Backus-Naur Form 3/bnf

Purpose

To weave any collated BNF grammar from the web into a nicely typeset form.

§**1. Parsing BNF.** See the manual for more on what these grammars are, and how they're notated in the web. They have no function in the compiled code, and are purely for documentation. It's a diversion rather than the main work of the weaver, but it may as well stay in `inweb`, though it won't be useful for many webs.

There are shameless Inform-specific hacks in this section, but they're not likely to do any harm.

When we read and parsed the web, we extracted the BNF grammar lines into an array. But they were otherwise raw, and we will have to manipulate them quite a bit to turn them into TEX.

First, a little bit of jargon. The tokens we represent in angle brackets are called "nonterminals", and the grammar lines which provide ways they can be made up are called "productions". Our grammar contains only lines in which the left hand side is a single nonterminal.

```
sub parse_bnf_grammar {
    ⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩;
    ⟨Initialise the nonterminals discovered 6⟩;
    ⟨Work out which nonterminals depend on which other ones 7⟩;
    ⟨Elect certain nonterminals as being fundamental 8⟩;
    ⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩;
    ⟨Any nonterminals remaining are placed at the top level 10⟩;
}
```

§**2.** Our basic plan is to use the hash `$productions{$nt}` to accumulate all productions which have the nonterminal `$nt` as their right-hand-side; in other words, all ways to make `$nt`.

⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩ ≡

```
    my $i;
    my $l;
    my $nonterminal_being_defined;
    $nonterminal_being_defined = "";
    for ($i=0; $i<$bnf_grammar_lines; $i++) {
        $l = $bnf_grammar[$i];
        if ($l =~ m/^\s*\<(.*?)\>\s*(.*)$/) {
            $nonterminal_being_defined = $1; $l = $2;
            if (not(exists $productions{$nonterminal_being_defined}))
                ⟨Create new nonterminal 3⟩;
        }
        if ($l =~ m/^\s*$/) { next; }                                    skip blank line
        if ($l =~ m/^\s*\:\=\s+(.*)$/) ⟨A new production begins here 4⟩;
        if ($l =~ m/^\s*\.\.\.\s+(.*)$/) ⟨The previous production continues onto this line 5⟩;
    }
```

This code is used in §1.

§**3.** When we see a nonterminal with a name we haven't seen before, we make it a key in the `%productions` hash, and also set its TEX representation, including a footnote identifying where in the web it originates.

⟨Create new nonterminal 3⟩ ≡
```
$productions{$nonterminal_being_defined} = $i;
$production_tex{$nonterminal_being_defined}
    = "\\nonterminal{" . $nonterminal_being_defined . "}\$_{\\rm ".
        $section_sigil[$line_sec[$bnf_grammar_source[$i]]] . "}\$";
```
This code is used in §2.

§**4.** `$productions{$nt}` is a single string, but it holds a concatenated list of productions for `$nt`. We can divide this up into its entries again because each one ends in a sentinel ampersand, `&`. (In `inweb` BNF grammar, a literal ampersand would be written out: `AMPERSAND`. So this won't cause ambiguities.)

⟨A new production begins here 4⟩ ≡
```
my $rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
$no_productions_for{$nonterminal_being_defined}++;
$productions_for{$nonterminal_being_defined} .= $rhs."&";
next;
```
This code is used in §2.

§**5.** A grammar line beginning with an ellipsis marks (a) a continuation of the previous line, and also (b) a place to weave a line-break in the final TEX output – which is why we combine such a line with its predecessor, but leave the ellipsis token in.

⟨The previous production continues onto this line 5⟩ ≡
```
my $more_rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
my $existing_rhs = $productions_for{$nonterminal_being_defined};
$existing_rhs =~ s/\&$//;
$productions_for{$nonterminal_being_defined} = $existing_rhs." ... ".$more_rhs."&";
```
This code is used in §2.

