Bottom-Up Parsing
Parsing Techniques

Top-down parsers \((LL(1), \text{recursive descent})\)
- Start at root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" \(\Rightarrow\) may need to backtrack
- Some grammars are backtrack-free \((\text{predictive parsing})\)

Bottom-up parsers \((LR(1), \text{operator precedence})\)
- Start at the leaves and grow toward root
- As input consumed, encode possibilities in internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars
Bottom-up Parsing (definitions)

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

- Each \( \gamma_i \) is a sentential form
  - If \( \gamma \) contains only terminal symbols, \( \gamma \) is a sentence in \( L(G) \)
  - If \( \gamma \) contains 1 or more non-terminals, \( \gamma \) is a sentential form
Bottom-up Parsing (definitions)

$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$

- To get $\gamma_i$ from $\gamma_{i-1}$, expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with $\beta$ to get $\gamma_i$
  - In a leftmost derivation, it would be first NT $A \in \gamma_{i-1}$
Bottom-up Parsing (definitions)

A *left-sentential form* occurs in a *leftmost* derivation

A *right-sentential form* occurs in a *rightmost* derivation

*Bottom-up parsers build rightmost derivation in reverse*
A bottom-up parser builds derivation by working from input sentence back toward the start symbol $S$.

$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$

assuming $A \Rightarrow \beta$, match some rhs $\beta$ here

replace $\beta$ with its corresponding lhs, $A$ here
Bottom-up Parsing (definitions)

In terms of parse tree, it works from leaves to root

- Nodes with no parent in partial tree form *upper fringe*
- Each replacement of $\beta$ with $A$ shrinks the upper fringe; we call this a *reduction*.
- “Rightmost derivation in reverse” processes words *left to right*

```
Term * Fact.

Fact.  <id, y>

<num, 2>
```

upper fringe
Bottom-up Parsing (definitions)

In terms of parse tree, it works from leaves to root

- Nodes with no parent in partial tree form upper fringe
- Each replacement of $\beta$ with $A$ shrinks the upper fringe, we call this a reduction.
- “Rightmost derivation in reverse” processes words left to right
Finding Reductions

Consider the grammar

<table>
<thead>
<tr>
<th>Level</th>
<th>Symbol</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>$a A B e$</td>
</tr>
<tr>
<td>1</td>
<td>$A$</td>
<td>$A b c$</td>
</tr>
<tr>
<td>2</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>3</td>
<td>$B$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

And the input string $abbcde$

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Next Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$abbcde$</td>
<td>2</td>
</tr>
<tr>
<td>$a A bcde$</td>
<td>2</td>
</tr>
</tbody>
</table>

“Position” specifies where the right end of $\beta$ occurs in the current sentential form. We call this position $k$. 
Finding Reductions

Consider the grammar

<table>
<thead>
<tr>
<th></th>
<th>Sentential Form</th>
<th>Next Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal → a A B e</td>
<td>a A B e</td>
</tr>
<tr>
<td>1</td>
<td>A → A b c</td>
<td>a A b c</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>a A b c d</td>
</tr>
<tr>
<td>3</td>
<td>B → d</td>
<td>a A B e</td>
</tr>
</tbody>
</table>

And the input string abbcde

“Position” specifies where the right end of β occurs in the current sentential form. We call this position $k$. 
Finding Reductions (Handles)

Parser must find substring $\beta$ at parse tree’s frontier that matches some production $A \rightarrow \beta$

($\Rightarrow \beta \rightarrow A$ is in Reverse Rightmost Derivation)

We call substring $\beta$ a handle
Finding Reductions (Handles)

Formally,

A *handle* of a right-sentential form $\gamma$ is a pair $<A \rightarrow \beta, k>$ where $A \rightarrow \beta \in P$ and $k$ is the position in $\gamma$ of $\beta$’s rightmost symbol.

If $<A \rightarrow \beta, k>$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right sentential form from which $\gamma$ is derived in the rightmost derivation.
On ChalkBoard Example

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
<th>Bottom up parsers can handle either left-recursive or right-recursive grammars.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal  →  Expr</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Expr  →  Expr + Term</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Term  →  Term * Factor</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Factor  →  number</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A simple left-recursive form of the classic expression grammar
On ChalkBoard Example

A simple left-recursive form of the classic expression grammar

<table>
<thead>
<tr>
<th>Prod' n</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><code>&lt;id,x&gt;</code> - <code>&lt;num,2&gt;</code> * <code>&lt;id,y&gt;</code></td>
<td>8,1</td>
</tr>
<tr>
<td></td>
<td><code>Factor - </code>&lt;num,2&gt;<code>*</code>&lt;id,y&gt;`</td>
<td></td>
</tr>
</tbody>
</table>

`Expr` → `Expr + Term`
`Term` → `Term * Factor`
`Factor` → `number`
`id`
`(Expr)`

Handles for rightmost derivation of `x - 2 * y`
On ChalkBoard Example

A simple left-recursive form of the classic expression grammar

<table>
<thead>
<tr>
<th>Prod' n</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal → Expr</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Expr → Expr + Term</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Term → Term * Factor</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Factor → number</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Handles for rightmost derivation of \( x = 2 * y \)
Bottom-up Parsing (Abstract View)

A bottom-up parser repeatedly finds a handle $A \rightarrow \beta$ in current right-sentential form and replaces $\beta$ with $A$.

