Lecture 6

Beyond Well-Formedness

Suppose we want to implement the SL_k learner from the previous lecture in Python. This involves two components, a mechanism for collecting all the k-grams in a string, and a data structure for memorizing all the k-grams seen so far across all read strings. A bigram collection mechanism was already part of the set-based scanner we devised in Cha. 3.

```python
def string_to_bigrams(w):
    """Convert string into non-repeating list of bigrams."""
    bigram_list = []
    for i in range(len(w)-1):
        current_bigram = w[i] + w[i+1]
        if current_bigram not in bigram_list:
            bigram_list.append(current_bigram)
    return bigram_list
```

This function can easily be extended to arbitrary k. In order to get a full learner, we only need a function that takes a finite sample of strings and runs the k-gram collection function on each (augmented) string in that sample. Below is one implementation of this for a bigram leaner.

```python
def bigram_learner(language):
    """Induce a strictly 2-local grammar from a finite sample of strings"""
    grammar = set([])
    for w in language:
        grammar = grammar.union(set(string_to_bigrams(augment_string(w))))
    return grammar
```

While this works, it is somewhat inelegant. Intuitively, the SL_2 learner above still has to use the scanner to extract the list of bigrams, yet the function does not directly call the bigram scanner but merely some of its subroutines. Consequently the program does not correspond to how we think about the actual computation. On a more practical level, all optimization steps and safeguards that might have been added to the scanner will be lost if they're not part of these subroutines. A better approach would thus be one where the scanner handles the job of extracting the bigrams in the most efficient way possible, and the learner directly calls the scanner.

You might be wondering how this is supposed to work if the scanner works incrementally, the way we originally described it. In this case the scanner never uses the function for collecting a set of bigrams, so how could it possibly supply such a set to the learner? Today we will see that there is a way to write the scanner in a more abstract way that allows it to carry out various tasks. The basic principles of the scanner will be preserved, but we add a few parameters to the mix. For the purposes of software engineering, this makes for a very versatile and easily maintained code base. But it is also interesting from a theoretical perspective, because it shows among other things that it does not matter all that much for formalisms whether they are categorical (well-formed VS ill-formed) or probabilistic (degrees of acceptability).

1 The Algebra of Composite Values

1.1 A Functional Recipe for Composite Values

Suppose that we want to compute the well-formedness of a given string with respect to some grammar. We have already seen two ways of doing this. In the definition of (positive) strictly k-local grammars, we let $L(G) := \{w \mid k\text{-grams}(\hat{w}) \subseteq G\}$, which is also the foundation of the set-based scanner: check whether the set of k-grams of the string is a subset of the grammar. The incremental scanner, on the other hand, checks at each step whether the current bigram is in the grammar. If one abstracts away from the temporal component of the scanner while keeping the piece-wise application, one can also define a well-formedness function f that computes the grammaticality value of a string from the grammaticality values of its k-grams.

Below is an example of a formula evaluating the string *aba* with respect to the bigram grammar $\{\rtimes a, ab, ba, b\ltimes\}$, which generates the language $(ab)^+$.

$$\begin{aligned} f(aba) &= g(\rtimes a) \times g(ab) \times g(ba) \times g(a\ltimes) \\ &= 1 \times 1 \times 1 \times 0 \\ &= 0 \end{aligned}$$

Here g is a function that returns 1 iff the bigram is contained in the grammar, and 0 otherwise. The function f then multiplies all these values in order to determine overall well-formedness. Note that we could have used other mathematical operations:

$$\begin{aligned} f(aba) &= 1 \times (1 \times (1 \times 0)) & &= 1 \min (1 \min (1 \min 0)) & &= 1 \wedge (1 \wedge (1 \wedge 0)) \\ &= 1 \times (1 \times 0) & &= 1 \min (1 \min 0) & &= 1 \wedge (1 \wedge 0) \\ &= 1 \times 0 & &= 1 \min 0 & &= 1 \wedge 0 \\ &= 0 & &= 0 & &= 0 \end{aligned}$$

So there are actually three different functions that all compute the grammaticality of a string.

While the functions behave exactly the same if g returns only 1 or 0 as values, they diverge in other cases. Suppose, for instance, that g tells us for each k-gram it's acceptability value, encoded as some real number in the interval $[0, 1]$. Then the choice of f produces different values. If f uses the logical operator $\wedge$ for composing values, we won't any output at all in most cases because $\wedge$ is only defined for argument that are 0 or 1. For the other two functions, we get vastly different values.

