Top-Down Parsing and LL(1) Grammars

- Remember that the goal of the parser is to determine the syntactic structure of the source program, given as input the classified words output by the scanner.

- To do this, the parser must use the productions of the CFG that defines the source language syntax to build a parse tree connecting the CFG's start symbol, which serves as the root of the parse tree, with the scanner's output, each individual word of which serves as a leaf of the parse tree.

- A ***top-down parser*** begins with the root of the parse tree and extends the tree downward until its leaves match the classified words output by the scanner.

- It does this by repeatedly choosing a single nonterminal α on the bottom fringe of the partially-built parse tree and expanding that nonterminal using some production $\alpha \rightarrow \beta$ from the CFG. Nodes corresponding to the symbols in β are added to the parse tree as children of α.

- If, at the end of this procedure, the fringe of the parse tree contains only terminal symbols exactly matching the classified words output by the scanner, then the parse succeeds, and the source program is a valid sentence in the source language.

- If, on the other hand, the top-down parsing procedure encounters a mismatch between the fringe of the parse tree and the classified words output by the scanner, the parser may backtrack to reconsider previous decisions, trying to fix a production that was incorrectly chosen at some earlier point in parsing.

- If the parser fails to complete parse tree even after backtracking, then the source program is an invalid sentence in the source language.

- Backtracking in a top-down parser can become an extremely expensive operation, especially if it must be done repeatedly. Luckily, a class of grammars exists for which backtracking is never needed.

- In fact, these grammars, called LL(1) grammars, have parsers that can discover a leftmost derivation for a source program without backtracking by scanning the classified words output by the scanner one word at a time, from left to right, with one word of lookahead.
    - The name LL(1) derives from the following properties of these parsers:
        - **L**eft-to-right scan of the source program
        - **L**eftmost derivation
        - **1** word of lookahead

- In what follows, we will explore techniques for transforming a given grammar into an LL(1) grammar, and we will look at two different LL(1) parser variants: recursive-descent parsers and table-driven LL(1) parsers.

- We will begin our exploration by looking again at the algebraic expression grammar from earlier in the course:

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

Eliminating left recursion

- ***Left recursion*** occurs when the first symbol on the right-hand side of a production in a CFG is the same as the symbol on the left-hand side (or can derive that symbol).

- As we can see, the expression grammar above uses left recursion in productions 1, 2, 4, and 5.

- To see why this is a problem, consider how a concrete implementation of a top-down parser might generate a leftmost derivation for the expression `a+b*c`.

- The parser would specifically try first to find a sequence of productions that allows it to match the leftmost term in the expression, `a`.

- To do this, a concrete parser implementation would have to apply productions in a systematic way. One possible choice would be to try productions in the order in which they were defined in the grammar.

- Such a parser, starting with the goal symbol *Expr* and trying first to match `a`, would try the following sequence of productions:

Rule	Sentential Form
start	*Expr*
1	*Expr* + *Term*
1	*Expr* + *Expr* + *Term*
1	*Expr* + *Expr* + *Expr* + *Term*
...	

- In other words, because the parser is seeking a leftmost derivation, it always chooses to rewrite the first nonterminal. Because the grammar is left recursive and because the parser chooses productions in a systematic way, it always chooses production 1.

- Because *Expr* will always be the leftmost nonterminal in this situation, the parser will enter into an infinite loop, never making any progress towards successfully parsing the input expression.

- Luckily, it is easy to transform a grammar to eliminate left recursion, and doing so will free us of this particular problem.

- There are two types of left recursion we need to eliminate:
    - ***Direct left recursion***, which occurs when the first symbol on the right-hand side of a production is exactly the same as the symbol on the left-hand side.
    - ***Indirect left recursion***, which occurs when the first symbol on the right-hand side of a production can *derive* the one on the left-hand side.

        - In other words, indirect left recursion occurs with a chain of rules such as $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, $\gamma \rightarrow \delta\alpha$.