To construct a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

Apply the following conceptual algorithm

for $i \leftarrow n$ to 1 by -1

Find the handle $<A_i \rightarrow \beta_i, k_i>$ in $\gamma_i$

Replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$

This takes $2n$ steps

of course, $n$ is unknown until the derivation is built
More on Handles

Bottom-up parsers finds rightmost derivation

• Process input left to right
• Handle always appears at upper fringe of partially completed parse tree
LR parsing

• Keep upper fringe of the partially completed parse tree on a **stack**
  — Stack makes position information irrelevant
  — Handles appear at top of the stack (TOS)

*If G is unambiguous, then every right-sentential form has a *unique* handle.*
### More on Handles

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><code>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</code></td>
<td>8,1</td>
</tr>
<tr>
<td>6</td>
<td><code>Factor - &lt;num,2&gt; * &lt;id,y&gt;</code></td>
<td>6,1</td>
</tr>
<tr>
<td>3</td>
<td><code>Term - &lt;num,2&gt; * &lt;id,y&gt;</code></td>
<td>3,1</td>
</tr>
<tr>
<td>7</td>
<td><code>Expr - &lt;num,2&gt; * &lt;id,y&gt;</code></td>
<td>7,3</td>
</tr>
</tbody>
</table>

Rest of input from scanner

K=3

`Expr` stack

TOS

7 `Factor` → `number`
Shift-Reduce Parsing

To implement a bottom-up parser, we adopt the shift-reduce paradigm

A shift-reduce parser is a stack automaton with four actions

- **Shift** — next word is shifted onto the stack
- **Reduce** — right end of handle is at top of stack
  
  Located handle (rhs) on top of stack
  Pop handle off stack & push appropriate lhs

*Shift* is just a push and a call to the scanner

*Reduce* means found a handle, takes |rhs| pops & 1 push

But how does parser know when to shift and when to reduce?

It shifts until it has a handle at the top of the stack.
Shift-Reduce Parsing

- **Accept** — stop parsing & report success
- **Error** — call an error reporting/recovery routine

*Accept if no input and Goal symbol on top of stack (TOS)*

*Error otherwise*
Bottom-up Parser

A simple *shift-reduce parser*:

```
push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle \( A \rightarrow \beta \)
        then // reduce \( \beta \) to \( A \)
            pop |\( \beta \)| symbols off the stack
            push \( A \) onto the stack
    else if (token ≠ EOF)
        then // shift
            push token
            token ← next_token()
    else // need to shift, but out of input
        report an error
```

What happens on an error?
- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
**Bottom-up Parser**

A simple *shift-reduce parser*:

```plaintext
push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle $A \rightarrow \beta$
    then // reduce $\beta$ to $A$
      pop $|\beta|$ symbols off the stack
      push $A$ onto the stack
  else if (token ≠ EOF)
    then // shift
      push token
      token ← next_token()
  else // need to shift, but out of input
    report an error
```

**What happens on an error?**

- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
Bottom-up Parser

A simple shift-reduce parser:

push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)

if the top of the stack is a handle $A \rightarrow \beta$
    then // reduce $\beta$ to $A$
        pop $|\beta|$ symbols off the stack
        push $A$ onto the stack
    else if (token ≠ EOF)
        then // shift
            push token
            token ← next_token()
    else // need to shift, but out of input
        report an error

What happens on an error?
• It fails to find a handle
• Thus, it keeps shifting
• Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
Bottom-up Parser

A simple shift-reduce parser:

\[
\text{push INVALID}
\]
\[
\text{token} \leftarrow \text{next_token()} \\
\text{repeat until (top of stack = Goal and token = EOF)}
\]

\[
\text{if the top of the stack is a handle } A \rightarrow \beta
\]
\[
\quad \text{then } // \text{ reduce } \beta \text{ to } A
\]
\[
\quad \text{pop } |\beta| \text{ symbols off the stack}
\quad \text{push } A \text{ onto the stack}
\]

\[
\text{else if (token } \neq \text{ EOF)}
\]
\[
\quad \text{then } // \text{ shift}
\quad \text{push token}
\quad \text{token} \leftarrow \text{next_token()}
\]

\[
\text{else } // \text{ need to shift, but out of input}
\]

\[
\text{report an error}
\]

What happens on an error?

