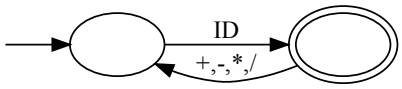
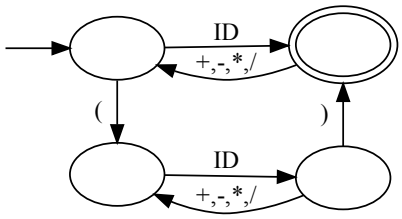
# 8 Parsing and context-free languages

In the last module, we learned about regular languages and applied them to scanning. In this module, we will try to identify nested tree structures in a program, for example the structure of an expression such as `a+b*c-d`. As a first step, we might notice that we can already specify expressions of this form using a DFA:




Here, we assume that we have scanned the expression first, so that the symbols in the alphabet of this DFA are entire tokens, and we use ID to stand for all *identifier* tokens that consist of a variable name such as `a`, `b`, `c`, or `d`.

If we allow expressions to contain parentheses, like `(a+b)*(c-d)`, we need a larger DFA:




However, this DFA would still reject expressions with *nested* parentheses, such as `((a+b)*(c-d))`. To accept this expression, we could add another level of two more nodes to the DFA, but then it would still reject parentheses nested more than two levels deep. In general, to allow arbitrary nesting, the DFA would need to keep track of how many open-parentheses it has seen so that it could match them with the same number of close-parentheses, but since there can be arbitrarily many open-parentheses, an infinite number of distinct states would be necessary to keep track of their number. A DFA with an infinite number of states would no longer be a deterministic *finite* automaton.

In general, nested constructs such as expressions (and if-statements, loops, functions, and classes) cannot be specified using regular languages, unless they are restricted to some fixed number of levels of nesting. To specify nested constructs, we will use more expressive *context-free languages*.

## 8.1 Context-free grammars

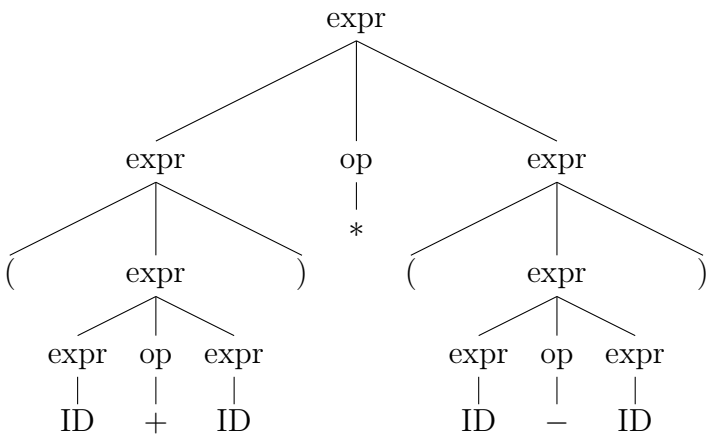
The following is an example of a *context-free grammar* for expressions with nested parentheses:

$$\begin{aligned} \text{expr} &\rightarrow \text{ID} \mid \text{expr op expr} \mid (\ \text{expr}\ ) \\ \text{op} &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

In words, this grammar says that an expression is either an identifier, or an expression followed by an operator and another expression, or an expression in parentheses. An operator is one of the four listed arithmetic operators.

Like a regular expression, a context-free grammar supports concatenation (e.g., expr op expr) and alternation (set union) (e.g., $+ \mid - \mid * \mid /$). Instead of iteration with the Kleene $^*$ operator of regular expressions, a context-free grammar supports recursion: on the right side of the arrow, we can refer back to the symbols expr and op defined on the left sides of the arrows. Unsurprisingly, recursion is expressive enough to model iteration, but also enables us to specify nested structures with arbitrary levels of nesting.

Given a context-free grammar, we can construct *parse trees* for words. Here is an example parse tree for the word ( ID + ID ) * ( ID - ID ) using the example grammar:




The leaves of the parse tree are the symbols in the word. Each internal node in a parse tree corresponds to a rule in the grammar: the label of the node is one of the left sides of a rule (e.g., expr or op), and the children of the node correspond to one of the alternatives on the right side of the rule. For example, some of the nodes labelled expr have three children labelled expr op expr, corresponding to the alternative expr $\to$ expr op expr, while other nodes labelled expr have a single child labelled ID, corresponding to the alternative expr $\to$ ID.

Formally, a *context-free grammar* is a four-tuple $\langle V, \Sigma, P, S \rangle$ where:

- $V$ is a finite set of *non-terminal symbols*.
- $\Sigma$ is a finite set of *terminal symbols* (the *alphabet*).
- $P$ is a finite set of *productions*, also called *rules*.
- $S \in V$ is the *start non-terminal*.

Each production in $P$ is of the form $A \to \alpha$, where $A \in V$ is a non-terminal symbol and $\alpha$ is a sequence of terminal and non-terminal symbols. For example, the example expression

grammar is written formally as:

$$\left\langle \{\text{expr}, \text{op}\}, \{+, -, *, /, ID, (, )\}, \left\{ \begin{array}{l} \text{expr} \to \text{ID}, \\ \text{expr} \to \text{expr op expr}, \\ \text{expr} \to (\ \text{expr}\ ), \\ \text{op} \to +, \\ \text{op} \to -, \\ \text{op} \to *, \\ \text{op} \to / \end{array} \right\}, \text{expr} \right\rangle$$

In this form, instead of separating the alternatives for a given non-terminal symbol using |, as in op $\to + \mid - \mid * \mid /$, we write a collection of separate productions: op $\to +$, op $\to -$, op $\to *$, and op $\to /$.

In the rest of this module, we will use the convention that:

- variables $a$, $b$, $c$, and $d$ always refer to terminal symbols in $\Sigma$,
- variables $A$, $B$, $C$, $D$, and $S$ always refer to non-terminal symbols in $V$,
- variables $W$, $X$, $Y$, and $Z$ always refer to terminal or non-terminal symbols in $\Sigma \cup V$,
- variables $w$, $x$, $y$, and $z$ always refer to sequences of terminal symbols from $\Sigma$, i.e., to words, and
- variables $\alpha$, $\beta$, $\gamma$, and $\delta$ always refer to sequences of terminal or non-terminal symbols from $\Sigma \cup V$.

As we did for regular languages, we will now formally define the language specified by a given context-free grammar $G = \langle V, \Sigma, P, S \rangle$.

Whenever $A \to \gamma \in P$ is a production in the grammar, we say that $\alpha A \beta$ *directly derives* $\alpha\gamma\beta$ and we write $\alpha A \beta \Rightarrow \alpha\gamma\beta$, for all non-terminals $A \in V$ and all sequences of terminal and non-terminal symbols $\alpha, \beta, \gamma$. In words, if a sequence of terminal and non-terminal symbols contains $A$, so the sequence can be written $\alpha A \beta$ for some $\alpha, \beta$, then the sequence *directly derives* another sequence in which the $A$ has been replaced by $\gamma$; this latter sequence can be written $\alpha\gamma\beta$. For example, using our expression grammar, expr *directly derives* expr op expr, and expr op expr *directly derives* expr+expr, so we write expr $\Rightarrow$ expr op expr $\Rightarrow$ expr+expr.

Whenever $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ is a sequence of zero or more *directly-derives* steps, we say that $\alpha_1$ *derives* $\alpha_n$ and we write $\alpha_1 \Rightarrow^* \alpha_n$. For example, using our expression grammar, expr *derives* ID + ID, so we write expr $\Rightarrow^*$ ID + ID. The sequence of *directly-derives* steps used to reach $\alpha_n$ from $\alpha_1$ is called a *derivation*.

We say that *the language generated by the grammar* $G = \langle V, \Sigma, P, S \rangle$ is $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. The language contains all words $w$ that are derived by the start symbol $S$ of the grammar. Sometimes we use the word *specified* instead of *generated*. Notice that in this definition, $w$ is specified as being in $\Sigma^*$: it must be a sequence of terminal symbols only, and cannot contain any non-terminal symbols.

A *language is context-free* if there exists a context-free grammar that generates it. There may be multiple grammars that generate the same language.