- Eliminating direct left recursion can be done by rewriting the productions to use right recursion. A direct left-recursive production will have the following form (where β could be the empty string ϵ):

$$\begin{aligned} \alpha &\rightarrow \alpha\ \gamma \\ &\mid \beta \end{aligned}$$

- The right-recursive version of this production introduces a new nonterminal α' and transfers the recursion onto α':

$$\begin{aligned} \alpha &\rightarrow \beta\ \alpha' \\ \alpha' &\rightarrow \gamma\ \alpha' \\ &\mid \epsilon \end{aligned}$$

- Indirect left recursion can be eliminated by systematically using grammar productions to expand nonterminals that occur as the first symbol on the right-hand side of a production, so that indirect left recursion is converted into direct left recursion. Then, the rewrite rule above can be used to eliminate the direct left recursion.
    - Our textbook describes a concrete algorithm for doing this.

- Our expression grammar above does not have any indirect left recursion, only direct left recursion. Thus, we can apply the rewrite rule above. First, we'll eliminate the left recursion that occurs in the productions for expanding *Expr*:

$$\begin{aligned} \textit{Expr} &\rightarrow \textit{Expr} + \textit{Term} \\ &\mid \textit{Expr} - \textit{Term} \\ &\mid \textit{Term} \end{aligned}$$

- Applying the rewrite rule above gives us the following productions:

$$\begin{aligned} \textit{Expr} &\rightarrow \textit{Term Expr'} \\ \textit{Expr'} &\rightarrow + \textit{ Term Expr'} \\ &\mid - \textit{ Term Expr'} \\ &\mid \epsilon \end{aligned}$$

- The productions for expanding *Term* are also left-recursive:

$$\begin{aligned} \textit{Term} &\rightarrow \textit{Term} * \textit{Factor} \\ &\mid \textit{Term} / \textit{Factor} \\ &\mid \textit{Factor} \end{aligned}$$

- These can be rewritten as right-recursive productions as follows:

$$\begin{aligned} \textit{Term} &\rightarrow \textit{Factor Term'} \\ \textit{Term'} &\rightarrow * \textit{Factor Term'} \\ &\mid / \textit{Factor Term'} \\ &\mid \epsilon \end{aligned}$$

Backtrack-free grammars

- Backtracking is the major source of inefficiency for a leftmost, top-down parser.

- Luckily, there is a large class of CFGs—the LL(1) grammars—for which a leftmost, top-down parser never needs to backtrack.

- For these grammars, we only need to provide the parser with a one-word ***lookahead***, i.e. the next (leftmost) word in the input stream that needs to be matched by the parse tree.

- Intuitively, for a leftmost, top-down parser, a grammar is backtrack-free with a one-word lookahead (i.e. it is an LL(1) grammar) if each alternative production that expands the leftmost nonterminal in the sentential form leads to a different terminal symbol.

- If this is the case, we simply need to compare the lookahead word with each of these distinct terminal symbols in order to choose the correct production.

- We can formalize the LL(1) property. To do so, we will introduce two different sets, *FIRST(α)* and *FOLLOW(α)*, where α is any terminal or nonterminal grammar symbol.

The *FIRST* sets

- The set *FIRST(α)* is the set of all terminal symbols that can appear as the first word in a string derived from α.
  - If α is either a terminal symbol or ϵ or `eof`, then *FIRST(α)* = {α}.
  - For any nonterminal symbol *A*, *FIRST(A)* is the set of all terminal symbols that can appear as the first symbol in a sentential form derived from *A*.