$$\begin{aligned}
f_\times(aba) &= g(\rtimes a) \times (g(ab) \times (g(ba) \times g(a\ltimes)))\\
&= 0.9 \times (0.25 \times (0.5 \times 0.75))\\
&= 0.9 \times (0.25 \times 0.375)\\
&= 0.9 \times 0.09375\\
&= 0.084375\\
f_{\text{min}}(aba) &= g(\rtimes a) \text{ min } (g(ab) \text{ min } (g(ba) \text{ min } g(a\ltimes)))\\
&= 0.9 \text{ min } (0.25 \text{ min } (0.5 \text{ min } 0.75))\\
&= 0.9 \text{ min } (0.25 \text{ min } 0.5)\\
&= 0.9 \text{ min } 0.25\\
&= 0.25
\end{aligned}$$

If we interpret g as a function from k-grams to acceptability values, then the function $f_\times$ treats overall acceptability as the product of the individual acceptability values of each k-gram. The function f_{min}, on the other hand, tells us the value of the least acceptable k-gram that occurs in the string. So these functions do very different things, but they do not look all that different from any of the other we have seen so far. They may compute different values, but they apply a helper function g to each k-gram and the compute some compound value from the individual values returned by g.

This strategy isn't even restricted to numerical values, either, just about anything of interest can be computed by some f in this fashion. For example, here is a function $f_\cup$ that computes the set of bigrams for a string.

$$\begin{aligned}
f_\cup &= g(\rtimes a) \cup (g(ab) \cup (g(ba) \cup g(a\ltimes)))\\
&= \{\rtimes a\} \cup (\{ab\} \cup (\{ba\} \cup \{a\ltimes\}))\\
&= \{\rtimes a\} \cup (\{ab\} \cup \{ba, a\ltimes\})\\
&= \{\rtimes a\} \cup \{ab, ba, a\ltimes\}\\
&= \{\rtimes a, ab, ba, a\ltimes\}
\end{aligned}$$

What we have here, then, is a general template for computing a string's compound value from the values of its parts, which happen to be k-grams. Depending on what those base values are (regulated through the choice of function g) and how we combine them (determined by the choice of f), we get different compound values.

1.2 Monoids

Despite the differences in values, the compound functions all behave pretty much the same on an abstract level. First, f operates on a fixed domain of values, and this domain is closed under f. Remember that a set S is closed under an operation o iff $o(s)$ is an element of S for every $s \in S$. The domain of grammaticality values, for instance, is $\{0, 1\}$, and applying $f_\times$, f_{min}, and $f_\wedge$ within this domain can only produce 0 and 1 as outputs. Similarly, the real interval $[0, 1]$ is closed under $f_\times$ and f_{min}. For $f_\cup$, the domain is the set of all sets of k-grams over alphabet Σ — $\wp(\Sigma^k)$ in mathematical notation — and obviously $f_\cup$ cannot produce a value that lies outside that set. We see, then, that each function is a map from a fixed domain into that very same domain.

It is also the case that the operation used by f to compute the composite values is associative. Remember, a function $\circ$ is associative iff $a \circ (b \circ c) = (a \circ b) \circ c$; in other words, brackets can be dropped without changing the composite value. This is clearly the case for $\times$, min, $\wedge$, and $\cup$. In all the examples above, we could have computed the values from left to right instead without affecting the final outcome. The order of application thus is irrelevant for all the variants of f in our examples.

Finally, every operation $\circ$ has an identity element $\mathbf{1}$ such that $a \circ \mathbf{1} = \mathbf{1} \circ a = a$ for all a. For $\times$, min and $\wedge$ this identity element is literally 1:

$$\begin{aligned} 0.75 \times 1 = 1 \times 0.75 &= 0.75 \\ 0.75 \min 1 = 1 \min 0.75 &= 0.75 \\ 0 \wedge 1 = 1 \wedge 0 &= 0 \end{aligned}$$

Can you explain why $\emptyset$ must be in the domain of $f_\cup$?

For $\cup$, the identity element is $\emptyset$ instead:

$$\{ba, a\ltimes\} \cup \emptyset = \emptyset \cup \{ba, a\ltimes\} = \{ba, a\ltimes\}$$

If you have some background in higher algebra, you may already know what kind of object is characterized by these properties: *monoids*.

Definition 6.1 (Monoid). A *monoid* $M := \langle S, \otimes \rangle$ is a set S with an operation $\otimes$ such that:

- S is closed under $\otimes$, and
- $\otimes$ is associative, and
- $\otimes$ has an identity element $\mathbf{1} \in S$.

Use $\otimes$ as a general placeholder, we can combine all the functions we have seen so far into a single template for a k-gram function f that computes composite values.

$$f_\otimes(a_1 \cdot a_2 \cdots a_{n-1} \cdot a_n) := g(a_1 \cdots a_k) \otimes \cdots \otimes g(a_{n-k} \cdots a_n)$$

Each $g(a_i \cdot a_{i+k})$ must be an element of S, and $\otimes$ is some function over S such that $\langle S, \otimes \rangle$ is a monoid.

The monoid perspective is appealing for two reasons. On a theoretical level it highlights computational similarities between superficially different processes. Computing the grammaticality of a string isn't all that different from computing its set of k-grams. The switch from a categorical grammar to a probabilistic one — which is a contentious issue among linguists — is tantamount to replacing the set $\{0, 1\}$ by the real interval $[0, 1]$, everything else stays exactly the same. Therefore every property that relies only on $\langle \{0, 1\}, \times \rangle$ being a monoid immediately carries over to $\langle [0, 1], \times \rangle$ and is consequently preserved by the switch from a categorical to a probabilistic formalism. Quite simply, monoids are yet another tool for keeping our results general and widely applicable without sacrificing mathematical rigor or linguistic substance.

Their generality is also what makes monoids interesting for a very different crowd: programmers. Programmers want their code to be simple, easily maintainable, and free of unnecessary code duplication. Yet the code should also be flexible so that it can be quickly adapted to new problems as they arise. Monoids offer a way of attaining both goals. One can write a basic program that works with any kind of monoid, and applying this program to a specific problem instance requires no more than defining the appropriate monoid. Going back to our problem at the outset of this chapter, this strategy allows us to write an incremental scanner that can be used directly by the learner to retrieve a string's set of k-grams.

2 Implementing a Monoid Scanner

A monoid scanner works exactly like the function $f_\otimes$ above: given a function for computing the base values and a suitable operation for combining the base values, it determines the composite value in a recursive fashion. This can be translated into python almost literally, with the actual code being only 5 lines.

```python
def monoid_scanner(base_value, compose, grammar, w):
    """
    Generalized bigram scanner that assigns strings
    a value over a given monoid.

    Arguments:
    base_value -- maps a bigram to an element of the monoid
    compose    -- implements the monoid operation
    grammar    -- set of licit bigrams
    w          -- the input string
    """
    # [Base Step]
    # value of a single bigram
    if len(w) == 2:
        return base_value(w, grammar)
    # [Recursion]
    # scan string from left to right and compose values
    else:
        return compose(
            base_value(w[0:2], grammar),
            monoid_scanner(base_value, compose, grammar, w[1:len(w)]))
```

Note that for the sake of exposition, the monoid scanner only works with bigrams, but it could easily be generalized to work with arbitrary k-grams. The string augmentation function has also been omitted here, and the scanner does not include any safety checks for the input (e.g. to avoid type errors).

In order to turn the monoid scanner into a standard recognizer, we have to supply two functions that, respectively, determine for each bigram whether it is licensed by the grammar and compute compound grammaticality values.

```python
def boolean_base(w, grammar):
        return w in grammar


def boolean_compose(s, t):
        return s & t
```

The probabilistic version is almost exactly the same, we only have to switch to a different function for the base value (assuming that a probabilistic grammar is a dictionary where every k-gram acts as the key for a real number).

```python
def prob_base(w, grammar):
    return grammar.get(w, 0)
```

The set of all bigrams in the string is computed by feeding the two functions below into the monoid scanner.

```python
def set_base(w, *args):
    return set(w)


def set_compose(s, t):
    return s.union(t)
```

And if we want to know how many bigram tokens occur in the string, we switch to yet another pair of functions.

```python
def token_base(w, *args):
    return 1


def token_compose(s, t):
    return s + t
```

So now we have four different algorithms, in less than 20 lines of code! Every function is incredibly small and easy to understand, which makes debugging almost trivial. We can also mix and match these functions to create new algorithms. We could even combine them into bigger functions, for instance if we wanted to keep track of the number of tokens for each individual bigram. The modularity of this approach also means that any optimizations and tweaks we do to the monoid scanner benefits every algorithm we implement this way. This saves a lot of time and effort, and there's no unnecessary duplication of code or work.

That's not to say that the monoid scanner is always the best solution. Generality always comes at a price, and in this case it means that optimizations primarily take the form of optimizing the monoid scanner. This may make it impossible to implement certain speed hacks that work well for one problem but fail miserably for another one. This can be alleviated to some extent by activating these tweaks only if specific functions are being used, but that makes the code base more complicated, of course. So there is no elegant way around the trade-off between speed and generality, but for our purposes speed is an afterthought at best, whereas generality is one of the main goals.

The monoid perspective is an incredibly fruitful one for computational linguistics, and we have only seen the tip of the iceberg. Monoids can be combined in a specific manner to form an algebraic structure known as a *semiring*, and these play a very important role in parsing and the analysis of OT. We won't have time to explore these topics in great detail, unfortunately, so for now we have to be content with the fact that monoids at the very least simplify the code for the bigram learner at the beginning of this section:

```python
def bigram_learner(language):
    """Induce a strictly 2-local grammar from a finite sample of strings"""
    grammar = set([])
    for w in language:
        grammar = grammar.union(
            monoid_scanner(
                set_base,
                set_compose,
                grammar,
                augment_string(w)))
    return grammar
```