NOTES ~~OF~~ ON CONTEXT-FREE GRAMMARS

by PRAKASH PANANGADEN

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

Some Definitions

$\Sigma \rightarrow$ a set of symbols

$\Sigma^* \rightarrow$ all finite sequences of Σ symbols

e.g. $\Sigma = \{a, b\}$ $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, \cdots\}$

where "ε" means empty sequence

A language over Σ is a subset of Σ^*.

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 bbabbaaa is.

To describe languages we use grammars.

Def A grammar has a set Σ 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 <u>productions</u> of the form

$$A \rightarrow \text{sequence from } (\Sigma \cup N.T.)$$

Discussion: We start with the start symbol and use <u>rules</u> to rewrite the sequences until we get a sequence of symbols from Σ. After this no more rewriting is possible, so we stop. For this reason, symbols in Σ are called <u>terminals</u>.

<u>Ex</u> 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 <u>generation</u> using this grammar

$S \xrightarrow{3} \underline{bSa} \xrightarrow{3} b\,\underline{bSa}\,a \xrightarrow{4} bb\underline{aSb}aa \xrightarrow{2}$
$bba\,\underline{SS}\,baa \xrightarrow{4} bba\underline{aSb}\,Sbaa \xrightarrow{3} bbaaSb\,\underline{bSa}\,baa$
$\xrightarrow{1,1} bbaabbabaa$ (rule 1 used twice)

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 <u>all</u> the sequences in the language and <u>only</u> the sequences in the language.

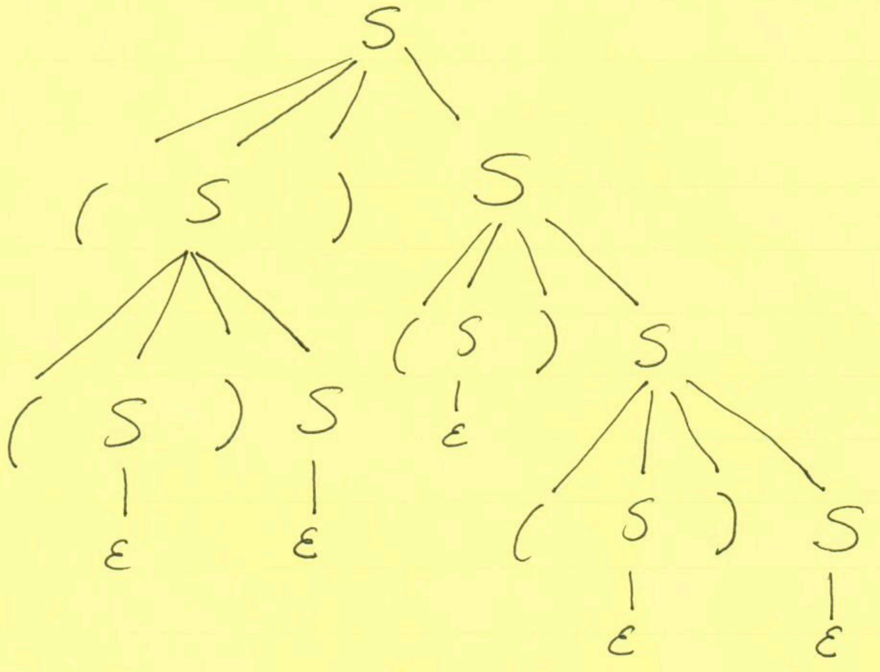
To save space we can put several rules with the same LHS on the same line:

$S \rightarrow \varepsilon \mid SS \mid bSa \mid aSb$

The generation process can be better displayed with a <u>tree</u>.

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

$$S \rightarrow (S)S \mid \varepsilon$$



The fringe is $(())()()$

The tree shows grouping or structure.

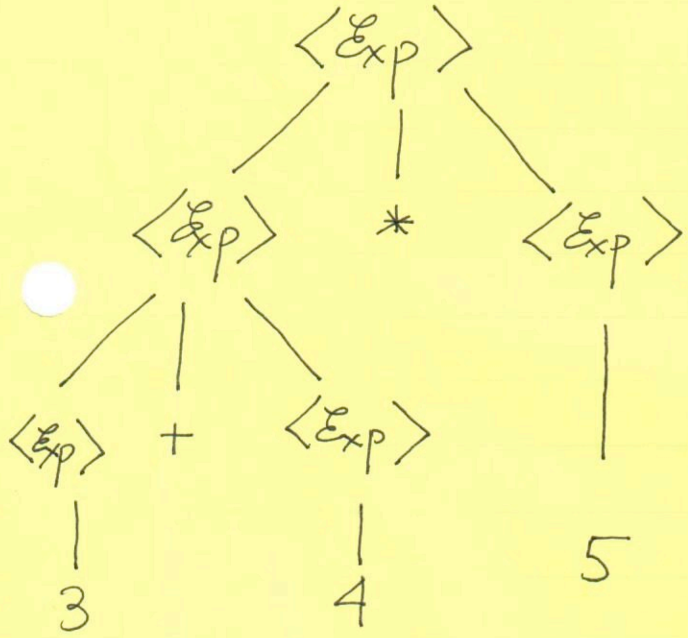
The goal of parsing is to construct this tree given a sequence in the language.

Grammars for expressions

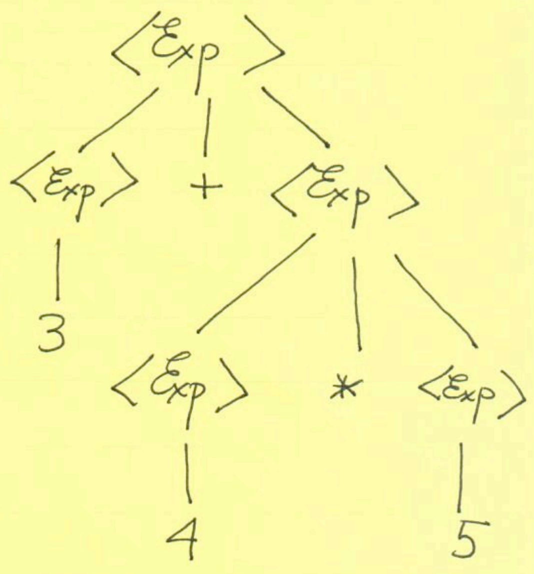
$$NT = \{ \langle Exp \rangle \} \qquad \Sigma = \{ \text{numbers}, +, *, (,) \}$$

$$\langle Exp \rangle \to (\langle Exp \rangle) \mid \langle Exp \rangle + \langle Exp \rangle \mid \langle Exp \rangle * \langle Exp \rangle \mid \text{number}$$

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

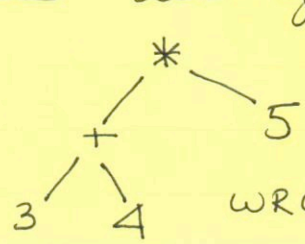


Parse tree for
$3 + 4 * 5$



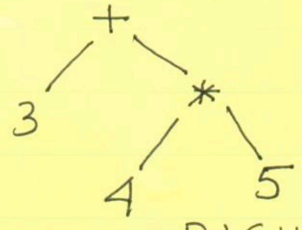
Another parse tree for
$3 + 4 * 5$

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



WRONG!

and



RIGHT!

When a grammar produces 2 or more parse trees for a sequence it is called <u>ambigiouus</u>. We need to design unambiguous grammars; but sometimes this is impossible. Here is how we do it for our expressions with generation of numbers thrown in as well.

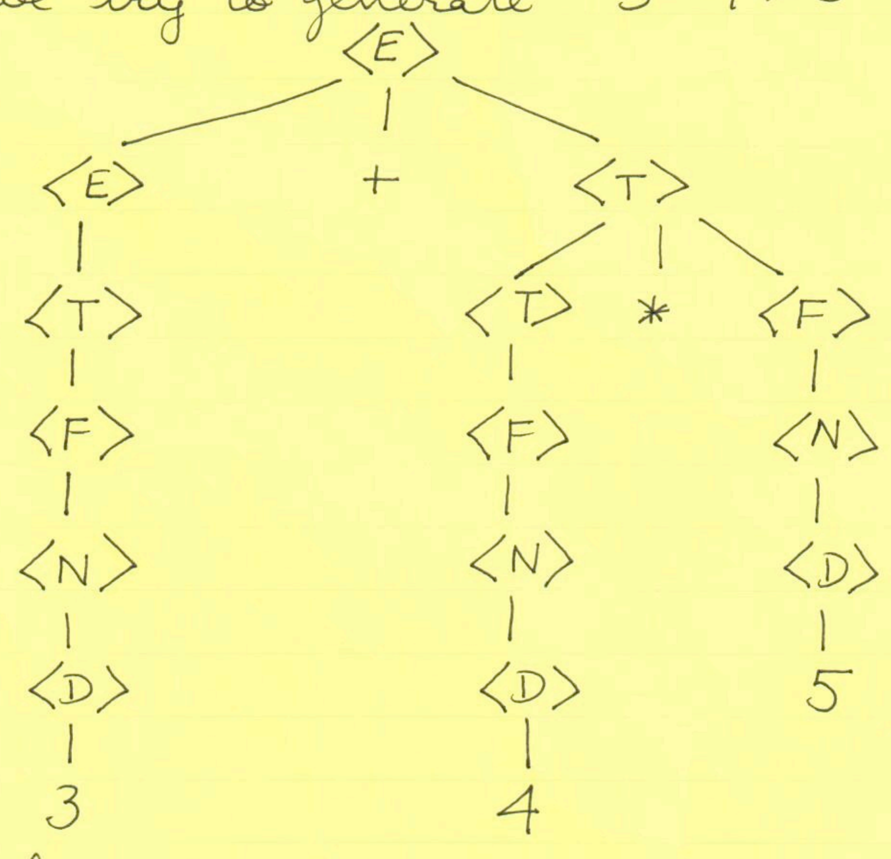
$NT = \{\langle N\rangle, \langle D\rangle, \langle E\rangle, \langle F\rangle, \langle T\rangle\}$

Start Symbol is $\langle E\rangle$

$\Sigma = \{+, *, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

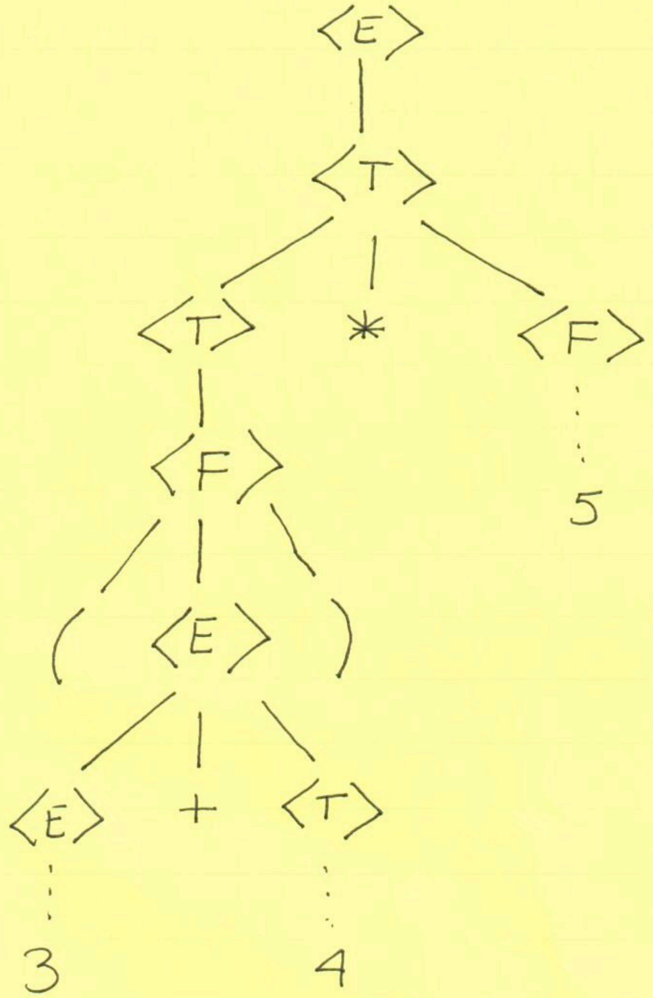
$$\langle E\rangle \longrightarrow \langle E\rangle + \langle T\rangle \mid \langle T\rangle$$
$$\langle T\rangle \longrightarrow \langle T\rangle * \langle F\rangle \mid \langle F\rangle$$
$$\langle F\rangle \longrightarrow (\langle E\rangle) \mid \langle N\rangle$$
$$\langle N\rangle \longrightarrow \langle N\rangle\langle D\rangle \mid \langle D\rangle$$
$$\langle D\rangle \longrightarrow 0|1|2|3|4|5|6|7|8|9$$

Now we try to generate $3+4*5$



This is the correct grouping.