*Remark.* It is undecidable (there cannot exist any algorithm to decide), given two context-free grammars $G_1$ and $G_2$, whether they generate the same language (set of words). For regular languages, on the other hand, it is decidable whether two DFAs, NFAs, or regular expressions specify the same language. Thus, when moving from regular languages to context-free languages, we gain more expressiveness in the languages that can be specified, but we lose some of the ability to reason about properties of the languages.

A context-free grammar is *ambiguous* if we can use it to construct two distinct parse trees for the same word. For example, using our expression grammar on the word ID+ID∗ID, we can construct the following two parse trees:


The tree on the left suggests that we should evaluate, for example, 1+2∗3 as (1+2)∗3 = 9, while the tree on the right suggests that we should evaluate the same expression as 1+(2∗3) = 7. For specifying programming languages, we prefer *unambiguous* grammars to ensure that a given program has a unique meaning. In particular, we would like a grammar that enforces a tree like the one on the right, which is consistent with conventional order of operations rules that give multiplication precedence over addition. The grammar that will be provided for the Lacs language is unambiguous and does enforce standard order of operations rules.

*Remark.* It is undecidable whether a given context-free grammar is ambiguous. It is possible to *prove* that some *specific* grammar (such as the provided Lacs grammar) is unambiguous, but there cannot exist any algorithm that would decide, for any arbitrary grammar given to it as input, whether the grammar is ambiguous.

For example, the following grammar generates the same language as the expression grammar given earlier, but this one is unambiguous and enforces the conventional order of operations:

$$\begin{aligned} \text{expr} &\to \text{term} \mid \text{expr} + \text{term} \mid \text{expr} - \text{term} \\ \text{term} &\to \text{factor} \mid \text{term} * \text{factor} \mid \text{term} \,/\, \text{factor} \\ \text{factor} &\to \text{ID} \mid (\ \text{expr}\ ) \end{aligned}$$

## 8.2 CYK parsing

The process of *parsing* takes as input a context-free grammar $G$ specifying a language and a word $w$, and decides whether $w$ is in the language of $G$. If it is, a parser typically generates a proof of this in the form of a derivation or a parse tree. There are many known parsing algorithms that vary in generality, complexity, and efficiency. We will start with one specific algorithm that you could invent yourself if you thought carefully about it. The algorithm is intuitive so that one can understand it completely and thoroughly in the equivalent of one

week of lectures. A later section will provide a brief comparison with some other popular parsing algorithms.

Letting $S$ be the start symbol of the input grammar $G$, a parser needs to decide whether $S$ derives $w$: $S \Rightarrow^* w$. In the process of answering this question, our parser will need to decide, more generally, whether $\alpha$ derives $x$, for various sequences of terminal and non-terminal symbols $\alpha$ and various sequences of terminal symbols $x$. The final question $S \Rightarrow^* w$ is a special case of $\alpha \Rightarrow^* x$.

For given $\alpha$ and $x$, how do we decide whether $\alpha \Rightarrow^* x$? As with many algorithmic problems, we break down the possible inputs into cases and solve each case. Since $\alpha$ is a sequence of symbols, one possibility is that $\alpha$ is the empty sequence $\varepsilon$. What can we derive from the empty sequence? Since it contains no non-terminal symbols, it cannot be expanded to anything else. The empty sequence derives only itself: $\varepsilon \Rightarrow^* \varepsilon$. We have the beginning of an algorithm:

```
parse(α, x) = // does α ⇒* x ?
  if(α.isEmpty) { // Case 1
    if(x.isEmpty) true
    else false
  } else if(α = aβ) { // Case 2
    if(x = az && parse(β, z)) true
    else false
  } else if(α = A){ // Case 3
    for each production A → γ ∈ P {
      if(parse(γ, x)) return true
    }
    false
  } else {// α = Aβ     Case 4
    for each way to split x into two substrings x = x1x2 {
      if(parse(A, x1) && parse(β, x2)) return true
    }
    false
  }
```

