Bottom Up (Shift/Reduce) Parsing
**Bottom Up Parsing** has the following advantages over top-down parsing.

Attribute computation is easy.

Since choices are made only at the end of a rule, shared prefixes are unproblematic. Because of this, there is usually no need to modify grammar rules.

The parser can be generated automatically.

One big disadvantage is the fact that bottom-up parsing does not support left/right information flow. (For example, checking symbol definitions.)
Shift/Reduce Parsing

Let $\mathcal{G} = (\Sigma, A, R, S')$ be an attribute grammar.

The shift/reduce parser operates on triples $(s, v, u) \in (\Sigma \otimes S)^* \times (\Sigma \otimes S)^* \times (\Sigma \otimes S)^*$, where

- $s \in (\Sigma \otimes A)^*$ is the stack.
- $v \in (\Sigma \otimes A)^*$ is the lookahead,
- $u \in (\Sigma \otimes A)^*$ is the input that is not yet read.
Shift/Reduce Parsing

We write $\vdash$ for the transition relation of the parser.

The parser starts in a state of form $(\epsilon, \epsilon, u)$.

(Empty stack, empty lookahead, no input read.)
Read

A read means that the parser moves one unread token to the lookahead:

\[(s, v, (\sigma, \alpha) \cdot u) \vdash (s, v \cdot (\sigma, \alpha), u).\]
Shift

A shift means that the parser shifts one token from lookahead to the stack:

\[(s, (\sigma, \alpha) \cdot v, u) \vdash (s \cdot (\sigma, \alpha), v, u).\]
Reduction

A **reduction** means that the parser replaces the right hand side of a grammar rule by the left hand side. It uses the attribute function of the grammar rule to compute the new attribute.

If $(A \rightarrow w_1 \cdot \ldots \cdot w_n) : f \in R$, then

$$(s \cdot (w_1, \alpha_1) \cdot \ldots \cdot (w_n, \alpha_n), v, u) \vdash (s \cdot (A, f(\alpha_1, \ldots, \alpha_n)), v, u).$$

Reductions can only be made at the top of the stack!
Accept

The shift/reduce parser accepts its input if it is in a state

\(( (S, \alpha), \epsilon, \epsilon) \).

This means that it has read all the input, has empty lookahead, and it managed to rewrite the input to \( S \).

In practice an EOF symbol is used. Let \( \# \not\in \Sigma \) be a special EOF symbol.

The shift/reduce parser accepts its input if it is in a state

\(( (S, \alpha), \#, \epsilon) \).
Making the Decisions

At each state, the parser has the following choices:

- If the top of the stack contains the right hand side of a rule, it can reduce.
- It it didn’t reach end of file, it can shift.

It is possible that more than one reduction is possible. If a reduction is possible, it is still possible to shift. In order to decide, the parser uses the lookahead.

A good parser makes its decisions as early as possible, that means with the smallest possible lookahead.

We will only consider parsers that use a lookahead of at most 1.
Parser Generation Tools/Practical Aspects

There exist many parser generation tools that support attribute grammars. (Yacc, Bison, Maploon). The attribute functions are usually represent by general C/C++ -statements. In the code, $1, 2, 3, ...$ refer to the attributes of the first, second, etc. token on the right hand side.

The notation $$ refers to the attribute of the token on the left hand side.

A rule of form $A \rightarrow A + B : f(x, y, z) = x + z$ is represented by:

\[
A \rightarrow A + B \quad // \quad $$ = $1 + $3;
\]
**LALR parsing**

LALR stands for *look ahead left right*. It is a technique for deciding when reductions have to be made in shift/reduce parsing. Often, it can make the decisions without using a look ahead. Sometimes, a look ahead of 1 is required.

Most parser generators (and in particular Bison and Yacc) construct LALR parsers.

In LALR parsing, a deterministic finite automaton is used for determining when reductions have to be made. The deterministic finite automaton is usually called *prefix automaton*. On the following slides, I will explain how it is constructed.
Items

Let $G = (\Sigma, R, S)$ be a context-free grammar.

**Definition** Let $A \in \Sigma$, $w_1, w_2 \in \Sigma^*$. If $A \rightarrow w_1 \cdot w_2 \in R$, then $A \rightarrow w_1 \cdot w_2$ is called an item.