Let us try to create the "wrong" grouping. We want to make the * appear at the top of the tree with the + nested inside it.



The ⋯ lines indicate the "obvious" steps that I have left out.

Notice how we were forced to put parens around 3 + 4 when we tried to do this.

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 (eg. a file) from which read is getting each character.

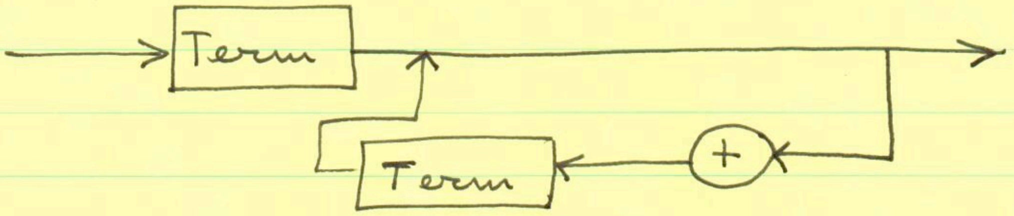
~~if~~ ~~if~~ nextsym = '(' then {
read next char into nextsym;
first = check(); [Recursion]
if (first and nextsym = ')') ~~then~~
{read into nextsym;
second = check;}
if second return TRUE
else return FALSE; }
~~else return FALSE~~
else return FALSE

Simple Expressions & Syntax Graphs

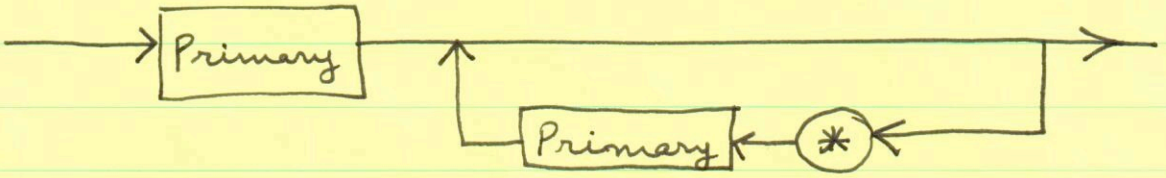
$+, *, A, B, \ldots, Z$

$$\langle Exp \rangle \rightarrow \langle Term \rangle \{ + \langle Term \rangle \}$$
$$\langle Term \rangle \rightarrow \langle Primary \rangle \{ * \langle Primary \rangle \}$$
$$\langle Primary \rangle \rightarrow A | B | \cdots | Z | (\langle Exp \rangle)$$

