Recursive Descent Parsing Algorithm - top down parsing.

- The parse tree is constructed
  - From the top
  - From left to right

- Terminals are seen in order of appearance in the token stream:
  \[ t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9 \]
• Consider the grammar
  \[ E \rightarrow T \mid T + E \]
  \[ T \rightarrow \text{int} \mid \text{int} \times T \mid (E) \]

• Token stream is: \((\text{int}_5)\)

• Start with top-level non-terminal \(E\)
  — Try the rules for \(E\) in order
$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} \times T \mid (E)$

Mismatch: int does not match (Backtrack ...)

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} \times T \mid (E)$

Mismatch: int does not match (Backtrack ...)
\[ E \rightarrow T | T + E \]
\[ T \rightarrow \text{int} | \text{int} \ast T | (E) \]

**Diagram:**

```
       E
      /|
     / \
   T   E
    /   \
   (   E)
     \   
      \  \  
       \ int
       \  \  
        \  \  
         \ int 
```

**Note:**

- Match! Advance input.
E → T | T + E
T → int | int * T | ( E )

Match! Advance input.

( int₅ )

E → T | T + E
T → int | int * T | ( E )

End of input, accept.
• Let TOKEN be the type of tokens
   — Special tokens INT, OPEN, CLOSE, PLUS, TIMES

• Let the global next point to the next input token

• Define boolean functions that check for a match of:
  — A given token terminal

```c
bool term(TOKEN tok) { return *next ++ == tok; }
```
advances next, returns boolean
- The nth production of S: \( S_n() \) { ... }
- Try all productions of S: \( S() \) { ... }

- For production \( E \rightarrow T \) \( E_1() \) { return T(); }
- For production \( E \rightarrow T + E \) \( E_2() \) { return T() && term(PLUS) && E(); }

&& evaluates arguments in left to right order

these advance next
\[
\begin{align*}
E & \rightarrow T \mid T + E \\
T & \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
\end{align*}
\]

- For all productions of E (with backtracking)
  
  ```
  bool E() {
    TOKEN *save = next;
    return (next = save, E_1())
    || (next = save, E_2());
  }
  ```

  || if first branch succeeds, do not bother with second branch

  backtracking

  if they all fail, the higher level will do the backtracking
\[ E \rightarrow T | T + E \]
\[ T \rightarrow \text{int} | \text{int} \ast T | (E) \]

- Functions for non-terminal T

```c
bool T_1() { return term(INT); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {
    TOKEN *save = next;
    return (next = save, T_1())
         || (next = save, T_2())
         || (next = save, T_3()); }
```
Once a non-terminal succeeds, there is no way to try another production. 

Example productions:

\[ E \rightarrow T \mid T + E \]

\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E() { TOKEN *save = next; return (next = save, E1()) || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OP) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T1()) || (next = save, T2()) || (next = save, T3()); }

Int + int will be rejected.
• If a production for non-terminal X succeeds
  – Cannot backtrack to try a different production for X later

• General recursive-descent algorithms support such “full” backtracking
  – Can implement any grammar
• Presented recursive descent algorithm is not general
  – But is easy to implement by hand

• Sufficient for grammars where for any non-terminal at most one production can succeed

• The example grammar can be rewritten to work with the presented algorithm
  – By left factoring
In the formal language theory of computer science, left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right).

- Consider a production $S \rightarrow S \alpha$

```cpp
bool S_1() { return S() && term(a); }
bool S() { return S_1(); }
```

- A left-recursive grammar has a non-terminal $S$

$$S \rightarrow^* S \alpha \text{ for some } \alpha$$

- Recursive descent does not work in such cases
• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha | \beta \]

\[ S \rightarrow S \alpha \rightarrow S \alpha \alpha \rightarrow S \alpha \alpha \alpha \rightarrow \cdots \rightarrow S \alpha \cdots \alpha \rightarrow \beta \alpha \cdots \alpha \]

• \( S \) generates all strings starting with a \( \beta \) and followed by any number of \( \alpha \)'s.

zero or more

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]

\[ S' \rightarrow \alpha S' | \varepsilon \]
• In general

\[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]

• All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \)
and continue with several instances of \( \alpha_1, \ldots, \alpha_n \)

zero or more

• Rewrite as

\[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]
\[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon \]
• The grammar
  \[ S \rightarrow A \alpha \mid \delta \]
  \[ A \rightarrow S \beta \]
  is also left-recursive because
  \[ S \rightarrow^+ S \beta \alpha \]

• This left-recursion can also be eliminated

• Recursive descent
  – Simple and general parsing strategy
  – Left-recursion must be eliminated first
  – ... but that can be done automatically

• Used in production compilers
  – E.g., gcc
Predictive Parsers

• Like recursive-descent but parser can “predict” which production to use
  – By looking at the next few tokens
  – No backtracking

• Predictive parsers accept LL(k) grammars
  \[ \text{left-to-right} \rightarrow k \text{ tokens lookahead} \]
  \[ \text{left-most derivation} \]
• In recursive descent,
  – At each step, many choices of production to use
  – Backtracking used to undo bad choices

• In LL(1),
  – At each step, only one choice of production
Left factoring is removing the common left factor that appears in two productions of the same non-terminal.

