CSE396 Lecture Tue. 3/23: Structural Induction and Context-Free Grammars

The natural numbers $\mathbb{N}$ can be "defined" via the context-free grammar

$$S \rightarrow 0 \mid S+1$$

where the $+$ is a terminal symbol---you can do the addition later. [Just FYI, if you literally want to derive numbers in tally notation, you can use a similar "list-of" pattern in a linearly-extending fashion:

$$\begin{aligned} S &\rightarrow BS \mid F \\ B &\rightarrow \cancel{1111} \\ F &\rightarrow \cancel{1111} \mid 1111 \mid 111 \mid 11 \mid 1 \mid \end{aligned}$$

Note: cave people did not have zero. No, I am not about to show another Geico commercial.] [I diddled with the garmmar and showed how if you combine the rules for B and F into one variable (or make $B \rightarrow F$ an option) then you allow strings like $\cancel{1111}$———111 $\cancel{1111}$—11 1 $\cancel{1111}$ which you might not regard as internally-sound tally strings. Whereas, the rules as-given uphold that only the Final block can be a "short tally" less than 5.]

When you do the "$n-1$ to n" or "n to $n+1$" style of induction as taught in CSE191, you can picture it as working "on" the first grammar above. You get a linearly extending pattern. The advantage of **Structural Induction** (**SI**) is that you can have tree-branching patterns, and more---all based on a context-free grammar, and without the "crutch" of using natural numbers.

There is an imperfect analogy to object-oriented programming that I try to promote. If you derive

```
<class> → <subclass> → <subclass of that> → <subclass of that>...
```

it's like induction on $\mathbb{N}$. But you can make a branching hierarchy of classes. The **factory** design patterns usually provide branching on-the-fly generation of objects. The mode of thinking that aligns with **SI** is to consider which properties of the base class are preserved by these generations, and which further properties are propagated.

We have seen a proof by SI already in this course, when I proved that every regular expression r can be converted into an NFA N_r such that $L(r) = L(N_r)$. That proof did not mention a "natural number n" but worked directly on the structure of r, technically according to a grammar such as:

$$R ::= \varnothing \mid \epsilon \mid \boldsymbol{a} \mid \boldsymbol{b} \mid (R+R) \mid (R \cdot R) \mid \left(R^*\right).$$

(Or: the grammar on the HW, but ignoring the rule $R ::= (R)$ since it adds no structure.) A proof by SI always works ***from*** a grammar ***to*** a target property P that holds for every object that the grammar G can generate. It proves: Every object generated "syntactically" via G has the "semantic" property P. If the latter is your "ground truth" then this means proving that the grammar generates no false positives.

Definition: Let T be any target language, such as T_P for all true positives of P.

- The grammar G is **sound** if $L(G) \subseteq T$. That is, G avoids false positives.
- G is **comprehensive** if $L(G) \supseteq T$.

Structural induction is a technique for proving *soundness*. Sometimes you can see comprehensiveness, but not always. In the case of every regular expression having an equivalent NFA, it took a whole separate proof to do the converse---to show that every NFA (or DFA or GNFA) has an equivalent regular expression. We will emphasize SI soundness proofs but "soft-pedal" the harder comprehensiveness proofs, which go ***from*** parsing strings ***to*** building derivations in the grammar.

I like to **personify** each variable of a grammar---this is a "poetic trope" but accords with human thinking while designing stuff. Here is the "proof script" in general given a grammar $G = (V, \Sigma, \mathcal{R}, S)$ and target language T:

1. Assign to each variable A a property P_A of strings it can derive. You can think of P_A as the "meaning" you give to A---though it need not be a comprehensive meaning, just enough to work with the other variables. You can personify it in the form,

   "P_A: Every x that I derive is such that..."

2. The meaning P_S of the start symbol should imply $x \in T$. Often you can simply take P_S to be the assertion, "Every x that I derive belongs to T," but sometimes you need to use a sharper property. [Technically this is called "strengthing" or "loading" the **induction hypothesis** (IH), but what I call "SI style" tries to make it more transparent.]

3. For each variable A and each rule $A \to X$ where $X \in (\Sigma \cup V)^*$, begin the body of the script by writing, "Suppose $A \Longrightarrow^* x$ using this rule first" (you can abbreviate the last four words to "utrf" or to "utpf" for "using this production first").

4. If the rule's whole right-hand side X is an all-terminal string, you immediately need to check that X obeys what P_A says. This is a base case.

5. Otherwise, X has one or more *occurrences* of variables. Note that the variable A itself can occur on the right-hand side of the rule. This may seem like circular logic but it's not. It or some other variable(s) can occur more than once. Regardless, for each occurrence---call it B_i---let u_i (or any letter you like) stand for a corresponding terminal substring that it derives. You have to be general and you have to include any terminals in the rule in the right order. If there are k occurrences of variables on the right-hand side (RHS) of the rule $A \to X$, then the script says to enunciate it by saying:

   "Then $x =: \ldots u_1 \ldots u_i \ldots u_k \ldots$ where $B_1 \Longrightarrow^* u_1$ and ... and $B_i \Longrightarrow^* u_i$ and ... $B_k \Longrightarrow^* u_k$."

6. Now we apply the corresponding properties $P_{B_1}, \ldots, P_{B_i}, \ldots, P_{B_k}$ of the variables on the RHS. Again, two of those occurrences may be the same variable, so you will use the same property twice, but you will generally be using the property *on* different substrings. Here is the scripted way to enunciate this:

   "By **IH** P_{B_1} on RHS, the substring u_1 satisfies..., and by **IH** P_{B_2} on RHS, u_2 satisfies ..."

   And so on through all the substrings.

7. Finally, you need to *argue* that the fact of each substring obeying its property *ensures* that the resulting string x (whatever it is---it is general) obeys the original property P_A of the variable A on the left-hand side (LHS) of the rule. You can then summarize by saying, "This upholds P_A on LHS."

8. When you show that **every** rule upholds the stated property of its left-hand variable, then you uphold P_S in particular, which is what entitles you to conclude "$L(G) \subseteq T$ by structural induction."

[cover examples from handout and previous-year notes:
https://cse.buffalo.edu/~regan/cse396/CSE396SI.pdf
https://cse.buffalo.edu/~regan/cse396/CSE396lect040518.pdf
https://cse.buffalo.edu/~regan/cse396/CSE396lect032416.pdf
as time permits, using the last as a parsing analogy for properties of programming syntax.