

Dao Macro system

Limin Fu (phoolimin@gmail.com)

September 19, 2009

Dao supports a decent macro system which can be used to create new syntax or to simplify scripts. A set of markups can be used in a macro definition to control the syntax matching and transformation. These markups are similar to what are commonly used in BNF representation of syntax. So that defining a macro resembles to write a BNF-like representation of the source syntax to be matched, and a BNF-like representation of the target syntax to be applied.

```
syntax{  
    source_syntax_pattern  
}as{  
    target_syntax_pattern  
}
```

The following control symbols are recognized by the Dao macro system,

Control Tokens	Meaning
()	grouping
?	optional(repeat zero or once), must follow ()
*	repeat zero or more times, must follow ()
+	repeat once or more times, must follow ()
!	repeat zero times, ie, the previous group should not appear, must follow ()
[]	optional, equivalent to () ?
{ }	repeat zero or more times, equivalent to () *
	alternative pattern, must be inside paired control tokens such as (), [], { }

If ?,*,+,! follow the special grouping[] or { }, these special grouping is considered the same as ()

The following prefixes can be used to define special variables that may also be use in the patterns,

Special Variables Prefix	Meaning
\$ID	a valid identifier
\$OP	a valid operater
\$EXP	an expression or subexpression
\$BL	a block of code may contain any type of syntax structures
\$VAR	a temporary variable used in the target syntax pattern

The Dao parser will generate unique names for each macro application for temporary variables represented by **VAR** -prefixed literal. **\$EXP** -prefixed variable normally matches a whole expression, but in the case that there is a normal token (neither control token, nor special variable) following the variable in the defintion, it may match a sub-expression. **\$BL** prefixed variable matches a set of statements (code BLock), and extend the matching as much as possible, similarly, if there is a normal token following the variable in the defintion, this normal token is used to decide the boundary of the code block. If **\$EXP,\$BL** variables are followed by a grouping pattern, the macro system will check the patterns in the grouping to figure out if there is a proper stopping token for the special variables.

Normal Dao operators can be used in the syntax patterns. If brackets are used, they must be properly paired. To use an unpaired bracket, enclose it in quotes such as '(', '[', '{' etc.

For special variables with the same names in the source and targe syntax patterns, it is preferable that they appear in the same grouping and repeating patterns in the source and target syntax. In this case, the syntax transformation has well defined behaviour. After the source pattern is matched and the tokens matching the special variables are extracted, the target pattern is scanned, and the extracted tokens are expanded at the position of the proper special variables.

Starting from a release in 2007, the core syntax of Dao no longer supports scoping by **do-end** , **else-end** , **routine-end** and **class-end** etc. Instead {} should be used for scoping, as in some early releases. However, scoping by **do(else,routine,class,...)-end** can be still supported by the following macros:

```
# Syntax transformation macros

syntax{ # if do elif else end
  if $EXP_1 do \[ $BL_1 \]
  { elif $EXP_2 do \[ $BL_2 \] }
  \[ else \[ $BL_3 \] \]
  end
}as{ # if(){}elif(){}else{}
```

```

    if( 1 ){
        if( $EXP_1 ){ \[ $BL_1 \] }
        { elif( $EXP_2 ){ \[ $BL_2 \] } }
        \[ else{ \[ $BL_3 \] } \]
    }
}

syntax{ if $EXP break }as{ if( $EXP ){ break } }
syntax{ if $EXP skip }as{ if( $EXP ){ skip } }

syntax{
    if $EXP_1 $ID { $EXP_2 } ;
}as{
    if( $EXP_1 ){ $ID { $EXP_2 } }
}

syntax{ # while do end
    while $EXP do \[ $BL \] end
}as{ # while(){
    while( $EXP ){ \[ $BL \] }
}

syntax{ # for in do end
    for $EXP_1 in $EXP_2 { ; $EXP_3 in $EXP_4 } do \[ $BL_1 \] end
}as{ # for( in ){
    for( $EXP_1 in $EXP_2 { ; $EXP_3 in $EXP_4 } ){ \[ $BL_1 \] }
}

syntax{ # for ; ; do end
    for \[ $EXP_1 \]; \[ $EXP_2 \]; \[ $EXP_3 \] do \[ $BL_1 \] end
}as{ # for( ; ; ){
    for( \[ $EXP_1 \]; \[ $EXP_2 \]; \[ $EXP_3 \] ){ \[ $BL_1 \] }
}

syntax{ # switch do end
    switch $EXP do \[ $BL \] end
}as{ # switch(){
    switch( $EXP ){ \[ $BL \] }
}

# routine definition
syntax{
    routine $ID1 { :: $ID2 } ( \[ $BL1 \] ) \ ( ; \) \!
        \ ( { $BL2 } \) \!
        \[ $BL3 \]
    end
}as{
    routine $ID1 { :: $ID2 } ( \[ $BL1 \] ){
        \[ $BL3 \]
    }
}

# class definition
syntax{

```

```

class $ID1 { :: $ID2 } \[ ( \[ $BL1 \] ) \]
  \[ : $ID3 { :: $ID4 } \[ ( \[ $BL2 \] ) \]
  { , $ID5 { :: $ID6 } \[ ( \[ $BL3 \] ) \] } \]
  \[ ( { $BL5 } \) \!
  \[ $BL4 \]
end
}as{
class $ID1 { :: $ID2 } \[ ( \[ $BL1 \] ) \]
  \[ : $ID3 { :: $ID4 } \[ ( \[ $BL2 \] ) \]
  { , $ID5 { :: $ID6 } \[ ( \[ $BL3 \] ) \] } \]
  {
  \[ $BL4 \]
  }
}

syntax{ # try rescue do rescue end
  try \[ $BL_1 \]
  { rescue $BL_2 do \[ $BL_3 \] }
  \[ rescue \[ $BL_4 \] \]
end
}as{ # try{}rescue(){}rescue{}
  if( 1 ){
    try{ \[ $BL_1 \] }
    { rescue( $BL_2 ){ \[ $BL_3 \] } }
    \[ rescue{ \[ $BL_4 \] } \]
  }
}

syntax{ if $EXP retry }as{ if( $EXP ){ retry } }

```
