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: ( int_{5} )

• Start with top-level non-terminal \( E \) — Try the rules for \( E \) in order
E → T | T + E
T → int | int * T | ( E )

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

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

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

(int₅)

(int₅)
\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

**Diagram:**

```
( (int5) )
```

**Match! Advance input.**

```
E  
|   
T  
( E )
```

**Diagram:**

```
( int2 )
```

```
E  
|   
T  
( E )
```

**Match! Advance input.**
$$E \rightarrow T \mid T + E$$

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

---

**Match! Advance input.**

- $$(\text{int}_5)$$

---

**End of input, accept.**

- $$(\text{int}_5)$$
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.

```c
/* recognize tokens for the calculator and print them out */

enum vytokentype {
    NUMBER = 258,
    ADD = 259,
    SUB = 260,
    MUL = 261,
    DIV = 262,
    ABS = 263,
    EOL = 264 /* end of line */
};
```
• 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

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

  advances next, returns boolean
The nth production of S:

bool $S_n()$ { ... }

Try all productions of S:

bool $S()$ { ... }

For production $E \rightarrow T$

bool $E_1()$ { return $T();$ }

For production $E \rightarrow T + E$

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

&& evaluates arguments in left to right order

these advance next
• For all productions of E (with backtracking)

```c
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
\[ E \rightarrow T \mid T + E \]
\[ T \rightarrow \text{int} \mid \text{int} \times T \mid (E) \]

- Functions for non-terminal T
  ```
  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, \( T_3() \));
  }
  ```
A limitation of recursive descent

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

int * int will be rejected

once a non terminal succeeds
no way to try another production
• 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 $S \rightarrow S \alpha$
  
  ```
  bool S_1() { return S() && term(a); }
  bool S() { return S_1(); }
  ```

- A left-recursive grammar has a non-terminal $S$
  
  $S \rightarrow^* S \alpha$ for some $\alpha$

- Recursive descent does not work in such cases
• Consider the left-recursive grammar

\[ S \rightarrow S \alpha | \beta \]

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

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

• Can rewrite using right-recursion

\[ S \rightarrow \beta S' \]

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

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

zero or more
• 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 \( \alpha_1, ..., \alpha_n \)

zero or more

• Rewrite as

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

• 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.

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
  
  left-to-right → k tokens lookahead
  left-most derivation

always k=1
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 \text{int} \n  
  – For \( E \) it is not clear how to predict

• We need to left-factor the grammar

\[ E \rightarrow TX \]

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

\[ Y \rightarrow \ast T \mid \epsilon \]

\[ X \rightarrow + E \mid \epsilon \]
- **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></td>
<td>+E</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td>+E</td>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>int Y</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 $E \rightarrow TX$"
• 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>(TX)</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></td>
<td>(\varepsilon)</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>

• Consider the \([E,\ast]\) entry
  
  - “There is no way to derive a string starting with \(\ast\) 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>(TX)</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></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

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

• 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>+E</td>
<td>+E</td>
<td>+E</td>
<td>+E</td>
<td>+E</td>
<td>+E</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
</tr>
<tr>
<td>Y</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
<td>*T</td>
<td>*T</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>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td>*T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

leftmost non-terminal

rhs of production to use

int Y X $  int * int $  terminal

Y X $  * int $  * T

*TX $  * int $  terminal

TX $  int $  int Y

E

I

Y

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

• 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>T X</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></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

int Y X $ int $ terminal

Y X $ $ $ ε
X $ $ $ ε

"When current non-terminal is Y and current token is +, get rid of Y"
• 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></td>
<td></td>
<td>+E</td>
<td>TX</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>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

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

\[ E \rightarrow X \rightarrow Y \rightarrow \varepsilon \]

\[ \text{int } \rightarrow \text{int } \rightarrow \text{int } \rightarrow \varepsilon \]

\[ \text{ACCEPT} \]