Bottom-Up Parsing

- More general than top-down parsing
- And just as efficient
- Builds on ideas in top-down parsing
- Preferred method in many instances

Specific algorithm: **LR parsing**
- L means that tokens are read left to right
- R means that it constructs a rightmost derivation
- Donald Knuth (1965)
  "On the Translation of Languages from Left to Right"

The Idea

- An LR parser reduces a string to the start symbol by inverting productions:
  \[
  \text{str} = \text{input string of terminals}
  \]
  
  repeat
  
  - Identify \( \beta \) in \( \text{str} \) such that \( A \rightarrow \beta \) is a production
    (i.e., \( \text{str} = \alpha \beta \gamma \))
  - Replace \( \beta \) by \( A \) in \( \text{str} \)
    (i.e., \( \text{str} \) becomes \( \alpha A \gamma \))
  
  until \( \text{str} = \mathcal{G} \)

An simple example

- LR parsers:
  - Can handle left-recursion
  - Don’t need left factoring

Consider the following grammar:

\[
E \rightarrow E + ( E ) | \text{int}
\]

Is this grammar LL(1) (as shown)?

A Bottom-up Parse in Detail (1)

\[
\text{int} + (\text{int}) + (\text{int})
\]

A Bottom-up Parse in Detail (2)

\[
E \rightarrow (\text{int}) + (\text{int})
\]

\[
E \rightarrow \text{int} + (\text{int}) + (\text{int})
\]
Another example

- Start with input stream
  - "Leaves" of parse tree
- Build up towards goal symbol
  - Called "reducing"
  - Construct the reverse derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential form</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>abcd</td>
</tr>
<tr>
<td>3</td>
<td>aAbcde</td>
</tr>
<tr>
<td>2</td>
<td>aAde</td>
</tr>
<tr>
<td>4</td>
<td>aABde</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G → ABde</td>
</tr>
<tr>
<td>2</td>
<td>A → b</td>
</tr>
<tr>
<td>3</td>
<td>aAde</td>
</tr>
<tr>
<td>4</td>
<td>B → a</td>
</tr>
</tbody>
</table>

Easy?

- Choosing a reduction:
  - Not good enough to simply find production right-hand sides and reduce
  - Example:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>abcd</td>
</tr>
<tr>
<td>3</td>
<td>aAbcde</td>
</tr>
<tr>
<td>2</td>
<td>aAde</td>
</tr>
<tr>
<td>?</td>
<td>...now what?</td>
</tr>
</tbody>
</table>

"aAAbcde" is not part of any sentential form
Key problems

- How do we make this work?
  - How do we know we won’t get stuck?
  - How do we find the next reduction?
  - Also: how do we find it efficiently?

- Key:
  - We are constructing the right-most derivation
  - Grammar is unambiguous
    - Unique right-most derivation for every string
    - Unique production applied at each forward step
    - Unique correct reduction at each backward step

Right-most derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential form</th>
</tr>
</thead>
<tbody>
<tr>
<td>- expr</td>
<td>expr op expr</td>
</tr>
<tr>
<td>1 expr</td>
<td>expr op expr</td>
</tr>
<tr>
<td>3 expr</td>
<td>expr op &lt;id,y&gt;</td>
</tr>
<tr>
<td>6 expr * &lt;id,y&gt;</td>
<td>expr op expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>1 expr * &lt;id,y&gt;</td>
<td>expr op expr * &lt;id,y&gt;</td>
</tr>
<tr>
<td>2 expr op &lt;num,2&gt;</td>
<td>expr op &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>5 expr - &lt;num,2&gt;</td>
<td>expr - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>3 &lt;id,x&gt; - &lt;num,2&gt;</td>
<td>&lt;id,y&gt;</td>
</tr>
</tbody>
</table>

LR parsing

- State of the parser:
  \[ \alpha \mid \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is string of unexamined terminals

- Two operations:
  - **Shift** – read next terminal, push on stack
    \[ E + ( \mid \text{int} ) \rightarrow E + ( \mid \text{int} ) \]
  - **Reduce** – pop RHS symbols off stack, push LHS
    \[ E + (E + (E)) \rightarrow E + (E) \]

Example

1. \( \text{int} + ( \mid \text{int} ) + ( \mid \text{int} ) \) Nothing on stack, get next token
2. \( \text{int} \mid + ( \mid \text{int} ) + ( \mid \text{int} ) \) Shift: push int