Now, what if $\alpha$ is non-empty? Then it must start with some symbol, a terminal or a non-terminal, followed by the rest of the sequence (which could be empty). Let's first consider the case when the first symbol is a terminal; call this first terminal symbol $a$ and the rest of the sequence $\beta$. When does $\alpha = a\beta$ derive $x$? Since $a$ is terminal symbol, it cannot change to anything else in the derivation. Therefore, in order for $a\beta$ to derive $x$, $x$ needs to start with $a$. Let $z$ be the rest of $x$, after the initial $a$, so $x = az$. Now $a\beta$ derives $az$ if and only if $\beta$ derives $z$, which we can decide using a recursive call to `parse`. We summarize this in the second case of the algorithm:

```
parse(α, x) = // does α ⇒* x ?
  if(α.isEmpty) { // Case 1
    if(x.isEmpty) true
    else false
  } else if(α = aβ) { // Case 2
    if(x = az && parse(β, z)) true
    else false
  } else if(α = A){ // Case 3
    for each production A → γ ∈ P {
      if(parse(γ, x)) return true
    }
    false
  } else {// α = Aβ     Case 4
    for each way to split x into two substrings x = x1x2 {
      if(parse(A, x1) && parse(β, x2)) return true
    }
    false
  }
```

We are left with the case that $\alpha$ is non-empty and starts with some non-terminal symbol $A$, followed by the rest of the sequence $\beta$. Let's further split this case into the subcases depending on whether $\beta$ is empty or non-empty. In the first subcase, when $\beta$ is empty, $\alpha$ is just the single non-terminal symbol $A$. Now we must decide whether $A$ derives $x$. Since $A$ is a non-terminal and $x$ is a sequence of terminals, $x$ cannot be equal to $A$, so any derivation must start with at least one derivation step: $A$ must derive some $\gamma$, which might later derive $x$, so $A \Rightarrow \gamma \Rightarrow^* x$. In the first step, $\gamma$ can be any sequence of symbols such that there is a rule $A \to \gamma$ in the grammar, so we loop over all such rules to try all the possible values of $\gamma$. For each $\gamma$, we test whether $\gamma \Rightarrow^* x$ using a recursive call to `parse`. If one of these calls succeeds, then we have found a derivation $\alpha = A \Rightarrow \gamma \Rightarrow^* x$, so we immediately return true in the loop body. If the loop finishes without finding any $\gamma$ that derives $x$, we return false. This is summarized in the third case of the algorithm:

```
parse(α, x) = // does α ⇒* x ?
  if(α.isEmpty) { // Case 1
    if(x.isEmpty) true
    else false
  } else if(α = aβ) { // Case 2
    if(x = az && parse(β, z)) true
    else false
  } else if(α = A){ // Case 3
    for each production A → γ ∈ P {
      if(parse(γ, x)) return true
    }
    false
  } else {// α = Aβ     Case 4
    for each way to split x into two substrings x = x1x2 {
      if(parse(A, x1) && parse(β, x2)) return true
    }
    false
  }
```

Finally, we have the case that $\alpha = A\beta$, where $A$ is a non-terminal and $\beta$ is a non-empty sequence of terminal and non-terminal symbols. In this case, $\alpha$ derives $x$ if we can split $x$ into two (possibly empty) substrings $x_1$ and $x_2$, such that $A$ derives $x_1$ and $\beta$ derives $x_2$. We therefore loop over all possible ways of splitting $x$ into two substrings and use recursive calls to `parse` to decide whether $A \Rightarrow^* x_1$ and $\beta \Rightarrow^* x_2$. If we find a split of $x$ that works, we return true immediately in the loop body. If the loop finishes without finding a way for $A\beta$ to derive $x_1x_2$, we return false. This is summarized in the final, fourth case of the algorithm:

```
parse(α, x) = // does α ⇒* x ?
  if(α.isEmpty) { // Case 1
    if(x.isEmpty) true
    else false
  } else if(α = aβ) { // Case 2
    if(x = az && parse(β, z)) true
    else false
  } else if(α = A){ // Case 3
    for each production A → γ ∈ P {
      if(parse(γ, x)) return true
    }
    false
  } else {// α = Aβ     Case 4
    for each way to split x into two substrings x = x1x2 {
      if(parse(A, x1) && parse(β, x2)) return true
    }
    false
  }
```

This completes all the cases of the algorithm.

