Tokens are Not Always Terminals

As indicated above, it is common to use BNF or EBNF specifications of programming language syntax in which the terminals are the tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why some authors (including Sethi) may sometimes use the word “terminal” to mean “token”, and vice versa.

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*: The commonest examples of such tokens are IDENTIFIER and literal tokens whose instances are literal constants.

Tokens are Not Always Terminals

As indicated above, it is common to use BNF or EBNF specifications of programming language syntax in which the terminals are the 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:

```
⟨real-number⟩  ::=  ⟨integer-part⟩ . ⟨fraction⟩
⟨integer-part⟩ ::=  ⟨digit⟩ | ⟨integer-part⟩ ⟨digit⟩
    ⟨fraction⟩ ::=  ⟨digit⟩ | ⟨digit⟩ ⟨fraction⟩
       ⟨digit⟩ ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 2.3 BNF rules for real numbers.

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

Translating an EBNF Specification into a BNF Grammar

A grammar can only have ***finitely*** many productions. But when we "expand out" an EBNF rule that contains { ... } into a set of BNF productions, we get an ***infinite*** set of BNF productions!

However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... } as follows.

- Replace each (x_1 | ... | x_k) with a new nonterminal (*D*, say) defined by these *k* productions: *D* ::= x_1 | ... | x_k
- Replace each [x_1 | ... | x_k] with a new nonterminal (*D*, say) defined by: *D* ::= *<empty>* | x_1 | ... | x_k
- Replace each {x_1 | ... | x_k} with a new nonterminal (*D*, say) defined by one of the following two sets of *k*+1 productions:
  Either: *D* ::= *<empty>* | Dx_1 | ... | Dx_k
  Or: *D* ::= *<empty>* | x_1D | ... | x_kD

Here *k* may be 1. For example, { *Digit* } can be replaced with a new nonterminal *DigitSeq* defined in one of these two ways:

Either: *DigitSeq* ::= *<empty>* | *DigitSeq Digit*
Or: *DigitSeq* ::= *<empty>* | *Digit DigitSeq*

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}          (*)
```

into a finite set of BNF productions.

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes:

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= *<empty>* | Rest Op Term
   (**) becomes:

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest   (***)

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= *<empty>* | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest   (***)
3. Finally, replace [+ | -] with a nonterminal OptSign
   defined by   OptSign ::= *<empty>* | + | -
   (***) becomes:

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}         (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= *<empty>* | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest   (***)
3. Finally, replace [+ | -] with a nonterminal OptSign
   defined by   OptSign ::= *<empty>* | + | -
   (***) becomes: Expr ::= OptSign Term Rest

Example: We now use the above method to translate

```
     Expr ::= [+ | -] Term {(+ | -) Term}        (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest   (***)
3. Finally, replace [+ | -] with a nonterminal OptSign
   defined by   OptSign ::= <*empty*> | + | -
   (***) becomes: Expr ::= OptSign Term Rest

The result is the following set of 8 BNF productions:

```
     Expr ::= OptSign Term Rest
  OptSign ::= <empty> | + | -
     Rest ::= <empty> | Rest Op Term
       Op ::= + | -
```

While the above method always works, it will often ***not*** find a simplest finite set of grammar productions that is equivalent to the given EBNF rule!

For example, here is a simpler set of grammar productions that is equivalent to the EBNF rule

```
      Expr ::= [+ | -] Term {(+ | -) Term}
```

considered above:

```
  Expr ::=   Term
          |  + Term
          |  - Term
          |  Expr + Term
          |  Expr - Term
```