An Introduction To Parsing

Wednesday, November 14, 2007
Reading: Stoughton 4.3

CS235 Languages and Automata

Department of Computer Science
Wellesley College

The Parsing Problem

Given a context-free grammar G and a string s:

- if $s \in$ Lang(G), return a parse tree in G that yields s;
- if $s \notin$ Lang(G), reject it.

Efficient solutions to this problem are important for analyzing both programming languages and natural languages.

The Parsing Problem: Examples

Example Grammar:

S → T | 0S1
T → % | 10T

Example Strings:

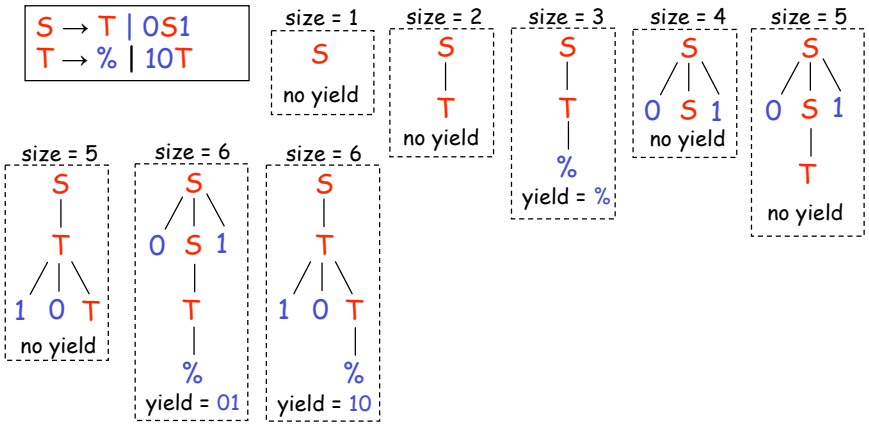
000111	011011
101010	001011
010101	011001

Goals For the Next Few Lectures

- Begin with some simple, but inefficient, parsing algorithms that work for all grammars.
- Study some more efficient approaches that work for various kinds of restricted grammars
- Build our way up to grammars suitable for a parser-generator program (ML-Yacc)

Naive Algorithm : Top-Down Generate and Test

Idea: generate all parse trees top down from the start variable in order of size (# of nodes or height) until find one that yields s.



How to know when to reject? Helps to have grammar in Chomsky Normal Form or Greibach Normal form. Why?

A Relevant Joke

An engineer, a physicist and a mathematician are staying in a hotel.

The engineer wakes up and smells smoke. He goes out into the hallway and sees a fire, so he fills a trash can from his room with water and douses the fire. He goes back to bed.

Later, the physicist wakes up and smells smoke. He opens his door and sees a fire in the hallway. He walks down the hall to a fire hose and after calculating the flame velocity, distance, water pressure, trajectory, etc. extinguishes the fire with the minimum amount of water and energy needed.

Later, the mathematician wakes up and smells smoke. He goes to the hall, sees the fire and then the fire hose. He thinks for a moment and then exclaims, "Ah, a solution exists!" and then goes back to bed.

(From http://www.math.utah.edu/~cherk/mathjokes.html)

Problems with Top-Down Generate and Test

Although it works for any grammar, top-down generate and test has big problems:

- Extremely inefficient approach to parsing.
- Repeat much of the same work for any string.
- Never take advantage of the structure of the string!

Working Bottom-Up: The Enzyme* Algorithm

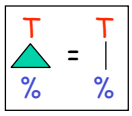
Idea: Build up all parse trees that can match substrings in the given string *s*. There are only a finite number of these. Accept if there is a parse tree for *s* rooted at the start variable. (This is the algorithm presented in Stoughton 4.3.)

grammar *G*

$S \rightarrow T \mid 0S1$
$T \rightarrow \% \mid 10T$

string *s* 001011

Step 1: Find all productions that yield % or some symbol in *s*.



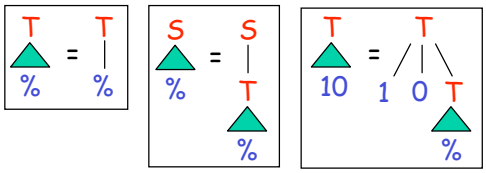
Step 2: Based on productions in *G*, add parse trees that yield some substring of *s*.

