Notes on Context-Free Grammars
by Prakash Panangaden

Idea: One can generate sentences by giving rules for rewriting "templates" into actual sentences.

Some Definitions

\[ \Sigma \rightarrow \text{a set of symbols} \]

\[ \Sigma^* \rightarrow \text{all finite sequences of } \Sigma \text{ symbols} \]

e.g. \[ \Sigma = \{a, b\} \quad \Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\} \]

where \( \epsilon \) means empty sequence

A language over \( \Sigma \) is a subset of \( \Sigma^* \).

e.g. The language \( L \) of all sequences of "a" s and "b" s with an equal number of "a" s and "b" s is a language over \( \{a, b\} \).

For example, \( aab \) is not in \( L \) but \( bbaaba \) is.

To describe languages we use grammars.

Def. A grammar has a set \( \Sigma \) of symbols, a set \( NT \) of non-terminal symbols, a special symbol (usually \( S \)) in \( NT \) called the start symbol and a set of rules...
or productions of the form

\[ A \rightarrow \text{sequence from } (\varepsilon, \cup, \text{ N.T.}) \]

Discussion: We start with the start symbol and use rules to rewrite the sequences until we get a sequence of symbols from \( \varepsilon \). After this no more rewriting is possible, so we stop. For this reason, symbols in \( \Sigma \) are called terminals.

Example. Here is a grammar for the language \( L \) consisting of sequences of \( a \)'s & \( b \)'s with an equal number of \( a \)'s & \( b \)'s:

1. \( S \rightarrow \varepsilon \)
2. \( S \rightarrow SS \)
3. \( S \rightarrow bSa \)
4. \( S \rightarrow aSb \)

An example of generation using this grammar:

\[
\begin{align*}
S & \rightarrow bSa \\
& \rightarrow bbbSa a \\
& \rightarrow bbSbaaa \\
& \rightarrow bbaaabaa \\
& \rightarrow bbaaabaaa (\text{Rule 1 used twice})
\end{align*}
\]

I have written the rule number above each arrow and I have put a dashed underline below the part of the sequence that was just generated.

A grammar must generate all the sequences in the language and only the sequences in the language.

To save space we can put several rules with the same LHS on the same line:

\[ S \rightarrow \varepsilon \mid SS \mid bSa \mid aSb \]
The generation process can be better displayed with a tree.

Here is the grammar for parentheses: $\Sigma = \{ (, ) \}$

$S \rightarrow (S)S | \epsilon$

The tree shows grouping or structure.

The goal of parsing is to construct this tree given a sequence in the language.
Grammars for expressions

\[ NT = \{ \langle Exp \rangle \} \quad \Sigma = \{ \text{numbers, \(+, *,\)} \}
\]

\[ \langle Exp \rangle \rightarrow (\langle Exp \rangle) \mid \langle Exp \rangle + \langle Exp \rangle \mid \langle Exp \rangle * \langle Exp \rangle \mid \text{number} \]

This correctly generates the language but produces trees with the wrong grouping.

Parse tree for \[ 3 + 4 * 5 \]

Another parse tree for \[ 3 + 4 * 5 \]

A terrible situation! The same string has 2 parse trees and one of them has the wrong grouping. If we look at the corresponding expression trees we get

\[ \ast \]

\[ + \]

\[ 5 \]

\[ 3 \]

\[ 4 \]

WRONG!

\[ 3 \]

\[ \ast \]

\[ 5 \]

\[ 4 \]

\[ \ast \]

\[ 5 \]

RIGHT!
Recursive descent parsing

Here is how you recognize balanced parentheses:

1. Find a left parenthesis
2. Find a balanced string
3. Find a right parenthesis
4. Find a balanced string

This is suggested by the grammar production

\[ S \rightarrow (S)S \]

How can this work? It is plainly recursive, just use recursion.

Variables: first, second: boolean
nextsym: char

Function: check returns boolean.
Assume a global structure (e.g., a file) from which read is getting each character.

\[
\begin{align*}
&\text{if nextsym} = '(' \text{ then } \\
&\quad \text{read next char into nextsym;} \\
&\quad \text{first} = \text{check}(); \quad [\text{Recursion}] \\
&\quad \text{if (first and nextsym = ')')# } \\
&\qquad \text{read into nextsym;} \\
&\qquad \text{second} = \text{check}; \\
&\quad \text{if second return TRUE} \\
&\quad \text{else return FALSE;} \\
&\quad \text{else return FALSE}
\end{align*}
\]
Simple Expressions & Syntax Graphs
+ *, A, B, Z

\[
\begin{align*}
\langle \text{Exp} \rangle & \rightarrow \langle \text{Term} \rangle \{+ \langle \text{Term} \rangle \} \\
\langle \text{Term} \rangle & \rightarrow \langle \text{Primary} \rangle \{* \langle \text{Primary} \rangle \} \\
\langle \text{Primary} \rangle & \rightarrow A/B/\ldots/21 (\langle \text{Exp} \rangle)
\end{align*}
\]

This is the grammar for your assignment. Assume a procedure called GETSYM which takes the next symbol puts it in a global called "sym" and advances the cursor. The code is best organized using mutual recursion. I will discuss the control flow and not show how the tree is constructed.
procedure Exp;
begin
  Term;
  while sym \in \{ + \} do
    begin
      getsym; term
    end
  end
procedure Term;
begin
  primary;
  while sym \in \{ \star \} do
    begin
      getsym; primary
    end
end
procedure primary;
begin
  if sym in [A'...Z'] then getsym
  else if sym = '(' then
    begin
      getsym; expression;
      if sym = ']' then error else getsym
    end
  else error
end

In your code these should be functions that return trees. This code only says whether there is an error or not but it gives the basic control flow. PLEASE produce expression trees not parse trees.
Lexical analysis & parsing are so well-understood that we can generate parsers & scanners automatically. The symbol table is the data-structure where variable names and their bindings are stored. The parse-tree is usually simplified to what is called an abstract-syntax tree. For our assignment, I want you to generate expression trees directly.