An item is a rule with a dot added somewhere in the right hand side.

The intuitive meaning of an item $A \rightarrow w_1 \cdot w_2$ is that $w_1$ has been read, and if $w_2$ will also be read, then the rule $A \rightarrow w_1 w_2$ can be reduced.
Items

Let $a \rightarrow bBc$ be a rule. The following items can be constructed from this rule:

$$a \rightarrow . bBc, \quad a \rightarrow b . Bc, \quad a \rightarrow bB . c, \quad a \rightarrow bBc .$$

For a given grammar $G$, the set of possible items is always finite.
Operations on Itemsets (1)

Definition: An itemset is a set of items.

Because for a given grammar, there exists only a finite set of possible items, the set of itemsets is also finite.

Let $I$ be an itemset. The closure $\text{CLOS}(I)$ of $I$ is defined as the smallest itemset $J$, s.t.

- $I \subseteq J$,
- If $A \rightarrow w_1 . B w_2 \in J$, and there exists a rule $B \rightarrow v \in R$, then $B \rightarrow . v \in J$. 
Operations on Itemsets (2)

Let $I$ be an itemset, let $\alpha \in \Sigma$ be a symbol. The set $\text{TRANS}(I, \alpha)$ is defined as

$$\{ A \rightarrow w_1 \alpha \cdot w_2 \mid A \rightarrow w_1 \cdot \alpha w_2 \in I \}.$$
The Prefix Automaton

Let $G = (\Sigma, R, S)$ be a grammar. The prefix automaton of $G$ is the deterministic finite automaton $A = (\Sigma, Q, Q_s, Q_a, \delta)$, that is the result of the following algorithm:

- Start with $A = (\Sigma, \{\text{CLOS}(I)\}, \{\text{CLOS}(I)\}, \emptyset, \emptyset)$, where $I = \{\hat{S} \rightarrow .S \#\}$, $\hat{S} \not\in \Sigma$ is a new start symbol, $S$ is the original start symbol of $G$, and $\# \not\in \Sigma$ is the EOF symbol.

- As long as there exist an $I \in Q$ and an $A \in \Sigma$, s.t. $I' = \text{CLOS}(\text{TRANS}(I, A)) \not\in Q$, put
  \[ Q := Q \cup \{I'\}, \quad \delta := \delta \cup \{(I, A, I')\}. \]

- As long as there exist $I, I' \in Q$, and an $A \in \Sigma$, s.t. $I' = \text{CLOS}(\text{TRANS}(I, A))$, and $(I, A, I') \not\in \delta$, put
  \[ \delta := \delta \cup \{(I, A, I')\}. \]
The Prefix Automaton (2)

The prefix automaton may be big, but it can be easily computed. Every context-free language has a prefix automaton, but not every language can be parsed by an LALR parser, because of the lookahead sets.

**Theorem:** Let $G = (\Sigma, R, S)$ be a context-free grammar. Let $L$ be its associated language, i.e. $L = \{w \in \Sigma^* | S \Rightarrow^* w\}$. Let $L'$ be the language defined by

$$\{w \in \Sigma^* | \exists w' \in \Sigma^* : ww' \in L\}.$$ 

Then the language $L'$ is regular.

**proof.** It follows from the construction of the prefix automaton on the previous slides.
Parse Algorithm (1)

```
std::vector< state > states;
    // Stack of states of the prefix automaton.

std::vector< token > tokens;
    // We assume that a token has attributes, so
    // we don’t encode them separately.

std::dequeue< token > lookahead;
    // Will never be longer than one.

states. push_back( q0 ); // The initial state.

while( true )
{
```
Parse Algorithm (2)

decision = unknown;

state topstate = states. back();
if(topstate has only one reduction R and no shifts)
    decision = reduce(R);