* "Enzyme" algorithm is my term. It is not standard.

Enzyme Example: The Next Round

grammar *G*: S → T | 0S1, T → % | 10T

string *s*: 001011

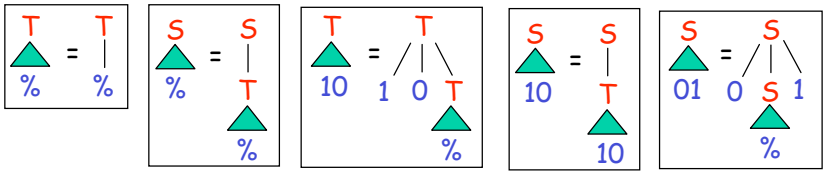


* "Enzyme" algorithm is my term. It is not standard.

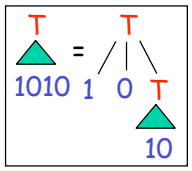
Enzyme Example Continued

grammar *G*: S → T | 0S1, T → % | 10T

string *s*: 001011

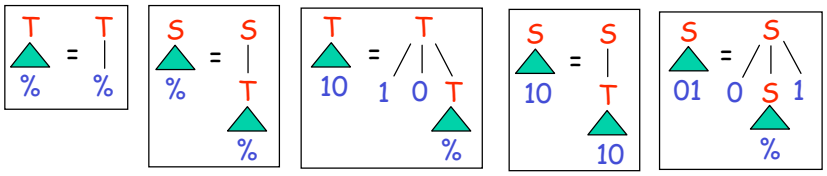


Note: we do not include this tree because 1010 isn't a substring of *s*.



Finishing The Enzyme Example

grammar G $\boxed{\begin{array}{l} S \rightarrow T \mid 0S1 \\ T \rightarrow \% \mid 10T \end{array}}$ string s $\boxed{001011}$



Flesh out the remaining trees in this example:

Problems with Enzyme Algorithm

Although it works for any grammar G and takes input string s into account, enzyme algorithm still can do much unnecssary work generating parse trees for substrings that can never appear in the final parse tree.

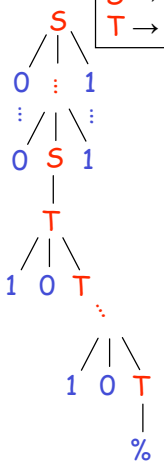
Towards A Deterministic Algorithm

Certain grammars (like our example) admit a deterministic parsing algorithm that depends on the structure of the grammar.

- Any initial sequence of 0s must be generated by $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0\ldots\ 0S1\ldots1 \Rightarrow 0\ldots\ 0A1\ldots1$
- If the first 1 is followed by a 0, it begins a sequence of 10s generated by $A \Rightarrow 01A \Rightarrow 0101A \Rightarrow 0\ldots01A \Rightarrow 01\ldots01\%$
- If the first 1 is followed by a 1, it begins a sequence of 11s generated by the end of $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0\ldots\ 0S1\ldots1 \Rightarrow 0\ldots\ 0A1\ldots1$ in which the number of 1s must match the number of 0s in the initial sequence.

Key ideas: the **first** symbols that can be generated by a variable, what symbols can **follow** the subtree generated by a variable, **lookahead** past the current symbol.

$S \rightarrow T \mid 0S1$
$T \rightarrow \% \mid 10T$



A Deterministic Algorithm

$S \rightarrow T \mid 0S1$
$T \rightarrow \% \mid 10T$

Here is a deterministic algorithm for our example:

- In the initial state (A), push bottom marker $ and enter state B.
- In state B consume each input 0 and push a corresponding 0 on the stack until there are no more input 0s.
  - If the end of input is reached, reject.
  - If there is an input 1 followed by a 0, enter state C (without consuming)
  - Otherwise, enter state D.
- In state C for every input 1 followed by a 0, consume both. Otherwise enter state D.
- In state D, for every input 1 with a corresponding 0 on the stack, consume the 1 and pop the 0.
  - If the end of input is reached with $ on the stack, accept.
  - Otherwise, reject

Can modify this algorithm to create the parse tree.