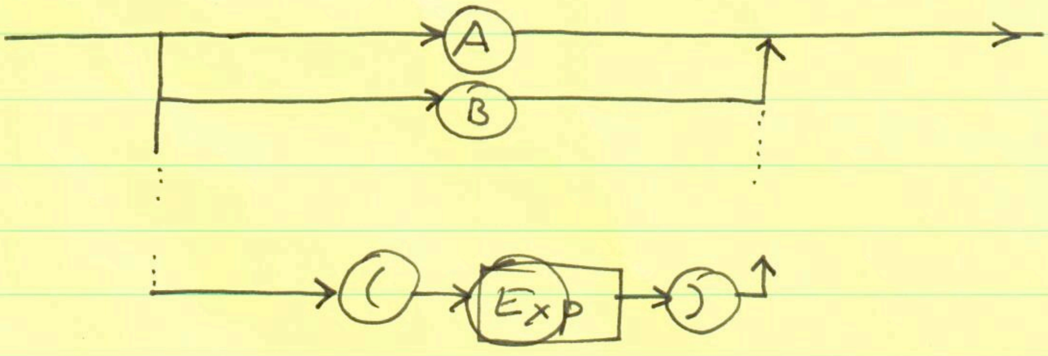
Exp =



Term =



Primary =



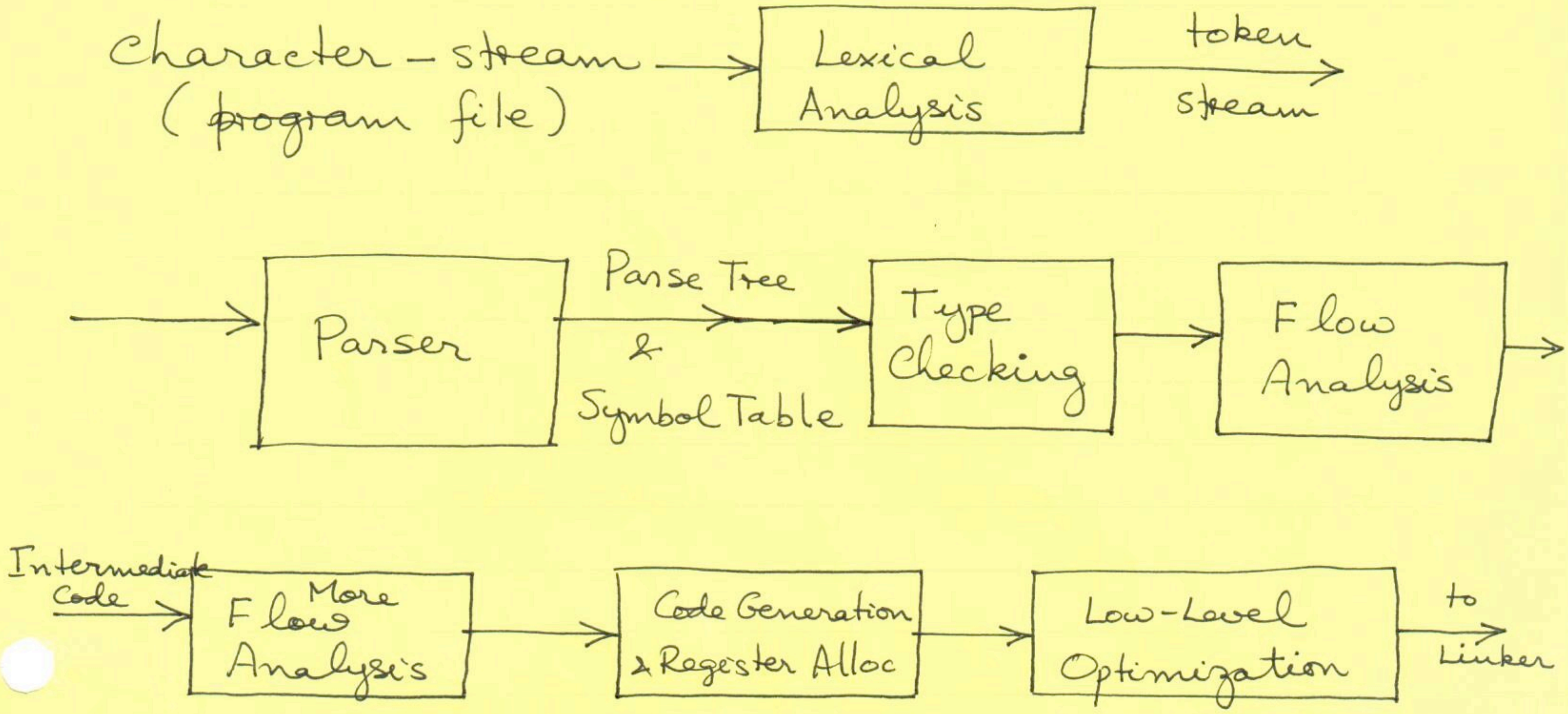
This is the grammar for your assignment. Assume a procedure called GETSYM which takes the next symbol puts it in a global called "sym" and advances the cursor. The code is best organized using mutual recursion. I will discuss the control flow and not show how the tree is constructed.

```
procedure Exp;
  begin
     Term;
      while sym = '+' do
           begin
               getsym; term
           end
  end
procedure Term;
  begin
     primary;
     while sym = '*' 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.

The structure of a compiler



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.