A Quick Introduction to Grammars

A *language* is a set of strings over some alphabet Σ. For a human language like English the alphabet is the set of words of the language and a string over this alphabet might be one valid sentence in the language. For a programming language the alphabet is the set of keywords, identifiers, and symbols of the language. A string of these is in the language if it represents a valid program.

A *grammar* for a language is a tool for saying which strings over the alphabet are in the language. Grammars are often used to help determine the meaning or semantics of valid strings in the language.

Grammars are very old -- Yaska and then Panini developed grammars for Sanskrit over 2500 years ago.

Grammars refer to two alphabets. The alphabet Σ for the language is often called the alphabet of *terminal symbols*. Grammars have a second alphabet Γ of *grammar symbols*, sometimes called *non-terminal symbols*. Grammar symbols for English include *NOUN*, *VERB-PHRASE* and so forth. With programming languages we usually write terminal symbols in lower-case and grammar symbols in upper-case.

The main part of a grammar consists of a list of grammar rules. Each rule has the form A ==> α, where A is a single grammar symbol and α is a string that might contain both grammar symbols and terminal symbols.

For example, one grammar rule for Scheme is

EXP ==> (if EXP EXP EXP)

That says that one form of a Scheme expression is a list (the parentheses are part of the terminal symbols) whose first element is the atom 'if and whose second third and fourth elements are themselves any Scheme expressions.

To make grammars a little more compact to write out, we often write all of the rules for a given grammar symbol on one line, with the different right-hand sides separated by a vertical bar |. So the grammar

E ==> E + T

E ==> T

T ==> T*F

T ==> F

F ==> number

might be more compactly written

E ==> E+T | T

T ==> T*F | F

F ==> number

A *derivation* with a grammar starts with one of the grammar symbols and at each step replaces one if its remaining grammar symbols with the right-hand side of one of the rules for that symbol. For example, with the grammar from the previous slide

E ==> E+T | T

T ==> T*F | F

F ==> number

we might have the following derivation:

Grammar: E ==> E+T | T

T ==> T*F | F

F ==> number

Derivation. At each step I underlined the grammar symbol being expanded for the next step

E ==> E+T

==> T+T

==> F+T

==> 3+T

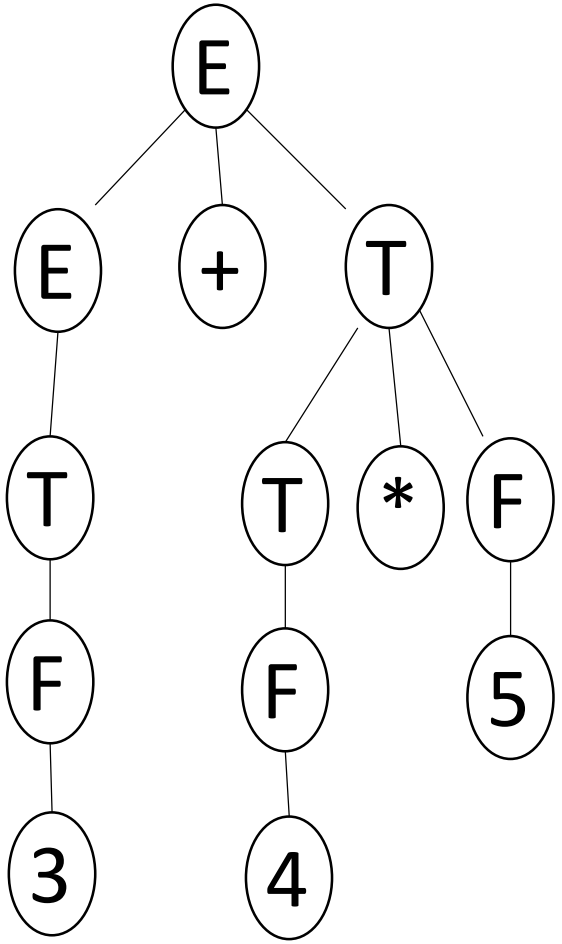
==> 3+T*F

==> 3+F*F

==> 3+4*F

==> 3+4*5

We can use a tree to represent such a derivation:



Such a tree is called a *parse tree*. Note that the expression it derives appears as a left-to-right traversal of the leaves of the parse tree.

One grammar symbol is usually designated as the *start symbol* and the language derived from the grammar consists of all strings of terminal symbols that can be derived from the start symbol in this way.

If no start symbol is specified we usually take it to be the left-hand side of the first grammar rule.

Here is a complete grammar for parenthesized arithmetic expressions over the operators + - * and /:

E ==> E+T | E-T | T

T ==> T*F | T/F | F

F ==> (E) | number

We need one final piece of notation. Sometimes we need to indicate that an element of a grammar can appear any number of times. The notation X^+ means that X can appear once, twice, thrice or any positive number of times. The notation X^* is similar, only X can appear 0 or more times (which means it could be absent entirely, or present any positive number of times).

Here is a full grammar for the portion of Scheme we will interpret:

```
EXP ==> number
        | symbol
        | (if EXP  EXP  EXP)
        | (let (LET-BINDINGS) EXP)
        | (lambda (PARAMS) EXP)
        | (set! symbol EXP)
        | (begin EXP*)
        | (letrec (LET-BINDINGS) EXP
        | (EXP  EXP*)
LET-BINDINGS ::= LET-BINDING*
LET-BINDING ::= (symbol EXP)
PARAMS ::= symbol*
```