§**6.** The scan is now complete, so we know all of the nonterminals. We do a little judicious respacing of the productions, to make square and curly braces tokens in their own rights (a convenience for later), and we strip out white space. We also initialise three more hashes:

(a) `$bnf_dependencies{$nt}` is a list of the other non-terminals which occur in productions leading to `$nt` – in effect, the things whose definitions we need to know in order to understand the definition of `$nt`.
(b) `$bnf_depth{$nt}` is used in weaving to print grammar in a hierarchical way. It will eventually be a positive integer, 1, 2, 3, ..., but will be 0 for nonterminals whose depth hasn't been determined. The idea is that if X depends on Y, and X is important enough to be worth giving a hierarchical display, then Y will have depth higher than X's.
(c) `$bnf_follows{$nt}` is either blank, or else the name of a nonterminal to which `$nt` "belongs" – meaning that it will be printed in the hierarchy underneath this owner. To a human reader, `$nt` follows this owner, but is indented further rightwards.

⟨Initialise the nonterminals discovered 6⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    $bnf_dependencies{$nt} = "";
    $bnf_depth{$nt} = 0;
    $bnf_follows{$nt} = "";
    $productions_for{$nt} =~ s/\[/ \[ /g;
    $productions_for{$nt} =~ s/\]/ \] /g;
    $productions_for{$nt} =~ s/\{/ \{ /g;
    $productions_for{$nt} =~ s/\}/ \} /g;
    $productions_for{$nt} =~ s/\s+/ /g;
}
```

This code is used in §1.

§**7.** This calculates the `$bnf_dependencies{$nt}` hash (see above).

⟨Work out which nonterminals depend on which other ones 7⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    my $lines = $productions_for{$nt};
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        my $line = $1; $lines = $2;
        while ($line =~ m/^\s*(\S+)(.*?)$/) {
            my $tok = $1;
            $line = $2;
            if ($tok =~ m/^\<(.*)\>$/) {
                $tok = $1;
                if (exists $production_tex{$tok}) {
                    $bnf_dependencies{$tok} .= $nt . ",";
                }
            }
        }
    }
}
```

This code is used in §1.

§**8.** Some nonterminals are worth making a fuss over, and printing with little hierarchical grammars of their own. (This helps us to give structure to what would otherwise be a shapeless and difficult to read mass of productions.) Such nonterminals are called "fundamental" by `inweb`. The following is, pretty shamelessly, a list of the fundamental nonterminals in Inform.

⟨Elect certain nonterminals as being fundamental 8⟩ ≡

```
foreach $nt (sort keys %productions) {
    if (($bnf_dependencies{$nt} eq "")
        || ($nt =~ m/\-sentence$/)
        || ($nt eq "and")
        || ($nt eq "or")
        || ($nt eq "and-or")
        || ($nt eq "paragraph")
        || ($nt eq "sentence")
        || ($nt eq "condition")
        || ($nt eq "action-pattern")
        || ($nt eq "literal-value")
        || ($nt eq "type-expression")
        || ($nt eq "physical-description")
        || ($nt eq "value")) {
        $bnf_depth{$nt} = 1;
        $bnf_follows{$nt} = "";
    }
}
```

This code is used in §1.

§**9.** So at this point fundamental nonterminals have depth 1, while all other nonterminals have depth 0. The following decides, not very efficiently, which of the un-fundamental nonterminals need to be given increased depth so as to place them below their fundamental owners in the hierarchy.

⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩ ≡

```
my $pass;
for ($pass = 1; $pass <= 2; $pass++) {
    my $changed = 1;
    while ($changed == 1) {
        $changed = 0;
        my $nt;
        foreach $nt (sort keys %productions) {
            if ($bnf_depth{$nt} == 0) {
                $scan = $bnf_dependencies{$nt};
                $min_depth = 10000000; $unfound = 0;
                while ($scan =~ m/^(.*?)\,(.*)$/) {
                    $scan = $2;
                    $used = $1;
                    if ($bnf_depth{$used} > 0) {
                        if ($min_depth > $bnf_depth{$used}) {
                            $min_depth = $bnf_depth{$used};
                            $min_depth_follows = $used;
                        }
                    } else { if ($pass == 1) { $unfound = 1; } }
                }
                if (($unfound == 0) && ($min_depth < 100000)) {
                    $bnf_depth{$nt} = $min_depth+1;
```

```
                    $bnf_follows{$nt} = $min_depth_follows;
                    $changed = 1;
                }
            }
        }
    }
}
```

This code is used in §1.

§**10.** Perhaps there were no fundamental nonterminals in the first place; or perhaps there were a few, but that they didn't cover the entire language. Either way, there could be nonterminals still having depth 0. We'll promote them to depth 1, so that they have a sensible margin setting when we come to weave.