- Our textbook describes an algorithm for computing the *FIRST* sets for all symbols in a grammar. The idea behind this algorithm is as follows:
  - Initialize *FIRST(α)* = {α} for every terminal symbol, ϵ and `eof`.
  - Initialize *FIRST(A)* = $\varnothing$ for all nonterminal symbols.
  - While the *FIRST* sets are still changing, loop over all productions of the form $A \rightarrow \beta_1\beta_2...\beta_n$.
    - Compute *FIRST(A)* = *FIRST(A)* $\cup$ *FIRST*($\beta_1\beta_2...\beta_k$), where β_k is the first symbol such that $\epsilon \notin$ *FIRST*(β_k), and *FIRST*($\beta_1\beta_2...\beta_k$) is the union of the *FIRST* sets of $\beta_1\beta_2...\beta_k$.
      - If $\epsilon \in$ *FIRST*(β_i) for all *i=1...n*, then *FIRST*($\beta_1\beta_2...\beta_k$) contains ϵ. Otherwise, ϵ is removed from *FIRST*($\beta_1\beta_2...\beta_k$).

- The key to understanding this algorithm is to recognize that, for a string of symbols $\beta_1\beta_2...\beta_n$, if $\epsilon \in$ *FIRST*(β_1), then it is possible that a nonterminal in *FIRST*(β_2) could be the first symbol in $\beta_1\beta_2...\beta_n$. Similarly, if $\epsilon \in$ *FIRST*(β_2), then it is possible that a nonterminal in *FIRST*(β_3) could be the first symbol in $\beta_1\beta_2...\beta_n$, and so forth.

- Going back to the right-recursive version of our expression grammar, we have the following *FIRST* sets for the nonterminals:

A	*FIRST(A)*
Expr	`(`, `name`, `num`
Expr'	`+`, `-`, ϵ
Term	`(`, `name`, `num`
Term'	`*`, `/`, ϵ
Factor	`(`, `name`, `num`

- The *FIRST* set of each terminal contains only that terminal. Likewise with ϵ and `eof`.

- In general, the *FIRST* sets are used in conjunction with the lookahead symbol during top-down parsing to choose between productions.

- For example, if the leftmost nonterminal in the partially-built parse tree is *Expr'*, there are three possible productions we could apply: *Expr'*→ + *Term Expr'*, *Expr'*→ − *Term Expr'*, and *Expr'*→ ϵ.

- In this situation, the following *FIRST* sets are relevant:
  - *FIRST(+ Term Expr')* = {+}
  - *FIRST(− Term Expr')* = {−}
  - *FIRST(ϵ)* = {ϵ}

- Thus, if the lookahead symbol is +, the parser should choose *Expr'*→ + *Term Expr'*. If the lookahead symbol is −, the parser should choose *Expr'*→ − *Term Expr'*.

- If the lookahead is neither + nor −, the situation is slightly more complicated.

- Intuitively, choosing *Expr'*→ ϵ has the effect of moving the parser past the leftmost *Expr'* symbol to focus on the next-leftmost nonterminal in the fringe of the parse tree.

- Thus, the parser should only choose that production if the lookahead symbol is one that can validly appear immediately after an *Expr'*. If it is not, the parser must report an error.

- It will be helpful, then, to know what symbols can immediately follow an *Expr'* in a valid expression. For this, we will introduce the *FOLLOW* set.

The *FOLLOW* sets

- For a nonterminal *A*, the set *FOLLOW(A)* is the set of all terminal symbols that can appear immediately after a string derived from *A* in a valid sentence.

- Again, our textbook describes an algorithm for computing the *FOLLOW* sets for all nonterminals in a grammar. At a high level, the algorithm works as follows:

    - Initialize *FOLLOW(A)* = ∅ for all nonterminal symbols.
    - While the *FOLLOW* sets are still changing, loop over all productions of the form $A \rightarrow \beta_1\beta_2...\beta_n$.
        - If β_n is a nonterminal, $FOLLOW(\beta_n) = FOLLOW(\beta_n) \cup FOLLOW(A)$
        - For i = *n-1* down to *1*:
            - If β_i is a nonterminal, $FOLLOW(\beta_i) = FOLLOW(\beta_i) \cup (FIRST(\beta_{i+1}...\beta_k) - \epsilon)$, where β_k is the first β after β_i whose *FIRST* set doesn't contain ϵ.

