

Parsing

Ernest Kirstein

April 3, 2015

Parsing, in the most abstract sense, is the process of attempting to fit a string (or list of tokens) to one or more structured representations. Typically, the set of possible structured representations, R , is defined by some formal grammar specification. [1–4] A parsing algorithm can be thought of as a routine which reduces R to only those representations which are valid for a particular string.

Definition 1.

$$P(s, R) = \{\forall r \in R : \text{valid}(r, s)\}$$

That explanation is a bit dense, so allow me to explain. Let's take the following context free grammar, G , as an example.

$$\begin{aligned} S &\rightarrow a \\ S &\rightarrow bS \end{aligned}$$

The structured representations specified by a CFG are all rooted, ordered trees called parse trees. [3] In this case, the set of parse trees defined under this CFS all have a root node which corresponds to the symbol S . All S nodes have either one or two child nodes: they can have a terminal child node corresponding to the symbol a ; or they can have a terminal and non terminal child node corresponding to b and S respectively.

The following are all examples of parse trees are in R_G :

$$\begin{aligned} &(S, \{a\}) \\ &(S, \{b, (S, \{a\})\}) \\ &(S, \{b, (S, \{b, (S, \{a\})\})\}) \\ &\dots \text{ etc.} \end{aligned}$$

All the trees in R_G can be generated by recursively apply the rules in G to non-terminal symbols according to their corresponding rules (more on that later). [3]

So now let's try to parse the string "bba" in this grammar: $P(\text{"bba"}, R_G)$. To first reduce R_G , we might consider only those structured representations which have less than 4 terminal nodes:

$$\begin{aligned} (S, \{a\}) \\ (S, \{b, (S, \{a\})\}) \\ (S, \{b, (S, \{b, (S, \{a\})\})\}) \end{aligned}$$

And to complete the parsing, we can simply scan those 3 representations and determine which ones (if any) correspond to "bba". We see that $(S, \{b, (S, \{b, (S, \{a\})\})\})$ (let's label it, r_v) is the only valid representation. So we can resolve $P(\text{"bba"}, R_G)$ to the set containing only r_v . This is, of course, a very ad hoc algorithm. There are much more robust approaches that will be discussed shortly.

Parsing - Divide and Conquire

Let's look at that last algorithm a little more closely. What allowed us to conclude that r_v was the only valid representation by searching only the 3 parse trees? Well, all the representations longer than 3 aren't valid so... It seems intuitive, but let's be explicit:

Theorem 1. *Let s be a string and let R , A , and B be sets of representations such that $R \subseteq A \cup B$ and $B \subseteq R$. If $\forall x \in A : \neg \text{valid}(x, s)$ then $P(s, R) = P(s, B)$.*

Proof. Let s be a string and let R , A , and B be sets of representations such that $R \subseteq A \cup B$ and $B \subseteq R$. For contradiction, let us assume that $\forall x \in A : \neg \text{valid}(x, s)$ and $P(s, R) \neq P(s, B)$. So either there must be some $r \in R$ and $r \notin B$ that is a valid representation of s or there must be some $b \in B$ and $b \notin R$ that is a valid representation of s . The later is impossible because B is a subset of R . And if $r \in R$ and $r \notin B$ then $r \in A$ since $R \subseteq A \cup B$ and $B \subseteq R$. But that contradicts our assumption that $\forall x \in A : \neg \text{valid}(x, s)$. \square

That's a useful theorem because it allows us to divide the problemset and efficiently tackle parsing problems recursively. Let's consider another example CFG, G :

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow bS \\ S &\rightarrow \epsilon \end{aligned}$$

Which defines the set of potential parse trees R_G . Now let's parse the string "bab": solving $P("bab", R_G)$.

We'll start by dividing the set R_G into three parts, R_{Ga} , R_{Gb} and $R_{G\epsilon}$ where:

$$\begin{aligned} R_{Ga} &= \{\forall r \in R_G : \exists n, r = (S, \{a, n\})\} \\ R_{Gb} &= \{\forall r \in R_G : \exists n, r = (S, \{b, n\})\} \\ R_{G\epsilon} &= \{(S, \{\})\} \end{aligned}$$

More simply, R_{Ga} contains all parse trees that are produced by first following the production rule $S \rightarrow aS$; R_{Gb} by first following the second production rule (for b); and $R_{G\epsilon}$ by following the third production rule first which results in the empty string.

We can rule out R_{Ga} and $R_{G\epsilon}$ because "bab" does not start with "a" and is not an empty string. Applying theorem 1, we can reduce the problem by noting that $P("bab", R_G) = P("bab", R_{Gb})$. Then we can further divide R_{Gb} into R_{Gba} , R_{Gbb} and $R_{Gb\epsilon}$ where:

$$\begin{aligned} R_{Gba} &= \{\forall r \in R_{Gb} : \exists n, r = (S, \{b, (S, \{a, n\})\})\} \\ R_{Gbb} &= \{\forall r \in R_{Gb} : \exists n, r = (S, \{b, (S, \{b, n\})\})\} \\ R_{Gb\epsilon} &= \{(S, \{b, (S, \{\})\})\} \end{aligned}$$

And we can further narrow the problem by noting that $P("bab", R_{Gb}) = P("bab", R_{Gba})$. Applying this same logic again leads to the conclusion $P("bab", R_{Gba}) = P("bab", R_{Gbab})$. Then dividing R_{Gbab} once more we finally have narrowed down the problem to a single result:

$$\begin{aligned} R_{Gbabab} &= \{\forall r \in R_{Gbab} : \exists n, r = (S, \{b, (S, \{a, (S, \{b, (S, \{a, n\})\})\})\})\} \\ R_{Gbabbb} &= \{\forall r \in R_{Gbab} : \exists n, r = (S, \{b, (S, \{a, (S, \{b, (S, \{b, n\})\})\})\})\} \\ R_{Gbab\epsilon} &= \{(S, \{b, (S, \{a, (S, \{b, (S, \{\})\})\})\})\} \end{aligned}$$

The answer only $R_{Gbab\epsilon}$ since all valid strings represented in R_{Gbabab} and R_{Gbabbb} are longer than "bab". In the whole procedure, we have shown:

$$\begin{aligned} P("bab", R_G) &= P("bab", R_{Gb}) \\ &= P("bab", R_{Gba}) \\ &= P("bab", R_{Gbab}) \\ &= P("bab", R_{Gbab\epsilon}) \\ &= \{(S, \{b, (S, \{a, (S, \{b, (S, \{\})\})\})\})\} \end{aligned}$$

This example procedure shows how top down parsing really works. Since iterating over all possible parse-tree prerepresentations would be impossible, a parser can divide the set of possible parse trees into smaller and smaller subsets and eliminate as many solutions possible all at once.

1 Parsing - Constructively

TODO - Bottom Up Parsing

References

- [1] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.
- [2] F. D. Lewis. Recursive descent parsing. <http://www.cs.engr.uky.edu/~lewis/essays/compiler/rec-des.html>, 2002.
- [3] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, Inc., Sudbury, MA, 2001.
- [4] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction*. HarperCollins College Publishers, New York, NY, 1993.