⟨Any nonterminals remaining are placed at the top level 10⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    if ($bnf_depth{$nt} == 0) {
        $bnf_depth{$nt} = 1;
    }
}
```

This code is used in §1.

§**11. Sequencing the output.** A large grammar with many nonterminals can be very difficult to read if listed in a jumbled-up order, so the following provides the ability to control how the thing comes out: we can output certain nonterminals of our choice first, and then ultimately follow up with the rest.

```
sub weave_bnf_grammar {
    my $toshow = $_[0];
    if ($bnf_already_parsed == 0) { parse_bnf_grammar(); $bnf_already_parsed = 1; }
    ⟨Handle a comma-separated list of commands 12⟩;
    ⟨Handle a command to show nonterminals including given text 13⟩;
    ⟨Handle a command to show a specific named nonterminal 14⟩;
    ⟨Handle a command to show all remaining nonterminals 15⟩;
    ⟨Finish up and check for errors 16⟩;
}
```

§**12.** A comma-separated list of commands is acted on from left to right:

⟨Handle a comma-separated list of commands 12⟩ ≡

```
if ($toshow =~ m/^\: (.*?)\,\s*(.*)$/) {
    my $a = $1;
    my $b = $2;
    weave_bnf_grammar(": ".$a);
    weave_bnf_grammar(": ".$b);
    return;
}
```

This code is used in §11.

§13. A dash indicates a partial match with the production name is enough. (For instance, to pick up every nonterminal with a name ending "-constant".)

⟨Handle a command to show nonterminals including given text 13⟩ ≡
```
if ($toshow =~ m/^\:\s*\-(.*)$/) {
    $tomatch = $1;
    foreach $nt (sort keys %productions) {
        if ($nt =~ m/\-$tomatch$/) {
            weave_nonterminal($nt);
        }
    }
    return;
}
```

This code is used in §11.

§14.

⟨Handle a command to show a specific named nonterminal 14⟩ ≡
```
if ($toshow =~ m/^\:\s*(.*)$/) {
    weave_nonterminal($1);
    return;
}
```

This code is used in §11.

§15.

⟨Handle a command to show all remaining nonterminals 15⟩ ≡
```
weave_nonterminal("value");
my $nt;
foreach $nt (sort keys %productions) {
    if ($bnf_depth{$nt} == 1) {
        weave_nonterminal($nt);
    }
}
```

This code is used in §11.

§16.

⟨Finish up and check for errors 16⟩ ≡
```
foreach $nt (sort keys %productions) {
    if ($production_done{$nt} == 0) {
        print "Error in weave of BNF grammar: omitted ", $production_tex{$nt}, "\n";
        print $nt, ": ", $bnf_depth{$nt}, " ", $bnf_follows{$nt}, " ",
            $bnf_dependencies{$nt}, "\n";
    }
}
foreach $nt (sort keys %unknown_productions) {
    print WEAVEOUT "\\medskip{\\bf Unknown production:} \\nonterminal{", $nt, "}\\par\n";
}
```

This code is used in §11.

§**17. Weaving a single production.** In other words, writing out the necessary TEX to display the grammar matching a single concept. Following each production are the ones used first in the course of defining it, and so on.

```
sub weave_nonterminal {
    my $nt = $_[0];
    if ($production_done{$nt} == 1) { return; }
    if ($bnf_depth{$nt} == 1) {
        print WEAVEOUT "\\smallbreak\\hrule\\smallbreak\n";
    } else {
        print WEAVEOUT "\\smallbreak\n";
    }
    ⟨Weave the actual nonterminal 19⟩;
    ⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩;
    $production_done{$nt} = 1;
}
```

§**18.**

⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩ ≡

```
    my $dep;
    foreach $dep (sort keys %productions) {
        if ($bnf_follows{$dep} eq $nt) {
            weave_nonterminal($dep);
        }
    }
```

This code is used in §17.

§**19.** There are various hacks here which are entirely to make the Inform grammar look prettier; I do not much repent.

⟨Weave the actual nonterminal 19⟩ ≡

