Recursive Descent Parsing (Top-down)

The parse tree is constructed (a) from the top, and (b) from left to right

Tokens are seen in order of appearence in the token stream

Example token stream (stream of terminals): t2 t5 t6 t8 t9

```
1 --> t2 3 9
3 --> 4 7
4 --> t5 t6
7 --> t8 t9
```

Consider the grammar

```
E --> T | T + E
T --> int | int * T | (E)
```

Token stream is: (int5)

Start with top-level non-terminal E (hence called top-down). Try rules for E *in order*. Walk through the input with an arrow left to right. Also try rules in order. As long as we are generating non-terminals, we do not know if we are on the right path or not. However if we hit a terminal, we can see if that matches our current token or not. If it does not, we can back-track and try another option (in order). In this case we go to T and then down to int, backtrack, then int * T, then backtrack, and then (E), and here '(' matches! So we might be on the right track, and we now need to advance the input pointer, and now we have to expand the non-terminal E. Again we will start with the first production T. And we will eventually match the input string with some production, and we are done.

General algorithm for Recursive Descent Parsing

Let TOKEN be the type of the tokens: e.g., INT, OPEN, CLOSE, PLUS, TIMES, ... Let `next` point to the next token in the input string (the arrow).

Define boolean functions that check for a match of:

- A given token terminal

```
      bool term(TOKEN, tok) { return *next++ == tok;  }//next pointer is incremented regardless!
Try the nth production of non-terminal S
bool Sn() { ... } //will succeed if input matches the nth production of S
Try all productions of S
bool S() { ... } //will succeed if input matches any production of S
```

Running on our example, functions for non-terminal E

- For production `E --> T`

```
bool E1() { return T(); }
```

- For production `E --> T + E`

```
bool E2() { return T() AND term(PLUS) AND E(); }
```

- For all productions of `E` (with backtracking)

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

  The first rule that returns true will cause a return (we will not evaluate the remaining rules due to semantics of logical OR ||)
  `next` pointer needs to be saved for backtracking

Functions for non-terminal T

- For production `T --> int`

```
bool T1() { return term(INT); }
```

- For production `E --> int * T`

```
bool T2() { return term(INT) AND term(TIMES) AND T(); }
```

- For production `E --> (E)`

```
bool T3() { return term(OPEN) AND E() AND term(CLOSE); }
```

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

The first rule that returns true will cause a return (we will not evaluate the remaining rules due to semantics of logical OR ||)
`next` pointer needs to be saved for backtracking

To start the parser:

1. Initialize `next` to first token
2. Invoke E()
   - Show full execution of E() on the example string

Can implement by hand, e.g., TinyCC

Left Recursion : the main problem with Recursive Descent Parsing

Consider

```
S --> S a
```

```
bool S1() { return S() AND term(a); }
bool S() { return S1(); }
```

This will go into an infinite loop for parsing *any* input.

The problem is that this grammar is left recursive

```
S -->+ S \alpha    for some \alpha
```

If for some sequence of productions we end up with left recursion, recursive descent parsing will just not work

Consider the left-recursive grammar

```
S --> S a | b
```

This generates a string starting with `b` with any number of `a`s following it. It produces the sentences right-to-left, and that's why it does not work with recursive descent parsing (which works left-to-right).

We can generate exactly the same language by producing strings left to right, which is also a solution to our problem.

```
S --> b S'
S' --> a S' | \epsilon
```

In general, for any left recursive grammar

```
S --> S a1 | S a2 | ... | S an | b1 | b2 | ... | bm
```

All strings starting with one of b1..bm and continue with several instances of a1..an
Rewrite as

```
S --> b1 S' | b2 S' | ... | bm S'
S' --> a1 S' | a2 S' | ... | an S' | \epsilon
```

There are more general ways of encoding left recursion in a grammar

```
S --> A a | d
A --> S b
```

is also left-recursive because

```
S -->+ S b a
```

This left recursion can also be eliminated (see Dragon book for general algorithm).

Recursive descent

- Simple and general parsing strategy
- Left-recursion must be eliminated first
  - This can be done automatically
  - In practice however, this is done manually. The reason is that we also need to specify semantic actions with the productions used. Hence, people do elimination of left-recursion on their own, and this is not difficult to do.
- Popular strategy in production compilers. e.g., gcc's parser is a hand-written recursive-descent parser.

Predictive Parsing

- Like recursive-descent but parser can "predict" which production to use
  - by looking at the next few tokens
  - without backtracking
- Predictive parsing accepts LL(k) grammars
  - Left-to-right scanning of tokens
  - Leftmost derivation
  - k tokens to lookahead. In practice, k = 1
- In LL(1), at every step there is at most one choice for a possible production
  - At each step, only one choice of production

```
wAb --> w \alpha b , next input: token t//only one choice for A at every step
```

Example

```
E --> T + E | T
T --> int | int * T | (E)
```

Here, with one lookahead, hard to predict because:

- we cannot choose the first two productions of T by one lookahead
- the same problem exists with E, because two productions start with non-terminal T. Here it is a non-terminal, but still there is an issue, because we do not know which production to use for one lookahead

Left factoring a grammar

Basic idea: Factor out common prefix into a single production

```
E --> TX
X --> + E | \epsilon
```

Left factoring delays the decision on which production to use. Here we decide which production to use *after* we have seen `T`.

