Context-Free

Grammars !

What is a grammar?

- MW: A system of rules that defines the structure of a language.

- CS: Very precise, theoretically sound rules defining computer languages.
  - There are many classes of grammars, this is the focus of a Theory of Computation course.

- A language is the set of all strings that satisfy the rules of a grammar.

- Intuitively, a context-free grammars is made of:
  - **Terminals -** the grammar's "alphabet" of symbols
  - **Non-terminals -** syntactic variables where each represents a set of strings of terminals
  - **Production rules -** the definition of a non-terminal that specifies its set of strings
  - **Start symbol -** the single non-terminal which represents the set of all strings in language

Example

- Terminals: `{'the', 'a', 'person', 'dog', 'spoke', 'ran', 'ate'}`
  - Our language's alphabet of **symbols**.
- Non-terminals: `{ phrase, article, noun, verb }`
  - These will be defined by production rules!
- Production rules:

```
phrase  -> article noun verb
article -> 'the' | 'a'
noun    -> 'person' | 'dog'
verb    -> 'spoke' | 'ran' | 'ate'
```

- Start symbol: `phrase`
  - Start symbol defines the set of all possible strings in our language.

Production Rules

- Each **non-terminal** is defined by a **production rule.**
  - Begins with the non-terminal being defined, followed by an arrow, followed by definition.

- Consider the production rule: `noun -> 'person' | 'dog'`
  - The non-terminal being defined is `noun`, the definition is `'person' | 'dog'`
  - The vertical bar symbol denotes a union relationship, or "**or**".
  - This definition can be read as a `noun` is either the string `'person'` ***or*** `'dog'`.

- Another production rule: `phrase -> article noun verb`
  - The non-terminal being defined is `phrase`, the definition is `article noun verb`
  - When one symbol follows another, such as `article noun`, concatenation is implicit.
  - This definition can be read as a `phrase` is an `article` ***and then*** a `noun` ***and then*** a `verb`.

Generating Strings from a Grammar

Goal: Generate a string of all non-terminals.

Strategy: Expand the non-terminals of a "working string".

```
phrase  -> article noun verb
article -> 'the' | 'a'
noun    -> 'person' | 'dog'
verb    -> 'spoke' | 'ran' | 'ate'
```

1. To generate a string accepted by a grammar, your initial "working string" is just the *start symbol*.
   - In the grammar right, the *start symbol* is phrase.
   - In COMP211 grammars, the *start symbol* is always underlined.

2. Choose a non-terminal in the "working string" and *expand* it by substituting an instance of the non-terminal's production rule.
   - When there is *union* ("or"), designated by the vertical bar symbol, choose only *one* of the two or more possible alternatives.
   - When there is a sequence of non-terminals or terminals, choose the entire sequence.

3. When all symbols in the working string are terminals, the process terminates. Otherwise, repeat Step 2.

Production Rules Continued

- The union "or" operator has **lower** precedence than concatenation "and then"
- Consider:

```
phrase  -> article noun verb | phrase 'and' phrase
```

- Compare:

```
phrase  -> (article noun verb) | (phrase 'and' phrase)
```

- These two definitions are equivalent because concatenation has higher precedence.
- Parenthesis can be used to override the standard order of operations .
  Consider: `phrase  -> article noun (verb | phrase) 'and' phrase`
  What is *terrifying* about *this* definition?
  It's infinitely recursive! We will avoid this dilemma.

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

1. To generate a string accepted by a grammar, your initial "working string" is just the *start symbol*.

Working String: phrase

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase
```

2. Choose a non-terminal in the "working string" and *expand* it by substituting an instance of the non-terminal's production rule.

Here we have a choice. Either we substitute with the string of symbols:

article noun verb

OR

phrase "and" phrase

Working String: phrase

Union has lower precedence than concatenation!

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

3. When all symbols in the working string are terminals, the process terminates. Otherwise, repeat Step 2.

~~phrase~~

Working String: phrase "and" phrase

Are all symbols terminals? No!

Only "and" is a terminal. We need to repeat Step 2.

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "dog" | "cat" | "man" | "woman"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

2. Choose a non-terminal in the "working string" and *expand* it by substituting an instance of the non-terminal's production rule.

Let's choose the first alternative:

article noun verb

~~phrase~~

Working String: phrase "and" phrase

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

3. When all symbols in the working string are terminals, the process terminates. Otherwise, repeat Step 2.

~~phrase~~

~~phrase~~ "and" phrase

Working String: article noun verb "and" phrase

Are all symbols terminals? No!

Only "and" is a terminal. Repeat Step 2.

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun

verb
                                 led" |
             "waved"
```

Choose either alternative!

2. Choose a non-terminal in the "working string" and *expand* it by substituting an instance of the non-terminal's production rule.

~~phrase~~

~~phrase~~ "and" phrase

Working String: article noun verb "and" phrase

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

3. When all symbols in the working string are terminals, the process terminates. Otherwise, repeat Step 2.

~~phrase~~

~~phrase~~ "and" phrase

~~article~~ noun verb "and" phrase

Working String: "a" noun verb "and" phrase

This process repeats in an intuitive way, so we're going to fast forward to the point where the process terminates...

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

3. When all symbols in the working string are terminals, the process terminates. Otherwise, repeat Step 2.

~~phrase~~

~~phrase~~ "and" phrase

~~article~~ noun verb "and" phrase

"a" ~~noun~~ verb "and" phrase

"a" "person" ~~verb~~ "and" phrase

"a" "person" "waved" "and" ~~phrase~~

"a" "person" "waved" "and" ~~article~~ noun verb

"a" "person" "waved" "and" "the" ~~noun~~ verb

"a" "person" "waved" "and" "the" "dog" ~~verb~~

Working String: "a" "person" "waved" "and" "the" "dog" "smiled"

When the working string reaches a point of *all* terminals you've generated a string accepted by the grammar!

Example: Using a Grammar to Generate Strings

```
phrase    -> article noun verb |
             phrase "and" phrase

article   -> "the" | "a"

noun      -> "person" | "dog"

verb      -> "spoke" | "ran" |
             "screamed" | "smiled" |
             "waved"
```

Rather than rewrite a "working string" over and over, it's more convenient to draw out a ***derivation tree***.

The ***root node*** is the *start symbol* and each child is a symbol from an instance of its production rule.

Derivation complete once all ***leaf nodes*** are *terminals*.