- The idea behind this algorithm is that the order of symbols in the right-hand sides of the productions helps us know what symbols can legally appear after each nonterminal symbol in the grammar.

- Running this algorithm on the right-recursive version of the expression grammar gives us the following *FOLLOW* sets for the grammar's nonterminals:

A	*FOLLOW(A)*
Expr	`eof`, `)`
Expr'	`eof`, `)`
Term	`eof`, `+`, `-`, `)`
Term'	`eof`, `+`, `-`, `)`
Factor	`eof`, `+`, `-`, `*`, `/`, `)`

- Coming back to our example above, if the parser is trying to expand an *Expr'* and the lookahead symbol is not `+` or `-`, the parser can check to see if the lookahead symbol is in *FOLLOW(Expr')*. If it is, the parser can apply the production $Expr' \rightarrow \epsilon$. Otherwise, the parser must report a syntax error.

Defining an LL(1) grammar

- With the *FIRST* and *FOLLOW* sets defined, we can precisely define an LL(1) grammar.
    - Remember, an LL(1) grammar is one that can be from which a leftmost derivation can be discovered with no backtracking by a top-down parser using a left-to-right scan of the input stream with one word of lookahead.

- First, for a production $A \rightarrow \beta$, define the set *FIRST+* as follows:
  - If $\epsilon \notin$ *FIRST*(β), *then FIRST+*($A \rightarrow \beta$) = *FIRST*(β).
  - Otherwise, *FIRST+*($A \rightarrow \beta$) = *FIRST*(β) $\cup$ *FOLLOW*(A).

- Then, an LL(1) grammar has the property that, for any nonterminal A with multiple right-hand sides, $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$, then:

  $FIRST+(A \rightarrow \beta_i) \cap FIRST+(A \rightarrow \beta_j) = \varnothing, \ \forall \ i \neq j$

- Intuitively, this property simply states that productions with a common left-hand side should be uniquely identifiable based on the lookahead symbol.

- If we examine the right-recursive version of our expression grammar, we can see that it has this property and is thus an LL(1) grammar.

Eliminating backtracking via left factoring

- We now have tools to determine whether a grammar is backtrack-free. If we determine that a grammar *is not* backtrack-free, there are some common techniques for eliminating backtracking.

- One of these techniques is ***left-factoring***, which involves detecting productions whose right-hand sides share a common prefix and transforming them to isolate that common prefix in a single production.

- More formally, left-factoring helps eliminate backtracking in a situation where we have a set of productions like this one:

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$

- Specifically, in this set of productions, α is a common prefix for several right-hand sides, and the γ_i's are right-hand sides that don't begin with α.

- Left-factoring involves rewriting such a set of productions by introducing a new nonterminal B to represent the alternate suffixes of α and rewriting the original productions as follows:

  $A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$
  $B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

- After this rewrite, α occurs as a prefix in only a single production for *A*.

- As we saw above, the right-recursive version of our expression grammar is already backtrack-free. To see how left-factoring can be applied to a real grammar, let's expand our expression grammar to include function calls, denoted by a `name` and parentheses, e.g. `name(...)`, and array element references denoted by a `name` and square brackets, e.g. `name[...]`:

  Factor → `name`
  | `name` `(` *ArgList* `)`
  | `name` `[` *ArgList* `]`
  ArgList → *Expr MoreArgs*
  MoreArgs → `,` *Expr MoreArgs*
  | ϵ

- Importantly, in this language, references to elements in multi-dimensional arrays use only a single set of square brackets, e.g. `arr[4, 8, 2]`.

- In this case, if the parser is trying to expand a *Factor* with a lookahead of `name`, it has no basis for choosing between the three productions above. It could thus make a wrong choice and eventually need to backtrack.
  - Of course, we could solve this problem using a two-word lookahead, but we could then devise a grammar where the two-word lookahead was still insufficient to prevent backtracking. This is the case for any finite lookahead.

