A→β•Xδ,a] \( \text{(obviously)} \)
- It also includes closure([A→β•Xδ,a]) to fill out the state

The algorithm

\[
\text{Goto}( s, X ) \\
\text{new} \leftarrow \emptyset \\
\forall \text{ items } [A \rightarrow \beta \cdot X \delta, a] \in s \\
\text{new} \leftarrow \text{new} \cup [A \rightarrow \beta \cdot X \delta, a] \\
\text{return closure(new)}
\]

- Not a fixed point method!
- Straightforward computation
- Uses closure()

*Goto() advances the parse*

**Example from SheepNoise**

S₀ is \{ [Goal→ • SheepNoise,EOF], [SheepNoise→ • SheepNoise baa,EOF], [SheepNoise→ • baa, EOF], [SheepNoise→ • SheepNoise baa], [SheepNoise→ • baa, baa] \}

*Goto* (S₀, baa)

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SheepNoise→ baa•, EOF]</td>
<td>Item 3 in s₀</td>
</tr>
<tr>
<td>[SheepNoise→ baa•, baa]</td>
<td>Item 5 in s₀</td>
</tr>
</tbody>
</table>

- Closure adds nothing since • is at end of rhs in each item

In the construction, this produces s₂

\{ [SheepNoise→ baa•, (EOF, baa)] \}

New, but obvious, notation for two distinct items

[SheepNoise→ baa•, EOF] & [SheepNoise→ baa•, baa]
Example from SheepNoise

\[ S_0 : \{ [\text{Goal} \rightarrow \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, baa}], [\text{SheepNoise} \rightarrow \text{baa, baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \text{SheepNoise}) = \{ [\text{Goal} \rightarrow \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \text{baa, baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise baa, baa}] \} \]

Building the Canonical Collection

Start from \( s_0 = \text{closure}([S' \rightarrow S, \text{EOF}]) \)
Repeatedly construct new states, until all are found

The algorithm

```
s_0 \leftarrow \text{closure}([S' \rightarrow S, \text{EOF}])
S \leftarrow \{ s_0 \}
k \leftarrow 1
while (S is still changing)
    \forall s_j \in S \text{ and } \forall x \in (T \cup NT)
    s_k \leftarrow \text{goto}(s_j, x)
    record \ s_j \rightarrow s_k \text{ on } x
    if \ s_k \notin S \text{ then}
        S \leftarrow S \cup s_k
        k \leftarrow k + 1
```

- Fixed-point computation
- Loop adds to \( S \)
- \( S \subseteq \text{ITEMS} \), so \( S \) is finite
- Worklist version is faster
Example (grammar & sets)

Simplified, right recursive expression grammar

\[
\begin{align*}
\text{Goal} & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Term} - \text{Expr} \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \ast \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \\
\text{Factor} & \rightarrow \text{ident} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>{ident}</td>
</tr>
<tr>
<td>Expr</td>
<td>{ident}</td>
</tr>
<tr>
<td>Term</td>
<td>{ident}</td>
</tr>
<tr>
<td>Factor</td>
<td>{ident}</td>
</tr>
<tr>
<td>-</td>
<td>{-}</td>
</tr>
<tr>
<td>*</td>
<td>{*}</td>
</tr>
<tr>
<td>ident</td>
<td>{ident}</td>
</tr>
</tbody>
</table>

Example (building the collection)

Initialization Step

\[
s_0 \leftarrow \text{closure}( \{ [\text{Goal} \rightarrow \text{Expr} , \text{EOF}] \} )
\]

\[
\{ [\text{Goal} \rightarrow \bullet \text{Expr} , \text{EOF}] , [\text{Expr} \rightarrow \bullet \text{Term} - \text{Expr} , \text{EOF}] , [\text{Expr} \rightarrow \bullet \text{Term} , \text{EOF}] , [\text{Term} \rightarrow \bullet \text{Factor} \ast \text{Term} , \text{EOF}] , [\text{Term} \rightarrow \bullet \text{Factor} , \text{EOF}] , [\text{Term} \rightarrow \bullet \text{Factor} , -] , [\text{Factor} \rightarrow \bullet \text{ident} , \text{EOF}] , [\text{Factor} \rightarrow \bullet \text{ident} , -] , [\text{Factor} \rightarrow \bullet \text{ident} , \ast] \}
\]

\[
S \leftarrow \{s_0\}
\]
Example (building the collection)

Iteration 1
\[
\begin{align*}
  s_1 & \leftarrow \text{goto}(s_0, \text{Expr}) \\
  s_2 & \leftarrow \text{goto}(s_0, \text{Term}) \\
  s_3 & \leftarrow \text{goto}(s_0, \text{Factor}) \\
  s_4 & \leftarrow \text{goto}(s_0, \text{ident})
\end{align*}
\]