• It fails to find a handle
• Thus, it keeps shifting
• Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
Bottom-up Parser

A simple *shift-reduce parser*:

```plaintext
push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle $A \rightarrow \beta$
        then // reduce $\beta$ to $A$
            pop $|\beta|$ symbols off the stack
            push $A$ onto the stack
    else if (token $\neq$ EOF)
        then // shift
            push token
            token ← next_token()
    else // need to shift, but out of input
        report an error
```

What happens on an error?
- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
Bottom-up Parser

A simple shift-reduce parser:

```plaintext
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A → β
        then // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
    else if (token ≠ EOF)
        then // shift
            push token
            token ← next_token( )
    else // need to shift, but out of input
        report an error
```

What happens on an error?
- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.
Back to $x - 2 * y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
Back to \( x - 2 \times y \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
Back to $x - 2 \times y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id none</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>8,1</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Factor$</td>
<td>- num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>8,1</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Factor</td>
<td>- num * id</td>
<td>6,1</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Term</td>
<td>- num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
Back to $x - 2 \cdot y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
<td>0</td>
</tr>
<tr>
<td>$ id</td>
<td>num * id</td>
<td>8,1</td>
<td>reduce 8</td>
<td>1</td>
</tr>
<tr>
<td>$ Factor$</td>
<td>num * id</td>
<td>6,1</td>
<td>reduce 6</td>
<td>2</td>
</tr>
<tr>
<td>$ Term$</td>
<td>num * id</td>
<td>3,1</td>
<td>reduce 3</td>
<td>3</td>
</tr>
<tr>
<td>$ Expr$</td>
<td>num * id</td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

$Expr$ is not a handle at this point because reducing now will cause backtracking.

While that statement sounds like oracular, we will see that the decision can be automated efficiently.

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
Back to $x - 2 \cdot y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>8,1</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Factor$</td>
<td>- num * id</td>
<td>6,1</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Term$</td>
<td>- num * id</td>
<td>3,1</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$ Expr$</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr -$</td>
<td>num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>8,1</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Factor</td>
<td>- num * id</td>
<td>6,1</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Term</td>
<td>- num * id</td>
<td>3,1</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - num</td>
<td>* id</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Expr - Factor</td>
<td>* id</td>
<td>6,3</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Expr - Term</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term * id</td>
<td></td>
<td>8,5</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Expr - Term * Factor</td>
<td></td>
<td>4,5</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$ Expr - Term</td>
<td></td>
<td>2,3</td>
<td>reduce 2</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>0,1</td>
<td>reduce 0</td>
</tr>
<tr>
<td>$ Goal</td>
<td></td>
<td>none</td>
<td>accept</td>
</tr>
</tbody>
</table>

5 shifts + 9 reduces + 1 accept
Back to $x - 2 \times y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Factor</td>
<td>- num * id</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Term</td>
<td>- num * id</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td>- num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr -</td>
<td>num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - num</td>
<td>* id</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$ Expr - Factor</td>
<td>* id</td>
<td>reduce 6</td>
</tr>
<tr>
<td>$ Expr - Term</td>
<td>* id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term *</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term * id</td>
<td></td>
<td>reduce 8</td>
</tr>
<tr>
<td>$ Expr - Term * Factor</td>
<td></td>
<td>reduce 4</td>
</tr>
<tr>
<td>$ Expr - Term</td>
<td></td>
<td>reduce 2</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>reduce 0</td>
</tr>
<tr>
<td>$ Goal</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>

Corresponding Parse Tree

```
Expr         
  -        
  Term     
  *       
  Fact.   
    <id,x>  
    <num,2>  
```

$Goal$
An Important Lesson about Handles

A handle must be a substring of a sentential form \( \gamma \) such that:

- Must match rhs \( \beta \) of some rule \( A \rightarrow \beta \);

and

- Simply looking for right hand sides that match strings is not good enough.
An Important Lesson about Handles

• **Critical Question**: How can we know when we have found a handle without generating lots of different derivations?
An Important Lesson about Handles

- **Critical Question:** How can we know when we have found a handle without generating lots of different derivations?
  
  - **Answer:** We use left context, encoded in the sentential form, left context encoded in a “parser state”, and a lookahead at the next word in the input. (Formally, 1 word beyond the handle.)
  
  - We build all of this knowledge into a handle-recognizing DFA
LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- The class of grammars that these parsers recognize is called the set of LR(1) grammars

LR(1) means **left-to-right scan** of the input, **rightmost derivation** (in reverse), and **1 word of lookahead**.
LR(1) Parsers

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 \( \gamma_i \).