–contents of Recursive Parsing–

–requires–

Context-free Grammars
Regular Expression Grammars
Input-output Grammars

Recursive Parsing

Recalling the original CFG for *Proposition* from the section on context-free grammars

$$
\begin{array}{lll}
\mathit{Proposition} & \leftarrow & \mathit{Disjunction}\ \mathrm{eof} \\
\mathit{Disjunction} & \leftarrow & \mathit{Disjunction} \vee \mathit{Conjunction} \\
\mathit{Disjunction} & \leftarrow & \mathit{Conjunction} \\
\mathit{Conjunction} & \leftarrow & \mathit{Conjunction} \wedge \mathit{Negation} \\
\mathit{Conjunction} & \leftarrow & \mathit{Negation} \\
\mathit{Negation} & \leftarrow & \neg\ \mathit{Boolean} \\
\mathit{Negation} & \leftarrow & \mathit{Boolean} \\
\mathit{Boolean} & \leftarrow & \mathrm{t} \\
\mathit{Boolean} & \leftarrow & \mathrm{f} \\
\mathit{Boolean} & \leftarrow & (\ \mathit{Disjunction}\ )
\end{array}
$$

one can append the rule numbers as output symbols. In this case there is no difficulty distinguishing V_O since the digits $0 \ldots 9$ are not in V_I.

$$
\begin{array}{llll}
\mathit{Proposition} & \leftarrow & \mathit{Disjunction}\ \mathrm{eof} & 0 \\
\mathit{Disjunction} & \leftarrow & \mathit{Disjunction} \vee \mathit{Conjunction} & 1 \\
\mathit{Disjunction} & \leftarrow & \mathit{Conjunction} & 2 \\
\mathit{Conjunction} & \leftarrow & \mathit{Conjunction} \wedge \mathit{Negation} & 3 \\
\mathit{Conjunction} & \leftarrow & \mathit{Negation} & 4 \\
\mathit{Negation} & \leftarrow & \neg\ \mathit{Boolean} & 5 \\
\mathit{Negation} & \leftarrow & \mathit{Boolean} & 6 \\
\mathit{Boolean} & \leftarrow & \mathrm{t} & 7 \\
\mathit{Boolean} & \leftarrow & \mathrm{f} & 8 \\
\mathit{Boolean} & \leftarrow & (\ \mathit{Disjunction}\ ) & 9
\end{array}
$$

Apply the transformation for CFG left recursion removal,

given: $A{\leftarrow}\alpha \in \Pi \wedge A{\leftarrow}A\beta \in \Pi$
add: $A{\leftarrow}\alpha(\beta)^*$
remove: $A{\leftarrow}\alpha, A{\leftarrow}A\beta$

dragging the output symbols along with the rest.

$$
\begin{array}{lcl}
\textit{Proposition} & \leftarrow & \textit{Disjunction}\ \text{eof}\ 0 \\
\textit{Disjunction} & \leftarrow & \textit{Conjunction}\ 2\ \textit{(}\vee\ \textit{Conjunction}\ 1\textit{)*} \\
\textit{Conjunction} & \leftarrow & \textit{Negation}\ 4\ \textit{(}\wedge\ \textit{Negation}\ 3\textit{)*} \\
\textit{Negation} & \leftarrow & \neg\ \textit{Boolean}\ 5 \mid \textit{Boolean}\ 6 \\
\textit{Boolean} & \leftarrow & \text{t}\ 7 \mid \text{f}\ 8 \mid (\ \textit{Disjunction}\ )\ 9
\end{array}
$$

The above grammar is a template for a recursive parser for *Proposition*. Each phrase name gives rise to a corresponding function. Each such function relies on the functions it calls to "do their job." There is a variable `next` which has the pending input symbol. There is a function `scan` which steps ahead in the input. There is a function `shift` which reports that an input symbol has been shifted and calls `scan`. There is a function `reduce` that reports that a grammar rule has been applied.

Here is the code in MATLAB, with `~` for $\neg$, `|` for $\vee$ and `&` for $\wedge$.

```
function sr = Proposition(src)

  EOF  = 0;  src = [src EOF];   % append artificial EOF
  next = ''; ip  = 1; scan();   % initialize next
  sr   = ''; op  = 1;           % next avail output position

  Disjunction();

  switch next
    case EOF; reduce('0');
    otherwise; error('missing EOF');
  end

  return;              % end of main function

  % ------------ nested functions ------------
  function Disjunction()
    Conjunction(); reduce('2');
    while next == '|'
      shift(); Conjunction(); reduce('1');
    end
  end

  function Conjunction()
    Negation(); reduce('4');
    while next == '&'
      shift(); Negation(); reduce('3');
    end
```

```
  end

  function Negation()
    switch next
      case '~'; shift(); Boolean(); reduce('5');
      otherwise; Boolean(); reduce('6');
    end
  end

  function Boolean()
    switch next
      case 't'; shift(); reduce('7');
      case 'f'; shift(); reduce('8');
      case '('; shift(); Disjunction();
        switch next
          case ')'; shift(); reduce('9');
          otherwise; error('missing )');
        end
      otherwise; error(['unexpected operand ' next]);
    end
  end

  function shift()
    emit(next); scan();
  end

  function reduce(r)
    emit(r);
  end

  function scan()
    next = src(ip); ip = ip+1;
  end

  function emit(s)
    sr(op) = s; op = op+1;
  end
end
```

The result of `Proposition('(f|t)')` is `(f8642|t7641)96420` To summarize,

two shifts, four reduces, two more shifts, four more reduces, another shift, 5 reduces.

As it turns out, the parenthesis-free notation (PFN or reverse Polish) of Lukasiewicz is embedded in the reduce sequence. If each input symbol consumed by a reduction is printed under the rule that consumed it we get

```
8  6  4  2  7  6  4  1  9  6  4  2  0
f           t           |  ()           eof
```

discarding the parentheses (which is the point, after all) only the `ft|` symbols and the eof remain. This small fact is the basis for compilers that go directly from the shift/reduce sequence to executable code.

Exercise

Why does the PFN show up in the reduce sequence?