

道语言宏系统

傅利民(phoolimin@gmail.com)

September 19, 2009

道语言支持比较好的宏系统，它可以被用来定义新的语法。道语言的宏比较象BNF形式的语法表示。道宏里，由一组语法标记来定义需要应用宏的语法模式，并由另一组语法标记需要转换到的道语言基本语法模式。当道解释器编译道代码时，它将用宏的语法模式去匹配代码，并将匹配的代码转换成目标遵循道语言基本语法的代码。

```
syntax{  
    源语法模式  
}as{  
    目标语法模式  
}
```

下面的控制符号可用来标记语法模式：

控制符号	含义
()	分组
?	可选(重复0或1次), 必须跟在()后面
*	重复0或任意多次, 必须跟在()后面
+	重复1或任意多次, 必须跟在()后面
!	重复0次, 即前面的分组必须不出现, 必须跟在()后面
[]	可选, 等价于()?
{ }	重复0或任意多次, ()*
	替代模式, 必须出现在(), [], {}这样的一对控制符号之间

如果?,*,+,! 出现在特殊分组[] 或{ } 后面, 这些特殊分组将被视为跟() 等同。

下面的特殊前缀可用来定义语法模式里的语法变量：

前缀	含义
\$ID	有效的标志名
\$OP	有效的操作符
\$EXP	表达式或子表达式
\$BL	可包含任意语法结构的语句块
\$VAR	目标语法模式里使用的临时变量

在每次进行宏语法转换时，道解释器的语法解析器将为宏里的每个临时变量（带\$VAR前缀）生成一个唯一的变量名。一般情况下，带\$EXP前缀的语法变量将匹配一个整的表达式，但如果该语法变量后面跟了一个普通的字符标志（非控制符，非语法变量），那么该语法变量将可以匹配一个子表达式。带\$BL前缀的语法变量将匹配一个语句块，并尽量地延伸其匹配。类似的，如果该\$BL前缀的语法变量后面跟了一个普通的字符标志，那么此标志将标记该语法变量匹配的语句块的边界。另外，如果带\$EXP或\$BL前缀的语法变量后面跟了一个模式分组，那么模式分组里的模式将被用来确定语法变量所要匹配的表达式或语句块的边界。

普通语言运算符也可被用到语法模式里。当括号被用语法模式里时，它们应该被合理配对。要使用不配对的括号，把它们放到单引号里，如`(`, `]`, `{}`等。

对于在源和目标语法模式里的同名语法变量，它们最好是出现在具有类似结构的分组和重复模式里。这样语法变换将有确定的行为。当源语法模式被匹配到某段代码时，相应于语法变量的代码将被提取出来，然后，目标语法模式被扫描，被提取出来的代码将在对应的语法变量处展开。展开的结果将被用来替换被源模式匹配的代码部分。

道语言的早期版本曾支持过do-end, else-end, routine-end和class-end等形式的语法结构。但从2007年的一次发布起不再对它们作核心支持，不过，还是可以使用下面的宏作支持：

```
# 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 } }

```
