Example Grammar

\[ V = \{ \text{expr}, \text{op} \} \quad \Sigma = \{ \text{ID}, + \} \quad P = \{ \text{expr} \rightarrow \text{ID}, \quad \text{expr} \rightarrow \text{expr op expr}, \quad \text{op} \rightarrow + \} \quad S = \text{expr} \]

Top-down parsing

**Algorithm 1** Generic algorithm

\[ \delta \leftarrow S \]

\[ \text{while } \delta \neq \text{input do} \]

\[ \text{choose any } A \text{ such that } \delta = \alpha A \beta \]

\[ \text{oracle chooses } \gamma \text{ such that } A \rightarrow \gamma \in P \text{ or rejects} \]

\[ \delta \leftarrow \alpha \gamma \beta \]

\[ \text{end while} \]

**Algorithm 2** Generic algorithm (left-canonical)

\[ \delta \leftarrow S \]

\[ \text{while } \delta \neq \text{input do} \]

\[ \text{choose any } A \text{ such that } \delta = x A \beta \]

\[ \text{oracle chooses } \gamma \text{ such that } A \rightarrow \gamma \in P \text{ or rejects} \]

\[ \delta \leftarrow x \gamma \beta \]

\[ \text{end while} \]

**Definition:** Given a context-free grammar \( G = (V, \Sigma, P, S) \), the corresponding augmented grammar is \( G' = (V \cup \{ S' \}, \Sigma \cup \{ \top, \bot \}, P \cup \{ S' \rightarrow \top S \bot \}, S') \), formed by adding a new production \( S' \rightarrow \top S \bot \). \( \top \) and \( \bot \) are special symbols that denote the beginning and end of input.

**Invariant:** Consumed input + Stack = \( \top \delta \bot \)

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Action</th>
<th>Consumed input</th>
<th>Stack</th>
<th>Remaining input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta )</td>
<td>expr</td>
<td>initialize</td>
<td>( \top ) expr ( \bot )</td>
<td>( \top ) ID + ID ( \bot )</td>
</tr>
<tr>
<td>expr</td>
<td>read ( \top )</td>
<td>expr ( \bot )</td>
<td>ID + ID ( \bot )</td>
<td></td>
</tr>
<tr>
<td>expr op expr</td>
<td>expand expr \rightarrow expr op expr</td>
<td>( \top ) expr op expr ( \bot )</td>
<td>ID + ID ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID op expr</td>
<td>expand expr \rightarrow ID</td>
<td>( \top ) ID op expr ( \bot )</td>
<td>ID + ID ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID op expr</td>
<td>read ID</td>
<td>( \top ) ID</td>
<td>ID ( \bot ) + op expr ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID + expr</td>
<td>expand op \rightarrow +</td>
<td>( \top ) ID +</td>
<td>ID ( \bot ) + expr ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID + expr</td>
<td>read +</td>
<td>( \top ) ID +</td>
<td>ID ( \bot ) + expr ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID + ID</td>
<td>expand expr \rightarrow ID</td>
<td>( \top ) ID +</td>
<td>ID ( \bot ) + ID ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID + ID</td>
<td>read ID</td>
<td>( \top ) ID + ID ( \bot )</td>
<td>ID ( \bot ) + ID ( \bot )</td>
<td></td>
</tr>
<tr>
<td>ID + ID</td>
<td>read ( \top )</td>
<td></td>
<td>ID ( \bot ) + ID ( \bot )</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 3 Stack-based top-down algorithm (augmented input)

```
push ⊥
push S
push ⊥
for each $a$ in $\sqsubset \text{input} \sqsupset$ from left to right do
  while top of stack is $A \in V$ do
    pop $A$
    oracle chooses $A \rightarrow \gamma \in P$ or rejects
    push the symbols in $\gamma$ (from right to left)
  end while
  reject if top of stack $\neq a$
  pop $a$
end for
accept (stack is necessarily empty)
```

LL(1) pre-computation:
First($\gamma$) = $\{b \mid \gamma \Rightarrow^* b\beta \text{ for some } \beta\}$
Follow($A$) = $\{c \mid S' \Rightarrow^* \alpha Ac\beta \text{ for some } \alpha, \beta\}$
Predict($A, a$) = $\{A \rightarrow \gamma \mid a \in \text{First}(\gamma) \text{ or } (\gamma \Rightarrow^* \epsilon \text{ and } a \in \text{Follow}(A))\}$

Definition: A grammar is LL(1) if $|\text{Predict}(A, a)| \leq 1$ for all $A, a$.

Algorithm 4 LL(1) algorithm

```
push ⊥
push S
push ⊥
for each $a$ in $\sqsubset \text{input} \sqsupset$ from left to right do
  while top of stack is $A \in V$ do
    pop $A$
    find $A \rightarrow \gamma$ in Predict[$A, a$] or reject
    push the symbols in $\gamma$ (from right to left)
  end while
  reject if top of stack $\neq a$
  pop $a$
end for
accept (stack is necessarily empty)
```

Bottom-up parsing

Algorithm 5 Generic algorithm

```
$\delta \leftarrow \text{input}$
while $\delta \neq S$ do
  oracle chooses $A \rightarrow \gamma$ such that $\delta = \alpha\gamma\beta$ or rejects
  $\delta \leftarrow \alpha A\beta$
end while
```

Algorithm 6 Generic algorithm (right-canonical)

```
$\delta \leftarrow \text{input}$
while $\delta \neq S$ do
  oracle chooses $A \rightarrow \gamma$ such that $\delta = \alpha\gamma x$ or rejects
  $\delta \leftarrow \alpha Ax$
end while
```
Invariant: Stack + Remaining input = ⊢ δ ⊣

