Building an LR(0) automaton

Definition: An *item* is a production with a dot (•) somewhere on the RHS (which indicates a partially completed rule)

How to construct the automaton:

- make the start state the first rule, with the dot (•) in front of the left-most symbol of the RHS
- for each state, label an arc with the symbol that follows • and advance the • one position to the right in the next state.
- If the • precedes a non-terminal (e.g., A) add all productions with that non-terminal A on the LHS to the current state, with the • in the leftmost position
A sample construction of the DFA

Small example CFG:

1. $S' \rightarrow \vdash E \vdash$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow \text{id}$
Using the automaton

- For each input token
  - Start in the start state
  - Read the stack (from the bottom up) and read the current input, and do the action indicated for the current input
    - If there is a transition out of our current state on the current input, then shift (push) that input onto the stack
    - We know we can reduce if the current state has only one item and the • is the rightmost symbol
    - To reduce, pop the RHS off the stack, reread the stack (from the bottom-up), follow the transition for the LHS and push the LHS onto the stack
  - Accept if $S'$ on the stack when all input is read
Using the transducer

Example input: $\langle id+id+id \rangle$

<table>
<thead>
<tr>
<th>Stack</th>
<th>States visited</th>
<th>Input read</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>1</td>
<td>$\epsilon$</td>
<td>$\langle id+id+id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>1 2</td>
<td>$\vdash$</td>
<td>$\langle id+id+id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash id$</td>
<td>1 2 6</td>
<td>$\vdash id$</td>
<td>$\langle +id+id \rangle$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash T$</td>
<td>1 2 5</td>
<td>$\vdash id$</td>
<td>$\langle +id+id \rangle$</td>
<td>reduce $E \rightarrow T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3</td>
<td>$\vdash id$</td>
<td>$\langle +id+id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E+$</td>
<td>1 2 3 7</td>
<td>$\vdash id+$</td>
<td>$\langle id+id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E+id$</td>
<td>1 2 3 7 6</td>
<td>$\vdash id+id$</td>
<td>$\langle +id \rangle$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash E+T$</td>
<td>1 2 3 7 8</td>
<td>$\vdash id+id$</td>
<td>$\langle +id \rangle$</td>
<td>reduce $E \rightarrow E+T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3 8</td>
<td>$\vdash id+id$</td>
<td>$\langle +id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E+$</td>
<td>1 2 3 7</td>
<td>$\vdash id+id+$</td>
<td>$\langle id \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E+id$</td>
<td>1 2 3 7 6</td>
<td>$\vdash id+id+id$</td>
<td>$\langle \rangle$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash E+T$</td>
<td>1 2 3 7 8</td>
<td>$\vdash id+id+id$</td>
<td>$\langle \rangle$</td>
<td>reduce $E \rightarrow E+T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3 8</td>
<td>$\vdash id+id+id$</td>
<td>$\langle \rangle$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E-$</td>
<td>1 2 3 4</td>
<td>$\vdash id+id+id-$</td>
<td>$\epsilon$</td>
<td>reduce $S' \rightarrow \vdash E-$</td>
</tr>
<tr>
<td>$S'$</td>
<td>1</td>
<td>$\vdash id+id+id-$</td>
<td>$\epsilon$</td>
<td>accept</td>
</tr>
</tbody>
</table>
What can go wrong?

Two distinct problems:

Problem 1: What if the state looks like this?

\[
\begin{align*}
A &\rightarrow \alpha \cdot c \beta \\
B &\rightarrow \gamma \cdot
\end{align*}
\]

Do we try to shift the next character (as suggested by \(A \rightarrow \alpha \cdot c \beta\)) or do we reduce by \(B \rightarrow \gamma \cdot\) (as suggested by \(B \rightarrow \gamma \cdot\))?

This is known as a *shift-reduce conflict*.

Problem 2: What if the state looks like this?

\[
\begin{align*}
A &\rightarrow \alpha \cdot \\
B &\rightarrow \beta \cdot
\end{align*}
\]

Do we reduce by \(A \rightarrow \alpha\) or by \(B \rightarrow \beta\)?

This is known as a *reduce-reduce conflict*.

If any item \(A \rightarrow \alpha \cdot\) occurs in a state in which it is not alone, then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0).
Example with conflicts

Consider right-associative expressions. Modify our grammar slightly to allow (reverse RHS of second-rule).