Example

1. \( \text{int} + ( \mid \text{int} ) + ( \mid \text{int} ) \) Nothing on stack, get next token
2. \( \text{int} \mid + ( \mid \text{int} ) + ( \mid \text{int} ) \) Shift: push int
3. \( \text{int} \mid + ( \mid \text{int} ) + ( \mid \text{int} ) \) Reduce: pop int
Example

1. `| int + { int } + { int }` Nothing on stack, get next token
2. `| int | + { int } + { int }` Shift: push int
3. `| int | + { int } + { int }` Reduce: pop int, push E
4. `| int | + { int } + { int }` Reduce: pop int, push E
5. `| int | + { int } + { int }` Shift: push +
6. `| int | + { int } + { int }` Shift: push ( 
7. `| int | + { int } + { int }` Shift: push int
8. `| int | + { int } + { int }` Reduce: pop int, push E

Stack

```
E + { E }
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + ( E )
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + ( int )
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + x 5, push E
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + x int
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + x int
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + x int
```

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `| int | + ( int ) + ( int )` Shift: push int
3. `| int | + ( int ) + ( int )` Reduce: pop int, push E
4. `| int | + ( int ) + ( int )` Reduce: pop int, push E
5. `| int | + ( int ) + ( int )` Shift: push +
6. `| int | + ( int ) + ( int )` Shift: push ( 
7. `| int | + ( int ) + ( int )` Shift: push int
8. `| int | + ( int ) + ( int )` Shift: push int

Stack

```
E + x int
```
Example

9. int + ( int ) | + ( int ) \( \text{Reduce: pop x 5, push E} \)
10. int + ( int ) | + ( int ) \( \text{Shift: push +} \)
11. int + ( int ) | + ( int ) \( \text{Shift: push (} \)
12. int + ( int ) | + ( int ) \( \text{Shift: push int} \)
13. int + ( int ) | + ( int ) \( \text{Reduce: pop int, push E} \)
14. int + ( int ) | + ( int ) \( \text{Shift: push)} \)

Stack \( E + ( E ) \)

Example

9. int + ( int ) | + ( int ) \( \text{Reduce: pop x 5, push E} \)
10. int + ( int ) | + ( int ) \( \text{Shift: push +} \)
11. int + ( int ) | + ( int ) \( \text{Shift: push (} \)
12. int + ( int ) | + ( int ) \( \text{Shift: push int} \)
13. int + ( int ) | + ( int ) \( \text{Reduce: pop int, push E} \)
14. int + ( int ) | + ( int ) \( \text{Shift: push)} \)

DONE!

Stack \( E \)

Key problems
1. Will this work?
   How do we know that shifting and reducing using a stack is sufficient to compute the reverse derivation?
2. How do we know when to \textit{shift} and \textit{reduce}?
   Can we efficiently match top symbols on the stack against productions?
   Right-hand sides of productions may have parts in common
   Will shifting a token move us closer to a reduction?
   Are we making progress?
   How do we know when an error occurs?

Why does it work?
1. Right-most derivation
   \[ G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input} \]
   Consider last step:
   To reverse this step:
   Read input until \( g, r, s \) on top of stack
   Reduce \( g, r, s \) to \( B \)

Right-most derivation
1. Could there be an alternative reduction?
   \( G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input} \)
   Production: \( B \rightarrow g, r, s \)
   To reverse this step:
   Read input until \( g, r, s \) on top of stack
   Reduce \( g, r, s \) to \( B \)

No
- Two right-most derivations for the same string
- I.e., the grammar would be ambiguous

Reductions
1. Where is the next reduction?
   \[ a b c B x y z \]
   \[ \gamma_1 \]
   \[ a b f y z \]
   \[ \gamma_2 \]
   Later in the input stream:
   \[ a b c B x y z \]
   \[ \gamma_3 \]
   \[ a b c B x y z \]
   \[ \gamma_4 \]
   Could it be earlier?
   \[ a b c B x y z \]
   \[ \gamma_5 \]
   \[ a b c B x y z \]
   \[ \gamma_6 \]
   \[ a b c B x y z \]

No – this is not the right-most derivation!
Implications

- Cases:
  - Parsing state:
    - Input: a b c q r s x y z
    - Stack: a b c B
  - Key: next reduction must consume top of stack
    - Possibly after shifting some terminal symbols
- How does this help?
  - Can consume terminal symbols in order
  - Never need to search inside the stack