### 8.2.1 Memoization

Although each case of the algorithm makes intuitive sense, the algorithm contains many recursive calls, so we might worry about its running time. In fact, on the following example, the algorithm gets into an infinite recursion, so its running time is even worse than exponential; it is infinite. Suppose that we want to know whether expr derives ID + ID according to the example grammar given at the beginning of Section 8.1. The following is a trace of the calls to `parse`, with indentation identifying recursive calls. Take some time to go through the steps of the trace, working through the cases of the algorithm.

```
326 parse(expr, ID + ID) // Case 3
327   // try the production expr -> ID
328   parse(ID, ID + ID) // Case 2
329     parse(ε, + ID) // Case 1
330     false
331   false
332   // try the production expr -> expr op expr
333   parse(expr op expr, ID + ID) // Case 4
334     // try splitting ID + ID into ε and ID + ID
335     parse(expr, ε) // Case 3
336       // try the production expr -> ID
337       parse(ID, ε) // Case 2
338       false
339       // try the production expr -> expr op expr
340       parse(expr op expr, ε) // Case 4
341         // try splitting ε into ε and ε
342         parse(expr, ε) // Case 3
```

Notice that in the last line, Line 342, we are asking the same question as we already asked in Line 335, whether expr derives $\varepsilon$. If we allow the algorithm to keep running, it will just repeat these lines, recursing infinitely.

The algorithm is exploring the possibility that there is a derivation expr $\Rightarrow$ expr op expr $\Rightarrow^*$ $\varepsilon$ if there is a derivation expr $\Rightarrow^*$ $\varepsilon$ (and another derivation op expr $\Rightarrow^*$ $\varepsilon$). But if we already had a derivation expr $\Rightarrow^*$ $\varepsilon$, then we could just use it as proof that expr $\Rightarrow^*$ $\varepsilon$ without having to extend it to a longer derivation expr $\Rightarrow$ expr op expr $\Rightarrow^*$ $\varepsilon$. Therefore, in general, whenever we encounter a case when deriving $x$ from $\alpha$ requires an existing, shorter derivation of $x$ from $\alpha$, we will make the algorithm return `false`: we do not allow a derivation of $\alpha \Rightarrow^* x$ to have some other derivation of $\alpha \Rightarrow^* x$ as a subderivation of itself.

To make the algorithm return false when it encounters a question that it is in the process of answering, we will use a technique called memoization. *Memoization* consists of keeping track of the questions that an algorithm has already been asked in a table. If a question is ever repeated, instead of recomputing the answer, the memoized algorithm simply looks it up in the table and returns the answer that it returned the first time the question was asked. In our parsing algorithm, if we repeat a question that the algorithm has already been asked but is still in the process of answering, we will make the algorithm return `false`.

Memoization will make the algorithm immediately return `false` for the repeated query in Line 342. The algorithm can then continue trying other alternatives to eventually complete

and successfully determine that expr op expr derives ID + ID.

To implement memoization, we need to add a table mapping the arguments to `parse` to the results that `parse` has returned for them in the past. The keys in the table are the pairs of the values of $\alpha$ and $x$ passed to `parse`. Although `parse` can give one of two possible answers, `true` or `false`, the table can be in one of *three* states for each pair of argument values:

1. `parse` has not yet been called with the given arguments $\alpha$ and $x$.
2. `parse` has been called with the given arguments and returned `true`.
3. `parse` has been called with the given arguments and returned `false`.

Thus, although the values in the map are `Booleans`, `true` or `false`, the map is also defined for some keys (arguments) and not yet defined for others.

Using $x$ as part of the key can be inefficient because $x$ is a sequence of terminal symbols that is $O(n)$ symbols long in the worst case, where $n$ is the length of the input. We can make the memoization more efficient (in terms of both time and space) by noting that every possible value of $x$ that `parse` is ever called on is a subset of the input $w$ being parsed. Therefore, every $x$ can be represented implicitly by just a pair of integers, the position in $w$ where $x$ *starts* and the *length* of $x$. Alternatively, the position in $w$ where $x$ *ends* can be used instead of the *length* of $x$. Thus, the keys of the map become triples of $\alpha$ and two integers that identify a particular substring $x$ of $w$. Although $\alpha$ is also a sequence like $x$, its length does not depend on the length of the input $w$. The possible values of $\alpha$ are either single non-terminal symbols, right-hand-sides of productions in the grammar, or suffixes of such right-hand-sides. (The `parse` function is called on a non-terminal symbol in Case 4 of the algorithm, on a right-hand-side of a production in Case 3, and on suffixes of existing values of $\alpha$ in Cases 2 and 4.) The lengths of possible values of $\alpha$ are constant for a given grammar and are typically quite short.