1. \( S' \rightarrow \vdash E \\vdash \)
2. \( E \rightarrow T + E \)
3. \( E \rightarrow T \)
4. \( T \rightarrow \text{id} \)

DFA:
Parsing with conflicts

Suppose we are parsing a string that looks like \( \vdash \text{id} \ldots \)

Picture of the stack:

Question: Should we reduce \( E \rightarrow T \)?
Answer: It depends.
  ▶ If input is \( \vdash \text{id} \vdash \), then yes.
  ▶ If input is \( \vdash \text{id} + \ldots \), then no.
Looking ahead

If we add a lookahead token to the automaton, we can fix the conflict.

For each $A \rightarrow \alpha \bullet$, attach $\text{Follow}(A)$.

For our grammar:

$$\text{Follow}(E) =$$

$$\text{Follow}(T) =$$

Consider our conflicting state:

$$E \rightarrow T \bullet$$

$$E \rightarrow T \bullet + E$$

Interpretation: A reduce action $A \rightarrow \alpha \bullet F$ (where $F$ is the $\text{Follow}(A)$) applies only if the next character is in $F$. 

8
When we add this one character of lookahead, we have an SLR(1) (Simple LR with 1 character of lookahead) parser. SLR(1) resolves many, but not all, conflicts.

- LR(1) parsing is more sophisticated than SLR(1) parsers
- LR(1) parses strictly more grammars
- LR(1) automaton is more complex
- LR(1) and SLR(1) are identical as parsing algorithms: the only difference is in the respective automaton they create

There is also a parser called LALR(1) (lookahead LR(1)), which falls between SLR(1) and LR(1).
- this is what Yacc and Bison use
Making this more efficient

Current running time of this algorithm:

Instead of scanning the stack each time...

Start the transducer in....

Running time:
Outputting a derivation

- Easy: each time we do a reduction, output the rule
- But, this isn't quite right. Derivations should start with the start symbol. Bottom-up parsing doesn’t.
A simple observation

- Didn’t we say that this was LR(1) parsing?
- Doesn’t the “R” mean rightmost derivation?
- Aren’t we always reducing the leftmost nonterminal?
- But notice the direction we are creating the derivation. Write the derivation in reverse.
Outputting the parse tree

Algorithm
▶ Create a “tree stack”
▶ Each time we reduce, pop the right hand side nodes from tree stack
▶ Push the left hand side node and make its children the nodes we just popped
▶ Example:
How the tree is actually built in LR parsing
How the tree is actually built in LL parsing
Assignment hints

Note that the automaton in cfg-r format specifies:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>shift/reduce</th>
<th>next state / production number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(terminal (for shifts) / (for reductions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or non-terminal)</td>
</tr>
</tbody>
</table>

P1, P2: write a cfg-r derivation by hand

P3: Given a DFA, a state and one input, follow one transition in the DFA.

P4: Write a parser
  ▶ Read a CFG, the DFA and input
  ▶ Output cfg-r (derivation)

P5: Write a parser for WLP4
  ▶ Your parser will read tokens, build a parse tree and output a left-most derivation
  ▶ Find a way to embed the WLP4 grammar and DFA table in your program.
Going back

Looking at: \( L = \{a^n b^m : n \geq m \geq 0\} \) (non-LL(1) language)

1. \( S' \rightarrow \vdash S \vdash \)
2. \( S \rightarrow a \ S \)
3. \( S \rightarrow T \)
4. \( T \rightarrow a \ T \ b \)
5. \( T \rightarrow \epsilon \)

What is this really saying?
What the DFA means

- When you see an “‘a’”, shift an “‘a’”
- When you see a “‘b’”, reduce by rule 5 (but only the first time, after that there will already be a T), shift the “‘b’”, then reduce by rule 4.
- When you see EOF, reduce by rule 3, then by rule 2 until the stack contains \( \vdash S \), then shift \( \dashv \), then reduce by rule 1.

There’s a simpler stack algorithm:

- When you see an “‘a’”, shift an “‘a’”
- When you see a “‘b’” pop an “‘a’” and pair that with the “‘b’” (rule 4).
- When you see \( \dashv \) the rest of the “‘a’” on the stack are made by rule 2 instead.

But the LR(1) is pretty close, and was derived mechanically from the grammar. No thinking required!
Final fun facts

- Theorem: For any augmented LR(1) grammar, there is an equivalent LR(0) grammar.
- Theorem: For any LR(k) grammar, there is an equivalent LR(1) grammar (with up to $k!$ times as many non-terminals).
- Theorem: The class of languages that can be parsed deterministically with a stack can be represented with an LR(1) grammar.
- Comparing LL(1) vs. LR(1)