LR parsing

repeat
  - if top symbols on stack match \( \beta \) for some \( A \rightarrow \beta \):
    - Reduce: "found an A"
      - Pop those symbols off
      - Push \( A \) on stack
  - else
    - Get next token from scanner
    - if token is useful:
      - Shift: "still working on something"
      - Push on stack
    - else error
until stack contains goal and no more input

Key problems

- (2) How do we know when to shift or reduce?
  - Shifts
    - Default behavior: shift when there’s no reduction
    - Still need to handle errors
  - Reductions
    - Good news:
      - At any given step, reduction is unique
      - Matching production occurs at top of stack
    - Problem:
      - How to efficiently find the right production

Identifying reductions

- Do viable prefixes have any special properties?
  - Hint:
    - If all available reductions have already been applied, we will no longer have arbitrary counting to perform
    - Example:
      - Production rule
        - \( E \rightarrow E + (E) \ | \ int \)
- Key: viable prefixes are a regular language
- Idea: a DFA that recognizes viable prefixes
  - Input: stack contents
    - (a mix of terminals, non-terminals)
  - Each state represents either
    - A right sentential form – labeled with the reduction to apply
      - A viable prefix – labeled with tokens to expect next

Shift/reduce DFA

- Using the DFA
  - At each parsing step run DFA on stack contents
  - Examine the resulting state \( X \) and the token \( t \) immediately following \( | \) in the input stream
    - If \( X \) has an outgoing edge labeled \( t \), then shift
    - If \( X \) is labeled \( A \rightarrow \beta \) on \( t \), then reduce
- Example:
  - Production rule
    - \( E \rightarrow E + (E) \ | \ int \)
  - First, we’ll look at how to use such a DFA...
For each symbol on the stack, remember the DFA state.
At each step, we rerun the DFA to compute the new state that represents the contents up to that point.

Example

Example

Example

Example

Improvements

- Each DFA state represents stack contents
  - At each step, we rerun the DFA to compute the new state.
  - Can we avoid this?
  - Two actions:
    - Shift: Push a new token
    - Reduce: Pop some symbols off, push a new symbol

- Idea:
  - For each symbol on the stack, remember the DFA state that represents the contents up to that point.
    - Push a new token = go forward in DFA
    - Pop a sequence of symbols = "unwind" DFA to previous state

Example
Algorithm components

- **Stack**
  - String of the form: \( (\text{sym}_1, \text{state}_1) \ldots (\text{sym}_n, \text{state}_n) \)
  - \( \text{sym} \): grammar symbol (left part of string)
  - \( \text{state} \): DFA state
    - Intuitively: represents what we’ve seen so far
    - \( \text{state}_k \) is the final state of the DFA on \( \text{sym}_1 \ldots \text{sym}_k \)
    - And, captures what we’re looking for next
  - Represent as two tables:
    - **action** – whether to shift, reduce, accept, error
    - **goto** – next state

Tables

- **Action**
  - Given state and the next token, \( \text{action}[s, a] = \)  
    - Shift \( s' \), where \( s' \) is the next state on edge \( a \)
    - Reduce by a grammar production \( A \rightarrow \beta \)
    - Accept
    - Error

- **Goto**
  - Given a state and a grammar symbol, \( \text{goto}[s, X] = \)  
    - After reducing an \( X \) production
    - Unwind to state ending with \( X \) (to keep going)

Representing the DFA

- **Combined table:**
  - \( \text{action(state, token)} \)  
  - \( \text{goto} \)

How is the DFA Constructed?

- What’s on the stack?
  - Viable prefix – a piece of a sentential form
    - \( E + ( \) \)
    - \( E + ( \text{int} \) \)
    - \( E + ( E \) \)
  - Idea: we’re part-way through some production
  - Problem: Productions can share pieces
  - DFA state represents the set of candidate productions
  - Represents all the productions we could be working on
  - Notation: \( LR(1) \) item shows where we are and what we need to see
LR Items

- An LR(1) item is a pair:
  \[ [A \to \alpha \bullet \beta, a] \]
  - \( A \to \alpha \beta \) is a production
  - \( a \) is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- \([A \to \alpha \beta, a]\) describes a context of the parser
  - We are trying to find an \( A \) followed by an \( a \), and
  - We have seen an \( \alpha \)
  - We need to see a string derived from \( \beta \)

LR Items

- In context containing
  \[ [E \to E + (E), +] \]
  - If \( "(" \) is next then we can a shift to context containing
  \[ [E \to E + (E), +] \]
- In context containing
  \[ [E \to E + (E), +] \]
  - We can reduce with \( E \to E + (E) \)
  - But only if a \( "+" \) follows

