Example Grammar

$$V = \{\text{expr}, \text{op}\} \qquad \Sigma = \{\text{ID}, +\} \qquad P = \{\text{expr} \to \text{ID}, \quad \text{expr} \to \text{expr op expr}, \quad \text{op} \to +\} \qquad S = \text{expr}$$

Top-down parsing

Algorithm 1 Generic algorithm

$\delta \leftarrow S$
while $\delta \neq \mathit{input}$ **do**
  choose any A st $\delta = \alpha A \beta$
  oracle chooses γ st $A \to \gamma \in P$ or rejects
  $\delta \leftarrow \alpha\gamma\beta$
end while

Algorithm 2 Generic algorithm (left-canonical)

$\delta \leftarrow S$
while $\delta \neq \mathit{input}$ **do**
  choose any A st $\delta = x A \beta$
  oracle chooses γ st $A \to \gamma \in P$ or rejects
  $\delta \leftarrow x\gamma\beta$
end while

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

Invariant: Consumed input + Stack = $\vdash \delta \dashv$

Derivation δ	Action	Consumed input	Stack top bottom	Remaining input
expr	initialize		$\vdash$ expr $\dashv$	$\vdash$ ID + ID $\dashv$
expr	read $\vdash$	$\vdash$	expr $\dashv$	ID + ID $\dashv$
expr op expr	expand expr $\to$ expr op expr	$\vdash$	expr op expr $\dashv$	ID + ID $\dashv$
ID op expr	expand expr $\to$ ID	$\vdash$	ID op expr $\dashv$	$\vdash$ ID + ID $\dashv$
ID op expr	read ID	$\vdash$ ID	op expr $\dashv$	+ ID $\dashv$
ID + expr	expand op $\to$ +	$\vdash$ ID	+ expr $\dashv$	+ ID $\dashv$
ID + expr	read +	$\vdash$ ID +	expr $\dashv$	ID $\dashv$
ID + ID	expand expr $\to$ ID	$\vdash$ ID +	ID $\dashv$	ID $\dashv$
ID + ID	read ID	$\vdash$ ID + ID	$\dashv$	$\dashv$
ID + ID	read $\dashv$	$\vdash$ ID + ID $\dashv$		

Algorithm 3 Stack-based top-down algorithm (augmented input)

```
push ⊣
push S
push ⊢
for each a in ⊢ input ⊣ from left to right do
  while top of stack is A ∈ V do
    pop A
    oracle chooses A → γ ∈ P or rejects
    push the symbols in γ (from right to left)
  end while
  reject if top of stack ≠ a
  pop a
end for
accept (stack is necessarily empty)
```

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

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

Algorithm 4 LL(1) algorithm

```
push ⊣
push S
push ⊢
for each a in ⊢ input ⊣ from left to right do
  while top of stack is A ∈ V do
    pop A
    find A → γ in Predict[A,a] or reject
    push the symbols in γ (from right to left)
  end while
  reject if top of stack ≠ a
  pop a
end for
accept (stack is necessarily empty)
```

Bottom-up parsing

Algorithm 5 Generic algorithm

```
δ ← input
while δ ≠ S do
  oracle chooses A → γ st δ = αγβ or rejects
  δ ← αAβ
end while
```

Algorithm 6 Generic algorithm (right-canonical)

```
δ ← input
while δ ≠ S do
  oracle chooses A → γ st δ = αγx or rejects
  δ ← αAx
end while
```

Invariant: Stack + Remaining input = $\vdash \delta \dashv$

Derivation δ	Action	Stack bottom top	Remaining input
ID + ID	initialize	$\vdash$	ID + ID $\dashv$
ID + ID	shift ID	$\vdash$ ID	+ ID $\dashv$
expr + ID	reduce expr $\to$ ID	$\vdash$ expr	+ ID $\dashv$
expr + ID	shift +	$\vdash$ expr +	ID $\dashv$
expr op ID	reduce op $\to$ +	$\vdash$ expr op	ID $\dashv$
expr op ID	shift ID	$\vdash$ expr op ID	$\dashv$
expr op expr	reduce expr $\to$ ID	$\vdash$ expr op expr	$\dashv$
expr	reduce expr $\to$ expr op expr	$\vdash$ expr	$\dashv$
expr	shift $\dashv$	$\vdash$ expr $\dashv$	

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 : $(\Sigma' \cup V')^* \to P \cup \{\text{shift}\}$ such that Reduce(stack a) returns $A \to \gamma$ if and only if the oracle says "Reduce $A \to \gamma$" and a function Reject : $(\Sigma' \cup V')^* \to \{true, false\}$ 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 $\{\text{stack} + a \mid \text{Reject}(\text{stack} + a)\}$ is a regular language.

Corollary : For any LR(1) grammar, for each $A \to \gamma \in P$, the set
$R_{A\to\gamma} = \{\text{stack } a \mid \text{Reduce}(\text{stack } a) = A \to \gamma\}$ 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+a 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(⊣))
```