Semantic Designs^{SM} Automated Tools for Software Engineering

Code Search (Find, Follow), Analysis (Metrics, Static, Dynamic) and Change (Modernization, Migration, Generation, Optimization, Rearchitecting)

- Home
- Services
  - Automated Migration
  - Custom Analysis and Transformation
  - Custom Development Toolkit
  - Application Modernization
  - Software Quality Analysis
  - Understanding Software Structure
- Products
  - DMS®
  - By Language
    - C
    - C++
    - Java
    - COBOL
    - C#/.Net
    - PHP
    - VHDL
    - Verilog
    - More...
  - By Tool
    - Search Engine
    - Clone Detection
    - Test Coverage
    - Formatters
    - Obfuscators
    - Metrics
    - Profilers
    - Smart Differencer
    - More...
  - By Application
    - Hogan (Banking) Analysis
    - More Effective Testing
    - Detecting Infringement
    - Agile Testing
  - Why Buy
  - Prices
  - Register
  - Downloads
- Company
  - About SD
  - Success Stories
  - News and Events
  - Partners
  - Customers
  - Careers
  - Papers
  - Visions
- Support
  - Support Policies
  - Register
  - Downloads
- Contact

DMS Attribute Grammars

The DMS Software Reengineering Toolkit is designed to allow the "domain" (language) engineer specify those languages quickly and accurately, so that she may spend most of her attention on the actual program analysis or transformation of interest.

At DMS Domains, we give some background on the necessary elements to build a DMS language processing domain. We have already described how to easily specify a grammar to DMS, automatically acquiring a parser and syntax tree builder. Once parsed, there is usually a need to analyze the syntax tree, often using the tree shape to directly guide the analysis. One could manually define a ad hoc, procedural recursive tree walk to compute such value. An attribute grammar is a more regular, organized way to define such analyzers; they are implemented largely by annotating the grammar rules for a language with computations over the tree nodes each grammar rule represents. (There are lot of other analyses one might do, but they usually depend on these analyses being done first, and as they get semantically more sophisticated, they become less and less driven by the shape of the AST, which models only syntax).

Attribute grammars can be used (with DMS) to compute:

- conventional metrics: line and variable count, maximum expression size, Halstead complexity, McCabe, etc.
- symbol tables, locating scope boundaries, and building a map of identifiers available and their types for each scope
- type determination and checks, propagating type information across expressions and verifying matches where required
- control flow graph extraction
- collection of set of side effects
- compute compositional semantics, such as denotational semantics

On this page, we show how attribute grammars are specified and executed by DMS. These attribute grammars are written in DMS's *Attribute Grammar* language, or *ATG*.

We use Nicholas Wirth's Oberon language as an example, as it is a real, practical language yet simple enough so the entire attribute grammar can be easily be displayed and understood here.

Attribute Grammar Concepts [1]

Attributes are additional values associated with something of central interest. In the case of ASTs, you can think of (AST) attributes as pairs *(attribute_name, attribute_value)* associated with each AST node, where the *attribute name* corresponds to some interesting fact-type, and the *attribute value* corresponds to the actual state of that fact (e.g., "**(constants_in_subtree_count,12)**").

The purpose of an *attribute grammar* is to specify how such attributes are to be computed over grammar rules (actually over the tree nodes that the grammar represent) in terms of other attributes in the tree or in terms of the tree nodes themselves. In practice, the attribute grammar rules compute attribute values for the subtree root and the leaves of (the shallow) subtree that the grammar rules represents. An *attribute grammar evaluator* then executes this specification, usually by visiting the tree nodes recursively and performing the tree-local specified computation, ending when all the attributes for all tree nodes have been computed.

Synthesized attributes are those whose value is computed from attribute values from children nodes, and are being passed up the tree. Often, values of synthesized attributes are combined to produce an attribute for the parent node. If an AST node has two children, each of which have their own attributes **(constants_in_subtree_count,5)** and **(constants_in_subtree_count,7)**, then by passing those attributes up, the parent can compute its corresponding (synthesized) attribute **(constants_in_subtree_count,12)**.

Inherited attributes are those passed from the parent down to the child. If the root of a function AST "knows" the function return type is **(return_type,integer)** as an attribute, it can pass that return type to the children of the function root, e.g. to the function body. Someplace deep down in that tree is an AST node for an actual return statement; if that node receives the inherited attribute **(return_type,*X*)**, it can check that the result it is computing is the correct type. Another commonly inherited attribute is a reference to the scope containing an AST, thus providing access to a symbol table for identifiers.