Closure operation

- Observation:
  - At \( A \to \alpha B \beta \) we expect to see \( B \beta \) next
  - Means if \( B \to \gamma \) is a production, then we could see a \( \gamma \)
- Algorithm:
  \[ \text{closure(items)} = \]
  \[ \text{repeat} \]
  \[ \text{for each} [A \to \alpha \bullet B \beta, a] \text{ in items} \]
  \[ \text{for each production} B \to \gamma \]
  \[ \text{for each} b \in \text{FIRST}(Ba) \]
  \[ \text{add} [B \to \bullet \gamma, b] \text{ to items} \]
  \[ \text{until items} \text{ is unchanged} \]

Building the DFA – part 1

- Starting context = \( \text{closure}(S \to \bullet E, \$) \)
  \[ S \to \bullet E, \$ \]
  \[ E \to \bullet E(\$), \$ \]
  \[ E \to \bullet \text{int}, \$ \]
  \[ E \to \bullet E(\$), + \]
  \[ E \to \bullet \text{int}, + \]
- Abbreviated:
  \[ S \to \bullet E, \$ \]
  \[ E \to \bullet E(\$), +/ \]
  \[ E \to \bullet \text{int}, +/ \]
Building the DFA – part 2

- DFA states
  - Each DFA state is a closed set of LR(1) items
  - Start state: \( \text{closure}(\{S \rightarrow \cdot, E, \$\}) \)

- Reductions
  - Label each item \( [A \rightarrow \alpha \beta \cdot, x] \) with "Reduce with \( A \rightarrow \alpha \beta \) on lookahead \( x \)"

- What about transitions?

DFA transitions

- Idea:
  - If the parser was in state \( [A \rightarrow \alpha X \beta] \) and then recognized an instance of \( X \), then the new state is \( [A \rightarrow \alpha X \beta] \)
  - Note: \( X \) could be a terminal or non-terminal

- Algorithm:
  - Given a set of items \( I \) (DFA states) and a symbol \( X \)
    - \( \text{transition}(I, X) = J = {} \)
    - for each \( [A \rightarrow \alpha \cdot X \beta, b] \) \( \in I \)
      - let \( J = \text{transition}(I, X) \)
    - \( T = T + J \)
    - \( E = E + \{ I \rightarrow J \} \)
  - return \( \text{closure}(J) \)

DFA construction

- Data structure:
  - \( T \) – set of states (each state is a set of items)
  - \( E \) – edges of the form \( I \rightarrow J \)
    where \( I, J \in T \) and \( X \) is a terminal or non-terminal

- Algorithm:
  - \( T = \{ \text{closure}(\{S \rightarrow \cdot, Y, \$\}), E = {} \} \)
  - repeat
    - for each state \( I \) in \( T \)
      - for each item \( [A \rightarrow \alpha \cdot X \beta, b] \) \( \in I \)
        - let \( J = \text{transition}(I, X) \)
      - \( T = T + J \)
      - \( E = E + \{ I \rightarrow J \} \)
  - until \( E \) and \( T \) no longer change

Example DFA

To form into tables

- Two tables
  - \( \text{action}(I, \text{token}) \)
  - \( \text{goto}(I, \text{symbol}) \)

- Layout:
  - One row for each states – each \( I \) in \( T \)
  - One columns for each symbol

- Entries:
  - For each edge \( I \rightarrow J \)
    - If \( X \) is a terminal, add shift \( J \) at position \( (I, X) \) in action
    - if \( X \) is a non-terminal, add goto \( J \) at position \( (I, X) \) goto
  - For each state \( [A \rightarrow \alpha \beta \cdot, x] \) \( \in I \)
    - Add reduce \( n \) at position \( (I, x) \) in action (where \( n \) is \( |\text{rhs}| \))

Issues with LR parsers

- What happens if a state contains: \( [X \rightarrow \alpha \cdot \alpha \beta, b] \) and \( [Y \rightarrow \gamma \cdot, a] \)

- Then on input \( a \) we could either
  - Shift into state \( [X \rightarrow \alpha \cdot \alpha \beta, b] \), or
  - Reduce with \( Y \rightarrow \gamma \)

- This is called a \textit{shift-reduce conflict}
  - Typically due to ambiguity
  - Like what?
Shift/Reduce conflicts

- Classic example: the dangling else
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]
- Will have DFA state containing
  \[
  [S \rightarrow \text{if } E \text{ then } S, \text{ else} ] \\
  [S \rightarrow \text{if } E \text{ then } S \text{ else } S, x]
  \]
- Practical solutions:
  - Painful: modify grammar to reflect the precedence of else
  - Many LR parsers default to “shift”

