ABNF to PEG translation rules

Dirk Brand

January 20, 2014

This is a guide to convert an ABNF specification to PEG format in-place.

Style

- Replace all assignment operators ('=' in ABNF) with ←.
- Comments start with ';'(semi-colon) in ABNF. Replace with '#' in PEG. Comments go to the end of the line.
- Use CamelCase when writing non-terminals (ex. 'dot-atom' becomes 'DotAtom').
- If the non-terminal represents a single symbol (optionally followed by a space), use all capital letters to write it (ex. `AND <- '&'`).

Rules

- Rules are concatenated by placing the rules in a sequence. No translation needed.
- Alternate rules are indicated with '/' in ABNF and alternate rules can be added by using the '=/' operator. Incremental alternatives are not used in PEG. Combine all incremental alternatives into one line of alternatives.
  Example:
  rule = alt1 / alt2
  rule =/ alt3
  becomes
  rule ← alt1 / alt2 / alt3.

Terminal Values

- Numeric terminal values in ABNF are specified with `\%` followed by either 'b', 'd' or 'x' (for binary, decimal and hexadecimal respectively) and then followed by the value. Values can be concatenated with the '.' symbol, or a range of values can be shown by placing '-' between subsequent values (ex. `\%d33-90`). Translate all numeric terminal values to the equivalent unicode values in hexadecimal (ex. `\%d35-91` becomes '`\u0023`'-'`\u005b`').
- Value ranges can optionally be translated to a character class (if applicable). Thus, the octal range is represented by `\%x30-37` in ABNF, but can be expressed as `[0-7]` in PEG.
- String literals in ABNF are enclosed in double quotes and are case-insensitive. Replace with either single or double quotes in PEG.

Regular Expressions

- Translate the repetition rule `1*element` in ABNF to `element+` in PEG.
- Translate the repetition rule `*element` in ABNF to `element*` in PEG.
- Translate the optional rule `[element]` in ABNF to `element?` in PEG. Other ABNF equivalent statements are `*1element` and `0*1element`.

Algorithm

The input file is an ABNF formal language specification. It can be defined as a 4-tuple $A = (S_N, S_T, R_A, e_{S_1})$ where S_N is the finite set of non-terminal symbols, S_T the finite set of terminal symbols, R_A the finite set of derivation rules and $e_{s_1} \in S_N$ the starting symbol. The rule-set R_A consists of rules of the form (A, e) written as $A = e$ where $A \in S_N$ and e is an expression.

The output file is a *Parsing Expression Grammar* (PEG) formally defined as a 4-tuple $G = (V_N, V_T, R_G, e_{S_2})$. V_N is the finite set of non-terminal symbols, V_T the finite set of terminal symbols, R_G the finite set of production rules and $e_{s_2} \in V_N$ the starting symbol. The rule-set R_G consists of rules of the form (A, e) written as $A \leftarrow e$ where $A \in V_N$ and e is an expression.

Numeric terminals in ABNF are defined as a terminal character of the form `%da`, `%ba` or `%xa` where `a` is a numeric value.

Constraints

$$S_N \cap S_T = \emptyset$$
$$V_N \cap V_T = \emptyset$$

Algorithm 1: ABNF to PEG translation algorithm.

input : ABNF formal language specification
output: Parsing Expression Grammar

foreach $r \in R_A \mid r \rightarrow A = e$ **do**
    Replace assignment symbol with $\leftarrow$;
    if *e consists of a single string literal* **then**
        Replace A with equivalent non-terminal in all caps;
    else
        Replace A with equivalent non-terminal in CamelCase;
    foreach *Numeric Terminal Value $t \in e$* **do**
        Replace t with equivalent terminal value in unicode;
    if $\exists a \in e \mid$ *a is of the form 1*(a)* **then**
        Replace with $a+$;
    if $\exists a \in e \mid$ *a is of the form *(a) or 0*(a)* **then**
        Replace with $a*$;
    if $\exists a \in e \mid$ *a is of the form [a] or *1(a) or 0*1(a)* **then**
        Replace with $a?$;
    Replace comment symbol with #;
    `RemoveLeftRecursion(`A_i`)`

foreach $r \in R_A \mid r \rightarrow A\ =/\ e'$ **do**
    Append e' to existing (A, e) rule, resulting in $A \leftarrow e/e'$;

if *Grammar has no ϵ-productions and is Acyclic* **then**
    for $i = 1...n$ **do**
        Let the current A_i production be (A_i, e_i);
        for $j = 1...i - 1$ **do**
            Let the current (A_j, e_j) production be $A_j \leftarrow \delta_1/.../\delta_k$;
            if $\exists A_j \in e_i$ **then**
                Replace $A_i \leftarrow A_j\gamma$ with $A_i \leftarrow \delta_1\gamma/.../\delta_k\gamma$;
        `RemoveLeftRecursion(`A_i`)`

Algorithm 2: Left Recursion Removal

input : Left recursive PEG rule of the form
$A_i \leftarrow A_i\alpha_1$ / ... / $A_i\alpha_n$ / β_1 / ... / β_m
output: Right recursive PEG rule

Replace A_i by $A_i \leftarrow \beta_1 A_i'$ / ... / $\beta_m A_i'$;
Where $A_i' \leftarrow \epsilon$ / $\alpha_1 A_i'$ / ... / $\alpha_n A_i'$;