Iteration 2
\[
\begin{align*}
  s_5 & \leftarrow \text{goto}(s_2, -) \\
  s_6 & \leftarrow \text{goto}(s_3, *)
\end{align*}
\]

Iteration 3
\[
\begin{align*}
  s_7 & \leftarrow \text{goto}(s_5, \text{Expr}) \\
  s_8 & \leftarrow \text{goto}(s_6, \text{Term})
\end{align*}
\]

Example (Summary)

\[
\begin{align*}
 S_0 : \{ & \text{[Goal} \rightarrow \text{Expr, EOF]}, \text{[Expr} \rightarrow \text{Term} \rightarrow \text{Expr, EOF]}, \text{[Expr} \rightarrow \text{Term, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_1 : \{ & \text{[Goal} \rightarrow \text{Expr, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_2 : \{ & \text{[Expr} \rightarrow \text{Term} \rightarrow \text{Expr, EOF]}, \text{[Expr} \rightarrow \text{Term, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_3 : \{ & \text{[Term} \rightarrow \text{Factor} \rightarrow \text{Term, EOF]}, \text{[Term} \rightarrow \text{Factor, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_4 : \{ & \text{[Factor} \rightarrow \text{ident, EOF]}, \text{[Factor} \rightarrow \text{ident, -]}, \text{[Factor} \rightarrow \text{ident, *]}
\end{align*}
\]

\[
\begin{align*}
 S_5 : \{ & \text{[Expr} \rightarrow \text{Term} \rightarrow \text{Expr, EOF]}, \text{[Expr} \rightarrow \text{Term, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_6 : \{ & \text{[Factor} \rightarrow \text{ident, EOF]}, \text{[Factor} \rightarrow \text{ident, -]}, \text{[Factor} \rightarrow \text{ident, *}]
\end{align*}
\]

\[
\begin{align*}
 S_7 : \{ & \text{[Term} \rightarrow \text{Factor} \rightarrow \text{Term, EOF]}, \text{[Term} \rightarrow \text{Factor, EOF]}
\end{align*}
\]

\[
\begin{align*}
 S_8 : \{ & \text{[Factor} \rightarrow \text{ident, -]}, \text{[Factor} \rightarrow \text{ident, *]}, \text{[Factor} \rightarrow \text{ident, EOF]}
\end{align*}
\]
Example (Summary)

\[ S_6 : \{ \text{Term} \rightarrow \text{Factor} \ast \text{Term} \text{, EOF}, \text{Term} \rightarrow \text{Factor} \ast \text{Term} \text{, -} \}, \]
\[ \text{Term} \rightarrow \ast \text{Factor} \text{, EOF}, \text{Term} \rightarrow \ast \text{Factor} \text{, -} \}, \]
\[ \text{Factor} \rightarrow \ast \text{ident} \text{, EOF}, \text{Factor} \rightarrow \ast \text{ident} \text{, -}, \text{Factor} \rightarrow * \text{ident} \text{, -} \} \]

\[ S_7 : \{ \text{Expr} \rightarrow \text{Term} - \text{Expr} \text{, EOF} \} \]

\[ S_8 : \{ \text{Term} \rightarrow \text{Factor} \ast \text{Term} \text{, EOF}, \text{Term} \rightarrow \text{Factor} \ast \text{Term} \text{, -} \} \]

Example (Summary)
The Goto Relationship (from the construction)

<table>
<thead>
<tr>
<th>State</th>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
<th>-</th>
<th>*</th>
<th>Ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Filling in the ACTION and GOTO Tables

The algorithm

∀ set \( s \in S \)
∀ item \( i \in s \)
if \( i \) is \( [A \rightarrow \beta •, a] \) and \( \text{goto}(s, a) = s_k, a \in T \)
then \( \text{ACTION}[x, a] \leftarrow \text{"shift } k \" \)
else if \( i \) is \( [S' \rightarrow S •, EOF] \)
then \( \text{ACTION}[x, a] \leftarrow \text{"accept"} \)
else if \( i \) is \( [A \rightarrow \beta •, a] \)
then \( \text{ACTION}[x, a] \leftarrow \text{"reduce } A \rightarrow \beta \" \)
∀ \( n \in NT \)
if \( \text{goto}(s, n) = s_k \)
then \( \text{GOTO}[x, n] \leftarrow k \)

Many items generate no table entry
> Closure( ) instantiates FIRST(\( \chi \)) directly for \( [A \rightarrow \beta •, a] \)

Example (Filling in the tables)

The algorithm produces the following table

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Ident</td>
<td>-</td>
</tr>
<tr>
<td>0 s 4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
</tr>
<tr>
<td>2 s 5</td>
<td>r 3</td>
</tr>
<tr>
<td>3 r 5</td>
<td>s 6</td>
</tr>
<tr>
<td>4 r 6</td>
<td>r 6</td>
</tr>
<tr>
<td>5 s 4</td>
<td>7</td>
</tr>
<tr>
<td>6 s 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r 2</td>
</tr>
<tr>
<td>8 r 4</td>
<td>r 4</td>
</tr>
</tbody>
</table>

END OF DUPLICATE DEFINITION OF LR(1)
What can go wrong?