// We know for sure that we need lookahead:

if( decision == unknown && lookahead. size( ) == 0 )
{
    lookahead. push_back( inputstream. readtoken( ) );
}
Parse Algorithm (3)

if( lookahead.front() == EOF ) {
    if( topstate is an accepting state )
        return tokens.back();
    else
        return error, unexpected end of input.
}
Parse Algorithm (4)

if( decision == unknown &&
   topstate has only one reduction R with
   lookahead. front( ) &&
   no shift is possible with lookahead. front( ))
{
    decision = reduce(R);
}

if( decision == unknown &&
   topstate has only a shift Q with
   lookahead. front( ) &&
   no reduction is possible with lookahead. front( ))
{
    decision = shift(Q);
}
Parse Algorithm (5)

if( decision == unknown )
{
    // Either we have a conflict, or the parser is
    // stuck.

    if( no reduction/no shift is possible )
        print error message, try to recover.
Parse Algorithm (6)

// A conflict can be shift/reduce, or
// reduce/reduce:

Let R, from the set of possible reductions,
(taking into account lookahead. front( )),
be the rule with the smallest number.

decision = reduce(R);
}
Parse Algorithm (7)

if( decision == push(Q))
{
    states. push_back( Q );
    tokens. push_back( lookahead. front( ));
    lookahead. pop_front( );
}
else
{
    // decision has form reduce(R)

    unsigned int n =
        the length of the rhs of R.
Parse Algorithm (8)

token lhs =
    compute_lhs( R,
        tokens. begin( ) + tokens. size( ) - n,
        tokens. begin( ) + tokens. size( ));
    // this also computes the attribute.

for( unsigned int i = 0; i < n; ++ i )
{
    states. pop_back( );
    tokens. pop_back( );
}
Parse Algorithm (9)

// The shift of the lhs after a reduction is
// usually called ’goto’

topstate = states. back( );
state newstate =
    the state reachable from topstate under lhs.

states. push_back( newstate );
tokens. push_back( lhs );
}
}

// Unreachable.
Lookahead Sets

We already have seen lookahead sets in action.

If a state has more than one reduction, or a reduction and a shift, the parser looks at the lookahead symbol, in order to decide what to do next.

\[ LA(I, A \rightarrow w) \subseteq \Sigma \] is defined as a set of tokens. If the parser is in state \( I \), and the lookahead \( \sigma \in LA(I, A \rightarrow w) \), then the parser can reduce \( A \rightarrow w \).

When should a token \( \sigma \) be in \( LA(I, A \rightarrow w) \)?
Lookahead Sets (2)

Definition:

\[ s \in \text{LA}(I, \ A \rightarrow w) \text{ if} \]

1. \( A \rightarrow w \) \( \in \) \( I \) (obvious)

2. There exists a correct input word \( w_1 \ s \ w_2 \ # \), such that

3. The parser reaches a state with state stack \( (\ldots, I) \) and token stack \( (\ldots, w) \), the lookahead (of the parser) is \( s \), and

4. the parser can reduce the rule \( A \rightarrow w \), after which

5. it can read the rest of the input \( w_2 \) and reach an accepting state.
Computing Look Ahead Sets

For every rule \( A \rightarrow w \) of the grammar \( G \), such that there exist states \( I_1, I_2, I_3 \), s.t. \( A \rightarrow . \ w \in I_1, \ A \rightarrow w . \in I_2 \), there exists a path from \( I_1 \) to \( I_2 \) in the prefix automaton that reads \( w \), and there is a transition from \( I_1 \) to \( I_3 \) that reads \( A \), the following must hold:

- For every symbol \( \sigma \in \Sigma \), for which a transition from \( I_3 \) to some other state is possible in the prefix automaton, \( \sigma \in \text{LA}( I_2, \ A \rightarrow w . ) \).

- For every item of form \( B \rightarrow v . \in I_3 \), \( \text{LA}( I_3, \ B \rightarrow v . ) \subseteq \text{LA}( I_2, \ A \rightarrow w . ) \)

Compute the LA as the smallest such sets.
Computing Look Ahead Sets (2)

Example

\[ S \to Aa, \]
\[ A \to B, \]
\[ A \to Bb, \]
\[ B \to C, \]
\[ B \to Cc, \]
\[ C \to d. \]
The algorithm on the previous slides can sometimes compute too big look ahead sets. You will see this in the exercises.
Computing the Lookahead Sets in the Correct Way

Definition: Let $G = (\Sigma, R, S)$ be a grammar. An LR(1)-item (based on $G$) is an object of form $A \rightarrow w_1 \cdot w_2 / s$, where $(A \rightarrow w_1 w_2) \in R$, and $s \in \Sigma$ is a terminal symbol of $G$.