Similarly,

```
T --> int Y | (E)
Y --> \epsilon | * T
```

Left-factored grammar

E --> TX X --> +E | \epsilon T --> int Y | (E) Y --> \epsilon | * T

LL(1) parsing table (assume it is somehow given, later we will discuss how to construct this):

```
     int     *     +     (     )          $
E    TX                TX
X                +E          \epsilon   \epsilon
T    int Y             (E)
Y           *T   \epsilon     \epsilon   \epsilon
```

Empty entries represent error states

Algorithm: similar to recursive descent, except

- For the left-most non-terminal S, we look at the next input token a
- Choose the production rule shown at (S, a)

Instead of using recursive functions, maintain a stack of the frontier of the parse tree. The stack contains all the entries on the right of the top of the frontier. In other words, the top of the stack is the leftmost pending terminal or non-terminal. Terminals in the stack are yet to be matched in the input. Non-terminals in the stack are yet to be expanded.

Reject on reaching error state.

Accept on end of input or empty stack.

Pseudo-code:

```
initialize stack = <S $> 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 == < >
```

Run this algorithm on the example

Constructing LL(1) Parsing Tables

Constructing **First sets**

Consider a non-terminal A, production A --> \alpha, and a token `t`

T(A, t) = \alpha in two cases:

1. If \alpha -->* t \beta
    - If \alpha can derive t in the first position
    - We say that t \in First(\alpha)
2. If \alpha -->* \epsilon and S -->* \beta A t \delta
    - Useful if stack has A, input is t, and A cannot derive t
    - In this case, the only option is to get rid of A by deriving \epsilon
        - Can work only if t can follow A in at least one derivation
    - We say that t \in Follow(\alpha)

Definition:

```
First(X) = {t | X -->* t \alpha} U {\epsilon | X -->* \epsilon}
```

Algorithm sketch:

1. First(t) = {t}
2. \epsilon \in First(X)
    - if X --> \epsilon
    - if X --> A1A2..An AND \epsilon \in First(Ai) for all i
3. First(\alpha) \subset First(X) if X --> A1..An \alpha and \epsilon \in First(Ai) for all i

Example

```
E --> TX
X --> +E | \epsilon
T --> int Y | (E)
Y --> \epsilon | * T
```

First set for terminals

```
First(t) = {t}
First(*) = {*}
....
```

First set for non-terminals

```
First(E) \superset First(T)
First(T) = {'(', 'int')}
First(X) = {+, \epsilon}
First(Y) = {*, \epsilon}
```

Constructing **Follow sets**

Definition

```
Follow(X) = {t | S -->* \beta X t \delta}
```

Intuition

- if `X --> A B, then First(B) \subset Follow(A) and Follow(X) \subset Follow(B)`
- if `B --> \epsilon`, Follow(X) \subset Follow(A)
- If S is the start symbol, then $ \in Follow(S)

Algorithm sketch:

1. $ \in Follow(S)
2. For each A --> \alpha X \beta,
    - First(\beta) - \epsilon \subset Follow(X)
3. For each A --> \alpha X \beta
    - Follow(A) \subset Follow(X) if \epsilon \in First(\beta)

Example

```
E --> TX
X --> +E | \epsilon
T --> int Y | (E)
Y --> \epsilon | * T
```

Running the algo

```
Follow(E) = {$, )}
Follow(E) \superset Follow(X)
Follow(X) \superset Follow(E)
(in other words Follow(X) = Follow(E))
```

Hence

```
Follow(X) = {$, )}
```

Looking at T, and the first production E --> TX

```
Follow(T) \superset First(X)
or
Follow(T) = {+} (ignoring \epsilon)
Follow(T) \superset Follow(E)    because X -->* \epsilon
Follow(T) = {+, $}
```

Looking at the second rule,

```
Follow(Y) \superset Follow(T)
Follow(T) \superset Follow(Y)
```

Follow(Y) = {+, $,)}

Looking at terminals

```
Follow('(') \superset First(E)
Follow('(') = {(, int}
Follow(')') \superset Follow(T)
Follow(')') = {+,$,)}
Follow(+) \superset First(E)
Follow(+) = {(, int}
E does not go to \epsilon, so this is it.
Follow(*) \superset First(T)
Follow(*) = {(, int}
T does not go to \epsilon, so this is it.
Follow(int) \superset First(Y) - \epsilon
Follow(int) = {*}
Follow(int) \superset Follow(T)   (because \epsilon \in First(Y))
Follow(int) = {*,+,$,)}
```

LL(1) Parsing Tables

Goal: Construct a parsing table T for CFG G

- For each production A --> \alpha in G, do:
  - For each terminal t \in First(\alpha), do T[A, t] = \alpha
- If \epsilon \in First(\alpha), for each t \in Follow(A) do T[A, t] = \alpha (for token t)
- If \epsilon \in First(\alpha), for each $ \in Follow(A) do T[A, $] = \alpha (just a special case for $)

Parsing table

```
     int     *     +     (     )          $
E    TX                TX
X                +E          \epsilon   \epsilon
T    int Y             (E)
Y           *T   \epsilon     \epsilon   \epsilon
```

Fill some entries in this table

What happens if we try and build a parsing table for a grammar that is not LL(1). Example: `S --> Sa | b`

```
First(S) = {b}
Follow(S) = {$, a}
```

In this case, both productions will appear at T[S, b]. If an entry is multiply defined (more than one productions appear in a table cell), G is not LL(1). Examples: a grammar that is ambiguous, not left-factored, left-recursive will all be not LL(1). But there are more grammars that may not be LL(1)

Most programming language CFGs are not LL(1). More powerful formulations for describing practical grammars that assemble some of these ideas in more sophisticated ways to develop more parsers