Another example

- Consider the ambiguous grammar
  \[ E \rightarrow E + E \mid E * E \mid \text{int} \]
- Part of the DFA:
  \[
  \begin{align*}
  & E \rightarrow E + E, + \\
  & E \rightarrow E * E, + \\
  \end{align*}
  \]
- We have a shift/reduce on input +
- What do we want to happen?
  - Consider: \( x \ast y + z \)
  - We need to reduce (\( \ast \) binds more tightly than \(+\))

Precedence

- Declare relative precedence
  - Explicitly resolve conflict
  - Tell parser: we prefer the action involving \( \ast \) over \(+\)

- In practice:
  - Parser generators support a precedence declaration for operators
  - What is the alternative?

More...

- Still a problem?
  - Shift/reduce conflict on +
    - Do we care?
    - Maybe: we want left associativity
      - parse: “\( a+b+c \)” as “\(((a+b)+c)\)”
    - Which rule should we choose?
    - Also handled by a declaration “+ is left-associative”

Other problems

- If a DFA state contains both \( [X \rightarrow \alpha \cdot, a] \) and \( [Y \rightarrow \beta \cdot, a] \)
  - What’s the problem here?
  - Two reductions to choose from when next token is \( a \)

- This is called a reduce/reduce conflict
  - Usually a serious ambiguity in the grammar
  - Must be fixed in order to generate parser
  - Think about relationship between \( \alpha \) and \( \beta \)

Reduce/Reduce conflicts

- Example: a sequence of identifiers
  \[ S \rightarrow \varepsilon \mid id \mid id S \]
- There are two parse trees for the string \( id S \rightarrow id S \rightarrow id \)
- How does this confuse the parser?
Reduce/Reduce conflicts

- Consider the DFA states:

  \[
  \begin{align*}
  \{G \rightarrow \epsilon, S \} & \quad \{S \rightarrow \epsilon, S \} \\
  \{S \rightarrow \epsilon, \} & \quad \{S \rightarrow \epsilon, S \} \\
  \{S \rightarrow \epsilon, id, \} & \quad \{S \rightarrow \epsilon, S \} \\
  \{S \rightarrow \epsilon, id S, \} & \quad \{S \rightarrow \epsilon, S \}
  \end{align*}
  \]

- Reduce/reduce conflict on input $\$

  \[
  G \rightarrow S \rightarrow id \\
  G \rightarrow S \rightarrow id S
  \]

  Fix: rewrite the grammar:
  \[
  S \rightarrow \epsilon \mid id S
  \]

Practical issues

- We use an LR parser generator...

  - Question: how many DFA states are there?
  - Does it matter?
  - What does that affect?
    - Parsing time is the same
    - Table size: occupies memory
  - Even simple languages have 1000s of states
    - Most LR parser generators don't construct the DFA as described

LR(1) Parsing tables

- But many states are similar, e.g.

  \[
  \begin{align*}
  E \rightarrow int \cdot, / \mid E \rightarrow int \cdot, + \\
  E \rightarrow int \cdot, $ \mid E \rightarrow int \cdot, +
  \end{align*}
  \]

  - How can we exploit this?
    - Same reduction, different lookahead tokens
    - Idea: merge the states...

The core of a set of LR Items

- When can states be merged?

  - Def: the core of a set of LR items is:
    - Just the production parts of the items
    - Without the lookahead terminals
  - Example: the core of \{ [X \rightarrow \alpha \beta, b], [Y \rightarrow \gamma \delta, d] \}

    \[
    \{ X \rightarrow \alpha \beta, Y \rightarrow \gamma \delta \}
    \]

Merging states

- Consider for example the LR(1) states

  \[
  \begin{align*}
  \{X \rightarrow \alpha \cdot, a \}, [Y \rightarrow \beta \cdot, c \} \\
  \{X \rightarrow \alpha \cdot, b \}, [Y \rightarrow \beta \cdot, d \}
  \end{align*}
  \]

  - They have the same core and can be merged
  - Resulting state is:

    \[
    \{ X \rightarrow \alpha \cdot, a/b \}, [Y \rightarrow \beta \cdot, c/d \}
    \]

  - These are called LALR(1) states
    - Stands for LookAhead LR
    - Typically 10X fewer LALR(1) states than LR(1)

The LALR(1) DFA

- Algorithm:

  \[
  \text{repeat} \\
  \hspace{1cm} \text{Choose two states with same core} \\
  \hspace{1cm} \text{Merge the states by combining the items} \\
  \hspace{1cm} \text{Point edges from predecessors to new state} \\
  \hspace{1cm} \text{New state points to all the previous successors} \\
  \hspace{1cm} \text{until all states have distinct core}
  \]

\[
\begin{array}{c}
A \quad B \quad C \\
D \quad E \quad F
\end{array}
\]
Conversion LR(1) to LALR(1).