What if set $s$ contains $[A \rightarrow \beta \cdot, a]$ and $[B \rightarrow \gamma \cdot, a]$?
- First item generates “shift”, second generates “reduce”
- Both define $\text{ACTION}[s,a]$ — cannot do both actions
- This is a fundamental ambiguity, called a shift/reduce error
- Modify the grammar to eliminate it (if-then-else)
- Shifting will often resolve it correctly

What is set $s$ contains $[A \rightarrow \gamma \cdot, a]$ and $[B \rightarrow \gamma \cdot, a]$?
- Each generates “reduce”, but with a different production
- Both define $\text{ACTION}[s,a]$ — cannot do both reductions
- This is a fundamental ambiguity, called a reduce/reduce conflict
- Modify the grammar to eliminate it (PL/I’s overloading of (...))

In either case, the grammar is not LR(1)

Shrinking the Tables

Three options:

- Combine terminals such as number & identifier, $+$ & $-$, $*$ & $/$
  > Directly removes a column, may remove a row
  > For expression grammar, 198 (vs. 384) table entries

- Combine rows or columns
  > Implement identical rows once & remap states
  > Requires extra indirection on each lookup
  > Use separate mapping for $\text{ACTION}$ & for $\text{GOTO}$

- Use another construction algorithm
  > Both LALR(1) and SLR(1) produce smaller tables
  > Implementations are readily available
LR(k) versus LL(k)  (Top-down Recursive Descent)

Finding Reductions

LR(k) ⇒ Each reduction in the parse is detectable with
1 the complete left context,
2 the reducible phrase, itself, and
3 the k terminal symbols to its right

LL(k) ⇒ Parser must select the reduction based on
1 The complete left context
2 The next k terminals

Thus, LR(k) examines more context

“… in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages”  J.J. Horning, “LR Grammars and Analysers”, in Compiler Construction, An Advanced Course, Springer-Verlag, 1976

• Summary

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down recursive descent</td>
<td>Fast</td>
</tr>
<tr>
<td></td>
<td>Good locality</td>
</tr>
<tr>
<td></td>
<td>Simplicity</td>
</tr>
<tr>
<td></td>
<td>Good error detection</td>
</tr>
<tr>
<td>LR(1)</td>
<td>Hand-coded</td>
</tr>
<tr>
<td></td>
<td>High maintenance</td>
</tr>
<tr>
<td></td>
<td>Right associativity</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
</tr>
<tr>
<td></td>
<td>Deterministic langs.</td>
</tr>
<tr>
<td></td>
<td>Automatable</td>
</tr>
<tr>
<td></td>
<td>Left associativity</td>
</tr>
<tr>
<td></td>
<td>Large working sets</td>
</tr>
<tr>
<td></td>
<td>Poor error messages</td>
</tr>
<tr>
<td></td>
<td>Large table sizes</td>
</tr>
</tbody>
</table>
**Left Recursion versus Right Recursion**

**Right recursion**
- Required for termination in top-down parsers
- Uses (on average) more stack space
- Produces right-associative operators

**Left recursion**
- Works fine in bottom-up parsers
- Limits required stack space
- Produces left-associative operators

**Rule of thumb**
- Left recursion for bottom-up parsers
- Right recursion for top-down parsers

---

**Associativity**

What difference does it make?
- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions

Consider \(x + y + z\)

![Diagram showing ideal, left, and right association](image)

What if \(y + z\) occurs elsewhere? Or \(x + y\) or \(x + z\)?

What if \(x = 2\) & \(z = 17\)? Neither left nor right exposes 19.

Best choice is function of surrounding context.
Hierarchy of Context-Free Languages

- Context-free languages
  - Deterministic languages (LR(\$))
  - LL(\$) languages
  - LL(1) languages
  - Simple precedence languages
  - Operator precedence languages

LR(\$) = LR(1)

Hierarchy of Context-Free Grammars

- Context-free grammars
  - Floyd-Evans Parsable
  - Unambiguous CFGs
    - LR(\$)
    - LR(1)
    - LALR(1)
    - SLR(1)
    - LR(0)
  - Operator Precedence
    - LL(\$)
    - LL(1)

- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)