To make use of the memoization map, we need to make the following modifications to the algorithm:

1. At each place where the algorithm returns a value, `true` or `false`, we add that value to the memoization map under the key derived from the values of $\alpha$ and $x$.
2. At the beginning of `parse`, we first check whether the memoization map is already defined for the current values of $\alpha$ and $x$. If it is, we just return the value from the map and skip the rest of the computation.
3. Also at the beginning of `parse`, after we have determined that the memoization map is not yet defined and before we begin the computation, we add the value `false` to the map under the current values of $\alpha$ and $x$. This ensures that if we encounter the same question again, the same values of $\alpha$ and $x$ in the course of determining whether $\alpha$ derives $x$, we will return `false` to break the infinite cycle that we discussed in the example above. If it turns out that $\alpha$ does derive $x$ without a cyclic dependence on $\alpha$ deriving $x$, when the algorithm returns `true`, it will overwrite the `false` value that we placed in the map at the beginning of `parse`, so that later queries about $\alpha$ and $x$ will correctly return `true`.

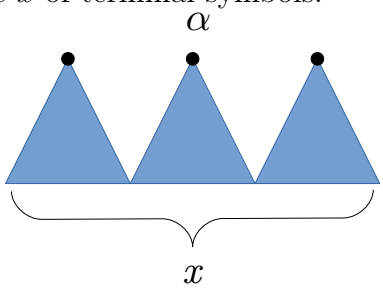
### 8.2.2 Time and space complexity

The most significant data structure that the algorithm builds is the memoization table. We can analyze its size by considering the number of keys stored in it. Each key is a triple $\langle \alpha, \mathit{start}, \mathit{length} \rangle$. As discussed earlier, the number of possible values of $\alpha$ is determined by the grammar being parsed and is constant for any given grammar. Since *start* and *length* are integers in the range from 0 to the length of the input string $w$ being parsed, they each can have $O(n)$ possible values, where $n$ is the length of $w$. Therefore, the total number of possible entries in the memoization table, and thus the space complexity of the algorithm, is bounded by $O(n^2)$.

Each execution of the main body of the `parse` algorithm, not counting the time spent in recursive calls, can take up to $O(n)$ time because the loop in Case 4 iterates as many times as the length of $x$, and $x$ can be as long as the input $w$ in the worst case. The number of iterations of the loop in Case 3 is constant for a given grammar, and the other two cases also require only constant time. Since each execution of the main body of the `parse` algorithm adds an entry to the memoization table, the number of executions is bounded by the number of table entries. There are $O(n^2)$ entries in the memoization table and each execution of the main body of `parse` takes $O(n)$ time, so the overall running time of the algorithm is $O(n^3)$, where $n$ is the length of $w$, the input string to be parsed.

### 8.2.3 Constructing the parse tree

The algorithm so far only decides whether a derivation $\alpha \Rightarrow^* x$ exists. Modifying it to produce a parse tree is quite straightforward. In all the places that the algorithm returns `true`, it needs to instead return a sequence of parse trees. The return type is a *sequence* because $\alpha$ is a sequence of symbols: `parse` will return one parse tree for each symbol in $\alpha$, with the symbol at the root of the tree. The concatenation of the leaves of all the trees returned should be the sequence $x$ of terminal symbols:




This diagram is for an example in which $\alpha$ is a sequence of three symbols. Each symbol derives some sequence of terminal symbols, and when these three sequences are concatenated, they form the sequence $x$.

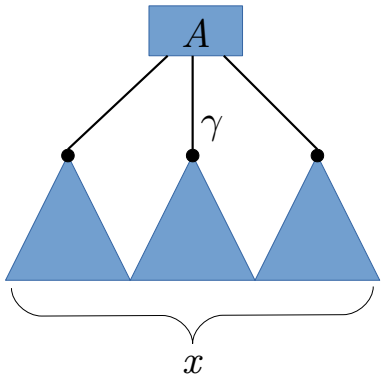
In the various cases, the algorithm constructs the parse trees to be returned by adding nodes to existing parse trees that are returned from recursive calls to `parse`. In each case, the number of parse trees returned is always equal to the number of symbols in $\alpha$.

