Parse Trees Based on EBNF Specifications

We define parse tree with root *N* that generates $t_1 \ldots t_k$ in the same way as we defined such trees for a grammar, but:

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a BNF production* (whose right side consists only of terminals and/or nonterminals) *that can be obtained from one of the EBNF rules of the EBNF specification*.

  For example, the productions

  ```
  Expr ::= Term
  Expr ::= – Term + Term + Term – Term
  ```

  are two examples of productions that can be obtained from this EBNF rule: `Expr ::= [+ | –] Term {(+ | –) Term}`

As in the case of BNF, an EBNF specification has a ***starting nonterminal***.

Unless otherwise indicated, ***the starting nonterminal is the nonterminal on the left of the 1st EBNF rule***, and ***parse tree*** means ***parse tree whose root is the starting nonterminal***.

As in the case of BNF, the set (of sequences of terminals) denoted by the starting nonterminal of an EBNF specification is called the ***language generated by*** that EBNF specification.

- EBNF can be used just like BNF to define what it means for source code to be "***syntactically*** valid":

  For many programming languages L, the language designer can construct ~~a grammar~~ an EBNF specification *G* (whose terminals are L's tokens) such that:

    T_1 ... T_n belongs to the language generated by *G* if (and to a limited extent only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

  We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the ~~grammar~~ EBNF specification.

  - More generally, when a nonterminal of *G* corresponds to a language construct *X* (e.g., *statement*), we say a piece of source code is a ***syntactically valid*** *X* if its sequence of tokens belongs to the set denoted by the nonterminal.

For many programming languages L, the language designer can construct a grammar / EBNF specification *G* (whose terminals are L's tokens) such that:

> T_1 ... T_n belongs to the language generated by *G*
> if (and to a limited extent only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar / EBNF specification *G*.

- More generally, when a nonterminal of *G* corresponds to a language construct *X* (e.g., *statement*), we say a piece of source code is a ***syntactically valid*** *X* if its sequence of tokens belongs to the set denoted by the nonterminal.
- Any sequence of tokens that belongs to the set denoted by a nonterminal corresponding to a language construct *X* is said to be ***syntactically valid for*** *X*. (So ***a piece of source code is a syntactically valid*** *X* ***if and only if its sequence of tokens is syntactically valid for the language construct*** *X*.)

- The grammar or EBNF specification *G* we use is expected to satisfy the following condition for every nonterminal that corresponds to a language construct *X*:

  A sequence of tokens $T_1 \ldots T_n$ belongs to the set denoted by the nonterminal if (and to a limited extent only if) $T_1 \ldots T_n$ is the sequence of tokens of a possibly legal *X*.

  This implies:

  A piece of source code is a ***syntactically valid*** *X* if (and to a limited extent only if) its sequence of tokens is the sequence of tokens of a possibly legal *X*.

Example Suppose a nonterminal <statement> in an EBNF spec. of Java corresponds to a Java statement. Then

IDENTIFIER [IDENTIFIER] = IDENTIFIER / UNSIGINED-INT-LITERAL ;

