Ian Neville

BNF grammar

```
N = {<term>, <doll>, <sha>, <expr>, <ned>, <factor>}
T = {$, *, @, #}
P = {
         <expr> ::= <term><doll>
         <doll> ::= $<term><doll>|ε
         <term> ::= <factor><sha>
         <sha> ::= #<factor><sha>|ε
         <factor> ::= <ned>|*<ned>|@<ned>
         <ned> ::= X|<expr>}
S = <expr>
```

Associativity

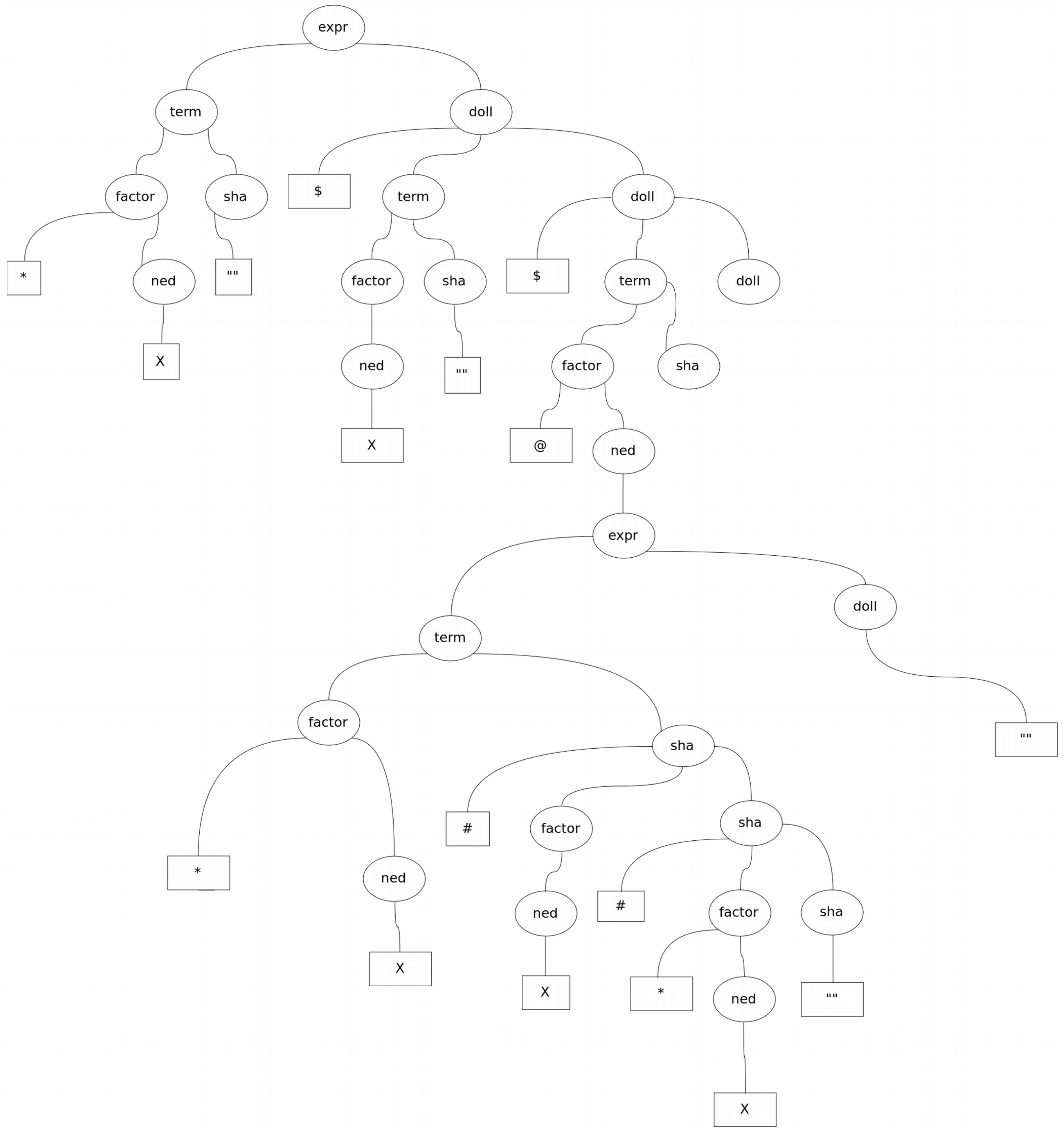
both $ and # are left and right associative.

Precedence table

@   *
#
$

Ian Neville

`*X$X$@*X#X#*X` **Parse Tree**



Ian Neville

Ambiguity proof of BNF

```
<expr>
<term><doll>
<term>$<term><doll>
<factor><sha>$<term><doll>
*<ned><sha>$<term><doll>
*X<sha>$<term><doll>
*X$<factor><sha><doll>
*X$<ned><sha><doll>
*X$X<sha><doll>
*X$X$<term><doll>
*X$X$<factor><sha><doll>
*X$X$@<ned><sha><doll>
*X$X$@<expr><sha><doll>
*X$X$@<term><doll><sha><doll>
*X$X$@<factor><sha><doll><sha><doll>
*X$X$@*<ned><sha><doll><sha><doll>
*X$X$@*X<sha><doll><sha><doll>
*X$X$@*X#<factor><sha><doll><sha><doll>
*X$X$@*X#<ned><sha><doll><sha><doll>
*X$X$@*X#X<sha><doll><sha><doll>
*X$X$@*X#X#<factor><sha><doll><sha><doll>
*X$X$@*X#X#*<ned><sha><doll><sha><doll>
*X$X$@*X#X#*X<sha><doll><sha><doll>
*X$X$@*X#X#*X<doll><sha><doll>
*X$X$@*X#X#*X<sha><doll>
*X$X$@*X#X#*X<doll>
*X$X$@*X#X#*X
```

and (next page)