In Case 1, since $\alpha$ is empty, the algorithm must return the empty sequence of parse trees.

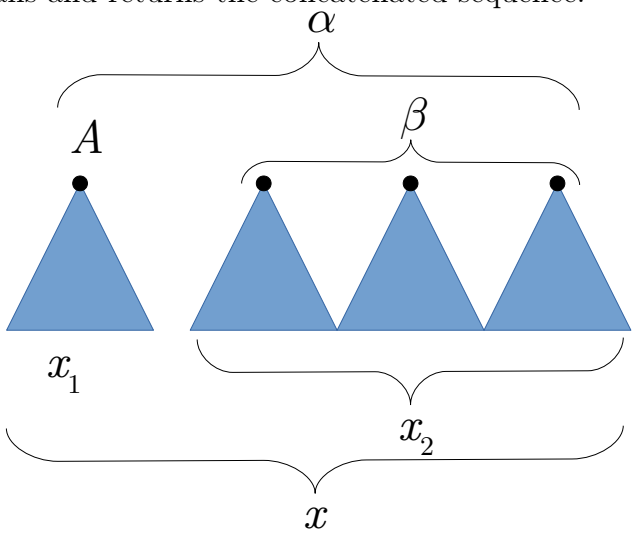
In Case 2, the sequence should start with a parse tree showing that $a \Rightarrow^* a$, followed by a sequence of parse trees showing that $\beta \Rightarrow^* z$. The algorithm creates a leaf node with just $a$ (to show that $a \Rightarrow^* a$), prepends it to the sequence of parse trees returned by the recursive

call `parse(`$\beta$`, z)`, and returns the resulting sequence.

In Case 3, $\alpha$ consists of one symbol $A$, so the algorithm must return a sequence containing one parse tree with $A$ at its root. The recursive call `parse(`$\gamma$`, x)` returns a sequence of trees with the symbols of $\gamma$ as their roots. The algorithm creates a new node with symbol $A$ to represent the derivation step $A \Rightarrow \gamma$. The trees returned by the recursive call, with the symbols of $\gamma$ at their roots, are used as the children of the new node:




In Case 4, the recursive call `parse(`$A$`, `$x_1$`)` returns a sequence of one tree whose root is $A$ and whose leaves are the symbols of $x_1$. The recursive call `parse(`$\beta$`, `$x_2$`)` returns a sequence of trees whose roots are the symbols of $\beta$ and whose leaves, concatenated together, are the symbols of $x_2$. Since the sequence to be returned should have the symbols of $\alpha$ as the roots of the trees, and the leaves of the trees concatenated together should be the symbols of $x$, and since $\alpha = A\beta$ and $x = x_1x_2$, the algorithm just concatenates the two sequences returned by the two recursive calls and returns the concatenated sequence.




The Scala return type of `parse` needs to be `Option[Seq[Tree]]`. An `Option[A]` can be either a value of type `A` wrapped in `Some()`, or the value `None`. The parser returns `None` when $\alpha$ does not derive $x$, so there is no sequence of trees to be returned. These are the cases in which the earlier algorithm returned `false`. In the cases in which the earlier algorithm returned `true`, the new algorithm returns `Some(s)`, where `s` is the sequence of trees to be returned. The value type of the memoization map is changed in the same way, from `Boolean` to `Option[Seq[Tree]]`.

*Remark.* The parsing algorithm that we have discussed is a variant of the *CYK parsing algorithm*

named after its inventors John Cocke, Daniel Younger and Tadao Kasami. Daniel Younger is a Professor Emeritus in Combinatorics and Optimization at Waterloo. The algorithm is usually presented in the form of dynamic programming. In the discussion here, we have replaced dynamic programming with memoization, which is computationally equivalent, but more clearly enumerates the various cases that the algorithm considers. Most presentations of the CYK algorithm assume that the grammar has been transformed to an equivalent grammar in a restricted form called Chomsky normal form, which simplifies the dynamic programming formulation. The variant we have discussed here works with any context-free grammar, without requiring transformation to Chomsky normal form.