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Action</th>
<th>Stack</th>
<th>Remaining input</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ</td>
<td>initialize</td>
<td>⊢</td>
<td>ID + ID ⊣</td>
</tr>
<tr>
<td>ID + ID</td>
<td>shift ID</td>
<td>⊢ ID</td>
<td>+ ID ⊣</td>
</tr>
<tr>
<td>expr + ID</td>
<td>reduce expr → ID</td>
<td>⊢ expr</td>
<td>+ ID ⊣</td>
</tr>
<tr>
<td>expr + ID</td>
<td>shift +</td>
<td>⊢ expr+</td>
<td>ID ⊣</td>
</tr>
<tr>
<td>expr op ID</td>
<td>reduce op → +</td>
<td>⊢ expr op</td>
<td>ID ⊣</td>
</tr>
<tr>
<td>expr op ID</td>
<td>shift ID</td>
<td>⊢ expr op ID</td>
<td>⊣ -</td>
</tr>
<tr>
<td>expr op expr</td>
<td>reduce expr → ID</td>
<td>⊢ expr op expr</td>
<td>⊣ -</td>
</tr>
<tr>
<td>expr</td>
<td>reduce expr → expr op expr</td>
<td>⊢ expr</td>
<td>⊣ -</td>
</tr>
<tr>
<td>expr</td>
<td>shift ⊣</td>
<td>⊢</td>
<td>expr ⊣</td>
</tr>
</tbody>
</table>

Algorithm 7 Stack-based bottom-up algorithm (augmented input)

push ⊢
for each symbol a in input ⊣ from left to right do
  while oracle says “Reduce A → γ” do
    pop |γ| times (the symbols in γ from right to left)
    push A
  end while
  reject if oracle says so
  push a
end for
accept (stack is necessarily ⊢ S ⊣)

Definition: It may be that for any given contents of the stack and value of a, the oracle always says the same thing. That is, there may exist a function Reduce : (Σ′ ∪ V′)∗ → P ∪ {shift} such that Reduce(stack a) returns A → γ if and only if the oracle says “Reduce A → γ” and a function Reject : (Σ′ ∪ V′)∗ → {true, false} that returns true if and only if the oracle says “Reject.” When this is the case, we say the grammar is LR(1). When the grammar is LR(1), we can replace the oracle with the Reduce and Reject functions.

Algorithm 8 LR(1) algorithm (abstract)

push ⊢
for each symbol a in input ⊣ from left to right do
  while Reduce(stack + a) is some production A → γ do
    pop |γ| times (the symbols in γ from right to left)
    push A
  end while
  reject if Reject(stack + a)
  push a
end for
accept (stack is necessarily ⊢ S ⊣)

Theorem (Knuth, 1965): For any LR(1) grammar the set \{stack + a | Reject(stack + a)\} is a regular language.

Corollary: For any LR(1) grammar, for each A → γ ∈ P, the set \(R_{A→γ} = \{\text{stack } a \mid \text{Reduce(stack } a) = A \to γ\}\) is a regular language.

Thus, there exists a finite transducer (DFA with output) that can replace the oracle. Knuth gives an algorithm to build one. The resulting transducer has a transition function Trans (which takes a state and a terminal or non-terminal, and returns the next state), and an output function Reduce (which takes a state and a terminal, and returns either nothing or a production to reduce by). As in DFAs, the Trans function is partial in that it may not be defined for a given input. In this case, the DFA enters an implicit error state, and the input string is rejected. Thus a single transducer implements the oracle functions Reduce and Reject.
Algorithm 9 LR(1) algorithm (concrete, quadratic time)

push ⊢
for each symbol a in input ⊢ from left to right do
  loop
    state ← q₀
    for each symbol X in stack from left to right (bottom to top) do
      state ← Trans[state, X]
    end for
    if Reduce[state,a] is some production A → γ then
      pop |γ| times (the symbols in γ from right to left)
      push A
    else
      exit loop
    end if
  end loop
reject if state = ERROR
push a
end for
accept (stack is necessarily ⊢ S ⊢)

Algorithm 10 LR(1) algorithm (concrete, linear time)

symStack.push ⊢
stateStack.push Trans[q₀, ⊢]
for each symbol a in input ⊢ from left to right do
  while Reduce[stateStack.top, a] is some production A → γ do
    symStack.pop |γ| times
    stateStack.pop |γ| times
    symStack.push A
    stateStack.push Trans[stateStack.top, A]
  end while
  symStack.push a
reject if Trans[stateStack.top, a] = ERROR
stateStack.push Trans[stateStack.top, a]
end for
accept (symStack is necessarily ⊢ S ⊢)

Algorithm 11 LR(1) algorithm (with tree building)

nodeStack.push leafnode(⊢)
stateStack.push Trans[q₀, ⊢]
for each symbol a in input ⊢ from left to right do
  while Reduce[stateStack.top,a] is some production A → γ do
    nodeStack.pop |γ| child nodes (right end first)
    stateStack.pop |γ| times
    nodeStack.push treenode(A, child nodes)
    stateStack.push Trans[stateStack.top, A]
  end while
  nodeStack.push leafnode(a)
reject if Trans[stateStack.top, a] = ERROR
stateStack.push Trans[stateStack.top, a]
end for
accept (nodeStack is necessarily leafnode(⊢) treenode(S,...) leafnode(⊣))