In practice, one wants to be able to define arbitrary sets of attributes for nodes, and pass them up and down the tree for the multiple purposes required to process ASTs (building symbol tables, constructing control flow graphs, doing type checking, computing metrics, ...). An *attribute grammar evaluator generator* is a code generator that will take grammar rules, sets of attribute definitions, and rules about how to compute synthesized and inherited attributes for the nodes involved in each rule, and generates both a parser and an AST walker that computes all the attributes.

DMS Attribute Grammar Specifications

Each attribute grammar is designed to compute a set of attributes that support some analysis purpose. Sometimes one need multiples analyses; sometimes, one needs multiple passes over the AST to conveniently compute something. DMS allows multiple attribute grammers for each single domain grammar; these grammars are, for historical reasons, individually called a *pass* ("over the tree").

Each Attribute Grammar pass is defined as a computation over a set of explicitly declared data types. One first defines the type names, and then defines the operators as signatures over those types. DMS attribute grammars also have a standad set of built-in types (tree nodes, integers, text strings, etc.) and corresponding operators (arithmetic, relational, etc.) All the data types and functions involved are realized in the underlying PARLANSE parallel programming langauge, and can thus be arbitrarily complex, including arbitrary PARLANSE compound data types. An attribute grammar type has a corresponding PARLANSE data type, and an attribute grammar operator has a corresponding PARLANSE function. Finally one defines a set of attribute names (*properties*) and the corresponding data type. The assumption is that the analyzer will compute the value of each property at every AST node.

```
TYPES
    constant_count integer;
    language_type Type;  --reference to PARLANSE union
                         --   encoding a target-language type description

ATTRIBUTES
    constant_count_in_subtree constant_count;
    expression_type Type;

FUNCTIONS
    constant_count CombineSubtreeConstantCounts(constant_count,constant_count);
    ComputeBinaryOperatorResultType(Node, Type, Type);
```

As a practical matter, it is acceptable that some properties may not be computed at some nodes. It is also practical to run an attribute grammar evaluation only on a subtree.

Given a DMS grammar rule:

```
  LHS_k = RHS_1 ... RHS_n ;
```

one writes an attribute grammar rule for a named attribute grammar computation for **Pass1** in the form:

```
  LHS_k = RHS_1 ... RHS_n ;
  <<Pass1>>:
      {  LHS.property_i=fn1(RHS_x.property_j,...);
           ...
         RHSa.property_b=fn2(RHS_y.property_q,...);
           ...
      }
```

The AGE computes the dataflows within each rule to determine a safe order in which to execute the functions to compute the set of properties visible at that node. (Circular dependencies are diagnosed and rejected). When a property value is needed from a child node, the AGE generates a recursive call to the attribute grammar rule for the child. In this way, control is passed around the tree in a maner that computes all the attributes properly. As property values are computed, they are stored either in variables local to the attribute evaluation process, if they are not needed after a node is processed, or in an external hash table associated with the node.

Defining a Semantics with an Attribute Grammar

With care, on can define a denotational semantics of a language computed by an attribute grammar. One of the key notions in denotational semantics is that of an *environment*, which maps identifiers to types and possibly symbolic values (this is essentially a "symbol table") At AST nodes that introduce new scopes, one would write an attribute function that created an new environment by combining the parent environment with newly introduced identifiers, and pass that down from the AST node to its children. One can then write denotational attribute grammar rules:

```
   exp = 'let' ID '=' exp1 'in' exp2;
   <<Denotation>>:
   { exp2.env = augment_environment(exp.env,
                                    new_variable_and_value_pair(name(ID.),
                                                                exp1.value));
     exp.value=exp2.value;
   }
```

This is mostly only for theoretical interest. For practical uses of DMS, it is not necessary to define a semantics. One need only code "correct" analysis and transformation steps.

Efficiency and Utility

The DMS Software Reengineering Toolkit provides an attribute grammar evaluator generator (AGE) providing efficient evalution.

The computations are compiled down to PARLANSE code and then compiled into native machine code. DMS's AGE heavily uses parallel attribute evaluation; it is obtained by computing a partial order over the attribute computations for a single grammar rule and compiling that in PARLANSE, which supports partial-order parallelism directly with low overhead. This emphasis on performance occurs because DMS is often applied to very systems of source code, having thousands of compilation units.

While one can demonstrate by thought experiment that attribute grammars are Turing-capable, it is more useful cto consider them in practice. DMS's attribute grammar system has been use on dozens of large projects, and computing semantic properties (including full name and type resolution) for all of C++14. While the DMS attribute grammar definition for C++14 is huge by most academic paper standards, it is that way because C++14 itself is huge and an astonishing mess ("camel by committee").

An Attribute Grammar for Oberon

This example shows the Oberon grammar decorated with a **<<Metrics>>** pass. The metrics pass computes class code metrics (Halstead, McCabe, loop nesting, ...) for each major structure ("Context", e.g., class or method) in the AST. (This is defined in rather general way, because this metrics framework is used internally by SD to implement metrics tools for many lanuages. Here we have instantiated it for Oberon).

One should note that the attribute grammar computations here are isolated from the prettypretter specification. This makes working on the entire (Oberon) domain more modular and easier to maintain.

The **ASSUME WRITTEN** declarations tell the AGE that certain properties have been initialized before the attribute evaluation starts; otherwise it might complain that it cannot find where a particular attribute has been initialized. Note the heavy use of long-distance **...** attributes; this allows one to compute attributes in one part of the grammar that are "far away" from other parts, but avoid the need to write trivial attribute value copies on intermediate grammar rules. Note the **>>** operator on **FinalizeContext** in attribute computations in the **Module** rules; this forces the following computation to occur after all other computations associated with the rule are complete.

```
--   <Oberon.atg>: Attribute grammar for the Oberon programming language.
--   Copyright (C) 2014-2015 Semantic Designs, Inc. All Rights Reserved.
--
--     Software Metrics Tool.
--
--     Update history:
--       Date       Initials   Changes Made
--   ------------------------------------------------------------
--     2014/12/15     KK         Created.
--

TYPES
{
    StringLiteral      "StringLiteralPool:StringLiteral";
    MetricsContext     "MetricsAttributeEvaluationContext";
    ContextName        "(reference MetricsContextNameBuilder)";
}

ATTRIBUTES
{
    MetricsContext   ctx;   -- Current context, like a namespace, class or method.
                            --    This attribute percolates down the tree.
    ContextName      name;  -- Name of the subtree if appropriate.
                            --    This attribute percolates up the tree.

    natural          conditonal_nesting; -- Current conditional Statement nesting of a node.
    natural          loop_nesting;       -- Current loop Statement nesting of a node.
}

CONSTANTS
{
    StringLiteral    VoidStringLiteral     "StringLiteralPool:VoidStringLiteral";
    ContextName      VoidContextName       "VoidMetricsContextNameBuilder";
}

FUNCTIONS
{
    MetricsContext   CreateContext(MetricsContext,   -- Parent context in the syntax tree.
                               text);            -- Name of the Type of the context,
                                                 -- like "Class", "Function", etc.
    void             FinalizeContext(MetricsContext, -- The context that is being finalized.
                               ContextName,      -- Name of the context instance,
                                                 -- like "CreateFile", "GetHttpHeaders", etc.
                                                 --       Can be NULL.
                               token);           -- Subtree that represents the context.
    ContextName      CreateContextName(MetricsContext, token);
    ContextName      CreateContextNameEx(MetricsContext, StringLiteral);
    ContextName      CreateContextNameGlobalScope(MetricsContext);
    ContextName      UpdateContextName(ContextName, token);
    ContextName      UpdateContextNameEx(ContextName, StringLiteral);
    ContextName      UpdateContextNameLastName(text, ContextName, text);

    void      ProcessCyclomaticComplexity(MetricsContext, natural);
    natural   ProcessConditionalStatement(MetricsContext, natural, token);
    natural   ProcessLoopStatement(MetricsContext, natural, token);

    void      ProcessHalsteadOperator(MetricsContext, text);
    void      ProcessHalsteadOperand(MetricsContext, token);
    void      ProcessHalsteadOperandWithText(MetricsContext, text);

    StringLiteral    Concatenate(MetricsContext, StringLiteral, StringLiteral) "ConcatenateStringLiterals";
    StringLiteral    ConcatenateEx(MetricsContext, text, StringLiteral, text) "ConcatenateStringLiteralsEx";
    StringLiteral    GetStringLiteral(MetricsContext, token) "GetStringLiteralFromLexemeToken";
    StringLiteral    RegisterString(MetricsContext, text) "AddStringToStringLiteralPool";
}

DIRECTIVES
{
    EvaluateOneSubtreeForAmbiguities in Metrics;
    EvaluateSequentially in Metrics;
}

-----------------------------------------------------------------------
-------------------- TERMINAL SYMBOLS ---------------------------------
-----------------------------------------------------------------------

 #IF Oberon
    charconst ;
        <<Metrics>>: { ProcessHalsteadOperand(charconst...ctx, charconst.); }
#ENDIF

string ;
    <<Metrics>>: { ProcessHalsteadOperand(string...ctx, string.); }
ident ;
    <<Metrics>>: { ProcessHalsteadOperand(ident...ctx, ident.); }
integer ;
    <<Metrics>>: { ProcessHalsteadOperand(integer...ctx, integer.); }
hexint ;
    <<Metrics>>: { ProcessHalsteadOperand(hexint...ctx, hexint.); }
real ;
    <<Metrics>>: { ProcessHalsteadOperand(real...ctx, real.); }

-------------------------------------------------------------------------
---(1)------------ Module
-------------------------------------------------------------------------

module = 'MODULE' ident ';'
            DeclarationSequence 'END' ident '.' ;
    <<Metrics>>:
    {
        ASSUME WRITTEN module.ctx;
        ASSUME WRITTEN module.conditonal_nesting;
        ASSUME WRITTEN module.loop_nesting;
    }
module = 'MODULE' ident ';'
            DeclarationSequence ProgramBody ident '.' ;
    <<Metrics>>:
    {
        ASSUME WRITTEN module.ctx;
        ASSUME WRITTEN module.conditonal_nesting;
        ASSUME WRITTEN module.loop_nesting;

        ProgramBody.ctx = CreateContext(module.ctx, "ProgramBody");
        >> FinalizeContext(ProgramBody.ctx, VoidContextName, ProgramBody.);
    }
module = 'MODULE' ident ';'
             'IMPORT' ImportList ';' DeclarationSequence 'END' ident '.' ;
    <<Metrics>>:
    {
        ASSUME WRITTEN module.ctx;
        ASSUME WRITTEN module.conditonal_nesting;
        ASSUME WRITTEN module.loop_nesting;
    }
module = 'MODULE' ident ';'
             'IMPORT' ImportList ';' DeclarationSequence ProgramBody ident '.' ;
    <<Metrics>>:
```

```
    {
        ASSUME WRITTEN module.ctx;
        ASSUME WRITTEN module.conditonal_nesting;
        ASSUME WRITTEN module.loop_nesting;

        ProgramBody.ctx = CreateContext(module.ctx, "ProgramBody");
        >> FinalizeContext(ProgramBody.ctx, VoidContextName, ProgramBody.);
    }

ImportList = import ;
    <<Metrics>>: { ProcessHalsteadOperator(ImportList...ctx, "Import_Import"); }
ImportList = ImportList ',' import ;
    <<Metrics>>: { ProcessHalsteadOperator(ImportList...ctx, "Import_Import"); }

import = ident ':=' ident ;
    <<Metrics>>: { ProcessHalsteadOperator(import...ctx, "Import_Assignment"); }

---------------------------------------------------------------------------------------
---(2)------------ Declarations
--------------------------------------------------------------------------------------

ConstantDeclarationList = ConstantDeclarationList IdentDef '=' ConstantExpression ';' ;
    <<Metrics>>: { ProcessHalsteadOperator(ConstantDeclarationList...ctx,
                                                 "Definition_Constant"); }

#IF Oberon
        TypeDeclarationList = TypeDeclarationList IdentDef '=' Type ';' ;
        <<Metrics>>: { ProcessHalsteadOperator(TypeDeclarationList...ctx,
                                                 "Definition_Type"); }
#ELSIF Oberon07
        TypeDeclarationList = TypeDeclarationList IdentDef '=' StructType ';' ;
        <<Metrics>>: { ProcessHalsteadOperator(TypeDeclarationList...ctx,
                                                 "Definition_StructType"); }
#ENDIF

ArrayType = 'ARRAY' LengthList 'OF' Type ;
    <<Metrics>>: { ProcessHalsteadOperator(ArrayType...ctx, "Definition_ArrayType"); }

LengthList = ConstantExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(LengthList...ctx, "Definition_ArrayDimension"); }
LengthList = LengthList ',' ConstantExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(LengthList...ctx, "Definition_ArrayDimension"); }

RecordType = 'RECORD' 'END' ;
    <<Metrics>>: { ProcessHalsteadOperator(RecordType...ctx, "Definition_RecordType"); }
RecordType = 'RECORD' FieldListSequence 'END' ;
    <<Metrics>>: { ProcessHalsteadOperator(RecordType...ctx, "Definition_RecordType"); }
RecordType = 'RECORD' '(' BaseType ')' 'END' ;
    <<Metrics>>: { ProcessHalsteadOperator(RecordType...ctx, "Definition_RecordWithBaseType"); }
RecordType = 'RECORD' '(' BaseType ')' FieldListSequence 'END' ;
    <<Metrics>>: { ProcessHalsteadOperator(RecordType...ctx, "Definition_RecordWithBaseType"); }

FieldNameList = IdentDef ;
    <<Metrics>>: { ProcessHalsteadOperator(FieldNameList...ctx, "Definition_RecordField"); }
FieldNameList = FieldNameList ',' IdentDef ;
    <<Metrics>>: { ProcessHalsteadOperator(FieldNameList...ctx, "Definition_RecordField"); }

PointerType = 'POINTER' 'TO' Type ;
    <<Metrics>>: { ProcessHalsteadOperator(PointerType...ctx, "Definition_PointerType"); }

ProcedureType = 'PROCEDURE' ;
    <<Metrics>>: { ProcessHalsteadOperator(ProcedureType...ctx, "Definition_ProcedureType"); }
ProcedureType = 'PROCEDURE' FormalParameters ;
    <<Metrics>>: { ProcessHalsteadOperator(ProcedureType...ctx, "Definition_ProcedureType"); }

VariableDefList = VariableDefList ',' IdentDef ;
    <<Metrics>>: { ProcessHalsteadOperator(VariableDefList...ctx, "Definition_Variable"); }

QualIdent = ident '.' ident ;
    <<Metrics>>: { ProcessHalsteadOperator(QualIdent...ctx, "Operator_QualIdent"); }

IdentDef = ident ;
    <<Metrics>>: { IdentDef.name = CreateContextName(IdentDef...ctx, ident.); }
IdentDef = ident '*' ;
    <<Metrics>>:
    {
        IdentDef.name = CreateContextNameEx(IdentDef...ctx,
                                            ConcatenateEx(IdentDef...ctx,
                                            "",
                                            GetStringLiteral(IdentDef...ctx, ident.), "*"));
        ProcessHalsteadOperator(IdentDef...ctx, "Identifier_StarSuffix");
    }
```

```
------------------------------------------------------------------------------------
---(3)------------ Procedures
------------------------------------------------------------------------------------

ProcedureDeclaration = ProcedureHeading ';' ProcedureBody ident ;
    <<Metrics>>:
    {
        ProcessHalsteadOperator(ProcedureDeclaration...ctx, "Definition_Procedure");
        ProcedureBody.ctx = CreateContext(ProcedureDeclaration...ctx, "Procedure");
        ProcedureHeading.ctx = ProcedureBody.ctx;
        >> FinalizeContext(ProcedureBody.ctx, ProcedureHeading.name, ProcedureDeclaration.);
    }

ProcedureHeading = 'PROCEDURE' IdentDef ;
    <<Metrics>>: { ProcedureHeading.name = IdentDef.name; }
ProcedureHeading = 'PROCEDURE' IdentDef FormalParameters ;
    <<Metrics>>: { ProcedureHeading.name = IdentDef.name; }

#IF Oberon
        ProcedureHeading = 'PROCEDURE' '*' IdentDef ;
        <<Metrics>>:
        {
            ProcedureHeading.name = UpdateContextNameLastName("*", IdentDef.name, "");
            ProcessHalsteadOperator(ProcedureHeading...ctx, "ProcName_StarPrefix");
        }
        ProcedureHeading = 'PROCEDURE' '*' IdentDef FormalParameters ;
        <<Metrics>>:
        {
            ProcedureHeading.name = UpdateContextNameLastName("*", IdentDef.name, "");
            ProcessHalsteadOperator(ProcedureHeading...ctx, "ProcName_StarPrefix");
        }
#ENDIF

FormalParameters = '(' ')' ':' QualIdent ;
    <<Metrics>>: { ProcessHalsteadOperator(FormalParameters...ctx,
                                                "Definition_ProcedureWithResultType"); }
FormalParameters = '(' FormalParameterList ')' ':' QualIdent ;
    <<Metrics>>: { ProcessHalsteadOperator(FormalParameters...ctx,
                                                "Definition_ProcedureWithResultType"); }

FormalParameterNameList = ident ;
    <<Metrics>>: { ProcessHalsteadOperator(FormalParameterNameList...ctx,
                                                "Definition_ProcedureParameter"); }
FormalParameterNameList = FormalParameterNameList ',' ident ;
    <<Metrics>>: { ProcessHalsteadOperator(FormalParameterNameList...ctx,
                                                "Definition_ProcedureParameter"); }

ArrayOfSequence = ArrayOfSequence 'ARRAY' 'OF' ;
    <<Metrics>>: { ProcessHalsteadOperator(ArrayOfSequence...ctx,
                                                "Definition_ProcedureArrayOfParameter"); }

#IF Oberon
        ForwardDeclaration = 'PROCEDURE' '^' ident ;
        <<Metrics>>: { ProcessHalsteadOperator(ForwardDeclaration...ctx,
                                                    "Definition_ForwardDeclaration"); }
        ForwardDeclaration = 'PROCEDURE' '^' ident FormalParameters ;
        <<Metrics>>: { ProcessHalsteadOperator(ForwardDeclaration...ctx,
                                                    "Definition_ForwardDeclaration"); }
        ForwardDeclaration = 'PROCEDURE' '^' ident '*' ;
        <<Metrics>>: { ProcessHalsteadOperator(ForwardDeclaration...ctx,
                                                    "Definition_ForwardDeclaration*"); }
        ForwardDeclaration = 'PROCEDURE' '^' ident '*' FormalParameters ;
        <<Metrics>>: { ProcessHalsteadOperator(ForwardDeclaration...ctx,
                                                    "Definition_ForwardDeclaration*"); }
#ELSIF Oberon07
        ProcedureBody = DeclarationSequence 'RETURN' Expression 'END' ;
        <<Metrics>>: { ProcessHalsteadOperator(ProcedureBody...ctx,
                                                    "Definition_ProcedureReturnStatement"); }
        ProcedureBody = DeclarationSequence
                              'BEGIN' StatementSequence 'RETURN' Expression 'END' ;
        <<Metrics>>: { ProcessHalsteadOperator(ProcedureBody...ctx,
                                                    "Definition_ProcedureReturnStatement"); }
#ENDIF

------------------------------------------------------------------------------------
---(4)------------ Statements
------------------------------------------------------------------------------------

#IF Oberon
        Statement = 'EXIT' ;
        <<Metrics>>:
```

```
        {
            ProcessCyclomaticComplexity(Statement...ctx, 1);
            ProcessHalsteadOperator(Statement...ctx, "Statement_Exit");
        }
        Statement = 'RETURN' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(Statement...ctx, 1);
            ProcessHalsteadOperator(Statement...ctx, "Statement_Return");
        }
        Statement = 'RETURN' Expression ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(Statement...ctx, 1);
            ProcessHalsteadOperator(Statement...ctx, "Statement_ReturnWithExpression");
        }
#ENDIF

Assignment = designator ':=' Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(Assignment...ctx, "Statement_Assignment"); }

IfStatement = 'IF' Expression 'THEN' StatementSequence 'END' ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(IfStatement...ctx, 1);
        StatementSequence.conditonal_nesting =
              ProcessConditionalStatement(IfStatement...ctx,
                                          IfStatement...conditonal_nesting,
                                          IfStatement.);
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_If");
    }
IfStatement = 'IF' Expression 'THEN' StatementSequence 'ELSE' StatementSequence 'END' ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(IfStatement...ctx, 1);
        StatementSequence[1].conditonal_nesting =
             ProcessConditionalStatement(IfStatement...ctx,
                                         IfStatement...conditonal_nesting,
                                         IfStatement.);
        StatementSequence[2].conditonal_nesting =
             ProcessConditionalStatement(IfStatement...ctx,
                                         IfStatement...conditonal_nesting,
                                         IfStatement.);
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_If");
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_Else");
    }
IfStatement = 'IF' Expression 'THEN' StatementSequence ElseIfSequence 'END' ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(IfStatement...ctx, 1);
        StatementSequence.conditonal_nesting =
             ProcessConditionalStatement(IfStatement...ctx,
                                         IfStatement...conditonal_nesting,
                                         IfStatement.);
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_If");
    }
IfStatement = 'IF' Expression 'THEN' StatementSequence ElseIfSequence
                  'ELSE' StatementSequence 'END' ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(IfStatement...ctx, 1);
        StatementSequence[1].conditonal_nesting =
             ProcessConditionalStatement(IfStatement...ctx,
                                         IfStatement...conditonal_nesting,
                                         IfStatement.);
        StatementSequence[2].conditonal_nesting =
             ProcessConditionalStatement(IfStatement...ctx,
                                         IfStatement...conditonal_nesting,
                                         IfStatement.);
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_If");
        ProcessHalsteadOperator(IfStatement...ctx, "Statement_Else");
    }

ElseIfSequence = 'ELSIF' Expression 'THEN' StatementSequence ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(ElseIfSequence...ctx, 1);
        StatementSequence.conditonal_nesting =
             ProcessConditionalStatement(ElseIfSequence...ctx,
                                         ElseIfSequence...conditonal_nesting,
                                         ElseIfSequence.);
        ProcessHalsteadOperator(ElseIfSequence...ctx, "Statement_ElseIf");
    }
ElseIfSequence = ElseIfSequence 'ELSIF' Expression 'THEN' StatementSequence ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(ElseIfSequence...ctx, 1);
        StatementSequence.conditonal_nesting =
             ProcessConditionalStatement(ElseIfSequence...ctx,
                                         ElseIfSequence...conditonal_nesting,
                                         ElseIfSequence.);
        ProcessHalsteadOperator(ElseIfSequence...ctx, "Statement_ElseIf");
    }

CaseStatement = 'CASE' Expression 'OF' CaseSequence 'END' ;
    <<Metrics>>:
    {
        ProcessHalsteadOperator(CaseStatement...ctx, "Statement_Case");
    }

#IF Oberon
        CaseStatement = 'CASE' Expression 'OF' CaseSequence 'ELSE' StatementSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(CaseStatement...ctx, 1);
            ProcessHalsteadOperator(CaseStatement...ctx, "Statement_CaseStatement");
        }
#ENDIF

CaseStatementCase = CaseLabelList ':' StatementSequence ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(CaseStatementCase...ctx, 1);
            ProcessHalsteadOperator(CaseStatementCase...ctx, "Statement_CaseStatementCase");
        }

#IF Oberon
        label = ConstantExpression ;
        <<Metrics>>: { ProcessHalsteadOperator(label...ctx, "Statement_CaseLabelConstExpr"); }
#ELSIF Oberon07
        label = integer ;
        <<Metrics>>: { ProcessHalsteadOperator(label...ctx, "Statement_CaseLabelInteger"); }
    label = hexint ;
        <<Metrics>>: { ProcessHalsteadOperator(label...ctx, "Statement_CaseLabelHexInt"); }
        label = string ;
        <<Metrics>>: { ProcessHalsteadOperator(label...ctx, "Statement_CaseLabelString"); }
        label = ident ;
        <<Metrics>>: { ProcessHalsteadOperator(label...ctx, "Statement_CaseLabelIdentifier"); }
#ENDIF

WhileStatement = 'WHILE' Expression 'DO' StatementSequence 'END' ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(WhileStatement...ctx, 1);
        StatementSequence.loop_nesting =
             ProcessLoopStatement(WhileStatement...ctx,
                                  WhileStatement...loop_nesting,
                                  WhileStatement.);
        ProcessHalsteadOperator(WhileStatement...ctx, "Statement_While");
    }

#IF Oberon07
        WhileStatement = 'WHILE' Expression 'DO' StatementSequence ElseIfDoSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(WhileStatement...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(WhileStatement...ctx,
                                      WhileStatement...loop_nesting,
                                      WhileStatement.);
            ProcessHalsteadOperator(WhileStatement...ctx, "Statement_While");
        }

        ElseIfWhileSequence = 'ELSIF' Expression 'DO' StatementSequence ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(ElseIfWhileSequence...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(ElseIfWhileSequence...ctx,
                                      ElseIfWhileSequence...loop_nesting,
                                      ElseIfDoSequence.);
            ProcessHalsteadOperator(ElseIfWhileSequence...ctx, "Statement_WhileElseIfDo");
        }
        ElseIfWhileSequence = ElseIfDoSequence 'ELSIF' Expression 'DO' StatementSequence ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(ElseIfWhileSequence...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(ElseIfWhileSequence...ctx,
                                      ElseIfWhileSequence...loop_nesting,
                                      ElseIfDoSequence.);
            ProcessHalsteadOperator(ElseIfWhileSequence...ctx, "Statement_WhileElseIfDo");
        }
#ENDIF

RepeatStatement = 'REPEAT' StatementSequence 'UNTIL' Expression ;
    <<Metrics>>:
    {
        ProcessCyclomaticComplexity(RepeatStatement...ctx, 1);
        StatementSequence.loop_nesting =
             ProcessLoopStatement(RepeatStatement...ctx,
                                  RepeatStatement...loop_nesting,
                                  RepeatStatement.);
        ProcessHalsteadOperator(RepeatStatement...ctx, "Statement_Repeat");
    }

#IF Oberon
        LoopStatement = 'LOOP' StatementSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(LoopStatement...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(LoopStatement...ctx,
                                      LoopStatement...loop_nesting,
                                      LoopStatement.);
            ProcessHalsteadOperator(LoopStatement...ctx, "Statement_Loop");
        }

        WithStatement = 'WITH' QualIdent ':' QualIdent
                          'DO' StatementSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(WithStatement...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(WithStatement...ctx,
                                      WithStatement...loop_nesting,
                                      WithStatement.);
            ProcessHalsteadOperator(WithStatement...ctx, "Statement_With");
        }
#ELSIF Oberon07
        ForStatement = 'FOR' ident ':=' Expression 'TO' Expression
                          'DO' StatementSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(ForStatement...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(ForStatement...ctx,
                                      ForStatement...loop_nesting,
                                      ForStatement.);
            ProcessHalsteadOperator(ForStatement...ctx, "Statement_For");
        }
        ForStatement = 'FOR' ident ':=' Expression 'TO' Expression 'BY' ConstantExpression
                          'DO' StatementSequence 'END' ;
        <<Metrics>>:
        {
            ProcessCyclomaticComplexity(ForStatement...ctx, 1);
            StatementSequence.loop_nesting =
                 ProcessLoopStatement(ForStatement...ctx,
                                      ForStatement...loop_nesting,
                                      ForStatement.);
            ProcessHalsteadOperator(ForStatement...ctx, "Statement_For");
        }
#ENDIF

--------------------------------------------------------------------------------
---(5)------------ Expressions
--------------------------------------------------------------------------------

Expression = Expression '=' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_EQ"); }
Expression = Expression '#' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_NE"); }
Expression = Expression '<' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_LT"); }
Expression = Expression '<=' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_LE"); }
```

```
Expression = Expression '>' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_GT"); }
Expression = Expression '>=' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_GE"); }
Expression = Expression 'IN' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_IN"); }
Expression = Expression 'IS' SimpleExpression ;
    <<Metrics>>: { ProcessHalsteadOperator(Expression...ctx, "Operator_IS"); }

SimpleExpression = '+' term ;
    <<Metrics>>: { ProcessHalsteadOperator(SimpleExpression...ctx, "Operator_Unary+"); }
SimpleExpression = '-' term ;
    <<Metrics>>: { ProcessHalsteadOperator(SimpleExpression...ctx, "Operator_Unary-"); }
SimpleExpression = SimpleExpression '+' term ;
    <<Metrics>>: { ProcessHalsteadOperator(SimpleExpression...ctx, "Operator_+"); }
SimpleExpression = SimpleExpression '-' term ;
    <<Metrics>>: { ProcessHalsteadOperator(SimpleExpression...ctx, "Operator_-"); }
SimpleExpression = SimpleExpression 'OR' term ;
    <<Metrics>>: { ProcessHalsteadOperator(SimpleExpression...ctx, "Operator_OR"); }

term = term '*' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(term...ctx, "Operator_*"); }
term = term '/' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(term...ctx, "Operator_/"); }
term = term 'MOD' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(term...ctx, "Operator_MOD"); }
term = term 'DIV' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(term...ctx, "Operator_DIV"); }
term = term '&' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(term...ctx, "Operator_&"); }

factor = 'NIL' ;
    <<Metrics>>: { ProcessHalsteadOperandWithText(factor...ctx, "NIL"); }

#IF Oberon07
    factor = 'TRUE' ;
        <<Metrics>>: { ProcessHalsteadOperandWithText(factor...ctx, "TRUE"); }
    factor = 'FALSE' ;
        <<Metrics>>: { ProcessHalsteadOperandWithText(factor...ctx, "FALSE"); }
#ENDIF

factor = '~' factor ;
    <<Metrics>>: { ProcessHalsteadOperator(factor...ctx, "Operator_~"); }

Set = '{' '}' ;
    <<Metrics>>: { ProcessHalsteadOperator(Set...ctx, "Operator_Set"); }
Set = '{' SetElementList '}' ;
    <<Metrics>>: { ProcessHalsteadOperator(Set...ctx, "Operator_Set"); }

SetElement =  Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(SetElement...ctx, "Operator_SetElement"); }
SetElement = Expression '..' Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(SetElement...ctx, "Operator_SetElementRange"); }

ProcedureCall = designator ;
    <<Metrics>>: { ProcessHalsteadOperator(ProcedureCall...ctx, "Operator_ProcedureCall"); }
ProcedureCall = designator '(' ActualParameterList ')' ;
    <<Metrics>>: { ProcessHalsteadOperator(ProcedureCall...ctx, "Operator_ProcedureCall"); }

ActualParameterList = Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(ActualParameterList...ctx,
                                           "Operator_ActualParameter"); }
ActualParameterList = ActualParameterList ',' Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(ActualParameterList...ctx,
                                           "Operator_ActualParameter"); }

designator = designator '.' ident ;
    <<Metrics>>: { ProcessHalsteadOperator(designator...ctx, "Operator_RecordField"); }
designator = designator '(' QualIdent ')' ;
    <<Metrics>>: { ProcessHalsteadOperator(designator...ctx, "Operator_(QualIdent)"); }
designator = designator '^' ;
    <<Metrics>>: { ProcessHalsteadOperator(designator...ctx, "Operator_Dereference"); }

ArrayIndexList = Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(ArrayIndexList...ctx, "Operator_ArrayIndex"); }
ArrayIndexList = ArrayIndexList ',' Expression ;
    <<Metrics>>: { ProcessHalsteadOperator(ArrayIndexList...ctx, "Operator_ArrayIndex"); }
```

Next: DMS Rewrite Rules

[1] Footnote: text in this section is an enhanced version of that originally published by Semantic Designs/Ira Baxter at StackOverflow. SD claims copyright, and thus the right to use it here with arbitrary changes of our choice.

Search SD

Topics

- Re-engineering
- Documentation
- Assessment
- Improvement
- Code Generation
- Hardware Description Languages
- All Topics

Language:
PL/SQL
Product:
Custom Transformation Go!

Semantic Designs- Our Goal

To enable our customers to produce and maintain timely, robust and economical software by providing world-class Software Engineering tools using deep language and problem knowledge with high degrees of automation.
For more information: info@semanticdesigns.com Follow us at Twitter: @SemanticDesigns

Comments or problems: webmaster@semanticdesigns.com
DMS
Attribute Grammars