## 8.3 Other parsing algorithms

The CYK algorithm has two important advantages. First, it can be understood intuitively by systematically considering the possible derivations in the various cases, and hopefully you understand fully how it works after reading these notes and implementing it in the assignment. Second, it works with *any* context-free grammar. A disadvantage is its $O(n^3)$ running time: although this will be sufficient to parse the small Lacs test programs in this course, it would be too inefficient for an industrial strength compiler.

The *LR(k) and LL(k)* classes of parsing algorithms are popular in practice, especially for the case when $k = 1$. The primary advantage of these algorithms is their guaranteed $O(n)$ running time. A disadvantage is that they can parse only with certain classes of grammars. In particular, LL($k$) parsers cannot parse with grammars that build left-associative parse trees: an LL($k$) parser must parse expressions such as $3 - 2 - 1$ as $3 - (2 - 1) = 2$, rather than as $(3 - 2) - 1 = 0$, as dictated by standard conventions for subtraction. LR($k$) parsers can parse grammars with both left-associative and right-associative operators, but there are still many grammars that they cannot parse. This limitation of LL($k$) and LR($k$) parsers to a restricted set of grammars can be used to advantage as an approximate test for grammar ambiguity. It is undecidable in general whether a context-free grammar is ambiguous, but all LL($k$) and LR($k$) grammars are known to be unambiguous and it is decidable whether a grammar is LL($k$) or LR($k$). Thus, if we verify that a grammar to be used for a programming language is LR($k$), such as the Lacs grammar, then we can be sure that it is unambiguous. However, if a grammar is not LL($k$) or LR($k$), then it could be ambiguous or unambiguous. A second disadvantage of the LL($k$) and LR($k$) parsing algorithms is their complexity: to understand them in full and convince oneself that they correctly find a derivation whenever one exists requires proving non-trivial theorems. The LL($k$) and LR($k$) algorithms are covered in full in CS 444.

Like the CYK algorithm, *Earley's* parsing algorithm can parse with *any* context-free grammar. Its worst-case running time is $O(n^3)$ for ambiguous grammars, $O(n^2)$ for unambiguous grammars, and $O(n)$ for a class of grammars that includes almost all LR($k$) and LL($k$) grammars. If a grammar can be used *at all* by an LR($k$) or LL($k$) parser, then an Earley parser can probably parse with it in the same linear time as the LR($k$) or LL($k$) parser. The functioning of the Earley parser is mostly intuitive, although proving its correctness does require a non-obvious theorem. Although Earley's parsing algorithm is somewhat more complicated than the CYK algorithm, and therefore an explanation is not included in these notes, clear and readable explanations can be found in the first two sections of

the paper John Aycock, R. Nigel Horspool: Directly-Executable Earley Parsing. Compiler Construction 2001: 229-243 or on the website Earley Parsing Explained of Loup Vaillant.

## 8.4 The correct prefix property

When a parser successfully parses a given input, it can return a parse tree or a derivation as proof that the input is a word in the language. When the input is *not* a word in the language, we would like a detailed error message indicating why the input could not be parsed.

One form of such feedback is an indication of the position in the input at which the parser got stuck. More precisely, the parser returns the longest prefix of the input that is also a prefix of some word in the language. Such a prefix is *correct* in the sense that, if it is completed with the right ending, it can become a word in the language. Formally, suppose that:

- $w$ is the input to a parser,
- $w$ can be decomposed as $w = xaz$, where $x$ and $z$ are sequences of terminal symbols and $a$ is a terminal symbol,
- there exists some sequence $y$ of terminal symbols such that $xy$ is a word in the language, and
- there is no sequence $y'$ of terminal symbols such that $xay'$ is a word in the language.

In this case, $x$ is the longest correct prefix and $a$ is informally the position of the bug. Then a parser has the *correct prefix property* if when parsing input $w = xaz$, it reports an error at the position of $a$. The error report enables the user to locate the bug in the input.

Earley's parsing algorithm and the LL($k$) and LR($k$) parsing algorithms have the correct prefix property but the CYK parsing algorithm does not. For practical purposes, to enable you to debug invalid programs that do not satisfy the Lacs grammar, the handout code includes an implementation of an Earley parser (without source!), so that when your CYK parser rejects a given input, the provided Earley parser can indicate the position of the bug in the input.