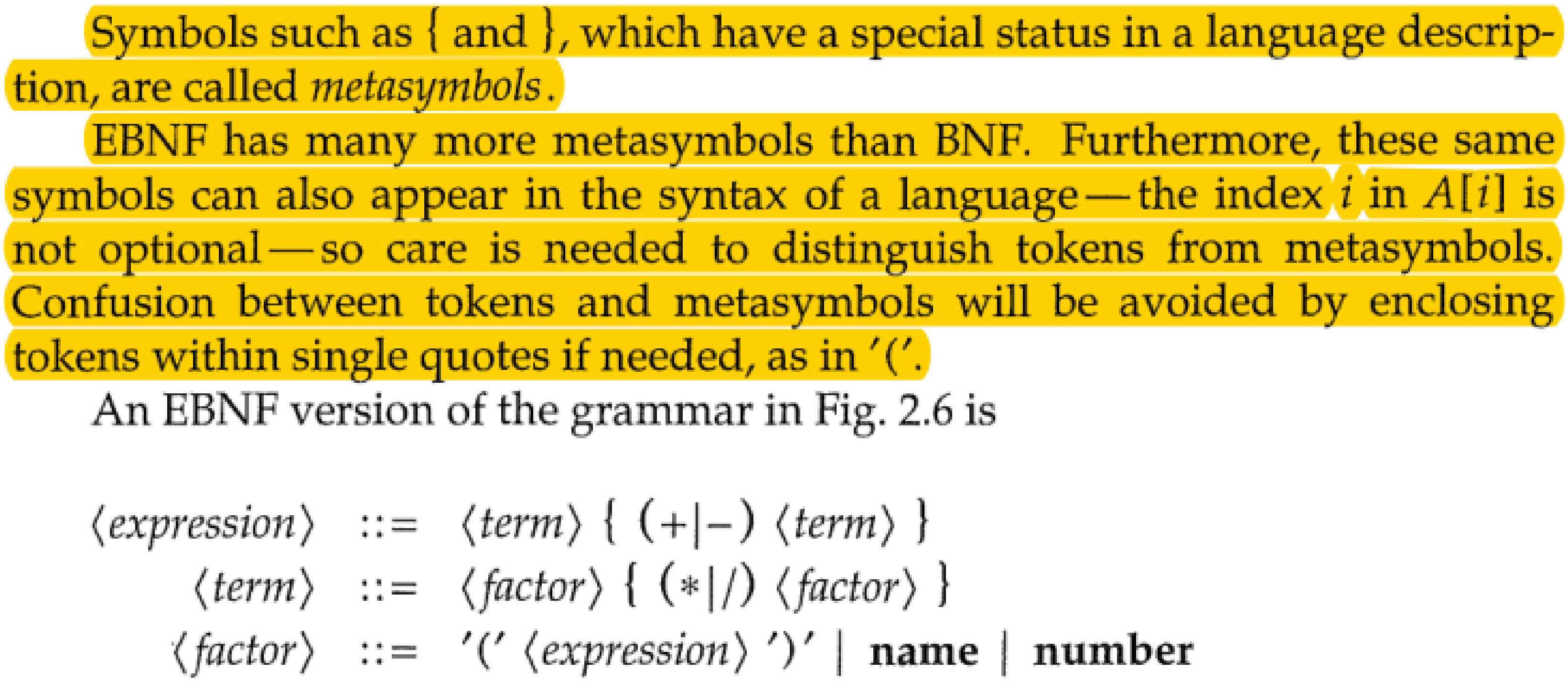
is a sequence of tokens that must belong to <statement>, because it is the sequence of tokens of the following possibly legal Java statement: `a[x] = b/3;`

`c[c] = d/0;` is ***not*** a possibly legal Java statement, but it is a ***syntactically valid*** Java statement because its sequence of tokens is the same as that of `a[x] = b/3;`.

Use ' ' to Distinguish Terminals from EBNF Metasymbols

In EBNF, when any of the characters | ( ) [ ] { } is a terminal, that terminal should be ***put in single quotes*** to make it clear that the character is ***not*** being used with its EBNF meaning!

Sethi says the following about this on p. 47 of his book (p. 48 of the course reader):



Symbols such as { and }, which have a special status in a language description, are called *metasymbols*.

EBNF has many more metasymbols than BNF. Furthermore, these same symbols can also appear in the syntax of a language—the index *i* in *A*[*i*] is not optional—so care is needed to distinguish tokens from metasymbols. Confusion between tokens and metasymbols will be avoided by enclosing tokens within single quotes if needed, as in '('.

An EBNF version of the grammar in Fig. 2.6 is

⟨*expression*⟩ ::= ⟨*term*⟩ { (+|−) ⟨*term*⟩ }
⟨*term*⟩ ::= ⟨*factor*⟩ { (∗|/) ⟨*factor*⟩ }
⟨*factor*⟩ ::= '(' ⟨*expression*⟩ ')' | **name** | **number**

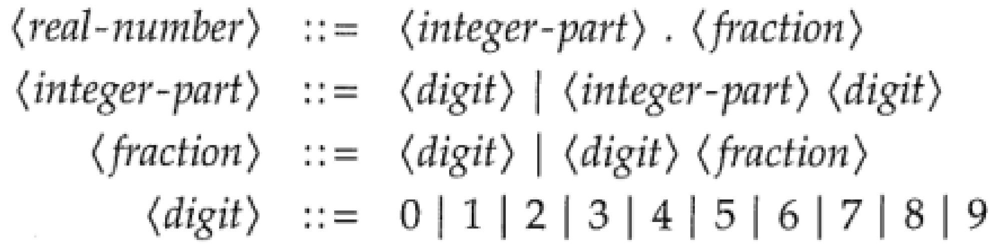
Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*.

In fact we have already seen an example (from the course reader) of how BNF can be used to specify such a token:



$$\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \,.\, \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Figure 2.3 BNF rules for real numbers.

This grammar specifies the ***unsigned floating point literal*** token of a very simple language.