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:
  \[ t_2 \ t_5 \ t_6 \ t_8 \ t_9 \]
• Consider the grammar
  \[ E \rightarrow T \mid T + E \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid ( E ) \]

• Token stream is: \( ( \text{int}_5 ) \)

• Start with top-level non-terminal \( E \)
  – Try the rules for \( E \) in order
Mismatch: int does not match (Backtrack ...)
E → T | T + E
T → int | int * T | ( E )

Match! Advance input.

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

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

Match! Advance input.

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

End of input, accept.
Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.
• 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 $S$: \( \text{bool } S_n() \{ \ldots \} \)

Try all productions of $S$: \( \text{bool } S() \{ \ldots \} \)

- For production $E \rightarrow T$

  \( \text{bool } E_1() \{ \text{return } T(); \} \)

- For production $E \rightarrow T + E$

  \( \text{bool } E_2() \{ \text{return } T() \&\& \text{term(PLUS)} \&\& \text{E();} \} \)

\&\& \text{evaluates arguments in left to right order}

these advance next
• For all productions of E (with backtracking)
  
  ```
  bool E() {
    TOKEN *save = next;
    return (next = save, E1())
      || (next = save, E2()); }
  ```

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

  backtracking

  if they all fail, the higher level will do the backtracking
E → T | T + E
T → int | int * T | ( E )

- Functions for non-terminal T
  
  bool T₁() { return term(INT); }
  bool T₂() { return term(INT) && term(TIMES) && T(); }
  bool T₃() { return term(OPEN) && E() && term(CLOSE); }

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

A limitation of recursive descent

Once a non-terminal succeeds, no way to try another production.

int * int will be rejected

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

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(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T1())
| | (next = save, T2())
| | (next = save, T3()); }
• 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 $S \rightarrow S \alpha$
  
  ```c
  bool S1() { return S() && term(a); }
  
  bool S() { return S1(); }
  ```

- A left-recursive grammar has a non-terminal $S$
  
  $$S \rightarrow^+ S\alpha \quad \text{for some } \alpha$$

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

\[ S \rightarrow S \alpha \rightarrow S \alpha \alpha \rightarrow S \alpha \alpha \alpha \rightarrow \ldots \rightarrow S \alpha \ldots \alpha \rightarrow \beta \alpha \ldots \alpha \]

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

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

\[ S' \rightarrow \alpha S' \mid \varepsilon \]

\[ S \rightarrow \beta S' \rightarrow \beta \alpha S' \rightarrow \beta \alpha \alpha S' \rightarrow \ldots \rightarrow \beta \alpha \ldots \alpha S' \rightarrow \beta \alpha \ldots \alpha \]
• In general
  \[ S \rightarrow S \alpha_1 | ... | S \alpha_n | \beta_1 | ... | \beta_m \]

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

• Rewrite as
  \[
  S \rightarrow \beta_1 S' | ... | \beta_m S' \\
  S' \rightarrow \alpha_1 S' | ... | \alpha_n S' | \varepsilon
  \]
• The grammar
  \[ S \rightarrow A \alpha | \delta \]
  \[ A \rightarrow S \beta \]
  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
A deterministic model of computation is a model of computation such that the successive states of the machine and the operations to be performed are completely determined by the preceding state.
LL(1) grammars cannot be left recursive since the leftmost nonterminal is the same as the LHS. This would result in infinite recursion.

• 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

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

• Hard to predict because
  – For T two productions start with int
  – For E it is not clear how to predict

• We need to left-factor the grammar

\[
E \rightarrow TX \\
T \rightarrow \text{int} \mid Y \mid (E) \\
Y \rightarrow \ast T \mid \epsilon
\]
• Left-factored grammar

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

• The LL(1) parsing table:

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

<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></td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>T</td>
<td>int (Y)</td>
<td></td>
<td></td>
<td>(E)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>

• Consider the \([E,*]\) entry

- “There is no way to derive a string starting with \(*\) from non-terminal \(E\)”

<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></td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>T</td>
<td>int (Y)</td>
<td></td>
<td></td>
<td>(E)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>
• 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 = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y_1...Y_n
      then stack ← <Y_1... Y_n rest>;
      else error ();
    <t, rest> : if t == *next ++
      then stack ← <rest>;
      else error ();
  until stack == <>
• **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>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td>TX</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+E</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>(E)</td>
<td>ε</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>*T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

- **Stack**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>TX</td>
</tr>
<tr>
<td>TX $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
</tbody>
</table>

- **next input token**

- **leftmost non-terminal**

- **rhs of production to use**
### 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>int</td>
<td><strong>TX</strong></td>
<td></td>
<td><strong>TX</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>int Y</td>
<td></td>
<td><strong>+ E</strong></td>
<td><strong>ε</strong></td>
<td><strong>ε</strong></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>*** T**</td>
<td><strong>ε</strong></td>
<td><strong>ε</strong></td>
<td><strong>ε</strong></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td><strong>( E )</strong></td>
<td><strong>ε</strong></td>
<td><strong>ε</strong></td>
</tr>
</tbody>
</table>

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

- **int Y X $**
- **int * int $**
- **Y X $**
- *** int $**
- *** T**
- *** T X $**
- *** int $**
- **terminal**
- **int X Y**
- **int + int**
- **T X $**
- **int $**
- **int Y**

- **E**
- **T X**
- **Y**
- **int Y**
- **int int**
- *** int**
- **Y X**
- **T X**
- **int**
- **int Y**

- **E**
- **T X**
- **Y**
- **int Y**
- **int**
- **int int**
- *** int**
- **Y X**
- **T X**
- **int**
- **int Y**
• Left-factored grammar
  \[ E \rightarrow TX \]
  \[ T \rightarrow (E) | \text{int } Y \]
  \[ X \rightarrow +E \mid \varepsilon \]
  \[ Y \rightarrow *T \mid \varepsilon \]

• The LL(1) parsing table:

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

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

int Y X $    int $    terminal

Y X $    $    $    \varepsilon
X $    $    \varepsilon

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

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

- The LL(1) parsing table:

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

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

```
x$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
```

```
E
<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>
```

- ACCEPT