It is done to avoid back-tracing by the parser.

Suppose the parser has a look-ahead consider this example

\[ A \rightarrow qB \mid qC \]

where \( A, B, C \) are non-terminals and \( q \) is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace.

After left factoring, the grammar is converted to

\[ A \rightarrow qD \]

\[ D \rightarrow B \mid C \]

In this case, a parser with a look-ahead will always choose the right production.
• Recall the grammar
  \[ E \rightarrow T + E | T \]
  \[ T \rightarrow \text{int} | \text{int} \ast T | (E) \]

• Hard to predict because
  – For \( T \) two productions start with \text{int}
  – For \( E \) it is not clear how to predict

• We need to left-factor the grammar
  \[ E \rightarrow T \ast X \]
  \[ X \rightarrow + E | \epsilon \]
  \[ T \rightarrow \text{int} \ast Y | (E) \]
  \[ Y \rightarrow \ast T | \epsilon \]
• Left-factored grammar
  \[ E \rightarrow T X \]
  \[ X \rightarrow + E \mid \epsilon \]
  \[ T \rightarrow ( E ) \mid \text{int} \ Y \]
  \[ Y \rightarrow * T \mid \epsilon \]

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+ E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td>* T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **next input token**
- **leftmost non-terminal**
- **rhs of production to use**

• Consider the [E, int] entry
  “When current non-terminal is E and next input is int, use production \( E \rightarrow T X \)”. 

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+ E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td>* T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Consider the \([Y,+]\) entry
  
  – “When current non-terminal is \(Y\) and current token is +, get rid of \(Y\)”
  
  – \(Y\) can be followed by + only if \(Y \rightarrow \varepsilon\)

\[
\begin{array}{|c|c|c|c|c|}
\hline
 & int & * & + & ( & \varepsilon & \varepsilon \\
\hline
E & TX & & & TX & & \\
X & \varepsilon & & +E & \varepsilon & \varepsilon \\
T & int Y & & \varepsilon & (E) & \varepsilon \\
Y & \varepsilon & * T & \varepsilon & \varepsilon & \varepsilon \\
\hline
\end{array}
\]

• Consider the \([E,*]\) entry
  
  – “There is no way to derive a string starting with * from non-terminal \(E\)”

\[
\begin{array}{|c|c|c|c|c|}
\hline
 & int & * & + & ( & \varepsilon & \varepsilon \\
\hline
E & TX & & & TX & & \\
X & \varepsilon & & +E & \varepsilon & \varepsilon \\
T & int Y & & \varepsilon & (E) & \varepsilon \\
Y & \varepsilon & * T & \varepsilon & \varepsilon & \varepsilon \\
\hline
\end{array}
\]
• Method similar to recursive descent, except
  – For the leftmost non-terminal $S$
  – We look at the next input token $a$
  – And choose the production shown at $[S,a]$

• A stack records frontier of parse tree
  – Non-terminals that have yet to be expanded
  – Terminals that have yet to matched against the input
  – Top of stack = leftmost pending terminal or non-terminal

• Reject on reaching error state
• Accept on end of input & empty stack
initialize stack = <$ > and next
repeat
    case stack of
        <X, rest>  : if T[X,*next] = Y₁...Yₙ
                     then stack ← <Y₁... Yₙ rest>;
                     else error ();
        <t, rest>  : if t == *next ++
                     then stack ← <rest>;
                     else error ();
    until stack == < >

terminal on top of stack
matches input, pop and advance
input
• Left-factored grammar

\[ E \rightarrow TX \]
\[ X \rightarrow +E \mid \varepsilon \]
\[ T \rightarrow (E) \mid \text{int } Y \]
\[ Y \rightarrow *T \mid \varepsilon \]

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+E</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*T</td>
<td>\varepsilon</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>

- **next input token**
- **leftmost non-terminal**
- **rhs of production to use**

Stack | Input      | Action
------|------------|--------
E $    | int * int $ | TX
TX $   | int * int $ | int Y
• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>int Y</td>
<td></td>
<td>+ E</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td>* T</td>
<td></td>
<td>( E )</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td>ε</td>
<td></td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

leftmost non-terminal
d next input token

rhs of production to use

int Y X $  int * int $  terminal

Y X $  * int $  * T

* T X $  * int $  terminal

T X $  int $  int Y

3071
• Left-factored grammar

\[
\begin{align*}
E & \rightarrow TX \\
X & \rightarrow +E \mid \varepsilon \\
T & \rightarrow (E) \mid \text{int} \ Y \\
Y & \rightarrow *T \mid \varepsilon
\end{align*}
\]

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+E</td>
<td></td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*T</td>
<td></td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>

- "When current non-terminal is Y and current token is +, get rid of Y"
• Left-factored grammar

$E \rightarrow T \ X$
$X \rightarrow + \ E \mid \varepsilon$
$T \rightarrow ( \ E ) \mid \text{int} \ Y$
$Y \rightarrow * \ T \mid \varepsilon$

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int</td>
<td></td>
<td>+E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>int</td>
<td>*T</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
</tr>
</tbody>
</table>

leftmost non-terminal  
next input token
rhs of production to use

E $X$ int $\varepsilon$

$\varepsilon$

ACCEPT