```
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT $production_tex{$nt}, "\n";
    $lines = $productions_for{$nt};
    $joined_lines = 0;
    if ($nt eq "debugging-aspect") { $joined_lines = 1; $join_count = 3; }
    if ($nt eq "determiner") { $joined_lines = 1; $join_count = 2; }
    $lc = 0; $lcm = $join_count - 1;
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        $lc++; $line = $1; $lines = $2;
        ⟨Get to the correct margin position for the production 20⟩;
        ⟨Weave out the tokens of the production 21⟩;
        print WEAVEOUT "\n";
    }
```

This code is used in §17.

§**20.**

⟨Get to the correct margin position for the production 20⟩ ≡

```
if ($joined_lines == 1) {
    $lcm++; if ($lcm == $join_count) { $lcm = 0; }
    if ($lc == 1) {
        print WEAVEOUT "\\par";
        bnf_indent($bnf_depth{$nt} - 1);
        print WEAVEOUT "\\qquad ::\$=\$ ";
    } else {
        if ($lcm == 0) {
            print WEAVEOUT "\\par";
            bnf_indent($bnf_depth{$nt} - 1);
            print WEAVEOUT "\\qquad \\phantom{::\$=\$} \\| ";
        } else {
            print WEAVEOUT "\\| ";
        }
    }
} else {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad ::\$=\$ ";
}
```

This code is used in §19.

§**21.** The term "token" is used loosely here, and not in its technical context free grammar sense. A production at this point is a sequence of textual globs divided by a single space, such as this:

```
COMMA while <condition> [ SEMICOLON ]
```

The tokens are simply these globs, so here there are six tokens. `<condition>` here is a nonterminal.

⟨Weave out the tokens of the production 21⟩ ≡

```
$last_tok = "";
while ($line =~ m/^\s*(\S+)(.*?)$/) {
    $tok = $1;
    $line = $2;
    ⟨Insert inter-token space where appropriate 22⟩;
    $last_tok = $tok;
    ⟨Weave an ellipsis token 23⟩;
    ⟨Weave a nonterminal token 24⟩;
    ⟨Weave a literal text token 25⟩;
}
```

This code is used in §19.

§**22.**

⟨Insert inter-token space where appropriate 22⟩ ≡

```
if (($last_tok ne "[") && ($last_tok ne "{") && ($last_tok ne "")
    && ($tok ne "]") && ($tok ne "}")) {
    print WEAVEOUT " ";
}
```

This code is used in §21.

§**23.** An ellipsis is a continuation onto the next line:

⟨Weave an ellipsis token 23⟩ ≡
```
if ($tok eq "...") {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad \\phantom{::\$=\$} ";
    next;
}
```

This code is used in §21.

§**24.**

⟨Weave a nonterminal token 24⟩ ≡
```
if ($tok =~ m/^\<(.*)\>$/) {
    $tok = $1;
    if (exists $production_tex{$tok}) {
        print WEAVEOUT $production_tex{$tok};
    } else {
        if ($tok =~ m/^(.*)-name$/) {
            print WEAVEOUT "{\\rm " . $1 . "}";
        } else {
            if ($tok =~ m/^(.*)-usage$/) {
                print WEAVEOUT "{\\it " . $tok . "}";
            } else {
                print WEAVEOUT "\\nonterminal{\\rm !!!!" . $tok . "!!!!}";
                $unknown_productions{$tok}++;
            }
        }
    }
    next;
}
```

This code is used in §21.

§**25.** And otherwise we have literal text in boldface, though the fact that curly braces have to be rendered in math mode complicates things a little.

⟨Weave a literal text token 25⟩ ≡
```
if ($tok eq "{") { $tok = "\$\\{\$"; }
if ($tok eq "}") { $tok = "\$\\}\$"; }
if ((length($tok) > 2) &&
    ($tok =~ m/^[A-Z\-][A-Z\-0-9]+$/)) {
    print WEAVEOUT "{\\eightbf ", $tok, "}";
    $lexical_productions{$tok}++;
} else {
    print WEAVEOUT "{\\bf ", $tok, "}";
}
```

This code is used in §21.

§**26.** Indentation, the crude way:

```
sub bnf_indent {
    my $dep = $_[0];
    my $d;
    for ($d=1; $d<=$dep; $d++) {
        print WEAVEOUT "\\quad";
    }
}
```