LALR states
- Consider the LR(1) states:
  \{ [X \rightarrow \alpha \cdot B \cdot \beta], [Y \rightarrow \beta \cdot \alpha] \}
- And the merged LALR(1) state
  \{ [X \rightarrow \alpha \cdot a/b], [Y \rightarrow \beta \cdot a/b] \}
- What's wrong with this?
  - Introduces a new reduce-reduce conflict
  - In practice such cases are rare

LALR vs. LR Parsing
- LALR is an efficiency hack on LR languages
- Any "reasonable" programming language has a LALR(1) grammar
  - Languages that are not LALR(1) are weird, unnatural languages
- LALR(1) has become a standard for programming languages and for parser generators

Another variation
- Lookahead symbol
  - How is it computed in LR, LALR parser?
  - In closure operation for each \([A \rightarrow \alpha \cdot \beta], \alpha] in items\)
    - for each production \(B \rightarrow \gamma\)
      - for each \(b \in \text{FIRST}(\beta)\)
        - add \([B \rightarrow \gamma], b\] to items
  - Based on context of use
- Simplify this process:
  - What symbol (set of symbols) could I use for \([B \rightarrow \gamma], \beta] ?
  - FOLLOW(B)
  - Called SLR (Simple LR) parser

More power?
- So far:
  - LALR and SLR: reduce size of tables
  - Also reduce space of languages
  - What if I want to expand the space of languages?
- What could I do at a reduce/reduce conflict?
  - Try both reductions!
  - GLR parsing
    - At a choice: split the stack, explore both possibilities
    - If one doesn’t work out, kill it
    - Run-time proportional to "amount of ambiguity"
  - Must design the stack data structure very carefully

General algorithms
- Parsers for full class of context-free grammars
  - Mostly used in linguistics – constructive proof of decidability
  - CYK (1965)
    - Bottom-up dynamic programming algorithm
    - \(O(n^3)\)
  - Earley's algorithm (1970)
    - Top-down dynamic programming algorithm
    - Developed the "•" notation for partial production
    - Worst-case \(O(n^3)\) running time
    - But, \(O(n^2)\) even for unambiguous grammars
  - GLR
    - Worse-case \(O(n^3)\), but \(O(n)\) for unambiguous grammars
LR parsing

Input: $a_1, a_2, \ldots, a_i, a_{i+1}$

Stack: $X_m, X_{m-1}, \ldots, X_1, X_0$

LR Parsing Engine

Action
Goto

Parser Generator

Grammar

Compiler construction

Real world parsers

- Real generated code
  - lex, flex, yacc, bison
- Interaction between lexer and parser
  - C typedef problem
  - Merging two languages
- Debugging
  - Diagnosing reduce/reduce conflicts
  - How to step through an LR parser

Parser generators

- Example: JavaCUP
  - LALR(1) parser generator
  - Input: grammar specification
  - Output: Java classes
  - Action/goto tables
- Separate scanner specification
- Similar tools:
  - SableCC
  - yacc and bison generate C/C++ parsers
  - JavaCC: similar, but generates LL(1) parser

JavaCUP example

- Simple expression grammar
  - Operations over numbers only

```java
// Import generic engine code
import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner. */
init with {
    scanner.init();
};
scan with {
    return scanner.next_token();
};
```

- Note: interface to scanner
- One issue: how to agree on names of the tokens

Example

- Define terminals and non-terminals
  - Indicate operator precedence

```java
/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UNMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;
/* Non terminals */
non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;
/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
```

Example

- Grammar rules

```java
expr_list ::= expr_list expr_part |
            expr_part;
expr_part ::= expr SEMI;
expr ::= expr PLUS expr |
       expr MINUS expr |
       expr TIMES expr |
       expr DIVIDE expr |
       expr MOD expr |
       LPAREN expr RPAREN |
       NUMBER;
```
Summary of parsing

- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

A Hierarchy of Grammar Classes

![A Hierarchy of Grammar Classes](image)

From Andrew Appel, "Modern Compiler Implementation in Java"