1 Algorithms for Context-Free Languages

The parsing problem is, given a string w and a context-free grammar G, to decide if $w \in L(G)$, and if so, to produce a parse tree for it. How fast can this be done in general?

One can put G into a special form called *Chomsky normal form* that makes parsing easier. It's still too slow for large programs, but it can be useful for rapid prototyping in the early stages of language development. Any context-free grammar can be put into Chomksy normal form, roughly speaking.

Definition 1.1 *A context-free grammar* $G = (V, \Sigma, R, S)$ *is in* Chomsky normal form *if the right-hand sides of all rules have length 2.*

Note that if G is in Chomsky normal form then $L(G)$ cannot contain any strings of length 0 or 1.

Theorem 1.1 *For any context-free grammar* G *there is a context-free grammar* G' *in Chomsky normal form such that* $L(G') = L(G) - (\Sigma \cup \{e\})$*. Thus* $L(G)$ *and* $L(G')$ *agree on strings of length greater than one. Also,* G' *can be obtained from* G *in polynomial time.*

1.1 Transforming to Chomsky Normal Form on an Example

We will just show the transformation on an example to illustrate the idea of the proof. Consider this context-free grammar:

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow \epsilon$$

The start symbol is S.

1.1.1 Step 1

The first step is to eliminate rules whose right-hand side has length greater than 2.

Here there is just one rule like that: $S \to (S)$. This is split up into smaller rules whose right-hand sides have length 2. This yields the following grammar:

$$S \to SS$$

$$S \to (S_1$$

$$S_1 \to S)$$

$$S \to \epsilon$$

Note that S_1 is a new nonterminal. How would you split up the rule $S \to UVXY$?

1.1.2 Step 2

The next step is to eliminate rules whose right-hand side is ϵ. This is done by substituting them in other rules so that they are not needed.

- For example, the rule $S \to \epsilon$ can be substituted into $S \to SS$;
- we replace one of the S on the right-hand side with ϵ, yielding $S \to S$. Of course, this rule is unnecessary.
- We can also do this on the rule $S_1 \to S)$ yielding the rule $S_1 \to)$. This is a new rule that should be kept.
- After this step, all rules with ϵ on the right-hand side can be removed, giving this grammar:

$$S \to S$$

$$S \to SS$$

$$S \to (S_1$$

$$S_1 \to S)$$

$$S_1 \to)$$

Of course, the first rule can be eliminated, giving this grammar:

$$S \to SS$$

$$S \to (S_1$$

$$S_1 \to S)$$

$$S_1 \to)$$

1.1.3 Step 3

Now all rules have right-hand sides of length one or two. It is necessary to eliminate the rules whose right-hand side has length one.

This can be done by substituting as before; however, it may be necessary to do a chain of substitutions, if one has something like $X \to Y$ and $Y \to Z$ and X occurs on the right-hand side of some rule.

- In our grammar, we can substitute the rule $S_1 \to)$ into the rule $S \to (S_1$ obtaining the rule $S \to ()$.
- Then the rule $S_1 \to)$ can be eliminated. This yields the following grammar:

$$S \to SS$$

$$S \to (S_1$$

$$S_1 \to S)$$

$$S \to ()$$

This grammar is in Chomsky normal form, and we are done.

1.2 Time to Parse in Chomsky Normal Form

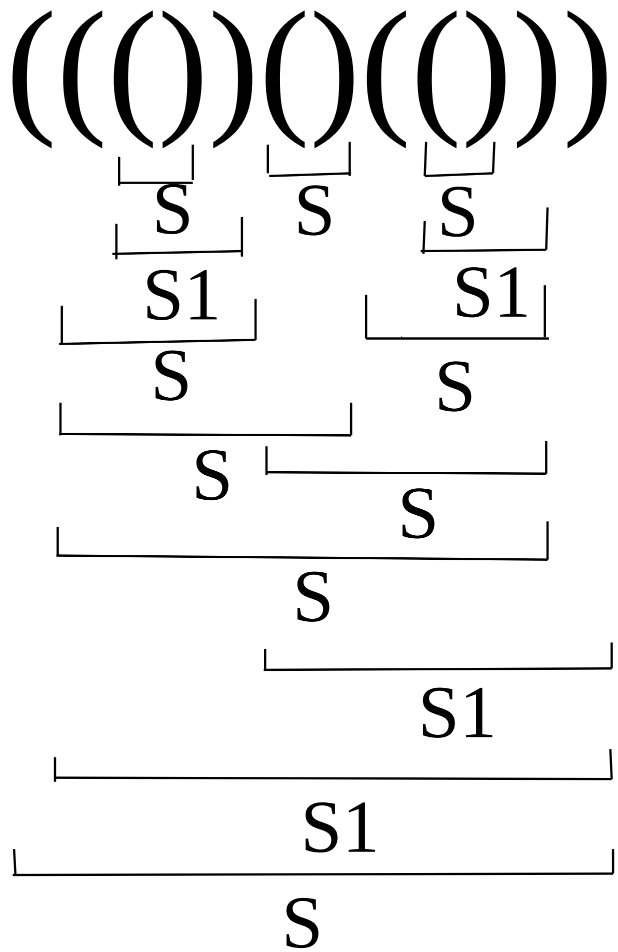
- Note that in a Chomsky normal form grammar, each replacement makes a string longer by one symbol.
- In our grammar, we have a derivation like this:

$$S \Rightarrow SS \Rightarrow (S_1 S \Rightarrow (S)S \ldots$$

  and note that each string is one symbol longer than the one before. So for example, a derivation of a string of length four has exactly three replacements in it.

- This gives a way to decide if a string w is in $L(G)$; just compute the length n of w and look at all derivations of length $n-1$ to see if w can be derived.
- However, this is very inefficient. It is possible to do much better. In fact, it can be done in $O(n^3)$ time.

This gives the idea of the method:



- The idea is that, to parse a string w, one considers all substrings v of w in order of size, and finds all nonterminals X such that $X \Rightarrow^* v$.
- Each such substring v has to be split up as $v_1 v_2$ in all possible ways,
- and for each way it is necessary to consider all X_1 and X_2 such that $X_1 \Rightarrow^* v_1$ and $X_2 \Rightarrow^* v_2$ and also all productions $X \to X_1 X_2$ in the grammar.
- The number of substrings of w is $O(n^2)$.
- For each substring there are $O(n)$ ways to split it up into $v_1 v_2$.

- For each way of splitting it there is a constant amount of work, so the total work is $O(n^3)$.

Here is an illustration how the method works on a substring in general: