BNF DA LINGUAGEM PERL

```
{
  generate=[psi-factory="no"]
  parserImports=[ "static com.intellij.lang.WhitespacesBinders.*" ]
  elementTypeHolderClass="com.perl5.lang.perl.lexer.PerlElementTypesGenerated"
  parserClass="com.perl5.lang.perl.parser.PerlParserGenerated"
  extends="com.perl5.lang.perl.psi.impl.PerlCompositeElementImpl"

  psiClassPrefix="PsiPerl"
  psiImplClassSuffix="Impl"
  psiPackage="com.perl5.lang.perl.psi"
  psiImplPackage="com.perl5.lang.perl.psi.impl"

  elementTypeHolderClass="com.perl5.lang.perl.lexer.PerlElementTypes"

  tokenTypeClass="com.perl5.lang.perl.parser.elementTypes.PerlTokenType"
  elementTypeClass="com.perl5.lang.perl.parser.elementTypes.PerlElementType"


elementTypeFactory="com.perl5.lang.perl.parser.elementTypes.PerlElementTypeFactory.get
ElementType"

tokenTypeFactory="com.perl5.lang.perl.parser.elementTypes.PerlElementTypeFactory.getTo
kenType"

  parserUtilClass="com.perl5.lang.perl.parser.PerlParserUtil"

  tokens=[
    COMMENT_LINE='COMMENT_LINE'
    COMMENT_BLOCK='COMMENT_BLOCK'
    POD='POD'
```

```
SIGIL_ARRAY='$@'
SIGIL_SCALAR='$$'
SIGIL_SCALAR_INDEX='$#'
SIGIL_GLOB='$*'
SIGIL_HASH='$%'
SIGIL_CODE='$&'

        LEFT_BRACE_SCALAR='${'
        LEFT_BRACE_ARRAY='@{'
        LEFT_BRACE_HASH='%{'
        LEFT_BRACE_GLOB='*{'
        LEFT_BRACE_CODE='&{'

        RIGHT_BRACE_SCALAR='$}'
        RIGHT_BRACE_ARRAY='@}'
        RIGHT_BRACE_HASH='%}'
        RIGHT_BRACE_GLOB='*}'
        RIGHT_BRACE_CODE='&}'

        // postfix deref
        DEREF_SCALAR='->$*'
        DEREF_SCALAR_INDEX='->$#*'
        DEREF_ARRAY='->@*'
        DEREF_HASH='->%*'
        DEREF_GLOB='->**'
        DEREF_CODE='->&*'

// generated tokens
SCALAR_NAME='SCALAR_NAME'
ARRAY_NAME='ARRAY_NAME'
HASH_NAME='HASH_NAME'
GLOB_NAME='GLOB_NAME'

HEREDOC_END='HEREDOC_END'
HEREDOC_END_INDENTABLE='HEREDOC_END_INDENTABLE'

FORMAT='FORMAT'
FORMAT_TERMINATOR='.'

VERSION_ELEMENT='VERSION_ELEMENT'

NUMBER_VERSION='NUMBER_VERSION'
NUMBER='NUMBER'
NUMBER_HEX = 'NUMBER_HEX'
NUMBER_OCT = 'NUMBER_OCT'
NUMBER_BIN = 'NUMBER_BIN'
```

```
STRING_SPECIAL_TAB='\t'
STRING_SPECIAL_NEWLINE='\n'
STRING_SPECIAL_RETURN='\r'
STRING_SPECIAL_FORMFEED='\f'
STRING_SPECIAL_BACKSPACE='\b'
STRING_SPECIAL_ALARM='\a'
STRING_SPECIAL_ESCAPE='\e'
STRING_SPECIAL_ESCAPE_CHAR='\\"'

STRING_SPECIAL_LCFIRST='\l'
STRING_SPECIAL_TCFIRST='\\u'

STRING_SPECIAL_SUBST='\c*'
STRING_SPECIAL_BACKREF = '\1'

STRING_SPECIAL_HEX='\x'

STRING_SPECIAL_OCT='\o'
STRING_SPECIAL_OCT_AMBIGUOUS='\0'

STRING_SPECIAL_UNICODE='\N'
STRING_SPECIAL_UNICODE_CODE_PREFIX='U+'

STRING_CHAR_NAME='charname'

STRING_SPECIAL_LEFT_BRACE='"{'
STRING_SPECIAL_RIGHT_BRACE='}"'
STRING_SPECIAL_RANGE='"-'

STRING_SPECIAL_LOWERCASE_START='\L'
STRING_SPECIAL_UPPERCASE_START='\U'
STRING_SPECIAL_FOLDCASE_START='\F'
STRING_SPECIAL_QUOTE_START='\Q'
STRING_SPECIAL_MODIFIER_END='\E'

RESERVED_IF='if'
RESERVED_UNLESS='unless'
RESERVED_ELSIF='elsif'
RESERVED_ELSE='else'
RESERVED_GIVEN='given'
RESERVED_WHILE='while'
RESERVED_UNTIL='until'
RESERVED_FOR='for'
RESERVED_FOREACH='foreach'

RESERVED_CONTINUE='continue'
RESERVED_WHEN='when'
RESERVED_DEFAULT='default'
```

```
RESERVED_FORMAT='format'
RESERVED_SUB='sub'
RESERVED_ASYNC='async'
RESERVED_PACKAGE='package'
RESERVED_USE='use'
RESERVED_NO='no'
RESERVED_REQUIRE='require'

RESERVED_PRINT='print'
RESERVED_PRINTF='printf'
RESERVED_SAY='say'

RESERVED_MAP='map'
RESERVED_GREP='grep'
RESERVED_SORT='sort'

RESERVED_SCALAR='scalar';
RESERVED_EACH='each'
RESERVED_KEYS='keys'
RESERVED_VALUES='values'
RESERVED_DELETE='delete'
RESERVED_SPLICE='splice'

RESERVED_DEFINED='defined'
RESERVED_WANTARRAY='wantarray'
RESERvED_BLESS='bless'

RESERVED_POP='pop'
RESERVED_SHIFT='shift'

RESERVED_PUSH='push'
RESERVED_UNSHIFT='unshift'

RESERVED_REF='ref'
RESERVED_SPLIT='split'
RESERVED_JOIN='join'
RESERVED_LENGTH='length'
RESERVED_EXISTS='exists'

RESERVED_UNDEF='undef'

RESERVED_QW='qw'

RESERVED_QQ='qq'
RESERVED_Q='q'
RESERVED_QX='qx'
```

```
RESERVED_TR='tr'
RESERVED_Y='y'

RESERVED_S='s'
RESERVED_QR='qr'
RESERVED_M='m'

RESERVED_FINALLY = 'finally';
RESERVED_TRY = 'try';
RESERVED_TRYCATCH = 'TryCatch::';
RESERVED_CATCH = 'catch';
RESERVED_CATCH_WITH = 'catch_with';
RESERVED_EXCEPT = 'except';
RESERVED_OTHERWISE = 'otherwise';
RESERVED_CONTINUATION = 'continuation';

RESERVED_SWITCH='switch'
RESERVED_CASE='case'

RESERVED_MY='my'
RESERVED_OUR='our'
RESERVED_STATE='state'
RESERVED_LOCAL='local'

RESERVED_DO='do'
RESERVED_EVAL='eval'

RESERVED_GOTO='goto'
RESERVED_REDO='redo'
RESERVED_NEXT='next'
RESERVED_LAST='last'

RESERVED_RETURN='return'
RESERVED_EXIT='exit'


RESERVED_METHOD='method'
RESERVED_FUNC='func'
RESERVED_FUN='fun'
RESERVED_METHOD_FP='fp_method'

RESERVED_AFTER_FP='fp_after'
RESERVED_AROUND_FP='fp_around'
RESERVED_AUGMENT_FP='fp_augment'
RESERVED_BEFORE_FP='fp_before'
RESERVED_OVERRIDE_FP='fp_override'

// Operators
```

```
OPERATOR_X='x'

OPERATOR_CMP_NUMERIC='<=>'
OPERATOR_LT_NUMERIC='<'
OPERATOR_GT_NUMERIC='>'

OPERATOR_DEREFERENCE='->'
FAT_COMMA='=>'
COMMA=','

OPERATOR_HELLIP='...'
OPERATOR_NYI='nyi'
OPERATOR_FLIP_FLOP='..'
OPERATOR_CONCAT='.'

OPERATOR_PLUS_PLUS='++'
OPERATOR_MINUS_MINUS='--'
OPERATOR_POW='**'

OPERATOR_RE='=~'
OPERATOR_NOT_RE='!~'

OPERATOR_HEREDOC='heredoc<<'
OPERATOR_SHIFT_LEFT='<<'
OPERATOR_SHIFT_RIGHT='>>'

OPERATOR_AND='&&'
OPERATOR_OR='||'
OPERATOR_OR_DEFINED='//'
OPERATOR_NOT='!'

OPERATOR_ASSIGN='='

QUESTION='?'
COLON=':'

OPERATOR_REFERENCE='\\'

OPERATOR_DIV='/'
OPERATOR_MUL='*'
OPERATOR_MOD='%'

OPERATOR_PLUS='+'
OPERATOR_MINUS='-'
```

```
    OPERATOR_BITWISE_NOT='~'
    OPERATOR_BITWISE_AND='&'
    OPERATOR_BITWISE_OR='|'
    OPERATOR_BITWISE_XOR='^'

    OPERATOR_AND_LP='and'
    OPERATOR_OR_LP='or'
    OPERATOR_XOR_LP='xor'
    OPERATOR_NOT_LP='not'

    OPERATOR_ISA='isa'

    OPERATOR_LT_STR='lt'
    OPERATOR_GT_STR='gt'
    OPERATOR_LE_STR='le'
    OPERATOR_GE_STR='ge'
    OPERATOR_CMP_STR='cmp'
    OPERATOR_EQ_STR='eq'
    OPERATOR_NE_STR='ne'

    // synthetic tokens
    OPERATOR_POW_ASSIGN='**='
    OPERATOR_PLUS_ASSIGN='+='
    OPERATOR_MINUS_ASSIGN='-='
    OPERATOR_MUL_ASSIGN='*='
    OPERATOR_DIV_ASSIGN='/='
    OPERATOR_MOD_ASSIGN='%='
    OPERATOR_CONCAT_ASSIGN='.='
    OPERATOR_X_ASSIGN='x='
    OPERATOR_BITWISE_AND_ASSIGN='&='
    OPERATOR_BITWISE_OR_ASSIGN='|='
    OPERATOR_BITWISE_XOR_ASSIGN='^='
    OPERATOR_SHIFT_LEFT_ASSIGN='<<='
    OPERATOR_SHIFT_RIGHT_ASSIGN='>>='
    OPERATOR_AND_ASSIGN='&&='
    OPERATOR_OR_ASSIGN='||='
    OPERATOR_OR_DEFINED_ASSIGN='//='

    OPERATOR_GE_NUMERIC='>='
    OPERATOR_LE_NUMERIC='<='
    OPERATOR_EQ_NUMERIC='=='
    OPERATOR_NE_NUMERIC='!='
    OPERATOR_SMARTMATCH='~~'
// end of synthetic operators

    OPERATOR_FILETEST='-t'

    // single mid-quote. e evaluatable s///e;
    REGEX_QUOTE='r/'
    REGEX_QUOTE_E='re/'
    REGEX_TOKEN='regex'

    // paired mid-quote. e for evaluatable s{}{}e;
    REGEX_QUOTE_OPEN='r{'
    REGEX_QUOTE_OPEN_E='re{' // block should be interpolated as a perl script

    REGEX_QUOTE_CLOSE='r}'
    REGEX_MODIFIER='/m'

/*
        REGEX_LEFT_BRACKET = '[['
        REGEX_RIGHT_BRACKET = ']]'
        REGEX_LEFT_PAREN = '(('
        REGEX_RIGHT_PAREN = '))'
        REGEX_LEFT_BRACE = '{{'
        REGEX_RIGHT_BRACE = '}}'
        REGEX_POSIX_LEFT_BRACKET = '[:'
        REGEX_POSIX_RIGHT_BRACKET = ':]'
        REGEX_POSIX_CLASS_NAME = ':name:'
        REGEX_CHAR_CLASS="\w"
*/

    STRING_CONTENT='STRING_CONTENT'
    STRING_CONTENT_QQ='STRING_CONTENT_QQ'
    STRING_CONTENT_XQ='STRING_CONTENT_XQ'


    TAG='TAG'
    TAG_END='__END__'
    TAG_DATA='__DATA__'
    TAG_PACKAGE='__PACKAGE__'

    LEFT_ANGLE='LEFT_ANGLE'
    RIGHT_ANGLE='RIGHT_ANGLE'

    TYPE_ARRAYREF="ArrayRef"
    TYPE_HASHREF="HashRef"

    LEFT_BRACKET='['
    RIGHT_BRACKET=']'

    LEFT_PAREN='('
    RIGHT_PAREN=')'

    LEFT_BRACE='{'
    RIGHT_BRACE='}'
```

```
SEMICOLON=';';

QUOTE_DOUBLE='QUOTE_DOUBLE'
QUOTE_DOUBLE_OPEN='QUOTE_DOUBLE_OPEN'
QUOTE_DOUBLE_CLOSE='QUOTE_DOUBLE_CLOSE'

QUOTE_SINGLE='QUOTE_SINGLE'
QUOTE_SINGLE_OPEN='QUOTE_SINGLE_OPEN'
QUOTE_SINGLE_CLOSE='QUOTE_SINGLE_CLOSE'

QUOTE_TICK='QUOTE_TICK'
QUOTE_TICK_OPEN='QUOTE_TICK_OPEN'
QUOTE_TICK_CLOSE='QUOTE_TICK_CLOSE'

// custom tokens
IDENTIFIER='IDENTIFIER'
        SUB_NAME='subname'

BUILTIN_LIST='list'
BUILTIN_UNARY='unary'
CUSTOM_UNARY='unary_custom'
BUILTIN_ARGUMENTLESS='argumentless'

ATTRIBUTE_IDENTIFIER='ATTRIBUTE_IDENTIFIER'

SUB_PROTOTYPE_TOKEN='SUB_PROTOTYPE_TOKEN'

PACKAGE='package::name'
QUALIFYING_PACKAGE='package::name::'

HANDLE='HANDLE'
BLOCK_NAME='BLOCK_NAME'

ANNOTATION_DEPRECATED_KEY='#@deprecated'
ANNOTATION_RETURNS_KEY='#@returns'
ANNOTATION_OVERRIDE_KEY='#@override'
ANNOTATION_METHOD_KEY='#@method'
ANNOTATION_ABSTRACT_KEY='#@abstract'
ANNOTATION_INJECT_KEY='#@inject'
ANNOTATION_NO_INJECT_KEY='#@noinject'
ANNOTATION_TYPE_KEY='#@type'
ANNOTATION_NOINSPECTION_KEY='#@noinspection'
ANNOTATION_UNKNOWN_KEY='#@unknown'

// lazy parsable tokens parsed in-place
LP_CODE_BLOCK = "LP_CODE_BLOCK"
LP_CODE_BLOCK_WITH_TRYCATCH = "LP_CODE_BLOCK_WITH_TRYCATCH"
```

```
    LP_STRING_RE = "LP_STRING_RE"
    LP_STRING_TR = "LP_STRING_TR"
    LP_STRING_QQ = "LP_STRING_QQ"
    LP_STRING_QQ_RESTRICTED = "LP_STRING_QQ_RESTRICTED"
    LP_STRING_Q = "LP_STRING_Q"
    LP_STRING_QX = "LP_STRING_QX"
    LP_STRING_QX_RESTRICTED = "LP_STRING_QX_RESTRICTED"
    LP_STRING_QW = "LP_STRING_QW"
    LP_REGEX = "LP_REGEX"
    LP_REGEX_X = "LP_REGEX_X"
    LP_REGEX_XX = "LP_REGEX_XX"
    LP_REGEX_SQ = "LP_REGEX_SQ"
    LP_REGEX_X_SQ = "LP_REGEX_X_SQ"
    LP_REGEX_XX_SQ = "LP_REGEX_XX_SQ"
  ]

    extends(".+expr")=expr
    name(".+expr")="expression"
    extends("number_constant")=expr


mixin("around_modifier|after_modifier|before_modifier|augment_modifier")="com.perl5.lang.
perl.psi.mixins.PerlMethodModifierMixin"

implements("around_modifier|after_modifier|before_modifier|augment_modifier")="com.perl5
.lang.perl.psi.PerlMethodModifier"


implements("next_expr|last_expr|redo_expr")="com.perl5.lang.perl.psi.PerlFlowControlExpr"

    implements("bless_expr")="com.perl5.lang.perl.psi.PerlBlessExpr"
    implements("trycatch_expr")="com.perl5.lang.perl.psi.PerlTryCatchExpr"
    implements("try_expr")="com.perl5.lang.perl.psi.PerlTryExpr"
    implements("catch_expr|continuation_expr")="com.perl5.lang.perl.psi.PerlCatchExpr"

    implements("return_expr")="com.perl5.lang.perl.psi.PerlReturnExpr"
    implements("defined_expr")="com.perl5.lang.perl.psi.PerlImplicitScalarExpr"


implements("condition_expr|foreach_iterator|signature_content|for_init|for_condition|for_mut
ator")="com.perl5.lang.perl.psi.PerlStatement"

          implements(".+_cast_expr")="com.perl5.lang.perl.psi.PerlCastExpression"
          mixin(".+_cast_expr")="com.perl5.lang.perl.psi.mixins.PerlCastExpressionMixin"

          implements("assign_expr")="com.perl5.lang.perl.psi.PerlAssignExpression"

          extends(".*statement_modifier")=statement_modifier
```

```
        implements("statement_modifier")="com.perl5.lang.perl.psi.PerlStatementModifier"


implements("variable_declaration_lexical")="com.perl5.lang.perl.psi.PerlLexicalVariableDecl
arationMarker"
        implements("signature_element")="com.perl5.lang.perl.psi.PerlSignatureElement"

  extends("heredoc_opener|anon_array|anon_hash")=expr

  mixin("package_expr")="com.perl5.lang.perl.psi.mixins.PerlPackageExpression"

  extends("string_list")=expr
  mixin("string_list")="com.perl5.lang.perl.psi.mixins.PerlStringListMixin"

  implements("replacement_regex")="com.perl5.lang.perl.psi.PerlReplacementRegex"
  implements("match_regex|compile_regex")="com.perl5.lang.perl.psi.PerlSimpleRegex"
  extends("replacement_regex|compile_regex|match_regex|tr_regex")=expr

  mixin("perl_regex")="com.perl5.lang.perl.psi.mixins.Perl5RegexpMixin"

  extends("heredoc_opener|tag_scalar")=expr

  mixin("unicode_char")="com.perl5.lang.perl.psi.mixins.PerlUnicodeSubstitutionMixin"
  mixin("hex_char")="com.perl5.lang.perl.psi.mixins.PerlHexSubstitutionMixin"
  mixin("oct_char")="com.perl5.lang.perl.psi.mixins.PerlOctSubstitutionMixin"
  mixin("esc_char")="com.perl5.lang.perl.psi.mixins.PerlEscSubstitutionMixin"

implements("hex_char|oct_char|unicode_char|esc_char")="com.perl5.lang.perl.psi.PerlChar
Substitution"

  extends("string_sq|string_dq|string_xq|string_bare")=expr

implements("string_sq|string_dq|string_xq|string_bare")="com.perl5.lang.perl.psi.PerlString"
  mixin("string_sq|string_dq|string_xq")="com.perl5.lang.perl.psi.mixins.PerlStringMixin"
  mixin("string_bare")="com.perl5.lang.perl.psi.mixins.PerlStringBareMixin"

  implements("sub_call")="com.perl5.lang.perl.psi.PerlMethodContainer"
  mixin("sub_call")="com.perl5.lang.perl.psi.impl.PerlSubCallElement"
  extends("sub_call")=expr
  stubClass("sub_call")="com.perl5.lang.perl.psi.stubs.calls.PerlSubCallElementStub"

  mixin("statement")="com.perl5.lang.perl.psi.mixins.PerlStatementMixin"

        mixin("call_arguments")="com.perl5.lang.perl.psi.mixins.PerlCallArguments"
        extends("parenthesised_call_arguments")=call_arguments
```

```
implements("named_block|conditional_block|unconditional_block")="com.perl5.lang.perl.psi.
PerlStatementsContainerWithBlock"

        implements("label_declaration")="com.perl5.lang.perl.psi.PerlLabelDeclaration"

extends("label_declaration")="com.perl5.lang.perl.psi.mixins.PerlLabelDeclarationMixin"

        implements("block")="com.perl5.lang.perl.psi.PerlBlock"

  implements("block_compound")="com.perl5.lang.perl.psi.PerlBlockCompound"
  implements("for_compound")="com.perl5.lang.perl.psi.PerlForCompound"
  implements("foreach_compound")="com.perl5.lang.perl.psi.PerlForeachCompound"

implements("while_compound|until_compound")="com.perl5.lang.perl.psi.PerlWhileUntilCom
pound"

implements("if_compound|unless_compound")="com.perl5.lang.perl.psi.PerlIfUnlessCompou
nd"

implements("when_compound")="com.perl5.lang.perl.psi.properties.PerlConvertableCompo
undSimple"
  implements("given_compound")="com.perl5.lang.perl.psi.PerlGivenCompound";
  implements("default_compound")="com.perl5.lang.perl.psi.properties.PerlCompound";
  implements("trycatch_compound")="com.perl5.lang.perl.psi.PerlTryCatchCompound"

  implements("heredoc_opener")="com.perl5.lang.perl.psi.PerlHeredocOpener"
  mixin("heredoc_opener")="com.perl5.lang.perl.psi.mixins.PerlHeredocOpenerMixin"

  implements("deref_expr")="com.perl5.lang.perl.psi.PerlDerefExpression"
  mixin("deref_expr")="com.perl5.lang.perl.psi.mixins.PerlDerefExpressionMixin"


mixin("parenthesised_expr")="com.perl5.lang.perl.psi.mixins.PerlParenthesizedExpressionM
ixin"


extends("variable_declaration_global|variable_declaration_lexical|variable_declaration_local
")=expr

implements("variable_declaration_lexical|variable_declaration_local|variable_declaration_glo
bal")="com.perl5.lang.perl.psi.PerlVariableDeclarationExpr"

mixin("variable_declaration_lexical|variable_declaration_local|variable_declaration_global")=
"com.perl5.lang.perl.psi.mixins.PerlVariableDeclarationExprMixin"

  extends("code_variable")=expr
extends("array_slice|hash_slice|hash_array_slice|hash_hash_slice|array_element|hash_ele
ment|glob_slot")=expr


stubClass("variable_declaration_element")="com.perl5.lang.perl.psi.stubs.variables.PerlVari
ableDeclarationStub"

mixin("variable_declaration_element")="com.perl5.lang.perl.psi.mixins.PerlVariableDeclaratio
nElementMixin"

implements("variable_declaration_element")="com.perl5.lang.perl.psi.PerlVariableDeclaratio
nElement"


extends("array_index_variable|scalar_variable|array_variable|hash_variable|glob_variable")=
expr

mixin("code_variable|scalar_variable|array_variable|hash_variable|array_index_variable")="c
om.perl5.lang.perl.psi.mixins.PerlVariableMixin"

implements("code_variable|scalar_variable|array_variable|hash_variable|array_index_variab
le")="com.perl5.lang.perl.psi.PerlVariable"


mixin("label_expr")="com.perl5.lang.perl.psi.impl.PerlCompositeElementWithReference"


stubClass("namespace_definition")="com.perl5.lang.perl.psi.stubs.namespaces.PerlNamesp
aceDefinitionStub"

mixin("namespace_definition")="com.perl5.lang.perl.psi.mixins.PerlNamespaceDefinitionMixi
n"

implements("namespace_definition")="com.perl5.lang.perl.psi.PerlNamespaceDefinitionWith
Identifier"


stubClass("method_definition")="com.perl5.lang.perl.psi.stubs.subsdefinitions.PerlSubDefinit
ionStub"
   mixin("method_definition")="com.perl5.lang.perl.psi.mixins.PerlMethodDefinitionMixin"
   implements("method_definition")="com.perl5.lang.perl.psi.PerlMethodDefinition"


stubClass("func_definition")="com.perl5.lang.perl.psi.stubs.subsdefinitions.PerlSubDefinition
Stub"
   mixin("func_definition")="com.perl5.lang.perl.psi.mixins.PerlFuncDefinitionMixin"
   implements("func_definition")="com.perl5.lang.perl.psi.PerlSubDefinitionElement"
```

```
stubClass("sub_definition")="com.perl5.lang.perl.psi.stubs.subsdefinitions.PerlSubDefinition
Stub"
    mixin("sub_definition")="com.perl5.lang.perl.psi.mixins.PerlSubDefinitionMixin"
    implements("sub_definition")="com.perl5.lang.perl.psi.PerlSubDefinitionElement"


stubClass("sub_declaration")="com.perl5.lang.perl.psi.stubs.subsdeclarations.PerlSubDeclar
ationStub"
    mixin("sub_declaration")="com.perl5.lang.perl.psi.mixins.PerlSubDeclarationBase"
    implements("sub_declaration")="com.perl5.lang.perl.psi.PerlSubDeclarationElement"
    extends("sub_declaration")=statement

    stubClass("glob_variable")="com.perl5.lang.perl.psi.stubs.globs.PerlGlobStub"
    mixin("glob_variable")="com.perl5.lang.perl.psi.mixins.PerlGlobVariableMixin"
    implements("glob_variable")="com.perl5.lang.perl.psi.PerlGlobVariable"

    mixin("require_expr")="com.perl5.lang.perl.psi.mixins.PerlRequireExprMixin"
    implements("require_expr")="com.perl5.lang.perl.psi.PerlRequireExpr"

stubClass("require_expr")="com.perl5.lang.perl.psi.stubs.imports.runtime.PerlRuntimeImport
Stub"

            implements("grep_expr")="com.perl5.lang.perl.psi.PerlGrepExpr"
            implements("map_expr")="com.perl5.lang.perl.psi.PerlMapExpr"
            implements("sort_expr")="com.perl5.lang.perl.psi.PerlSortExpr"
            implements("eval_expr")="com.perl5.lang.perl.psi.PerlEvalExpr"

    implements("sub_expr|fun_expr|method_expr")="com.perl5.lang.perl.psi.PerlSubExpr"

mixin("sub_expr|fun_expr|method_expr")="com.perl5.lang.perl.psi.mixins.PerlSubExpression
"

    implements("do_block_expr")="com.perl5.lang.perl.psi.PerlDoBlockExpr"

    mixin("do_expr")="com.perl5.lang.perl.psi.mixins.PerlDoExprMixin"
    implements("do_expr")="com.perl5.lang.perl.psi.PerlDoExpr"

stubClass("do_expr")="com.perl5.lang.perl.psi.stubs.imports.runtime.PerlRuntimeImportStub
"

    mixin("method")="com.perl5.lang.perl.psi.mixins.PerlMethodMixin";
    implements("method")="com.perl5.lang.perl.psi.PerlMethod";


implements("namespace_content")="com.perl5.lang.perl.psi.properties.PerlStatementsConta
iner"
```

```
implements("annotation_type|annotation_returns")="com.perl5.lang.perl.psi.PerlAnnotationW
ithValue"
   implements("annotation_.*")="com.perl5.lang.perl.psi.PerlAnnotation"
   mixin("annotation_inject")="com.perl5.lang.perl.psi.mixins.PerlAnnotationInjectMixin"
 }

////////////////////////////// main code structure ///////////////////////////////////////////////////////////////////
root ::= <<parseFileContent>> file_items
// invoked by parser
private file_items ::= file_item*

private file_item ::= !<<eof>>
{
         namespace_definition
         | label_declaration [statement_item]
   | statement_item
}

private statement_item ::=
      <<parseSemicolon>> +
      | nyi_statement
      | <<parseParserExtensionStatement>>
      | named_definition
      | compound_statement
      | format_definition
      | <<parseUse>>
      | <<parseNo>>
      | block_compound
      | statement
      | annotation
      | pod_section
      | end_or_data
      | <<parseBadCharacters>> // Fallback for bad characters

private pod_section ::= POD {name="pod section"}
private end_or_data ::= '__DATA__' | '__END__' {name="__END__ or __DATA__"}

// invoked from PerlUseVarsDeclarationsParser
private use_vars_declarations ::=
{variable_declaration_element|glob_variable|code_variable}*

namespace_definition ::= namespace_definition_name (block | <<parseSemicolon>>
<<parseNamespaceContent>>) {pin=1 recoverWhile=recover_statement}
private namespace_definition_name ::= 'package' any_package [perl_version] {pin=1
recoverWhile=recover_statement name="namespace definition"}
namespace_content ::= real_namespace_content
```

```
private real_namespace_content ::= {!'package' file_item} *  {extends=block
recoverWhile=recover_statement} //

// used in com.perl5.lang.perl.parser.PerlLazyBlockParser
block_braceless ::= file_items {extends=block
hooks=[rightBinder="GREEDY_RIGHT_BINDER" leftBinder="GREEDY_LEFT_BINDER"]}
block ::= '{' block_content '}' {extraRoot=true pin=1}
// this is uncertain or derivative block. should be not reparseable, may be a hash
private block_content ::= file_item * {recoverWhile=recover_statement}

nyi_statement ::= 'nyi' {name="statement"}
format_definition ::=  'format' ['subname'] '=' [FORMAT] FORMAT_TERMINATOR {pin=1
name="format definition"}

private named_definition ::=
    sub_definition
    | named_block
    | method_definition
    | func_definition
    | before_modifier
    | after_modifier
    | around_modifier
    | augment_modifier

private compound_statement ::=
    if_compound
    | unless_compound
    | given_compound
    | while_compound
    | until_compound
    | for_or_foreach
    | when_compound
    | default_compound
    | trycatch_compound
    | switch_compound
    | cases_sequence {name="compound statement"}


named_block ::= BLOCK_NAME block {name="named block"}

if_compound ::=  'if' conditional_block if_compound_elsif * [if_compound_else]  {pin=1}
unless_compound ::=  'unless' conditional_block if_compound_elsif * [if_compound_else]
{pin=1}
private if_compound_elsif ::= [POD]  'elsif' conditional_block  {pin=2}
private if_compound_else ::= [POD]  'else' unconditional_block {pin=2}
unconditional_block ::= block

/*
```

```
 Hybrid parsing for try/catch/finally.
 Following syntaxes supported:
 - https://metacpan.org/pod/Try::Catch
 - https://metacpan.org/pod/Try::Tiny
 - https://metacpan.org/pod/Exception::Class::TryCatch
 - https://metacpan.org/pod/TryCatch
 - https://metacpan.org/pod/Error
 - https://metacpan.org/pod/Dancer::Exception
 */
trycatch_compound ::= 'TryCatch::' <<try_expr block>> [<<catch_expr block>>]

trycatch_expr ::= <<try_expr sub_expr_simple_ensured>> (<<catch_expr
sub_expr_simple_ensured>>|finally_expr|except_expr|otherwise_expr|continuation_expr)*
meta try_expr ::= 'try' <<x1>> {pin=1}
meta catch_expr ::= 'catch' [catch_condition] <<x1>> {pin=1}
catch_condition ::= catch_condition_parenthesised | catch_condition_with
private catch_condition_parenthesised ::= '(' catch_condition_content ')' {pin=1}
private catch_condition_with ::= 'package::name' 'catch_with' {pin=2}
private catch_condition_content ::= [catch_condition_type] variable_declaration_element
[where_clause]
private where_clause ::= expr
private catch_condition_type ::=  [type_constraints]
type_constraints ::= any_package ['[' expr ']']

finally_expr ::= 'finally' sub_expr_simple_ensured {pin=1}
except_expr ::= 'except' sub_expr_simple_ensured {pin=1}
otherwise_expr ::= 'otherwise' sub_expr_simple_ensured {pin=1}
continuation_expr ::= 'continuation' sub_expr_simple_ensured {pin=1}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

conditional_block ::= parse_conditional_block
private parse_conditional_block ::= condition_expr block {pin=1}
condition_expr ::= parse_parenthesized_expression {extraRoot=true}

given_compound ::=  'given' parse_conditional_block  {pin=1}
when_compound ::=  'when' parse_conditional_block  {pin=1}
default_compound ::=  'default' block {pin=1}

while_compound ::=  'while' parse_conditional_block [[POD] continue_block]  {pin=1}
until_compound ::=  'until' parse_conditional_block [[POD] continue_block]  {pin=1}

continue_block ::= continue_block_opener block {pin=1}
private continue_block_opener ::= 'continue' &'{'

block_compound ::= parse_block_compound {named="code block"}
private parse_block_compound ::= &('{') !(anon_hash_lookahead) block [[POD]
continue_block] !('->')
```

```
// for/foreach
// fixme why the heck there is no parsing error on "for" and there is an error on "use", both
pinned
// fixme add recover
private for_or_foreach ::= for_compound|foreach_compound

for_compound ::= {'for'|'foreach'} for_iterator block {pin=2}
private for_iterator ::= '(' [for_init]  ';' [for_condition] ';' [for_mutator] ')' {pin=3}
for_init ::= expr {recoverWhile=recover_parenthesised}
for_condition ::= expr {recoverWhile=recover_parenthesised}
for_mutator ::= expr {recoverWhile=recover_parenthesised}

foreach_compound ::= {'for'|'foreach'} [ foreach_iterator ] condition_expr
parse_block_compound {pin=1} // foreach works as a fallback
foreach_iterator ::= variable_declaration | variable


statement ::= sub_declaration | statement_body <<statementSemi>>

private statement_body ::= normal_statement {recoverWhile=recover_statement}

private normal_statement ::= expr [statement_modifier | <<parseStatementModifier>>]
{pin=1}

// fixme adjust parsing of this thing to avoid duplicates
sub_definition ::=  ['my'|'our'|'state'|'async'] 'sub' sub_names_token
sub_definition_parameters block
sub_declaration ::=  ['my'|'our'|'state'] 'sub' sub_names_token sub_declaration_parameters
<<statementSemi>>
private sub_declaration_parameters ::= sub_definition_parameters
{recoverWhile=recover_statement}

private sub_names_token ::= ['package::name::'] 'subname'

private sub_definition_parameters ::=
  sub_attributes [sub_signature_in_parens] |
  [sub_prototype_or_signature] [sub_attributes]
private sub_prototype_or_signature ::= '(' sub_prototype_or_signature_content ')' {pin=1}
private sub_prototype_or_signature_content ::= sub_signature | sub_prototype
private sub_signature_in_parens ::= '(' [sub_signature] ')' {pin=1}

private sub_prototype ::= SUB_PROTOTYPE_TOKEN*

/************************************ Sub signatures
****************************************************************/
private sub_signature ::= <<signature_content parse_sub_signature>>
private parse_sub_signature ::= sub_signature_element (',' sub_signature_element) * ','*
private sub_signature_element ::= <<signature_element parse_sub_signature_element>>
private parse_sub_signature_element ::= signature_left_side ['=' [parse_scalar_expr]]
private signature_left_side ::= variable_declaration_element | sub_signature_element_ignore
sub_signature_element_ignore ::= '$$' | '$@' | '$%'

/************************************ Sub signatures
****************************************************************/

private sub_attributes ::= <<attributes <<parse_sub_attributes>>>>
private parse_sub_attributes ::= ':' attribute ([':'] attribute)*  {pin=1}
private var_attributes ::= <<attributes <<parse_var_attributes>>>>
private parse_var_attributes ::= ':' attribute ([':'] attribute)*  // {pin=1} pin disable because of
$something ? my $var : $other;
meta attributes ::= <<x1>>
attribute ::= ATTRIBUTE_IDENTIFIER [quoted_sq_string]

last_expr ::=  'last' [lnr_param] {pin=1}
next_expr ::=  'next' [lnr_param] {pin=1}
redo_expr ::=  'redo' [lnr_param] {pin=1}
goto_expr ::=  'goto' [goto_param] {pin=1}

private optional_scalar_expr_arguments ::= <<custom_expr_arguments
optional_scalar_expr>>
private unary_expr_arguments ::= <<custom_expr_arguments unary_expr>>
private optional_unary_expr_arguments ::= <<custom_expr_arguments
optional_unary_expr>>
private list_expr_arguments ::= <<custom_expr_arguments parse_list_expr>>
private custom_single_expr_argument ::= <<custom_expr_arguments
single_argument_expr>>

return_expr ::=  'return' [parse_list_expr] {pin=1}
exit_expr ::= 'exit' [optional_scalar_expr_arguments] {pin=1}
scalar_expr ::= 'scalar' custom_single_expr_argument {pin=1}
keys_expr ::= 'keys' custom_single_expr_argument {pin=1}
values_expr ::= 'values' custom_single_expr_argument {pin=1}
each_expr ::= 'each' custom_single_expr_argument {pin=1}
defined_expr ::= 'defined' [optional_unary_expr_arguments] {pin=1}
wantarray_expr ::= 'wantarray' [parenthesised_call_arguments] {pin=1}
delete_expr ::= 'delete' unary_expr_arguments {pin=1}
splice_expr ::= 'splice' list_expr_arguments {pin=1}
bless_expr ::= 'bless' list_expr_arguments {pin=1}

array_unshift_expr ::= 'unshift' any_call_arguments {pin=1
implements="com.perl5.lang.perl.psi.PerlUnshiftPushExpr"}
array_push_expr ::= 'push' any_call_arguments {pin=1
implements="com.perl5.lang.perl.psi.PerlUnshiftPushExpr"}
array_shift_expr ::= 'shift' [any_unary_call_arguments] {pin=1
implements="com.perl5.lang.perl.psi.PerlShiftPopExpr"}
```

```
array_pop_expr ::= 'pop' [any_unary_call_arguments] {pin=1
implements="com.perl5.lang.perl.psi.PerlShiftPopExpr"}

private lnr_param ::= label_expr | expr // fixme scalar_expr ?
private goto_param ::= label_expr | code_primitive  !'(' | expr


statement_modifier ::= statement_modifier_variant !('{'|'(')
private statement_modifier_variant ::=
   if_statement_modifier
   | unless_statement_modifier
   | while_statement_modifier
   | until_statement_modifier
   | for_statement_modifier
   | when_statement_modifier

if_statement_modifier ::=    'if' expr {pin=1 name="Postfix if"}
unless_statement_modifier ::=    'unless' expr {pin=1 name="Postfix unless"}
while_statement_modifier ::=    'while' expr {pin=1 name="Postfix while"}
until_statement_modifier ::=    'until' expr {pin=1 name="Postfix until"}
for_statement_modifier ::=    {{'for'|'foreach'} !(for_iterator)} expr  {pin=1 name="Postfix for"}
when_statement_modifier ::=    'when' expr {pin=1 name="Postfix when"}

private parse_use_statement ::=  'use' <<parseUseParameters use_no_parameters>>
<<statementSemi>> {pin=1}
private parse_no_statement ::=  'no' use_no_parameters <<statementSemi>> {pin=1}

private use_no_parameters ::= use_module_parameters | use_version_parameters
{recoverWhile=recover_statement}
private use_module_parameters ::= any_package [perl_version [comma]] [expr];
private use_version_parameters ::= perl_version;

undef_expr ::=  'undef' (undef_params | '(' undef_params ')') ? {pin=1}
private undef_params ::= deref_expr | variable

require_expr ::=  'require' (any_package| perl_version | parse_scalar_expr)  {pin=1}//
multiline string is possible too

private any_package ::= 'package::name' | '__PACKAGE__'

private recover_statement ::= <<recoverStatement>>

// expression
expr ::=
   lp_or_xor_expr         // 0
   | lp_and_expr          // 1
   | lp_not_expr          // 2
   | comma_sequence_expr   // 3 for list
```

```
    | assign_or_flow_expr   // 4
    | ternary_expr          // 5
    | flipflop_expr         // 6
    | or_expr               // 7
    | and_expr              // 8
    | bitwise_or_xor_expr   // 9
    | bitwise_and_expr      // 10
    | isa_expr              // 11
    | equal_expr            // 12
    | compare_expr          // 13
    | shift_expr            // 14  for unary
    | add_expr              // 15
    | mul_expr              // 16
    | regex_expr            // 17
    | op_5_expr             // 18
    | pow_expr              // 19
    | op_3_expr             // 20
    | deref_expr            // 21 for a single argument
    | atom_expr             // 22

// above list operators
private parse_list_expr ::= <<parseExpressionLevel 2>>
private optional_list_expr ::= [parse_list_expr]

// List expression elements, code checks if priority < than number, see
com.perl5.lang.perl.parser.PerlParserGenerated.expr_0
private parse_scalar_expr ::= <<parseExpressionLevel 3>>
private optional_scalar_expr ::= [parse_scalar_expr]

// Unary expression argument
private unary_expr ::= <<parseExpressionLevel 13>>
private optional_unary_expr ::= [unary_expr]

private single_argument_expr ::= <<parseExpressionLevel 20>>

// ordered for best performance of perltidy
private atom_expr ::=
    composite_atom_expr
    | string
    | number_constant
    | variable_declaration_lexical
    | match_regex
    | return_expr
    | scalar_expr
    | keys_expr
    | values_expr
    | each_expr
    | defined_expr
    | delete_expr
    | splice_expr
    | bless_expr
    | array_shift_expr
    | array_unshift_expr
    | array_push_expr
    | array_pop_expr
    | wantarray_expr
    | exit_expr
    | array_index_variable
    | scalar_index_cast_expr
    | anon_array
    | undef_expr
    | print_expr
    | replacement_regex
    | sub_expr
    | fun_expr
    | method_expr
    | eval_expr
    | do_block_expr
    | do_expr
    | anon_hash
    | variable_declaration_local
    | sort_expr
    | grep_expr
    | map_expr
    | continue_expr

    | tag_scalar
    | variable_declaration_global
    | compile_regex
    | tr_regex
    | file_read_expr
    | file_glob_expr
    | require_expr
    | perl_handle_expr
    | custom_atom_expr

    | trycatch_expr

    | sub_call
    | package_expr

composite_atom_expr ::=
    scalar_or_element
    | parenthesised_expr [array_element]
    | array_or_slice
    | hash_or_slice
    | glob_or_element
    | code_primitive

custom_atom_expr ::= <<parseParserExtensionTerm>>

package_expr ::= any_package
continue_expr ::= 'continue' ['(' ')'] {pin=1}
grep_expr ::=  'grep' parse_grep_map_arguments {pin=1}
map_expr ::=  'map' parse_grep_map_arguments {pin=1}

private parse_grep_map_arguments ::= <<custom_expr_arguments
grep_map_arguments_variants>>

private grep_map_arguments_variants ::=
    grep_map_sort_with_block |
    parse_scalar_expr comma grep_map_sort_tail |
    expr

private grep_map_sort_with_block ::= block [comma] grep_map_sort_tail

private grep_map_sort_tail ::= parse_list_expr

private meta custom_expr_arguments ::='(' <<x1>> ')' !'[' | <<x1>>

sort_expr ::=  'sort' parse_sort_arguments {pin=1}
private parse_sort_arguments ::= <<custom_expr_arguments  sort_arguments_variants>>

private sort_arguments_variants ::=
    grep_map_sort_with_block |
    sorter grep_map_sort_tail |
    grep_map_sort_tail

private sorter ::= scalar_variable | method

parenthesised_expr ::= parse_parenthesized_expression {extraRoot=true}
private parse_parenthesized_expression ::= '(' parenthesised_expr_content ')' {pin=1
name="Parenthesised expression"}
private parenthesised_expr_content ::= [expr] {recoverWhile=recover_parenthesised}
private recover_parenthesised ::= !(')' | '{' | '}' | <<checkSemicolon>> )

deref_expr ::= expr (<<parseArrowSmart>> nested_element_variation) + //{pin(".*")=1}

private op_3_expr ::= pref_pp_expr | suff_pp_expr
pref_pp_expr ::= ('++'|'--') expr
suff_pp_expr ::= expr ('++'|'--')

pow_expr ::= expr ('**' expr)+ { rightAssociative=true }
```

```
private op_5_expr ::= ref_expr | prefix_unary_expr
ref_expr ::= '\\' expr { rightAssociative=true }
prefix_unary_expr ::= {'~' | '!' | '+' | '-'} expr { rightAssociative=true }
regex_expr ::= expr ('=~'|'!~') expr
mul_expr ::= expr ({'*'|'/'|'%'|'x'} expr)+
add_expr ::= expr ({'+'|'-'|'.'} expr)+
shift_expr ::= expr ({'<<'|'>>'} expr)+
compare_expr ::= expr ({'>='|'<='|'>'|'<'|'ge'|'le'|'gt'|'lt'} expr )+
equal_expr ::= expr ({'<=>'|'cmp'|'~~'|'=='|'!='|'eq'|'ne'} expr)+
isa_expr ::= expr 'isa' expr
bitwise_and_expr ::= expr ('&' expr)+
bitwise_or_xor_expr ::= expr ({'|'|'^'} expr)+
and_expr ::= expr ( '&&' expr)+
or_expr ::= expr ( {'||'|'//'} expr)+
flipflop_expr ::= expr ('..'|'...') expr
ternary_expr ::= expr '?' parse_scalar_expr ':' parse_scalar_expr { rightAssociative=true }

private assign_or_flow_expr ::=
   assign_expr
   | last_expr
   | next_expr
   | goto_expr
   | redo_expr

// fixme do we need to collapse tokens?
assign_expr ::= expr
({'**='|'+='|'-='|'*='|'/='|'%='|'.='|'x='|'&='|'|='|'^='|'<<='|'>>='|'&&='|'||='|'//='|'='} expr ) + {
rightAssociative=true }
comma_sequence_expr ::= expr {comma [parse_scalar_expr]}+ {pin(".*")=1}
lp_not_expr ::= 'not' expr { rightAssociative=true }
lp_and_expr ::= expr ('and' expr)+
lp_or_xor_expr ::= expr ({'or'|'xor'} expr)+

print_expr ::=  ('print'|'printf'|'say') ( print_parenthesized_call_arguments | [print_arguments] )
{pin=1}
print_parenthesized_call_arguments ::= print_parenthesized_call_arguments_body !('[')
{elementType=parenthesised_call_arguments}
private print_parenthesized_call_arguments_body ::= '(' [print_arguments_contents] ')'
{pin=1}
print_arguments ::= print_arguments_contents {elementType=call_arguments}
print_arguments_contents ::= [perl_handle] [print_arguments_contents_tail] {extends=expr
elementType=comma_sequence_expr}
private print_arguments_contents_tail ::= parse_scalar_expr {comma [parse_scalar_expr]}*
{pin(".*")=1}


sub_expr_simple ::= block !('->') {elementType=sub_expr}
sub_expr_simple_ensured ::= block {elementType=sub_expr}
sub_expr ::=  ['async'] 'sub' sub_definition_parameters block  // fixme make sure that this one
checked after definition and declaration

file_read_expr ::= LEFT_ANGLE [perl_handle_expr|scalar_variable] RIGHT_ANGLE
file_glob_expr ::= LEFT_ANGLE qq_string_content_with_lp RIGHT_ANGLE

////////////////////////// regular expressions ///////////////////////////////////////////////////////////////////////////
// pinning quotes leads to bug with replacement block
compile_regex ::=   'qr' match_regex_body {pin=1}
match_regex ::=   ['m'] match_regex_body
private match_regex_body ::= regex_match REGEX_QUOTE_CLOSE
[perl_regex_modifiers]

replacement_regex ::=
     's'
    regex_match
    regex_replace
    'r}'
    [perl_regex_modifiers] {pin=1}

private regex_match ::= 'r{' [perl_regex]
private regex_replace ::= regex_replace_regex | regex_replace_code

private regex_replace_regex ::= {'r/' | 'r}' 'r{' } regex_replacement {pin=1}
regex_replacement ::= regex_replacement_content
{recoverWhile=recover_regex_replacement}
private regex_replacement_content ::= [qq_string_content]
private regex_replace_code ::= {'re/' | 'r}' 're{' } [regex_code] {pin=1}
private regex_code ::= block_braceless {recoverWhile=recover_regex_replacement}
private recover_regex_replacement ::= !('r}')

perl_regex_modifiers ::= '/m' +
perl_regex ::= perl_regex_item * {hooks=[rightBinder="GREEDY_RIGHT_BINDER"
leftBinder="GREEDY_LEFT_BINDER"]}
private perl_regex_item ::=
           'regex' |
        block_compound |
           interpolated_constructs

tr_regex ::=  ('tr'|'y') tr_search tr_replacement [tr_modifiers] {pin=1}
private tr_search ::= 'r{' [tr_searchlist] {pin=1}
tr_searchlist ::= [tr_block_content]
private tr_block_content ::= {qq_string_element | '"-'}+
private tr_replacement ::= {'r/' | 'r}' 'r{'} [tr_replacementlist] 'r}' {pin=1}
tr_replacementlist ::= [tr_block_content]
tr_modifiers ::= '/m' +

////////////////////////// end of regular expressions //////////////////////////////////////////////////////////////////
```

```
do_block_expr ::=  'do' block
do_expr ::= 'do' expr {pin=1}

eval_expr ::=  'eval' [eval_argument] {pin=1}
private eval_argument ::= parenthesised_expr | block | expr

private variable_declaration ::=
    variable_declaration_global
    | variable_declaration_lexical
    | variable_declaration_local

// @todo attributes support
variable_declaration_local ::=  'local' local_variable_declaration_variation {pin=1}
variable_declaration_lexical ::=  ('my' | 'state') [any_package] variable_declaration_variation
[var_attributes] {pin=1}
variable_declaration_global ::=  'our' [any_package] variable_declaration_variation
[var_attributes] {pin=1}


private local_variable_declaration_variation ::= local_parenthesised_declaration |
local_variable_declaration_argument
private local_parenthesised_declaration ::= '(' local_variable_declaration_argument (comma
+ local_variable_declaration_argument ) * comma * ')' {pin=1}
private local_variable_declaration_argument ::= strict_variable_declaration_argument |
parse_scalar_expr

private variable_declaration_variation ::= variable_parenthesised_declaration |
variable_declaration_argument
private variable_parenthesised_declaration ::= '('
variable_parenthesised_declaration_contents ')' {pin=1}
private variable_parenthesised_declaration_contents ::=
strict_variable_declaration_argument (comma + strict_variable_declaration_argument ) *
comma*

private strict_variable_declaration_argument ::= strict_variable_declaration_wrapper |
undef_expr
private variable_declaration_argument ::= variable_declaration_element |  undef_expr

private strict_variable_declaration_wrapper ::= variable_declaration_element !('{' | '[' | '->' )
variable_declaration_element ::= '\\'? lexical_variable

////////////////////////////////// REFERENCES ////////////////////////////////////////////////////////////////////
anon_array ::= '[' [expr] ']' {pin=1 name="anonymous array"}
anon_hash ::= '{' [expr] '}' {pin=1 name="anonymous hash"}

//////////////////////////// END OF REFERENCES //////////////////////////////////////////////////////////////////////
// fixme it's not a variable, its variable expression
private variable ::= scalar_or_element | array_or_slice | hash_variable | hash_cast_expr |
glob_or_element

private array_or_slice ::= array_primitive [array_slice | hash_slice]
private array_primitive ::= array_variable | array_cast_expr | string_list
left array_slice ::= array_index
left hash_slice ::= hash_index

array_cast_expr ::= '$@' array_cast_target {name="array dereference"  extraRoot=true}
private array_cast_target ::= {block_array | scalar_primitive}
block_array ::= '@{' block_content '@}' {extends=block pin=1}

hash_cast_expr ::= '$%' hash_cast_target {name="hash dereference"  extraRoot=true}
private hash_cast_target ::= { block_hash  | scalar_primitive}
block_hash ::= '%{' block_content '%}' {extends=block pin=1}

private scalar_primitive ::= scalar_variable | scalar_cast_expr | undef_expr  // shouldn't it be
in term ? (check declarations)

// |'$}'|'@}'|'%}'|'*}'|'&}'
scalar_cast_expr ::= '$$' scalar_cast_target {name="scalar dereference" extraRoot=true}
scalar_index_cast_expr ::= '$#' scalar_cast_target {name="array last index dereference"
extraRoot=true}
private scalar_cast_target ::= {block_scalar | scalar_primitive}
block_scalar ::= '${' block_content '$}' {extends=block pin=1}

private scalar_or_element ::=  scalar_primitive [array_element | hash_element]

left array_element ::= array_index
left hash_element ::=  hash_index

private glob_or_element ::= glob_primitive [glob_slot]
private glob_primitive ::= glob_variable | glob_cast_expr
left glob_slot ::= hash_index

glob_cast_expr ::= '$*' glob_cast_target {name="typeglob dereference" extraRoot=true}
private glob_cast_target ::= {block_glob | scalar_primitive}
block_glob ::= '*{' block_content '*}' {extends=block pin=1}


private code_primitive ::= code_primitive_variation [primitive_call]
private code_primitive_variation ::= code_variable | code_cast_expr
left primitive_call ::= parenthesised_call_arguments {elementType=sub_call}
code_cast_expr ::= '$&' code_cast_target { extraRoot=true}
private code_cast_target ::= {block_code | scalar_primitive}
block_code ::= '&{' block_content '&}' {extends=block pin=1}
```

```
// extended nested element for using in ()
private nested_element_variation ::=
      hash_index
    | array_index
    | regular_nested_call
    | parenthesised_call_arguments
    | scalar_call
    | post_deref_expr
    | post_deref_glob_expr
    | post_deref_array_slice_expr
    | post_deref_hash_slice_expr

post_deref_expr ::= '->$*'|'->$#*'|'->@*'|'->%*'|'->**'|'->&*'     {name="Postderef"}
post_deref_glob_expr ::= '$*' hash_index
{name="Glob expr"}
post_deref_array_slice_expr ::= '$@' {hash_index|array_index}      {name="Array slice"}
post_deref_hash_slice_expr ::= '$%' {hash_index|array_index}       {name="Hash slice"}

hash_index ::= '{' hash_index_content '}' {pin=1 extraRoot=true}
private hash_index_content ::= expr {recoverWhile=recover_braced_expression}
private recover_braced_expression ::= !'}'
array_index ::= '[' array_index_content ']' {pin=1 extraRoot=true}
private array_index_content ::= expr {recoverWhile=recover_bracketed_expression}
private recover_bracketed_expression ::= !']'

private hash_or_slice ::= hash_primitive [hash_array_slice| hash_hash_slice]
private hash_primitive ::= hash_variable | hash_cast_expr
left hash_array_slice ::=  array_index
left hash_hash_slice ::=  hash_index

//////////////////////////////////////// CALLABLE ///////////////////////////////////////////////////////////////////////////k
call_arguments ::= parse_call_arguments
// fixme this should depend on prototype
private parse_call_arguments ::=
    &('+'|anon_hash_lookahead) parse_list_expr
    | arguments_list_with_codeblock
    | parse_list_expr
arguments_list_with_codeblock ::=
  sub_expr_simple [[comma] parse_scalar_expr {comma parse_scalar_expr}*] {extends=expr
elementType=comma_sequence_expr}

parenthesised_call_arguments ::= parenthesised_call_arguments_body !'[' {extraRoot=true}
unary_call_arguments ::= unary_expr {elementType=call_arguments}
private parenthesised_call_arguments_body ::= '(' optional_expression ')' {pin=1}
private optional_expression ::= [expr]
private any_unary_call_arguments ::= parenthesised_call_arguments |
unary_call_arguments
```

```
private any_call_arguments ::= parenthesised_call_arguments | [call_arguments]

scalar_call ::= scalar_or_element [parenthesised_call_arguments]

sub_call ::= parse_sub_call
regular_nested_call ::= regular_nested_call_variations {elementType=sub_call}
private regular_nested_call_variations ::= leftward_call | method

private parse_sub_call ::=
        leftward_call |
        named_unary_call |
        argumentless_call |
        rightward_call

private argumentless_call ::= argumentless_method
argumentless_method ::= 'argumentless' {elementType=method}

private named_unary_call ::= unary_method [unary_call_arguments]
unary_method ::= 'unary' | 'unary_custom' | '-t' {elementType=method}

type_specifier ::= parenthesised_expr | type_specifier_call
type_specifier_call ::= type_specifier_method  [!('$$'|'$@'|'$%')unary_call_arguments]
{elementType=sub_call}
type_specifier_method ::= type_specifier_tokens {elementType=method}
private type_specifier_tokens ::= 'unary_custom'|'subname'

private rightward_call ::= method  [call_arguments]
private leftward_call ::= {method|code_primitive_variation} parenthesised_call_arguments

private anon_hash_lookahead ::= '{' anon_hash_lookahead_body '}'
private anon_hash_lookahead_body ::=
  array_or_slice  |
  hash_or_slice  |
  // tried to optimize this, but technically not possible. E.g. if first line contains some comma
sequence, see heredocWrappingTest
  parse_scalar_expr {comma parse_scalar_expr} +

method ::= method_tokens
// the rest are fallback
private method_tokens ::= 'list' | 'unary' | 'unary_custom'| 'argumentless' |
  'package::name::' 'subname' |
  'subname' ['package::name'] |
  'method' | 'func' | 'default' | 'fun' |
  'finally' | 'try' | 'catch' |
  'switch' | 'case' |
  'fp_override' | 'fp_after' | 'fp_before' | 'fp_around' | 'fp_augment'
```

```
//////////////////////////////////END OF CALLABLE ///////////////////////////////////////////////////////////////////////

label_declaration ::= <<parseLabelDeclaration>> // custom faster method
label_expr ::= IDENTIFIER|'subname' !'('
private perl_version ::= <<parsePerlVersion>>
private perl_handle ::=  perl_handle_expr | block | scalar_variable
!('{'|'['|<<isOperatorToken>>)
perl_handle_expr ::= [QUALIFYING_PACKAGE] HANDLE

//////////////////////////////////// constants ///////////////////////////////////////////////////////////////////////

tag_scalar ::= TAG

number_constant ::= NUMBER | NUMBER_VERSION | NUMBER_HEX | NUMBER_OCT |
NUMBER_BIN

private string ::= string_bare | string_sq | string_dq | string_xq | heredoc_opener

string_dq ::= [ 'qq'] quoted_qq_string
private quoted_qq_string ::= QUOTE_DOUBLE_OPEN [qq_string_content_with_lp]
QUOTE_DOUBLE_CLOSE {pin=1}
private qq_string_content_with_lp ::= qq_string_content
private qq_string_content ::= qq_string_element+
private qq_string_element ::= STRING_CONTENT_QQ | special_constructs |
interpolated_constructs
private special_constructs ::= '\t'|'\n'|'\r'|'\f'|'\b'|'\a'|'\e'|'\l'|'\\u'| '\1' |
                            '\\"'|
                            '\L'|'\U'|'\F'|'\Q'|'\E'|
                            unicode_char | hex_char | oct_char | esc_char

esc_char ::= '\c*'

unicode_char ::= '\N' unicode_char_body {pin=1}
private unicode_char_body ::= '"{' unicode_char_body_content '}"' {pin=1}
private unicode_char_body_content ::= unicode_char_body_numbered |
unicode_char_name {recoverWhile=recover_string_braces}
private unicode_char_name ::= 'charname' {name="character name"}
private unicode_char_body_numbered ::= 'U+' char_code_hex {pin=1 name="character code
with U+ prefix" }
private char_code_hex ::= NUMBER_HEX {name="hex character code"}

private recover_string_braces ::= !('}"')

hex_char ::= '\x' hex_char_body {pin=1}
private hex_char_body ::=  hex_char_body_braced | [char_code_hex]
private hex_char_body_braced ::= '"{' [hex_char_body_content_in_brace] '}"' {pin=1
name="braced character code"}
```

```
private hex_char_body_content_in_brace ::= char_code_hex
{recoverWhile=recover_string_braces}

oct_char ::= oct_char_ambiguous | oct_char_unambiguous
private oct_char_ambiguous ::= '\0' [char_code_oct] {pin=1}
private oct_char_unambiguous ::= '\o' oct_char_body_braced {pin=1}
private oct_char_body_braced ::= '"{' oct_char_body_content_braced '}"' {pin=1}
private oct_char_body_content_braced ::= char_code_oct
{recoverWhile=recover_string_braces}
private char_code_oct ::= NUMBER_OCT {name="octal character code"}

string_xq ::= [ 'qx'] quoted_xq_string
private quoted_xq_string ::= QUOTE_TICK_OPEN [qx_string_content]
QUOTE_TICK_CLOSE {pin=1}
private qx_string_content ::= qx_string_element+
private qx_string_element ::= STRING_CONTENT_XQ | special_constructs
|interpolated_constructs

private interpolated_constructs ::= deref_expr

string_bare ::= &(STRING_CONTENT|'\\"')<<parseBareString>>

string_sq ::= [ 'q'] quoted_sq_string
string_list ::=  'qw' qw_string {pin=1 extraRoot=true}
private qw_string ::= QUOTE_SINGLE_OPEN [parse_qw_string_content]
QUOTE_SINGLE_CLOSE {pin=1}
private parse_qw_string_content ::= <<isUseVars>> <<mapUseVars qw_string_content>>+|
qw_string_content
private qw_string_content ::= string_bare+
private quoted_sq_string ::= QUOTE_SINGLE_OPEN [smart_sq_string_content]
QUOTE_SINGLE_CLOSE {pin=1}
private smart_sq_string_content ::= <<isUseVars>> <<mapUseVars
sq_string_content_element>>+ | sq_string_content
private sq_string_content ::= sq_string_content_element +
private sq_string_content_element ::= STRING_CONTENT | '\\"'

heredoc_opener ::= 'heredoc<<' ( '\\' string_bare | string ){pin=1}

///////////////////////////////////// variables ///////////////////////////////////////////////////////////////////////////
private lexical_variable ::= scalar_variable | array_variable | hash_variable

array_index_variable ::= '$#' {SCALAR_NAME | '${' SCALAR_NAME '$}'} {name="array last
index"}
scalar_variable ::= '$$' {SCALAR_NAME | '${' SCALAR_NAME '$}'} {name="scalar"}
array_variable ::= '$@' {ARRAY_NAME | '@{' ARRAY_NAME '@}'} {name="array"}
hash_variable ::= '$%' {HASH_NAME | '%{' HASH_NAME '%}'} {name="hash"}
code_variable ::= '$&' {method | '&{' method '&}'} {name="code"}
glob_variable ::= '$*' {GLOB_NAME | '*{' GLOB_NAME '*}'} {name="typeglob"}
```

```
private comma ::= ',' | '=>' {name="comma"}


/********************************************* Extensions for Method::Signatures
*************************************/
// we can make this smareter and use keywords from settings or import opitions; We can't
pin here because MooseX method works othewise
method_definition ::= {['async'] 'method'|'fp_override'} sub_names_token method_body
func_definition ::= {'func'|'fun'} sub_names_token func_body
fun_expr ::= 'fun' func_body
method_expr ::= 'fp_method' method_body
private method_body ::= [method_signature] func_or_method_body
private func_body ::= [func_signature] func_or_method_body
private func_or_method_body ::= [sub_attributes] block

private fp_modifier_named_body ::= sub_names_token method_signature
func_or_method_body

around_modifier ::= 'fp_around' sub_names_token around_signature func_or_method_body
{name="around modifier"}
private around_signature ::= <<parse_signature_content
parse_around_signature_content>>
private parse_around_signature_content ::= [around_signature_invocants]
[func_signature_elements]
around_signature_invocants ::= <<scalarDeclarationWrapper>> ','
<<scalarDeclarationWrapper>>  ':'

after_modifier ::= 'fp_after' fp_modifier_named_body {name="after modifier"}
augment_modifier ::= 'fp_augment' fp_modifier_named_body {name="augment modifier"} //
we should probably stub this one to navigate easier
before_modifier ::= 'fp_before' fp_modifier_named_body {name="before modifier"}

private meta parse_signature_content ::= '(' <<signature_content <<x1>>>> ')' {pin=1}
meta signature_content ::= <<x1>> {recoverWhile=recover_signature_content}
private recover_signature_content ::= !(')'|'{'|SUB_PROTOTYPE_TOKEN)
meta signature_element ::= <<x1>>

// not sure that we need a wrapper for signatures
private method_signature ::= <<parse_signature_content
parse_method_signature_content>>
private parse_method_signature_content ::= [method_signature_invocant]
[func_signature_elements]
method_signature_invocant ::= <<scalarDeclarationWrapper>> ':'
private func_signature ::= <<parse_signature_content parse_func_signature_content>>
private parse_func_signature_content ::= [func_signature_elements]
private func_signature_elements ::= func_signature_element (comma +
func_signature_element ) * comma*
```

```
private func_signature_element ::= <<signature_element parse_func_signature_element>>
private parse_func_signature_element ::= [type_specifier]':' ?
strict_variable_declaration_wrapper [parse_func_initializer] |  undef_expr |
sub_signature_element_ignore
private parse_func_initializer ::= '=' [parse_scalar_expr]

/************************************************ Extensions for Moose
*****************************************************/

/********************************************* Annotations
*****************************************************************/
private annotation ::=
          annotation_abstract
          | annotation_deprecated
          | annotation_method
          | annotation_override
          | annotation_returns
          | annotation_type
          | annotation_inject
          | annotation_no_inject
          | annotation_noinspection
          | '#@unknown' {name="perl annotation"}

annotation_abstract ::= '#@abstract' {pin=1}
annotation_deprecated ::= '#@deprecated' {pin=1}
annotation_method ::=  '#@method' {pin=1}
annotation_no_inject ::=  '#@noinject' {pin=1}
annotation_override ::=  '#@override' {pin=1}
annotation_returns ::=  '#@returns' annotation_type_param {pin=1}
private annotation_type_param ::=
  '*' |
  arrayref_type |
  hashref_type |
  any_package

arrayref_type ::= 'ArrayRef' '[' annotation_type_param ']' {pin=1}
hashref_type ::= 'HashRef' '[' annotation_type_param ']' {pin=1}

annotation_type ::=  '#@type' annotation_type_param {pin=1}
annotation_inject ::=  '#@inject' string_bare {pin=1}
annotation_noinspection ::=  '#@noinspection' string_bare {pin=1}
/********************************************* End of annotations
**********************************************************/

/********************************************* Lazy parsable elements
**********************************************************/
parsable_string_use_vars ::= use_vars_declarations {extraRoot=true}
comment_annotation ::= annotation {extraRoot=true}
```

```
heredoc ::= sq_string_content {extraRoot=true}
heredoc_qq ::= qq_string_content {extraRoot=true}
heredoc_qx ::= qx_string_content{extraRoot=true}
/************************************************ End of Lazy parsable elements
*************************************************************/

/****************************************************************  switch.pm
*************************************************************************/
switch_compound ::= 'switch' switch_condition block {pin=1}
switch_condition ::= '(' parse_scalar_expr ')'
private cases_sequence ::= case_compound + [case_default]
case_compound ::= 'case' case_condition block {pin=1}
case_condition ::= '(' parse_scalar_expr ')' | block | string | number_constant | anon_array |
match_regex | compile_regex
case_default ::= if_compound_else
/************************************************************* end of switch.pm
**********************************************************************/
```