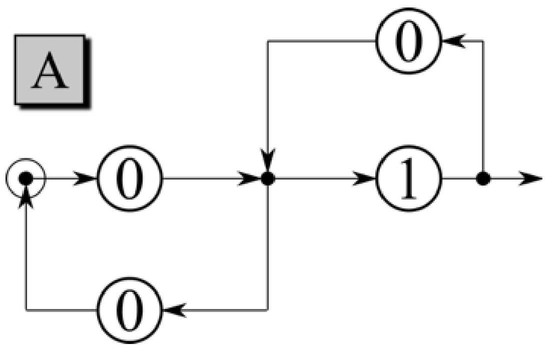
```
<expr>
<term><doll>
<factor><sha><doll>
*<ned><sha><doll>
*X<sha><doll>
*X<doll>
*X$<term><doll>
*X$<factor><sha><doll>
*X$<ned><sha><doll>
```

Ian Neville

```
*X$X<sha><doll>
*X$X$<term><doll>
*X$X$<factor><sha><doll>
*X$X$@<ned><sha><doll>
*X$X$@<expr><sha><doll>
*X$X$@<term><doll><sha><doll>
*X$X$@<factor><sha><doll><sha><doll>
*X$X$@*<ned><sha><doll><sha><doll>
*X$X$@*X<sha><doll><sha><doll>
*X$X$@*X#<factor><sha><doll><sha><doll>
*X$X$@*X#<ned><sha><doll><sha><doll>
*X$X$@*X#X<sha><doll><sha><doll>
*X$X$@*X#X#<factor><sha><doll><sha><doll>
*X$X$@*X#X#*<ned><sha><doll><sha><doll>
*X$X$@*X#X#*X<sha><doll><sha><doll>
*X$X$@*X#X#*X<doll><sha><doll>
*X$X$@*X#X#*X<sha><doll>
*X$X$@*X#X#*X<doll>
*X$X$@*X#X#*X
```

are different derivations for the same string, thus the BNF is ambigious.

Ian Neville



EBNF for Syntax chart A

`A::= 0 (0 A | 1 0 0 A | {1 0}) 1`

Justification

A can be interpreted as all combinations of the following recursive loops:
(1)

Start → 0 → Branch₁ → 0 → Start

Production
Path₁ ::= 0 0

In English: follow the arrows to the first 0, then to Branch₁, follow the downward arrow to the second 0, then to Start.

(2)

Start → 0 → Branch₁ → 1 → Branch₂ → 0 → Branch₁ → 0 → Start

Production
Path₂ ::= 0 1 0 0

in English: follow the arrows to the first 0, then to Branch₁, follow the forward arrow to the 1, then to Branch₂, follow the up arrow towards the 0, then to Branch₁, follow the downward arrow to the 0, then to the Start.

(3)

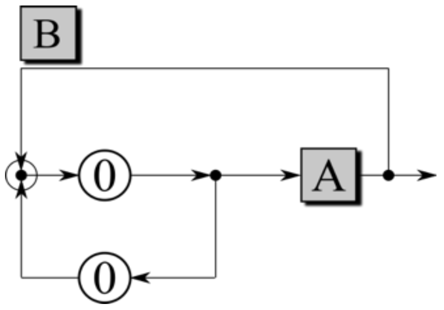
Branch₁ → 1 → Branch₂ → 0 → Branch₁

Production
InnerPath ::= {1 0} 1 End

In English: following any combination of the above recursive paths, whenever you reach Branch₁ you can loop through the 1 0 path, 0 or more times.

With tracing the combinations of these paths, and understanding the options given at each branch, I used the tool of EBNF to come up with the production of A.

Ian Neville



EBNF for Syntax Chart B

`B::= 0 ({A B} | 0 B) A`

Justification

B can be interpreted as all combinations of the following recursive loops:

(1)

Start → 0 → Branch → 0 → Start

Production

$Path_1$ ::= 0 0

This is identical to the first path in A's chart: Following the arrows back to the beginning, encountering only 0s.

(2)

Start → 0 → Branch → A → Start

Production

$Path_2$ ::= 0 A

This path actually contains the syntax of A, so the paths inside A will be defined using A's EBNF

Using these paths you can traverse every combination of the syntax, as long as you end with A, with the power of EBNF this can be represented as

`B::= 0 ({A B} | 0 B) A`

Ian Neville

Ambiguity proof for BNF

```
N = {<S>, <A>, <I>}
T = {a, b, c, x}
P = {<S> ::= <A>
     <A> ::= <A>x<A>
     <A> ::= <I>
     <I> ::= a
     <I> ::= b
     <I> ::= c}
S = <S>
```

axaxa **derivations**

```
<S>
<A>
<A>x<A>
<A>x<I>
<A>xa
<A>x<A>xa
<I>x<A>xa
ax<A>xa
ax<I>xa
axaxa
```

and

```
<S>
<A>
<A>x<A>
<I>x<A>
ax<A>
ax<A>x<A>
ax<I>x<A>
axax<A>
axax<I>
axaxa
```

are different derivations for the same string, thus the BNF is ambigious.