A LR(1)-item set is a set of LR(1)-items.

The intuitive meaning of $A \rightarrow w_1 \cdot w_2 / s$ is something like: ‘We have read $w_1$, and are prepared to read $w_2 \cdot s$ after that’.
Closure of LR(1)-Itemsets

Let $I$ be an LR(1)-itemset. The closure CLOS($I$) of $I$ is defined as the smallest LR(1)-itemset $J$, s.t.

- $I \subseteq J$,

- If $A \rightarrow w_1 . Bw_2/s \in J$, and there exists a rule $B \rightarrow v \in R$, then for each terminal symbol $s' \in \text{FIRST}(w_2s)$, also $B \rightarrow . v/s' \in J$.

(FIRST is defined in the slides on top-down parsing.)
Transitions of LR(1)-Itemsets

Let \( I \) be an LR(1)-itemset, let \( \alpha \in \Sigma \) be a symbol. \( \text{TRANS}(I, \alpha) \) is defined as

\[
\{ A \rightarrow w_1 \alpha \cdot w_2/s \mid A \rightarrow w_1 \cdot \alpha w_2/s \in I \}.
\]
Core of an LR(1)-Itemset

Let $I$ be an LR(1)-itemset. The core of $I$, written as $\text{CORE}(I)$ is defined as

$$\{A \rightarrow w_1 . w_2 \mid \exists s \in \Sigma : A \rightarrow w_1 . w_2/s \in I\}.$$

(The set of LR(0)-items that one obtains when one removes all the lookaheads.)
Construction of the Prefix Automaton with LR(1)-Items

Let \( \mathcal{G} = (\Sigma, R, S) \) be a grammar. The prefix automaton of \( \mathcal{G} \) is the deterministic finite automaton \( \mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta) \), that is the result of the following algorithm:

- Start with \( \mathcal{A} = (\Sigma, \{CLOS(I)\}, \{CLOS(I)\}, \emptyset, \emptyset) \), where \( I = \{\hat{S} \rightarrow . S/#\} \), \( \hat{S} \not\in \Sigma \) is a new start symbol, \( S \) is the original start symbol of \( \mathcal{G} \), and \( # \not\in \Sigma \) is the EOF symbol.
• As long as there exist an $I \in Q$ and an $A \in \Sigma$, s.t. $I' = \text{CLOS}(\text{TRANS}(I, A))$, and there is no state $I'' \in Q$ with $\text{CORE}(I'') = \text{CORE}(I')$, set

$$Q := Q \cup \{I'\}, \quad \delta := \delta \cup \{(I, A, I')\}.$$ 

• As long as there exist $I, I' \in Q$, and an $A \in \Sigma$, s.t. $\text{CORE}(I') = \text{CORE}(\text{CLOS}(\text{TRANS}(I, A)))$, and either
  1. $(I, A, I') \not\in \delta$, or
  2. $I' \neq I$,

set

$$\begin{cases} I' := I' \cup \text{CLOS}(\text{TRANS}(I, A)), \\ \delta := \delta \cup \{(I, A, I')\}. \end{cases}$$

(Formally, one must define a predicate between automata, and construct the fixed point of this predicate. It would be unpleasant.)
Once the prefix automaton $A = (\Sigma, Q, Q_s, Q_a, \delta)$ has been constructed, the lookahead sets can be obtained from the LR(1)-items as follows:

If a state $I$ contains items of form $A \to w/s'$, the lookahead set for reducing $A \to w$ equals

$$\{ s' \in \Sigma \mid A \to w/s' \in I \}.$$
The construction on the previous slides is carried out automatically by parser generators. Examples are YACC, Bison, and also Maphoon.

Using a parser generator, it is easier to extend the language later. Also, the parser generator automatically analyzes the language, and shows where the conflicts are.

Top-Down parsing (recursive descend) has the advantage that one doesn’t need to study a tool, but it will be a lot harder to change the language later. Developers often avoid use of a parser generator, and then regret later, when they have to change the language.