- To fix this problem, we can left-factor the grammar.

- Left-factoring here involves isolating the common prefix, `name`, in a single production and introducing a new nonterminal to move the alternate suffixes of `name` into new productions:

  Factor → `name` *Arguments*
  Arguments → `(` *ArgList* `)`
  | `[` *ArgList* `]`
  | ϵ

- After this transformation, the grammar will be backtrack-free (and thus an LL(1) grammar).

- Left-factoring can often completely eliminate backtracking to convert a grammar into an LL(1) grammar. However, in general, CFGs exist for which it is not possible to eliminate backtracking.

- And, while we can decide whether or not a *specific* grammar is an LL(1) grammar, it is undecidable whether or not an arbitrary grammar can be converted an LL(1) grammar.

Recursive-descent parsers

- A ***recursive-descent parser*** is a specific type of LL(1) parser (i.e. a top-down parser that uses a left-to-right scan of the input to construct a leftmost derivation using a one-word lookahead).

- A recursive descent parser is implemented using a set of functions, each of which recognizes an instance of a single nonterminal in the grammar and each of which recursively calls other functions to recognize to other nonterminals.

- The function corresponding to a given nonterminal would use the lookahead symbol in conjunction with the *FIRST+* sets for the productions for which the nonterminal was the left-hand side to choose between those productions (or to report a syntax error).

- When a production is chosen based on the lookahead, the function can do any of the following things:
    - Advance the lookahead forward by calling the scanner (i.e. `nextWord()`) if the current lookahead matches a terminal symbol at the beginning of the production's right-hand side.
    - Call one or more functions corresponding to other nonterminals to test for the presence of those nonterminals.
    - Return true, to indicate that an ϵ-production was chosen and that the parser should move on to the next nonterminal.

- As an example, recall the productions we wrote above for *Expr'* in our expression grammar:

Expr' → + *Term Expr'*
| − *Term Expr'*
| ϵ

- A recursive-descent parser would contain a function, say `exprPrime()`, corresponding to the nonterminal *Expr'*. This function might look something like this ():

```
exprPrime()
     if (lookahead == "+" or lookahead == "-")
          lookahead = nextWord()
          if (term())
               return exprPrime()
          else
               fail()
     else if (lookahead == ")" || lookahead == eof)
          return true
     else
          fail()
```

- Here, `lookahead` is a global variable holding the lookahead symbol, `term()` is a function corresponding to the nonterminal *Term*, and `fail()` is a function to report a syntax error.

- The `if` condition here handles the first two productions for *Expr'* (which are combined here because they share a common suffix), advancing the lookahead if either a + or a − terminal symbol is matched and then testing for a *Term* followed by another *Expr'*.

- The `else if` condition tests for the ϵ-production, simply returning `true` to indicate that an *Expr'* was matched.

- The `else` condition handles a syntax error, where the lookahead does not match the *FIRST+* set for any production of *Expr'*.

- Functions could be written in a similar fashion for the other nonterminals in the grammar.
    - Our textbook has a complete example implementation of these functions.

- Given these functions, parsing would start by calling the function `expr()` to test for the grammar's goal symbol, *Expr:*

```
lookahead = nextWord()
if (expr())
     if (lookahead == eof)
          return SUCCESS
fail()
```

Automatically-generated LL(1) parsers

- As was the case with scanners, an LL(1) parser can be automatically generated given an LL(1) grammar.

- A generated LL(1) parser is typically centered around a table that essentially encodes the *FIRST+* sets for the grammar. Specifically, for each possible combination of lookahead symbol and nonterminal, the table encodes a specific grammar production that should be chosen when expanding that particular nonterminal given that particular lookahead.

- This table is used in conjunction with a skeleton parser that implements the mechanics of parsing: keeping track of the current nonterminal being expanded, advancing the lookahead when a nonterminal is matched, etc.

- Our textbook provides a pseudocode implementation of such a skeleton parser and an accompanying parse table generator.